DOM

Introduction

The **DOM (Document Object Model)** is a programming interface for web documents. It represents the page so that programs can manipulate the structure, style, and content of a document. The DOM represents the page as a tree structure, where each node corresponds to a part of the document (such as an element, attribute, or text).

In simple terms, the DOM allows JavaScript (and other programming languages) to interact with the content of a web page

Key Concepts of DOM

1. **Document Object**:
   The `document` object is the entry point for JavaScript to interact with the HTML document. It represents the entire HTML document and gives you access to everything within it, such as elements, attributes, and text.
2. **DOM Tree Structure**:
   The DOM is structured as a tree of nodes. The top-level node is the `document` object, and it branches out to various elements and other nodes.

- **Root**: The `document` node is the root.
- **Children**: Elements inside the document (like `<html>`, `<head>`, `<body>`) are children of the document.
- **Leaf Nodes**: The text content inside tags or attributes are leaf nodes.

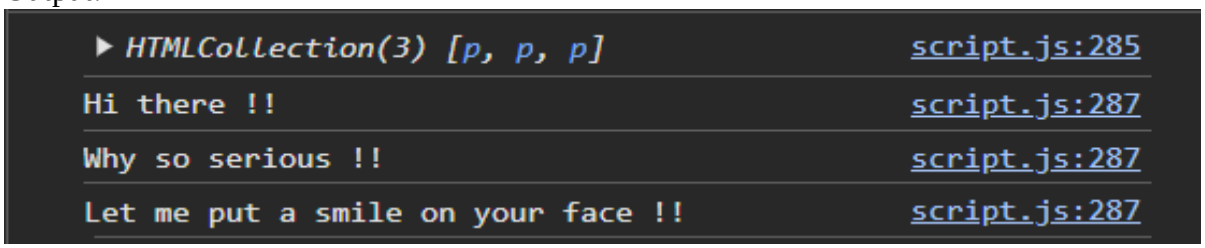
DOM Methods and Properties

- **Accessing Elements**
  1. getElementsByTagName
     - Returns a collection of elements with the specified tag name

Example

```
const paragraphs=document.getElementsByTagName('p')

console.log(paragraphs)

for (let i of paragraphs){

   console.log(i.textContent);

}
```

Output:

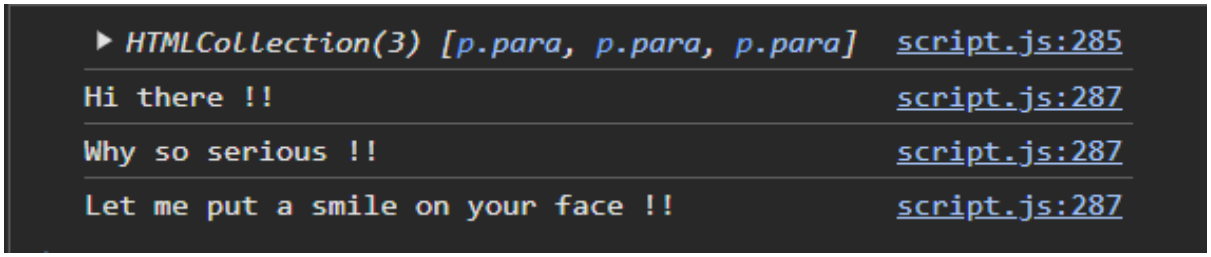


2. getElementsByClassName
    - Returns a collection of elements with the specified class.

Example:

```
const paragraphs=document.getElementsByClassName('para')
console.log(paragraphs)
for (let i of paragraphs){
   console.log(i.textContent);
}
```

Output:



3. getElementById
    - Returns an element with the specified id
    - It is one of the most common and fastest ways to access an element in the DOM because id values are guaranteed to be unique within the document

Example :

```
const paragraph=document.getElementById('pr')
console.log(paragraph.textContent)
```

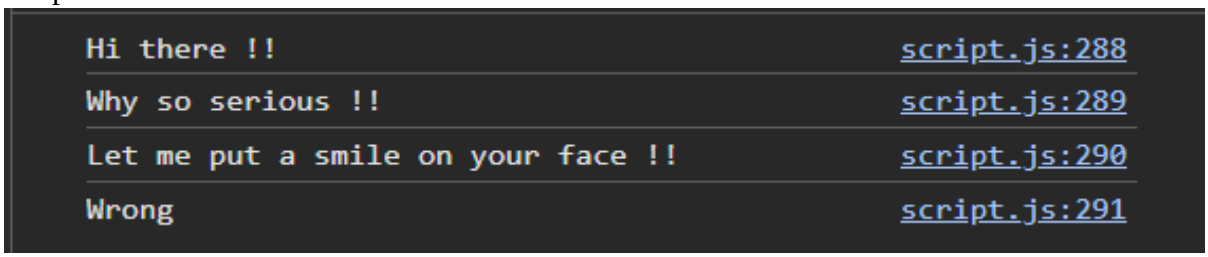
Ouput:

