Module -17 : JavaScript For Full Stack Assignment

THEORY ASSIGNMENT

---

1. JavaScript Introduction:

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

JavaScript is a high-level, interpreted programming language that is one of the core technologies of the web, alongside **HTML** and **CSS**. It was originally developed to add interactive behaviour to web pages and is now used for both client-side and server-side development.

Role of JavaScript in Web Development:

1. **Client-Side Interactivity**:
    - JavaScript enables dynamic behaviour on web pages without needing to reload them.
    - Examples: dropdown menus, sliders, form validation, modals, animations.
2. **DOM Manipulation**:
    - JavaScript can access and modify the Document Object Model (DOM), allowing developers to change the structure, style, and content of a web page dynamically.
3. **Event Handling**:
    - JavaScript responds to user actions such as clicks, key presses, mouse movements, etc., enabling interactive user experiences.

4. **Asynchronous Programming**:
    - Through features like **AJAX**, **fetch API**, and **Promises**, JavaScript can load data in the background without refreshing the page.
5. **Front-End Frameworks**:
    - JavaScript powers modern frameworks and libraries like **React**, **Angular**, and **Vue.js**, which simplify building complex user interfaces.
6. **Server-Side Development**:
    - Using **Node.js**, JavaScript can be used to build scalable and fast server-side applications.
7. **Cross-Platform Applications**:
    - JavaScript can be used with technologies like **React Native** or **Electron** to build mobile and desktop apps.

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

JavaScript differs from other programming languages such as **Python** and **Java** in several key ways, including its execution environment, syntax, and primary use cases:

1. Execution Environment

- **JavaScript**:
    - Primarily runs in the **web browser** (client-side).
    - Can also run on the server using **Node.js**.
- **Python/Java**:
    - Typically run on the **server-side** or desktop applications.
    - Not natively supported in web browsers.

2. Use Case

- **JavaScript**:
    - Designed for building **interactive web pages** and **front-end** development.
- **Python**:
    - Commonly used for **data science**, **AI**, **automation**, scripting, and **backend development** (e.g., Django, Flask).
- **Java**:
    - Used for **enterprise applications**, **Android development**, and large-scale systems.

3. Syntax and Typing

- **JavaScript**:
    - **Dynamically typed** and **loosely typed**.
    - More flexible but prone to runtime errors.
- **Python**:
    - Also **dynamically typed**, but emphasizes **readability** and simplicity.
- **Java**:
    - **Statically typed** – variable types must be declared explicitly.
    - Enforces strict type-checking at compile-time.

4. Compilation vs Interpretation

- **JavaScript**:
    - Interpreted or Just-In-Time (JIT) compiled in the browser.
- **Python**:
    - Interpreted at runtime.

- **Java**:
    - Compiled into **bytecode**, which runs on the **Java Virtual Machine (JVM)**.

5. Object-Oriented Programming (OOP) Style

- **JavaScript**:
    - Uses **prototype-based** inheritance.
- **Python and Java**:
    - Use **class-based** inheritance.

6. Concurrency Model

- **JavaScript**:
    - Uses an **event-driven, single-threaded** model with **async/await** and **Promises**.
- **Python**:
    - Has threading, multiprocessing, and async capabilities.
- **Java**:
    - Strong support for multithreading and concurrency.

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

The <script> tag in HTML is used to embed or reference JavaScript code within an HTML document. JavaScript is a scripting language that enables dynamic interactions, such as responding to user input, modifying HTML/CSS content, validating forms, and more.

Purpose of the <script> Tag:

- It tells the browser to interpret and execute the enclosed JavaScript code.
- It can be used either to write inline scripts directly in the HTML or to link external JavaScript files.

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

Linking an External JavaScript File:

To include JavaScript from an external file, the <script> tag is used with the src (source) attribute. This helps separate HTML structure from JavaScript behaviour, improving maintainability and organization.

2. Variables and Data Types:

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

Variables in JavaScript are used to store data that can be referenced and manipulated in a program. They act as symbolic names for values and help in writing flexible, dynamic code.

