

---

# Session 22

---

Mohamed Emary

June 10, 2024

## 1 innerHTML and innerText

### 1.1 innerHTML

- `innerHTML` returns the HTML content of an element.
- When assigning a value with HTML tags to `innerHTML`, the browser will render the HTML tags as HTML elements.
- When printing the value of `innerHTML` of an element that contains HTML tags, the browser will show the HTML tags in the output.

Example:

HTML:

```
1 | <p id="example">My <strong>example</strong> paragraph</p>
```

JavaScript:

```
1 | var example = document.getElementById('example');
2 | console.log(example.innerHTML); // My <strong>example</strong> paragraph
3 |
4 | example.innerHTML = 'My <strong>new</strong> paragraph';
5 | console.log(example.innerHTML); // My <strong>new</strong> paragraph
```

### 1.2 innerText

- `innerText` returns the text content of an element.
- When assigning a value with HTML tags to `innerText`, the browser will render the HTML tags as plain text.
- When printing the value of `innerText` of an element that contains HTML tags, the browser will show the HTML tags in the output.

Example:

HTML:

```
1 | <p id="example">My <strong>example</strong> paragraph</p>
```

JavaScript:

```
1 | var example = document.getElementById('example');
2 | console.log(example.innerHTML); // My example paragraph
3 |
4 | example.innerHTML = 'My <strong>new</strong> paragraph';
5 | console.log(example.innerHTML); // My <strong>new</strong> paragraph
```

Here the `<strong>new</strong>` appear in the web page as it is because `innerHTML` does not render HTML tags.

## 2 Creating Elements

To create an element, you can use the `document.createElement()` method. This method creates a new element with the specified tag name.

Example:

```
1 | var newElement = document.createElement('div');
```

To set the attributes of the new element, you can use the `setAttribute()` method or the `.` notation.

Example:

```
1 | // Using the setAttribute() method
2 | newElement.setAttribute('id', 'new-element');
3 | newElement.setAttribute('class', 'new-class');
4 |
5 | // Or using the . notation
6 | newElement.id = 'new-element';
7 | newElement.className = 'new-class';
```

### 2.1 Appending & Prepending Elements (Child)

To append an element inside another element in the DOM, you can use the `append()` method, and to prepend an element, you can use the `prepend()` method.

Example:

HTML:

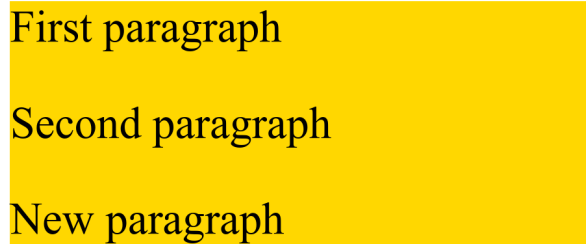
```
1 | <div id="parent" style="background-color: gold">
2 |   <p>First paragraph</p>
3 |   <p>Second paragraph</p>
4 | </div>
```

JavaScript:

```
1 | var parent = document.getElementById('parent');
2 | var newElement = document.createElement('p');
3 | newElement.innerHTML = 'New paragraph';
4 |
```

```
5 // Append the new element inside the parent element
6 parent.append(newElement);
```

Now the result will look like this:



First paragraph

Second paragraph

New paragraph

Figure 1: Using `append`

### 2.2 Add Element Before or After Another (Sibling)

To add an element before or after another element in the DOM, you can use the `before()` and `after()` methods.

Example:

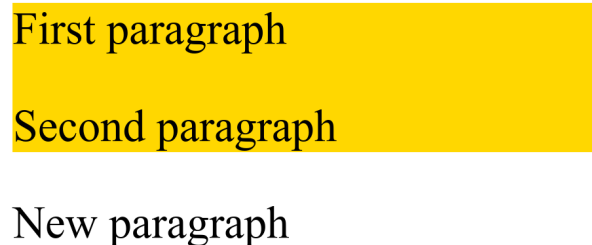
HTML:

```
1 <div id="parent" style="background-color: gold">
2   <p>First paragraph</p>
3   <p>Second paragraph</p>
4 </div>
```

JavaScript:

```
1 var parent = document.getElementById('parent');
2 var newElement = document.createElement('p');
3 newElement.innerText = 'New paragraph';
4
5 // Add the new element after the second paragraph
6 parent.after(newElement);
```

Now the result will look like this:



First paragraph

Second paragraph

New paragraph

Figure 2: Using `after`

#### Note

You can only send elements as arguments to the `append()`, `prepend()`, `before()`, and `after()` methods. If you send HTML tag or text, it will be treated as a string and not as an element.

