DOCUMENT OBJECT MODEL (DOM)

Introduction

The most popular way to run JavaScript is through web browser, and big part of this is to interact with elements on a page. When you write something in HTML and run in the browser, the browser will show the HTML code in the form of Document Object Model or DOM. It is not actually what we have written, it takes that, convert it into DOM, allows us to interface DOM via JavaScript. The DOM is represented in the form of tree which looks familiar to HTML.

Windows Object Reference

A window is where our global variables are stored and as well as helpful properties like windows.location(). If you go to any website console and type windows.location() it will provide you the website url and hole bunch of information. We can also find out things like innerwidth, which tells us how wide the browser is if you type it in the browser.

The Document Object Introduction

The document object is everything that is from opening <html> to doctype as well and closing </html>. The difference is the DOM is not concerned about other browser pages or something that is outside of the document. It's just concerned with the entire DOM. The entire DOM will be available to us with the document keyword. If you type document into the console and hover over what it returns, you will see that it highlights the entire page.

Where to Load JavaScript When Selecting Elements

The first thing we need to talk about is where to load our JavaScript if we are selecting elements.

Add the following the the JavaScript file:

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

If you wanted to load this JavaScript into the index.html, you might think you want to add it inside of the head tag, as shown below

```
<head> <meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1.0">
<title>The DOM</title>
<link rel="stylesheet" href="../../base.css">
<script src="./the-dom.js"></script>
</head>
```

Let's try that. Reload the page, which will cause the JavaScript to run, then open up the dev tools and take a look at what was logged in the console. It should be null. What we are doing in the JavaScript code we just added is we are trying to select something on the page, and then log it. If you try to run document.querySelector('p') in the console, it will return an element like so

```
document.querySelector('p')
<p> Hi, I’m an Item<p/>
```

Why does it work in the console, but not when we loaded it via JavaScript?

It is not working because our JavaScript is loading before of HTML is created. This causes the JavaScript to be downloaded and run before the actual elements have been created on page. It’s always better to write the script tag just above the closing body tag. That will ensure that all your HTML is first downloaded and parsed to the page before the JavaScript is in run.

Selecting DOM Elements

Before working with elements, you need to go get them, this is called selecting. We need to access the specific elements on the page (whether it’s an h2, div, button or an image). Once we have it, we can listen to clicks, change the content etc., There are 2 main ways of selecting an element.

1. querySelector()
2. querySelectorAll()

We generally use them with documents although that’s not always same. The difference between querySelector(‘p’) and querySelectorAll(‘div’) is in querySelector(‘p’) it selects the first matching paragraph where as in querySelectorAll(‘div’) it selects all the div elements that is available in the page and it is shown in the form of a nodelist. Nodelist looks like an array but it is list of things. Nodelist does not have all the methods like array does like map, filter and reduce build in to it.

You can get more specific than using elements as selectors.

document.querySelectorAll('.item');

Let's say you only wanted divs with a class of item. You would then use

document.querySelectorAll("div.item");

If you only wanted to grab an image inside of an item div, you could achieve that using the code below.

const divs = document.querySelectorAll('.item img');

If you wanted to just select the first item and then the image inside of it, you could add the code below instead, which will return the first match.

document.querySelector('.item img')

Searching Inside Already Selected Element

If you want to search inside of an element that you already selected, you can use querySelector on any other element like so.

const item = document.querySelector(‘.item’);

const item2 = item.querySelector(‘img’);

<h2 id=’web’>Frontend Development</h2>

If you have an element with an id, as shown in the code above,you would be able to grab it using document.getElementById('wes');.

The same works with document.getElementsByClassName(), but you don't pass the . from the classname.

You can get all the work done with document.querySelector() and document.querySelectorAll() however, and they are much more flexible.

Elements Properties and Methods

Start by selecting an h2 element from the page and logging it.

const heading = document.querySelector('h2');

console.log(heading);

Although in console it looks like the heading element is the actual variable, in reality it an object that has lot of properties and methods inside it. If you change console.log() into console.dir() that will show the object properties instead of actual element itself. There is outerText, textContent, outerHTML, and many other properties.

Getters and Setters

We will demonstrate this using textContent.

const heading=document.querySelector('h2');

console.dir(heading.textContent);

The code above is an example of getter. A setter is when you update a property. An example of that would be heading.textContent = ' cool';. Now when you reload the page, you will see Wes is cool in the console.

TextContent and InnerText

textContent and innerText are both similar properties, textContent is the new one. Difference between textContent and innerText is innerText returns human readable content will textContent will get the content of all the elements, including script and style elements.

