**Module 8) JavaScript**

**JavaScript Introduction**

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

**JavaScript (JS)** is a **high-level, interpreted, object-oriented programming language** that runs in the browser. It is one of the **core technologies of the web** along with **HTML** and **CSS**.

**Role in Web Development:**

* **HTML** → provides structure (what’s on the page).
* **CSS** → provides styling (how it looks).
* **JavaScript** → provides interactivity (how it behaves).

Uses in web development:

* Make web pages **dynamic & interactive** (e.g., sliders, dropdowns, popups).
* **Form validation** before submission.
* **DOM manipulation** (change content, styles, and elements dynamically).
* **Asynchronous programming** with AJAX/Fetch to get data from servers without reloading the page.
* Power **modern web apps** (React, Angular, Vue are built on JS).

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

| **Feature** | **JavaScript** | **Python** | **Java** |
| --- | --- | --- | --- |
| **Execution** | Runs in browsers (client-side), also on servers via Node.js | Runs on Python interpreter | Compiled into bytecode, runs on JVM |
| **Typing** | Dynamically typed (no need to declare variable types) | Dynamically typed | Statically typed (must declare types) |
| **Syntax** | Similar to C-style; event-driven, asynchronous support | Easy, human-readable, indentation-based | Verbose, OOP-heavy, class-based |
| **Usage** | Web apps, front-end, back-end, mobile apps | AI, ML, data science, automation, backend | Enterprise software, Android apps, backend |
| **Compilation** | Interpreted (line-by-line) | Interpreted | Compiled then executed |
| **Object-Oriented** | Prototype-based | Class-based (but flexible) | Strict class-based |

**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 is used to **embed or reference JavaScript code** inside an HTML document.

**Ways to use <script>:**

1. **Inline JavaScript** (inside HTML file)

<!DOCTYPE html>

<html>

<head>

<title>Inline JS Example</title>

</head>

<body>

<h1>Hello</h1>

<script>

alert("Hello from JavaScript!");

</script>

</body>

</html>

1. **External JavaScript** (best practice )

* Create a script.js file:
* console.log("Hello from external JS!");
* Link it in HTML:
* <script src="script.js"></script>

1. **Placing <script> tag**
   * Normally placed **before </body>** to ensure HTML loads first (better performance).
   * Or use defer/async attributes in <head> for better control:

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

**Variables and Data Types**

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

A **variable** is a named container that stores data values.

**Ways to declare variables:**

1. **var**
   * Old way of declaring variables (before ES6).
   * Function-scoped.
   * Can be **redeclared** and **updated**.
2. var name = "John";
3. var name = "Doe"; // Redeclaration allowed
4. **let** (ES6)
   * Block-scoped.
   * Can be **updated**, but **not redeclared** in the same block.
5. let age = 25;
6. age = 26; // allowed
7. // let age = 30; not allowed in same scope
8. **const** (ES6)
   * Block-scoped.
   * **Cannot be updated or redeclared.**
   * Must be assigned a value at declaration.
9. const pi = 3.14;
10. // pi = 3.1415; Error

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

JavaScript has **two categories** of data types:

**1. Primitive Data Types (immutable, single value)**

* **String** → Textual data
* let str = "Hello";
* **Number** → Integers & decimals
* let num = 42;
* **Boolean** → true or false
* let isOnline = true;
* **Undefined** → Declared but not assigned a value
* let x;
* console.log(x); // undefined
* **Null** → Explicitly empty value
* let y = null;
* **Symbol** (ES6) → Unique identifiers
* let sym = Symbol("id");
* **BigInt** (ES11) → Large integers beyond Number limit
* let big = 123456789012345678901234567890n;

**2. Non-Primitive (Reference) Data Types**

* **Object** → Collection of key-value pairs
* let person = { name: "Alice", age: 30 };
* **Array** → Ordered list of values
* let arr = [1, 2, 3, 4];
* **Function** → Reusable block of code
* function greet() { return "Hello"; }

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

| **Feature** | **Undefined** | **null** |
| --- | --- | --- |
| **Meaning** | Variable is declared but not assigned a value. | Intentional absence of any value (empty). |
| **Type** | undefined (its own type) | object (legacy quirk in JS) |
| **Assigned By** | Automatically by JavaScript. | Manually by programmer. |
| **Example** | let x; console.log(x); // undefined | let y = null; console.log(y); // null |

**JavaScript Operators**

**Question 1: What are the different types of operators in JavaScript?**

Operators are special symbols used to perform operations on values or variables.