### 3 Traversing the DOM

Traversing the DOM which is a way to move around the DOM tree and select elements based on their relationship to other elements.

Some useful properties and methods for traversing the DOM are:

1. `parentElement`: returns the parent **element** of an element.
2. `parentNode`: returns the parent **node** of an element.
3. `firstElementChild`: returns the first child **element** of an element.
4. `lastElementChild`: returns the last child **element** of an element.
5. `children`: returns an **HTML collection** of an element's child elements.
6. `childNodes`: returns a **NodeList** of an element's child nodes.
7. `nextElementSibling`: returns the next sibling **element** of an element.
8. `previousElementSibling`: returns the previous sibling **element** of an element.
9. `nextSibling`: returns the next sibling **node** of an element.
10. `previousSibling`: returns the previous sibling **node** of an element.

Example:

HTML:

```
1 <div id="parent" style="background-color: gold">
2   <p id="p1">First paragraph</p>
3   <p>Second paragraph</p>
4 </div>
```

JavaScript:

```
1 var parent = document.getElementById('parent');
2 var p1 = document.getElementById('p1');
3
4 // Get the parent element of the first paragraph
5 var parentElement = p1.parentElement;
6 console.log(parentElement.id); // parent
7
8 // Get the parent node of the first paragraph
9 var parentNode = p1.parentNode;
10 console.log(parentNode.id); // parent
11
12 // Get the first child of the parent element
13 var firstChild = parent.firstElementChild;
14 console.log(firstChild.innerText); // First paragraph
15
16 // Get the last child of the parent element
17 var lastChild = parent.lastElementChild;
18 console.log(lastChild.innerText); // Second paragraph
19
20 // Get all the child elements of the parent element
21 var children = parent.children;
```

## 4 IMPORTANT DIFFERENCES

---

```
22 console.log(children.length); // 2
23 console.log(children[1]); // <p>Second paragraph</p>
24
25 // Get all the child nodes of the parent element
26 var childNodes = parent.childNodes;
27 console.log(childNodes.length); // 3
28 console.log(childNodes[1]); // #text
29
30 // Get the next sibling element of the first paragraph
31 var nextSibling = p1.nextElementSibling;
32 console.log(nextSibling.innerHTML); // Second paragraph
33
34 // Get the previous sibling element of the second paragraph
35 var previousSibling = lastChild.previousElementSibling;
36 console.log(previousSibling.innerHTML); // First paragraph
37
38 // Get the next sibling node of the first paragraph
39 var nextNode = p1.nextSibling;
40 console.log(nextNode); // #text
41
42 // Get the previous sibling node of the second paragraph
43 var previousNode = lastChild.previousSibling;
44 console.log(previousNode); // #text
```

## 4 Important Differences

### 4.1 previousSibling, nextSibling VS previousElementSibling, nextElementSibling

- previousSibling and nextSibling return nodes and these nodes include both element nodes and non-element nodes (like text and comment nodes).
- previousElementSibling and nextElementSibling return only element nodes and ignore text and comment nodes.

```
1 <div>
2   <p>Paragraph 1</p>
3   Text
4   <p>Paragraph 2</p>
5 </div>
```

If the current node is the first `<p>` element, `nextSibling` would return the text node `Text`, while `nextElementSibling` would return the second `<p>` element.

### 4.2 NodeList VS HTML Collection

---

NodeList	HTML Collection
Static (Does not update when the DOM changes)	Live (Updates when the DOM changes)
Returns a list of nodes	Returns a list of elements

---

## 5 BROWSER OBJECT MODEL (BOM)

---

NodeList	HTML Collection
Nodes can be of any type like element, text, comment, etc.	Elements only
Returned by methods like <code>querySelectorAll</code> , <code>childNodes</code>	Returned by methods like <code>getElementsByTagName</code> , <code>children</code>

By saying that the `NodeList` is static, it means that if you add an element to the DOM after getting the `NodeList`, the `NodeList` will not include the new element. On the other hand, the `HTML Collection` is live, which means that it will include the new element even after getting the `HTML Collection`.

