JANUARY 10, 2024 / #DOM

The JavaScript DOM Manipulation Handbook

Benjamin Semah

learn about in JavaScript. This is because one of JavaScript's main uses is to make web pages interactive – and the Document Object Model (DOM) plays a major role in this.

The DOM is a powerful tool that allows you to interact with and manipulate the elements on a web page. And this handbook will help you understanding and become confident in working with it.

You will begin by learning what the DOM is and what you can do with it. Then we'll dive into how to select, modyfy, and style DOM elements. You will also learn how to create new elements and add them to your web page.

The handbook also cover topics like how to traverse the DOM what DOM events are, and includes some project ideas for practice.

Let's get started!

Table of Contents

- How to Change the Content of DOM Elements
  - The innerHTML Property
  - Security Risks with innerHTML
  - The innerText and textContent Properties
- How to Work with Attributes of DOM Elements
  - The getAttribute Method
  - The setAttribute Method
  - The removeAttribute Method
  - The hasAttribute Method
- How to Change the Styles on DOM Elements
  - Setting Styles with the .style Property
  - Setting Styles with Classes
- How to Traverse the DOM
  - Difference Between a Node and an Element
  - Selecting a Parent with parentNode vs parentElement
  - Selecting Elements with childNodes vs children
  - Selecting the First or Last Child/Element
  - Selecting a Sibling of Nodes in the DOM
- DOM Events and Event Listeners
  - Difference Between Event Listener and Event Handler
  - Three Ways to Register Events in JavaScript

What is the DOM?

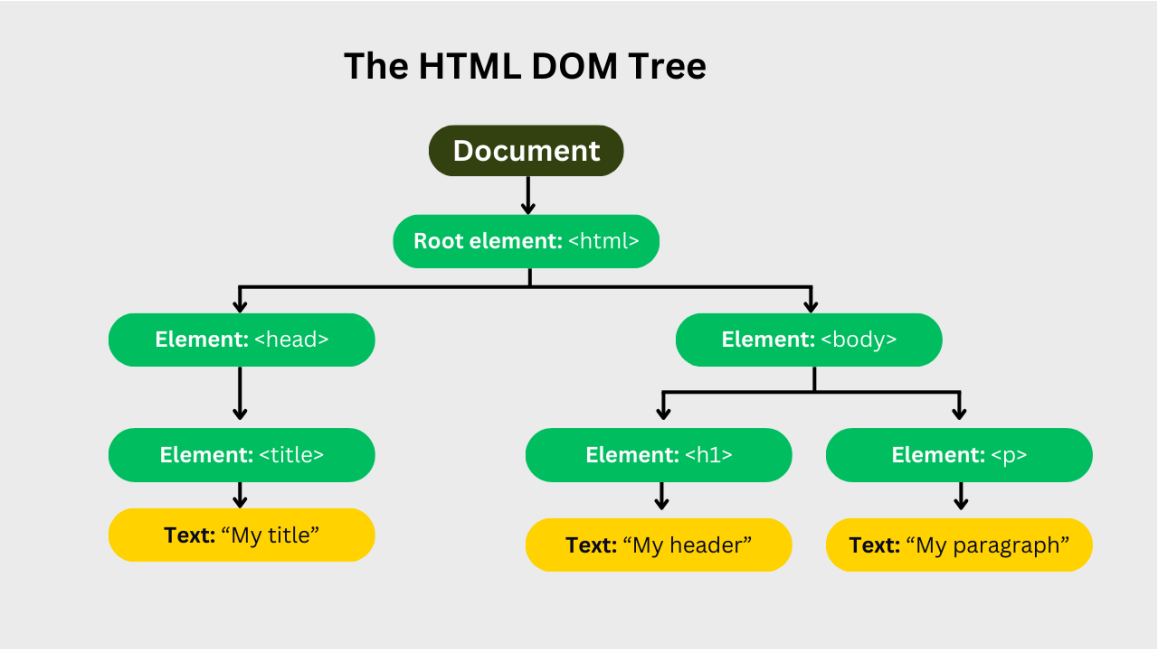
DOM stands for Document Object Model. But what does that mean? Let's break it down.

The **Document** part refers to the webpage you see in the browser. Specifically, the HTML Document which handles the structure of the page's content. This includes the text, images, links, and other elements that make up the page.

Object means the elements like images, headers, and paragraphs are treated like objects. Each object has its properties (like id, class, style) and methods. Using these properties and methods, you can manipulate the elements.

The **Model** in DOM means it's a representation or copy of the HTML document as a hierarchical tree. This tree includes all the elements.

that they are in sync. So if something changes in the HTML, the DOM changes too, and vice versa.



A graphical representation of the HTML DOM tree

At the top of the hierarchy is the Document object. It has only one child – the `html` element. The `html` element, also known as the root element, has two children, the `head` and `body` elements. And each of them also have their own children elements.

The parent-child relationship between the elements is what allows you to traverse or move between and select them. More on that later.

What You Can Do With the DOM

you can do with the DOM:

- Change and remove existing elements in the DOM.
- Create and add new elements to the page.
- Change the styles for elements.
- Add event listeners to the elements to make them interactive.

How to Select DOM Elements

To do something with DOM elements, you first have to select or access the element in question. In this section, you will learn some common methods for selecting DOM elements.

Let's use the following phonebook markup to show how the various DOM selector methods work.

```
<h1 id="page-title">Phonebook</h1>

<p class="family">Marie</p>
<p class="family">Jose</p>
<p class="work">Anne</p>
<p class="work">Joan</p>
```

The markup includes a header with an id of `page-title` and four paragraphs. The first two paragraphs both have a class of `family`, and the last two have a class of `work`.

1. getElementById

have an id with the same value.

This means anytime you use the `getElementById` method, you are going to select only one element from the DOM.

Let's look at an example:

The `h1` header has an id value of `page-title`. Here is how you can select it using the `getElementById` method:

```
const titleElement = document.getElementById("page-title")
console.log(titleElement)
```

The example selects the header element and assigns it to the `titleElement` variable.

```
<h1 id="page-title">Phonebook</h1>
```

Result of accessing element with getElementById method.

If there's no element in the DOM with the given id, the `getElementById()` method will return `null`.

takes in the value of a class attribute as an argument and selects all elements in the DOM that has the given class. Unlike ids, you can give the same class value for different HTML elements.

Here's an example:

```
const famContacts = document.getElementsByClassName("family")
console.log(famContacts)
```

This returns an HTML collection of all elements with the given class. The log statement will print the following in the console:

```
▼ HTMLCollection(2) [p.family, p.family] i
  ▶ 0: p.family
  ▶ 1: p.family
    length: 2
  ▶ [[Prototype]]: HTMLCollection
```

The `getElementsByClassName()` method returns an HTML collection.

Note: The HTML collection looks like an array, but it is not. You can access the elements using bracket notation just as you would with an array – but you cannot apply array methods like `map`, `filter`, and `forEach` on it.

This will get the first element in the HTML collection, which is the paragraph with the name Marie.

```
<p class="family">Marie</p>
```

Result of accessing HTMLCollection element with bracket notation.

But what if you wanted to loop through all the elements in the `famContacts` HTML collection? You'd first need to convert the HTML collection into an array. Then you could use any of the array methods.

