

# Module 6 – Mernstack – Javascript Essential and Advanced

---

## JavaScript Introduction:

### Theory Assignment:

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

### What is JavaScript?

JavaScript is a **high-level, interpreted scripting language** used to make web pages interactive. It runs in the browser and allows dynamic content updates, event handling, and animations.

### Role of JavaScript in Web Development

1. **Enhances Interactivity** – Handles user actions like clicks, form submissions, and animations.
2. **DOM Manipulation** – Modifies HTML and CSS dynamically.
3. **Form Validation** – Validates user input before sending data to the server.
4. **Asynchronous Communication** – Uses AJAX and Fetch API for background data fetching.
5. **Supports Frameworks & Libraries** – Works with React, Angular, and Vue.js for scalable applications.
6. **Server-Side Development** – Used with Node.js to build APIs and backend services.

JavaScript is essential for creating **dynamic, interactive, and efficient** web applications.

## Question 2: How is JavaScript different from other programming languages like Python or Java?

Difference Between JavaScript, Python, and Java:

Feature	JavaScript	Python	Java
Type	Scripting (interpreted)	General-purpose (interpreted)	General-purpose (compiled)
Use Case	Web development (frontend & backend)	Data science, AI, web, automation	Enterprise applications, mobile, web
Execution	Runs in browser (or Node.js for server-side)	Runs on Python interpreter	Runs on JVM (Java Virtual Machine)
Syntax	Loosely typed, dynamic	Readable, indentation-based	Strictly typed, verbose
Speed	Fast in browsers, optimized for web	Slower than Java, but efficient	Faster than JS & Python, optimized for performance
Concurrency	Single-threaded (event-driven)	Multi-threaded (supports multiprocessing)	Multi-threaded (built-in support)
Learning Curve	Easy for web development beginners	Beginner-friendly	Moderate to steep

### Key Differences

- **JavaScript is mainly used for web development, while Python and Java are general-purpose languages.**
- **Java is statically typed and compiled, whereas JavaScript and Python are dynamically typed and interpreted.**
- **Python is best for AI, ML, and automation, while JavaScript is ideal for interactive web applications.**

Each language is powerful in its domain, with JavaScript excelling in **web interactivity and full-stack development**.

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

## **Use of <script> Tag in HTML**

The <script> tag is used in HTML to embed or reference JavaScript code. It allows adding interactivity, handling events, and modifying the webpage dynamically.

### *Ways to Use the <script> Tag*

#### **1. Inline JavaScript (Within HTML)**

```
<script>
    alert("Hello, JavaScript!");
</script>
```

- Code is written directly inside the <script> tag within the HTML file.

#### **2. Internal JavaScript (Inside <script> in <head> or <body>)**

```
<script>
    function greet() {
        document.getElementById("demo").innerHTML = "Hello, World!";
    }
</script>
```

- Placed inside the <head> or before the closing <body> tag.

#### **3. External JavaScript (Recommended Approach)**

- JavaScript is stored in a separate .js file and linked to HTML.

## Linking an External JavaScript File

To link an external JavaScript file, use the `<script>` tag with the `src` attribute:

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

This should be placed **before the closing `<body>` tag** for better performance.

### Example:

#### *HTML File (`index.html`)*

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

#### *JavaScript File (`script.js`)*

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

## Advantages of Using External JavaScript

- **Code Reusability** – The same script can be used across multiple pages.
- **Better Organization** – Separates HTML and JavaScript for cleaner code.
- **Improved Performance** – Browser caches external JS files for faster loading.

Using external JavaScript is a **best practice** for maintainability and efficiency in web development.

## Variables and Data Types:

### Theory Assignment:

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

### What are Variables in JavaScript?

Variables store data in JavaScript and are declared using **var**, **let**, or **const**.