Declaring Variables in JavaScript

JavaScript allows you to declare variables using three keywords: var, let, and const. Each has distinct characteristics regarding scope, re-declaration, reassignment, and hoisting.

var

- **Scope**: Function-scoped.
- **Re-declaration**: Allowed within the same scope.
- **Reassignment**: Allowed.
- **Hoisting**: Variables declared with var are hoisted to the top of their scope and initialized as undefined.

let

- **Scope**: Block-scoped (within {}).
- **Re-declaration**: Not allowed in the same scope.
- **Reassignment**: Allowed.
- **Hoisting**: Variables are hoisted but not initialized; accessing them before declaration causes an error.

const

- **Scope**: Block-scoped.
- **Re-declaration**: Not allowed in the same scope.
- **Reassignment**: Not allowed; the value must be assigned at declaration and cannot be changed.
- **Hoisting**: Variables are hoisted but not initialized; accessing them before declaration causes an error.

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

JavaScript supports a variety of data types, which are broadly categorized into **primitive** and **non-primitive (reference)** types:

1. Primitive Data Types

These are the most basic data types and are immutable.

a. Number

Represents both integers and floating-point numbers.

let age = 25;

let price = 99.99;

b. String

A sequence of characters enclosed in single, double, or backticks.

let name = "Alice";

let greeting = `Hello, ${name}`;

c. Boolean

Represents a logical value: true or false.

let isLoggedIn = true;

let hasAccess = false;

d. Undefined

A variable that has been declared but not assigned a value.

let result;

console.log(result); // undefined

e. Null

Represents an intentional absence of any value or object.

let user = null;

f. Symbol (ES6)

A unique and immutable value often used as object property keys.

let sym = Symbol("id");

g. BigInt (ES11)

Used to represent very large integers beyond the safe limit of Number.

let bigNumber = 1234567890123456789012345678901234567890n;

2. Non-Primitive (Reference) Data Types

These types can hold multiple values and are mutable.

a. Object

A collection of key-value pairs.

let person = {

```
  name: "John",
  age: 30
};
```

b. Array

A special type of object used to store ordered collections.

```
let colors = ["red", "green", "blue"];
```

c. Function

A block of code designed to perform a task.

```
function greet() {
  console.log("Hello");
}
```

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

In JavaScript, undefined and null are both primitive values used to represent the absence of a meaningful value, but they are used in different contexts and have distinct meanings.

1. undefined

- **Meaning**: A variable has been declared but has not been assigned any value.
- **Automatically assigned** by JavaScript to uninitialized variables.
- Also returned by functions that do not explicitly return a value.

Example:

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

2. null

- **Meaning**: An intentional assignment representing "no value" or "empty".

- **Manually assigned** by the developer to indicate that a variable should be empty.

Example:

let y = null;

3. JavaScript Operators:

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

o Arithmetic operators

o Assignment operators

o Comparison operators

o Logical operators

JavaScript provides various types of operators that are used to perform operations on variables and values. The main types include **Arithmetic**, **Assignment**, **Comparison**, and **Logical** operators.

1. Arithmetic Operators

Used to perform mathematical calculations.

Operator	Description	Example
+	Addition	5 + 3 → 8
-	Subtraction	10 - 4 → 6
*	Multiplication	6 * 2 → 12
/	Division	9 / 3 → 3

Operator	Description	Example
%	Modulus (remainder)	7 % 2 → 1
**	Exponentiation	2 ** 3 → 8
++	Increment	a++
--	Decrement	a--

2. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	x = 10
+=	Add and assign	x += 5 → x = x + 5
-=	Subtract and assign	x -= 3 → x = x - 3
*=	Multiply and assign	x *= 2
/=	Divide and assign	x /= 4
%=	Modulus and assign	x %= 2

3. Comparison Operators

Used to compare two values and return a boolean (true or false).

Operator	Description	Example
==	Equal to (loose equality)	5 == '5' → true
===	Strictly equal (type + value)	5 === '5' → false
!=	Not equal to	5 != 3 → true

Operator	Description	Example
!==	Strictly not equal	5 !== '5' → true
>	Greater than	7 > 5 → true
<	Less than	3 < 6 → true
>=	Greater than or equal to	5 >= 5 → true
<=	Less than or equal to	4 <= 5 → true

4. Logical Operators

Used to combine or invert boolean expressions.