A simple way to create an array from the HTML collection is to use the spread operator, like this:

```
let famContactsArray = [...famContacts]

famContactsArray.forEach(element => console.log(element))
```

```
<p class="family">Marie</p>
<p class="family">Jose</p>
```

Using the `forEach` method, you can access each of the items in the `famContactsArray`. The browser will throw an error if you try to apply an array method like `map` to the HTML collection without first creating an array from it.

Error message when you use array methods on an HTMLCollection.

3. getElementsByTagName

This method will select elements using their tag name. For example, `getElementByTagName('p')` will select all paragraphs on the page.

Like `getElementsByClassName`, this method also returns an HTML collection of the selected elements.

Here's an example:

```
const allContacts = document.getElementsByTagName('p')
console.log(allContacts)
```

```
▶ 1: p.family
▶ 2: p.work
▶ 3: p.work
  length: 4
▶ [[Prototype]]: HTMLCollection
```

An HTMLCollection containing all paragraph elements.

You can create an array from the HTML collection and use any of the array methods on it.

```
let allContactsArray = [...allContacts]

allContactsArray.map(element => console.log(element))
```

```
<p class="family">Marie</p>
<p class="family">Jose</p>
<p class="work">Anne</p>
<p class="work">Joan</p>
```

Result of using the map method on `allContactsArray`.

4. querySelector

The `querySelector` method works like how CSS selectors work.

For example, what do you do when you want to select an element with an id? You use the hash `#` symbol. How about when you want to select elements with a class? You put a dot `.` in front of the class name.

Here's an example:

```
const firstWorkContact = document.querySelector('.work')
console.log(firstWorkContact)
```

```
<p class="work">Anne</p>
```

An example of using the `querySelector` method.

The example above returns only the first element with a class of `work` and ignores the rest.

Let's see another example to show how querySelector works like CSS selectors. The following is a `div` element with four buttons:

```
    <button>Third button</button>
    <button>Fourth button</button>
</div>
```

Assuming you wanted to select the third button, you could use `querySelector` like the one below. The code uses the CSS `nth-child` selector to get the third button inside the div.

```
const thirdBtn = document.querySelector('div button:nth-child(3)')
console.log(thirdBtn)
```

```
<button>Third button</button>
```

Result of selecting the third button with `querySelector` method.

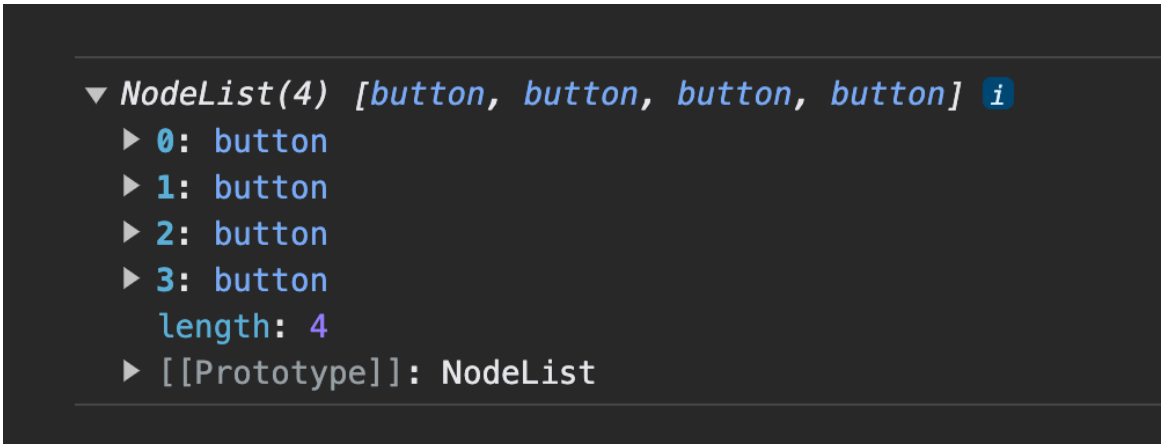
But what if you want to select all four button elements and not only the first one? Then you could use the `querySelectorAll` method instead.

5. querySelectorAll

Like the `querySelector` method, `querySelectorAll` also selects HTML elements using CSS selectors. The difference is that it returns

Using the previous example, let's select all the buttons with `querySelectorAll`.

```
const allBtns = document.querySelectorAll('button')
console.log(allBtns)
```



The `querySelectorAll` method returns a NodeList of selected elements.

Note: `querySelectorAll` returns a `NodeList`. A `NodeList` is slightly different from an HTML collection. You don't need to convert it to an array to apply a method like `forEach` on it.

```
allBtns.forEach(btn => console.log(btn))
```

```
<button>Second button</button>
<button>Third button</button>
<button>Fourth button</button>
```

Result of using `forEach` method on the NodeList.

But you still cannot apply array methods like `map`, `filter`, and others on a NodeList. You will need to first create an array from it.

You can read this freeCodeCamp article on the difference between HTML collection and NodeList to learn more.

How to Change the Content of DOM Elements

So far you've learned about different ways to select DOM elements. But that is only the beginning. Now, let's see how you can manipulate the DOM to change the content of a webpage.

The first thing you need to do is to select the element. You can do that using any of the methods you learned from the previous section.

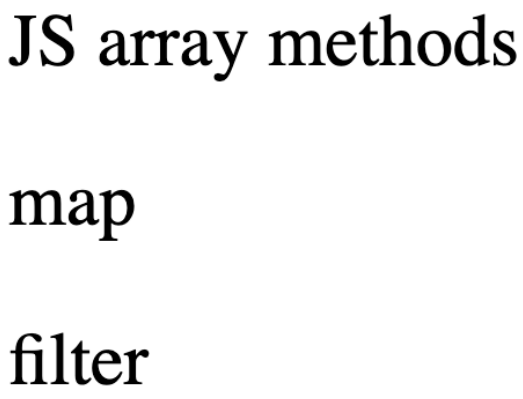
After selecting the element, there are several methods you can use to add or update the content.

The `innerHTML` Property

You can use the `innerHTML` method.

The following is some markup of three paragraphs, each with an id.

```
<p id="topic">JS array methods</p>
<p id="first-method">map</p>
<p id="second-method">filter</p>
```



Simple markup with three paragraph elements

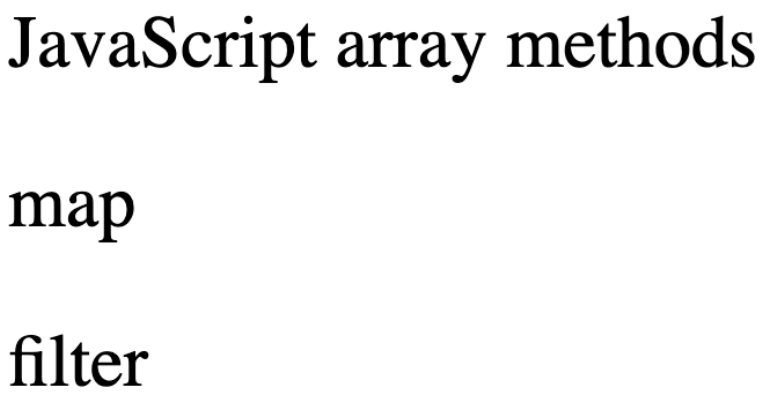
You can read or get the content of the any of the paragraphs using `innerHTML`. For example, let's get the content of the first paragraph.

```
const topicElement = document.querySelector('#topic')
console.log(topicElement.innerHTML)
```

Log statement of the `innerHTML` of the `topicElement`

Now, let's say you want to update the topic from "JS array methods" to "JavaScript array methods". You can do that by assigning the new text to the `innerHTML` of the element.