**1. Arithmetic Operators (perform math calculations)**

let a = 10, b = 3;

console.log(a + b); // 13 (Addition)

console.log(a - b); // 7 (Subtraction)

console.log(a \* b); // 30 (Multiplication)

console.log(a / b); // 3.333... (Division)

console.log(a % b); // 1 (Modulus → remainder)

console.log(a \*\* b); // 1000 (Exponentiation → 10³)

**2. Assignment Operators (assign values to variables)**

let x = 5

x += 3; // x = x + 3 → 8

x -= 2; // x = x - 2 → 6

x \*= 4; // x = x \* 4 → 24

x /= 6; // x = x / 6 → 4

x %= 3; // x = x % 3 → 1

**3. Comparison Operators (compare two values → returns true/false)**

let a = 5, b = "5", c = 10;

console.log(a == b); // true (equal, value only)

console.log(a === b); // false (strict equal, value + type)

console.log(a != c); // true (not equal)

console.log(a < c); // true

console.log(a > c); // false

console.log(a <= 5); // true

console.log(c >= 10); // true

**4. Logical Operators (used with boolean values)**

let p = true, q = false;

console.log(p && q); // false (AND → both must be true)

console.log(p || q); // true (OR → at least one true)

console.log(!p); // false (NOT → reverses value)

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

| **Operator** | **Name** | **Compares** | **Example** |
| --- | --- | --- | --- |
| == | Loose Equality | Only **values** (performs type conversion if needed) | 5 == "5" → **true** |
| === | Strict Equality | **Values and Data Types** (no type conversion) | 5 === "5" → **false** |

.

**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 determines **the order in which code executes**.  
Normally, code runs **top to bottom**, but with control flow statements (if-else, switch, loops), you can make decisions and repeat tasks.

**if-else statement**

* Executes **one block of code** if a condition is true, otherwise another block.

**Syntax:**

if (condition) {

// Code runs if condition is true

} else {

// Code runs if condition is false

}

**Example:**

let age = 20;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

**Output**: "You are an adult." (since 20 ≥ 18)

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

A **switch statement** is used to test a variable against **multiple possible values**.  
Instead of writing many if-else if conditions, you use switch for cleaner code.

**Syntax:**

switch(expression) {

case value1:

// Code if expression === value1

break;

case value2:

// Code if expression === value2

break;

default:

// Code if no case matches

}

**Example:**

let day = 3;

let dayName;

switch(day) {

case 1:

dayName = "Monday";

break;

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;

default:

dayName = "Invalid day";

}

console.log(dayName);

**Output**: "Wednesday"

**When to use switch vs if-else?**

* Use **if-else** → when you need to check **ranges or complex conditions** (if (x > 10 && y < 5) etc.).
* Use **switch** → when checking a **single value** against multiple possible options (like days, months, menu choices).

**Loops (For, While, Do-While)**

**Question 1: Explain the different types of loops in JavaScript**

JavaScript has **3 main types of loops**:

**1. for loop**

* Used when you know **how many times** you want to run the loop.
* It has **3 parts**: initialization, condition, and increment/decrement.

Example: print numbers from 1 to 5

for (let i = 1; i <= 5; i++) {

console.log(i);

}

**Output:**

1

2

3

4

5

**2. while loop**

* Used when you **don’t know exactly how many times** the loop should run.
* It keeps running **as long as the condition is true**.

Example: print numbers from 1 to 5

let i = 1;

while (i <= 5) {

console.log(i);

i++;

}

**3. do-while loop**

* Similar to while, but it **executes the block at least once** before checking the condition.
* The condition is tested **after** the loop body.

Example: print numbers from 1 to 5

let i = 1;

do {

console.log(i);

i++;

} while (i <= 5);

**Question 2: Difference between while and do-while loop**

| **Feature** | **while loop** | **do-while loop** |
| --- | --- | --- |
| Condition check | Checked **before** executing the loop body | Checked **after** executing the loop body |
| Minimum execution | May **not run at all** if condition is false | Runs **at least once** even if condition is false |
| Syntax | while (condition) { ... } | do { ... } while (condition); |

Example:

let x = 10;

while (x < 5) {

console.log("This will NOT run");

}

do {

console.log("This WILL run once, even though condition is false");

} while (x < 5);

**Output:**

This WILL run once, even though condition is false

**Functions**

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

**Definition:**

A **function** in JavaScript is a block of code designed to perform a particular task.  
It allows you to **reuse code** instead of writing it multiple times.

**Syntax (Function Declaration):**

// Function declaration

function greet(name) {

console.log("Hello, " + name + "!");

}

// Function call (invocation)

greet("John");

**Output:**

Hello, John!