4. querySelector:
    - Selectors
        - When using querySelector and querySelectorAll, you can use a wide range of CSS selectors to target elements. These selectors allow for very flexible and precise element selection.
        - Universal
        - Tag selector
        - Class selector
        - ID selector
        - Attribute Selector
        - Combinators
            - Descendant Selector
            - Child Selector
            - Adjacent Selector
            - General Sibling Selectors
    - Returns the first element that matches a CSS selector.
    - It can replace getElementById(), getElementsByClassName(), and getElementsByTagName() with one unified syntax.

Example:

```
function get(){
    const paragraph=document.querySelector('p') //tag selector
    const paragraph1=document.querySelector('.para') //class selector
    const paragraph2=document.querySelector('#pr') //id selector
    const input=document.querySelector('input[type="text"]') //attribute selector
    console.log(paragraph.textContent)
    console.log(paragraph1.textContent)
    console.log(paragraph2.textContent)
    console.log(input.value)
}
```

Output



5. querySelectorAll
    - Returns all elements that match a CSS selector.
    - Returns as a Nodelist[A **NodeList** is a collection of DOM (Document Object Model) nodes in JavaScript]
      The returned NodeList can be iterated using loops like forEach, for...of, or traditional for loops.

Example:

```
const paragraph1=document.querySelectorAll('.para')

paragraph1.forEach((paras)=>{

   console.log(paras.textContent)

})
```

Output:

```
Hi there !!                                    script.js:285
Why so serious !!                              script.js:285
Let me put a smile on your face !!             script.js:285
```

- **Modifying Elements**
    1. **innerHTML**
        - The innerHTML property allows you to access or modify the HTML content of an element. It represents everything inside an element, including child elements, text, and markup.
        - Parses HTML content

Example:

```
const element=document.getElementById('pr')

console.log(element.innerHTML) //get the content

element.innerHTML='It is Wrong' //modify the content in the html page
```

Output:

```
Let me put a smile on your face !!             script.js:284
```

2. **innerText**
    - Retrieves or sets the visible text of an element, ignoring HTML tags.
    - Only retrieves the visible text. Text inside elements hidden by CSS (display: none or visibility: hidden) will not be included.

Example:

```
const element=document.getElementById('pr')
console.log(element.innerText) //get the content
element.innertext='It is Wrong' //modify the content in the html page
```

Output:

```
Let me put a smile on your face !!          script.js:284
```

3. **textContent**
    - Similar to innerText, but retrieves the full text content, including hidden elements.

Example:

```
const element=document.getElementById('pr')
console.log(element.textContent) //get the content
element.textContent='It is Wrong' //modify the content in the html page
```

Output:

```
Let me put a smile on your face !!          script.js:284
```

Feature	`innerHTML`	`innerText`	`textContent`
Returns/Modifies	HTML and text content	Visible text only	All text, including hidden text
Aware of Styles	No	Yes	No
Security Risk	Yes (XSS possible)	No	No
Performance	Slower (parses HTML)	Faster	Fastest

4. **setAttribute**
    - Modifies the value of an element's attribute.

Example:

```
const element=document.getElementById('pr')
console.log(element.setAttribute('style','color:red'))
```

Output:

Hi there !!

Why so serious !!

Let me put a smile on your face !!

