Javascript and Document Object Model

Contents

Introduction

Purpose and Scope

The purpose of this document is to provide a comprehensive and practical understanding of JavaScript and the Document Object Model (DOM) and how they work together. It aims to serve as a valuable resource for web developers, both beginners and experienced, looking to harness the full potential of these technologies in their web projects.

The scope of this document encompasses:

Explaining the basics of JavaScript, including its history and relevance in web development.

Detailing the structure and functionality of the Document Object Model, or DOM.

Demonstrating how JavaScript interacts with the DOM to manipulate web page content and respond to user actions.

Providing practical examples and best practices for effective JavaScript and DOM coding.

Offering insights into the ongoing importance of these technologies in the rapidly evolving web development landscape.

Importance of JavaScript and DOM in Web Development

JavaScript and the Document Object Model are foundational technologies in web development for several compelling reasons:

- Interactivity: JavaScript allows web developers to create interactive elements, such as form validation, sliders, and interactive maps, making web applications more engaging and user-friendly.
- Dynamic Content: JavaScript enables the real-time modification of webpage content, reducing the need for full page reloads, resulting in a smoother and more responsive user experience.
- Cross-Browser Compatibility: JavaScript plays a crucial role in ensuring web applications work consistently across different browsers, helping developers overcome compatibility issues.
- DOM Manipulation: The DOM provides a structured representation of web pages, allowing developers to access and manipulate page elements with ease, making it a central aspect of web development.
- Event Handling: JavaScript facilitates event handling, allowing developers to respond to user interactions, such as clicks and keystrokes, effectively.
- Improved User Experience: Combining JavaScript and DOM enables the development of feature-rich web applications that keep users engaged, ultimately leading to a better user experience.

JavaScript Basics

What is JavaScript?

JavaScript is a scripting or programming language that allows you to implement complex features on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc. — you can bet that JavaScript is probably involved. It is the third layer of the layer cake of standard web technologies.

In web browsers, JavaScript consists of three main parts:

1. ECMAScript provides the core functionality.
2. The Document Object Model (DOM) provides interfaces for interacting with elements on web pages
3. The Browser Object Model (BOM) provides the browser API for interacting with the web browser.

JavaScript allows you to add interactivity to a web page. Typically, you use JavaScript with HTML and CSS to enhance a web page's functionality, such as validating forms, creating interactive maps, and displaying animated charts.

When a web page is loaded, i.e., after HTML and CSS have been downloaded, the JavaScript engine in the web browser executes the JavaScript code. The JavaScript code then modifies the HTML and CSS to update the user interface dynamically.

A Brief History of JavaScript

In 1995, JavaScript was created by a Netscape developer named Brendan Eich. First, its name was Mocha. And then, its name was changed to LiveScript.

Netscape decided to change LiveScript to JavaScript to leverage Java's fame, which was popular. The decision was made just before Netscape released its web browser product Netscape Navigator 2. As a result, JavaScript entered version 1.0.

Netscape released JavaScript 1.1 in Netscape Navigator 3. In the meantime, Microsoft introduced a web browser product called Internet Explorer 3 (IE 3), which competed with Netscape. However, IE came with its own JavaScript implementation called JScript. Microsoft used the name JScript to avoid possible license issues with Netscape.

Hence, two different JavaScript versions were in the market:

1. JavaScript in Netscape Navigator
2. JScript in Internet Explorer.

JavaScript had no standards that governed its syntax and features. And the community decided that it was time to standardize the language.

In 1997, JavaScript 1.1 was submitted to the European Computer Manufacturers Association (ECMA) as a proposal. Technical Committee #39 (TC39) was assigned to standardize the language to make it a general-purpose, cross-platform, and vendor-neutral scripting language.

TC39 came up with ECMA-262, a standard for defining a new scripting language named ECMAScript (often pronounced Ek-ma-script).

After that, the International Organization for Standardization and International Electrotechnical Commissions (ISO/IEC) adopted ECMAScript (ISO/IEC-16262).

JavaScript's Role in Web Development

JavaScript plays a pivotal role in web development for several reasons:

1. **Client-Side Scripting**: JavaScript is primarily a client-side scripting language, executed by web browsers. It enables dynamic page updates without the need to reload the entire page.
2. **Enhanced User Experience**: JavaScript enables the creation of interactive elements, real-time form validation, image sliders, and more, enhancing the user experience.
3. **Cross-Browser Compatibility**: JavaScript helps ensure that web applications work consistently across various browsers, thanks to its widespread support.
4. **DOM Manipulation**: JavaScript can access and manipulate the Document Object Model (DOM), making it an essential part of modern web development.

Document Object Model (DOM)

What is the Document Object Model?

The Document Object Model (DOM) is a fundamental component of web development, serving as a standardized programming interface for web documents. It acts as an intermediary between web pages and the underlying code, allowing dynamic manipulation of the document's structure, style, and content. In essence, it provides a mechanism to access and modify web pages using code. The DOM operates independently of the platform and programming language, making it a universal and essential tool for web development. By offering a structured representation of web documents, the DOM empowers developers to create interactive and responsive web applications, enhancing user experiences and functionality. The DOM represents an HTML document as a tree of nodes. The DOM provides functions that allow you to add, remove, and modify parts of the document effectively.

The Structure of the DOM

The Document Object Model (DOM) is structured as a hierarchical tree, and it is this tree structure that underpins its representation of web documents. In this hierarchical representation, each element of an HTML document is represented as a node within the tree. This tree-like structure facilitates the systematic organization of elements, their relationships, and their accessibility via JavaScript.

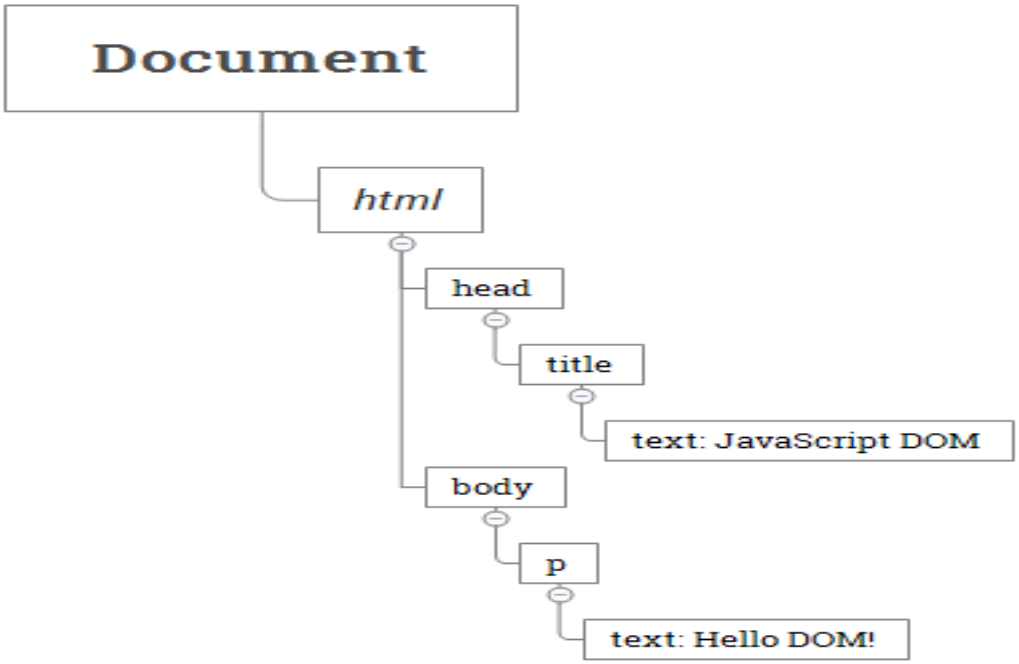
The DOM Tree

The DOM tree consists of nodes that represent different parts of the web page, including the document itself, elements, attributes, text, and more. Elements are organized hierarchically, and the relationship between nodes is parent-child.

Consider the following HTML document:

```
<html>
  <head>
    <title>JavaScript DOM</title>
  </head>
  <body>
    <p>Hello DOM!</p>
  </body>
</html>
```

The following tree represents the above HTML document:



In this DOM tree, the document is the root node. The root node has one child node which is the <html> element. The <html> element is called the document element.

Each document can have only one document element. In an HTML document, the document element is the <html> element. Each markup can be represented by a node in the tree.

