Module 17: Javascript For Full Stack Assignment

❖ Theory Assignment :-

1. JavaScript Introduction :

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

➢ JavaScript (often abbreviated JS) is a high-level, interpreted programming language.

➢ It is a dynamic, weakly typed (or loosely typed) language.

➢ It supports multiple programming paradigms (procedural, object-oriented, functional).

➢ It is executed by web browsers (client-side), and also on servers (server-side) via environments like Node.js.

➢ Role of JavaScript in web development:
➢ In web development, HTML, CSS, and JavaScript are the three core technologies:

➢ HTML provides the structure of a web page (elements, content).

➢ CSS handles the styling (colors, layout, fonts).

➢ JavaScript brings interactivity and dynamic behavior. For example:
• Responding to user actions (clicks, mouse movements, keyboard input).
• Validating form inputs before submission(client-side).
• Dynamically modifying and updating content in the page (adding/removing elements).
• Animations, sliders, popups, modal dialogs.
• Fetching data asynchronously from servers (AJAX / fetch) so the page can update the page without full reload.
• Building entire single-page applications (SPA) using libraries like React, Vue, Angular
• On the server side (Node.js), JavaScript can be used to handle routing, database operations, real-time communication, APIs, etc.

➢ Thus, JavaScript "brings web pages to life" by making them interactive rather than static content.

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

Feature	**JavaScript**	**Python / Java**
Typing	Dynamically typed, weak typing / implicit conversions allowed	Python: also dynamic, but more strict in conversions; Java: statically typed
Compilation / Execution	Interpreted or JIT-compiled; runs in browser environment	Python: interpreted; Java: compiled to bytecode and run on JVM
Run-time environment	Browser (client) + server (Node.js)	Python: on server or local; Java: JVM on server or client
Object model	Prototype-based inheritance	Java: class-based; Python: class-based, supports multiple inheritance
Scope rules	Before ES6: function scope (var), now block scope (let / const)	Python / Java: block / function / class scope more rigid
Concurrency model	Single-threaded event loop, asynchronous (callbacks, promises, async/await)	Java: multi-threading; Python: threads, async (in newer versions)
Use in web	Native in browsers, primary for front-end	Java (via applets—rare nowadays) or via server-side frameworks (Django, Spring)

➢ JavaScript was built with the web in mind and has features (like DOM interaction, event-driven model, asynchronous I/O) that are tightly integrated with browsers, which is a key differentiator compared to general-purpose languages.

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

➢ The <script> tag is used to embed or reference JavaScript code in an HTML document.

➢ You can put JavaScript code **inline** within the <script> tag,
e.g.: <script>
console.log("Hello from inline JS");
</script>

➢ Or you can link to an **external** .js file:-

➢ <script src="script.js"></script>

➢ Best practices / details:
- It is common to place the <script> tag just before the closing </body> tag,

so that the HTML content loads first and your script can safely manipulate DOM elements.

• Alternatively, you can place the <script> tag in <head> but use defer or async attributes to control loading behavior.
• The browser will fetch and execute the script. Using src ensures separation of concerns (HTML vs JavaScript).
• You should avoid having script tags try to act on DOM elements that haven't yet been loaded; that's why placing them at bottom or using DOMContentLoaded events is common.

2. Variables and Data Types :

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

- **Variables** are named storage locations in a program that hold values which can change (or sometimes not change). They allow you to store data and refer to it by name.
- In JavaScript, you can declare variables using three keywords: var, let, and const. The differences among them relate to scope, re-assignment, and hoisting behavior.

- **VAR:-**

- var
- var x = 10;
- var name = "Alice";

- Function-scoped (or globally scoped if declared outside a function).
- Variables declared with var are hoisted (i.e. the declaration is moved to the top of its scope) and initially have the value undefined until assignment.
- You can reassign (change) the value of a var variable.
- You can also redeclare the same variable with var in the same scope without error.

- **LET:-**

- let
- let y = 20;
- let city = "Mumbai";

- Block-scoped (i.e. limited to the block { … } in which it is declared).
- Not hoisted in the same way as var (the variable is in a "temporal dead zone" until its declaration is encountered).
- You can reassign the value, but you cannot redeclare the same variable name in the same block scope.

➢ **CONST:-**

- const
- const PI = 3.14159;
- const user = { name: "Bob" };

➢ Also block-scoped.
➢ Must be initialized at the time of declaration (you have to give it a value immediately).
➢ You cannot reassign the variable to a new value. However, if the value is an object (or array), you *can* mutate its contents (e.g. change object properties or push into the array) because the const only protects the binding, not the contents.

