Module 8) JavaScript

JavaScript Introduction

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

JavaScript is a **high-level, interpreted programming language** primarily used to create **interactive and dynamic content on web pages**. It is one of the core technologies of web development, alongside **HTML** and **CSS**.

◆ **Role of JavaScript in Web Development:**

- **HTML** provides the **structure**.
- **CSS** handles the **styling/layout**.
- **JavaScript** adds **functionality and behavior**, such as:
  - Form validation
  - Interactive elements (dropdowns, sliders, modals)
  - Fetching data from servers using APIs (AJAX/Fetch)
  - Real-time updates (e.g., chat apps)
  - Animations and dynamic UI changes

JavaScript runs **in the browser**, enabling real-time interaction without requiring a page reload.

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

Feature	JavaScript	Python	Java
Execution	Runs in browser (client-side)	Runs on server (backend)	Runs on JVM (Java Virtual Machine)
Syntax	Loosely typed, uses `{}`, `;`	Readable, indentation-based	Strict, uses class-based structure
Use in Web Dev	Frontend interactivity	Backend scripting (with frameworks)	Backend apps, Android development
Compilation	Interpreted	Interpreted	Compiled to bytecode

Feature	JavaScript	Python	Java
Typing	Dynamically typed	Dynamically typed	Statically typed
Object Orientation	Prototype-based	Class-based	Class-based

Summary: JavaScript is mainly used for browser-based interactivity, while Python and Java are more commonly used for backend development or application software.

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

🔹 `<script>` Tag Usage

The `<script>` tag is used in HTML to include JavaScript code.

a) Inline JavaScript (directly in HTML):

```
html
CopyEdit
<script>
  alert("Hello from JavaScript!");
</script>
```

b) External JavaScript File (recommended for large projects):

You can write your JavaScript in a `.js` file and link it to the HTML using the `src` attribute.

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

🔽 Place the `<script>` tag **before the closing `</body>` tag** to ensure the HTML content loads before JavaScript executes.

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 values**. You can think of them as containers that hold information you can use and modify later.

◆ **Variable Declaration Methods:**

1. **`var`** – Function-scoped (older way):

```
javascript
CopyEdit
var name = "Alice";
```

2. **`let`** – Block-scoped (modern and preferred for variables that change):

```
javascript
CopyEdit
let age = 25;
```

3. **`const`** – Block-scoped and **cannot be reassigned** after declaration:

```
javascript
CopyEdit
const pi = 3.14159;
```

✓ Use `let` for changeable variables and `const` for constants or fixed references.

✓ 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):**

Type	Example
String	`"Hello"` or `'World'`

Type	Example
Number	`42`, `3.14`
Boolean	`true`, `false`
Undefined	A variable declared but not assigned a value
Null	Intentional empty value
BigInt	`1234567890123456789012345678901234567890n`
Symbol	`Symbol('id')`

```
javascript
CopyEdit
let name = "John";       // String
let age = 30;            // Number
let isLoggedIn = true;   // Boolean
let score;               // Undefined
let value = null;        // Null
let big = 1234567890123456789012345678901234567890n;  // BigInt
let sym = Symbol("id");  // Symbol
```

◆ 2. Non-Primitive (Reference) Types:

Type	Example
Object	`{name: "Alice", age: 25}`
Array	`["apple", "banana", "cherry"]`
Function	`function greet() {}`

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

Feature	`undefined`	`null`
Meaning	Variable declared but **not assigned**	Intentional **absence of a value**
Type	`undefined`	`object` (this is a known JavaScript quirk)
Set By	JavaScript engine automatically	Developer manually
Example	`let x;` → `x` is `undefined`	`let y = null;` → `y` is `null`

```
javascript
CopyEdit
let a;
console.log(a); // undefined

let b = null;
console.log(b); // null
```

🔥 **Use `null`** when you want to clear or reset a value on purpose.