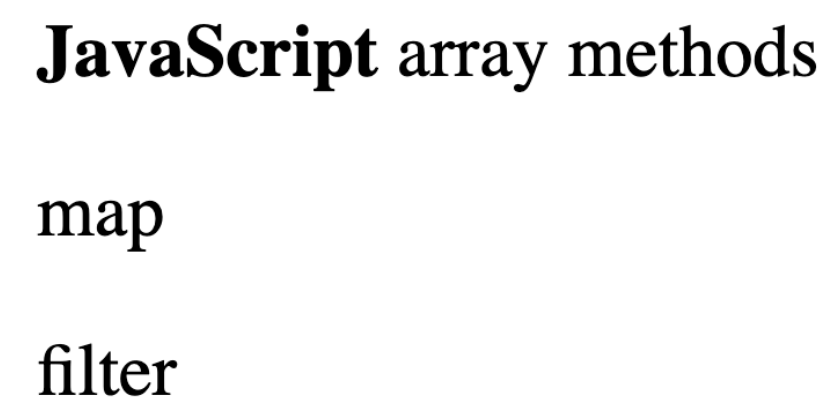
```
const topicElement = document.querySelector('#topic')
topicElement.innerHTML = "JavaScript array methods"
```



The topic is updated from "JS Array methods" to "JavaScript array methods"

Want to make the word "JavaScript" bold, you could do this:

```
topicElement.innerHTML = "<b>JavaScript</b> array methods"
```



The word "JavaScript" is made bold using innerHTML

Security Risks With `innerHTML`

Using the `innerHTML` poses potential security risks. An example is cross-site scripting (XSS) attacks.

If the content you're inserting comes from user input or any untrusted source, be sure to validate and sanitize it before using `innerHTML` to prevent XSS attacks. You can use a library like DOMPurify to do this.

Also, if you are working with plain text content, consider using methods like `innerText` and `textContent`.

Both `innerText` and `textContent` ignore HTML tags and treat them as part of a string. You can use both methods to read or update the text of DOM elements.

A key difference between the two is in how they treat the text. Using `innerText` returns the text as it appears on the screen. And using `textContent` returns text as it appears in the markup. Let's see an example below.

```
<p>Key =<span style="display: none;">     ABC123<span></p>
```

Key =

A paragraph element with the some text and a hidden span element inside

The example includes a paragraph element. Inside the paragraph is a span that contains a key. The key does not appear on screen because its inline style is set to a display of none.

Now, let's select the paragraph and print both the `innerText` value and `textContent` value to see the difference.

```
console.log("textContent: ", paragraph.textContent)
```

```
innerText: Key =
textContent: Key =     ABC123
```

Result of log statement for `innerText` and `textContent`.

Note how `innerText` returns the text as it appears on the screen (without the value of the key which is hidden with CSS). And note how `textContent` returns the text including hidden elements and whitespaces.

Let's see another example for adding text to an element. The following code includes two paragraphs, each with bold text and an empty span, as well as a horizontal line between them.

```
<p>
  <b>innerText Example</b>
  <span id="inner-text"></span>
</p>

<hr>

<p>
  <b>textContent Example</b>
```

innerText Example

---

textContent Example

Example to demo the `innerText` and `textContent` properties

Now, let's select the two span elements and add the same text to them. This will help you better understand the difference between `innerText` and `textContent`.

```
const example1 = document.querySelector('#inner-text');
const example2 = document.querySelector('#text-content');

let address = `
  ABC Street,
  Spintex Road,
  Accra,
  Ghana.
`;

example1.innerText = address;
example2.textContent = address;
```

The code gives the same variable `address` to the two examples. The first uses `innerText` and the second uses `textContent`. See the

innerText Example
ABC Street,
Spintex Road,
Accra,
Ghana.

---

textContent Example ABC Street, Spintex Road, Accra, Ghana.

Result of updating content with `innerText` and `textContent`.

Notice how `innerText` uses the line breaks but the `textContent` example doesn't.

Another key difference between the two methods is how they behave when used inside loops. `innerText` can be slower than `textContent` when dealing with bulk operations or frequent updates in a loop.

Read this freeCodeCamp article to learn more about the difference between `innerHTML`, `innerText`, and `textContent`.

How to Work with Attributes of DOM Elements

HTML attributes provide useful information about HTML elements. These attributes are always included in the opening tag of the element. An attribute is made up of a name and a value (though there are exceptions where only a name is present).

objects.

Here's an example:

Assume there's a button in the HTML document with the following attributes:

```
<button id="myBtn" type="submit">Click Here</button>
```

For this example, the browser will create an `HTMLButtonElement` object in the DOM. And the object will have properties matching the attributes.

- `HTMLButtonElement.id` with a value of `myBtn`.
- `HTMLButtonElement.type` with a value of `submit`.

To interact with and manipulate these attributes using JavaScript, you can use methods like `getAttribute` and `setAttribute` to directly access the properties.

The `getAttribute` Method

Like the name suggests, you can use this method to get the value of an existing attribute on an element.

It accepts an argument (the name of the attribute) and returns the value of the attribute. If the attribute you passed to it as an argument

```
<img src="image.jpg" alt="An example image">
```

```
const imageElement = document.querySelector('img')
console.log(imageElement.getAttribute('src'))
```

```
image.jpg
```

The `getAttribute` method is used to get the value of the `src` attribute.

Using the `getAttribute` method is the above example, you can get the value of the `src` attribute for the image element.

The `setAttribute` Method

This is used to set or change the attribute of an element. The method takes in two arguments. The first argument is the name of attribute you want to change, and the second is the new value you want to give the attribute.

Here's an example:

```
const imageElement = document.querySelector('img')

console.log("BEFORE:", imageElement.getAttribute('src'))
imageElement.setAttribute('src', 'new-image.jpg')
console.log("AFTER:", imageElement.getAttribute('src'))
```

```
BEFORE: image.jpg
AFTER: new-image.jpg
```

The `setAttribute` method is used to update the value of the `src` attribute.

This code example logs the value of the `src` attribute before and after using `setAttribute` to change it from `image.jpg` to `new-image.jpg`.

When you pass an argument to the `setAttribute` method and that attribute doesn't exit on the element, it will create a new attribute. For example, you can add a height property to the image element like so:

```
const imageElement = document.querySelector('img')

console.log("BEFORE:", imageElement.getAttribute('height'))
imageElement.setAttribute('height', '200')
console.log("AFTER:", imageElement.getAttribute('height'))
```

```
BEFORE: null
AFTER: 200
```

An example of adding a new height attribute to the image element.