### Declaring Variables

1. **var** – Function-scoped, can be redeclared and updated.

```
var x = 10;  
x = 20; // Allowed
```

2. **let** – Block-scoped, can be updated but not redeclared.

```
let y = 15;
y = 25; // Allowed
```

3. **const** – Block-scoped, cannot be updated or redeclared.

```
const z = 50;
// z = 60; ❌ Error
```

### Key Differences

Feature	var	let	const
Scope	Function	Block	Block
Redeclare	✅ Yes	❌ No	❌ No
Reassign	✅ Yes	✅ Yes	❌ No
Hoisting	✅ Yes (undefined)	✅ Yes (not initialized)	✅ Yes (not initialized)

**Best Practice:** Use `let` for changeable values and `const` for constants. Avoid `var`.

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

JavaScript has **8 data types**, categorized into **Primitive** and **Non-Primitive**

#### 1. Primitive Data Types (Immutable, stored by value)

**Number** – Represents numeric values.

```
let num = 25;
```

**String** – Represents text values

```
let str = "Hello";
```

**Boolean** – Represents true/false values

```
let isTrue = true;
```

**Undefined** – A variable declared but not assigned a value.

```
let x;  
console.log(x); // undefined
```

**Null** – Represents an empty or unknown value.

```
let y = null;
```

**BigInt** – Represents large integers beyond Number limits.

```
let bigNum = 12345678901234567890n;
```

**Symbol** – Represents unique, immutable values.

```
let sym = Symbol("id");
```

## 2. Non-Primitive Data Type (Mutable, stored by reference)

**Object** – Collection of key-value pairs.

```
let person = {  
  name: "Dev",  
  age: 20  
};
```

### Question 3: What is the difference between undefined and null in JavaScript?

#### Difference between `undefined` and `null` in JavaScript

Feature	<code>undefined</code>	<code>null</code>
Meaning	A variable is declared but not assigned a value.	Represents an intentional empty or unknown value.
Type	<code>undefined</code> (primitive)	<code>object</code> (primitive, but a known JavaScript bug)
Usage	Default value of uninitialized variables.	Used to explicitly indicate "no value".
Example		

```
//Undefined
let x;
console.log(x);

//Null
let y = null;
console.log(y);
```

#### Key Point:

- **JavaScript assigns `undefined` automatically** when a variable is declared but not assigned.
- **`null` is manually assigned** to indicate "no value".

## JavaScript Operators:

### Theory Assignment:

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



- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators

## Types of Operators in JavaScript

### 1. Arithmetic Operators (Perform mathematical calculations)

Operator	Description	Example ( a = 10, b = 5 )	Output
+	Addition	a + b	15
-	Subtraction	a - b	5
*	Multiplication	a * b	50
/	Division	a / b	2
%	Modulus (Remainder)	a % b	0
**	Exponentiation	a ** b	100000

### 2. Assignment Operators (Assign values to variables)

Operator	Example	Equivalent To	Output ( a = 10 )
=	a = b	a = 5	5
+=	a += b	a = a + b	15
-=	a -= b	a = a - b	5
*=	a *= b	a = a * b	50
/=	a /= b	a = a / b	2

### 3. Comparison Operators (Compare two values, return true or false)

Operator	Description	Example ( a = 10, b = 5 )	Output
<code>==</code>	Equal to	<code>a == b</code>	false
<code>!=</code>	Not equal to	<code>a != b</code>	true
<code>===</code>	Strict equal (checks type + value)	<code>a === "10"</code>	false
<code>!==</code>	Strict not equal	<code>a !== 10</code>	false
<code>&gt;</code>	Greater than	<code>a &gt; b</code>	true
<code>&lt;</code>	Less than	<code>a &lt; b</code>	false
<code>&gt;=</code>	Greater than or equal to	<code>a &gt;= 10</code>	true
<code>&lt;=</code>	Less than or equal to	<code>b &lt;= 5</code>	true

#### 4. Logical Operators (Used with boolean values)

Operator	Description	Example ( x = true, y = false )	Output
<code>&amp;&amp;</code> (AND)	Returns <code>true</code> if both are <code>true</code>	<code>x &amp;&amp; y</code>	false
<code> </code>		<code> </code> (OR)	Returns <code>true</code> if at least one is <code>true</code>
<code>!</code> (NOT)	Reverses boolean value	<code>!x</code>	false