Operator	Description	Example
&&	Logical AND	true && false → false
`		`
!	Logical NOT	!true → false

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

In JavaScript, == and === are comparison operators used to evaluate whether two values are equal. However, they differ in how they perform the comparison.

== (Loose Equality Operator)

The == operator checks for equality **after performing type conversion** if the values being compared are of different types. This means JavaScript will attempt to convert the values to a common type before making the comparison. As a result, it may return true even if the values are of different data types.

=== (Strict Equality Operator)

The === operator checks for equality **without performing any type conversion**. It returns true only if **both the value and the data type are the same**. This makes it a more precise and reliable comparison operator in most situations.

4. Control Flow (If-Else, Switch)

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

Control flow in JavaScript refers to the order in which individual statements, instructions, or function calls are executed or evaluated. By default, JavaScript runs code **from top to bottom**, **left to right**—but control flow statements can change that order.

Examples of control flow statements:

- if, else if, else
- switch
- for, while, do...while loops
- break, continue
- try...catch

How if-else Statements Work

An if-else statement is used to make decisions in code. It evaluates a condition:

- If the condition is true, the code inside the if block runs.
- If the condition is false, and there is an else block, the code inside the else block runs instead.

You can also use else if to test multiple conditions.

Example:

```
let temperature = 30;


if (temperature > 35) {
  console.log("It's very hot outside.");
} else if (temperature > 25) {
  console.log("It's warm outside.");
} else {
  console.log("It's cool outside.");
}
```

Output:

rust

It's warm outside.

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 a given expression and matches its result against multiple **case** values. Each case represents a possible match. If a match is found, the code under that case is executed. If no match is found, the default block is executed (if present).

To prevent execution from continuing into the next case, each case usually ends with a break statement.

When to Use switch Instead of if-else:

You should use a switch statement instead of if-else when:

- You are comparing **the same variable or expression** to multiple different constant values.
- You want to improve **code readability** and **organization**, especially when there are many conditions.

Use if-else instead when:

- Conditions are **not based on the same value**.
- Conditions involve **relational operators** (e.g., <, >) or **logical combinations** (e.g., &&, ||).
- You need **more complex logic** that switch cannot handle.

5. Loops (For, While, Do-While)

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

Loops are control structures used to **repeat a block of code** as long as a specified condition is true. JavaScript provides several types of loops, each suited for different scenarios.

1. for Loop

Used when the **number of iterations is known** in advance.

Syntax:

```
for (initialization; condition; update) {
  // code to execute
}
```

Example:

```
for (let i = 1; i <= 3; i++) {
  console.log("Count: " + i);
```

```
}
```

Output:

Count: 1

Count: 2

Count: 3

2. while Loop

Used when the **number of iterations is not known**, and you want to loop **as long as a condition is true**.

Syntax:

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

Example:

```
let i = 1;
while (i <= 3) {
  console.log("While count: " + i);
  i++;
}
```

Output:

While count: 1

While count: 2

While count: 3

3. do-while Loop

Similar to while, but **executes the code block at least once** before checking the condition.

Syntax:

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

Example:

```
let i = 1;
do {
  console.log("Do-While count: " + i);
  i++;
} while (i <= 3);
```

Output:

Do-While count: 1

Do-While count: 2

Do-While count: 3

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

Main Difference:

Feature	**while Loop**	**do-while Loop**
Condition Check	Checks the condition **before** executing.	Checks the condition **after** executing.
Execution Guarantee	May **not run at all** if the condition is false initially.	**Runs at least once**, even if the condition is false.
Syntax	while (condition) { /* code */ }	do { /* code */ } while (condition);

Example:

while Loop:

```
let x = 0;
while (x > 0) {
  console.log("While loop runs");
}
```

Output: *(nothing prints, because condition is false)*

do-while Loop:

```
let x = 0;
do {
  console.log("Do-while loop runs");
} while (x > 0);
```

Output:

Do-while loop runs

6. Functions:

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

In JavaScript, **functions** are blocks of code designed to perform a particular task. They allow you to write reusable code, making programs more modular and easier to maintain.