5. **getAttribute**
    - Retrieves the value of an element's attribute

Example:

```
const element=document.getElementById('pr')
console.log(element.getAttribute('style'))
```

Output:

```
   color: red;                                    script.js:284
>
```

6. **classList**
    - Manipulates the classes of an element

Example:

```
const element2=document.getElementById('pr')
element2.classList.add('para')
//we can remove the class that we added using
//element2.classList.remove('para')
```

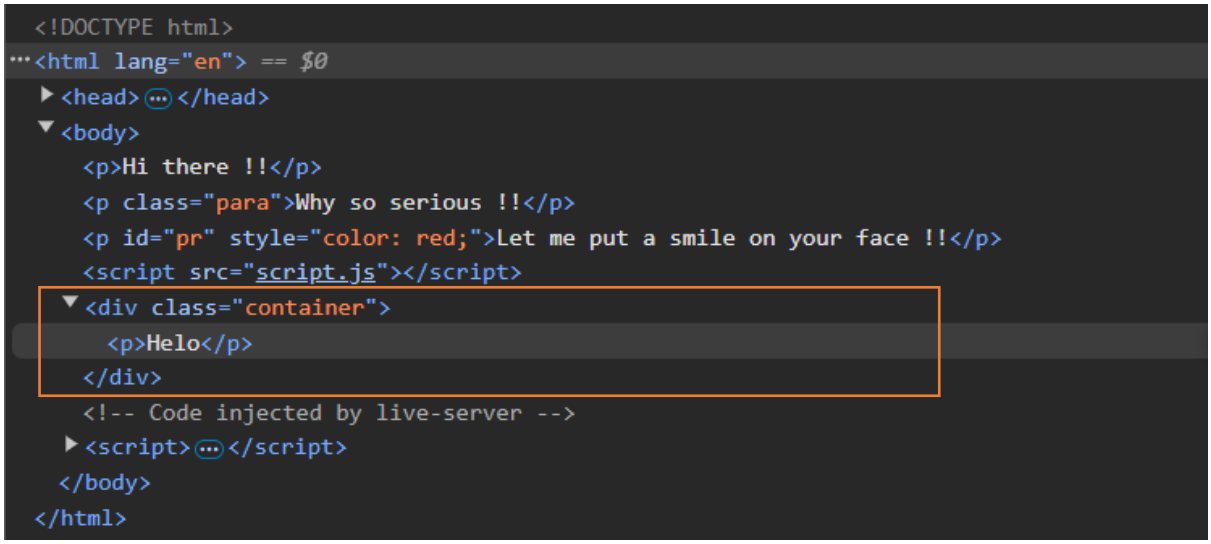
Output:

- **Creating and Removing Elements**
  1. createElement:
     - Creates new HTML element
  2. appendChild:
     - Adds new element as a child of another element

Example:

```
const newElement=document.createElement('div')

newElement.innerHTML='<p>Helo</p>'

newElement.classList.add('container')

document.body.appendChild(newElement)
```

Output:

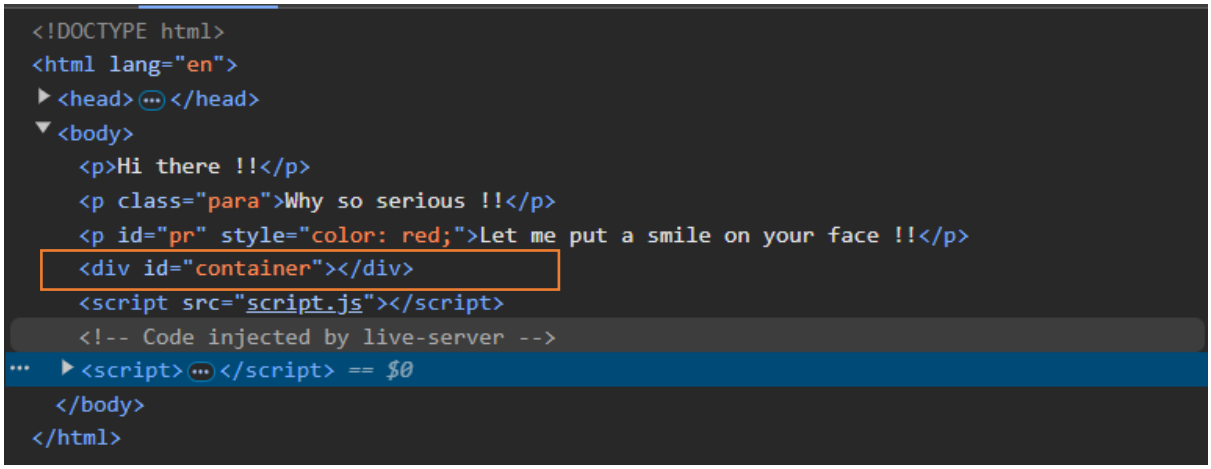


3. removeChild
    - Removes child from its parent

Example:

```
const element=document.getElementById('container')
const para=document.getElementById('p2')
element.removeChild(para)
```

Output:

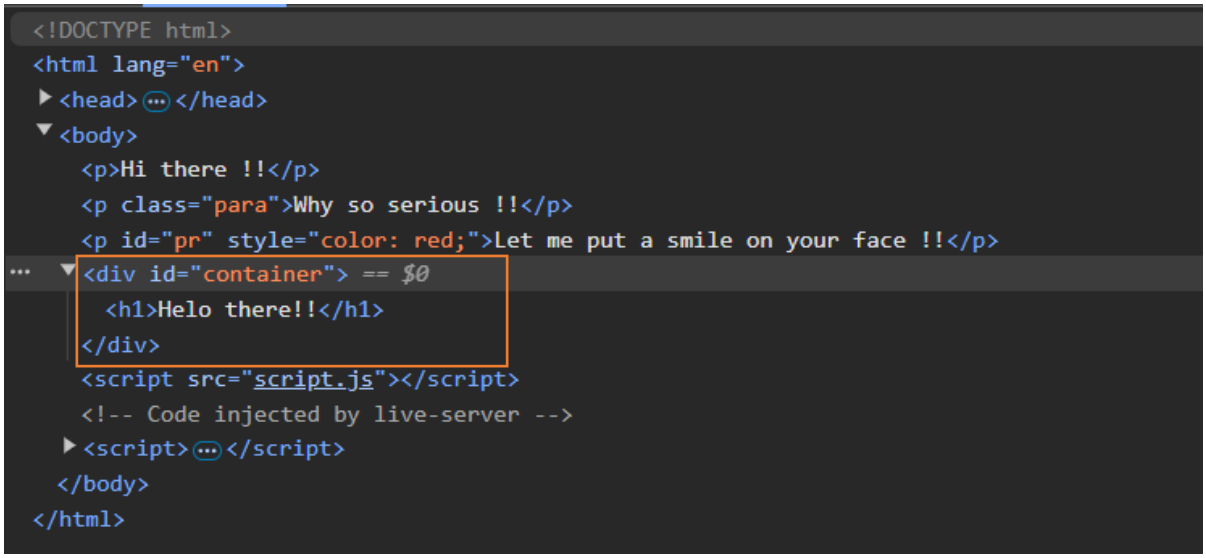


4. replaceChild
    - Replace a child element with a new one

Example:

```
const element=document.getElementById('container')
const para=document.getElementById('p2')
const newheading=document.createElement('h1')
newheading.innerHTML='Helo there!!'
element.replaceChild(newheading,para)
```

Output:



- **Event Handling Methods**

  Events

  - **onclick**
    - **When It Occurs**: The `onclick` event is triggered when the user clicks on an element (usually with the mouse), but it can also be triggered by other devices, such as touch on mobile.
    - **Use Case**: It is typically used to handle mouse click actions like submitting a form, opening a link, toggling visibility, etc.
  - **keyup**
    - **When It Occurs**: The `keyup` event is triggered when the user releases a key on the keyboard.
    - **Use Case**: It is useful when you want to perform an action after the user has finished typing or releasing a key, such as checking input validation or searching as the user types.

Event Type	Description
Mouse Events	
`click`	Triggered when the mouse is clicked.
`dblclick`	Triggered when the mouse is double-clicked.
`mousedown`	Triggered when the mouse button is pressed.
`mouseup`	Triggered when the mouse button is released.
`mousemove`	Triggered when the mouse moves within the element.
`mouseenter`	Triggered when the mouse enters an element.
`mouseleave`	Triggered when the mouse leaves an element.
`mouseover`	Triggered when the mouse pointer enters an element or its child.
`mouseout`	Triggered when the mouse pointer leaves an element or its child.
Keyboard Events	
`keydown`	Triggered when a key is pressed down.
`keypress`	Triggered when a key is pressed and produces a character (deprecated in modern browsers).
`keyup`	Triggered when a key is released.

Event handling refers to the process of defining how an event (e.g., a click or hover) is managed when it occurs on a specific element.