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

Difference between `==` and `===` in JavaScript

Operator	Name	Comparison Type	Example ( x = 10 , y = "10" )	Output
<code>==</code>	Loose Equality	Compares values only (type conversion happens)	<code>x == y</code>	true
<code>===</code>	Strict Equality	Compares both values and types (no conversion)	<code>x === y</code>	false

**Example:**

```
js                                                                    Copy Edit
console.log(10 == "10"); // true (type conversion happens)
console.log(10 === "10"); // false (type mismatch)
```

**Key Point:**

- Use `==` when type conversion is acceptable.
- Use `===` for precise comparison (recommended for cleaner code).

## Control Flow (If-Else, Switch):

**Theory Assignment:**

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

**Control Flow in JavaScript:**

Control flow determines the order in which statements are executed in a program. JavaScript executes code **line by line**, but we can control execution using **conditional statements** like `if-else`.

**How if-else Works**

- **if Statement:** Executes a block of code **if** the condition is `true`.
- **else Statement:** Executes a block of code **if** the condition is `false`.
- **else if Statement:** Checks multiple conditions.

Example:

```
let num = 10;

if (num > 10) {
    console.log("Number is greater than 10");
} else if (num == 10) {
    console.log("Number is exactly 10");
} else {
    console.log("Number is less than 10");
}
```

OUTPUT:

```
Number is exactly 10
```

### Key Points:

- Only one block executes based on the condition.
- `else if` allows multiple conditions to be checked in order.
- Use `else` as a default case if none of the conditions match.

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

### How `switch` Statements Work in JavaScript

A `switch` statement evaluates an expression and matches it against multiple `case` values. When a match is found, the corresponding block executes. If no match is found, the `default` block runs.

### Example:

```
let day = "Monday";

switch (day) {
  case "Monday":
    console.log("Start of the week!");
    break;
  case "Friday":
    console.log("Weekend is near!");
    break;
  default:
    console.log("It's a regular day.");
}
```

Start of the week!

### When to Use `switch` Instead of `if-else`?

- Use `switch` for multiple fixed values (e.g., days of the week, user roles, menu options).
- Use `if-else` for range-based conditions (e.g.,  $x > 10, y < 5$ ).
- `switch` is **cleaner and more readable** than multiple `if-else` statements when checking **multiple exact values**.

## Loops (For, While, Do-While):

### Theory Assignment:

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

## Types of Loops in JavaScript

Loops are used to execute a block of code multiple times. JavaScript has three main types of loops:

### 1. **for Loop** (Used when the number of iterations is known)

Example:

```
js
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

[Copy](#)[Edit](#)

Output:

```
1 2 3 4 5
```

[Copy](#)[Edit](#)

### 2. **while Loop** (Used when the number of iterations is unknown)

Example:

```
js
let i = 1;
while (i <= 5) {
  console.log(i);
  i++;
}
```

[Copy](#)[Edit](#)

Output:

```
1 2 3 4 5
```

[Copy](#)[Edit](#)

### 3. **do-while Loop** (Executes the code block **at least once**, even if the condition is false)

Example:

```
js
let i = 1;
do {
  console.log(i);
  i++;
} while (i <= 5);
```

[Copy](#)[Edit](#)

Output:

```
1 2 3 4 5
```

[Copy](#)[Edit](#)

### Key Differences:

- **for**: Best when the number of iterations is known.
- **while**: Best when the condition depends on external factors.
- **do-while**: Ensures the loop runs at least once.

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

## Difference Between `while` and `do-while` Loops in JavaScript

Feature	<code>while</code> Loop	<code>do-while</code> Loop
Execution	Checks condition <b>before</b> executing the loop body.	Executes the loop body <b>at least once</b> , then checks the condition.
Condition Check	If the condition is <code>false</code> initially, the loop <b>does not run</b> .	Runs <b>once</b> , even if the condition is <code>false</code> .
Use Case	Used when the condition is checked <b>before</b> execution.	Used when at least <b>one execution</b> is required before checking.

## Functions:

### Theory Assignment:

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

### Functions in JavaScript

A **function** in JavaScript is a reusable block of code designed to perform a specific task. Functions help in making code modular and avoid repetition.



**Example:**