Node Types

Each node in the DOM tree is identified by a node type. JavaScript uses integer numbers to determine the node types. The following table illustrates the node type constants:

Constant	Value	Description	Example
Node.ELEMENT_NODE	1	An Element node	This is an example of an element node. In this example, is an element node.
Node.TEXT_NODE	3	The actual text inside an Element or Attribute.	This is a text node example. The text inside the element, such as "This is a ", "text", and " node example.", are text nodes.
Node.CDATA_SECTION_NODE	4	CDATA sections are not commonly used in HTML, but they can be found in XML.	<message><![CDATA[This is a CDATA section]]></message> The content within <![CDATA[...]]> is a CDATA section.
Node.PROCESSING_INSTRUCTION_NODE	7	A ProcessingInstruction of an XML document.	<?xml-stylesheet type="text/xsl" href="styles.xsl"?> The line starting with <? and ending with ?> is a processing instruction node.
Node.COMMENT_NODE	8	Comments are often used for documentation or notes.	<!-- This is a comment node example --> The content within <!-- ... --> is a comment node.
Node.DOCUMENT_NODE	9	The document node represents the entire document, such as the entire HTML page.	<!-- Example of a Document Node --> <!DOCTYPE html> <html>

			<head> <title>Document Node Example</title> </head> <body> This is an example of a Document Node. </body> </html> In this HTML example, the entire HTML document, from <!DOCTYPE html> to </html>, is represented by the document node. Its nodeType is 9, and its nodeName typically returns "#document".
Node.DOCUMENT_TYPE_NODE	10	In an HTML document, the <!DOCTYPE html> declaration at the beginning is a DocumentType node.	<!-- Example of a Document Type Node --> <!DOCTYPE html> <html> <head> <title>Document Type Node Example</title> </head> <body> This is an example of a Document Type Node. </body> </html> In this HTML example, the <!DOCTYPE html> declaration at the beginning is a

			DocumentType node. Its nodeType is 10

Node.DOCUMENT_FRAGMENT_NODE	11	Document fragments are often used for temporary grouping of nodes. They don't have a direct HTML or XML representation but can be created programmatically.	var fragment = document.createDocumentFragment();
			
			var p = document.createElement('p');
			
			p.textContent = 'This is a document fragment example.';
			
			fragment.appendChild(p);
			
			In this example, fragment is a document fragment node

To get the type of node, you use the nodeType property:

```
node.nodeType
```

You can compare the nodeType property with the above constants to determine the node type. For example:

```
// Get a reference to an element in the DOM

var myElement = document.getElementById('myDiv');


// Check if the node is an element node

if (myElement.nodeType == Node.ELEMENT_NODE) {

   // It's an element node, so you can perform specific operations on it

   myElement.style.backgroundColor = 'lightblue';

}
```

Node and Element

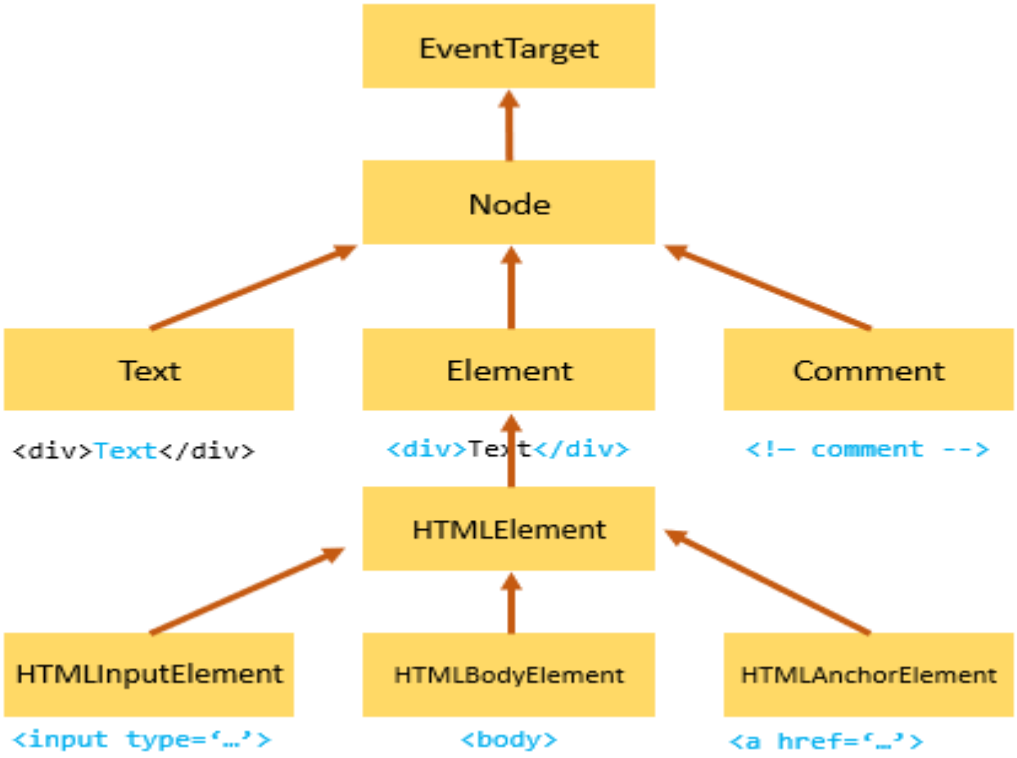
Sometimes it’s easy to confuse between the Node and the Element.

A node is a generic name of any object in the DOM tree. It can be any built-in DOM element such as the document. Or it can be any HTML tag specified in the HTML document like or <p>.

An element is a node with a specific node type Node.ELEMENT_NODE, which is equal to 1.

In other words, the node is the generic type of element. The element is a specific type of the node with the node type Node.ELEMENT_NODE.

The following picture illustrates the relationship between the Node and Element types:



Note that the getElementById() and querySelector() returns an object with the Element type while getElementsByTagName() or querySelectorAll() returns NodeList which is a collection of nodes.

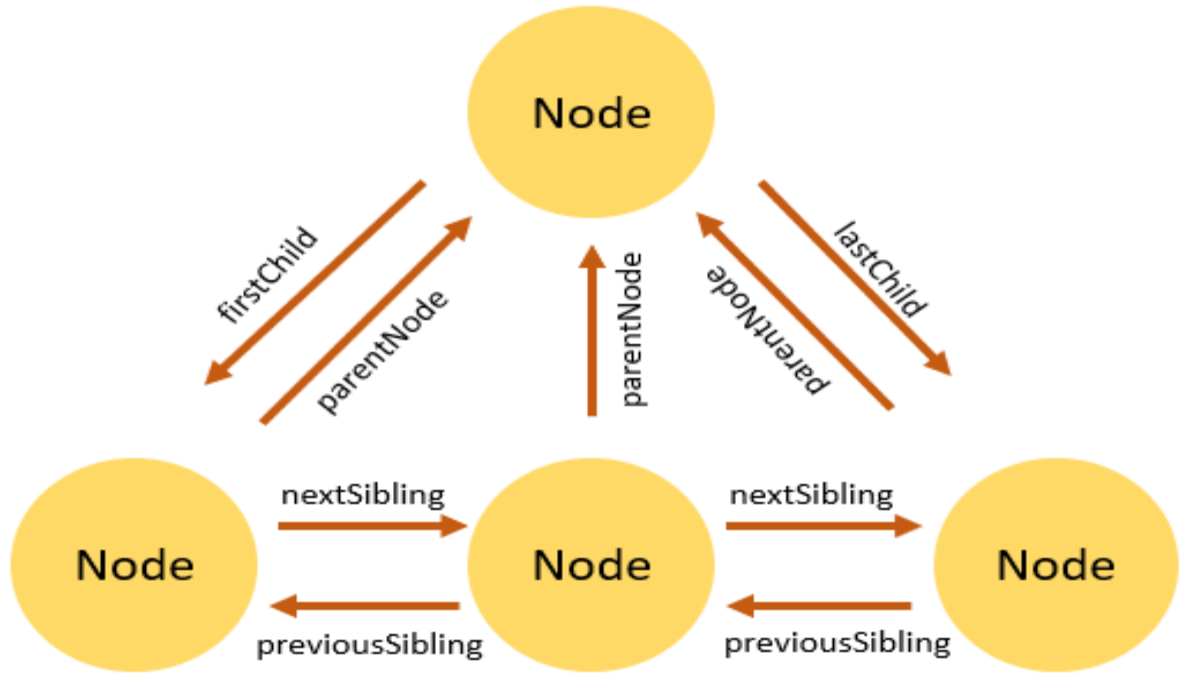
Node Relationships

Any node has relationships to other nodes in the DOM tree. The relationships are the same as the ones described in a traditional family tree.

For example, <body> is a child node of the <html> node, and <html> is the parent of the <body> node.

The <body> node is the sibling of the <head> node because they share the same immediate parent, which is the <html> element.

The following picture illustrates the relationships between nodes:



Demonstrating DOM Relationship

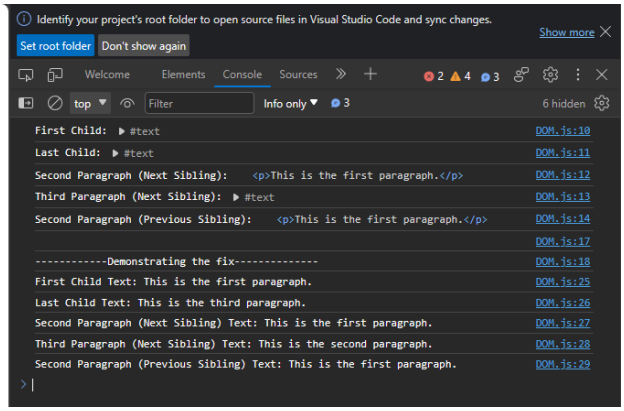
```html
<!DOCTYPE html>
<html>
<head>
    <title>DOM Navigation</title>
</head>
<body>
    <div id="parent">
        <p>This is the first paragraph.</p>
        <p>This is the second paragraph.</p>
        <p>This is the third paragraph.</p>
    </div>
    <script src="DOM.js"></script>
</body>
</html>
```

```javascript
// JavaScript code to demonstrate DOM navigation and display text content
const parent = document.getElementById("parent");
const firstChild = parent.firstChild; // Get the first child
const lastChild = parent.lastChild;   // Get the last child
const secondParagraph = firstChild.nextSibling; // Get the next sibling
const thirdParagraph = secondParagraph.nextSibling; // Get the next sibling
const previousSibling = thirdParagraph.previousSibling; // Get the previous sibling

//NOTE: If we use firstChild, lastChild, nextSibling, previousSibling then it would include the whitespaces in the document
console.log("First Child:", firstChild);
console.log("Last Child:", lastChild);
console.log("Second Paragraph (Next Sibling):", secondParagraph);
console.log("Third Paragraph (Next Sibling):", thirdParagraph);
console.log("Second Paragraph (Previous Sibling):", previousSibling);

//NOTE: To fix the above issue we use firstElementChild, lastElement, nextElementSibling, previousElementSibling. This would only get the elements in a document.
console.log("")
console.log("------------Demonstrating the fix-------------")
const firstElementChild = parent.firstElementChild;
const lastElementChild = parent.lastElementChild;
const secondParagraph1 = firstChild.nextElementSibling;
const thirdParagraph1 = secondParagraph.nextElementSibling;
const previousSibling1 = thirdParagraph.previousElementSibling;

console.log("First Child Text:", firstElementChild.textContent);
console.log("Last Child Text:", lastElementChild.textContent);
console.log("Second Paragraph (Next Sibling) Text:", secondParagraph1.textContent);
console.log("Third Paragraph (Next Sibling) Text:", thirdParagraph1.textContent);
console.log("Second Paragraph (Previous Sibling) Text:", previousSibling1.textContent);
```

This is the first paragraph.