Example:

HTML:

```
1 <div>
2   <p>Paragraph 1</p>
3   <p>Paragraph 2</p>
4 </div>
```

JavaScript:

```
1 var div = document.querySelector('div');
2 var paragraphsCollection = div.getElementsByTagName('p');
3 var paragraphsNodeList = div.querySelectorAll('p');
4
5 console.log(paragraphsCollection.length); // 2
6 console.log(paragraphsNodeList.length);   // 2
7
8 var newParagraph = document.createElement('p');
9 newParagraph.innerText = 'New paragraph';
10
11 div.append(newParagraph);
12
13 console.log(paragraphsCollection.length); // 3
14 console.log(paragraphsNodeList.length);   // 2
```

In this example, the `paragraphsCollection` will have a length of 3, while the `paragraphsNodeList` will have a length of 2 because the `NodeList` is static and does not change when the DOM changes while the `HTML Collection` is live and changes when the DOM changes.

## 5 Browser Object Model (BOM)

Browser Object Model or BOM is a set of objects provided by the browser to interact with the browser itself.

### 5.1 DOM VS BOM

The DOM can be accessed via the BOM through the `window.document` property. So, you can say that the DOM is part of the BOM in a browser environment.

## 5 BROWSER OBJECT MODEL (BOM)

---

`window` is a super global object in the browser environment.

DOM is concerned with the content of the web document, while the BOM is concerned with the browser environment.

### 5.2 BOM Methods & Properties

Some of the BOM methods include:

#### 5.2.1 `setInterval`

`setInterval()`: Calls a function or evaluates an expression each time a specified number of milliseconds elapses.

For example, to display the value of a counter every second:

```
1 function incrementCounter() {  
2   console.log(counter);  
3   counter++;  
4 }  
5  
6 var counter = 0;  
7 var interval = setInterval(incrementCounter, 1000);
```

#### 5.2.2 `clearInterval`

`clearInterval()`: Stops the intervals set by `setInterval()`.

For example, to stop the counter we made earlier when the user clicks a button:

```
1 var button = document.getElementById('stop');  
2  
3 button.addEventListener('click', function() {  
4   console.log("Counter stopped");  
5   clearInterval(interval);  
6 });
```

#### 5.2.3 `setTimeout`

`setTimeout()`: Calls a function or evaluates an expression **once** after a specified number of milliseconds.

For example, to display a message after 3 seconds:

```
1 function showMessage() {  
2   console.log("Hello, world!");  
3 }  
4  
5 setTimeout(showMessage, 3000);
```

#### 5.2.4 `alert`

`alert()`: Displays an alert box with a message and an OK button.

For example:

## 5 BROWSER OBJECT MODEL (BOM)

---

```
1 | alert("Hello, world!");
```

### 5.2.5 open

`open()`: Opens a new browser window or a new tab.

For example, This will open a new tab with Google's homepage.

```
1 | var googleBtn = document.getElementById('open');
2 | googleBtn.addEventListener('click', function() {
3 |     open('https://www.google.com', '_blank');
4 | });
```

`_blank` is the name of the target window. It specifies that the URL should be opened in a new tab and it's the default value. To open the URL in the same tab, you can use `_self`.

`open` also has other parameters like `width`, `height`, `top`, `left`, etc.

```
1 | open('https://www.google.com', '_blank',
    ↪   'width=500,height=500,top=100,left=100');
```

Notice that `width`, `height`, `top`, and `left` are passed as a string with a comma separating them.

`open` is also one of `window`'s methods, so you can use it in the form `window.open()`.

### 5.2.6 close

`close()`: Closes the current window.

For example, to close the current window when the user clicks a button:

```
1 | var closeBtn = document.getElementById('close');
2 | closeBtn.addEventListener('click', function() {
3 |     close();
4 | });
```

### 5.2.7 innerWidth and innerHeight

`innerWidth` and `innerHeight` properties return the width and height of the content area of the browser window.

If you resize the browser window, the values of `innerWidth` and `innerHeight` will change accordingly.

```
1 | console.log(window.innerWidth);
2 | console.log(window.innerHeight);
```

### 5.2.8 screen Object

The `screen` object provides information about the user's screen.

Some of the properties of the `screen` object include:

- `screen.width`: Returns the width of the screen.
- `screen.height`: Returns the height of the screen.



Those properties don't change when you resize the browser window because they are related to the user's screen (the hardware) and not the browser window.

```
1 | console.log(screen.width);  
2 | console.log(screen.height);
```

`screen` object also has other properties like `availWidth`, `availHeight`.

These are the areas of the screen that you can use to display content, it doesn't include the taskbar or any other system-related areas.

```
1 | console.log(screen.availWidth);  
2 | console.log(screen.availHeight);
```

### 5.3 location Object

The `location` object contains information about the current URL.