```
js                                                                    Copy Edit

// Function to add two numbers
function add(a, b) {
    return a + b;
}

// Calling the function
let result = add(5, 3);
console.log("Sum:", result);
```

**Output:**

```
makefile                                                                Copy Edit

Sum: 8
```

**Key Points:**

- Functions are declared using the `function` keyword.
- Parameters act as placeholders for values passed during the function call.
- The `return` statement gives back a value (optional).
- Functions must be **called** to execute their code.

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

## Difference Between Function Declaration and Function Expression in JavaScript

Feature	Function Declaration	Function Expression
Definition	Uses the <code>function</code> keyword with a function name.	Assigns a function to a variable.
Hoisting	<b>Hoisted</b> (can be called before declaration).	<b>Not hoisted</b> (cannot be called before declaration).
Usage	Best for defining reusable functions.	Useful for anonymous functions and dynamic function assignments.

### Key Takeaways:

- **Function Declarations** are hoisted, so they can be called before being defined.
- **Function Expressions** are not hoisted, so they must be defined before use.
- Function expressions are useful for **callback functions** and **storing functions in variables**.

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

Parameters and Return Values in Functions:

### *1. Parameters in Functions*

- **Parameters** are variables passed into a function to provide input.
- They allow functions to work dynamically with different values.

Example:

```
js                                                                    Copy Edit

function greet(name) {
  console.log("Hello, " + name + "!");
}

greet("Alice"); // Output: Hello, Alice!
greet("Bob");   // Output: Hello, Bob!
```

## 2. Return Values in Functions

- The `return` statement **sends a value back** to the caller.
- If a function doesn't have `return`, it returns `undefined` by default.

Example:

```
js                                                                    Copy Edit

function add(a, b) {
  return a + b;
}

let sum = add(5, 3);
console.log(sum); // Output: 8
```

### Key Takeaways:

- **Parameters** allow functions to accept input values.
- **Return values** send output back from a function.
- Functions without `return` just execute code but don't provide a value.

## Arrays:

### Theory Assignment:

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

### What is an Array in JavaScript?

An **array** in JavaScript is a special variable that can store multiple values in a **single variable**. Arrays allow easy storage and manipulation of collections of data.

### Declaring and Initializing an Array

#### 1. Using Array Literal (Recommended)

```
let fruits = ["Apple", "Banana", "Cherry"];
```

#### 2. Using new Array() Constructor

```
let numbers = new Array(1, 2, 3, 4, 5);
```

#### 3. Empty Array and Adding Values Later

```
let colors = [];  
colors[0] = "Red";  
colors[1] = "Blue";
```

## Example Usage

```
js

let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits[0]); // Output: Apple
console.log(fruits.length); // Output: 3
```

### Key Takeaways:

- Arrays store **multiple values** in a single variable.
- Values in an array are accessed using **index numbers** (starting from 0).
- The `.length` property gives the number of elements in an array.

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

### Array Methods in JavaScript

These methods help in adding and removing elements from an array:

Method	Action	Modifies Original Array?	Returns
<code>push()</code>	Adds an element to the <b>end</b>	✓ Yes	New length
<code>pop()</code>	Removes the <b>last</b> element	✓ Yes	Removed element
<code>shift()</code>	Removes the <b>first</b> element	✓ Yes	Removed element
<code>unshift()</code>	Adds an element to the <b>beginning</b>	✓ Yes	New length

```

let fruits = ["Apple", "Banana", "Cherry"];

// push() - Adds "Orange" at the end
fruits.push("Orange");
console.log(fruits); // ["Apple", "Banana", "Cherry", "Orange"]