This is the second paragraph.

This is the third paragraph.



Attributes and Properties

In the DOM, HTML attributes are exposed as properties of elements. For example, the id attribute of an element can be accessed as the id property of the element in JavaScript. Understanding the difference between attributes and properties is important when working with the DOM.

How DOM Represents Web Pages

The DOM represents a web page as a tree of objects, where each object corresponds to a part of the page, such as elements, attributes, and text. This representation allows developers to access and modify the page's content and structure.

Accessing DOM Elements

JavaScript provides various methods to access DOM elements, including getElementById, getElementsByClassName, and querySelector. These methods enable developers to locate and manipulate specific elements on a web page.

DOM Methods and Properties

The DOM exposes methods and properties that allow developers to interact with the page. For instance, methods like appendChild can be used to add elements to the DOM, and properties like innerHTML can be used to access or modify the content of an element.

DOM Events

DOM events allow web developers to respond to user interactions, such as clicks, mouse movements, and keyboard input. JavaScript code can be used to define event listeners that trigger specific actions when these events occur.

JavaScript and DOM Interaction

JavaScript is a powerful tool for modifying the DOM, allowing you to change the content, structure, and style of a web page dynamically. You can add, update, or delete elements and their attributes using JavaScript.

Selecting elements

getElementById() – select an element by id

The document.getElementById() method returns an Element object that represents an HTML element with an id that matches a specified string.

If the document has no element with the specified id, the document.getElementById() returns null.

Because the id of an element is unique within an HTML document, the document.getElementById() is a quick way to access an element.

Unlike the querySelector() method, the getElementById() is only available on the document object, not other elements.

The following shows the syntax of the getElementById() method:

```
const element = document.getElementById(id);
```

In this syntax, the id is a string that represents the id of the element to select. The id is case-sensitive. For example, the 'root' and 'Root' are totally different.

The id is unique within an HTML document. However, HTML is a forgiving language. If the HTML document has multiple elements with the same id, the document.getElementById() method returns the first element it encounters.

getElemenetByName() – select elements by name

Every element on an HTML document may have a name attribute:

```
<input type="radio" name="language" value="JavaScript">
```

Unlike the id attribute, multiple HTML elements can share the same value of the name attribute like this:

```
<input type="radio" name="language" value="JavaScript">

<input type="radio" name="language" value="TypeScript">
```

To get all elements with a specified name, you use the getElementsByName() method of the document object:

```
let elements = document.getElementsByName(name);
```

The getElementsByName() accepts a name which is the value of the name attribute of elements and returns a live NodeList of elements.

The return collection of elements is live. It means that the return elements are automatically updated when elements with the same name are inserted and/or removed from the document.

getElementsByTagName() – select elements by a tag name

The getElementsByTagName() is a method of the document object or a specific DOM element.

The getElementsByTagName() method accepts a tag name and returns a live HTMLCollection of elements with the matching tag name in the order which they appear in the document.

The following illustrates the syntax of the getElementsByTagName():

```
let elements = document.getElementsByTagName(tagName);
```

The return collection of the getElementsByTagName() is live, meaning that it is automatically updated when elements with the matching tag name are added and/or removed from the document.

Note that the HTMLCollection is an array-like object, like arguments object of a function.

getElementsByClassName() - select elements by one or more class

The getElementsByClassName() method returns an array-like of objects of the child elements with a specified class name. The getElementsByClassName() method is available on the document element or any other elements.

When calling the method on the document element, it searches the entire document and returns the child elements of the document:

```
let elements = document.getElementsByClassName(names);
```

However, when calling the method on a specific element, it returns the descendants of that specific element with the given class name:

```
let elements = rootElement.getElementsByClassName(names);
```

The method returns the elements which is a live HTMLCollection of the matches elements.

The names parameter is a string that represents one or more class names to match; To use multiple class names, you separate them by space.

querySelector() – select elements by CSS selectors.

The querySelector() is a method of the Element interface. The querySelector() method allows you to select the first element that matches one or more CSS selectors.

The following illustrates the syntax of the querySelector() method:

```
let element = parentNode.querySelector(selector);
```

In this syntax, the selector is a CSS selector or a group of CSS selectors to match the descendant elements of the parentNode.

If the selector is not valid CSS syntax, the method will raise a SyntaxError exception.

If no element matches the CSS selectors, the querySelector() returns null.

The querySelector() method is available on the document object or any Element object.

Besides the querySelector(), you can use the querySelectorAll() method to select all elements that match a CSS selector or a group of CSS selectors:

```
let elementList = parentNode.querySelectorAll(selector);
```

The querySelectorAll() method returns a static NodeList of elements that match the CSS selector. If no element matches, it returns an empty NodeList.

Note that the NodeList is an array-like object, not an array object. However, in modern web browsers, you can use the forEach() method or the for...of loop.

To convert the NodeList to an array, you use the Array.from() method like this:

```
let nodeList = document.querySelectorAll(selector);

let elements = Array.from(nodeList);
```

Traversing elements

Get the parent child – get the parent node of an element

To get the parent node of a specified node in the DOM tree, you use the parentNode property:

```
let parent = node.parentNode;
```

The parentNode is read-only.

The Document and DocumentFragment nodes do not have a parent. Therefore, the parentNode will always be null.

If you create a new node but haven't attached it to the DOM tree, the parentNode of that node will also be null.

Get child elements – get children of an element

Get the first child element

To get the first child element of a specified element, you use the firstChild property of the element:

```
let firstChild = parentElement.firstChild;
```

If the parentElement does not have any child element, the firstChild returns null. The firstChild property returns a child node which can be any node type such as an element node, a text node, or a comment node.

Get the last child element

To get the last child element of a node, you use the lastChild property:

```
let lastChild = parentElement.lastChild;
```

In case the parentElement does not have any child element, the lastChild returns null. Similar to the the firstChild property, the lastChild property returns the first element node, text node, or comment node. If you want to select only the last child element with the element node type, you use the lastElementChild property:

```
let lastChild = parentElement.lastElementChild;
```

Get all child elements

To get a live NodeList of child elements of a specified element, you use the childNodes property:

```
let children = parentElement.childNodes;
```

The childNodes property returns all child elements with any node type. To get the child element with only the element node type, you use the children property:

```
let children = parentElement.children;
```

Get siblings of an element – get siblings of element

Let's say you have the following list of items:

```
<ul id="menu">
  <li>Home</li>
  <li>Products</li>
  <li class="current">Customer Support</li>
  <li>Careers</li>
  <li>Investors</li>
  <li>News</li>
  <li>About Us</li>
```

```
</ul>
```

Get the next siblings

To get the next sibling of an element, you use the nextElementSibling attribute:

```
let nextSibling = currentNode.nextElementSibling;
```

The nextElementSibling returns null if the specified element is the last one in the list.

The following example uses the nextElementSibling property to get the next sibling of the list item that has the current class:

```
let current = document.querySelector('.current');

let nextSibling = current.nextElementSibling;

console.log(nextSibling);
```

Output:

```
<li>Careers</li>
```

In this example:

- First, select the list item whose class is current using the querySelector().
- Second, get the next sibling of that list item using the nextElementSibling property.

To get all the next siblings of an element, you can use the following code:

```
let current = document.querySelector('.current');

let nextSibling = current.nextElementSibling;

while(nextSibling) {

  console.log(nextSibling);

  nextSibling = nextSibling.nextElementSibling;

}
```

Get the previous siblings

To get the previous siblings of an element, you use the previousElementSibling attribute:

```
let current = document.querySelector('.current');

let prevSibling = currentNode.previousElementSibling;
```

The previousElementSibling property returns null if the current element is the first one in the list.

The following example uses the previousElementSibling property to get the previous siblings of the list item that has the current class:

```
let current = document.querySelector('.current');

let prevSiblings = current.previousElementSibling;

console.log(prevSiblings);
```

And the following example selects all the previous siblings of the list item that has the current class:

```
let current = document.querySelector('.current');

let prevSibling = current.previousElementSibling;

while(prevSibling) {

   console.log(prevSibling);

   prevSibling = current.previousElementSibling;

}
```

Get all siblings of an element

To get all siblings of an element, we'll use the logic:

- First, select the parent of the element whose siblings you want to find.
- Second, select the first child element of that parent element.
- Third, add the first element to an array of siblings.
- Fourth, select the next sibling of the first element.
- Finally, repeat the 3rd and 4th steps until there are no siblings left. In case the sibling is the original element, skip the 3rd step.