JavaScript Operators

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

JavaScript supports several types of operators. Below are the main ones:

◆ 1. Arithmetic Operators (used for mathematical calculations)

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`10 - 4`	`6`
`*`	Multiplication	`6 * 2`	`12`
`/`	Division	`8 / 2`	`4`
`%`	Modulus (remainder)	`10 % 3`	`1`
`**`	Exponentiation	`2 ** 3`	`8`

◆ 2. Assignment Operators (assign values to variables)

Operator	Description	Example	Equivalent To
`=`	Assign	`x = 5`	
`+=`	Add and assign	`x += 3`	`x = x + 3`
`-=`	Subtract and assign	`x -= 2`	`x = x - 2`
`*=`	Multiply and assign	`x *= 4`	`x = x * 4`
`/=`	Divide and assign	`x /= 2`	`x = x / 2`

◆ 3. Comparison Operators (used for comparing values)

Operator	Description	Example	Result
==	Equal to	5 == "5"	true
===	Strict equal to (type + value)	5 === "5"	false
!=	Not equal to	5 != "5"	false
!==	Strict not equal to	5 !== "5"	true
>	Greater than	8 > 5	true
<	Less than	3 < 7	true
>=	Greater than or equal	5 >= 5	true
<=	Less than or equal	4 <= 3	false

◆ 4. Logical Operators (used for combining boolean expressions)

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

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

Operator	Name	Compares	Example	Result
==	Loose equality	**Values only**	5 == "5"	true
===	Strict equality	**Values and types**	5 === "5"	false

◆ Example:

```
javascript
CopyEdit
let x = 5;
let y = "5";

console.log(x == y);   // true (only compares value)
```

```
console.log(x === y);  // false (compares value AND type)
```

✅ **Use `===` and `!==`** (strict equality) in most cases to avoid unexpected results due to type coercion.

Control Flow (If-Else, Switch)

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

◆ What is Control Flow?

Control flow refers to the **order in which the code is executed** in a program. In JavaScript, code is usually executed **from top to bottom**, but you can **change the flow** using control structures like:

- `if-else`
- `switch`
- `for`, `while` loops
- `break`, `continue`, etc.

◆ `if-else` Statement:

The `if-else` statement is used to execute **different blocks of code** depending on whether a condition is **true** or **false**.

◆ Syntax:

```
javascript
CopyEdit
if (condition) {
  // code to run if condition is true
} else {
  // code to run if condition is false
}
```

◆ Example:

```
javascript
CopyEdit
let age = 20;

if (age >= 18) {
  console.log("You are eligible to vote.");
} else {
  console.log("You are not eligible to vote.");
}
```

✅ If `age` is 20, the output will be: **"You are eligible to vote."**

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

◆ What is a `switch` statement?

A `switch` statement is used to **perform different actions based on multiple possible values** of a single variable or expression.

◆ Syntax:

```
javascript
CopyEdit
switch (expression) {
  case value1:
    // code to run if expression === value1
    break;
  case value2:
    // code to run if expression === value2
    break;
  default:
    // code to run if no case matches
}
```

◆ Example:

```
javascript
CopyEdit
let day = "Tuesday";

switch (day) {
  case "Monday":
    console.log("Start of the work week");
    break;
  case "Tuesday":
    console.log("Second day of the week");
    break;
  case "Friday":
    console.log("End of the work week");
    break;
  default:
    console.log("Just another day");
}
```

✅ Output: **"Second day of the week"**

◆ When to use `switch` instead of `if-else`:

- When you need to compare **one variable** against **many exact values**.

- When you want cleaner, more organized code instead of writing multiple `else if` conditions.

✅ Use `switch` when checking **multiple fixed options** (e.g., days, commands, menu choices).

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 used to **repeat a block of code** multiple times until a condition is met.

◆ 1. `for` loop

Used when the number of iterations is known.

Syntax:

```
javascript
CopyEdit
for (initialization; condition; increment) {
  // code to run
}
```

Example:

```
javascript
CopyEdit
for (let i = 1; i <= 5; i++) {
  console.log("Count:", i);
}
```

✅ Output: `1 2 3 4 5`

◆ 2. `while` loop

Used when the number of iterations is unknown and depends on a condition.