<h2> I am a heading

 <style>

h2 { color:red; }

</style>

</h2>

If you log both the heading.textContent and heading.innerText, you will see that textContent includes the content within the style tag whereas innerText only returns the text I am a heading. innerText is aware of styling and won't return text of hidden elements.

Example:

```
<article class="item">

<h2>Im an article</h2>

<p class="pizza">This is how many pizzas I ate! 🍕</p>

</article>
```

What we are going to do is build something to add more pizza emojis to the end of it. First we need to select it.

```
const pizzaList = document.querySelector('.pizza');

console.log(pizzaList.textContent);
```

To update the textContent we could use the code below

```
pizzaList.textContent = `${pizzaList.textContent} 🍕`;
```

That will take what was already there and adds a pizza emoji to the end. That method can be slow in some applications that have lots of text and html, because it causes the browser to re-render the entire list.

InsertAdjacentText and InsertAdjacentElements

We can add text to the end using different method, either insertAdjacentText or insertAdjacentElement. That will give us the ability to add stuff to the back or the front of it. If you go to Mozilla Developer Network(MDN) and search for insertAdjacentText() you will see that it is not a property, it is a method meaning it is a function run against the element, like we do for querySelector() or querySelectorAll().

It takes two arguments:

1. The position (beforebegin, afterbegin, beforeend, afterend)
2. The text you want to pass.

```
pizzaList.insertAdjacentText('beforeend','🍕');
```

If you add the above code in the previous example you will see 2 pizza emojis at the end. If you use the below code you will see a pizza emoji infront of the pizzaList.

```
pizzaList.insertAdjacentText('beforebegin','🍕');
```

Working with classes

When you select an element it will have a classList attribute on it, and that attribute have methods on it for adding and removing classes. Duplicate one of the image tags and add it right after the opening body tag, and give it a class of nice.

```
<body><img class="nice" src="https://picsum.photos/500" />
```

Add the following code to select the element using the class of nice, and then we will log the classList attribute of the element.

```
const pic = document.querySelector('.nice');

console.log(pic.classList)
```

In the console we get a DomTokenList which is like an array of all the classes that are on that image. In the HTML add one more class name as cool to the image. When you refresh a page, you will see we got both of them as well as the value of all classes. ClassList has many methods lets see few of them

1. add
2. remove
3. contains
4. foreach

A lot of them are methods working for classes, which we are going to learn.

Adding a Class

We going to use pic.classList.add() to add a class called open as shown below.

```
const pic = document.querySelector(".nice");

pic.classList.add("open");

console.log(pic.classList);
```

Refresh the page, and inspect the image element. You will see the image now has a class of open.

Removing a Class

What if we wanted to remove the class of "cool" which already exists on the element?

You could do that with the following code

```
pic.classList.remove('cool');
```

Toggling a Class

Let’s write a bit of CSS so we can visually see what’s going on. Add a style tag in the HTML page as shown below.

```
<style> .round { border-radius: 50%; }</style>
```

Now using JavaScript we will add a class of round.

```
pic.classList.add(‘round’);
```

Now the image has a class of round and make the image circular. We can replace the add method given above with toggle(). Toggle will add class if it is not in there and will remove the class if it is present. If we go into our CSS and add a transition all for .2seconds that’s will give us an animation when the class is toggled.

```
img {

transition: all 0.2s;

}
```

```
.round {

 border-radius: 50%;

 transform: rotate(1turn) scale(2);

box-shadow: 0 0 10px black;

}
```

If you add the below code. When ever you click the image it calls the function called toggleRound(), which is toggle on and off the class of .round for the image element.

```
function toggleRound(){

pic.classList.toggle('round');

}

pic.addEventListener('click', toggleRound);
```

To have a roution transition to the image we can add the following code.

```
.round{

border-radius: 50%;

transform: rotate(20deg);

}
```

A lot of JavaScript interaction is just adding and removing classes at different points in time. That allows JavaScript developers to use CSS to add and remove transitions.

Contains Method

There is one more method called contains method as shown in below. Which will return a Boolean value which is either true or false it says weather the class is present or not.

```
pic.classList.contains(‘open’);
```

This is useful when you want to check the current state of the element by looking at its classList.

Build-in and Custom DataAttribute

When you are working with HTML elements those elements have something like attributes. Attributes are anything that provides the element with information things like alt, src, classes etc., are attributes. Most attributes are both setters and getters it means we can set the attribute and retrieve the attribute.

pic.alt = ‘Cute pup’; //setter

Console.log(pic.alt);//getter

Some attributes are how ever only getters for example naturalWidth.

console.log(pic.naturalWidth);

If you run the above code in JavaScript file it will return 0 in console. If you run it in console it will return 600.

