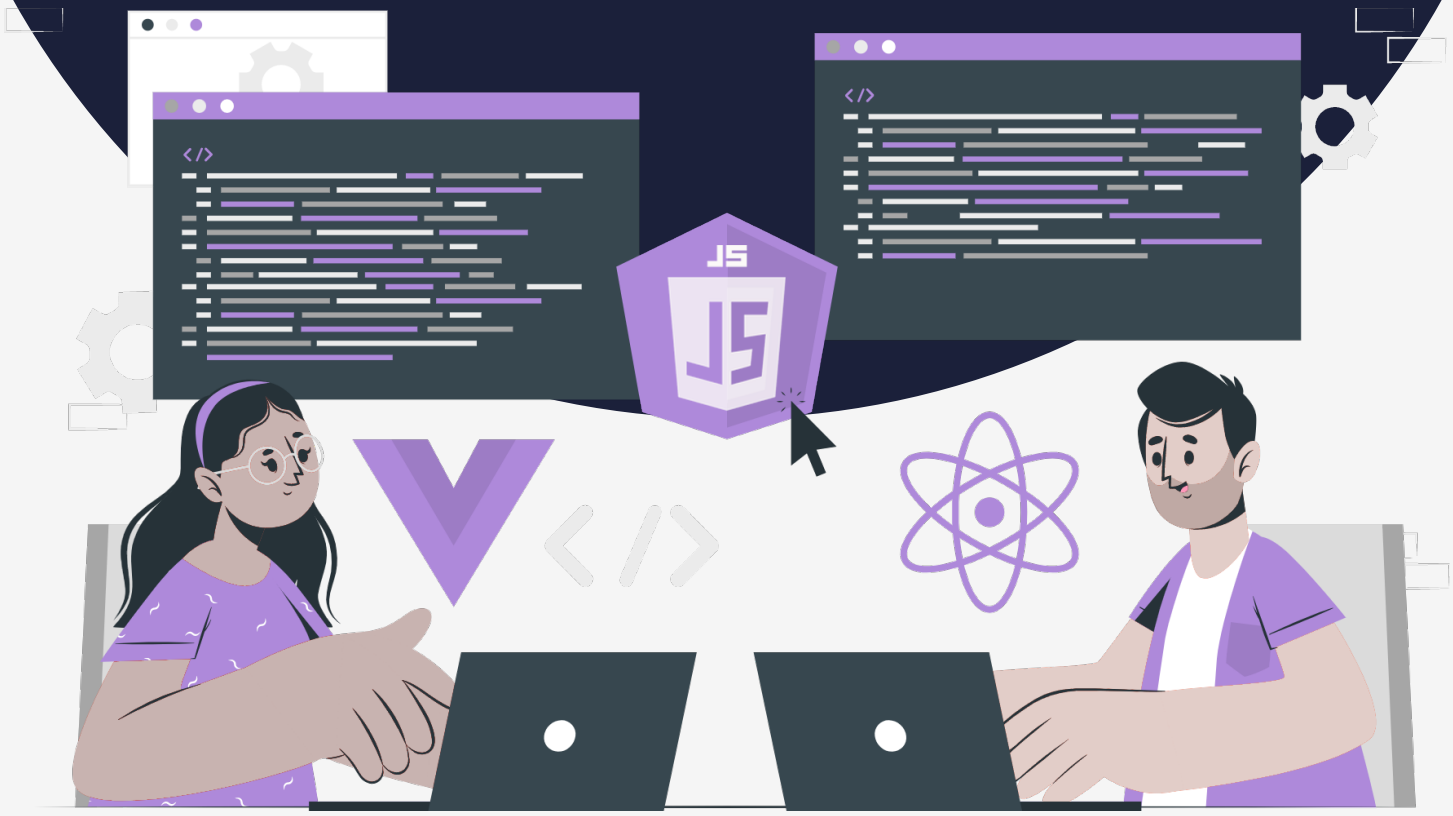
Lesson:

JavaScript DOM



PW SKILLS

DOM and its Working

Topics Covered:

- What is DOM?
- How DOM works?

What is DOM?
The Document Object Model (DOM) is a programming interface for web documents. It represents the web page as a tree-like structure of objects, where each object corresponds to an element or attribute in the web page. The DOM provides a way for JavaScript to interact with and manipulate the elements and content of a web page.

Here's an example of how the DOM works:

Consider the following HTML code:

```
<!DOCTYPE html>
<html>
<head>
 <title>My Web Page</title>
</head>
<body>
 <h1>Welcome to my web page!</h1>
 <p>This is a paragraph of text.</p>
 <ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
 </ul>
</body>
</html>
```

When this code is loaded in a web browser, the browser creates a DOM tree that represents the structure and content of the web page, as shown below:

```
- Document
  - html
   - head
     - title: "My Web Page"
   - body
     - h1: "Welcome to my web page!"
     - p: "This is a paragraph of text."
     - ul
       - li: "Item 1"
       - li: "Item 2"
       - li: "Item 3"
```

Each element in the web page corresponds to a node in the DOM tree, and each attribute corresponds to a property of the corresponding node.