Syntax:

```
javascript
CopyEdit
while (condition) {
  // code to run
}
```

Example:

```
javascript
CopyEdit
let i = 1;
while (i <= 5) {
  console.log("While loop count:", i);
  i++;
}
```

✅ Output: `1 2 3 4 5`

◆ 3. `do-while` loop

Same as `while`, but **guarantees the loop runs at least once**, even if the condition is false.

Syntax:

```
javascript
CopyEdit
do {
  // code to run
} while (condition);
```

Example:

```
javascript
CopyEdit
let i = 1;
do {
  console.log("Do-while loop count:", i);
  i++;
} while (i <= 5);
```

✅ Output: `1 2 3 4 5`

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

Feature	`while` loop	`do-while` loop
Condition checked	Before executing the loop	After executing the loop
Minimum runs	May not run at all if condition is false	Always runs **at least once**

Feature	`while` loop	`do-while` loop
Use case	When loop should run only if condition is true at the start	When loop should run at least once

---

◆ Example to highlight the difference:

```
javascript
CopyEdit
let x = 10;

while (x < 5) {
  console.log("While loop runs");
}

do {
  console.log("Do-while loop runs");
} while (x < 5);
```

✅ Output:

```
vbnet
CopyEdit
Do-while loop runs
```

(The `while` loop doesn't run at all, but the `do-while` loop runs once.)

Functions

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

◆ What is a Function?

A **function** is a **block of reusable code** designed to perform a specific task. You can **call** it whenever you need that task done.

◆ Syntax for Declaring a Function:

```
javascript
CopyEdit
function greet() {
  console.log("Hello, world!");
}
```

◆ Syntax for Calling a Function:

```
javascript
CopyEdit
greet(); // Output: Hello, world!
```

✅ Functions help avoid repeating code and make your program more organized and modular.

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

Feature	Function Declaration	Function Expression
Syntax	`function name() { }`	`const name = function() { }`
Hoisting	**Yes** – can be called before it's defined	**No** – must be defined before being called
Name	Always has a name	Can be named or anonymous

◆ Example of Function Declaration:

```
javascript
CopyEdit
function sayHi() {
  console.log("Hi!");
}
```

```
sayHi(); // Works even if placed before declaration
```

◆ **Example of Function Expression:**

```
javascript
CopyEdit
const sayHello = function() {
  console.log("Hello!");
};
sayHello(); // Must be placed after definition
```

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

◆ **Parameters:**

- Parameters are **placeholders** for values passed into a function.
- They allow functions to be **dynamic** and work with different inputs.

◆ **Return Values:**

- A function can **return** a value using the `return` keyword.
- The returned value can be stored or used elsewhere.

◆ **Example:**

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

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

- `a` and `b` are **parameters**.
- `add(5, 3)` passes **arguments** to the function.
- `return a + b;` sends back the **result**.

✅ A function can have **no parameters**, **multiple parameters**, or **no return value**, depending on its purpose.

Arrays

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

◆ What is an Array?

An **array** is a **special variable** that can hold **multiple values** at once, stored in an **ordered list**. Arrays are useful for grouping related data.

◆ Example:

```
javascript
CopyEdit
let fruits = ["apple", "banana", "mango"];
```

- This array contains 3 elements: `"apple"`, `"banana"`, and `"mango"`.
- Indexes start from 0: `fruits[0]` is `"apple"`.

◆ How to Declare and Initialize:

```
javascript
CopyEdit
let numbers = [1, 2, 3, 4, 5];           // using array literal
let emptyArray = [];                     // empty array
let colors = new Array("red", "blue");  // using Array constructor
```

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

These methods allow you to **add** or **remove** items from arrays easily.