The following function illustrates the steps:

```
let getSiblings = function (e) {

   // for collecting siblings

   let siblings = [];

   // if no parent, return no sibling

   if(!e.parentNode) {

      return siblings;

   }
```

```
    // first child of the parent node
    let sibling  = e.parentNode.firstChild;
    // collecting siblings
    while (sibling) {
      if (sibling.nodeType === 1 && sibling !== e) {
        siblings.push(sibling);
      }
      sibling = sibling.nextSibling;
    }
    return siblings;
};
```

Manipulating elements

createElement() – create a new element.

To create an HTML element, you use the document.createElement() method:

```
let element = document.createElement(htmlTag);
```

The document.createElement() accepts an HTML tag name and returns a new Node with the Element type.

appendChild() – append a node to a list of child nodes of a specified parent node.

The appendChild() is a method of the Node interface. The appendChild() method allows you to add a node to the end of the list of child nodes of a specified parent node.

The following illustrates the syntax of the appendChild() method:

```
parentNode.appendChild(childNode);
```

In this method, the childNode is the node to append to the given parent node. The appendChild() returns the appended child.

If the childNode is a reference to an existing node in the document, the appendChild() method moves the childNode from its current position to the new position.

textContent – get and set the text content of a node.

Reading textContent from a node

To get the text content of a node and its descendants, you use the textContent property:

```
let text = node.textContent;
```

Suppose that you have the following HTML snippet:

```
<div id="note">
   JavaScript textContent Demo!
   <span style="display:none">Hidden Text!</span>
   <!-- my comment -->
</div>
```

The following example uses the textContent property to get the text of the element:

```
let note = document.getElementById('note');
console.log(note.textContent);
```

How it works.

First, select the div element with the id note by using the getElementById() method.

Then, display the text of the node by accessing the textContent property.

Output:

```
JavaScript textContent Demo!
Hidden Text!
```

As you can see clearly from the output, the textContent property returns the concatenation of the textContent of every child node, excluding comments (and also processing instructions).

Setting textContent for a node

Besides reading textContent, you can also use the textContent property to set the text for a node:

```
node.textContent = newText;
```

When you set textContent on a node, all the node's children will be removed and replaced by a single text node with the newText value. For example:

```
let note = document.getElementById('note');
```

```
note.textContent = 'This is a note';
```

innerHTML – get and set the HTML content of an element.

The innerHTML is a property of the Element that allows you to get or set the HTML markup contained within the element:

```
element.innerHTML = 'new content';

element.innerHTML;
```

Reading the innerHTML property of an element

To get the HTML markup contained within an element, you use the following syntax:

```
let content = element.innerHTML;
```

When you read the innerHTML of an element, the web browser has to serialize the HTML fragment of the element's descendants.

Setting the innerHTML property of an element

To set the value of innerHTML property, you use this syntax:

```
element.innerHTML = newHTML;
```

The setting will replace the existing content of an element with the new content.

For example, you can remove the entire contents of the document by clearing the contents of the document.body element:

```
document.body.innerHTML = '';
```

DocumentFragment – learn how to compose DOM nodes and insert them into the active DOM tree.

The DocumentFragment interface is a lightweight version of the Document that stores a piece of document structure like a standard document. However, a DocumentFragment isn't part of the active DOM tree.

If you make changes to the document fragment, it doesn't affect the document or incurs any performance.

Typically, you use the DocumentFragment to compose DOM nodes and append or insert it to the active DOM tree using appendChild() or insertBefore() method.

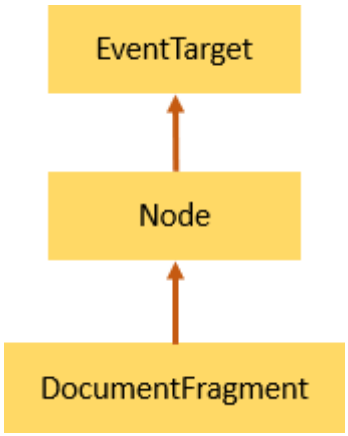
To create a new document fragment, you use the DocumentFragment constructor like this:

```
let fragment = new DocumentFragment();
```

Or you can use the createDocumentFragment() method of the Document object:

```
let fragment = document.createDocumentFragment();
```

This DocumentFragment inherits the methods of its parent, Node, and also implements those of the ParentNode interface such as querySelector() and querySelectorAll().



after() – insert a node after an element.

The after() is a method of the Element type. The element.after() method allows you to insert one or more nodes after the element.

Here's the syntax of the after() method:

```
Element.after(node)
```

In this syntax, the after() method inserts the node after the Element in the DOM tree.

For example, suppose you have a <h1> element and you want to insert a <p> element after it, you can use the after() method like this:

```
h1.after(p)
```

To insert multiple nodes after an element, you pass the nodes to the after() method as follows:

```
Element.after(node1, node2, ... nodeN)
```

The after() method also accepts one or more strings. In this case, the after() method treats the strings as Text nodes:

```
Element.after(str1, str2, ... strN)
```

The after() method returns undefined. If a node cannot be inserted, it'll throw a HierarchyRequestError exception.

before () – insert a node before an element.

The element.before() method allows you to insert one or more nodes before the element. The following shows the syntax of the before() method:

```
Element.before(node)
```

In this syntax, the before() method inserts the node before the Element in the DOM tree.

For example, suppose you have a <p> element and you want to insert a <h1> element before it, you can use the before() method like this:

```
p.before(h1)
```

To insert multiple nodes before an element, you pass the nodes to the before() method as follows:

```
Element.before(node1, node2, ... nodeN)
```

Also, the before() method accepts one or more strings rather than nodes. In this case, the before() method treats the strings as Text nodes:

```
Element.before(str1, str2, ... strN)
```

The before() method returns undefined. If a node cannot be inserted, the before() method throws a HierarchyRequestError exception.

append() – insert a node after the last child node of a parent node.

The parentNode.append() method inserts a set of Node objects or DOMString objects after the last child of a parent node:

```
parentNode.append(...nodes);

parentNode.append(...DOMStrings);
```

The append() method will insert DOMString objects as Text nodes.

Note that a DOMString is a UTF-16 string that maps directly to a string.

The append() method has no return value. It means that the append() method implicitly returns undefined.

prepend() – insert a node before the first child node of a parent node.

The prepend() method inserts a set of Node objects or DOMString objects before the first child of a parent node:

```
parentNode.prepend(...nodes);

parentNode.prepend(...DOMStrings);
```

The prepend() method inserts DOMString objects as Text nodes. Note that a DOMString is a UTF-16 string that directly maps to a string.

The prepend() method returns undefined.

insertAdjacentHTML() – parse a text as HTML and insert the resulting nodes into the document at a specified position.

The insertAdjacentHTML() is a method of the Element interface so that you can invoke it from any element.

The insertAdjacentHTML() method parses a piece of HTML text and inserts the resulting nodes into the DOM tree at a specified position:

element.insertAdjacentHTML(positionName, text);Code language: JavaScript (javascript)

The insertAdjacentHTML() method has two parameters:

1) position

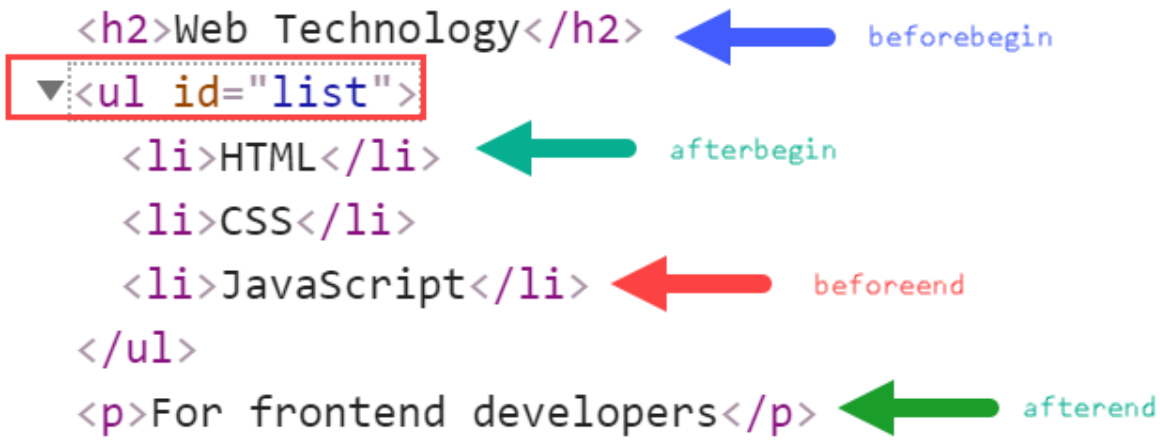
The positionName is a string that represents the position relative to the element.
The positionName accepts one of the following four values:

- 'beforebegin': before the element
- 'afterbegin': before its first child of the element.
- 'beforeend': after the last child of the element
- 'afterend': after the element

Note that the 'beforebegin' and 'afterend' are only relevant if the element is in the DOM tree and has a parent element.