Why is it zero when we log it from the JavaScript file, but running the code in the console returns 600?

This is the problem we frequently run into we have to wait until the image gets downloaded to know the actual width of it.

How can we overcome this problem?

One option is to addEventListener() on the load event as shown below.

windows.addEventListener(‘load’, function(){

console.log(pic.naturalWidth);

});

When you refresh your HTML page now, after a split second, you should see the naturalwidth value logged in the console. We can even add eventListener specially on the image as below.

pic.addEventListener(‘load’, function(){

console.log(pic.naturalWidth);

});

That still works, because once the image is loaded, the event listener will fire. All attributes on an element are done with getters and setters. You can use dot notation to access them.

You may have run into these methods already:

- getAttribute
- setAttribute
- hasAttribute

You can use getAttribute like console.log(pic.getAttribute('alt'); The console with return Cute Pup. You can use setters like pic.setAttribute('alt', 'REALLY CUTE PUP');. And there is hasAttribute which will return true or false based on whether that attribute is set on an element or not console.log(pic.hasAttribute('alt'));.

Data Attributes

If you do want your own custom attributes you need to use data attributes. Let's see an example to have a clear idea about data attributes. Let's write a HTML code by duplicating the image tag 3 times as shown below.

<img class='custom' data-name='web' src=https://picsum.photos/200/>

<img class='custom' data-name='kait' src=https://picsum.photos/200/>

<img class='custom' data-name='lux' src=https://picsum.photos/200/>

In this example, we wanted to attach a piece of metadata to each of the images.

If you want to attach metadata or something to an element that does not have any sort of standard attribute like a name attribute, then you can use data- anything.

For example: data-dog, data-name, data-description are all valid data attributes. That will allow you to add metadata to an element. Lets access the data attribute on the image with class name.

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

console.log(custom.data);
```

However, this will not work. The console returns undefined. If you access the data attribute on the element, we must use dataset as shown below. That gives us an object that is full of all the property values that you have.

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

console.log(custom.dataset);
```

Why would this be useful?

This is useful because we can do things like listen for a click on an element, and when someone clicks on it, we can alert them as shown below.

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

alert(´Welcome ${custom.dataset}´);

});
```

If you add that, and then click on the image, you would get an alert.

Creating HTML

There are few ways to create HTML in JavaScript. The main way is document.createElement(). To use this method you need to pass a tag name and then there are optional options you can pass. Lets try to make a paragraph tag using this method as shown below.

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

console.log(myParagraph);
```

When you open the HTML file, you will see the paragraph in the console but it is not visible on the page.

That is because we haven't actually put it on the page yet, we just created it and it's living in what is called memory right now. If you want to create a paragraph using JavaScript lets try.

```
const myParagraph = document.createElement(‘p’);

myParagraph.textContent(‘I am so cool’);

myParagraph.ClassList.add(‘special’);

console.log(myParagraph);
```

We now have that paragraph with a class of special and the text content that we supplied it with. Lets create few more elements before learning how to insert them into DOM.

```
const pic = document.createElement(‘img’);

pic.src = "https://picsum.photos/500";

pic.setAttribute(‘alt’, "Nice photo");

console.log(myImage);
```

We can create a div in JavaScript.

```
const myDiv = document.createElement(‘div’);

myDiv.classList.add(‘wrapper’);

console.log(myDiv);
```

Now we have created 3 elements:

1. a paragraph
2. an image
3. div

How do we add it to the page?

We use another API called appendChild. To use it, you have to first select an element to call .appendChild() against. If you want to add elements to the body you can directly grab document.body and insert it.

document.body is available to us because the document element gives us access to the body element quickly via a property. Not every element is as easily accessible to us like body is. appendChild() can be called against any node. So we can do something as shown below.

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

myDiv.appendChild(myParagraph);

myDiv.appendChild(pic);
```

It is best to do in reverse model because every time you are using appendChild you are modifying DOM. Which causes something called reflow in the browser. Which tells the browser that something changed and need to repaint HTML. This means that if you call appendChild() 3 times in a row, you are causing the browser to re-render 3 times in a row.

To solve that, you could modify the code like so

```
myDiv.appendChild(pic);

myDiv.appendChild(myParagraph);

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

What we are doing here is:

1. we are creating the elements and inserting the paragraph inside of the div
2. then we are inserting the image inside of the div as well
3. finally, we call document.body.appendChild(myDiv) last to modify the DOM

This approach causes the browser to re-paint only once as opposed to 3 times in the earlier approach.

