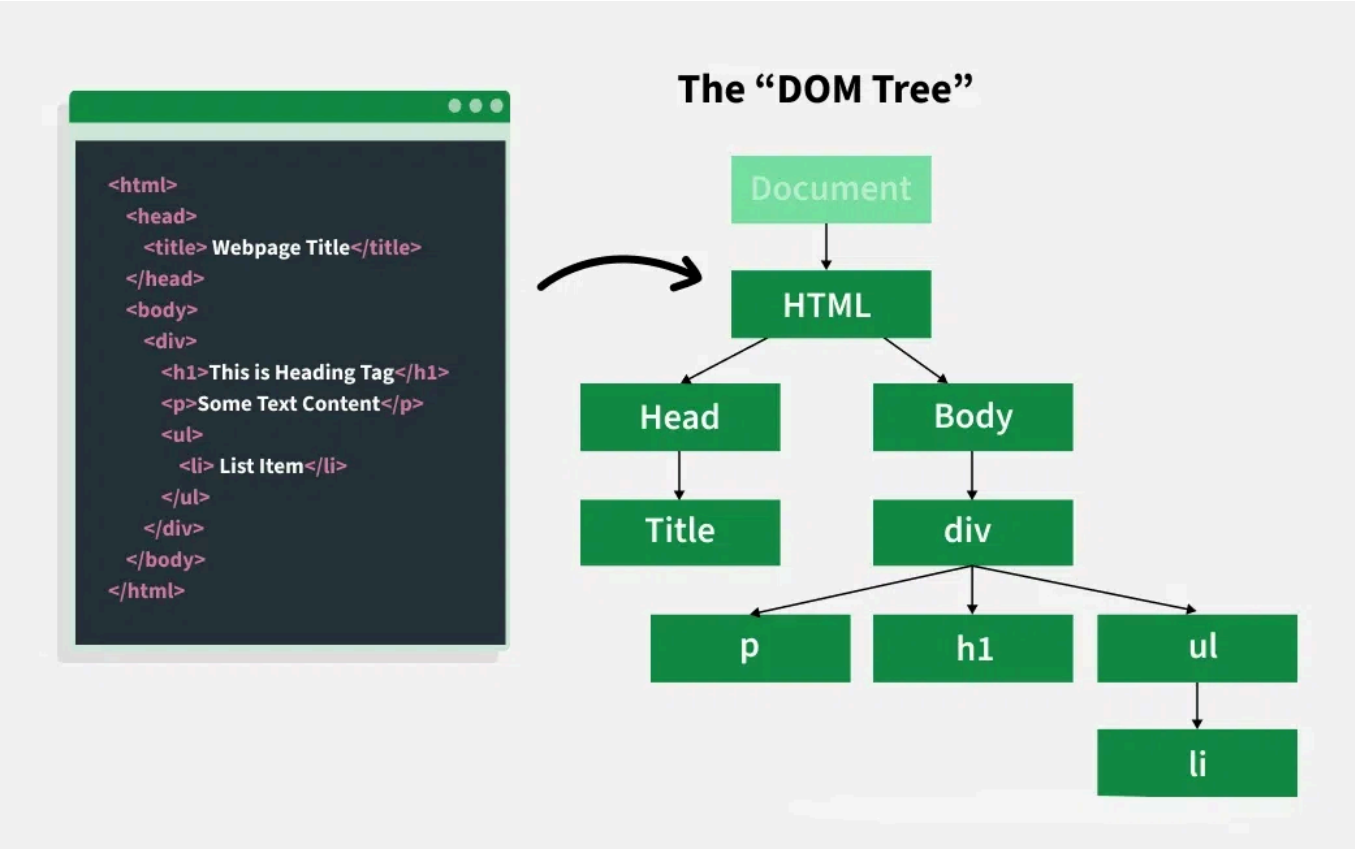
Objectives

- DOM Manipulation
- Browser Object Model (BOM) & Client-Side Data Storage
- WebSockets & Real-Time Data

DOM Manipulation

Introduction

The Document Object Model, or DOM, is a programming interface for HTML, XML, and SVG documents. It provides a structured representation of the document as a tree and defines how that structure can be manipulated by programs in terms of style and content. The DOM represents the document as a set of nodes and objects, each with its own properties and methods. These nodes can also have event handlers that are triggered when events occur. This enables web pages to be dynamically manipulated using scripts and/or programming languages. When an event is triggered, the associated event handlers are executed. JavaScript gives us an easy way to access document objects by selecting elements from our HTML page and manipulating them by modifying, removing, or updating them. It also allows us to create new elements and add them to the DOM.