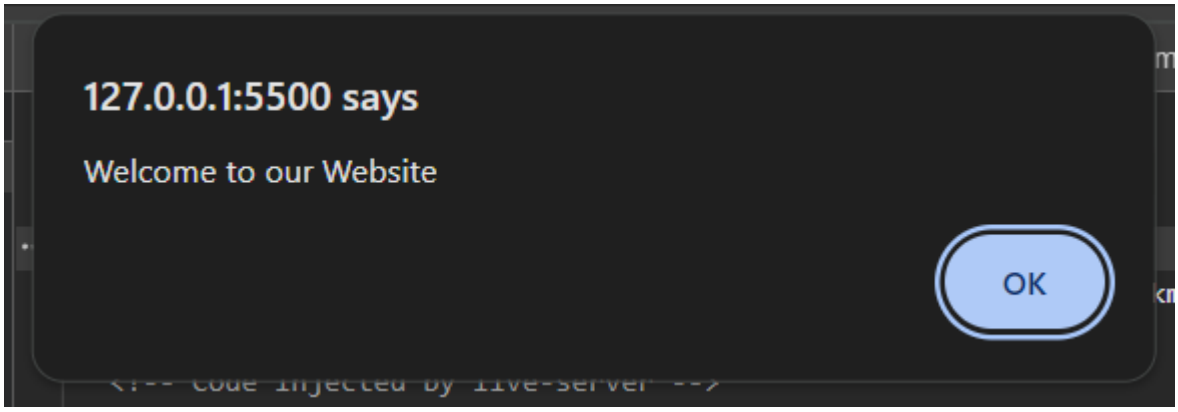
Event Handling Methods

1. **Inline Event Handlers**
   - Add the event directly in the HTML element

Example:

```
<!DOCTYPE html>
<html lang="en">
▶ <head> ⋯ </head>
⋯ ▼ <body> == $0
    <button id="btn" onclick="alert('Welcome to our Website')">Clickme</button>
    <script src="script.js"></script>
    <!-- Code injected by live-server -->
  ▶ <script> ⋯ </script>
  </body>
</html>
```

Output:



2. **Event Properties**
   Assign an event handler to an event property.

Example:

```
const myBtn=document.getElementById('btn')
myBtn.onclick=function(){
   console.log('Button clicked')
}
```

Output:

3. **Event Listeners**
   - ***addEventListener***

Syntax:

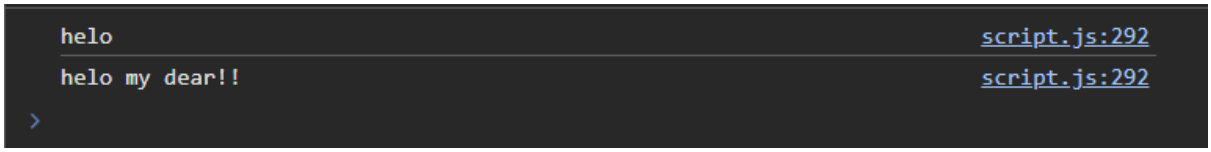
```
element.addEventListener(eventType, listener, options);
```

- eventType : A string representing the event (eg : “click”,”mouse”,”keydown”)
- listener : The function that will be executed when the even occurs
- Options : Optional parameter to specify even behaviour

Example:

```
const mybtn=document.getElementById('btn')
mybtn.addEventListener('click',function(){
    const text=document.getElementById('textbx')
    console.log(text.value)
})
```

Output:



Example 2 :

```
document.getElementById('textbx').addEventListener('keyup',(e)=>{
    console.log(e.target.value)
})
```

Ouput:

- ***removeEventListener***
  Removes an event listener from an element.

Example:

```
const addListenerButton = document.getElementById("addListener");
const removeListenerButton = document.getElementById("removeListener");
const targetButton = document.getElementById("targetButton");
const output = document.getElementById("output");
function handleClick() {
  output.textContent = "Target button clicked!";
}
addListenerButton.addEventListener("click", () => {
  targetButton.addEventListener("click", handleClick);
  output.textContent = "Event listener added.";
})
removeListenerButton.addEventListener("click", () => {
  targetButton.removeEventListener("click", handleClick);
  output.textContent = "Event listener removed.";
});
```

Possible output Sequence

1. If you click the "Add Listener" button, then the "Target Button", and then the "Remove Listener" button:

   "Event listener added."

   "Target button clicked!"

   "Event listener removed."

2. If you click the "Add Listener" button and then the "Target Button" (without clicking "Remove Listener"):

   "Event listener added."

   "Target button clicked!"

3. If you click the "Remove Listener" button before clicking the "Target Button":

   "Event listener removed."