InsertAdjacentElement Method

There is another API method which is useful that is insertAdjacentElement as insertAdjacentText as we discussed which is used to add text before, after and inside elements.

The difference is insertAdjacentElement is used to insert elements before, after and inside other elements. This is handy when you need to do something like insert an element before a paragraph in a div.

```
const heading = document.createElement(‘h2’);
```

heading.textContent(‘I am so cool’);

myDiv.appendChild(heading);

If you try using appendChild as shown in the code above, the heading is inserted after the image, which is not what we want. insertAdjacentElement takes 2 parameters.

1. The position where you want to insert the element.
2. The element you want to insert.

myDiv.insertAdjacentElement(‘afterbegin’, heading);

That will put the heading before the paragraph and image tag. If you were to use beforebegin, it would insert the header before the div tag, as a sibling to the wrapper. That may be something you will need to use. Let's say you have a few cards, and you need to add something after the first card. You can grab the second card and use beforebegin to insert an element before it.

HTML from Strings and XSS

We are going to learn creating HTML with strings using backtick strings. The main security flaw for this method is something called XSS cross-site scripting. There is a property called innerHTML like innerText as we discussed above. InnerHTML is a string of all the HTML that makes up what is inside of it.

const myDiv = document.createElement('div');

myDiv.classList.add('item');

const myParagraph = document.createElement('p');

myParagraph.textContent = 'I am so cool';

myDiv.appendChild(myParagraph);

document.body.appendChild(myDiv);

console.log(myDiv);

const myItem = document.querySelector(‘.item’);

console.log(myItem.innerHTML);

<u>OutPut:</u>

```
<div class="item">
    <p>I am so cool</p>
    </div>
```

```
<p>I am so cool</p>
```

Now this is the getter we used. Lets see a setter method for this.

const myItem = document.querySelector('.item');

myItem.innerHTML = `<h2> I love web </h2>`

console.log(myItem.innerHTML);

<u>OutPut:</u>

```
<div class="item">
    <h2> I love web </h2>
    </div>
```

```
<h2> I love web </h2>
```

The inside of the div.item has been updated with some HTML. One benefit of using backticks for single or double is that you can have a string that spans multiple lines as shown below.

item.innerHTML = ` <div>

<h1>Hey How are you?</h1>;

</div>`;

XSS

XSS is a cross-site scripting were people try's to insert script tags using a method like entering HTML string in a text input such as your profile name. Cross-site scripting is when a third part allows to inject a script tag through a security hole.

Traversing and Removing

Traversing means going up, down, over etc., when you have an element, you often need to select an element based on its position. For example, sometimes you are working on a button and need to select the parent div, or you need to look inside of the button for all the elements inside of it.

The difference between node and an element

We will see an example to explain this topic.

```
<p class="web">I am Web, I <em>love</em> to code</p>
```

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

```
console.log(web);
```

If instead you log web.children, it will return a collection of one thing in the console, which is the em tag we just added. If you log web.childer it will return an nodeList of three things

1. text
2. em
3. text

In the HTMLCollection in console it returns only em. If you place the cursor on one of the node in NodeList it highlights the corresponding node on the HTML page. Everything in our nodeList is a node if it is wrapped inside a tag then it is an element, but it does not work the other way around. If you only select elements, you won't have nodes returned. But if you select the nodes, you get all of the three different pieces.

Properties to work with nodes and elements

Let's add a few more elements inside of the p tag.

```
<p class="wes"> I am Web, I <em>love</em> to code and
<strong>Make websites!</strong></p>
```

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

```
console.log(web);
```

If you refresh that page, you will get the following results:

1. firstChildElement returns the emphasis element.
2. lastChildElement returns the strong element.
3. previousElementSibling is null.

Why the previousElement is null?

This is because we selected class web which does not conatin any previous class called as previous sibling so it returned null. There is nextSibilingElement which will grab the sibiling element after the current element. There is parentElement which will go up and grab the parentElement of the current element.

Removing Element

There is a method on every single element and every single node which have the ability to remove something. Let's try it with the code below.

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

p.textContent = "I will be removed";

document.body.appendChild(p);

p.remove();
```

If you open the index file in the browser, you won't see the p tag because we added it and then immediately removed it. The question now is what if you log p after you call remove(), will it be null, undefined? The fact that we had created that element and it exists in our JavaScript memory means that we do still have access to that paragraph element, and we could add it back in to the DOM. If you have reference to that element in JavaScript, and you've created it in your JavaScript, you can add it again.