The insertAdjacentHTML() method has no return value, or undefined by default.

The following visualization illustrates the position name:



2) text

The text parameter is a string that the insertAdjacentHTML() method parses as HTML or XML. It cannot be Node objects

replaceChild() – replace a child element by a new element.

Node.replaceChild() method to replace an HTML element by a new one.

To replace an HTML element, you use the node.replaceChild() method:

```
parentNode.replaceChild(newChild, oldChild);
```

In this method, the newChild is the new node to replace the oldChild node which is the old child node to be replaced.

cloneNode() – clone an element and all of its descendants.

The cloneNode() is a method of the Node interface that allows you to clone an element:

```
let clonedNode = originalNode.cloneNode(deep);
```

Pamaraters

deep

The cloneNode() method accepts an optional parameter deep:

If the deep is true, then the original node and all of its descendants are cloned.

If the deep is false, only the original node will be cloned. All the node's descendants will not be cloned.

The deep parameter defaults to false if you omit it.

```
originalNode
```

The originalNode is the element to be cloned.

Return value

The cloneNode() returns a copy of the originalNode.

removeChild() – remove child elements of a node.

To remove a child element of a node, you use the removeChild() method:

```
let childNode = parentNode.removeChild(childNode);
```

The childNode is the child node of the parentNode that you want to remove. If the childNode is not the child node of the parentNode, the method throws an exception.

The removeChild() returns the removed child node from the DOM tree but keeps it in the memory, which can be used later.

If you don't want to keep the removed child node in the memory, you use the following syntax:

```
parentNode.removeChild(childNode);
```

The child node will be in the memory until it is destroyed by the JavaScript garbage collector.

Working with Attributes

HTML Attributes & DOM Object's Properties – understand the relationship between HTML attributes & DOM object's properties.

When the web browser loads an HTML page, it generates the corresponding DOM objects based on the DOM nodes of the document.

For example, if a page contains the following input element:

```
<input type="text" id="username">
```

The web browser will generate an HTMLInputElement object.

The input element has two attributes:

- The type attribute with the value text.
- The id attribute with the value username.

The generated HTMLInputElement object will have the corresponding properties:

- The input.type with the value text.
- The input.id with the value username.

In other words, the web browser will automatically convert attributes of HTML elements to properties of DOM objects.

However, the web browser only converts the *standard* attributes to the DOM object's properties. The standard attributes of an element are listed on the element's specification.

Attribute-property mapping is not always one-to-one. For example:

```
<input type="text" id="username" secured="true">
```

In this example, the secured is a non-standard attribute:

```
let input = document.querySelector('#username');

console.log(input.secured); // undefined
```

Attribute methods

To access both standard and non-standard attributes, you use the following methods:

- element.getAttribute(name) – get the attribute value
- element.setAttribute(name, value) – set the value for the attribute

- element.hasAttribute(name) – check for the existence of an attribute
- element.removeAttribute(name) – remove the attribute

element.attributes

The element.attributes property provides a live collection of attributes available on a specific element. For example:

```
let input = document.querySelector('#username');

for(let attr of input.attributes) {

   console.log(`${attr.name} = ${attr.value}` )

}
```

Output:

```
type = text

id = username

secure = true
```

Note that element.attributes is a NamedNodeMap, not an Array, therefore, it has no Array's methods.

Attribute-property synchronization

When a standard attribute changes, the corresponding property is auto-updated with some exceptions and vice versa.

Suppose that you have the following input element:

```
<input type="text" id="username" tabindex="1">
```

The following example illustrates the change of the tabindex attribute is propagated to the tabIndex property and vice versa:

```
let input = document.querySelector('#username');

// attribute -> property

input.setAttribute('tabindex', 2);

console.log(input.tabIndex);  // 2



// property -> attribute

input.tabIndex = 3;
```

```
console.log(input.getAttribute('tabIndex')); // 3
```

The following example shows when the value attribute changes, it reflects in the value property, but not the other way around:

```
let input = document.querySelector('#username');

// attribute -> property: OK

input.setAttribute('value','guest');

console.log(input.value);  // guest


// property -> attribute: doesn't change

input.value = 'admin';

console.log(input.getAttribute('value')); // guest
```

DOM properties are typed

The value of an attribute is always a string. However, when the attribute is converted to the property of a DOM object, the property value can be a string, a boolean, an object, etc.

The following checkbox element has the checked attribute. When the checked attribute is converted to the property, it is a boolean value:

```
<input type="checkbox" id="chkAccept" checked> Accept

let checkbox = document.querySelector('#chkAccept');

console.log(checkbox.checked); // true
```

The following shows an input element with the style attribute:

```
<input type="password" id="password" style="color:red;with:100%">
```

The style attribute is a string while the style property is an object:

```
let input = document.querySelector('#password');

let styleAttr = input.getAttribute('style');

console.log(styleAttr);

console.dir(input.style);
```

Output:

```
[object CSSStyleDeclaration]
```

The data-* attributes

If you want to add a custom attribute to an element, you should prefix it with the data- e.g., data-secured because all attributes start with data- are reserved for the developer's uses.

To access data-* attributes, you can use the dataset property. For example, we have the following div element with custom attributes:

```
<div id="main" data-progress="pending" data-value="10%"></div>
```

The following shows how to access the data-* attributes via the dataset property:

```
let bar = document.querySelector('#main');

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

Output:

```
[object DOMStringMap] {
   progress: "pending",
   value: "10%"
}
```

setAttribute() – get the value of an attribute on an element.

To set a value of an attribute on a specified element, you use the setAttribute() method:

```
element.setAttribute(name, value);
```

Parameters

The name specifies the attribute name whose value is set. It's automatically converted to lowercase if you call the setAttribute() on an HTML element.

The value specifies the value to assign to the attribute. It's automatically converted to a string if you pass a non-string value to the method.

Return value

The setAttribute() returns undefined.

Note that if the attribute already exists on the element, the setAttribute() method updates the value; otherwise, it adds a new attribute with the specified name and value.

Typically, you use the querySelector() or getElementById() to select an element before calling the setAttribute() on the selected element.

To get the current value of an attribute, you use the getAttribute() method. To remove an attribute, you call the removeAttribute() method.

getAttribute() – get the value of an attribute on an element.

To get the value of an attribute on a specified element, you call the getAttribute() method of the element:

```
let value = element.getAttribute(name);
```

Parameters

The getAttribute() accepts an argument which is the name of the attribute from which you want to return the value.

Return value

If the attribute exists on the element, the getAttribute() returns a string that represents the value of the attribute. In case the attribute does not exist, the getAttribute() returns null.

Note that you can use the hasAttribute() method to check if the attribute exists on the element before getting its value.

removeAttribute() – remove an attribute from a specified element.

The removeAttribute() removes an attribute with a specified name from an element:

```
element.removeAttribute(name);
```

Parameters

The removeAttribute() accepts an argument which is the name of the attribute that you want to remove. If the attribute does not exist, the removeAttribute() method wil not raise an error.

Return value

The removeAttribute() returns a value of undefined.

Usage notes

HTML elements have some attributes which are Boolean attributes. To set false to the Boolean attributes, you cannot simply use the setAttribute() method, but you have to remove the attribute entirely using the removeAttribute() method.

hasAttribute() – check if an element has a specified attribute or not.

To check an element has a specified attribute or not, you use the hasAttribute() method:

```
let result = element.hasAttribute(name);
```

Parameters

The hasAttribute() method accepts an argument that specifies the name of the attribute that you want to check.

Return value

The hasAttribute() returns a Boolean value that indicates if the element has the specified attribute.

If the element contains an attribute, the hasAttribute() returns true; otherwise, it returns false.

Manipulating Element's Styles

style property – get or set inline styles of an element.

To set the inline style of an element, you use the style property of that element:

```
element.style
```

The style property returns the read-only CSSStyleDeclaration object that contains a list of CSS properties. For example, to set the color of an element to red, you use the following code

```
element.style.color = 'red';
```

If the CSS property contains hyphens (-) for example -webkit-text-stroke you can use the array-like notation ([]) to access the property:

```
element.style.['-webkit-text-stock'] = 'unset';
```

To completely override the existing inline style, you set the cssText property of the style object. For example:

```
element.style.cssText = 'color:red;background-color:yellow';
```

Or you can use the setAttribute() method:

```
element.setAttribute('style','color:red;background-color:yellow');
```

Once setting the inline style, you can modify one or more CSS properties:

```
element.style.color = 'blue';
```

If you do not want to completely overwrite the existing CSS properties, you can concatenate the new CSS property to the cssText as follows:

```
element.style.cssText += 'color:red;background-color:yellow';
```

In this case, the += operator appends the new style string to the existing one.

The following css() helper function is used to set multiple styles for an element from an object of key-value pairs:

```
function css(e, styles) {
  for (const property in styles)
    e.style[property] = styles[property];
```

}

You can use this css() function to set multiple styles for an element with the id #content as follows:

```
let content = document.querySelector('#content');

css(content, { background: 'yellow', border: 'solid 1px red'});
```

getComputedStyle() – return the computed style of an element.

The getComputedStyle() is a method of the window object, which returns an object that contains the computed style an element:

```
let style = window.getComputedStyle(element [,pseudoElement]);
```

Parameters

The getComputedStyle() method accepts two arguments:

element is the element that you want to return the computed styles. If you pass another node type e.g., Text node, the method will throw an error.

pseudoElement specifies the pseudo-element to match. It defaults to null.

For example, if you want to get the computed value of all the CSS properties of a link with the hover state, you pass the following arguments to the getComputedStyle() method:

```
let link = document.querySelector('a');

let style = getComputedStyle(link,':hover');

console.log(style);
```

Note that window is the global object, therefore, you can omit it when calling get the getComputedStyle() method.

Return value

The getComputedStyle() method returns a live style object which is an instance of the CSSStyleDeclaration object. The style is automatically updated when the styles of the element are changed.

className property – return a list of space-separated CSS classes

The className is the property of an element that returns a space-separated list of CSS classes of the element as a string:

```
element.className
```

To add a new class to an element using the className property, you can concatenate the existing class name with a new one:

```
element.className += newClassName;
```

To completely overwrite all the classes of an element, you use a simple assignment operator. For example:

```
element.className = "class1 class2";
```

To get a complete list of classes of an element, you just need to access the className property:

```
let classes = element.className;
```

Because the class is a keyword in JavaScript, the name className is used instead of the class.

Also the class is an HTML attribute while while className is a DOM property of the element

classList property – manipulate CSS classes of an element.

The classList is a read-only property of an element that returns a live collection of CSS classes:

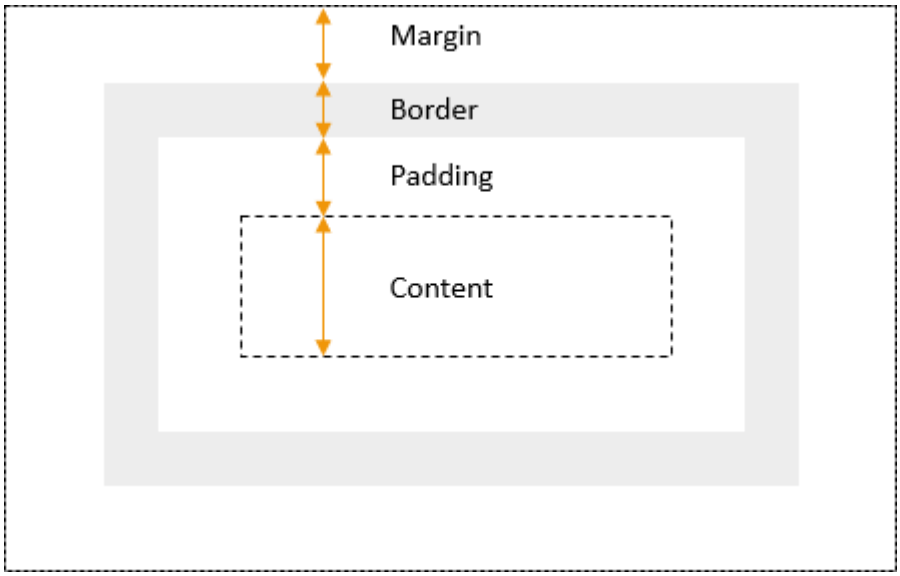
```
const classes = element.classList;Code language: JavaScript (javascript)
```

The classList is a DOMTokenList object that represents the contents of the element's class attribute.

Even though the classList is read-only, but you can manipulate the classes it contains using various methods.

Element's width & height – get the width and height of an element

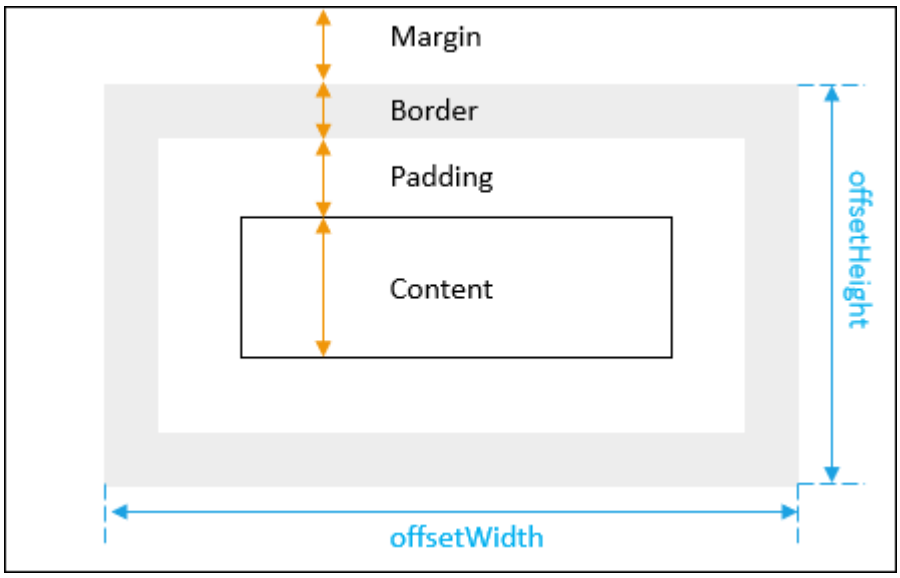
The following picture displays the CSS box model that includes a block element with content, padding, border, and margin:



To get the element's width and height that include the padding and border, you use the offsetWidth and offsetHeight properties of the element:

```
let box = document.querySelector('.box');
let width = box.offsetWidth;
let height = box.offsetHeight;
```

The following picture illustrates the offsetWidth and offsetHeight of an element:



Working with events:

JavaScript events – introduce you to the JavaScript events, the event models, and how to handle events

An event is an action that occurs in the web browser, which the web browser feedback to you so that you can respond to it.

For example, when users click a button on a webpage, you may want to respond to this click event by displaying a dialog box.

Each event may have an event handler which is a block of code that will execute when the event occurs.

An event handler is also known as an event listener. It listens to the event and executes when the event occurs.

Suppose you have a button with the id btn:

<button id="btn">Click Me!**</button>**Code language: HTML, XML (xml)

To define the code that will be executed when the button is clicked, you need to register an event handler using the addEventListener() method.

```
let btn = document.querySelector('#btn');

function display() {
```

```
    alert('It was clicked!');
}

btn.addEventListener('click',display);
```

The following table shows the most commonly-used properties and methods of the event object:

Property / Method	**Description**
bubbles	true if the event bubbles
cancelable	true if the default behavior of the event can be canceled
currentTarget	the current element on which the event is firing
defaultPrevented	return true if the preventDefault() has been called.
detail	more informatio nabout the event
eventPhase	1 for capturing phase, 2 for target, 3 for bubbling
preventDefault()	cancel the default behavior for the event. This method is only effective if the `cancelable` property is true
stopPropagation()	cancel any further event capturing or bubbling. This method only can be used if the `bubbles` property is true.
target	the target element of the event
type	the type of event that was fired

Handling Events

When an event occurs, you can create an event handler which is a piece of code that will execute to respond to that event. An event handler is also known as an event listener. It listens to the event and responds accordingly to the event fires.

An event listener is a function with an explicit name if it is resuable or an anonymous function in case it is used one time.

An event can be handled by one or multiple event handlers. If an event has multiple event handlers, all the event handlers will be executed when the event is fired.

There are three ways to assign event handlers.

1) HTML event handler attributes

Event handlers typically have names that begin with on, for example, the event handler for the click event is onclick.

To assign an event handler to an event associated with an HTML element, you can use an HTML attribute with the name of the event handler. For example, to execute some code when a button is clicked, you use the following:

```
<input type="button" value="Save" onclick="alert('Clicked!')">
```

In this case, when the button is clicked, the alert box is shown.

When you assign JavaScript code as the value of the onclick attribute, you need to escape the HTML characters such as ampersand (&), double quotes ("), less than (<), etc., or you will get a syntax error.

An event handler defined in the HTML can call a function defined in a script. For example:

```
<script>
   function showAlert() {
      alert('Clicked!');
   }
</script>
<input type="button" value="Save" onclick="showAlert()">
```

In this example, the button calls the showAlert() function when it is clicked.

The showAlert() is a function defined in a separate <script> element, and could be placed in an external JavaScript file.

Important notes

The following are some important points when you use the event handlers as attributes of the HTML element:

First, the code in the event handler can access the event object without explicitly defining it:

```
<input type="button" value="Save" onclick="alert(event.type)">
```

Second, the this value inside the event handler is equivalent to the event's target element:

```
<input type="button" value="Save" onclick="alert(this.value)">
```

Third, the event handler can access the element's properties, for example:

```
<input type="button" value="Save" onclick="alert(value)">
```

Disadvantages of using HTML event handler attributes

Assigning event handlers using HTML event handler attributes are considered as bad practices and should be avoided as much as possible because of the following reasons:

First, the event handler code is mixed with the HTML code, which will make the code more difficult to maintain and extend.

Second, it is a timing issue. If the element is loaded fully before the JavaScript code, users can start interacting with the element on the webpage which will cause an error.

For example, suppose that the following showAlert() function is defined in an external JavaScript file:

```
<input type="button" value="Save" onclick="showAlert()">
```

And when the page is loaded fully and the JavaScript has not been loaded, the showAlert() function is undefined. If users click the button at this moment, an error will occur.

2) DOM Level 0 event handlers

Each element has event handler properties such as onclick. To assign an event handler, you set the property to a function as shown in the example:

```
let btn = document.querySelector('#btn');

btn.onclick = function() {

   alert('Clicked!');

};
```

In this case, the anonymous function becomes the method of the button element. Therefore, the this value is equivalent to the element. And you can access the element's properties inside the event handler:

```
let btn = document.querySelector('#btn');

btn.onclick = function() {

   alert(this.id);