How DOM works?

The Document Object Model (DOM) is a programming interface for web documents that is used by JavaScript to interact with and manipulate the elements and content of a web page. Here's a detailed explanation of how the DOM works in JavaScript:

When a web page is loaded in a web browser, the browser creates a DOM tree that represents the structure and content of the web page. The DOM tree is a hierarchical data structure that is composed of nodes, where each node corresponds to an element, attribute, or text content in the web page. The root of the tree is the "document" object, which represents the entire web page.

Each node in the DOM tree is an object that has properties and methods that can be accessed and manipulated by JavaScript. For example, the "document" object has properties and methods that allow JavaScript to access and manipulate the elements and content of the web page.

To interact with the DOM in JavaScript, you can use various methods and properties that are provided by the DOM API. For example, to select an element in the DOM, you can use the "getElementById()", "querySelector()", or "getElementsByTagName()" methods. These methods return a reference to the selected element as a DOM node, which can then be used to access or modify the properties of the element.

Once you have a reference to a DOM node, you can use its properties and methods to access or modify its content, style, or other attributes. For example, you can use the "innerHTML" property to get or set the HTML content of an element, the "style" property to access or modify the CSS style of an element, or the "setAttribute()" method to set the value of an attribute on an element.

When you modify the properties or content of a DOM node in JavaScript, the corresponding element on the web page is updated in real-time, without the need to reload the web page. This enables dynamic web development, where web pages can respond to user interactions, events, or data changes in real time, without requiring a server-side refresh.

Overall, the DOM provides a powerful and flexible programming interface for web documents that is used by JavaScript to interact with and manipulate the elements and content of a web page and is a key component of modern web development.

Visualize of DOM

Topics Covered:

- Visualization of DOM
- step-by-step guide to visualizing the DOM using the Chrome browser

Visualization of DOM

The Document Object Model (DOM) in JavaScript represents the HTML or XML document as a tree-like structure of objects. To visualize the DOM in JavaScript, you can use the developer tools in your web browser.

In most web browsers, you can open the developer tools by pressing the F12 key or right-clicking on the web page and selecting "Inspect" or "Inspect Element". Once the developer tools are open, you can navigate to the "Elements" or "DOM" tab to view the DOM tree.

The DOM tree is displayed as a hierarchical structure of nodes, with each node representing an HTML or XML element, attribute, or text content. You can expand or collapse nodes to view their children or siblings, and you can hover over a node to highlight the corresponding element on the web page.

You can also use the developer tools to modify the DOM in real-time. For example, you can add, delete, or modify nodes, attributes, and text content by clicking on the relevant element and using the editor that appears.

In addition, the developer tools allow you to view the CSS styles that are applied to each element, as well as to modify the styles and see the effects on the web page in real-time.

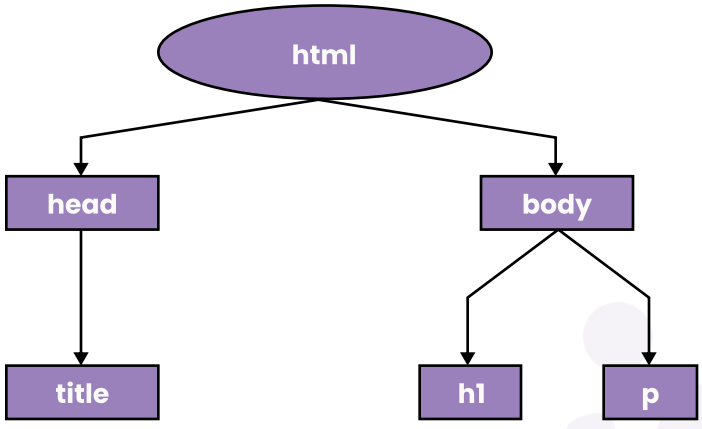
Overall, visualizing the DOM in JavaScript involves using the web browser's developer tools to view and manipulate the tree-like structure of objects that represent the HTML or XML document in the browser. This can be a powerful tool for debugging and understanding the structure and content of a web page.