TASKS

1. Create a FAQ section where clicking on a question reveals or hides the answer dynamically.
2. Design a feature where a small circle follows the cursor's movement, like interactive pointers on creative websites.
3. Display a tooltip when the mouse hovers over a specific element, as seen in navigation bars or icons on dashboards.
4. Implement a button that toggles between showing and hiding a section of content, like expandable dropdowns in settings pages.
5. Create an interactive button where hovering over it triggers an animation, such as scaling or changing its background color.
6. Create a character counter that displays real-time feedback while typing into a text box, like on Twitter or Instagram.
7. Create a parallax scrolling effect where elements move at different speeds based on mouse movement or scroll position.
8. Implement a feature where clicking on an image opens a larger, zoomed-in version in a modal.
9. Build a webpage that updates a section's content dynamically when a button is clicked (without reloading the page).
10. Implement a modal window that appears when a user clicks a button, and can be closed by clicking a close icon.

SYNCHRONOUS AND ASYNCHRONOUS JS

In JavaScript, synchronous and asynchronous programming models define how tasks are executed and how they interact with each other during runtime.

Synchronous JavaScript

- In synchronous execution, tasks are performed one after the other, blocking subsequent tasks until the current task is completed.
- JavaScript runs in a single-threaded environment, meaning it can only execute one task at a time.

Characteristics:

- Blocking: If a task takes time (e.g., a large computation or reading a file), it blocks the program's execution until the task is completed.
- Easier to read and debug due to its sequential nature.

Example:

```
console.log("Start");
function syncTask() {
console.log("Running synchronous task...");
}
syncTask();

console.log("End");
```

Output:

```
Start
Running synchronous task...
End
```

Here, each task waits for the previous one to complete.

Asynchronous JavaScript

- In asynchronous execution, tasks are started and can run independently, allowing other tasks to execute in the meantime.
- This is essential for tasks like:
    - Fetching data from a server.
    - Reading files.
    - Timers or delays.
    - Event handling.

How Asynchronous JavaScript Works:

JavaScript uses the event loop mechanism to handle asynchronous operations:

1. Asynchronous tasks (e.g., API calls) are delegated to a different thread (via the Web APIs in browsers or Node.js).
2. Once the task is completed, a callback is pushed to the callback queue.
3. The event loop picks up these callbacks and processes them when the main thread is idle.

Characteristics:

- Non-blocking: Other tasks can continue while the asynchronous task is running in the background.
- Uses mechanisms like callbacks, Promises, and async/await.

setInterval()

The **setInterval()** method in JavaScript is used to repeatedly execute a specified function or code snippet at fixed time intervals (in milliseconds).

Syntax

```
let intervalID = setInterval(function, delay, param1, param2, ...);
```

- **function**: The function or code snippet to be executed.
- **delay**: The time interval (in milliseconds) between each execution.
- **param1, param2, ...**: (Optional) Additional parameters passed to the function.

Example: Repeated Execution

This example logs a message (“*”) every 1 second (1000 milliseconds):

```
setInterval(function(){
        document.write(“*”)
},1000);
```

Stopping setInterval

You can stop a setInterval using the **clearInterval()** method:

```
let i=0;
let x=setinterval(function(){
        document.write("*");
        i++;
        if(i==10)
{
clearInterval(x);
},1000)
```

Passing Arguments

You can pass arguments to the function being executed:

```
function greet(name) {
console.log(`Hello, ${name}!`);
}

// Pass arguments to the function
const intervalID = setInterval(greet, 2000, "Alice");

// Stop after 6 seconds
setTimeout(() => clearInterval(intervalID), 6000);
```

Important Notes

1. **Infinite Loop Risk**: setInterval runs indefinitely unless you explicitly stop it with clearInterval().
2. **Execution Delays**: The interval does not guarantee precise timing, especially for delays shorter than the time required to execute the callback function.
3. **Alternative**: For more control, you can use setTimeout recursively instead of setInterval.

setTimeout()

The setTimeout() method in JavaScript is used to execute a function or a block of code after a specified delay (in milliseconds). Unlike setInterval(), it only executes the code once after the delay.

Syntax

```
let timeoutID = setTimeout(function, delay, param1, param2, ...);
```

- **function**: The function or code snippet to execute.
- **delay**: The time (in milliseconds) to wait before executing the code.
- **param1, param2, ...**: (Optional) Additional parameters to pass to the function.