};
```

Output:

```
btn
```

By using the this value inside the event handler, you can access the element's properties and methods.

To remove the event handler, you set the value of the event handler property to null:

```
btn.onclick = null;
```

The DOM Level 0 event handlers are still being used widely because of its simplicity and cross-browser support.

3) DOM Level 2 event handlers

DOM Level 2 Event Handlers provide two main methods for dealing with the registering/deregistering event listeners:

- addEventListener() – register an event handler

- removeEventListener() – remove an event handler

These methods are available in all DOM nodes.

The addEventListener() method

The addEventListener() method accepts three arguments: an event name, an event handler function, and a Boolean value that instructs the method to call the event handler during the capture phase (true) or during the bubble phase (false). For example:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function(event) {

   alert(event.type); // click

});
```

It is possible to add multiple event handlers to handle a single event, like this:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function(event) {

   alert(event.type); // click

});



btn.addEventListener('click',function(event) {

   alert('Clicked!');

});
```

The removeEventListener() method

The removeEventListener() removes an event listener that was added via the addEventListener(). However, you need to pass the same arguments as were passed to the addEventListener(). For example:

```
let btn = document.querySelector('#btn');

// add the event listener

let showAlert = function() {

   alert('Clicked!');

};

btn.addEventListener('click', showAlert);
```

```
// remove the event listener

btn.removeEventListener('click', showAlert);
```

Using an anonymous event listener as the following will not work:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function() {

  alert('Clicked!');

});


// won't work

btn.removeEventListener('click', function() {

  alert('Clicked!');

});
```

JavaScript page load events

When you open a page, the following events occur in sequence:

- DOMContentLoaded – the browser fully loaded HTML and completed building the DOM tree. However, it hasn't loaded external resources like stylesheets and images. In this event, you can start selecting DOM nodes or initialize the interface.
- load – the browser fully loaded the HTML and also external resources like images and stylesheets.

When you leave the page, the following events fire in sequence:

- beforeunload – fires before the page and resources are unloaded. You can use this event to show a confirmation dialog to confirm if you really want to leave the page. By doing this, you can prevent data loss in case you are filling out a form and accidentally click a link to navigate to another page.
- unload – fires when the page has completely unloaded. You can use this event to send the analytic data or to clean up resources.

Handling JavaScript page load events

To handle the page events, you can call the addEventListener() method on the document object:

```
document.addEventListener('DOMContentLoaded',() => {
```

```
    // handle DOMContentLoaded event
});


document.addEventListener('load',() => {
    // handle load event
});


document.addEventListener('beforeunload',() => {
    // handle beforeunload event
});


document.addEventListener('unload',() => {
    // handle unload event
});
```

JavaScript onload

The window's load event

For the window object, the load event is fired when the whole webpage (HTML) has loaded fully, including all dependent resources, including JavaScript files, CSS files, and images.

To handle the load event, you register an event listener using the addEventListener() method:

```
window.addEventListener('load', (event) => {
    console.log('The page has fully loaded');
});
```

Or use the onload property of the window object:

```
window.onload = (event) => {
    console.log('The page has fully loaded');
};
```

If you maintain a legacy system, you may find that the load event handler is registered in of the body element of the HTML document, like this:

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

```
<head>
   <title>JS load Event Demo</title>
</head>
<body onload="console.log('Loaded!')">
</body>
</html>
```

It's a good practice to use the addEventListener() method to assign the onload event handler whenever possible.

The image's load event

The load event also fires on images. To handle the load event on images, you use the addEventListener() method of the image elements.

The following example uses the load event handler to determine if an image, which exists in the DOM tree, has been completely loaded:

```
<!DOCTYPE html>
<html>
<head>
   <title>Image load Event Demo</title>
</head>
<body>
   <img id="logo">
   <script>
      let logo = document.querySelector('#logo');

      logo.addEventListener('load', (event) => {
         console.log('Logo has been loaded!');
      });

      logo.src = "logo.png";
   </script>
</body>
</html>
```

You can assign an onload event handler directly using the onload attribute of the <img> element, like this:

```
<img id="logo"
    src="logo.png"
    onload="console.log('Logo loaded!')">
```

If you create an image element dynamically, you can assign an onload event handler before setting the src property as follows:

```
window.addEventListener('load' () => {
   let logo = document.createElement('img');
```

```
    // assign and onload event handler
    logo.addEventListener('load', (event) => {
        console.log('The logo has been loaded');
    });
    // add logo to the document
    document.body.appendChild(logo);
    logo.src = 'logo.png';
});
```

The script's load event

The <script> element also supports the load event slightly different from the standard ways. The script's load event allows you to check if a JavaScript file has been completely loaded.

Unlike images, the web browser starts downloading JavaScript files only after the src property has been assigned and the <script> element has been added to the document.

The following code loads the app.js file after the page has been completely loaded. It assigns an onload event handler to check if the app.js has been fully loaded.

```
window.addEventListener('load', checkJSLoaded)

function checkJSLoaded() {
    // create the script element
    let script = document.createElement('script');

    // assign an onload event handler
    script.addEventListener('load', (event) => {
        console.log('app.js file has been loaded');
    });

    // load the script file
    script.src = 'app.js';
    document.body.appendChild(script);
}
```

Javascript DOMContentLoaded

The DOMContentLoaded fires when the DOM content is loaded, without waiting for images and stylesheets to finish loading.

You need to handle the DOMContentLoaded event when you place the JavaScript in the head of the page but referencing elements in the body, for example:

```
<!DOCTYPE html>
<html>

<head>
    <title>JS DOMContentLoaded Event</title>
```

```
    <script>
      let btn = document.getElementById('btn');
      btn.addEventListener('click', (e) => {
        // handle the click event
        console.log('clicked');
      });
    </script>
</head>

<body>
    <button id="btn">Click Me!</button>
</body>

</html>
```

In this example, we reference the button in the body from the JavaScript in the head.

Because the DOM has not been loaded when the JavaScript engine parses the JavaScript in the head, the button with the id btn does not exist.

To fix this, you place the code inside an DOMContentLoaded event handler, like this:

```
<!DOCTYPE html>
<html>
<head>
    <title>JS DOMContentLoaded Event</title>
    <script>
      document.addEventListener('DOMContentLoaded', () => {
        let btn = document.getElementById('btn');
        btn.addEventListener('click', () => {
          // handle the click event
          console.log('clicked');
        });
      });
    </script>
</head>

<body>
    <button id="btn">Click Me!</button>
</body>
</html>
```

When you place JavaScript in the header, it will cause bottlenecks and rendering delays, so it's better to move the script before the </body> tag. In this case, you don't need to place the code in the DOMContentLoaded event, like this:

```
<!DOCTYPE html>
<html>

<head>
  <title>JS DOMContentLoaded Event</title>
</head>
<body>
  <button id="btn">Click Me!</button>
  <script>
    document.addEventListener('DOMContentLoaded', () => {
      let btn = document.getElementById('btn');
      btn.addEventListener('click', () => {
        // handle the click event
        console.log('clicked');
      });
    });
  </script>
</body>
</html>
```

JavaScript beforeunload event

Before the webpage and its resources are unloaded, the beforeunload event is fired. At this time, the webpage is still visible and you have an opportunity to cancel the event.

To register for the beforeunload event, you use the window.addEventListener() method:

```
window.addEventListener('beforeunload',(event) =>{
  // do something here
});
```

Since the window is the global object, you can omit it like this:

```
addEventListener('beforeunload',(event) =>{
  // do something here
})
```

If a webpage has a beforeunload event listener and you are about to leave the page, the beforeunload event will trigger a confirmation dialog to confirm if you really want to leave the page.

If you confirm, the browser navigates you to the new page. Otherwise, it cancels the navigation.

According to the specification, you need to call the preventDefault() method inside the beforeunload event handler in order to show the confirmation dialog. However, not all browsers implement this:

```
addEventListener('beforeunload',(event) => {
  event.preventDefault();
})
```

Historically, some browsers allow you to display a custom message on the confirmation dialog. This was intended to inform the users that they will lose data if they navigate away. Unfortunately, this feature is often used to scam users. As a result, a custom message is no longer supported.

Based on the HTML specification, the calls to alert(), confirm(), and prompt() are ignored in the beforeunload event handler.

JavaScript unload Event

The unload event fires when a document has completely unloaded. Typically, the unload event fires when you navigate from one page to another.

The unload event is fired after:

- beforeunload event
- pagehide event

At this moment, the HTML document is in the following state:

- UI is not visible to the users and is not effective.
- All the resources like img, iframe, etc., still exist.
- An error won't stop the unloading flow.

In practice, you should never use the unload event because it is not reliable on mobile devices and causes an issue with bfcache.

Handling the JavaScript unload event

To handle the unload event, you can use the addEventListener() method:

```
addEventListener('unload', (event) => {

console.log('The page is unloaded'); });
```

Or assign an event handler to the onunload property of the window object:

```
window.onunload = (event) => {

  console.log('The page is unloaded');

};
```

Or assign an event handler to the onunload attribute of the <body> element:

```
<!DOCTYPE html>
```

```
<html>
<head>
   <title>JS unload Event Demo</title>
</head>
<body onunload="console.log('The page is unloaded')">
</body>
</html>
```

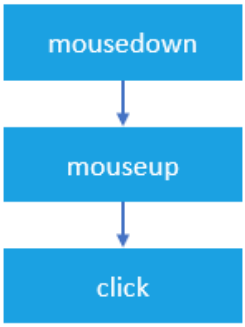
It's a good practice to use the addEventListener() to register the unload event handler.