Adding JavaScript to the Webpage

Two Methods to Include JavaScript

Internal Script

The JavaScript code is written directly within `<script>` tags in the HTML document

```
<script>
  // JavaScript code goes here
  alert("This is an internal script!");
  console.log("Executed directly in HTML");
</script>
```

External Script

This method involves creating separate .js files that are then linked to HTML documents using the `<script>` tag's src attribute.

```
<script src="scripts/main.js"></script>
```

Accessing The DOM Elements

Before manupulating the Dom element we have to access and select them there is many ways to select element from the doom tree we will see the following

Selecting By Tag Name

We can select all elements with a specific HTML tag name by using `document.getElementsByTagName()`. This method returns a live **HTMLCollection** of all elements that match the specified tag name.

```
// Select all <div> elements
const divElements = document.getElementsByTagName('div');
```

Selecting By Class Name

We can select all elements with a specific class name using `document.getElementsByClassName()`.

```
// Select all elements with 'menu-item' class
const menuItems = document.getElementsByClassName('menu-item');
```

Selecting By ID

We can select a single element by its unique ID using `document.getElementById()`. Since IDs must be unique within a document, this method always returns:

- The matching **Element object** if found
- `null` if no element exists with that ID

```
// Select element with ID "header"
const headerElement = document.getElementById('header');
```

CSS Selectors

JavaScript provides methods to select elements using CSS selectors, offering flexibility beyond basic ID, class, and tag selection:

1. **Selecting the First Matching Element**

```
// Select first <button> with class "primary"
const btn = document.querySelector('button.primary');
```

2. **Selecting All Matching Elements**

```
// Select all <input> elements with type "text"
const textInputs = document.querySelectorAll('input[type="text"]');
```

Manipulating DOM Elements

After selecting elements, we can manipulate them in several ways:

Changing Element Content

1. We can change all HTML content inside our element by using .innerHTML

```
const element = document.querySelector('#target');
element.innerHTML = '<strong>New</strong> content';
```

2. We can change text inside our element by using .textContent

```
const element = document.querySelector('#target');
element.textContent = 'Plain text content';
```

3. We can change the value inside form elements by using .value

```
const element = document.querySelector('#target');
element.value = 'New input value';
```

Changing Element Style

JavaScript provides two primary methods for modifying element styles:

1. Modify specific style properties one at a time using the `style` attribute:

```
const element = document.querySelector('#target');
element.style.backgroundColor = 'blue';
element.style.fontSize = '1.2rem';
```

2. Apply multiple styles simultaneously using `cssText`:

```
const element = document.querySelector('#target');
element.style.cssText = `
  color: white;
  padding: 1rem;
  display: flex;
`;
```

Managing Attributes

JavaScript also lets us access element attributes to read, modify, or add new attributes

1. Access attribute values directly

```
const link = document.querySelector('a');
const href = link.href;
const title = link.getAttribute('title');
```

2. Retrieve the complete attribute list

```
const attributes = document.getElementById('myElement').attributes;
```

3. Set or update attributes

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

img.setAttribute('alt', 'Description of image');
img.src = 'new-image.jpg';
```

3. Access form field values

```
const input = document.querySelector('input[type="text"]');

// Current value (always returns string)
const value = input.value;

// For checkboxes/radio buttons
const isChecked = checkbox.checked;
```

4. Managing Class Names

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

// Get all classes
const classList = div.classList;

// Check for class
const hasClass = div.classList.contains('active');

// Add class
div.classList.add('highlight');

// Remove class
div.classList.remove('old-style');
```

```
// Replace all classes
div.className = 'new-class another-class';
```

Removing Elements

We can remove an element from the DOM by using the `.remove()` method.

```
const element = document.querySelector('#target');
element.remove();
```

Event Handling

When users open our pages, they often do more than just read the content. Sometimes they click on elements, scroll the page, hover over input fields to fill them out, or press keys on the keyboard. JavaScript allows us to handle all of these actions through **event handling**.

Adding an Event Listener to an Element

We can add an event listener to an element using the `addEventListener` method. This method takes two main arguments:

1. The **event type** (e.g., `"click"`, `"keydown"`, `"mouseover"`), and
2. The **callback function** that will run when the event occurs.