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

➢ JavaScript has several built-in data types. Broadly, they fall into **primitive** types and **non-primitive** (reference) types.
➢ **Primitive types:-**

1. **Number**
   Represents both integer and floating-point numbers.

   let a = 42;

   let b = 3.14;

2. **String**
   Sequence of characters.

   let name = "John";

   let greeting = 'Hello';

3. **Boolean**
   Either true or false.

   let isActive = true;

   let done = false;

4. **Undefined**
   A variable that has been declared but not assigned a value will have the value undefined.

   let x;

   console.log(x); // undefined

5. **Null**
   Represents “no value” or “empty.” It’s an explicit assignment to indicate absence of any object value.

```
let obj = null;
```

6. **Symbol** (ES6)
   A unique and immutable primitive value, often used as object property keys.

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

7. **BigInt** (ES2020)
   For integers beyond the safe integer limit for Number.

   ```
   let big = 1234567890123456789012345678901234567890n;
   ```

➢ **Non-primitive / Reference types:-**

1. **Object**
   Collections of key-value pairs (properties), arrays, date, etc., are all objects under the hood.

   ```
   let person = { name: "Alice", age: 30 };
   ```

2. **Array**
   A special kind of object for ordered collections.

   ```
   let arr = [1, 2, 3, "hello"];
   ```

3. **Function**
   Functions are also first-class objects in JavaScript.

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

4. **Date**, **RegExp**, etc.
   These are built-in object types.

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

➢ undefined is the default value when a variable is declared but not assigned a value. It is also the return value of functions that don't explicitly return anything, or when accessing non-existent object properties.

➢ null is an assignment value — it is used to indicate "no value" or "nothing." It's a way for a programmer to explicitly say the variable has no value.

▪ **Key distinctions:**

➢ typeof undefined gives "undefined", whereas typeof null gives "object" (this is a known quirk in JavaScript).

➢ You typically use null when you want to clear a variable or denote an intentional absence of object. Using undefined explicitly is less common (because undefined is more used by the JS engine itself).

3. JavaScript Operators :

 Q - 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.

➢ Here are the main categories of operators with examples:

1. **Arithmetic operators:-**

- o Some of the arithmetic opertors are as follows:
- o + addition
- o - subtraction
- o * multiplication
- o / division
- o % modulus
- o ** exponentiation
- o ++ increment
- o – decrement

For Examples:-

```
let x = 5 + 3;  // 8

let y = 10 % 3; // 1

let z = 2 ** 3; // 8

x++;  // x becomes 9
```

2. **Assignment operators**

- o = (simple assignment)
- o +=, -=, *=, /=, %=, **= (compound assignments).

For Example-

```
let a = 10;

a += 5; // a = a + 5 → 15

a *= 2; // a = a * 2 → 30
```

3. **Comparison operators** (relational)

- o == (equal to)
- o != (not equal)
- o === (strict equal)
- o !== (strict not equal)
- o > (greater than)

- < (less than)
- >=, <=

For Example:-

- 5 == "5";   // true (because type coercion happens)
- 5 === "5"; // false (no type coercion)
- 10 > 7;   // true

4. **Logical operators**
   - && (logical AND)
   - || (logical OR)
   - ! (logical NOT)

   For Example:-

   let p = true, q = false;

   p && q; // false

   p || q; // true

   !p;   // false

5. **Bitwise operators**
   - This operator operates on 32-bit integers at the bit level (e.g. &, |, ^, ~, <<,>>).

   Example: let b = 5 & 3; // bitwise AND → 1

6. **Conditional (ternary) operator**
   - condition ? expr1 : expr2
   - Example: let status = (age >= 18) ? "adult" : "minor";

7. **Comma operator**
   - **,** — evaluates multiple expressions and returns the last one.
   - let c = (1, 2, 3); // c = 3

8. **Type operators**
   - typeof — returns the type of a value
   - instanceof — checks if an object is an instance of a constructor

   For Exmaple:

   - typeof 123;   // "number"
   - [] instanceof Array; // true

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

- **== (loose equality):-** It checks for equality **after** performing type conversion (coercion) if needed. So it may convert one or both operands to a common type before comparison.
- **=== (strict equality):-** It checks for equality **without** type conversion — both the value and the type must be the same.
  In general, === is recommended because it avoids unexpected coercion and leads to more predictable comparisons.