◆ `push()` – Adds item to the end of the array

```
javascript
CopyEdit
let fruits = ["apple", "banana"];
fruits.push("orange");
console.log(fruits); // ["apple", "banana", "orange"]
```

◆ `pop()` – Removes the last item

```
javascript
CopyEdit
let fruits = ["apple", "banana", "orange"];
fruits.pop();
console.log(fruits); // ["apple", "banana"]
```

◆ `shift()` – Removes the first item

```
javascript
CopyEdit
let fruits = ["apple", "banana"];
fruits.shift();
console.log(fruits); // ["banana"]
```

◆ `unshift()` – Adds item to the beginning

```
javascript
CopyEdit
let fruits = ["banana"];
fruits.unshift("apple");
console.log(fruits); // ["apple", "banana"]
```

✅ Summary Table:

Method	Action	Affects Start or End?
`push()`	Add item to the end	End
`pop()`	Remove item from the end	End
`shift()`	Remove item from the start	Start
`unshift()`	Add item to the start	Start

Objects

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

◆ What is an Object?

An **object** in JavaScript is a data structure used to store **key-value pairs**. Each key is a string (or symbol), and each key maps to a value.

Example:

```
javascript
CopyEdit
let person = {
  name: "Rahul",
  age: 25,
  isStudent: false
};
```

◆ How are Objects Different from Arrays?

Feature	Object	Array
Structure	Stores data as **key-value pairs**	Stores data as a **list of values**
Access method	Access using **keys**	Access using **index numbers**
Use case	Best for representing **entities**	Best for **lists or collections**
Example	`person.name` → `"Rahul"`	`fruits[0]` → `"Apple"`

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

◆ Dot Notation

Used when the property name is a valid identifier (no spaces or special characters).

Example:

```
javascript
CopyEdit
let car = {
  brand: "Toyota",
```

```
  year: 2020
};

// Access
console.log(car.brand); // "Toyota"

// Update
car.year = 2022;
console.log(car.year);  // 2022
```

◆ Bracket Notation

Used when:

- The property name has spaces or special characters.
- The property name is stored in a variable.

Example:

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

// Access
console.log(car["car brand"]); // "Toyota"

// Update
car["year"] = 2023;
console.log(car["year"]); // 2023

// Access using a variable
let key = "year";
console.log(car[key]); // 2023
```

JavaScript Event

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

◆ What are JavaScript Events?

Events are actions that occur in the browser, usually triggered by the user. Common examples include:

- Clicking a button
- Hovering over a link
- Typing into a form
- Loading a page

These events allow your web page to **respond** dynamically to user interaction.

◆ Role of Event Listeners

An **event listener** is a function that **waits for an event** to happen, and **executes a block of code** when that event occurs.

Example scenario:

- When a user **clicks** a button, an event listener runs a function that changes the text on the page.

✅ Question 2: How does the `addEventListener()` method work in JavaScript?

◆ Syntax:

```
javascript
CopyEdit
element.addEventListener("event", function);
```

- **`element`** – The HTML element to watch
- **`event`** – The event type (e.g., `"click"`, `"mouseover"`)
- **`function`** – The code to run when the event happens

◈ Example: Click a button to change text

✅ HTML:

```
html
CopyEdit
<button id="myBtn">Click Me</button>
<p id="message">Hello!</p>
```

✅ JavaScript:

```
javascript
CopyEdit
document.getElementById("myBtn").addEventListener("click", function() {
  document.getElementById("message").textContent = "You clicked the button!";
});
```

🔍 Explanation:

- The button with ID `myBtn` listens for a **click** event.
- When clicked, it runs a function that changes the text of the `<p>` element.

✅ Summary:

Term	Description
Event	An action like click, hover, input, etc.
Event Listener	A function that runs in response to an event
addEventListener	Method to attach an event to an element

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 **tree-like structure** created by the browser that represents the entire content of a webpage.
- It turns all HTML elements into **JavaScript objects**, allowing JavaScript to **access and manipulate** the content, structure, and style of the page.

Example Structure:

```
html
CopyEdit
<html>
  <body>
    <h1>Hello</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

This gets converted into a tree where `html`, `body`, `h1`, and `p` are nodes in the DOM.