* function → keyword
* greet → function name
* (name) → parameter
* { ... } → function body
* greet("John") → function call

**Question 2: Difference between Function Declaration and Function Expression**

| **Feature** | **Function Declaration** | **Function Expression** |
| --- | --- | --- |
| Syntax | Uses the function keyword with a name | Function assigned to a variable |
| Hoisting | **Hoisted** (can be called before it’s defined) | **Not hoisted** (must be defined before calling) |
| Example |  |  |

// Function Declaration

function add(a, b) {

return a + b;

}

console.log(add(2, 3));

|

// Function Expression

const add = function(a, b) {

return a + b;

};

console.log(add(2, 3)); If you try to call a **function expression** before it’s defined, you’ll get an error.

**Question 3: Parameters and Return Values**

**Parameters:**

* Variables that you pass into a function so it can use them inside.
* Example:

function multiply(a, b) { // a and b are parameters

console.log(a \* b);

}

multiply(4, 5); // Arguments (4, 5) are passed to parameters

**Output:**

20

**Return Values:**

* A function can return a result using the return keyword.
* Example:

function square(n) {

return n \* n; // returns the result instead of just printing

}

let result = square(6);

console.log(result);

**Output:**

36

**Array**

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

**Definition:**

An **array** in JavaScript is a special type of object used to **store multiple values** in a single variable.  
Each value in an array has an **index** (starting from 0).

**Declaring & Initializing an Array**

There are two common ways:

**1. Using square brackets (most common):**

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

console.log(fruits);

**2. Using the new Array() constructor:**

let numbers = new Array(10, 20, 30, 40);

console.log(numbers);

**Question 2: Explain push(), pop(), shift(), and unshift()**

These are **array methods** to add or remove elements.

**1. push()**

* Adds one or more elements **to the end** of the array.
* Returns the new length of the array.

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

fruits.push("Mango");

console.log(fruits); // ["Apple", "Banana", "Mango"]

**2. pop()**

* Removes the **last element** of the array.
* Returns the removed element.

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

let removed = fruits.pop();

console.log(fruits); // ["Apple", "Banana"]

console.log(removed); // "Mango"

**3. shift()**

* Removes the **first element** of the array.
* Shifts all other elements to lower indexes.
* Returns the removed element.

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

let first = fruits.shift();

console.log(fruits); // ["Banana", "Mango"]

console.log(first); // "Apple"

**4. unshift()**

* Adds one or more elements **to the beginning** of the array.
* Returns the new length of the array.

let fruits = ["Banana", "Mango"];

fruits.unshift("Apple");

console.log(fruits); // ["Apple", "Banana", "Mango"]

**object**

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

**Object Definition:**

An **object** in JavaScript is a collection of **key–value pairs** (properties).  
Each property has a **name (key)** and a **value**.

**Example**:

let person = {

name: "John",

age: 25,

isStudent: true

};

console.log(person);

**Difference between Objects and Arrays**

| **Feature** | **Objects** | **Arrays** |
| --- | --- | --- |
| Structure | Stores data as **key–value pairs** | Stores data as **ordered elements** |
| Key/Index | Keys are **custom (string or symbol)** | Indexes are **numeric (0,1,2,...)** |
| Usage | Used to represent real-world entities | Used for lists, sequences, collections |
| Example | { name: "John", age: 25 } | [ "Apple", "Banana", "Mango" ] |

**Question 2: Access and Update Object Properties**

There are **two main ways** to access and update object properties:

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

* Easiest and most common way.
* Works when property names are valid identifiers (no spaces, no special chars).

let car = { brand: "Toyota", year: 2020 };

// Access

console.log(car.brand); // Toyot

// Update

car.year = 2022;

console.log(car.year); // 2022

**2. Bracket Notation (object["property"])**

* Useful when property names have spaces, special characters, or are stored in variables.

let student = { "first name": "Alice", grade: "A" };

// Access

console.log(student["first name"]); // Alice

// Update

student["grade"] = "A+";

console.log(student["grade"]); // A+

You can also use **variables** with bracket notation:

let key = "grade";

console.log(student[key]); // A+

**java script events**

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

**What are Events?**

* **Events** are actions or occurrences that happen in the browser.
* Examples: clicking a button, pressing a key, moving the mouse, submitting a form, loading a page, etc.

**Role of Event Listeners**

* An **event listener** is a function in JavaScript that waits for an event to occur on a specific element.
* When the event happens, the function is **triggered (executed)**.