For Examples:

5 == "5";   // true, because "5" is converted to number 5

5 === "5";   // false, because types differ (number vs string)

0 == false;   // true (false coerced to 0)

0 === false; // false (different types).

4. Control Flow (If-Else, Switch) :

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

- Control flow refers to the order in which statements, instructions, or function calls are executed in a program. In JavaScript, you can control this flow using decision-making constructs like if, else if, else, switch, loops, etc.
- The if-else statement lets you execute a block of code if a specified condition is true, and optionally another block if it's false.
- Syntax:

```
if (condition) {
  // code if condition is true
} else if (Condition) {
  // code to run if first was false and this is true
} else {
  // code to run if  none of the above conditions are true
}
```

For Example:

```
let score = 85;
if (score >= 90) {
  console.log("Grade: A");
  } else if (score >= 75) {
```

```
  console.log("Grade: B");
} else if (score >= 60) {
  console.log("Grade: C");
} else {
  console.log("Grade: D");
}
```

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

- A switch statement evaluates an expression and matches its value against various case labels. When it finds a matching case, it executes statements associated with that case. Optionally, there is a default case if none match.
- **Syntax:**

```
switch (expression) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;
  default:
    // code if none of the cases match
}
```

Example:

```
let color = "green";
switch (color) {
  case "red":
    console.log("Stop");
    break;
  case "green":
    console.log("Go");
    break;
  case "yellow":
    console.log("Caution");
    break;
  default:
    console.log("Unknown color");
}
```

Here, since color is "green", it matches the second case and prints "Go".

➢ **When to prefer switch over if-else:**

- When you have to compare the same variable or expression against many constant values.
- It often results in cleaner, more readable code when you have many branches.
- switch can be more efficient in some situations, though in most JS engines performance difference is marginal.
- Use if-else when your conditions are more complex (ranges, inequalities, logical combinations) that aren't just checking equality to fixed values.

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

 Q - 1: Explain the different types of loops in JavaScript (for, while, do-while). Provide a basic example of each.
1. for loop:- The for loop is commonly used when you know how many times you want to iterate.

Example:-

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
// Output: 0, 1, 2, 3, 4
```

- i = 0 initializes the loop counter.
- i < 5 is the loop condition.
- i++ updates the counter after each iteration.

2. While loop
Repeats as long as a certain condition is true. The condition is checked before each iteration.

Exmaple:-

```
let j = 0;
while (j < 5) {
  console.log(j);
  j++;   }
```

3.do-while loop
Similar to while, but executes the block **at least once** before checking the condition (because the condition is checked after the block).

Example:-

```
let k = 0;
do {
  console.log(k);
  k++;
} while (k < 5);
```

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

1. While loop:-

This loop checks the condition **before** executing the loop body. If the condition is false initially, the body may not run even once.

2. Do-While loop:-

This loop executes its body once first, then checks the condition. So the body is guaranteed to run at least one time, even if the condition is false initially.

5. Functions:

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

- Functions are reusable blocks of code that perform a specific task or compute a value. They can accept inputs, process them, and optionally return an output.
- **Function declaration syntax:**

```
function add(a, b) {

  return a + b;

}
```

- To call (invoke) this function:

```
let sum = add(2, 3);  // sum will be 5
```

You can also define **anonymous** or **arrow functions** (in ES6):

```
const subtract = function(a, b) {

  return a - b;

};
```

```
const multiply = (a, b) => {
  return a * b;
};

// Or for a single expression:
const square = x => x * x;
```

➢ To call:

```
subtract(5, 2);
multiply(3, 4);
square(7);
```

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

1.Function Declaration:-

Syntax:-
```
function foo() {
  // ...
}
```

- Hoisted fully: you can call foo() even before the line where it appears in the code.

2. Function Expression:-

Syntax:-
```
const bar = function() {
  // ...
};
```

- Only the variable bar is hoisted (but without assignment). The actual function definition is not hoisted. So you cannot call bar() before that line.
- Can be anonymous or named.
- Often used in callbacks or to assign functions to variables.

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

- **Parameters** are placeholders for the values (arguments) that are passed into a function. In the definition function foo(a, b), a and b are parameters.
- **Arguments** are the actual values you pass when calling the function,

- e.g. foo(2, 3) — here 2 and 3 are arguments.
- A function may **return** a value using the return statement. After return is executed, the function stops executing further. If there is no return, the function returns undefined by default.

For Example:

```
function multiply(a, b) {
  return a * b;
}
let result = multiply(4, 5);  // result = 20.
```

6. Arrays :

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

➢ An **array** is a special object in JavaScript used to store ordered collections of values (elements). Elements can be of any type (numbers, strings, objects, etc.).

➢ You declare/initialize arrays like this:

```
let arr1 = [];              // empty array
let arr2 = [1, 2, 3, 4];    // array with numbers
let arr3 = ["apple", "banana", "cherry"];
let mixed = [1, "hello", { name: "Bob" }, [2, 3]];
```

➢ You access elements by index starting at 0:

```
console.log(arr2[0]);  // 1
console.log(arr3[2]);  // "cherry"
```

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

➢ These methods let you add or remove elements at the beginning or end of an array.

1. push(element1, element2, …)
Adds one or more elements to the *end* of the array, returns the new length.

- let arr = [1, 2];
- arr.push(3);   // arr becomes [1, 2, 3]
- arr.push(4, 5); // arr becomes [1, 2, 3, 4, 5]

2. .pop()
Removes the last element from the array and returns it.

- let arr = [1, 2, 3];
- let last = arr.pop(); // last = 3, arr becomes [1, 2]

3. shift()
Removes the *first* element from the array and returns it. All remaining elements shift to a lower index.

- let arr = [1, 2, 3];
- let first = arr.shift(); // first = 1, arr becomes [2, 3]
- .unshift(element1, element2, …)
  Adds one or more elements to the *beginning* of the array, returns the new length.
- let arr = [2, 3];
- arr.unshift(1);   // arr becomes [1, 2, 3]
- arr.unshift(-1, 0); // arr becomes [-1, 0, 1, 2, 3]

7. Objects Theory Assignment :

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

➢ An **object** is a collection of key-value pairs (properties), where the keys are (usually) strings (or symbols) and the values can be any type (primitive, object, function, etc.). Objects are used to model more complex data and entities with properties and behavior.

Example:

```
let person = {
  name: "Alice",
  age: 30,
  greet: function() {
    console.log("Hello, I'm " + this.name);
  }
};
```

➢ **Differences from arrays:**

- Arrays are ordered collections indexed by numbers (0, 1, 2, …), while object properties are accessed by key names.
- Arrays come with special behaviours (methods like push/pop), iteration semantics, and length property.
- Use arrays when you need lists or ordered data; use objects when you want named properties or key-value pairs.

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

- If you have an object:

```
let car = {
  make: "Toyota",
  model: "Camry",
  year: 2025,
  features: {
    color: "black",
    transmission: "automatic"
  }
};
```

- You can **access** properties by:

1. Dot notation

```
console.log(car.make);       // "Toyota"
console.log(car.features.color);  // "black"
```

2.Bracket notation

- Bracket notation is especially useful when the property name is dynamic or not a valid identifier (e.g. has spaces, starts with number, or stored in a variable):

```
console.log(car["model"]);     // "Camry"
console.log(car["features"]["transmission"]);  // "automatic"
let prop = "year";
console.log(car[prop]);  // 2025
let weird = {
  "some-key": 42
};
console.log(weird["some-key"]);  // 42
```

- You can **update** properties similarly:

```
car.year = 2026;
car["make"] = "Honda";
```

car.features.color = "red";

car["features"]["transmission"] = "manual";

➢ You can also add new properties:

car.owner = "John";

car["mileage"] = 12000;

8. JavaScript Events Theory Assignment:

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

➢ **Events** are actions or occurrences that happen in the browser — for example, a user clicking a button, moving the mouse, pressing a key, loading a page, resizing the window, etc. In JavaScript, you can respond to these events to make your web pages interactive.
➢ An **event listener** (or event handler) is a function you assign to listen for a specific event on an element, so that when that event occurs, your function is run.
➢ For example, you might listen for a click event on a button, and then run some code when the button is clicked.

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

➢ The addEventListener() method attaches an event handler to a specified element without overwriting existing event handlers. It takes at least two arguments:

- element.addEventListener(eventType, callbackFunction);
- eventType — a string like "click", "mouseover", "keydown", etc.
- callbackFunction — the function to call when the event occurs.

Example:

```
<button id="myBtn">Click Me</button>

<script>

  const btn = document.getElementById("myBtn");

  btn.addEventListener("click", function(event) {

    alert("Button was clicked!");

  });