For example, consider the following HTML code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <h1>Welcome to my page</h1>
    <p>This is some text on my page</p>
  </body>
</html>
```

The DOM representation of this HTML code would be a tree structure that looks like this:

```
- Document
  - html
    - head
      - title
        - "My Web Page"
    - body
      - h1
        - "Welcome to my page"
      - p
        - "This is some text on my page"
```



step-by-step guide to visualizing the DOM using the Chrome browser

To visualize the Document Object Model (DOM) in JavaScript, you can use the developer tools in your web browser. Here's a step-by-step guide to visualizing the DOM using the Chrome browser:

- Open the Chrome browser and navigate to the web page you want to inspect.
- Press the F12 key to open the developer tools.
- In the developer tools, select the "Elements" tab to view the DOM tree.
- You can expand or collapse nodes to view their children or siblings.
- To select an element in the DOM, click on it in the "Elements" tab or on the web page itself. The corresponding element in the DOM tree will be highlighted.
- You can also modify the DOM in real-time using the developer tools. To add, delete, or modify an element or attribute, right-click on the relevant node and select "Edit as HTML" or "Edit attribute", then make your changes in the editor that appears.
- To view or modify the CSS styles that are applied to each element, select the "Styles" tab in the developer tools. You can view and modify the styles and see the effects on the web page in real-time.

Other web browsers, such as Firefox and Safari, have similar developer tools for visualizing and manipulating the DOM. By visualizing the DOM in JavaScript, you can better understand the structure and content of a web page, and use this knowledge to write more effective JavaScript code.

How we target DOM

Topics Covered:

- getElementById()
- getElementsByClassName()
- getElementsByTagName()
- querySelector()
- querySelectorAll()

getElementById()

The **getElementById()** method is a commonly used method in the Document Object Model (DOM) of web pages. It is a JavaScript method that allows you to retrieve a reference to an HTML element on a web page using its unique ID. This method returns a reference to the first element with the specified ID that is found in the document.

Here's a detailed explanation of how to use the **getElementById()** method in JavaScript:

Identify the ID of the HTML element you want to retrieve.
Before using the **getElementById()** method, you need to know the ID of the HTML element you want to retrieve. The ID should be unique within the HTML document.

Use the **getElementById()** method to retrieve the HTML element.
The **getElementById()** method is called on the document object, which represents the entire HTML document. The method takes one argument, which is a string representing the ID of the element you want to retrieve. For example, the following code retrieves the HTML element with an ID of "myElement":

```
var element = document.getElementById("myElement");
```

The **getElementById()** method returns a reference to the HTML element with the specified ID. This reference can be used to access the properties and methods of the element.

getElementsByClassName()

The **getElementsByClassName()** method is a built-in method in JavaScript that is used to retrieve a list of all the elements that have a specific class name. This method is used to select one or more elements on a web page based on the value of their class attribute. The method returns an HTMLCollection, which is an array-like object containing all the elements that match the specified class name.

Here's a detailed explanation of how to use the **getElementsByClassName()** method in JavaScript:

Identify the class name of the HTML elements you want to retrieve.
Before using the **getElementsByClassName()** method, you need to know the class name of the HTML elements you want to retrieve. The class name should be unique enough to select only the elements you want to retrieve.
Use the **getElementsByClassName()** method to retrieve the HTML elements.

The **getElementsByClassName()** method is called on the document object, which represents the entire HTML document. The method takes one argument, which is a string representing the class name of the elements you want to retrieve. For example, the following code retrieves all the HTML elements with a class name of "myClass":

```
var elements = document.getElementsByClassName("myClass");
```

The **getElementsByClassName()** method returns an HTMLCollection, which is an array-like object that contains all the HTML elements that match the specified class name. This collection can be accessed using a numeric index or using a loop.

In summary, the **getElementsByClassName()** method is a JavaScript method used to retrieve a list of HTML elements by their class name. This method is commonly used to manipulate the content and attributes of HTML elements on a web page.

getElementsByTagName()

The **getElementsByTagName()** method is a built-in method in JavaScript that is used to retrieve a list of all the elements with a specific tag name. This method is used to select one or more elements on a web page based on the value of their HTML tag. The method returns an HTMLCollection, which is an array-like object containing all the elements that match the specified tag name.

Here's a detailed explanation of how to use the **getElementsByTagName()** method in JavaScript:

Identify the tag name of the HTML elements you want to retrieve.
Before using the **getElementsByTagName()** method, you need to know the tag name of the HTML elements you want to retrieve. The tag name should be unique enough to select only the elements you want to retrieve.

Use the **getElementsByTagName()** method to retrieve the HTML elements.
The **getElementsByTagName()** method is called on the document object, which represents the entire HTML document. The method takes one argument, which is a string representing the tag name of the elements you want to retrieve. For example, the following code retrieves all the HTML elements with a tag name of "p":

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

The **getElementsByTagName()** method returns an HTMLCollection, which is an array-like object that contains all the HTML elements that match the specified tag name. This collection can be accessed using a numeric index or using a loop.

In summary, the **getElementsByTagName()** method is a JavaScript method used to retrieve a list of HTML elements by their tag name. This method is commonly used to manipulate the content and attributes of HTML elements on a web page.

querySelector()

The **querySelector()** method is a built-in method in JavaScript that allows you to select the first element that matches a specified CSS selector. It is commonly used to retrieve and manipulate HTML elements on a web page.

The method is called on the document object, which represents the entire HTML document, and takes one argument, which is a string representing the CSS selector of the element you want to retrieve. The CSS selector can be any valid CSS selector, including class names, IDs, element types, attribute selectors, and more.

Here are some examples of CSS selectors that can be used with the querySelector() method:

- **Retrieving an element by ID:**
  #myId: Selects the element with the ID of "**myId**"