- **Declaring a Function**

There are several ways to declare functions in JavaScript, but here's the most common syntax using a **function declaration**:

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

- function: Keyword used to define a function.
- functionName: The name you give to the function.
- parameters: Optional. A list of inputs the function can accept (separated by commas).
- The code block inside {} is the function body that contains the logic.

🔹 Example of a Function Declaration

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

This function takes one parameter (name) and prints a greeting message.

🔹 Calling a Function

You **call** or **invoke** a function by using its name followed by parentheses:

```
greet("Alice");
```

Output:

```
Hello, Alice!
```

🔹 Other Ways to Declare Functions

- **Function Expression:**
- const greet = function(name) {
- console.log("Hello, " + name + "!");
- };
- **Arrow Function (ES6+):**

- const greet = (name) => {
- console.log("Hello, " + name + "!");
- };
-

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

Sure! Here's the **theoretical explanation** of the difference between **function declaration** and **function expression** in JavaScript:

◇ Function Declaration (a.k.a. Function Statement)

A **function declaration** is when you define a function using the function keyword **directly** with a name in the general scope.

- It is **hoisted**, meaning the JavaScript engine loads the entire function into memory during the **compilation phase**, before any code is executed.
- As a result, you can call the function **before or after** its declaration.

Example:

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

◇ Function Expression

A **function expression** is when you define a function and **assign it to a variable**. It can be **anonymous** (without a name) or **named**.

- Unlike declarations, function expressions are **not hoisted** in the same way.
- Only the variable is hoisted (declared), but the function is **not initialized** until the assignment is executed.
- Therefore, trying to call the function **before the assignment** results in a runtime error.

Example:

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

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

Sure! Here's the **theoretical explanation** of **parameters** and **return values** in JavaScript functions:

Parameters (Theoretical Concept)

- **Parameters** are symbolic names listed in a function definition that **accept input values** (called *arguments*) when the function is called.
- They **define the interface** between the function and the caller, allowing data to be passed into the function.
- Parameters are **local variables** within the function's scope, meaning they exist only during the function's execution.

Theory Point:
Parameters enable **function generalization** — instead of hardcoding values, functions can operate on different inputs.

Return Values (Theoretical Concept)

- A **return value** is the output that a function **sends back** to the part of the program that called it.
- The return statement **ends function execution** and optionally specifies the value to return.
- If no return is used, the function implicitly returns **undefined**.

Theory Point:
The return mechanism enables functions to behave like **expressions** — they can compute and provide results, which can be assigned to variables or used in further computation.

7. Arrays

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

An **array** in JavaScript is a **data structure** used to store a **collection of elements** under a single variable name. These elements are organized in a **sequential, ordered list**, and each item in the list can be accessed using its **numeric index**, starting from zero.

Declaring and Initializing an Array :

To **declare** an array means to **create a variable** that is capable of holding multiple values in a list format.

To **initialize** an array means to **assign values** to that array, either at the time of declaration or afterward.

There are two primary ways to do this in JavaScript:

- Using a **literal notation** (by directly listing values),
- Using a **constructor function** (by creating an instance of the array structure).

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

Certainly! Here's the **theoretical explanation** of the array methods push(), pop(), shift(), and unshift() in JavaScript:

- **push()**
  - The push() method is used to **add one or more elements to the end** of an array.
  - It **modifies the original array** by increasing its length.
  - It is commonly used when you want to grow an array dynamically.
- **pop()**

- The pop() method is used to **remove the last element** from an array.
- It **reduces the length** of the array by one.
- The method returns the element that was removed.
- It is useful when you want to treat an array like a **stack** (last-in, first-out).

🔹 **shift()**

- The shift() method is used to **remove the first element** from an array.
- It **shifts all remaining elements** one position to the left.
- Like pop(), it returns the removed element.
- It is useful when processing data in a **queue-like** (first-in, first-out) manner.

🔹 **unshift()**

- The unshift() method is used to **add one or more elements to the beginning** of an array.
- It **shifts existing elements** to higher indexes to make room.
- It modifies the array's structure and increases its length.

8. Objects

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

An **object** in JavaScript is a **complex data structure** used to store **key-value pairs**. Each key (also called a **property**) is a string (or symbol), and it maps to a corresponding **value**, which can be any data type — including numbers, strings, arrays, functions, or even other objects.