</script>


function sayHi() {
  console.log("Hi there!");
}
btn.addEventListener("click", sayHi);
// or
btn.addEventListener("mouseover", () => {
  console.log("Mouse over the button");
});
```

9. DOM Manipulation Theory Assignment :

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

- ➢ The **DOM (Document Object Model)** is a programming interface for web documents. It represents the HTML document as a tree of nodes (elements, text, attributes). Using the DOM, JavaScript can access, traverse, and modify the HTML structure, styles, and content dynamically.
- ➢ Essentially, the DOM is the bridge between JavaScript and the webpage.
- ➢ Through the DOM, JavaScript can:

- Select elements
- Read or change element attributes, content, styles
- Add or remove elements
- React to user interactions (via events)
- Create new elements
- Change the structure or layout dynamically

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

1. **document.getElementById(id)**

- o Returns the element whose id attribute matches the given string.
- o If no matching element is found, returns null.
- o Only one element can have a given id in valid HTML, so it returns a single element (not a list).
- o Example: let header = document.getElementById("main-header");

2. **document.getElementsByClassName(className)**

- Returns a live **HTMLCollection** (array-like) of all elements that have the specified class.
- If no elements match, returns an empty collection (not null).
- Example: let items = document.getElementsByClassName("list-item");
- // items[0], items[1], etc.

3. **document.querySelector(selector)**

- Returns the first element that matches the CSS selector string.
- If no match, returns null.
- Example: let firstItem = document.querySelector(".list-item");
- let nav = document.querySelector("#nav > ul > li.active");

4. **document.querySelectorAll(selector)**

- Returns a static **NodeList** of all elements matching the CSS selector.
- Example: let allItems = document.querySelectorAll(".list-item");
- querySelector/querySelectorAll are more flexible because they accept any valid CSS selector (class, id, attribute selectors, pseudo-classes, descendant selectors, etc.).

10. JavaScript Timing Events (setTimeout, setInterval) :

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

- **setTimeout(callback, delay, [args])**
  Schedules the callback function to run **once** after a specified delay in milliseconds.
- setTimeout(() => {
-   console.log("This message appears after 2 seconds");
- }, 2000);

You can cancel a timeout before it runs by using clearTimeout(timeoutID), where timeoutID is the value returned by setTimeout.

```
let timer = setTimeout(() => {
  console.log("Won't show if cleared");
}, 3000);


clearTimeout(timer);
```

- **setInterval(callback, delay, [args])**
  Schedules the callback function to run **repeatedly**, every delay milliseconds, until it is canceled.

```
let count = 0;

let intervalId = setInterval(() => {

  count++;

 console.log("Count:", count);

 if (count === 5) {

    clearInterval(intervalId);

  }

}, 1000);
```

To stop it, use clearInterval(intervalId).

These timing functions are very useful for delayed actions, animations, polling, repeated tasks, etc.

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

Example: Display a message after 2 seconds

```
setTimeout(function() {

   console.log("This message is shown after 2 seconds");

   }, 2000);                    // 2000 milliseconds = 2 seconds
```

- setTimeout() is a JavaScript function that executes a function after a specified delay (in milliseconds).
- In this case, the message will appear in the console after 2000 milliseconds (2 seconds).

11. JavaScript Error Handling :

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

Error Handling: In JavaScript allows developers to catch and respond to errors that occur during the execution of code, instead of letting the program crash.

Syntax:-

```
try {

   // Code that may throw an error
} catch (error) {
   // Code to handle the error
} finally {
   // Code that will always run, regardless of error
}
```

Example:

```
try {
   let result = 10 / 0;
   console.log("Result is:", result);
   let name = null;
   console.log(name.toUpperCase()); // This will throw an error
   } catch (error) {
   console.log("An error occurred:", error.message);
   } finally {
   console.log("This will always run.");
   }
```

Explanation:

- try: Contains code that might cause an error.
- catch: Executes if an error is thrown in the try block. It receives the error object.
- finally: Executes whether or not an error occurred, often used for cleanup actions.

 Q - 2: Why is error handling important in JavaScript applications?

➢ **Importance of Error Handling:**

1. Prevents Application Crashes:
   Catching and handling errors ensures that the app doesn't stop working unexpectedly.
2. Improves User Experience:
   Instead of showing technical errors, you can show user-friendly messages.
3. Debugging and Logging:
   Errors can be logged for developers to track and fix issues later.
4. Maintains Application Flow:
   Even if an error occurs, other parts of the application can continue to run.

5. Security:
   Proper error handling avoids exposing sensitive details about the application to users.