```
var element = document.querySelector("#myId");
```

- **Retrieving an element by class name:**
  .myClass: Selects all elements with the class name of "**myClass**"

```
var element = document.querySelector(".myClass");
```

- **Retrieving an element by tag name:**
  p: Selects all paragraph elements

```
var element = document.querySelector("p");
```

This code retrieves the first paragraph element on the page.

- **Retrieving an element using a complex selector:**

```
var element = document.querySelector("ul li:nth-child(2)");
```

This code retrieves the second li element within a ul element on the page.

- **Retrieving an element with an attribute selector:**

```
var element = document.querySelector("a[href='#']");
```

This code retrieves the first anchor element on the page with an href attribute equal to #.

- **Retrieving a child element within a parent element:**

```
var element = document.querySelector("#myParent .myChild");
```

This code retrieves the first child element with a class name of "myChild" within an element with an ID of "myParent".

These are just a few examples of how the querySelector() method can be used to retrieve elements on a web page. The method is very versatile and can be used with a wide range of CSS selectors to target specific elements.

querySelectorAll()

querySelectorAll() is a method available in the Document Object Model (DOM) API that allows you to retrieve a list of all elements in the document that match a specified CSS selector. This method returns a NodeList object, which is similar to an array, that contains all of the matching elements in the order in which they appear in the document.

The querySelectorAll() method takes a single parameter, which is a string representing the CSS selector to use when selecting elements. The CSS selector syntax is used to specify the criteria for selecting elements based on their attributes, classes, or other properties. The syntax for using querySelectorAll() is as follows:

```
document.querySelectorAll(selector)
```

The selector parameter is a string that represents the CSS selector to be used when selecting elements. It can be any valid CSS selector, including class selectors, ID selectors, attribute selectors, and more. Here are some examples of valid CSS selectors:

```
/* Select all elements with a class of "my-class" */
.my-class

/* Select the element with an ID of "my-id" */
#my-id

/* Select all elements with a "data-type" attribute */
[data-type]

/* Select all "a" elements with a "href" attribute that contains "example.com" */
a[href*="example.com"]
```

querySelectorAll() returns a NodeList object that contains all of the elements that match the specified selector. The NodeList is similar to an array, but it is a static list that does not update automatically as the document changes. You can access individual elements in the NodeList using the item() method or by using array-style bracket notation. For example, to get the first element in the NodeList, you can use:

```
const firstElement = document.querySelectorAll(selector)[0];
```

You can also loop through the NodeList using a for loop or using array methods such as forEach(), map(), or filter(). For example, to loop through all of the elements that match a selector and log their text content to the console, you could use the following code:

```
const elements = document.querySelectorAll(selector);

for (let i = 0; i < elements.length; i++) {
  console.log(elements[i].textContent);
}
```

Alternatively, you could use the forEach() method to achieve the same result:

```
document.querySelectorAll(selector).forEach((element) => {
  console.log(element.textContent);
});
```

In summary, querySelectorAll() is a powerful method in the DOM API that allows you to select and manipulate elements in the document based on their attributes, classes, or other properties.

Methods of DOM – Part 1

Topics Covered:

- createElement()
- appendChild()
- prepend()
- removeChild()

createElement()

createElement() is a method in the Document Object Model (DOM) API that allows you to create a new HTML element. You can then manipulate the element and add it to the document using other DOM methods.

The syntax for createElement() is:

```
// syntax
document.createElement('tagname')
```

The **createElement()** method takes a single parameter, which is a string that represents the name of the element to be created. For example, to create a new div element, you would use the following code:

```
const newDiv = document.createElement('div');
```

This creates a new div element, which is not yet part of the document. You can then add attributes and content to the element using other DOM methods.

appendChild()

In the Document Object Model (DOM) of a web page, every element is an object that can have child elements. The **appendChild()** method is a JavaScript DOM method that allows you to add a new child element to an existing parent element. The new child element is added to the end of the list of existing children for the parent element.

The syntax for using the appendChild() method is:

```
parentElement.appendChild(childElement);
```

In this syntax, **parentElement** is a reference to the parent element to which the new child element will be added, and **childElement** is the new child element that will be added to the end of the list of existing children for the parent element.

Here is an example that demonstrates how to use the **appendChild()** method to add a new div element to an existing div element:

```
const parentDiv = document.querySelector('#parent-div');
const newDiv = document.createElement('div');
parentDiv.appendChild(newDiv);
```

In this example, we first use the **querySelector()** method to select an existing div element with an ID of parent-div. We then create a new div element using the **createElement()** then we use the **appendChild()** method to add the new div element to the end of the **parentDiv** element.

The resulting HTML would look something like this:

```
<div id="parent-div">
  ...existing child elements...
  <div></div>
</div>
```

As you can see, the new div element is added at the end of the parentDiv element, after any existing child elements.

Note that the **appendChild()** method is supported in most modern web browsers, but may not be supported in some older browsers. If you need to support older browsers, you can use a polyfill or a similar workaround to achieve the same functionality.

prepend()

The **prepend()** method in JavaScript is a DOM method that allows you to add a new child element to the beginning of an existing parent element. This method is similar to the **appendChild()** method, but instead of adding the new child element to the end of the list of existing children for the parent element, it adds the new child element to the beginning of the list.

The syntax for using the **prepend()** method is :

```
parentElement.prepend(childElement);
```

In this syntax, parentElement is a reference to the parent element to which the new child element will be added, and childElement is the new child element that will be added to the beginning of the list of existing children for the parent element.

Here is an example that demonstrates how to use the **prepend()** method to add a new div element to an existing div element:

```
const parentDiv = document.querySelector('#parent-div');
const newDiv = document.createElement('div');
parentDiv.prepend(newDiv);
```

In this example, we first use the querySelector() method to select an existing div element with an ID of parent-div. We then create a new div element using the createElement() method and then we use the **prepend()** method to add the new div element to the beginning of the parentDiv element.

The resulting HTML would look something like this:

```
<div id="parent-div">
  <div></div>
  ...existing child elements...
</div>
```

As you can see, the new div element is added at the beginning of the parentDiv element, before any existing child elements.

Note that the **prepend()** method is supported in most modern web browsers, but may not be supported in some older browsers. If you need to support older browsers, you can use a polyfill or a similar workaround to achieve the same functionality.

removeChild()

The **removeChild()** method in JavaScript is a DOM method that allows you to remove an existing child element from its parent element. This method can be used to remove any child element of a parent, whether it was added dynamically using JavaScript or was part of the original HTML content of the page.

The syntax for using the **removeChild()** method is:

```
parentElement.removeChild(childElement);
```

In this syntax, **parentElement** is a reference to the parent element from which the child element will be removed, and **childElement** is the child element that will be removed.

Here is an example that demonstrates how to use the **removeChild()** method to remove a div element from an existing div element:

```
const parentDiv = document.querySelector('#parent-div');
const childDiv = document.querySelector('#child-div');
parentDiv.removeChild(childDiv);
```

In this example, we first use the **querySelector()** method to select the parent div element with an ID of parent-div, and the child div element with an ID of child-div. We then use the **removeChild()** method to remove the **childDiv** element from the **parentDiv** element.

The resulting HTML would look something like this:

```
<div id="parent-div">
  ...existing child elements except the one that was removed...
</div>
```

As you can see, the **childDiv** element is removed from the **parentDiv** element, and the resulting HTML only contains the remaining child elements of the **parentDiv** element.

Note that the **removeChild()** method is supported in most modern web browsers, but may not be supported in some older browsers. If you need to support older browsers, you can use a polyfill or a similar workaround to achieve the same functionality.

Methods of DOM – Part 2

Topics Covered:

- innerHTML
- innerText
- classList
- style

innerHTML

The **innerHTML** property in JavaScript is a DOM property that allows you to get or set the HTML content of an element. This property can be used to get the inner HTML of an element, which includes all the child elements and their contents or to set the inner HTML of an element, which can be used to add or remove child elements and their contents dynamically.

The syntax for using the **innerHTML** property to get the HTML content of an element is

```
const elementHTML = element.innerHTML;
```

In this syntax, **element** is a reference to the element whose HTML content you want to retrieve, and **elementHTML** is a string that contains the HTML content of the element, including all its child elements.

Here is an example that demonstrates how to use the **innerHTML** property to retrieve the HTML content of a **div** element:

```
<div id="my-div">
  <p>Some text inside the div</p>
  <ul>
    <li>List item 1</li>
    <li>List item 2</li>
    <li>List item 3</li>
  </ul>
</div>

<script>
const myDiv = document.querySelector('#my-div');
const myDivHTML = myDiv.innerHTML;
console.log(myDivHTML);
</script>
```

In this example, we use the **querySelector()** method to select an existing div element with an ID of **my-div**. We then use the **innerHTML** property to retrieve the HTML content of the **myDiv** element and log it to the console. The resulting output would be:

```
<p>Some text inside the div</p>
<ul>
  <li>List item 1</li>
  <li>List item 2</li>
  <li>List item 3</li>
</ul>
```

As you can see, the **innerHTML** property returns the full HTML content of the myDiv element, including all its child elements.

The syntax for using the **innerHTML** property to set the HTML content of an element is:

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

In this syntax, **element** is a reference to the element whose HTML content you want to set, and **newHTML** is a string that contains the new HTML content that you want to set for the element.

Here is an example that demonstrates how to use the **innerHTML** property to set the HTML content of a **div** element:

```
<div id="my-div">
  <p>Some text inside the div</p>
</div>

<script>
const myDiv = document.querySelector('#my-div');
const newHTML = '<p>New text inside the div</p>'
myDiv.innerHTML = newHTML;
</script>
```

In this example, we use the **querySelector()** method to select an existing **div** element with an ID of **my-div**. We then use the **innerHTML** property to set the new HTML content of the **myDiv** element to ** New text inside the div **. The resulting HTML would look like this:

```
<div id="my-div">
  <p>New text inside the div</p>
</div>
```

As you can see, the **innerHTML** property replaces the existing HTML content of the myDiv element with the new HTML content specified in the **newHTML** variable.

Note that the **innerHTML** property can be used to add or remove child elements and their contents dynamically, but it can also be a security risk if you are not careful with the content that you insert into the page. You should never use the **innerHTML** property to insert untrusted content into the page, as this can open up security vulnerabilities such as cross-site scripting.

innerText

The **innerText** property is a DOM property in JavaScript that allows you to get or set the text content of an element, without including any HTML markup that may be present. This property can be used to get the text content of an element, which includes all the text nodes that are direct children of the element, or to set the text content of an element, which can be used to update the text of an element dynamically.

The syntax for using the **innerText** property to get the text content of an element is:

```
const elementText = element.innerText;
```

In this syntax, **element** is a reference to the element whose text content you want to retrieve, and **elementText** is a string that contains the text content of the element, without any HTML markup.

Here is an example that demonstrates how to use the **innerText** property to retrieve the text content of a **div** element:

```
<div id="my-div">
  Some text inside the div
  <p>Some more text inside the div</p>
</div>

<script>
const myDiv = document.querySelector('#my-div');
const myDivText = myDiv.innerText;
console.log(myDivText);
</script>
```

In this example, we use the **querySelector()** method to select an existing **div** element with an ID of **my-div**. We then use the **innerText** property to retrieve the text content of the **myDiv** element and log it to the console. The resulting output would be:

```
Some text inside the div
Some more text inside the div
```

As you can see, the **innerText** property returns the text content of the **myDiv** element, without any HTML markup that may be present.

The syntax for using the **innerText** property to set the text content of an element is:

```
element.innerText = newText;
```

In this syntax, **element** is a reference to the element whose text content you want to set, and newText is a string that contains the **new text** content that you want to set for the element.

Here is an example that demonstrates how to use the **innerText** property to set the text content of a **div** element:

```
<div id="my-div">
  Some text inside the div
  <p>Some more text inside the div</p>
</div>

<script>
const myDiv = document.querySelector('#my-div');
myDiv.innerText = 'New text inside the div';
</script>
```

In this example, we use the **querySelector()** method to select an existing div element with an ID of **my-div**. We then use the **innerText** property to set the new text content of the **myDiv** element to **New text inside the div**. The resulting HTML would look like this:

```
<div id="my-div">
  New text inside the div
  <p>Some more text inside the div</p>
</div>
```

As you can see, the **innerText** property replaces the existing text content of the myDiv element with the new text content specified in the newText variable, without affecting any of the child elements of the element.

Note that the **innerText** property is similar to the **textContent** property, which also allows you to get or set the text content of an element. However, the textContent property includes all text nodes, including whitespace, while the innerText property removes leading and trailing whitespace and collapses consecutive whitespace characters. Additionally, the **innerText** property is not supported in older versions of Internet Explorer, while the **textContent** property is.

textContent

The **textContent** property is a DOM property in JavaScript that allows you to get or set the text content of an element, including all text nodes that are direct and indirect children of the element. Unlike the **innerText** property, it returns all the text nodes, including whitespace, and doesn't collapse whitespace characters.

The syntax for using the **textContent** property to get the text content of an element is:

```
const elementText = element.textContent;
```

In this syntax, **element** is a reference to the element whose text content you want to retrieve, and **elementText** is a string that contains the text content of the element, including any whitespace and text nodes.

Here is an example that demonstrates how to use the **textContent** property to retrieve the text content of a **div** element:

```
<div id="my-div">
  Some text inside the div
  <p>Some more text inside the div</p>
</div>

<script>
const myDiv = document.querySelector('#my-div');
const myDivText = myDiv.textContent;
console.log(myDivText);
</script>
```

In this example, we use the **querySelector()** method to select an existing **div** element with an ID of **my-div**. We then use the **textContent** property to retrieve the text content of the **myDiv** element and log it to the console. The resulting output would be:

```
Some text inside the div
Some more text inside the div
```

As you can see, the **textContent** property returns the text content of the **myDiv** element, including any whitespace and text nodes.

The syntax for using the **textContent** property to set the text content of an element is:

```
element.textContent = newText;
```

In this syntax, **element** is a reference to the element whose text content you want to set, and **newText** is a string that contains the new text content that you want to set for the element.

Here is an example that demonstrates how to use the **textContent** property to set the text content of a **div** element:

```
<div id="my-div">
  Some text inside the div
  <p>Some more text inside the div</p>
</div>

<script>
const myDiv = document.querySelector('#my-div');
myDiv.textContent = 'New text inside the div';
</script>
```

In this example, we use the **querySelector()** method to select an existing **div** element with an ID of **my-div.** We then use the **textContent** property to set the new text content of the **myDiv** element to **New text inside the div**. The resulting HTML would look like this:

```
<div id="my-div">
  New text inside the div
  <p>Some more text inside the div</p>
</div>
```

As you can see, the **textContent** property replaces the existing text content of the **myDiv** element with the new text content specified in the **newText** variable, including any whitespace and text nodes.

In summary, the textContent property returns all the text nodes of an element, including any whitespace, and allows you to set the text content of an element with the new text content including any whitespace. It is useful when you need to get or set the entire text content of an element including whitespace and text nodes.

classList

The **classList** property is a DOM property in JavaScript that allows you to add, remove, toggle, and check for the presence of classes on an element's list of classes. It returns a **DOMTokenList** object that represents the class attribute of an element as a space-separated list of strings.

The syntax for accessing the classList property is:

```
const elementClasses = element.classList;
```

In this syntax, **element** is a reference to the element whose class list you want to retrieve, and **elementClasses** is a **DOMTokenList** object representing the element's list of classes.

Here is an example that demonstrates how to use the **classList** property to retrieve the class list of a **div** element:

```
<div id="my-div" class="red border">Some content here</div>

const myDiv = document.querySelector('#my-div');
const myDivClasses = myDiv.classList;
console.log(myDivClasses); // Output: DOMTokenList ["red", "border"]
```

In this example, we use the **querySelector()** method to select an existing **div** element with an ID of **my-div**. We then use the classList property to retrieve the list of classes assigned to the **myDiv** element and log it to the console. The resulting output would be a **DOMTokenList** object with the classes **"red"** and **"border"**.

You can perform several operations on the classList object to add, remove, toggle or check for the presence of classes:

- **add():** Adds one or more classes to the class list of an element.

```
element.classList.add(class1, class2, ...);
```

In this syntax, class1, class2, and so on are strings representing the names of the classes you want to add to the element's class list. If the class already exists, it won't be added again.

- **remove():** Removes one or more classes from the class list of an element.

```
element.classList.remove(class1, class2, ...);
```

In this syntax, class1, class2, and so on are strings representing the names of the classes you want to remove from the element's class list. If the class doesn't exist, nothing happens.

- **toggle():** Toggles the presence of a class in the class list of an element.

```
element.classList.toggle(class, force);
```

In this syntax, class is a string representing the name of the class you want to toggle. If the class exists in the class list, it will be removed, and if it doesn't exist, it will be added. You can also use the optional force parameter to force the class to be added or removed, depending on whether it's true or false.

- **contains():** Checks if an element has a specified class in its class list.

```
const hasClass = element.classList.contains(class);
```

In this syntax, **class** is a string representing the name of the class you want to check for. The **contains()** method returns a boolean value indicating whether the class is present in the element's class list.

style

In the Document Object Model (DOM), you can manipulate an element's style by accessing its **style** property. The **style** property is an object that represents the inline styles of an element, which are defined using the **style** attribute in the element's HTML.

Here are some of the ways you can manipulate an element's style using the style property:

- **Set a single style property:**

You can set a single style property using the style property's property notation. For example, to set the color property of an element to red, you can use the following code:

```
const myElement = document.querySelector('.my-element');
myElement.style.color = 'red';
```

- **Set multiple style properties:**

You can set multiple style properties using the style property's property notation. Separate each property assignment with a semicolon. For example, to set the color, background-color, and border properties of an element, you can use the following code:

```
const myElement = document.querySelector('.my-element');
myElement.style.cssText = 'color: red; background-color: yellow; border: 1px
solid black';
```

- **Get a style property:**

You can get a style property by accessing the style property's property notation. For example, to get the color property of an element, you can use the following code:

```
const myElement = document.querySelector('.my-element');
const color = myElement.style.color;
console.log(color); // "red"
```

Note that this method only retrieves the inline styles of an element, not any styles applied to it via CSS.

- **Remove a style property:**

You can remove a style property by setting its value to an empty string. For example, to remove the color property of an element, you can use the following code:

```
const myElement = document.querySelector('.my-element');
myElement.style.color = '';
```

This will remove the color property from the element's inline styles.

- **Use CSS classes:**

An easier and more maintainable way to manipulate styles is to add or remove CSS classes from an element. You can use the classList property to add or remove classes from an element, and then define the styles for those classes in a separate CSS file or in the style tag of the HTML document.