JavaScript mouse events

Mouse events fire when you use the mouse to interact with the elements on the page. DOM Level 3 events define nine mouse events.

When you click an element, there are no less than three mouse events fire in the following sequence:

1. The mousedown fires when you depress the mouse button on the element.
2. The mouseup fires when you release the mouse button on the element.
3. The click fires when one mousedown and one mouseup detected on the element.



If you depress the mouse button on an element and move your mouse off the element, and then release the mouse button. The only mousedown event fires on the element.

Likewise, if you depress the mouse button, move the mouse over the element, and release the mouse button, the only mouseup event fires on the element.

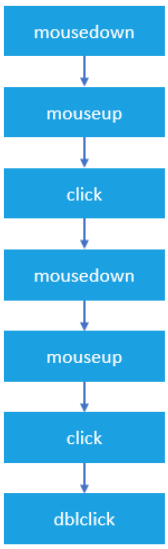
In both cases, the click event never fires.

dblclick

In practice, you rarely use the dblclick event. The dblclick event fires when you double click over an element.

It takes two click events to cause a dblclick event to fire. The dblclick event has four events fired in the following order:

1. mousedown
2. mouseup
3. click
4. mousedown
5. mouseup
6. click
7. dblclick



As you can see, the click events always take place before the dblclick event. If you register both click and dblclick event handlers on the same element, you will not know exactly what user actually has clicked or double-clicked the element.

JavaScript keyboard events

When you interact with the keyboard, the keyboard events are fired. There are three main keyboard events:

- keydown – fires when you press a key on the keyboard and fires repeatedly while you're holding down the key.
- keyup – fires when you release a key on the keyboard.
- keypress – fires when you press a character keyboard like a,b, or c, not the left arrow key, home, or end keyboard, … The keypress also fires repeatedly while you hold down the key on the keyboard.

The keyboard events typically fire on the text box, though all elements support them.When you press a character key once on the keyboard, three keyboard events are fired in the following order:

1. keydown
2. keypress
3. keyup

Both keydown and keypress events are fired before any change is made to the text box, whereas the keyup event fires after the changes have been made to the text box. If you hold down a character key, the keydown and keypress are fired repeatedly until you release the key.

When you press a non-character key, the keydown event is fired first followed by the keyup event. If you hold down the non-character key, the keydown is fired repeatedly until you release the key.

JavaScript Scroll Events

When you scroll a document or an element, the scroll events fire. You can trigger the scroll events in the following ways, for example:

- Using the scrollbar manually
- Using the mouse wheel
- Clicking an ID link
- Calling functions in JavaScript

To register a scroll event handler, you call the addEventListener() method on the target element, like this:

```
targetElement.addEventListener('scroll', (event) => {
  // handle the scroll event
});
```

or assign an event handler to the onscroll property of the target element:

```
targetElement.onscroll = (event) => {
  // handle the scroll event
};
```

JavaScript scrollIntoView

Suppose you have a list of elements and you want a specific element to be highlighted and scrolled into view.

To achieve this, you can use the element.scrollIntoView() method. The element.scrollIntoView() accepts a boolean value or an object:

```
element.scrollIntoView(alignToTop);
```

or

```
element.scrollIntoView(options);
```

The method accepts one of the following two arguments:

alignToTop

The alignToTop is a boolean value.

If it is set to true, the method will align the top of the element to the top of the viewport or the top of the visible area of the scrollable ancestor.

If the alignToTop is set to false, the method will align the bottom of the element to the bottom of the viewport or the bottom of the visible area of the scrollable ancestor.

By default, the alignToTop is true.

options

The options argument is an object that gives more control over of alignment of the element in the view. However, the web browser support may be slightly different.

The options object has the following properties:

- behavior property defines the transition animation. The behavior property accepts two values: auto or smooth. It defaults to auto.
- block property defines the vertical alignment. It accepts one of four values: start, center, endor nearest. By default, it is start.
- inline property defines horizontal alignment. It also accepts one of four values: start, center, endor nearest. It defaults to nearest.

JavaScript focus events

The focus events fire when an element receives or loses focus. These are the two main focus events:

- focus fires when an element has received focus.
- blur fires when an element has lost focus.

The focusin and focusout fire at the same time as focus and blur, however, they bubble while the focus and blur do not.

The following elements are focusable:

- The **window** gains focus when you bring it forward by using Alt+Tab or clicking on it and loses focus when you send it back.
- **Links** when you use a mouse or a keyboard.
- **Form fields** like input text when you use a keyboard or a mouse.
- Elements with **tabindex**, also when you use a keyboard or a mouse.

The JavaScript hashchange event

The URL hash is everything that follows the pound sign (#) in the URL. For example, suppose that you have the following URL:

https://www.javascripttutorial.net/javascript-dom/javascript-hashchange/#headerCode

The URL hash is header. If the URL hash changes to footer, like this:

https://www.javascripttutorial.net/javascript-dom/javascript-hashchange/#footerCode

The hashchange event fires when the URL hash changes from one to another. In this example, it changes from #header to #footer.

To attach an event listener to the hashchange event, you call the addEventListener() method on the window object:

```
window.addEventListener('hashchange',() =>{
   console.log('The URL has has changed');
});
```

To get the current URL hash, you access the hash property of the location object:

```
window.addEventListener('hashchange',() => {
   console.log(`The current URL hash is ${location.hash}`);
});
```

Additionally, you can handle the hashchange event by assigning an event listener to the onhashchange property of the window object:

```
window.onhashchange = () => {
   // handle hashchange event here
};
```

JavaScript Event Delegation

Suppose that you have the following menu:

```
<ul id="menu">
   <li><a id="home">home</a></li>
   <li><a id="dashboard">Dashboard</a></li>
   <li><a id="report">report</a></li>
</ul>
```

To handle the click event of each menu item, you may add the corresponding click event handlers:

```
let home = document.querySelector('#home');
home.addEventListener('home',(event) => {
   console.log('Home menu item was clicked');
});
let dashboard = document.querySelector('#dashboard');
dashboard.addEventListener('dashboard',(event) => {
   console.log('Dashboard menu item was clicked');
});
let report = document.querySelector('#report');
report.addEventListener('report',(event) => {
   console.log('Report menu item was clicked');
});
```

In JavaScript, if you have a large number of event handlers on a page, these event handlers will directly impact the performance because of the following reasons:

- First, each event handler is a function which is also an object that takes up memory. The more objects in the memory, the slower the performance.
- Second, it takes time to assign all the event handlers, which causes a delay in the interactivity of the page.

To solve this issue, you can leverage the event bubbling.

Instead of having multiple event handlers, you can assign a single event handler to handle all the click events:

```
let menu = document.querySelector('#menu');

menu.addEventListener('click', (event) => {

  let target = event.target;


  switch(target.id) {

    case 'home':

      console.log('Home menu item was clicked');

      break;

    case 'dashboard':

      console.log('Dashboard menu item was clicked');

      break;

    case 'report':

      console.log('Report menu item was clicked');

      break;

  }

});
```

How it works.

- When you click any <a> element inside the <ul> element with the id menu, the click event bubbles to the parent element which is the <ul> element. So instead of handling the click event of the individual <a> element, you can capture the click event at the parent element.

- In the click event listener, you can access the target property which references the element that dispatches the event. To get the id of the element that the event actually fires, you use the target.id property.
- Once having the id of the element that fires the click event, you can have code that handles the event correspondingly.

The way that we handle the too-many-event-handlers problem is called the event delegation.

The event delegation refers to the technique of levering event bubbling to handle events at a higher level in the DOM than the element on which the event originated.

JavaScript event delegation benefits

When it is possible, you can have a single event handler on the document that will handle all the events of a particular type. By doing this, you gain the following benefits:

- Less memory usage, better performance.
- The document object is available immediately. As long as the element is rendered, it can start functioning correctly without delay. You don't need to wait for the DOMContentLoaded or load events.

JavaScript custom events

Creating JavaScript custom events

To create a custom event, you use the CustomEvent() constructor:

```
let event = new CustomEvent(eventType, options);
```

The CustomEvent() has two parameters:

1. The eventType is a string that represents the name of the event.
2. The options is an object has the detail property that contains any custom information about the event.

The following example shows how to create a new custom event called highlight:

```
let event = new CustomEvent('highlight', {
  detail: {backgroundColor: 'yellow'}
});
```

Dispatching JavaScript custom events

After creating a custom event, you need to attach the event to a DOM element and trigger it by using the dispatchEvent() method:

```
domElement.dispatchEvent(event);
```

Conclusion:

In summary, the Document Object Model (DOM) and JavaScript are fundamental components of web development. The DOM, structured as a hierarchical tree, provides a structured representation of web documents, allowing developers to access and manipulate web page content, structure, and style dynamically. JavaScript serves as the language for interacting with the DOM, making web pages dynamic and interactive.

Understanding the relationship between HTML attributes and DOM properties is crucial for effective web development. While attributes define initial values in HTML, properties enable dynamic interaction and manipulation through JavaScript. These concepts are central to creating modern, responsive, and user-friendly web applications.

By following best practices and optimizing JavaScript and DOM code, web developers can ensure the efficiency and maintainability of their projects. The ongoing relevance of JavaScript and the DOM in modern web development highlights their importance in creating dynamic and interactive user experiences on the web.