Some of the properties of the `location` object include:

- `location.href`: Returns the entire URL.
- `location.hostname`: Returns the domain name of the web host.
- `location.pathname`: Returns the path and filename of the current page.
- `location.history`: Returns the history of the current page.
  - `location.history.back()`: Goes back to the previous page.
  - `location.history.forward()`: Goes forward to the next page.

## 6 API

API (Application Programming Interface) is universal way for different software applications to communicate with each other.

APIs can be used to **recieve** data from a server, **send** data to a server, **modify** data on a server, and **delete** data from a server.

The API comes in the form of a URL that you can send a request to and get a response from.

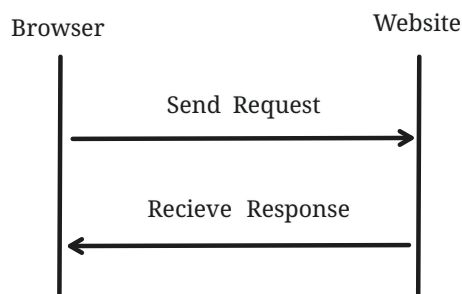


Figure 3: Request & Response

When dealing with APIs the front-end developer takes the API from the back-end developer with the documentation of how to use it. [Example API documentation](#).

Then the front-end developer uses the API to get the data needed to display on the web page.

A good way to test an APIs is to use a tool like [Postman](#), which is a collaboration platform for API development.

### 6.1 JSON

When you get a response from an API, it's usually in the form of JSON (JavaScript Object Notation).

JSON objects are easy to read and write. They are human-readable and can be parsed by JavaScript. JSON objects are written in key/value pairs and can be either an object or an array of objects.

Example of a JSON object:

```
1 {  
2   "name": "Mohamed",  
3   "age": 30,  
4   "city": "Cairo"  
5 }
```

### 6.2 Free APIs

There are many free APIs available that you can use to practice working with APIs.

Example Free APIs:

- [JSONPlaceholder](#)
- [Forkify Meals API](#)
- [Random User Generator API](#)
- [Weather API](#)
- [News API](#)
- [MovieDB API](#)
- [Fake Store API](#)

You can find many more on [this public-apis GitHub repo](#).

### 6.3 Terms Related to APIs

There is some terms related to APIs:

Lets use this API to explain the terms: <https://api.github.com/users/Microsoft>

- **Base URL:** The main URL of the API. For example, <https://api.github.com/>
- **Endpoint:** The part of the URL after the base URL that specifies a particular resource or collection of resources. For example, [/users/Microsoft](#) is the endpoint in the URL <https://api.github.com/users/Microsoft>.
- **Request:** The action you want the API to perform. In this case, a **GET** request to <https://api.github.com/users/Microsoft> to retrieve the data of the user **Microsoft**.

- **Response:** The data you get back from the API. This is typically in the form of a JSON object or array.
- **Status Code:** A number returned by the server that indicates the result of the request. For example, 200 means the request was successful, while 404 means the requested resource could not be found.
- **Method:** The type of request you are making. Common methods include GET, POST, PUT, DELETE, and PATCH.
  - GET: To get data from the server.
  - POST: To send data to the server.
  - PUT: To update data on the server.
  - DELETE: To delete data on the server.
  - PATCH: To partially update data on the server.
  - PUT: To update data on the server.

## 6.4 How to Use an API

To use an API, you need to know about AJAX (Asynchronous JavaScript and XML) first.

AJAX allows you to send and receive data from a server asynchronously without reloading the page.

You can use the `XMLHttpRequest` object to interact with the server and get data from it. We don't use the object directly, but we create a `new` instance of it and use its methods.

Example of creating an instance of `XMLHttpRequest`:

```
1 | var xhr = new XMLHttpRequest();
```

Then you can use the `open()` method to establish a connection with the server by specifying the request **method** and the **API URL**.

```
1 | xhr.open('METHOD', 'API_URL');
```

For example to establish a connection with the forkify API using the GET method:

```
1 | xhr.open('GET', 'https://forkify-api.herokuapp.com/api/search?q=pizza');
```

After opening the connection, you can use the `send()` method to send the request to the server.

```
1 | xhr.send();
```

To handle the response from the server, you can use the `onload` event handler.

```
1 | xhr.addEventListener('load', function() {  
2 |   console.log(xhr.response);  
3 | });
```

The `response` property of the `XMLHttpRequest` object contains the response data from the server as a **string**, so you need to parse it to a JSON object.

```
1 | xhr.addEventListener('load', function() {  
2 |   var data = JSON.parse(xhr.response);  
3 |   console.log(data);  
4 | });
```