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

button.addEventListener('click', function() {
  // Code to execute when button is clicked
  console.log('Button was clicked!');
});
```

Common Event Types

Event	Description
`click`	Mouse click
`input`	Form field changes
`keydown`	Keyboard key press
`scroll`	Page/element scrolling

Event	Description
`mouseover`	Mouse pointer enters element

Creating and Inserting Elements

Sometimes, we want to add new elements to a web page using JavaScript — for example, adding a new paragraph, button, or image dynamically. JavaScript makes this easy using the `document.createElement` method.

Creating New Elements

We can make new elements using `document.createElement()`:

```
// Create a new div
const newDiv = document.createElement('div');

// Create a paragraph
const newParagraph = document.createElement('p');
```

Setting Up Our Elements

```
// Add text content
newParagraph.textContent = 'This is our new paragraph';

// Add a class
newDiv.className = 'container';

// Style it
newDiv.style.backgroundColor = '#f0f0f0';
```

Adding Elements to Our Page

We have several ways to put elements where we want them:

Method	What It Does	Example
`append()`	Adds to the end	`parent.append(newDiv)`
`prepend()`	Adds to the start	`parent.prepend(newDiv)`
`before()`	Puts before another	`element.before(newDiv)`

Method	What It Does	Example
`after()`	Puts after another	`element.after(newDiv)`
`insertBefore()`	Inserts a node before a specified child node.	`insertBefore(newDiv, referenceNode)`
`replaceChild()`	Replaces one child node with another.	`replaceChild(newDiv, oldDiv)`:

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

newParagraph.textContent = 'This is our new paragraph';
newDiv.className = 'container';
newDiv.style.backgroundColor = '#f0f0f0';

const container = document.querySelector('#container');
container.append(newDiv);
const oldItem = document.querySelector('.old');
oldItem.before(newParagraph);

container.insertAdjacentHTML('beforeend', '<span>New item</span>');
```

Making Copy of Element

We can duplicate elements with `cloneNode()`:

```
const original = document.querySelector('.template');
const copy = original.cloneNode(true);

copy.id = 'new-copy';
document.body.append(copy);
```

Browser Object Model & Client-Side Data Storage

Introduction to the Browser Object Model (BOM)

The **Browser Object Model (BOM)** provides a way for JavaScript to interact with the browser itself, giving us access to various browser features such as the window, history, location, navigator, and more. Unlike the DOM, which represents the document structure, BOM is concerned with the browser environment and allows us to manipulate the browser's behavior

and state.
Some important BOM objects include:

- `window`: Represents the browser window or frame.
- `document`: Represents the HTML document loaded in the browser.
- `navigator`: Provides information about the browser.
- `location`: Provides information about the current URL and allows manipulation of the browser's address bar.
- `history`: Allows interaction with the browser's session history.

Accessing the BOM

To interact with the BOM, we generally use the `window` object. The `window` object is automatically available in the global scope, we don't need to explicitly reference `window` to access most of its properties.

Accessing the Window Object

```
console.log(window.innerWidth); // Width of the window
console.log(window.innerHeight); // Height of the window
```

Using the Location Object

```
console.log(window.location.href); // Access the current URL
window.location.href = "https://www.example.com"; // Change the current URL
```

Working with Client-Side Data Storage

Modern web browsers provide ways to store data on the client side, meaning that data can be saved locally on a user's device without the need for server-side storage. JavaScript provides several mechanisms for client-side storage, such as **Cookies**, **LocalStorage**, and **SessionStorage**.

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is primarily used to send data between a server and a client in web applications.

JSON Structure

A JSON object is a collection of key-value pairs, similar to JavaScript objects. However, in JSON:

- Keys must be strings enclosed in double quotes.
- Values can be strings, numbers, arrays, booleans, or other objects.

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "address": {
    "street": "123 Main St",
    "city": "Anytown"
  },
  "courses": ["Math", "Science", "History"]
}
```

Converting Between JSON and JavaScript Objects

To send or receive JSON in a JavaScript program, we often need to convert between JSON strings and JavaScript objects using `JSON.parse()` and `JSON.stringify()`.

Parsing JSON (String to Object)

We sse `JSON.parse()` to convert a JSON string into a JavaScript object.

```
const jsonString = '{"name": "John", "age": 30}';
const jsonObject = JSON.parse(jsonString);
console.log(jsonObject.name); // Output: John
```

Converting to JSON (Object to String)

We sse `JSON.stringify()` to convert a JavaScript object into a JSON string.

```
const user = { name: "John", age: 30 };
const jsonString = JSON.stringify(user);
console.log(jsonString); // Output: '{"name":"John","age":30}'
```

Cookies

Cookies are small pieces of data stored by the browser that can be sent with every HTTP request to the server. Cookies are often used for tracking, storing session data, and remembering user preferences.