Objects are used to represent **real-world entities** or structured data by grouping related information together.

🔹 **How Are Objects Different from Arrays?**

Feature	Objects	Arrays
Structure	Key-value pairs	Ordered list of values
Indexing	Keys (typically strings)	Numeric indexes (0, 1, 2, ...)
Order	Unordered (keys don't follow a strict order)	Ordered (elements have a defined position)
Use Case	Represent entities with named properties	Store collections or lists of similar items
Access Syntax	Accessed via property names	Accessed via numeric indexes
Best For	Structured, descriptive data	Iterating over lists or performing batch tasks

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

Accessing and Updating Object Properties in JavaScript

JavaScript provides two primary ways to **access** and **modify** the properties of an object:

- **Dot Notation**
- **Bracket Notation**

Both are used to interact with the key-value pairs stored in an object, but they differ slightly in **syntax and use cases**.

◆ Dot Notation (.)

Definition:
Dot notation is a simple and commonly used method where the **property name is written directly** after a dot (.) following the object name.

Theoretical Use:

- Best used when property names are **known**, **valid identifiers**, and **do not contain spaces or special characters**.
- Example in theory: To get the value of the name property, use object.name.
- To update, simply assign a new value: object.name = newValue.

🔹 Bracket Notation [ ]

Definition:
Bracket notation uses **square brackets** and requires the property name to be passed as a **string** (or variable containing a string).

Theoretical Use:

- Required when the property name is **dynamic**, contains **special characters**, **spaces**, or **starts with a number**.
- Example in theory: To access the full name property, use object["full name"].
- It also allows property access using a variable, like object[variableName].

9. JavaScript Events

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

In JavaScript, **events** are actions or occurrences that happen in the browser, usually as a result of **user interaction** (such as clicks, key presses, or mouse movements) or due to specific **changes in the state of the document** (like a page load or an element being modified).

Examples of events include:

- **User-initiated events**: click, mouseover, keydown, submit

- **Document-related events**: load, DOMContentLoaded, resize
- **Form-related events**: focus, blur, change
- **Mouse events**: mousemove, mouseup, mouseenter

Events are the core of **user interaction** with a webpage. By responding to these events, you can create **interactive and dynamic** web applications.

🔹 Event Listeners

An **event listener** is a function that waits for a specific event to occur and then responds by executing some code. It essentially **listens** for an event to happen and executes logic when that event occurs.

Role of Event Listeners:

- **Listening for events**: Event listeners allow the web page to **react to user actions** or changes in the document.
- **Separating logic**: They enable you to define actions separately from the HTML structure, leading to cleaner, more maintainable code.
- **Asynchronous**: Event listeners are non-blocking, meaning they don't stop the execution of the rest of the code. The event handler runs only when the event is triggered.

🔹 How Event Listeners Work

1. **Attach an Event Listener**:
   You use the addEventListener() method to attach an event listener to an element. It specifies the event you are waiting for and the function to call when the event occurs.
2. **Trigger the Event**:
   When the specified event occurs (e.g., a user clicks a button), the corresponding function is executed.

🔸 Example

Imagine you have a button element on your webpage. You could set up an event listener to **listen for a click event** on that button. When the user clicks the button, the event listener would call a function that handles that click (for example, showing an alert or changing the page content).

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

The addEventListener() method is used to **attach an event listener** to an element in the DOM (Document Object Model). It allows you to listen for specific events on the target element (like a button, link, or div) and then **execute a function** when that event occurs.

🔹 Syntax of addEventListener()

element.addEventListener(event, function, useCapture);

- **event**: A string representing the type of the event (e.g., "click", "keydown", "submit").
- **function**: The function to be executed when the event is triggered.
- **useCapture** (optional): A boolean value indicating whether the event should be captured in the capturing phase (true) or in the bubbling phase (false, which is the default).

🔹 How It Works

1. **Attach an Event Listener**:
   The addEventListener() method attaches a listener function to an event type on the specified element.
2. **Triggering the Event**:
   When the event occurs on the target element (like a user clicking a button), the **listener function** is executed.
3. **Event Propagation**:
   JavaScript allows events to either propagate **upward** (bubbling) or **downward** (capturing) through the DOM. The useCapture argument determines this behavior.