Example

This example logs a message after 2 seconds (2000 milliseconds):

```
setTimeout(() => {
console.log("Hello after 2 seconds!");
}, 2000);
```

Stopping a setTimeout

You can stop a setTimeout using **clearTimeout()** before it executes:

```
const timeoutID = setTimeout(() => {
console.log("This will not be logged.");
}, 5000);

// Cancel the timeout
clearTimeout(timeoutID);
```

Passing Parameters

You can pass arguments to the function being executed:

```
function greet(name) {
console.log(`Hello, ${name}!`);
}

// Pass "Alice" as an argument
setTimeout(greet, 3000, "Alice");
```

Chaining Delays

You can use multiple setTimeout calls to create a sequence of delays:

```
setTimeout(() => {
console.log("First message after 1 second");
setTimeout(() => {
console.log("Second message after 3 more seconds");
setTimeout(() => {
console.log("Third message after 6 more seconds");
 }, 3000);
}, 2000);
}, 1000);
```

Difference Between **setTimeout** and **setInterval**

Feature	**setTimeout**	**setInterval**
Execution	Executes a function once after the delay.	Executes a function repeatedly at intervals.
Stopping	Can be stopped using clearTimeout().	Can be stopped using clearInterval().
Use Case	For delayed, one-time operations	For repeated tasks like timers.

Notes

1. Delay Precision: The actual delay can be slightly longer than the specified time due to the event loop.
2. Minimum Delay: The minimum delay in modern browsers is 4ms (except for nested calls, which may be 0ms).
3. Recursive setTimeout: You can use setTimeout recursively to achieve behavior similar to setInterval but with more precise control:

```
function repeatTask() {
  console.log("Task executed");
  setTimeout(repeatTask, 1000); // Call itself
}

repeatTask();
```

Tasks

1. Write a function fetchDataWithCallback(callback) that simulates fetching data after 2 seconds using setTimeout. Once the data is fetched, it should execute the callback function with the fetched data.

2. Write a function that logs "Hello, World!" after 2 seconds using setTimeout.

3. Create a function performTask(callback) that executes a task (e.g., logging "Task Complete") and then calls the provided callback function.

4. Write a function addNumbersAsync(a, b, callback) that takes two numbers, adds them after 1 second, and calls the callback with the result.

5. Write a function fetchData(url) that uses fetch to get data from a public API (e.g., https://jsonplaceholder.typicode.com/todos/1) and logs the result.

6. Create a promise that resolves with the message "Promise Resolved!" after 3 seconds and logs the message when resolved.

7. Create a function that:

1. Resolves with a random number after 1 second.
2. Chains another `.then()` to check if the number is odd or even.
3. Logs "Odd" or "Even" based on the result.

8. Write a function fetchUserPosts(userId) that:

1. Fetches user details from /users/:id (e.g., https://jsonplaceholder.typicode.com/users/1).
2. Fetches the posts of that user from /posts?userId=:id.
3. Logs the user's name and their post titles.

9. Write a function that:

1. Creates a promise that resolves after 2 seconds with a number (e.g., `10`).
2. Chains another `.then()` to add 5 to the number.
3. Logs the final result.

10. Convert the following code to use async/await:

```
fetch('https://jsonplaceholder.typicode.com/users')
.then((response) => response.json())
.then((data) => console.log(data))
.catch((error) => console.error(error));
```

SPREAD OPERATOR

The spread operator (...) is a feature in JavaScript that allows you to **expand an array, object, or iterable** into its individual elements. It provides a concise and flexible way to work with arrays and objects.

Common Use Cases

1. Copying Arrays

   The spread operator can create a shallow copy of an array

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
console.log(copiedArray); // Output: [1, 2, 3]
```

2. Merging Arrays

   Combine multiple arrays into one.

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const mergedArray = [...array1, ...array2];
console.log(mergedArray); // Output: [1, 2, 3, 4, 5, 6]
```

3. Copying Objects

   Create a shallow copy of an object.

```
const originalObject = { name: 'Alice', age: 25 };
const copiedObject = { ...originalObject };
console.log(copiedObject); // Output: { name: 'Alice', age: 25 }
```

4. Find Min/Max using spread operator

```
// Min in an array using Math.min()
let a = [1,2,3,-1];
console.log(Math.min(a)); //NaN

// Now using spread
console.log(Math.min(...a));
```

5. Merging Objects

   Combine properties from multiple objects into one.

```
const obj1 = { name: 'Alice' };

const obj2 = { age: 25, city: 'New York' };

const mergedObject = { ...obj1, ...obj2 };

 console.log(mergedObject); // Output: { name: 'Alice', age: 25, city: 'New York' }
```

6. Passing Elements to Functions

   The spread operator allows you to pass array elements as individual arguments to a function.

```
const numbers = [1, 2, 3];
 function sum(a, b, c)
{
        return a + b + c;
}
console.log(sum(...numbers)); // Output: 6
```

DESTRUCTURING

Destructuring is a JavaScript feature that allows you to unpack values from arrays or properties from objects into individual variables. It provides a more concise and readable way to extract data from arrays or objects.