// pop() - Removes last element "Orange"
let removed = fruits.pop();
console.log(fruits); // ["Apple", "Banana", "Cherry"]
console.log(removed); // "Orange"

// shift() - Removes first element "Apple"
removed = fruits.shift();
console.log(fruits); // ["Banana", "Cherry"]
console.log(removed); // "Apple"

// unshift() - Adds "Grapes" at the beginning
fruits.unshift("Grapes");
console.log(fruits); // ["Grapes", "Banana", "Cherry"]

```

```

[ 'Apple', 'Banana', 'Cherry', 'Orange' ]
[ 'Apple', 'Banana', 'Cherry' ]
Orange
[ 'Banana', 'Cherry' ]
Apple
[ 'Grapes', 'Banana', 'Cherry' ]

```

## Key Takeaways:

- `push()` and `unshift()` **add** elements.
- `pop()` and `shift()` **remove** elements.
- They **modify the original array**.

# Objects:

## Theory Assignment:

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

### What is an Object in JavaScript?

An **object** in JavaScript is a collection of key-value pairs where each key (or property) has an associated value. Objects allow you to store **multiple related data types** in a structured way.

#### Example of an Object:

```
js                                                                    Copy Edit

let person = {
  name: "John",
  age: 25,
  isStudent: false
};

console.log(person.name); // Output: John
console.log(person["age"]); // Output: 25
```

### Difference Between Objects and Arrays

Feature	Objects	Arrays
Data Structure	Key-value pairs ( { key: value } )	Indexed list ( [value1, value2] )
Access Method	Accessed using <code>obj.key</code> or <code>obj["key"]</code>	Accessed using <code>array[index]</code>
Use Case	Best for storing <b>structured data</b>	Best for storing <b>lists of data</b>
Ordering	<b>Unordered</b> collection of properties	<b>Ordered</b> collection of elements

## Example of Array vs. Object

```
js                                                                    Copy Edit

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

// Object (Descriptive data)
let fruit = { name: "Apple", color: "Red" };
console.log(fruit.name); // Output: Apple
```

## Key Takeaways:

- **Objects** store data in **key-value pairs**, while **arrays** store **ordered** lists.
- **Objects** are best for representing **real-world entities** with multiple properties.
- **Arrays** are best for **storing collections** of similar data types

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

## Accessing and Updating Object Properties in JavaScript

You can access and update object properties using **dot notation** (.) or **bracket notation** ([]).

### 1. Dot Notation (object.property)

- **Simpler and more readable.**
- **Used when the property name is a valid identifier** (no spaces, special characters, or numbers at the start).



## Example: Access & Update Using Dot Notation

```
js

let person = {
  name: "Alice",
  age: 25
};

// Access properties
console.log(person.name); // Output: Alice
console.log(person.age);  // Output: 25

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

## 2. Bracket Notation (`object["property"]`)

- Used when the property name has spaces, special characters, or is stored as a variable.
- More dynamic—property names can be passed as strings or variables.

## Example: Access & Update Using Bracket Notation

```
js

let person = {
  "full name": "Alice Johnson",
  age: 25
};

// Access properties
console.log(person["full name"]); // Output: Alice Johnson

// Update properties
person["age"] = 26;
console.log(person["age"]); // Output: 26

// Using a variable as a key
let key = "full name";
console.log(person[key]); // Output: Alice Johnson
```

### Key Takeaways:

Notation	Usage	When to Use
Dot ( . )	<code>obj.property</code>	When the property name is a valid identifier (no spaces/special chars).
Bracket ( [ ] )	<code>obj["property"]</code>	When the property name has spaces, special characters, or is stored in a variable.

 Dot notation is preferred, but bracket notation is useful for dynamic keys!

## JavaScript Events:

### Theory Assignment:

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

### What are JavaScript Events?

JavaScript **events** are actions or occurrences detected by the browser, such as user interactions (clicks, keypresses, mouse movements) or system actions (page load, resizing).