- **Example: Using addEventListener() to Handle a Click Event**

Let's assume you have a button on your page, and you want to display an alert message when the button is clicked. Here's how you'd set it up:

1. **HTML Structure**:
   This is your basic HTML button.

```
<button id="myButton">Click Me!</button>
```

2. **JavaScript Code**:
   You attach the click event to the button using addEventListener().

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


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

- **Explanation of the Example**

1. **Selecting the Element**:
   let button = document.getElementById("myButton"); — This selects the button element with the id of "myButton".
2. **Adding the Event Listener**:
   button.addEventListener("click", function() { alert("Button was clicked!"); }); — This attaches a listener for the "click" event. When the button is clicked, the function inside the addEventListener() runs, triggering an alert.

10. DOM Manipulation

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** for web documents. It represents the **structure** of an HTML or XML document as a **tree-like structure** where each node represents a part of the document (such as elements, attributes, and text).

In simple terms, the DOM allows JavaScript to **access and manipulate the content, structure, and style** of web pages. It essentially bridges the gap between **JavaScript** and the **HTML** or **XML** document, enabling dynamic updates to the webpage content.

- **How Does JavaScript Interact with the DOM?**

JavaScript interacts with the DOM through a set of **methods and properties** that allow it to:

1. **Access Elements**:
   JavaScript can access HTML elements using methods like getElementById(), getElementsByClassName(), querySelector(), etc. These methods allow you to grab specific elements on the page to work with.
2. **Modify Elements**:
   Once an element is accessed, JavaScript can change its content (innerHTML, textContent), modify its attributes (like src for images or href for links), or adjust its styles (style property).
3. **Add/Remove Elements**:
   JavaScript can also create new elements (createElement()), add them to the document (appendChild()), or remove elements (removeChild()).
4. **Listen for Events**:
   JavaScript can attach **event listeners** to DOM elements to respond to user interactions, like clicks, key presses, or mouse movements.

5. **Manipulate the Structure**:
   JavaScript can change the structure of the DOM by adding, removing, or rearranging elements, enabling dynamic webpage behavior.

- **Example of JavaScript Interacting with the DOM**

Imagine you have an HTML document like this:

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Example</title>
</head>
<body>
  <h1 id="heading">Welcome to my page!</h1>
  <button id="changeTextBtn">Change Text</button>
</body>
</html>
```

You can use JavaScript to interact with this DOM:

1. **Accessing the Element**:
   Get the heading and button elements:
2. let heading = document.getElementById("heading");
3. let button = document.getElementById("changeTextBtn");
4. **Modifying the Element**:
   Change the text of the heading when the button is clicked:
5. button.addEventListener("click", function() {
6. heading.textContent = "Hello, JavaScript!";

7. });
8. **Result**:
   When the user clicks the button, the text content of the <h1> element changes dynamically.

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

Methods to Select DOM Elements in JavaScript

JavaScript provides several methods to access and select elements from the DOM (Document Object Model). Among the most commonly used methods are getElementById(), getElementsByClassName(), and querySelector(). These methods allow developers to interact with specific elements on the page, enabling them to manipulate or modify them as needed.

- **1. getElementById()**

Definition:
getElementById() is a method used to **select an element by its unique id attribute**. Since id attributes must be unique within a document, this method returns a **single element** that matches the provided id.

Usage:

- Returns the **first matching element** with the specified id or null if no element with the given id is found.

Example:

let element = document.getElementById("header");

Here, element will reference the element with the id="header". If no such element exists, element will be null.

- **2. getElementsByClassName()**

Definition:
getElementsByClassName() is a method used to **select all elements** that have a specific class name. It returns a **live HTMLCollection** of all matching elements.

- **Live HTMLCollection**: This means that if elements with the specified class are added or removed from the DOM after the method is called, the collection is **automatically updated**.

Usage:

- Returns a **live collection** of elements that have the specified class name.

Example:

let elements = document.getElementsByClassName("btn");

Here, elements will contain a collection of all elements with the class name "btn". If there are no matching elements, it returns an empty collection.

Important:

- Since it returns a collection, you must access individual elements by their index, like elements[0], elements[1], etc.

🔹 3. querySelector()

Definition:
querySelector() is a method used to **select the first matching element** based on a **CSS selector**. This allows you to use a wide variety of selectors, such as #id, .class, or more complex combinations (e.g., div > p, input[type="text"], etc.).

- **Note**: Unlike getElementById() and getElementsByClassName(), querySelector() works with any valid CSS selector and returns only **the first element** that matches the selector.

Usage:

- Returns the **first matching element** that matches the provided CSS selector or null if no matching element is found.

Example:

let element = document.querySelector(".container");

Here, element will reference the **first element** with the class "container". If no such element exists, element will be null.

11. JavaScript Timing Events (setTimeout, setInterval)

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

In JavaScript, **setTimeout()** and **setInterval()** are two commonly used functions for **delaying or repeatedly executing code** after a certain amount of time. They are both part of the **timing events** mechanism in JavaScript, which allows you to schedule functions to run after a delay or at regular intervals.

- **1. setTimeout()**

Definition:
The **setTimeout()** function is used to **execute a function** after a specified number of milliseconds (milliseconds = 1/1000 of a second). The function is executed **once**, after the delay.

Syntax:

setTimeout(function, delay, [parameters]);

2. setInterval()

Definition:
The **setInterval()** function is used to **execute a function repeatedly**, with a fixed time delay between each execution. It continues running at intervals until it is stopped using clearInterval().

Syntax:

setInterval(function, interval, [parameters]);

Stopping Timers

- **clearTimeout()**: Can be used to stop the execution of a function set by setTimeout() before the delay time is reached.

Syntax:

clearTimeout(timeoutID);

- **clearInterval()**: Can be used to stop the repeated execution of a function set by setInterval().

Syntax:

clearInterval(intervalID);

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

Using setTimeout() to Delay an Action by 2 Seconds

The setTimeout() function allows you to execute a function after a specified delay. Here's how you can use setTimeout() to delay an action by 2 seconds (2000 milliseconds):

Example:

```
// Function to execute after a 2-second delay
setTimeout(function() {
  console.log("This message is displayed after a 2-second delay.");
}, 2000);
```

12. JavaScript Error Handling

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 is a mechanism that allows you to manage **runtime errors** and prevent the program from crashing. By using error handling techniques, you can **gracefully handle unexpected issues** that arise during the execution of your code, such as invalid user input or failed network requests.

In JavaScript, error handling is primarily done using the **try...catch...finally** statement. This structure enables you to try running code that might throw an error, catch those errors if they occur, and execute certain code regardless of whether an error occurred or not.

- **The try, catch, and finally Blocks**

1. **try Block**:
   The try block is used to wrap the code that **might throw an error**. If an error occurs in the try block, JavaScript will immediately stop executing that block and move to the catch block to handle the error.
2. **catch Block**:
   The catch block is used to handle the error thrown by the try block. If an error occurs in the try block, the catch block is executed, where you can define how to handle the error (e.g., log it, display an error message to the user, etc.).
3. **finally Block**:
   The finally block is optional and is used for **code that must run regardless of whether an error occurred or not**. This block will be executed after the try and catch blocks have completed, making it useful for cleaning up resources or performing actions that need to happen regardless of an error (e.g., closing a file or database connection).

- **Syntax of try...catch...finally**

```
try {
  // Code that may throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code that runs regardless of an error
}
```

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

Error handling in JavaScript is crucial for ensuring that applications run smoothly and reliably. It allows you to manage and respond to errors that might occur during runtime, preventing the application from crashing. Here's why it's important:

1. **Prevents Crashes**: It stops the application from halting unexpectedly when an error occurs.
2. **Improves User Experience**: Provides meaningful error messages instead of generic ones, guiding the user when something goes wrong.
3. **Aids Debugging**: Helps developers identify and fix issues by logging detailed error information.
4. **Enhances Security**: Protects sensitive data by preventing the exposure of internal details when errors occur.
5. **Ensures Reliability**: Allows the application to fail gracefully, maintaining functionality even when some operations fail.
6. **Handles Asynchronous Errors**: Manages errors in asynchronous code (e.g., API calls or promises).
7. **Promotes Consistency**: Ensures a uniform approach to handling errors across the application.