The first log statement returns `null` because the height attribute was non-existent. But after setting it with the `setAttribute` method, the second log statement returns the correct value of the height.

The `removeAttribute` Method

In the previous section, you learned how to add a new attribute using the `setAttribute` method. What if you wanted to remove an existing attribute?

You can use the `removeAttribute` method. You pass in the name of the attribute you want to remove from the element as arguments to the method.

Here's an example:

```
<img src="image.jpg" alt="An example image" height="200">
```

Let's use the `removeAttribute` method to remove the `height` attribute from the image element.

```
imageElement.removeAttribute('height', '200')
console.log("AFTER:", imageElement.getAttribute('height'))
```

```
BEFORE: 200
AFTER: null
```

An example of using the `removeAttribute` method.

Before using `removeAttribute`, the first log statement prints the value of the height attribute, `200`. But after using the `removeAttribute` method, the second log statement prints `null`, confirming the removal of the height attribute from the image element.

The `hasAttribute` Method

Another method for working with attributes in the DOM is the `hasAttribute` method. You can use this method to check whether or not an element has a specific attribute.

The `hasAttribute` method returns `true` if the specified attribute exists and returns `false` if it doesn't.

Here's an example:

```
<img src="image.jpg" alt="An example image" height="200">
```

`width` values on the image element.

```
const imageElement = document.querySelector('img')

console.log("HEIGHT", imageElement.hasAttribute('height'))
console.log("WIDTH", imageElement.hasAttribute('width'))
```

```
HEIGHT: true
WIDTH: false
```

Example of using the `hasAttribute` method to check if an attribute exists.

The check for height returns `true` because it's an existing attribute while the check for width returns `false` because it doesn't exist.

How to Change the Styles on DOM Elements

There are two main ways of styling elements when working with the DOM in JavaScript. You can use the `.style` property or you can use classes. Each has its benefits and situations it's best situated for.

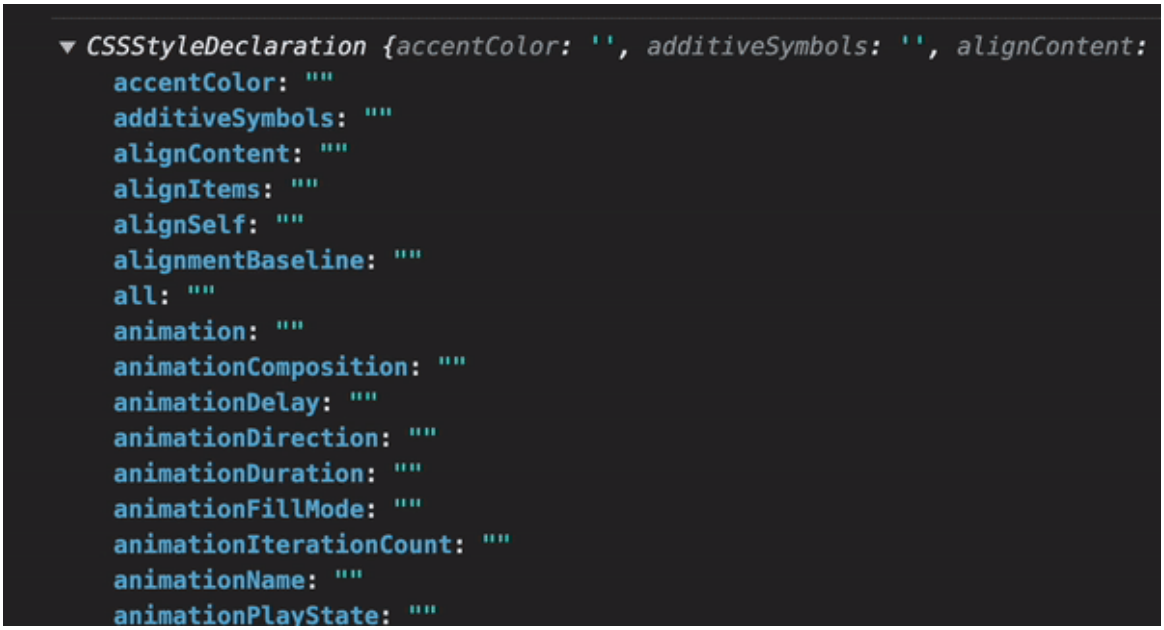
Setting Styles With the `.style` Property

you have in your CSS stylesheet.

Using the `.style` property, you get access to all the individual CSS properties. See the demo below:

```
<h1>Styling elements with JavaScript</h1>
```

```
const header = document.querySelector('h1')
console.log(header.style)
```



CSSStyleDeclarations for an `h1` element logged to the console.

Now, let's see an example of how to use the `.style` property.

```
<h1>I love JavaScript</h1>
```

An example `h1` header element

Here is an `h1` header. Now, let's add style to it using the `.style` property.

```
const paragraph = document.querySelector('h1')

paragraph.style.color = 'white'
paragraph.style.backgroundColor = 'green'
```

The style property is used to add a background colour to the `h1` element.

you would write `background-color`. But in your JavaScript code, you need to use camel case. So `background-color` becomes `backgroundColor`.

You can also delete a style on an element by setting the value of the property to an empty string.

```
element.style.propertyName = ""
```

Setting Styles with Classes

With classes, you can create styles once and apply it to different elements. This helps make your code become more maintainable.

The `className` Property

The `className` property represent the class attribute of a DOM element. And you can use it to get or set the value of the class attribute.

Here's an example:

```
<p class="food rice-dish">Jollof rice</p>
```

```
jollofParagraph.className = 'favorite'

console.log(jollofParagraph.className)
```

```
food rice-dish
favorite
```

Example of changing the value of a class with the `className` property.

The `className` also reads or replace the current class. In the example above, the first log statement prints the original value of the class. And after updating the `className`, the second log statement prints the new value for class.

But there is a more flexible property. For example, what if instead of replacing the old class with the new class, you wanted to add another class? That's where the `classList` property comes in.

The `classList` Property

With the `classList` property, you can add and remove classes. You can also toggle classes, replace existing class values with new ones, and even check if the class contains a specific value.

Here's an example:

```
const jollofParagraph = document.querySelector('p')
console.log(jollofParagraph.classList)
```

```
▶ DOMTokenList ['food', value: 'food']
```

Shows the current classList with only one value

Adding Classes with `classList.add()`

```
jollofParagraph.classList.add('fav', 'tasty')

console.log(jollofParagraph.classList)
```

```
▶ DOMTokenList(3) ['food', 'fav', 'tasty', value: 'food fav tasty']
```

Example of adding new classes with `classList.add` .

The code adds two new classes `fav` and `tasty` to the class list.

Removing Classes With `classList.remove()`

```
▶ DOMTokenList(2) ['food', 'fav', value: 'food fav']
```

Example of removing classes with `classList.remove`.

The code removes the class `tasty` from the class list.

Replacing Classes with `classList.replace()`

```
jollofParagraph.classList.replace('fav', 'favorite')

console.log(jollofParagraph.classList)
```

The code replaces the class `fav` with `favorite`

```
▶ DOMTokenList(2) ['food', 'favorite', value: 'food favorite']
```

Example of replacing classes with `classList.replace`.