```
const myElement = document.querySelector('.my-element');
myElement.classList.add('red-text');

// css file
.red-text {
  color: red;
}
```

This method separates the styling from the JavaScript code, making it easier to maintain and modify.

Methods of DOM – Part 3

Topics Covered:

- eventListener

eventListener

In JavaScript, an event is an action that occurs on the web page, such as a user clicking a button, scrolling the page, or typing on the keyboard. You can use event listeners to detect when an event occurs and execute code in response.

An event listener is a function that you attach to an element, and it is called whenever a specific event occurs on that element. In JavaScript, you can attach an event listener to an element using the **addEventListener** method. Here is the basic syntax for using **addEventListener:**

```
element.addEventListener(eventType, listenerFunction, useCapture);
```

- **element** is the element to which you want to attach the event listener.
- **eventType** is a string that specifies the type of event you want to listen for. Examples of event types include "click", "keydown", "scroll", etc.
- **listenerFunction** is the function that you want to be executed when the event occurs.
- **useCapture** parameter can be set to either **true** or **false**. When useCapture is true, the event listener is executed in the capture phase, which is the first phase of the event propagation process. When useCapture is false (the default), the event listener is executed in the bubbling phase, which is the second phase of the event propagation process.

Here's an example that shows how to add a click event listener to a button element:

```
const myButton = document.querySelector('#my-button');

myButton.addEventListener('click', function() {
  alert('Button clicked!');
});
```

In this example, we select the button element with the ID **my-button**, and then attach a click event listener to it using **addEventListener**. The event listener is an anonymous function that displays an alert when the button is clicked.

You can also remove an event listener using the **removeEventListener** method, which takes the same arguments as **addEventListener**. Here's an example that shows how to remove an event listener:

```
const myButton = document.querySelector('#my-button');

function onClick() {
  alert('Button clicked!');
}

myButton.addEventListener('click', onClick);

// Remove the event listener
myButton.removeEventListener('click', onClick);
```

In this example, we attach a click event listener to the button element using a named function called onClick. We then remove the event listener using **removeEventListener**.

It is important to use a named function instead of an anonymous function when adding an event listener if you plan to remove that event listener later using the **removeEventListener** method.

The reason for this is that the **removeEventListener** method needs a reference to the same function that was used to add the event listener in order to remove it. When you use an anonymous function to add the event listener, there is no way to reference that function later, so you cannot remove the event listener.

In addition to the event type and listener function, you can also specify additional options when using **addEventListener**. For example,

- you can set the **once** option to **true** to only execute the event listener once:

```
const myButton = document.querySelector('#my-button');

myButton.addEventListener('click', function() {
  alert('Button clicked!');
}, { once: true });
```

In this example, the event listener will only be executed the first time the button is clicked. Subsequent clicks will not trigger the event listener.

- Here is an example of how to use the **passive** option:

```
const element = document.querySelector('div');

element.addEventListener('touchstart', function(event) {
  // Handle touchstart event
}, { passive: true });
```

In this example, we add a touchstart event listener to a div element, and set the **passive** option to **true**. This tells the browser that the event listener will not call **preventDefault()**, and allows the browser to handle the touch event more efficiently.

Note that not all event types support the **passive** option, and the behavior of the option can vary between browsers. In general, you should only set **passive** to **true** if you are sure that the event listener will not call **preventDefault()**, and if you have tested the performance of your code on multiple devices and browsers.

In JavaScript, there are many types of events that you can listen to using event listeners. Here are some of the most commonly used event types:

- **Mouse events:** These events are triggered by the user interacting with the mouse. Examples of mouse events include **click, mousemove, mousedown,** and **mouseup.**
- **Keyboard events:** These events are triggered by the user interacting with the keyboard. Examples of keyboard events include **keydown, keyup,** and **keypress.**
- **Form events:** These events are triggered by user actions within a form element. Examples of form events include **submit, reset,** and **change.**
- **Document events:** These events are triggered by changes to the document or the window. Examples of document events include **load, resize,** and **scroll.**
- **Touch events:** These events are triggered by touch screens and mobile devices. Examples of touch events include **touchstart, touchmove,** and **touchend.**
- **Drag and drop events:** These events are triggered by dragging and dropping elements on the page. Examples of drag and drop events include **dragstart, dragend, dragenter,** and **dragleave.**

Event listeners are a fundamental concept in web development, and they are used extensively to make web pages interactive and responsive to user input. By using event listeners, you can create web pages that respond to user actions in real-time, making for a more engaging and dynamic user experience.