Module 6

MERNstack – JavaScript Essential and Advanced

JavaScript Introduction:

Question 1: What is JavaScript? Explain the role of JavaScript in web development.

Ans:-

- Javascript is a High-level , lightweight interpreted programming language that is primarily used to create dynamic and interactive features on websites. It runs in the browser, enabling functionalities like real-time updates, animations, and form validations.

- JavaScript plays a critical role in modern web development, enabling dynamic, interactive, and engaging user experiences.

1. **Client-Side Interactivity -**

   Dynamic Content , Event Handling , Form Validation

2. **Webpage Behaviour -**

   Animation , DOM Manipulation

3. **Backend Development -**

With frameworks like **Node.js**, JavaScript extends beyond the browser, allowing developers to create server-side applications and APIs.

4. **Full-Stack Development -**

JavaScript enables developers to use a single language for both frontend and backend development, simplifying the development process.

6. **Cross-Platform Development -**

JavaScript enables building cross-platform applications through frameworks like **Electron** (desktop apps) and **React Native** (mobile apps).

Question 2: How is JavaScript different from other programming languages like Python orJava?

Ans:-

JavaScript differs from other programming languages like Python or Java in several ways, including:

1. Execution Environment

- **JavaScript:**

- Primarily executed in web browsers, enabling dynamic and interactive web content.
- Can also run on servers with **Node.js**.

- **Python:**
    - A general-purpose language, commonly used for web development, data analysis, AI, and scientific computing.
    - Runs on desktops and servers but not natively in browsers.

- **Java:**
    - A general-purpose, platform-independent language used for building large-scale applications, Android apps, and backend systems.
    - Requires a Java Virtual Machine (JVM) for execution.

2. Use Cases

- **JavaScript:**
    - Best suited for web development, building dynamic and interactive web pages, real-time applications, and frontend/backend development (e.g., React, Angular, Node.js).

- **Python:**
    - Known for simplicity and versatility, ideal for web backends (e.g., Django, Flask), data science, machine learning, automation, and scripting.

- **Java:**
    - Used for enterprise-grade backend development, mobile applications (Android), and high-performance systems.

3. Performance

- **JavaScript:**
  - Designed for real-time interactivity; optimized for quick execution in browsers.
  - Faster for tasks that rely on event loops (e.g., user interactions, web APIs).

- **Python:**
  - Generally slower due to being an interpreted language, but excels in computational tasks when combined with libraries like NumPy or TensorFlow.

- **Java:**
  - Faster than JavaScript and Python in most scenarios due to compilation into bytecode and optimization by the JVM.
  - Suitable for high-performance, multi-threaded applications.

- **JavaScript:** Specializes in web interactivity and full-stack development.
- **Python:** Focuses on simplicity and versatility, excelling in data science, AI, and backend development.
- **Java:** Suited for enterprise applications, large systems, and mobile app development.

Question 3: Discuss the use of <script> tag in HTML. How can you link an external JavaScript file to an HTML document?

Ans:-

The `<script>` tag in HTML is used to embed JavaScript code or reference an external JavaScript file. It plays a key role in adding dynamic functionality and interactivity to web pages.

Using the `<script>` Tag

The `<script>` tag can be placed in the `<head>` or `<body>` section of an HTML document. It supports two main use cases:

Linking an External JavaScript File

To include JavaScript from an external file, use the `src` attribute in the `<script>` tag.

```html
<!DOCTYPE html>
<html>
<head>
  <title>External JavaScript Example</title>
</head>
<body>
    <button onclick="sayHello()">Click Me</button>
    <script src="script.js"></script>
</body>
</html>
```

In the above example, `script.js` is the external JavaScript file containing the function `sayHello`.

Contents of `script.js` :

```javascript
function sayHello() {
  alert("Hello from an external file!");
}
```

Variables and Data Types

Question 1: What are variables in JavaScript? How do you declare a variable using var, let, and const ?

Ans:-

In JavaScript, **variables** are used to store data that can be referenced and manipulated throughout a program. A variable acts like a container for storing information, which can later be retrieved or updated.

- JavaScript provides three keywords to declare variables:

1. Var
2. Let
3. Const

- declare a variable using var, let, and const

```
let variableName = value;  // Preferred for variables that may change
        const constantName = value;  // For variables that won't change
        var oldVariableName = value;  // Avoid in modern code
```

Question 2: Explain the different data types in JavaScript. Provide examples for each.