Check the Presence of a Class with `classList.contains()`

```
console.log("Contains favorite: ", isFavorite)
console.log("Contains soup: ", isSoup)
```

```
Contains favorite:  true
Contains soup:  false
```

Example checking if a class exists with `classList.contains`.

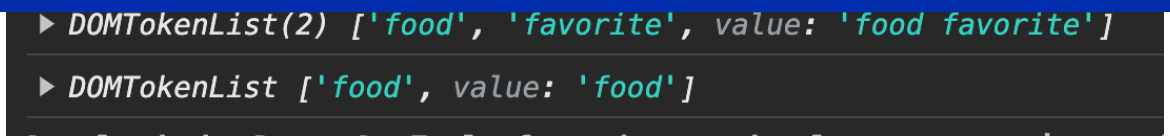
The code checks if the class passed to it is contained in the class list.

It returns `true` if it is included in the class list (for example `favorite`) and `false` if it is not included in the class list (for example `soup`)

Toggling a Class with the `classList.toggle()`

When you use the `toggle` property, it first checks if the class exists. If it exists, it will remove it. And if it doesn't exist, it will add it.

```
jollofParagraph.classList.toggle('favorite')
console.log(jollofParagraph.classList)

jollofParagraph.classList.toggle('favorite')
console.log(jollofParagraph.classList)

jollofParagraph.classList.toggle('favorite')
console.log(jollofParagraph.classList)
```



Example of toggling a class value with `classList.toggle`.

The first time the toggle runs, `favorite` exists in the class list. So, the toggle removes it.

The second time the toggle runs, `favorite` doesn't exist so the toggle adds `favorite` to the class list.

The next time the toggle runs, `favorite` now exists again. So it removes it from the class list.

The toggle keeps adding or removing the value from the class list depending on whether it's present or absent.

How to Traverse the DOM

To traverse the DOM means to move between the different elements/nodes within the HTML document. This may includes selecting or accessing parent, child, or sibling elements (or nodes). You do this to get information or manipulate the document structure.

But before we get into how to traverse the DOM, you need to understand the difference between nodes and elements.

Difference Between a Node and an Element

Elements are a specific type of node, but not all nodes are elements. Other types of content like attributes of elements, text content, and comments within the code are nodes too. But they are not elements.

An element is a specific type of node that defines the structure of the document's content. Think of elements as the familiar HTML tags you use. Examples include `<div>`, `<p>`, and `<ul>`. Each element can consist of attributes, text content, and other nested elements.

Selecting a Parent with `parentNode` vs `parentElement`

When it comes to selecting the parent of a DOM element, you can use either the `parentNode` or `parentElement`. Both will get the parent of the element you pass to it.

From a practical viewpoint, the parent of an element or a node will always be an element. So it doesn't matter which one you use, you will get the right parent of the selected element.

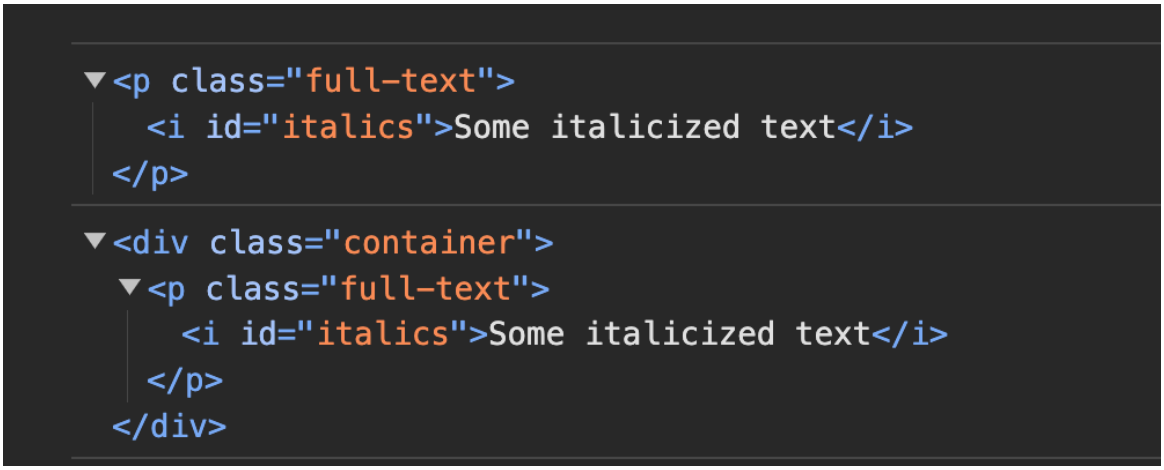
Let's see an example of selecting the parent of an element.

```
<div class="container">
  <p class="full-text">
      <i id="italics">Some italicized text</i>
  </p>
</div>
```

```
console.log(italicizedText.parentNode.parentNode)
```

First, you select the element. Then, you chain the `parentNode` method to it to get the parent. You can also chain another `parentNode` property to get the parent of a parent element like the second log statement.

The screenshot below shows the output of the two log statements.



Example of selecting the parent of an element.

Selecting Elements with `childNodes` vs `children`

You can select the contents of an element using both the `.childNodes` and `.children` properties. But they work differently.

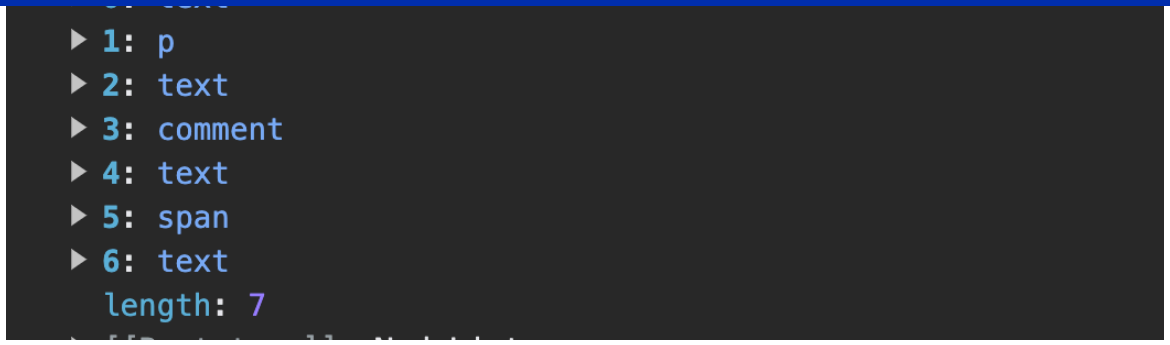
`.children`: returns an HTML collection of only the child elements (element nodes) of the selected objects. It will not include any non-element nodes like texts or comments.