**Example events:**

* click → when a user clicks an element
* mouseover → when mouse hovers
* keydown → when a key is pressed

**Question 2: How does addEventListener() work in JavaScript?**

**Syntax:**

element.addEventListener(event, function, useCapture);

* element → the HTML element to attach the event to
* event → type of event (like "click", "keydown")
* function → the function to run when the event happens
* useCapture (optional) → true/false (controls event bubbling, default is false)

**Example: Button Click**

<script>

// Select button element

let button = document.getElementById("myBtn");

// Add event listener

button.addEventListener("click", function() {

alert("Button was clicked!");

});

</script>

**DOM Manipulation**

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

**What is the DOM?**

* The **DOM (Document Object Model)** is a programming interface that represents an HTML or XML document as a **tree structure**.
* Each HTML element becomes a **node** (object) in this tree.
* Example:

<h1>Hello</h1>

<p>Paragraph</p>

</body>

</html>

DOM Tree (simplified):

Document

└── html

└── body

├── h1

└──

**How JavaScript interacts with the DOM**

* JavaScript can **access, modify, add, or remove** elements from the DOM.
* Example: Change a paragraph’s text:

<p id="demo">Old Text</p>

<script>

document.getElementById("demo").innerText = "New Text!";

</script>

**Output on page:**

New Text!

So, DOM lets JavaScript **dynamically update content, styles, and structure** of a webpage.

**Question 2: Methods to Select Elements from the DOM**

**1. getElementById()**

* Selects **one element** by its id.
* Returns a single element (or null if not found).

<p id="para">Hello</p>

<script>

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

console.log(element.innerText); // Hello

</script>

**2. getElementsByClassName()**

* Selects **all elements** with a given class name.
* Returns an **HTMLCollection** (like an array, but not exactly).

<p class="text">First</p>

<p class="text">Second</p

<script>

let items = document.getElementsByClassName("text");

console.log(items[0].innerText); // First

console.log(items[1].innerText); // Second

</script>

**3. querySelector()**

* Selects the **first element** that matches a **CSS selector**.
* Very flexible (you can use id, class, or any CSS rule).

<p class="note">Note 1</p>

<p class="note">Note 2</p>

<script>

let item = document.querySelector(".note");

console.log(item.innerText); // Note 1 (only first match)

</script>

**JavaScript Timing Events (setTimeout, setInterval)**

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

**setTimeout()**

* Runs a function **once**, after a specified delay (in milliseconds).
* Syntax:

setTimeout(function, delay);

* Example: Run a function after 3 seconds:

setTimeout(() => {

console.log("Hello after 3 seconds!");

}, 3000);

**setInterval()**

* Runs a function **repeatedly**, at a specified interval (in milliseconds).
* Syntax:

setInterval(function, interval);

* Example: Run every 2 seconds:

setInterval(() => {

console.log("Repeating every 2 seconds...");

}, 2000);

**Key Difference**

| **Feature** | **setTimeout()** | **setInterval()** |
| --- | --- | --- |
| Execution | Runs **once** after delay | Runs **repeatedly** at interval |
| Use Case | Delayed actions (e.g., show popup after 5s) | Repeated tasks (e.g., live clock, animations) |

**Question 2: Example of setTimeout() with 2 seconds delay**

<!DOCTYPE html>

<html>

<head>

<title>setTimeout Example</title>

</head>

<body>

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

<script>

let button = document.getElementById("myBtn");

button.addEventListener("click", function() {

setTimeout(() => {

alert("This alert shows after 2 seconds!");

}, 2000); // 2000 ms = 2 seconds

});

</script>

</body>

</html>

**JavaScript Error Handling**

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

Error handling is the process of **detecting, catching, and managing errors** in a program so it doesn’t crash unexpectedly.

In JavaScript, we use the **try…catch…finally** construct for error handling.

**Syntax:**

try {

// Code that may throw an error

} catch (error) {

// Code that runs if an error occurs

} finally {

// Code that always runs (optional)

}

**Explaation:**

* **try block** → contains code that may throw an error.
* **catch block** → handles the error (gets the error object).
* **finally block** → (optional) runs **no matter what**, whether an error occurred or not.

**Example:**

try {

let num = 10;

console.log(num.toUpperCase()); // Error: toUpperCase is not a function

} catch (err) {

console.log("An error occurred: " + err.message);

} finally {

console.log("This block always runs.");

}

**Output:**

An error occurred: num.toUpperCase is not a function

This block always runs.

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

Error handling is important because it:

1. **Prevents application crashes**
   * Without error handling, a single bug can stop the entire program.
2. **Improves user experience**
   * Instead of showing a blank screen or broken app, you can show friendly error messages.
3. **Makes debugging easier**
   * Errors can be logged and tracked for fixing.
4. **Handles unexpected situations gracefully**
   * For example, invalid user input, network failure, missing data, etc.
5. **Ensures reliability**
   * Applications become more stable and trustworthy when errors are managed.