Now you can access the data returned by the API.

```
1 xhr.addEventListener('load', function() {
2   var data = JSON.parse(xhr.response);
3   console.log(data.recipes);
4 });
```

You can also use `readystatechange` event handler to check the status of the request before accessing the data.

```
1 xhr.addEventListener('readystatechange', function() {
2   if (xhr.readyState === 4 && xhr.status === 200) {
3       var data = JSON.parse(xhr.response);
4       console.log(data.recipes);
5   }
6 });
```

`readyState` values are:

- 0: request not initialized
- 1: server connection established
- 2: request sent
- 3: processing request
- 4: request finished and response is ready

`status` values are:

- 200: OK (request successful)
- 403: Forbidden (access denied)
- 404: Not Found (resource not found)
- 500: Internal Server Error

Here is the complete code to get data from the forkify API:

```
1 var xhr = new XMLHttpRequest();
2 xhr.open('GET', 'https://forkify-api.herokuapp.com/api/search?q=pizza');
3 xhr.send();
4
5 xhr.addEventListener('load', function() {
6   if (xhr.readyState === 4 && xhr.status === 200) {
7       var data = JSON.parse(xhr.response);
8       console.log(data.recipes);
9   }
10 });
```

We also have `error` event handler to handle errors when the request fails.

```
1 xhr.addEventListener('error', function() {
2   console.log('An error occurred');
3 });
```

### 6.5 Displaying Data from an API

This is an example of how to display data from an API on a web page.

HTML:

```
1 <div class="container">
2   <div class="row" id="rowBody"></div>
```

```
3 | </div>
```

JavaScript:

```
1 | var xhr = new XMLHttpRequest();
2 | var allRecipies = [];
3 | xhr.open("get", "https://forkify-api.herokuapp.com/api/search?q=pizza");
4 | xhr.send();
5 | xhr.addEventListener("readystatechange", function () {
6 |     if (xhr.readyState == 4 && xhr.status == 200) {
7 |         allRecipies = JSON.parse(xhr.response).recipes;
8 |         display();
9 |     }
10 | });
11 |
12 | function display() {
13 |     var content = ``;
14 |     for (var i = 0; i < allRecipies.length; i++) {
15 |         content += `
16 |             <div class="col-md-4">
17 |                 
22 |                 <h3>${allRecipies[i].title}</h3>
23 |                 <p>${allRecipies[i].publisher}</p>
24 |             </div>`;
25 |     }
26 |     document.getElementById("rowBody").innerHTML = content;
27 | }
```

### 7 Summary

- `innerHTML` returns the HTML content of an element, while `innerText` returns the text content of an element.
- When assigning a value with HTML tags to `innerHTML`, the browser will render the HTML tags as HTML elements.
- When assigning a value with HTML tags to `innerText`, the browser will render the HTML tags as plain text.
- To create an element, you can use the `document.createElement()` method.
- To append an element inside another element in the DOM, you can use the `append()` method, and to prepend an element, you can use the `prepend()` method.
- To add an element before or after another element in the DOM, you can use the `before()` and `after()` methods.
- Traversing the DOM is a way to move around the DOM tree and select elements based on their relationship to other elements.
- `previousSibling` and `nextSibling` return nodes and include both element nodes and non-element nodes, while `previousElementSibling` and `nextElementSibling` return only element nodes.
- The Browser Object Model (BOM) is a set of objects provided by the browser to interact with the browser itself.
- Some of the BOM methods include `setInterval`, `clearInterval`, `setTimeout`, `alert`, `open`, and `close`.
- The `screen` object provides information about the user's screen, and the `location` object contains information about the current URL.
- APIs (Application Programming Interfaces) are used to communicate between different software applications.
- JSON (JavaScript Object Notation) is a common format for data exchange in APIs.
- To use an API, you need to know about AJAX (Asynchronous JavaScript and XML).
- You can use the `XMLHttpRequest` object to interact with the server and get data from it.
- To display data from an API on a web page, you can create an instance of `XMLHttpRequest`, send a request to the API, and handle the response to access the data.
- Common HTTP methods include `GET`, `POST`, `PUT`, `DELETE`, and `PATCH`.
- The `readyState` property of the `XMLHttpRequest` object indicates the state of the request, and the `status` property indicates the status of the response.
  - `readyState` values: 0, 1, 2, 3, 4
  - `status` values: 200, 403, 404, 500