#### Common Events in JavaScript:

Event	Description
<code>click</code>	Triggered when an element is clicked
<code>keydown</code>	Triggered when a key is pressed
<code>mouseover</code>	Triggered when the mouse hovers over an element
<code>load</code>	Triggered when the page fully loads

### Role of Event Listeners (`addEventListener`)

Event listeners allow JavaScript to **wait for an event to occur** and then execute a function when that event happens.

#### Example: Using `addEventListener`

```
js

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

- This waits for a **click** event on the button with `id="btn"`.
- When clicked, it executes the function and shows an alert.

## Why Use Event Listeners?

- ✓ **Separates JavaScript from HTML** (cleaner code).
- ✓ **Allows multiple event handlers on the same element.**
- ✓ **More flexible than inline event attributes** (`onclick="..."`).

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

## How `addEventListener()` Works in JavaScript

The `addEventListener()` method allows you to attach an event to an element **without overwriting existing event handlers**.

- **event** → The event type (e.g., "click", "keydown").
- **function** → The function to execute when the event occurs.
- **useCapture** (*optional*) → `true` for capturing phase, `false` for bubbling (default).

### Example: Using `addEventListener()`

```
js

// Select the button
let button = document.getElementById("myButton");

// Attach a click event listener
button.addEventListener("click", function() {
    alert("Button was clicked!");
});
```

- When the button is clicked, an **alert** message appears.

- This keeps JavaScript **separate from HTML**, making the code cleaner.

## DOM Manipulation:

### Theory Assignment:

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

#### What is the DOM (Document Object Model) in JavaScript?

The **DOM (Document Object Model)** is a programming interface that represents an **HTML document as a tree structure** where each element is a node. JavaScript uses the DOM to **dynamically access, modify, and manipulate web pages**.

#### How JavaScript Interacts with the DOM

JavaScript can:

- ✓ **Select elements** (`getElementById`, `querySelector`)
- ✓ **Modify content** (`innerHTML`, `textContent`)
- ✓ **Change styles** (`style.property`)
- ✓ **Handle events** (`addEventListener`)
- ✓ **Add or remove elements** (`appendChild`, `removeChild`)

## Example: Modifying the DOM Using JavaScript

```
js

// Select an element by ID
let heading = document.getElementById("title");

// Modify the text content
heading.textContent = "Hello, DOM!";

// Change the color of the heading
heading.style.color = "blue";
```

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

### Methods to Select Elements from the DOM

JavaScript provides different methods to select HTML elements for manipulation.

Method	Description	Returns	Example
<code>getElementById()</code>	Selects an element by its <code>id</code>	A <b>single</b> element ( <code>HTMLElement</code> )	<code>document.getElementById("myId")</code>
<code>getElementsByClassName()</code>	Selects all elements with a given <code>class</code>	A <b>collection</b> ( <code>HTMLCollection</code> )	<code>document.getElementsByClassName("myClass")</code>
<code>querySelector()</code>	Selects the <b>first</b> element matching a CSS selector	A <b>single</b> element ( <code>HTMLElement</code> )	<code>document.querySelector(".myClass")</code>

## Examples

### 1. Using `getElementById()` (Select by ID)

js

 Copy Edit

```
let title = document.getElementById("header");
title.textContent = "Hello, JavaScript!";
```

✅ Best when selecting a single unique element.

### 2. Using `getElementsByClassName()` (Select by Class)

js

 Copy Edit

```
let items = document.getElementsByClassName("item");
items[0].style.color = "red"; // Modify the first item
```

✅ Returns a collection (like an array), so use indexing ( `[0]` ) to access elements.

### 3. Using `querySelector()` (Select First Matching Element)

js

Copy

Edit