Ans:-

JavaScript has **primitive** and **non-primitive** data types. Primitive types are immutable and directly contain the data, while non-primitive types are objects that hold references to data.

Primitive Data Types

1. Number

Represents numeric values (both integers and floating-point numbers).

```
let age = 25;          // Integer
let price = 99.99;     // Floating-point number
```

2. String

Represents text, enclosed in single `' '`, double `" "`, or backticks `` ` ``.

```
let name = "Alice";
```

3. Boolean

Represents a logical value: `true` or `false`.

```
let isLoggedIn = true;
let hasPremiumAccount = false;
```

4. Undefined

A variable is `undefined` if it has been declared but not assigned a value.

```javascript
let user;
console.log(user); // Output: undefined
```

5. Null

Represents the intentional absence of any value.

```javascript
let emptyValue = null;
console.log(emptyValue); // Output: null
```

6. Symbol (Introduced in ES6)

Represents a unique, immutable value, often used as identifiers for object properties.

```javascript
let id = Symbol('uniqueId');
console.log(id); // Output: Symbol(uniqueId)
```

7. BigInt (Introduced in ES2020)

Used for representing integers larger than the `Number` type can handle.

```javascript
let bigNumber = 123456789012345678901234567890n; // Note the 'n' at the end
console.log(bigNumber);
```

Non-Primitive Data Types

1. Object

A collection of key-value pairs

```javascript
let user = {
            name: "Alice",
            age: 25,
            isPremium: true
        };
        console.log(user);
```

2. Array

A special type of object used to store lists of data.

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits);
```

3. Function

Functions are objects but can be invoked.

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

Question 3: What is the difference between undefinedand null in JavaScript ?

Ans:-

- Use `undefined` to indicate a variable is declared but not yet initialized, or when a function parameter is omitted.

- Use `null` to explicitly assign a value that represents "no value" or "empty."

Feature	undefined	null
Type	A primitive type.	A primitive type, but considered an object(typeof null returns “object” due to a historical quirk).
Meaning	Indicates a variable has been declared but not assigned a value.	Represents the intentional absence of any value. It is explicitly set by developer.

Default value	Assigned by JavaScript to uninitialized variables or missing function parameters.	Not assigned by JavaScript; it must be explicitly set.
Use Case	Used to signify an uninitialized variable, missing property, or function with no return statement.	Used to deliberately indicate "no value" or "empty."
Equality	`undefined == null` evaluates to `true` (loose equality).	`undefined === null` evaluates to `false` (strict equality).

JavaScript Operators

Question 1: What are the different types of operators in JavaScript? Explain with examples.

• Arithmetic operators

• Assignment operators

• Comparison operators

• Logical operators

Ans:-

There are following types of operators in JavaScript.

1. Arithmetic Operators
2. Assignment Operators
3. Comparison (Relational) Operators
4. Logical Operators
5. Bitwise Operators
6. Special Operators

1. Arithmetic Operators

Operator	Description	Example	Output
+	Addition	5+2	7
-	Subtraction	5-2	3
*	Multiplication	5*2	10
/	Division	10/2	5
%	Modulus(Remainder)	5%2	1

**	Exponentiation	2**3	8

2. Assignment Operators

Operator	Description	Example	Equivalent To
=	**Assignment**	**X = 5**	**X = 5**
+=	**Add and Assign**	**X += 5**	**X = X + 5**
-=	**Subtract and Assign**	**X -= 5**	**X = X - 5**
***=**	**Multiply and Assign**	**X *= 5**	**X = X * 5**
/=	**Divide and Assign**	**X /= 5**	**X = X / 5**
%=	**Modulus and Assign**	**X %= 5**	**X = X % 5**

3. Comparison Operators

Operator	**Description**	**Example**	**Output**
==	**Equal to (loose equality)**	**5 == 5**	**true**
===	**Strict equal to**	**5 === ‘5’**	**false**
!=	**Not equal to (loose inequality)**	**5 != ‘5’**	**false**
!==	**Strict not equal to**	**5 !== ‘5’**	**true**
>	**Greater than**	**5 > 3**	**true**
<	**Less than**	**5 < 3**	**false**
>=	**Greater than or equal to**	**5 >= 5**	**true**
<=	**Less than or equal to**	**5 <= 5**	**true**

4. Logical Operators

Operator	Description	Example	Output
&&	Logical AND	True && false	false
`		`	Logical OR
!	Logical NOT	!true	false

Question 2: What is the difference between ==and ===in JavaScript?

Ans:-

In JavaScript, equality operators like double equals **(==)** and triple equals (===) are used to compare two values. But both operators do different jobs. Double equals (==) will try to convert the values to the same data type and then try to compare them. But triple equals (===) strictly compares the value and the datatype.

Control Flow (If-Else, Switch)

Question 1: What is control flow in JavaScript? Explain how if-else statements work withan example.

Ans:-

Control flow is the order in which the computer executes statements in a script.

Code is run in order from the first line in the file to the last line, unless the computer runs across the (extremely frequent) structures that change the control flow, such as conditionals and loops.

The `if-else` statement is a conditional control structure that enables decision-making in your code. It executes a block of code if a specified condition is `true`. If the condition is `false`, the `else` block (if provided) will execute.