Let's see an example that shows the difference:

```
<div id="container">
  A text node
  <p>Some paragraph</p>
  <!-- This is a comment -->
  <span>Span Element</span>
</div>
```

The code above has only 2 child elements (element nodes): the paragraph and the span. But there are other elements too – a text node and a comment:

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

const containerChildNodes = container.childNodes;
const containerChildren = container.children;

console.log(containerChildNodes);
console.log(containerChildren);
```

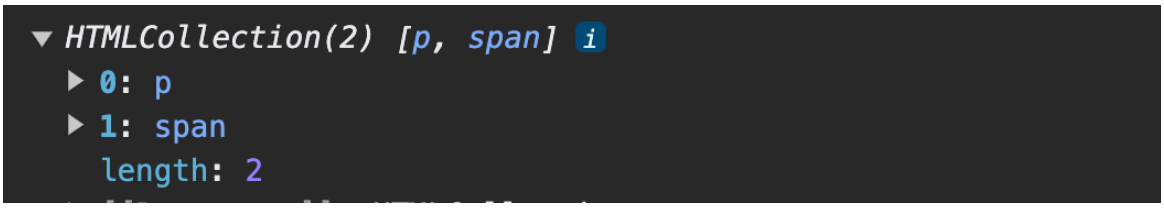


An example of using the .childNodes property

The `childNodes` will return all the child nodes (both elements and non-elements). It also includes the whitespaces between elements as text nodes.

This can be confusing to work with. So, unless you have a good reason not to, you should stick with the `.children` property.

The `children` will only return the child elements (the paragraph and the span).



An example of using the `.children` property

Selecting the First or Last Child/Element

- `firstChild`: Selects only the first child node of the parent element.
- `lastChild`: Selects only the last child node of the parent element.
- `firstElementChild`: Selects only the first child element of the parent.
- `lastElementChild`: Selects only the last child element of the parent.

Let's use the same example from the previous section to see how each works:

```
<div id="container">
  A text node
  <p>Some paragraph</p>
  <!-- This is a comment -->
  <span>Span Element</span>
</div>
```

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

console.log("FIRST CHILD:", container.firstChild)
console.log("LAST CHILD:", container.lastChild)
console.log("FIRST ELEMENT: ", container.firstElementChild)
console.log("LAST ELEMENT:", container.lastElementChild)
```

```
LAST CHILD:  ▶ #text
FIRST ELEMENT:    <p>Some paragraph</p>
LAST ELEMENT:   <span>Span Element</span>
```

Example demo selecting first child/element and last child/element

Note how `firstChild` returns the first text node but the `firstElementChild` returns the first paragraph instead. This means it ignored the text node which comes before the paragraph.

And also note how the `lastChild` returns a text node – even though from the markup, it looks like there's nothing after the span. That is because the `lastChild` property considers the linebreak/whitespace between the closing tag of the span and the closing tag of the div elements as a node.

That's why it's generally safer to stick to `firstElementChild` and `lastElementChild`.

Selecting a Sibling of Nodes in the DOM

You've learned how to select a parent or a child of an element. You can also select a sibling of an element. You do that using the following properties:

- `nextSibling`: Selects the next node within the same parent element.

- `previousSibling`: Selects the previous node within the same parent element.
- `previousElementSibling`: Selects the previous element, and ignores any non-element nodes.

Here's an example:

```
<div>
  <p id="one">First paragraph</p>
  text node
  <p id="two">Second paragraph</p>
  another text node
  <p id="three">Third paragraph</p>
  <p id="four">Fourth paragraph</p>
</div>
```

```
const paragraphTwo = document.getElementById('two')

console.log("nextSibling: ", paragraphTwo.nextSibling)
console.log("nextElementSibling: ", paragraphTwo.nextElementSibling)
console.log("previousSibling: ", paragraphTwo.previousSibling)
console.log("previousElementSibling: ", paragraphTwo.previousElementSibli
```

```
previousSibling:    " text node "
previousElementSibling:    <p id="one">First paragraph</p>
```

Examples of selecting siblings of a node.

`nextSibling` and `previousSibling` select the text nodes because they consider all nodes within the parent. While `nextElementSibling` and `previousElementSibling` select only the paragraph elements because they ignore non-element nodes like text.

DOM Events and Event Listeners

DOM events are actions that take place in the browser. These events are what allows you to make websites interactive.

Some DOM events are user-initiated like clicking, moving the mouse, or typing on the keyboard. Others are browser-initiated like when a page finishes loading.

Difference Between Event Listener and Event Handler

An event listener is a method that lets you know when an event has taken place. It allows you to "listen" or keep an eye out for DOM events. That way, when an event happens, you can do something.

An event handler is a response to the event. It's a function that runs when an event occurs.

handler (a function) that prints something on screen anytime a click event occurs.

In this case, the event listener is what informs your app when a click occurs and then trigger a response. And the response (the function that runs when the click occurs) is an example of an event handler.

Three Ways to Register Events in JavaScript

The following are three different ways you can listen to and respond to DOM events using JavaScript.

- **Using inline event handlers:** This is when you add the event listener as an attribute to HTML elements. In the early days of JavaScript, this was the only way to use events. See the example below:

```
// Example of using an inline event handler

<button onclick="alert('Hello')">Click me!</button>
```

- **Using on-event handlers:** You use this when an element has only one event handler. When you add more than one event handler using this method, only the last event handler will run, as it will override others before it.

```
<script>
  const myButton = document.querySelector('button')

  myButton.onclick = function() {
    console.log("Run first handler")
  }

  myButton.onclick = function() {
    console.log("Run second handler")
  }
</script>
```

```
Run second handler
```

Only the second event handler is executed.

As you can see from the result in the console, the browser runs the code for only the second event handler.

- **Using the** `addEventListener` method: This method allows you to attach more than one event handlers to an element. And it will execute them in the order in which they were added.

As a general rule, you should stick with the `addEventListener`, unless you have a compelling reason not to.

The `addEventListener` method takes two arguments. The first is the event you want to listen to, and the second is the event handler which

```
<!-- An example of using the addEventListener method -->

<button>Click me!</button>

<script>
  const myButton = document.querySelector('button')

  myButton.addEventListener('click', function() {
    console.log("Run first handler")
  })

  myButton.addEventListener('click', function() {
    console.log("Run second handler")
  })
</script>
```

```
Run first handler
Run second handler
```

The `addEventListener` method executes both event handlers.

Practice Challenge

Here is a challenge for you before you move on. Try solving it on your own before you take a look at the solution.

Consider the HTML and CSS code below.

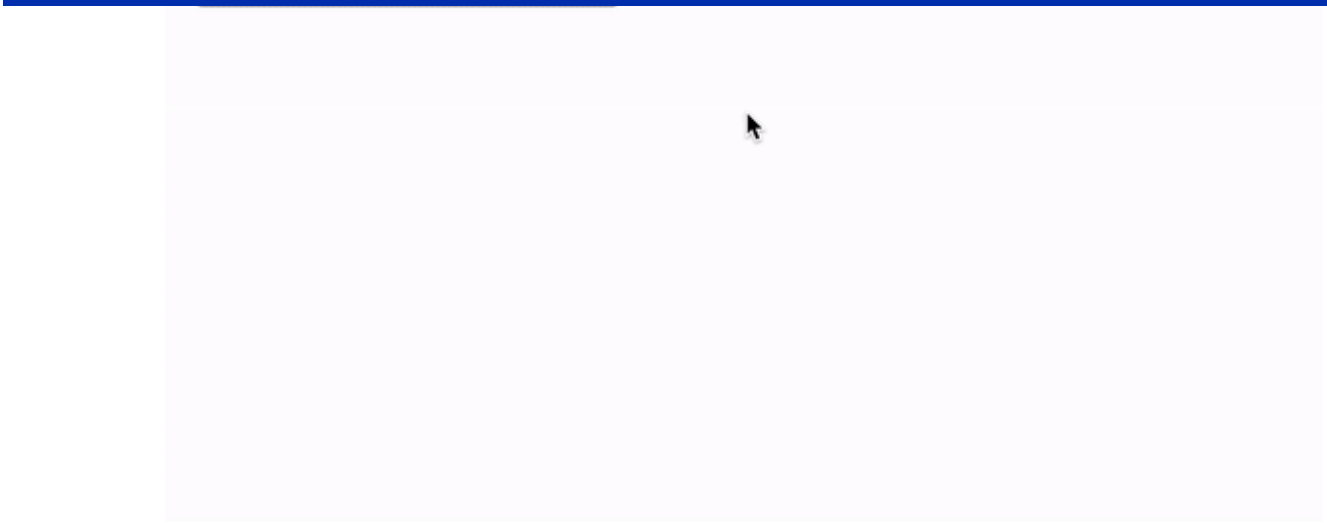
The challenge includes two elements. A `#gift-box` div and a `#click-btn` button. The gift box is hidden with the `.hide` class.