```
// Set a cookie with name, value, and expiration
document.cookie = "username=JohnDoe; expires=Thu, 18 Dec 2025 12:00:00 UTC";

// Access cookies from the document
console.log(document.cookie);

// Delete a cookie by setting its expiration date to the past
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC";
```

LocalStorage

LocalStorage allows us to store data that persists even when the browser is closed and reopened. Data stored in LocalStorage has no expiration date.

Storing Data in LocalStorage

```
// Store data in LocalStorage
localStorage.setItem('username', 'JohnDoe');
```

Retrieving Data from LocalStorage

```
// Retrieve data from LocalStorage
const username = localStorage.getItem('username');
console.log(username); // Output: JohnDoe
```

Removing Data from LocalStorage

```
// Remove data from LocalStorage
localStorage.removeItem('username');
```

Clearing All Data from LocalStorage

```
// Clear all data from LocalStorage
localStorage.clear();
```

SessionStorage

SessionStorage works similarly to LocalStorage but with a key difference: the data is only available for the duration of the page session. Once the browser tab is closed, the data is erased.

Storing Data in SessionStorage

```
// Store data in SessionStorage
sessionStorage.setItem('sessionID', 'abc123');
```

Retrieving Data from SessionStorage

```
// Retrieve data from SessionStorage
const sessionID = sessionStorage.getItem('sessionID');
console.log(sessionID); // Output: abc123
```

Removing Data from SessionStorage

```
// Remove data from SessionStorage
sessionStorage.removeItem('sessionID');
```

Clearing All Data from SessionStorage

```
// Clear all data from SessionStorage
sessionStorage.clear();
```

Encoding Data Using URL Encoding and Blob

In web development, we often need to encode data for various purposes, such as submitting form data, sending data through URLs, or handling files. Two common ways of encoding data are **URL encoding** and **using Blobs**.

URL Encoding

URL encoding (also known as **percent encoding**) is used to encode data into a valid format for use in URLs. It replaces special characters with a `%` followed by their ASCII code in hexadecimal.
For example:

- A space character becomes `%20`

- An ampersand `&` becomes `%26`

This encoding ensures that data is correctly transmitted as part of a URL without breaking the structure of the URL itself.

Encoding Data for URLs

To encode data in JavaScript, we use the built-in `encodeURIComponent()` function. It is used to encode individual components of a URL (such as query parameters), ensuring special characters are properly encoded.

```
const name = "John Doe & Co.";
const encodedName = encodeURIComponent(name);
console.log(encodedName); // Output: John%20Doe%20%26%20Co.
```

Decoding URL-encoded Data

To decode data that has been URL-encoded, use the `decodeURIComponent()` function.

```
const encodedName = "John%20Doe%20%26%20Co.";
const decodedName = decodeURIComponent(encodedName);
console.log(decodedName); // Output: John Doe & Co.`
```

Download URL-encoded Data

Once the data is encoded, we can enable the user to download and save it. This is achieved by creating a link element, assigning the `href` attribute to the encoded URL, and using the `download` attribute to specify the desired file name for saving the data.

```
const name = "John Doe & Co.";
const encodedName = encodeURIComponent(name);
const link = document.createElement('a');
link.href = 'data:text/plain;charset=utf-8,' + encodedContent;
link.download = 'download.txt';
document.body.appendChild(link);
link.click();
document.body.removeChild(link);
```

Using Blobs

A **Blob (Binary Large Object)** represents raw data of any type. It can be used to store binary data, such as images, audio, or video files, and is often used when working with files in web applications.
Blobs can be created from strings or other data sources and are typically used with APIs like the `FileReader`, `fetch`, and `URL.createObjectURL()` to handle files in the browser.

Creating a Blob from Data

We can create a Blob using the `Blob()` constructor, which takes an array of data and an optional MIME type.

```
const textData = "Hello, this is some text content!";
const blob = new Blob([textData], { type: 'text/plain' });  // Create an object URL
for the Blob
const url = URL.createObjectURL(blob); console.log(url); // Output: blob URL
```

Downloading Text Files with Blobs

Blobs are often used to trigger downloads of content in the browser. For instance, we can create a text file and allow the user to download it.

```
const textData = "This is the content of my text file.";
const blob = new Blob([textData], { type: 'text/plain' });
const url = URL.createObjectURL(blob);  // Create a link and simulate a click to
download the file
const link = document.createElement('a');
link.href = url;
link.download = 'file.txt'; // File name
link.click();
// Clean up the object URL after use
URL.revokeObjectURL(url);
```

Reading Blobs with FileReader

We can read Blobs with the FileReader API, which allows you to read the contents of a Blob as text or binary data.