```
let firstItem = document.querySelector(".item");  
firstItem.style.fontWeight = "bold";
```

✓ More flexible, allows CSS selectors ( `#id`, `.class`, `tag` ).

## Key Takeaways

- Use `getElementById()` for unique elements.
- Use `getElementsByClassName()` when selecting multiple elements.
- Use `querySelector()` for more flexibility with CSS selectors.

## JavaScript Timing Events (`setTimeout`, `setInterval`):

### Theory Assignment:

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

### `setTimeout()` and `setInterval()` in JavaScript

Both functions are used for **timing events** in JavaScript but work differently.

#### 1. `setTimeout()` (Execute After a Delay)

- Executes a function **once** after a specified delay (in milliseconds).



**Example:** (Run code after 3 seconds)

```
js                                                                    Copy Edit

setTimeout(() => {
  console.log("Hello after 3 seconds!");
}, 3000);
```

✅ Use when you need a delayed execution.

## 2. `setInterval()` (Execute Repeatedly)

- Executes a function **repeatedly** at a fixed time interval.

**Example:** (Run code every 2 seconds)

```
js                                                                    Copy Edit

setInterval(() => {
  console.log("This prints every 2 seconds!");
}, 2000);
```

✅ Use when you need to repeat a task continuously.

## Stopping the Timers

- `clearTimeout(timerID)` → Stops a `setTimeout()`.
- `clearInterval(timerID)` → Stops a `setInterval()`.

**Example: Stopping** `setInterval()`

```
js                                                                    Copy Edit

let counter = setInterval(() => console.log("Repeating..."), 1000);

setTimeout(() => {
  clearInterval(counter); // Stops after 5 seconds
  console.log("Stopped!");
}, 5000);
```

**Key Differences**

Function	Purpose	Executes	Stops With
<code>setTimeout()</code>	Delay execution	Once after delay	<code>clearTimeout()</code>
<code>setInterval()</code>	Repeat execution	At fixed intervals	<code>clearInterval()</code>

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

**Example: Using** `setTimeout()` **to Delay an Action by 2 Seconds**

```
js                                                                    Copy Edit

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

✅ After 2 seconds (2000ms), "This message appears after 2 seconds!" will be printed to the console.

**Example with an Alert**

```
js                                                                    Copy Edit

setTimeout(() => {
  alert("Hello! This alert appears after 2 seconds.");
}, 2000);
```

✅ Displays an alert box after 2 seconds.

# JavaScript Error Handling:

## Theory Assignment:

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

### Error Handling in JavaScript

Error handling in JavaScript allows developers to **catch and handle errors gracefully** instead of crashing the program. The `try...catch...finally` blocks help in managing errors.

#### 1. **try** Block

- ☐ Contains the code that may throw an error.

#### 2. **catch** Block

- ☐ Handles the error if one occurs inside `try`.

#### 3. **finally** Block (*Optional*)

- ☐ Executes **always**, regardless of whether an error occurred or not.

## Example: Handling an Error Using `try...catch...finally`

```
js

try {
  let result = 10 / x; // x is not defined (causes an error)
  console.log(result);
} catch (error) {
  console.log("An error occurred:", error.message);
} finally {
  console.log("Execution completed.");
}
```

### Output:

```
vbnet

An error occurred: x is not defined
Execution completed.
```

- ✓ Use `try...catch` to prevent program crashes.
- ✓ `finally` ensures important cleanup code runs (e.g., closing files, clearing memory).

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

**Why Is Error Handling Important in JavaScript Applications?**

Error handling ensures that JavaScript applications **run smoothly** even when unexpected issues occur. It improves reliability, security, and user experience.

## Key Reasons for Error Handling

- ✓ **Prevents Application Crashes** → Catches errors and prevents the program from stopping unexpectedly.
- ✓ **Improves Debugging** → Provides meaningful error messages, making it easier to find and fix issues.
- ✓ **Enhances User Experience** → Displays friendly error messages instead of breaking the app.
- ✓ **Ensures Data Integrity** → Prevents data corruption or loss due to unexpected failures.
- ✓ **Handles External Issues** → Manages errors from APIs, databases, or network failures gracefully

## Example: Handling API Errors

```
js

try {
  let response = fetch("https://invalid-url.com");
  console.log(response.data);
} catch (error) {
  console.log("Failed to fetch data:", error.message);
}
```