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

      <div id="gift-box" class="hide">🎁</div>
      <button id="click-btn">Show the box</button>

  </body>
</html>
```

```
.hide {
  display: none;
}

#gift-box {
  font-size: 5em;
}
```

<a href="https://stackblitz.com/edit/js-cywa91?file=index.html,style.css,index.js" target="_blank"

Solve the challenge on StackBlitz



Demo gif for the final solution of the challenge

Solution to Practice Challenge

Congratulations if you were able to solve the challenge. If you were not, that's okay. The solution and explanation is provided below:

```
const giftBoxElement = document.getElementById('gift-box')
const buttonElement = document.getElementById('click-btn')

buttonElement.addEventListener('click', function() {
  giftBoxElement.classList.remove('hide')
})
```

To solve this challenge, first you need to select both the `#gift-box` and `#click-btn` element.

Then, you add an event listener to the button. As mentioned earlier, the `addEventListener` method takes in two arguments.

The goal is to display the box. The box has a class `hide` which sets `display` to `none` in the CSS. One way to display the box using JavaScript is to remove `hide` from the classList.

The Event Object

This is a JavaScript object the browser passes as an argument to the event handler function anytime an event occurs. The object includes some useful properties and methods like the following:

- `type`: the type of event that occurred (like click, mouseover, keydown, and so on)
- `target`: the element on which the event occurred
- `clientX` and `clientY`: the horizontal and vertical coordinates of the mouse pointer at the time the event occurred.
- `preventDefault()`: prevents default actions associated with the events like preventing a form submission on the submit event.
- `stopPropagation()`: prevents the event from propagating through the DOM. More on that later.

You can see a full list of the properties and methods on the MDN web docs.

Types of Events

Mouse events:

- `click`: when the element is clicked.
- `dbclick`: when the element is double clicked.
- `mouseover`: when the mouse pointer enters the element.
- `mouseleave`: when the mouse pointer leaves the element.
- `mousedown`: when the mouse is pressed down over an element.
- `mouseup`: when the mouse is released over an element.

Keyboard events:

- `keydown`: when a key on the keyboard is pressed down.
- `keyup`: when a key on the keyboard is released.
- `keypress`: when a key is pressed and shows the actual key that was pressed. Note that this event is not fired for all keys, especially non-printable keys.

Form events:

- `submit`: when a form is submitted.
- `input`: when the value of an input field changes.
- `change`: when the value of a form element changes and loses focus.

Window events:

- `resize`: when the browser window is resized.
- `scroll`: when the user scrolls through the document.

You can see a comprehensive list of DOM events here.

Event Flow in JavaScript

When a JavaScript event occurs, the event is propagated or travels either from the target where the event occurred to the outermost element in the DOM or vice versa.

For example, let's say you click a button on a page. By clicking the button, you've also clicked its parent element and any element the button is inside within the DOM hierarchy.

Event Bubbling

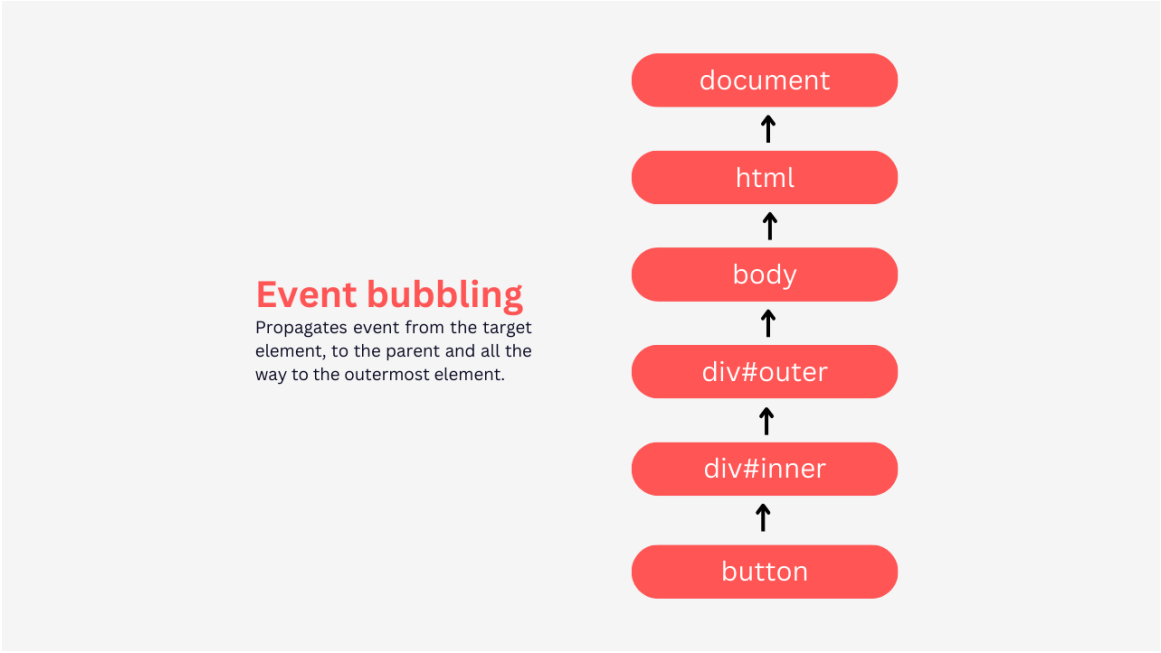
This is when the event is first registered on the target (or specified element) on which the event happened, and then registered outwards to the parent and onwards to the outermost element.

Here's an example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Event bubbling DEMO</title>
    </head>
    <body>
        <div id="outer">
```

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

The example here contains a button `#btn`. With event bubbling, when an event occurs (say a click) on the button, the event goes through the following sequence.