◆ How JavaScript interacts with the DOM:

JavaScript can:

- Read and change HTML content (`innerHTML`)
- Update styles (`style.color`)
- Handle events like clicks (`addEventListener`)
- Add or remove elements dynamically

Example:

```
javascript
CopyEdit
document.getElementById("myTitle").textContent = "New Title";
```

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

◆ `getElementById()`

- Selects a **single element** by its **ID**.
- Returns one element.

```
html
CopyEdit
<p id="demo">Hello</p>
javascript
CopyEdit
let element = document.getElementById("demo");
console.log(element.textContent); // "Hello"
```

◆ `getElementsByClassName()`

- Selects **all elements** with a specific **class name**.
- Returns an **HTMLCollection** (like an array).

```
html
CopyEdit
<p class="note">Note 1</p>
<p class="note">Note 2</p>
javascript
CopyEdit
let elements = document.getElementsByClassName("note");
console.log(elements[0].textContent); // "Note 1"
```

◆ `querySelector()`

- Selects the **first matching element** using a **CSS selector** (ID, class, tag).
- More flexible than `getElementById`.

```
html
CopyEdit
<p class="info">First</p>
<p class="info">Second</p>
javascript
CopyEdit
let element = document.querySelector(".info");
console.log(element.textContent); // "First"
```

JavaScript Timing Events (setTimeout, setInterval)

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

◆ `setTimeout()`

- Executes a function **once** after a specified delay (in **milliseconds**).
- Used when you want to **delay** an action.

Syntax:

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

◆ `setInterval()`

- Repeatedly executes a function **at regular intervals**.
- Keeps running until stopped using `clearInterval()`.

Syntax:

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

◆ Use in Timing Events:

These functions help:

- Show a message after a delay.
- Create animations or countdowns.
- Periodically update data or UI.

✅ Question 2: Example using `setTimeout()` to delay an action by 2 seconds

```
html
CopyEdit
<button onclick="showMessage()">Click Me</button>
<p id="output"></p>

<script>
```

```
  function showMessage() {
    setTimeout(function() {
      document.getElementById("output").textContent = "Action performed after
2 seconds!";
    }, 2000); // 2000 ms = 2 seconds
  }
</script>
```

🔍 What happens:

- When the button is clicked, the `showMessage()` function runs.
- `setTimeout()` delays the action inside by 2 seconds.
- After 2 seconds, the message appears in the `<p>` tag.

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 in JavaScript allows developers to **catch and respond to runtime errors** (such as undefined variables or failed API calls) **without crashing the program**.

JavaScript provides `try`, `catch`, and `finally` blocks to manage errors gracefully.

◆ Structure:

```
javascript
CopyEdit
try {
  // Code that might throw an error
} catch (error) {
  // Code to run if there is an error
} finally {
  // Code that always runs (optional)
}
```

◆ Example:

```
javascript
CopyEdit
try {
  let num = 5;
  let result = num.toUpperCase();  // Error: toUpperCase() is not a function
for numbers
} catch (error) {
  console.log("An error occurred:", error.message);
} finally {
  console.log("This will run no matter what.");
}
```

Output:

```
vbnet
CopyEdit
An error occurred: num.toUpperCase is not a function
```

```
This will run no matter what.
```

◆ Explanation:

- `try` block runs code that might cause an error.
- `catch` block handles the error if one occurs.
- `finally` block runs **always**, whether there's an error or not — useful for cleanup (like closing files or stopping loaders).

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

◆ Reasons Why Error Handling Is Important:

1. **Prevents Program Crashes:**
   - Without error handling, unexpected problems can stop the entire script.
2. **Improves User Experience:**
   - You can show user-friendly error messages instead of breaking the page.
3. **Debugging:**
   - Catch blocks help log specific errors, making it easier to fix issues.
4. **Control Flow:**
   - Allows you to decide what the application should do when an error occurs — retry, skip, alert, etc.
5. **Security:**
   - Prevents the application from exposing sensitive data or behaving unpredictably during failures.