Destructuring Arrays

```
const numbers = [1, 2, 3];

// Destructuring the array
const [a, b, c] = numbers;

console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(c); // Output: 3
```

Skipping Elements

You can skip elements in the array by leaving empty spaces

```
const numbers = [1, 2, 3, 4];

// Skipping the second element
const [first, , third] = numbers;

console.log(first); // Output: 1
console.log(third); // Output: 3
```

Using Default Values

Provide default values in case an element

```
const numbers = [1, 2];

// Destructure with default values
const [a, b, c = 10] = numbers;

console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(c); // Output: 10
```

Destructuring Objects

Basic Example

```
const person = {
  name: 'Alice',
  age: 25,
  city: 'New York',
};

// Destructuring the object
const { name, age, city } = person;

console.log(name); // Output: Alice
console.log(age);  // Output: 25
console.log(city); // Output: New York
```

Renaming Variables

You can rename variables while destructuring.

```
const person = {
  name: 'Alice',
  age: 25,
};

// Rename `name` to `firstName`
const { name: firstName } = person;

console.log(firstName); // Output: Alice
```

Using Default Values

Provide default values for missing properties

```
const person = {
  name: 'Alice',
};

// Destructure with default value
const { name, age = 30 } = person;

console.log(name); // Output: Alice
console.log(age);  // Output: 30
```

Nested Objects

Destructure nested objects by matching the structure.

```
const person = {
  name: 'Alice',
  address: {
    city: 'New York',
    zip: 10001,
  },
};

const { address: { city, zip } } = person;

console.log(city); // Output: New York
console.log(zip);  // Output: 10001
```

Combining Destructuring with Functions

Extracting Arguments

Destructure function parameters directly.

```
function greet({ name, age }) {
  console.log(`Hello, ${name}. You are ${age} years old.`);
}

const person = { name: 'Alice', age: 25 };
greet(person);
// Output: Hello, Alice. You are 25 years old.
```

Destructuring with Rest Operator

Arrays

```
const numbers = [1, 2, 3, 4];
const [a, b, ...rest] = numbers;
 console.log(a); // Output: 1
 console.log(b); // Output: 2
console.log(rest); // Output: [3, 4]
```

Objects

```
const person = {
  name: 'Alice',
  age: 25,
  city: 'New York',
};

const { name, ...rest } = person;

console.log(name); // Output: Alice
console.log(rest); // Output: { age: 25, city: 'New York' }
```

TASK

1. Copying Arrays: Write a function duplicateArray(arr) that takes an array and returns a new array with the same elements using the spread operator.
   Example Input: [1, 2, 3]
   Expected Output: [1, 2, 3]
2. Merge the following two arrays using the spread operator:
   const fruits = ['apple', 'banana'];
   const vegetables = ['carrot', 'cucumber'];
3. Use the spread operator to create a shallow copy of the following object:
   const user = { name: 'Alice', age: 25, city: 'New York' };
4. Write a function mergeObjects(obj1, obj2) that takes two objects as arguments and returns a new object that combines the properties of both objects using the spread operator.
   Example Input:
   obj1 = { a: 1, b: 2 }
   obj2 = { b: 3, c: 4 }
   Expected Output: { a: 1, b: 3, c: 4 }
5. Use the spread operator to extract the first two elements of an array and store the rest in another array.
   const numbers = [10, 20, 30, 40];
6. Destructure the following object to extract the name and age properties:
   const person = { name: 'John', age: 30, city: 'Los Angeles' };
7. Destructure the following object and provide default values for country and state:
   const location = { city: 'San Francisco' };
8. Extract the city and zip properties from the nested address object:
   const user = { name: 'Alice', address: { city: 'Seattle', zip: 98101, }, };
9. Write a function swap(a, b) that swaps the values of two variables using array destructuring?
10. Extract the id and name into variables and store the rest of the properties in another variable.
    const user = { id: 1, name: 'Alice', age: 25, city: 'New York' };
    Expected Output:
    id = 1, name = 'Alice', rest = { age: 25, city: 'New York' }