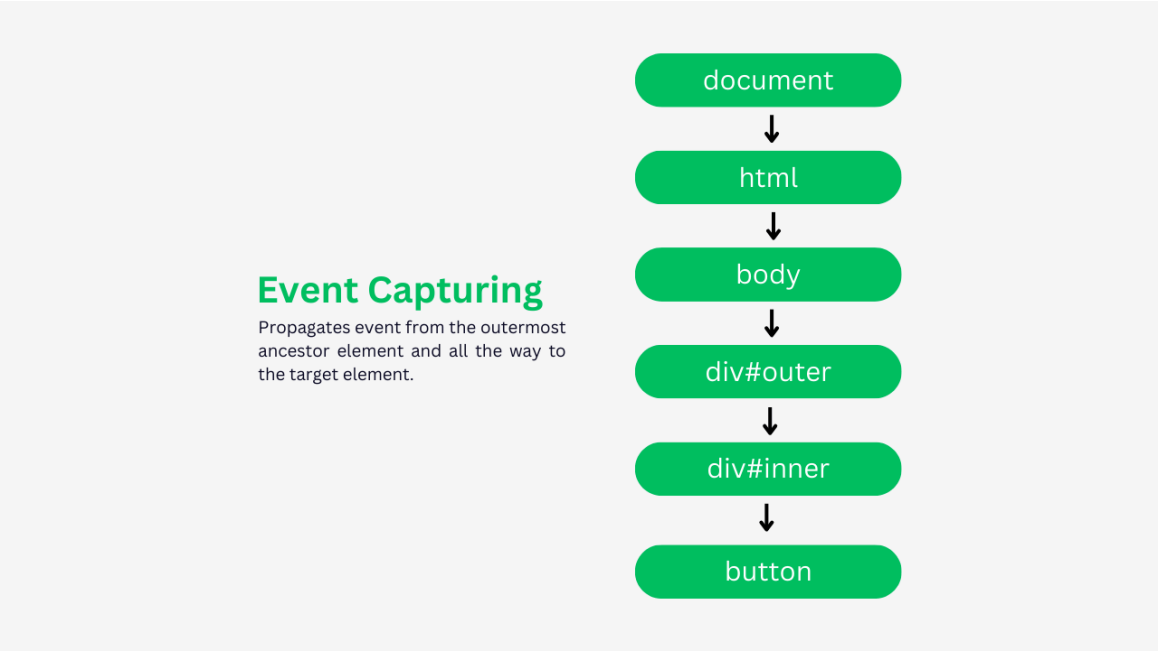


Event bubbling in DOM Manipulation: from button to div#inner to div#outer to body to HTML to document.

The event starts to bubble up from the target element back to the outermost ancestor.

Event Capturing

to the target element.



Event capturing in DOM Manipulation

During the capturing phase, event listeners attached to elements are executed in the order of the hierarchy from the topmost ancestor to the target element.

In case you're wondering why this matters, let's see a practical example using the same HTML markup example from above:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Event bubbling DEMO</title>
    </head>
    <body>
```

```
        </div>
    </body>
</html>
```

Let's add event listeners to the button, the `#inner` div, and the `#outer` div:

```
const button = document.getElementById('btn')
const innerDiv = document.getElementById('inner')
const outerDiv = document.getElementById('outer')

button.addEventListener('click', function() {
  console.log('Click on button')
})

innerDiv.addEventListener('click', function() {
  console.log('Click on inner Div')
})

outerDiv.addEventListener('click', function() {
  console.log('Click on outer Div')
})
```

By default, browsers use the event bubbling approach. So there is no need to add any argument to the event listener. This is the order in which the event handlers will run in response to a click on the button:

1. `button`
2. `#innerDiv`
3. `#outerDiv`

```
Click on inner Div
Click on outer Div
```

Events are executed from the element to the outermost element in the bubbling phase.

To use the event capturing model, you need to pass a third argument `true` to the event listener.

```
const button = document.getElementById('btn')
const innerDiv = document.getElementById('inner')
const outerDiv = document.getElementById('outer')

button.addEventListener('click', function() {
  console.log('Click on button')
}, true)

innerDiv.addEventListener('click', function() {
  console.log('Click on inner Div')
}, true)

outerDiv.addEventListener('click', function() {
  console.log('Click on outer Div')
}, true)
```

The order for executing the event handlers will now run in the opposite direction, like this:

1. `#outerDiv`
2. `#innerDiv`

```
Click on outer Div
Click on inner Div
Click on button
```

Events are executed from the outermost element to the element in the capturing phase.

The Event `stopPropagation()` Method

You've learned about how the event bubbling registers an event on an element and continues registering the event all the way to the outermost ancestor element. You've also seen how event capturing does the opposite.

But what if you don't want the event to register on all the ancestors? That's where the `stopPropagation` method comes in. You can use this method to prevent the event from propagating through the whole DOM.

Let's use the `stopPropagation` method on the same example from before:

```
button.addEventListener('click', function(event) {
  event.stopPropagation()
  console.log('Click on button')
})

innerDiv.addEventListener('click', function() {
```

```
  console.log('Click on outer Div')
})
```

```
Click on button
```

The `stopPropagation` method allows the execution of only the first event listener.

Now, only the event handler on the button is fired. The ones on the `innerDiv` and `outerDiv` are not because of the `stopPropagation` method on the button.

Also, note that to get the event object, you need to pass it as an argument to the event handler function.

JS DOM Manipulation Project Ideas

Building projects is an excellent way to improve your understanding of coding concepts. So roll up your sleeves and get ready to work!

Here are five JS DOM manipulation project ideas to help you practice and solidify your skills.

Toggle Switch

the current state of the toggle switch.

Random Color Picker

Create a simple app where users can click a button to generate a random color. Include a shape on the screen that gets filled with the chosen color. Also display the color code on screen.

Countdown Timer

Build a timer that starts from a specified time. The app updates in real time and shows the remaining time on the screen.

Word Counter

Develop an app that provides an input field or text area for the user to type. Display the number of words in real time on the screen as the user types.

An Interactive To-Do List

Create an app that allows users to add, delete, or edit tasks. You can have fun with this one and add as many advanced features as you want. For example, adding features like marking tasks as completed, filtering tasks, or sorting them.

Conclusion

If you've come this far, then you now have a good understanding of JavaScript DOM manipulation. With practice, you'll be confident

A good foundation of Vanilla JS DOM manipulation concepts will also come in handy when picking JavaScript libraries/frameworks like React, Angular, Vue, Svelte, and so on.

Thank you for reading, and happy coding! For more in-depth tutorials, feel free to subscribe to my YouTube channel.

Benjamin Semah

I am a full stack developer passionate about web accessibility. When I'm not behind the screens, you'll likely find me reading or going for a 5K run.

If you read this far, thank the author to show them you care.

Say Thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

You can make a tax-deductible donation here.

Trending Books and Handbooks

REST APIs	Clean Code	TypeScript
JavaScript	AI Chatbots	Command Line
GraphQL APIs	CSS Transforms	Access Control
REST API Design	PHP	Java
Linux	React	CI/CD
Docker	Golang	Python
Node.js	Todo APIs	JavaScript Classes
Front-End Libraries	Express and Node.js	Python Code Examples
Clustering in Python	Software Architecture	Programming Fundamentals
Coding Career Preparation	Full-Stack Developer Guide	Python for JavaScript Devs

Mobile App

Download on the App Store

GET IT ON Google Play

Our Charity