```
        if (condition) {
            // Code to execute if the condition is true
        } else {
            // Code to execute if the condition is false
        }
```

Condition: A boolean expression that evaluates to `true` or `false`.
Blocks of Code: Statements enclosed within curly braces `{}`.

Example: If-Else Statement

```
let age = 18;

if (age < 18) {
    console.log("You are a minor.");
} else if (age === 18) {
    console.log("You just became an adult!");
} else {
    console.log("You are an adult.");
}
```

Explanation:

- If `age` is less than 18, the first block executes.
- If `age` is exactly 18, the second block (`else if`) executes.
- If neither condition is true (i.e., `age > 18`), the `else` block executes.

Question 2: Describe how switch statements work in JavaScript. When should you use a switch statement instead of if-else?

Ans:-

The switch statement executes a block of code depending on different cases. The switch statement is a part of JavaScript's "Conditional" Statements, which are used to perform different actions based on different conditions. Use switch to select one of many blocks of code to be executed.

```
switch (expression) {
    case value1:
        // Code to execute if expression === value1
        break;
    case value2:
        // Code to execute if expression === value2
```

```
                break;
            default:
            // Code to execute if no case matches
        }
```

How It Works

1. The `expression` is evaluated once.
2. Its value is compared against the values in the `case` labels (using strict equality, ===).
3. If a match is found, the corresponding block of code is executed.
4. The `break` statement exits the `switch` block to prevent the execution from "falling through" to subsequent cases.
5. If no `case` matches, the code in the `default` block is executed (if provided).

When to Use a Switch Statement Instead of If-Else

The choice between using a `switch` statement and an `if-else` statement depends on the specific requirements of your code. Here are some scenarios where a `switch` statement is preferable.

- You have multiple fixed values to check for a single variable or expression.
- The code needs to be more readable and structured.
- Grouping logic for multiple cases is required.

Loops (For, While, Do-While)

Question 1: Explain the different types of loops in JavaScript (for, while, do-while). Provide abasic example of each.

Ans:-

Loops in JavaScript allow you to repeatedly execute a block of code as long as a specified condition is `true`. JavaScript provides several types of loops, each suited for different scenarios.

- **`for` Loop:** When you know the exact number of iterations (e.g., iterating through an array).
- **`while` Loop:** When the number of iterations depends on a condition (e.g., waiting for user input).
- **`do-while` Loop:** When the code must run at least once (e.g., input validation).

1. `for` Loop

The `for` loop is ideal when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and increment/decrement.

```
for (initialization; condition; increment / decrement) {
    // Code to execute
}
```

Example:

```
for (let i = 0; i < 5; i++) {
    console.log("Count:", i);
}
```

2. `while` Loop

The `while` loop is used when the number of iterations is not predetermined. It executes as long as the condition is `true`.

```
while (condition) {
    // Code to execute
}
```

Example:

```
let count = 0;

while (count < 3) {
    console.log("Count:", count);
    count++;
}
```

3. `do-while` Loop

The `do-while` loop is similar to the `while` loop but guarantees that the block of code runs **at least once**, regardless of the condition.