```
const blob = new Blob(['Hello, world!'], { type: 'text/plain' });
const reader = new FileReader();
reader.onload = function(event) {
```

```
console.log(event.target.result); // Output: Hello, world!
};
reader.readAsText(blob);
```

Blob vs URL-encoded

Feature	URL Encoding	Blob
Data Type	Strings only	Any data (text/binary)
Use Case	URL parameters	File handling
Size Limit	URL length limits (~2000 chars)	Large files possible
Memory	No cleanup needed	Requires manual cleanup
Performance	Very lightweight	Heavier resource use
Reversable	Yes (decodeURIComponent)	Yes (FileReader)

WebSockets & Real-Time Data

Introduction

Traditional web communication between the browser and the server uses the **HTTP protocol**, where the browser sends a request and waits for the server to respond. But in many modern applications, such as chat apps, online games, or stock tickers, we need a way to **continuously receive data** from the server without reloading the page.
This is where **WebSockets** come in. A WebSocket provides a **full-duplex** communication channel meaning both the browser and the server can send messages to each other at any time. Once a WebSocket connection is opened, it stays open, allowing for **real-time data exchange**.

```
const socket = new WebSocket('wss://echo.websocket.org');

socket.addEventListener('open', () => {
  console.log('WebSocket connection opened');
  socket.send('Hello Server!');
});

socket.addEventListener('message', (event) => {
  console.log('Message from server:', event.data);
});
```

```
socket.addEventListener('close', () => {
  console.log('WebSocket connection closed');
});
```

How WebSockets Work

1. The client (browser) initiates a WebSocket connection using the `WebSocket` object.
2. The connection is established with a WebSocket server.
3. Messages can then be sent and received in real time.
4. Either side can close the connection when it's no longer needed.

Creating a WebSocket Connection

To open a connection to a WebSocket server, we create a new instance of the `WebSocket` class, passing the URL of the WebSocket server.

```
const socket = new WebSocket('wss://example.com/socket');
```

- Use `wss://` for secure connections (like `https://`)
- Use `ws://` for non-secure connections (like `http://`)

Listening for WebSocket Events

The WebSocket object provides several events that we can listen to:

Event	Description
`open`	Triggered when the connection opens
`message`	Triggered when a message is received
`error`	Triggered when an error occurs
`close`	Triggered when the connection closes

```
socket.addEventListener('open', () => {
  console.log('Connected to server');
});

socket.addEventListener('message', (event) => {
  console.log('Received:', event.data);
});
```

```
socket.addEventListener('error', (error) => {
  console.error('WebSocket error:', error);
});

socket.addEventListener('close', () => {
  console.log('Connection closed');
});
```

WebSocket Methods

Component	Description
`new WebSocket(url)`	Creates a new WebSocket instance
`send(data)`	Sends data to the server (strings, Blob, ArrayBuffer)
`close()`	Terminates the connection

Example

```
const chatSocket = new WebSocket('wss://chat.example.com');
chatSocket.addEventListener('message', (event) => {
  const messageBox = document.getElementById('chat');
  const msg = document.createElement('p');
  msg.textContent = event.data;
  messageBox.appendChild(msg);
});

document.getElementById('sendBtn').addEventListener('click', () => {
  const input = document.getElementById('messageInput');
  chatSocket.send(input.value);
  input.value = '';
});
```

When we finish using a WebSocket connection, we should properly close it using the `.close()` method.

```
chatSocket.close()
```

Task

Objective

Create a responsive Todo List application that:

- Uses **DOM manipulation** to add, edit, and delete tasks
- Saves tasks to **localStorage** for persistence
- Allows exporting/importing tasks as **JSON files**
- Implements **modern CSS** (Grid, Flexbox, Animations)
- Follows **OOP principles** and uses **async operations**

Requirements

Core Functionality

- Add new tasks with titles, descriptions, and due dates
- Mark tasks as complete/incomplete
- Edit existing tasks
- Delete tasks
- Filter tasks (All/Active/Completed)

Data Persistence

- Automatically save tasks to **localStorage**
- Load tasks from localStorage on page refresh
- **Export tasks** as a JSON file (using Blobs)
- **Import tasks** from a JSON file (using FileReader)

UI/UX Features

- **Responsive layout** (works on mobile/desktop) using:
  - CSS Grid for main structure
  - Flexbox for task items
- **Animations**:
  - Smooth transitions when adding/removing tasks
  - Hover effects on buttons
- **Visual feedback**:
  - Gradient backgrounds
  - Transform effects (e.g., scale on hover)

Advanced JavaScript

- Use **OOP** (Classes) to manage tasks
- Implement **async/await** for file operations
- **Error handling** for file imports

Solution:

You can find our solution here:
https://alitigui.github.io/Front_end_solutions/Lecture12/Solution/index.html