```
        do {
            // Code to execute
        } while (condition);
```

Example:

```
        let num = 5;

        do {
            console.log("Number is:", num);
            num++;
        } while (num < 3);
```

Question 2: What is the difference between a whileloop and a do-whileloop?

Ans:-

The key difference between a `while` loop and a `do-while` loop in JavaScript lies in **when the condition is checked** and **how many times the loop body is executed**:

Aspect	While Loop	Do-While Loop
Condition Check	Beginning	End
Minimum Executions	0	1
Syntax(C-style)	While (condition){...}	Do {...} while (condition);
Variable initialization	Variables are initialised before the execution of the loop.	Variables may initialise before or within the loop.
Exit Control	Pre-test	Post-test
Execution Guarantee	No guarantee	At least once
Semicolon	Not required after }	Required after condition
Initialization	Before loop	Can be within the loop body
Use Case	When you need to check the condition	When you want to ensure at least one execution

Functions

Question 1: What are functions in JavaScript? Explain the syntax for declaring and calling afunction.

Ans:-

Functions in JavaScript are reusable blocks of code designed to perform a specific task. They help organize and modularize code, making it more readable, reusable, and easier to debug.

Declaring a Function

To declare a function in JavaScript, you can use the `function` keyword. Here's the syntax.

```
function functionName(parameters) {
    // Code to be executed
}
```

- **`function`**: The keyword used to declare a function.
- **`functionName`**: A unique identifier for the function (optional for anonymous functions).
- **`parameters`**: Input values passed to the function (optional).
- **`{ ... }`**: The block of code that runs when the function is called.

Calling a Function

To execute a function, you "call" it by using its name followed by parentheses, optionally passing arguments.

```
        functionName(arguments);
```

Question 2: What is the difference between a function declaration and a function expression?

Ans:-

Feature	Function Declaration	Function Expression
Definition	Uses the function keyword with a name.	Function is assigned to a variable.
Name Requirement	Must have a name.	Can be named or anonymous.
Hoisting	Hoisted; can be called before definition.	Not histed; must be defined before use.
When Used	For general, reusable functions.	For more dynamic, inline, or anonymous use.

Question 3: Discuss the concept of parameters and return values in functions.

Ans:-

In JavaScript, **parameters** and **return values** are essential components of functions that enable them to process input and produce output.

1. Parameters

Parameters are variables defined in the function declaration that act as placeholders for the values (arguments) passed to the function when it is called.

- They enable a function to operate on dynamic input.
- Parameters can have default values.

```
function functionName(parameter1, parameter2) {
    // Use parameters inside the function
}
```

2. Return Values

Return values are the output a function sends back to the caller using the `return` statement.

- If a function doesn't explicitly return a value, it returns `undefined` by default.
- The `return` statement ends the execution of the function.

```
function functionName(parameters) {
    // Perform operations
    return result;
}
```

Arrays

Question 1: What is an array in JavaScript? How do you declare and initialize an array?

Ans:-

An **array** in JavaScript is a special type of object used to store a collection of values in a single variable. Arrays can hold multiple values of different data types, such as numbers, strings, objects, or even other arrays.

Arrays are ordered, indexed collections, meaning each item in the array is assigned a numerical index starting from `0`.

- **Declaring and Initializing an Array**

This is the most common and recommended way to create an array. You define the array by enclosing its elements in square brackets (`[]`), separating them by commas.

Syntax:

```
let arrayName = [element1, element2, element3, ...];
```

Example:

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits); // Output: ["Apple", "Banana", "Cherry"]
```

- **Accessing Elements in an Array**

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits[0]); // Output: Apple
```

```
    console.log(fruits[2]); // Output: Cherry
```

Question 2: Explain the methods push(), pop(), shift(), and unshift()used in arrays.

Ans:-

These four methods are commonly used to manipulate arrays by adding or removing elements at the ends of the array. Here's a detailed explanation of each.

1. `push()` – Add Elements to the End of an Array

The `push()` method adds one or more elements to the **end** of an array and returns the new length of the array.

Syntax:

```
  array.push(element1, element2, ..., elementN);
```

2. `pop()` – Remove the Last Element of an Array

The `pop()` method removes the **last** element from an array and returns that element. It changes the length of the array.

Syntax:

```
    let removedElement = array.pop();
```

3. `shift()` – Remove the First Element of an Array

The `shift()` method removes the **first** element from an array and returns that element. It also shifts the indices of the remaining elements to the left, so the array shrinks by one.

Syntax:

```
        let removedElement = array.shift();
```

4. `unshift()` – Add Elements to the Beginning of an Array

The `unshift()` method adds one or more elements to the **beginning** of an array and returns the new length of the array.

Syntax:

```
        let newLength = array.unshift(element1, element2, ..., elementN);
```

Objects

Question 1: What is an object in JavaScript? How are objects different from arrays?

Ans:-

An **object** in JavaScript is a complex data type used to store collections of data in the form of key-value pairs (also known as properties). Objects can store a wide range of data types, including other objects, arrays, or even functions.

Objects vs Arrays in JavaScript

Both **objects** and **arrays** are used to store collections of data in JavaScript, but they have distinct characteristics, and their use cases differ. Here's a comparison of the two.

Feature	Array	Object
Type	Ordered collection of values	Unordered collection of key-value pairs
Access	Indexed by numeric values(0,1,2, ...)	Accessed by string or symbol keys
Syntax	Let arr = [value1, value2, ...]	Let obj = {key1: value1, key2: value2}
Best Use	For ordered data and sequences	For structured data with key-value pairs
Methods	push(),pop(),shift(),map(),etc.	Object.keys(),object.values(), etc.
Example	Let arr = [1,2,3]	Let person = {name:"John" ,age:30}

Question 2: Explain how to access and update object properties using dot notation andbracket notation.

Ans:-

In JavaScript, you can access and update the properties of an object using two notations: **dot notation** and **bracket notation**. Both methods allow you to interact with an object's properties, but they have different use cases and syntax.

1. Dot Notation

Dot notation is the most common and straightforward way to access and update object properties. It uses a period (`.`) followed by the property name.

- **Accessing Properties with Dot Notation**

To access a property, you simply write the object name, followed by a dot (`.`), and the property name.

Syntax:

```
objectName.propertyName
```

Example:

```
let person = {
    name: "Alice",
    age: 25,
    city: "New York"
};
```

```
console.log(person.name);  // Output: "Alice"
console.log(person.age);   // Output: 25
```

- **Updating Properties with Dot Notation**

You can also update the value of an object property using dot notation by assigning a new value to it.

Syntax:

```
objectName.propertyName = newValue;
```

Example:

```
person.age = 26;  // Update the age property
console.log(person.age);  // Output: 26
```

2. Bracket Notation

Bracket notation is a more flexible way to access and update object properties. It uses square brackets ([]) and requires the property name to be enclosed in quotes (strings) or a variable holding the key.

- **Accessing Properties with Bracket Notation**

To access a property, you place the property name inside square brackets.

Syntax:

```
objectName["propertyName"]
```

Example:

```
console.log(person["city"]);  // Output: "New York"
```

- **Updating Properties with Bracket Notation**

You can update an object property using bracket notation by assigning a new value to the property name inside the square brackets.

Syntax:

```
objectName["propertyName"] = newValue;
```

Example:

```
person["city"] = "Los Angeles";  // Update the city property
console.log(person.city);  // Output: "Los Angeles"
```

JavaScript Events

Question 1: What are JavaScript events? Explain the role of event listeners.

Ans:-

In JavaScript, **events** are actions or occurrences that happen in the browser, which you can respond to or handle using JavaScript. These events can be triggered by user interactions (such as clicking a button, pressing a key, or moving the mouse) or other occurrences (like the page loading or resizing the window). Events are central to creating interactive and dynamic web applications.

1. **User Interaction Events:**
   - **`click`**: Triggered when a user clicks on an element.
   - **`keydown`**: Triggered when a key is pressed down.
   - **`keyup`**: Triggered when a key is released.
   - **`mouseenter`**: Triggered when the mouse pointer enters an element.
   - **`mouseleave`**: Triggered when the mouse pointer leaves an element.
2. **Form Events**:
   - **`submit`**: Triggered when a form is submitted.
   - **`change`**: Triggered when the value of an element (like input or select) changes.
   - **`input`**: Triggered when a user types in an input field.
3. **Window Events**:

    - `load`: Triggered when the page and all its resources (images, scripts, etc.) are fully loaded.
    - `resize`: Triggered when the window is resized.
    - `scroll`: Triggered when the page or element is scrolled.
4. **Mouse Events**:
    - `mousedown`: Triggered when a mouse button is pressed.
    - `mouseup`: Triggered when a mouse button is released.
    - `mousemove`: Triggered when the mouse moves within an element.
5. **Error and Miscellaneous Events**:
    - `error`: Triggered when an error occurs during page loading (e.g., missing images).
    - `unload`: Triggered when the page is being unloaded (e.g., navigating away from the page).

Role of Event Listeners

An **event listener** is a function that waits for an event to occur on an element, and then executes a callback function when that event occurs. Event listeners are used to handle and respond to user interactions with the page, making the page dynamic and interactive.

Question 2: How does the addEventListener()method work in JavaScript? Provide an example.

Ans:-

Event Listener Syntax:

```
element.addEventListener("event", function, useCapture);
```

- **`element`**: The HTML element you want to attach the event listener to.
- **`event`**: The type of event you want to listen for (e.g., `click`, `keydown`, etc.).
- **`function`**: The function that will run when the event is triggered. This is called a **callback function**.
- **`useCapture`**: (Optional) A boolean indicating whether the event should be captured in the **capturing phase** (`true`) or the **bubbling phase** (`false`). Default is `false`, which means the event bubbles up.

Example:

```
let button = document.getElementById("myButton");

button.addEventListener("click", function () {
    alert("Button clicked!");
});
```

In this example, the event listener waits for a `click` event on the button. When the button is clicked, the callback function (`alert("Button clicked!")`) is executed.

DOM Manipulation

Question 1: What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?

Ans:-

The **DOM (Document Object Model)** is a programming interface for web documents. It represents the structure of an HTML or XML document as a tree of objects, where each object corresponds to a part of the document, such as an element, an attribute, or text content. The DOM allows JavaScript to interact with the content and structure of a webpage.

- The DOM treats the document as a tree of nodes, with each node representing a part of the document (such as elements, text, attributes).
- It provides a way for JavaScript (or other programming languages) to access and manipulate the document's structure, style, and content dynamically.

How JavaScript Interacts with the DOM

JavaScript interacts with the DOM to manipulate the content and structure of a webpage. This can include actions like changing text, adding or removing elements, handling user interactions, or modifying styles.

Key DOM Methods and Properties for JavaScript Interaction

1. **Selecting Elements**: JavaScript can select elements in the DOM using various methods, such as `getElementById()`, `getElementsByClassName()`, or `querySelector()`. These methods return references to DOM elements.
2. **Modifying Elements**: JavaScript can modify the content or attributes of selected elements.
   - **Text Content**: Change the text content of an element.
   - **Inner HTML**: Change the HTML content inside an element.
3. **Adding/Removing Elements:** JavaScript can add, remove, or modify elements in the DOM.
   - **Creating New Elements**: Use `createElement()` to create a new element.
   - **Removing Elements**: Use `removeChild()` to remove an element.

4. **Event Handling**: JavaScript can attach event listeners to DOM elements to handle user interactions (e.g., clicks, key presses).

5. **Modifying Styles**: JavaScript can modify the style of DOM elements directly by accessing their `style` property.

6. **Traversing the DOM:** JavaScript can navigate through the DOM tree using various methods like `parentNode`, `childNodes`, `firstChild`, `lastChild`, and `nextSibling`.

Question 2: Explain the methods getElementById(), getElementsByClassName(),and querySelector()used to select elements from the DOM.

Ans:-

1. `getElementById()`

The `getElementById()` method is used to select an element by its unique `id` attribute. Since `id` values should be unique within a page, `getElementById()` will return only one element or `null` if no element with the specified `id` is found.

Syntax:

```
        let element = document.getElementById("id");
```

- **`id`**: The `id` attribute of the element you want to select.
- **Returns**: A single element (the first element with the specified `id`) or `null` if no element with that `id` is found.

2. `getElementsByClassName()`

The `getElementsByClassName()` method is used to select all elements that have a specific class name. This method returns a

live HTMLCollection of all matching elements. Since it returns a collection, you can access each element in the collection by using an index (just like an array).

Syntax:

```
let elements = document.getElementsByClassName("className");
```

- **`className`**: The class name of the elements you want to select. It can be a single class or a space-separated list of classes.
- **Returns**: A **live HTMLCollection** of elements with the specified class name.

3. `querySelector()`

The `querySelector()` method allows you to select the first element that matches a CSS selector. This is a more flexible and powerful method than `getElementById()` or `getElementsByClassName()` because it supports any valid CSS selector (including classes, IDs, tags, and complex selectors).

Syntax:

```
let element = document.querySelector("selector");
```

- **`selector`**: A string representing a CSS selector (e.g., `.className`, `#id`, `div`, `div > p`, etc.).
- **Returns**: The first element that matches the specified selector, or `null` if no element matches.

JavaScript Timing Events (setTimeout, setInterval)

Question 1: Explain the setTimeout() and setInterval() functions in JavaScript. How are they used for timing events?

Ans:-

Both **`setTimeout()`** and **`setInterval()`** are used to schedule and manage timed events in JavaScript. They allow you to delay the execution of a function or run a function repeatedly after a certain interval.

1. `setTimeout()`

The **`setTimeout()`** function is used to execute a function **once** after a specified delay (in milliseconds).

Syntax:

```
setTimeout(function, delay);
```

- **`function`**: The function to execute after the specified delay.
- **`delay`**: The time, in milliseconds, to wait before executing the function.

2. `setInterval()`

The **`setInterval()`** function is used to execute a function **repeatedly** at a specified interval (in milliseconds).

Syntax:

```
        setInterval(function, interval);
```

- **`function`**: The function to execute repeatedly.
- **`interval`**: The time, in milliseconds, between each execution of the function.

1. `setTimeout()` for Timing Events

`setTimeout()` is used when you want to delay the execution of a function by a specified amount of time. This is often useful for events that need to occur after a delay, such as showing a message after a user action, performing a delayed animation, or creating time-based transitions.

2. `setInterval()` for Timing Events

`setInterval()` is used when you want to execute a function **repeatedly** at regular intervals. This is ideal for cases where you need something to occur repeatedly after a fixed time period, such as updating a clock, animating elements, or checking for new data from a server.

Question 2: Provide an example of how to use setTimeout()to delay an action by 2 seconds.

Ans:-

Example: Delaying an Action by 2 Seconds

```
setTimeout(function () {
    console.log("This message appears after 2 seconds.");
}, 2000);  // 2000 milliseconds = 2 seconds

console.log("This message is shown immediately.");
```

Explanation:

1. **First `console.log`**: The first message is logged immediately.
2. **`setTimeout()`**: The function inside `setTimeout()` will execute after 2 seconds (2000 milliseconds), printing "This message appears after 2 seconds."
3. **Second `console.log`**: This message is shown immediately, before the delayed action.

JavaScript Error Handling

Question 1: What is error handling in JavaScript? Explain the try, catch, and finally blocks with an example.

Ans:-

Error handling in JavaScript is a mechanism that allows you to deal with runtime errors in a way that prevents the program from crashing. It lets you catch and respond to errors, ensuring that the program can continue running even when something goes wrong. JavaScript provides a `try...catch` statement for handling errors and an optional `finally` block for code that should run regardless of whether an error occurs.

1. **`try` block**: This block contains the code that might throw an error. If an error occurs in the `try` block, it is immediately passed to the `catch` block.

2. **`catch` block**: This block is executed if an error is thrown in the `try` block. It allows you to handle the error (e.g., logging the error or displaying a user-friendly message).

3. **`finally` block**: This block contains code that is always executed, regardless of whether an error occurred in the `try` block or not. It's useful for cleanup operations, like closing files or network connections.

Example of Error Handling:

```
try {
            // Code that might throw an error
            let result = 10 / 0; // Division by zero
            if (result === Infinity) {
                throw new Error("Division by zero is not allowed!");
            }
            console.log("Result: " + result);
        } catch (error) {
            // Handling the error
            console.error("Error occurred: " + error.message);  // Logs the
error message
        } finally {
            // This code will run regardless of whether there was an error or
not
            console.log("Finally block executed. Clean-up operations.");
        }
```

Question 2: Why is error handling important in JavaScript applications?

Ans:-

Without proper error handling, an error that occurs anywhere in the code can bring down the entire application. JavaScript runs in a single thread, so an uncaught exception (error) can cause the program to stop running, which may result in a poor user experience. Error handling ensures that your application can recover from errors and continue functioning.

1. Prevents Application Crashes.

2. Provides a Better User Experience.

3. Helps Identify and Debug Issues.

4. Improves Code Maintainability.

5. Enables Graceful Error Recovery.

6. Prevents Unhandled Promise Rejections.

7. Allows Fine-Grained Control Over Execution Flow.

Properly handling errors makes your code more resilient, user-friendly, and easier to maintain, and it ensures that your application can deal with real-world challenges like network failures, invalid inputs, and unexpected conditions.