Module 8) JavaScript

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 widely used for adding interactivity and dynamic functionality to web pages. It is one of the core technologies of web development, alongside HTML (HyperText Markup Language) and CSS (Cascading Style Sheets).

JavaScript is a versatile language that can be used for both client-side and server-side development. Initially designed for web browsers, it has now expanded to other environments, including mobile applications, desktop applications, and even backend development (using Node.js).

Role of JavaScript in Web Development

JavaScript plays a crucial role in modern web development. Its main responsibilities include:

1. **Enhancing Interactivity**
    - Enables dynamic content updates without reloading the page (e.g., auto-suggest in search bars, form validation).
    - Handles user inputs and responds to events like clicks, keystrokes, and mouse movements.
2. **Manipulating the DOM (Document Object Model)**
    - JavaScript allows developers to modify the structure and content of a web page dynamically.
    - Example: Changing text, adding or removing elements, modifying styles.
3. **Handling Asynchronous Operations**
    - JavaScript enables asynchronous data fetching using **AJAX (Asynchronous JavaScript and XML)** and **Fetch API**, which helps in creating smooth user experiences.
    - Example: Loading new content in social media feeds without refreshing the page.
4. **Client-Side Validation**

- Prevents users from submitting incorrect or incomplete data by validating forms before sending data to the server.

5. **Creating Animations and Effects**
    - JavaScript, along with CSS animations and libraries like GSAP (GreenSock Animation Platform), can create visually appealing effects.
6. **Building Web Applications**
    - JavaScript is the backbone of modern web applications, especially with frameworks/libraries like React.js, Angular, and Vue.js.
    - Single Page Applications (SPAs) heavily rely on JavaScript for seamless transitions between views.
7. **Enabling Server-Side Development**
    - With Node.js, JavaScript can be used to build backend services, manage databases, and handle authentication.
8. **Game Development and Web APIs**
    - JavaScript is used in game development (e.g., using the Canvas API) and can interact with various web APIs, including Geolocation, WebSockets, and WebRTC for real-time communication.

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

1. Execution Environment

- **JavaScript**: Primarily runs in web browsers but can also be executed on the server using **Node.js**.
- **Python**: Runs on almost any platform, used for scripting, automation, AI, and web backend development.
- **Java**: Runs on the **Java Virtual Machine (JVM)**, making it platform-independent.

2. Syntax & Readability

- **JavaScript**: Uses curly braces {} and semicolons ; (though optional). It is relatively simple but can be inconsistent.
- **Python**: Uses indentation for blocks, making it highly readable and beginner-friendly.
- **Java**: More verbose, requiring explicit types and detailed syntax.

3. Type System

- **JavaScript**: Dynamically typed, meaning variables do not need explicit types (var, let, const).
- **Python**: Also dynamically typed but more structured than JavaScript.
- **Java**: Statically typed, meaning variable types must be declared (int, String, boolean).

4. Performance & Speed

- **JavaScript**: Faster for web applications due to just-in-time (JIT) compilation but not ideal for CPU-intensive tasks.
- **Python**: Slower due to interpretation and **GIL (Global Interpreter Lock)**, which limits multi-threading.
- **Java**: Generally the fastest because of **JIT compilation** and better multi-threading support.

5. Object-Oriented Programming (OOP)

- **JavaScript**: Uses **prototype-based OOP**, meaning objects inherit directly from other objects.
- **Python**: Uses **class-based OOP**, similar to Java but more flexible.
- **Java**: Fully **class-based OOP**, requiring everything to be inside classes.

6. Concurrency & Multi-threading

- **JavaScript**: Uses an **event-driven, single-threaded model** with **async/await & Promises**.
- **Python**: Supports multi-threading, but **GIL** restricts full parallelism in CPU-bound tasks.
- **Java**: Fully supports multi-threading and is optimized for large, multi-threaded applications.

7. Use Cases

- **JavaScript**: Best for **web development** (front-end and back-end with Node.js), mobile apps (React Native), and game development.
- **Python**: Popular for **AI, machine learning, data science, automation, and backend development**.
- **Java**: Preferred for **enterprise applications, large-scale systems, Android development, and financial applications**.

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

The <script> tag in HTML is used to embed JavaScript code within an HTML document. It allows developers to add interactivity, dynamic content, and client-side functionality to web pages.

1. Types of <script> Usage in HTML

JavaScript can be included in an HTML file in two ways:

1. **Inline JavaScript (Internal)**
2. **External JavaScript (Linked File)**

1. Inline JavaScript (Internal Script)

JavaScript code can be written directly inside the <script> tag within the HTML file.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Inline JavaScript</title>
  <script>
    function showMessage() {
      alert("Hello! This is an inline JavaScript message.");
    }
  </script>
</head>
<body>
  <button onclick="showMessage()">Click Me</button>
</body>
</html>
```

When to Use?

- Small scripts or quick functionalities.

- Not recommended for large-scale applications as it makes HTML harder to maintain.

2. Linking an External JavaScript File

For better organization and maintainability, JavaScript is often written in a separate file and linked to the HTML document using the <script> tag.

How to Link an External JavaScript File?

- Use the src attribute in the <script> tag to specify the file path.
- Place the <script> tag inside <head> or before </body> for better performance.

Example:

Step 1: Create an External JavaScript File (script.js)

```
function showMessage() {
  alert("Hello! This is an external JavaScript file.");
}
```

Step 2: Link the File in HTML (index.html)

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

When to Use?

- Recommended for large projects and reusable scripts.
- Improves maintainability and code separation.

3. Best Practices for Using the <script> Tag

1. Place <script> Before </body> (For Performance)

Instead of placing <script> in <head>, place it **just before </body>** to allow the page to load faster.

<body>

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

</body>

</html>

- This ensures the HTML content loads before JavaScript executes.

2. Use defer or async Attributes

- **defer**: Ensures the script runs only after the HTML is fully loaded.
- **async**: Loads script asynchronously (useful for independent scripts).

Example:

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

or

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

3. Keep JavaScript Separate from HTML

- Avoid writing JavaScript directly inside HTML elements (like onclick attributes).
- Instead, use **event listeners** in JavaScript.

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 containers for storing data values. They allow developers to store, retrieve, and manipulate data throughout their code.

Declaring Variables in JavaScript

In JavaScript, variables can be declared using var, let, and const.

1. Using var (Old Method, Function Scoped)

- Can be redeclared and updated.
- Function-scoped (not block-scoped).
- Hoisted, but initialized as undefined.

Example:

```
var name = "John";

console.log(name); // Output: John

var name = "Doe";  // Redeclaration allowed

console.log(name); // Output: Doe
```

2. Using let (Modern, Block Scoped)

- Can be updated but not redeclared within the same scope.
- Block-scoped (only accessible inside {} where it is declared).
- Hoisted, but not initialized (causes ReferenceError if accessed before declaration).

Example:

```
let age = 25;

console.log(age); // Output: 25

age = 30; // Allowed (update)

console.log(age); // Output: 30

// let age = 35;  //  Error: Cannot redeclare 'age' in the same scope
```

3. Using const (Constant, Block Scoped)

- Cannot be updated or redeclared.
- Must be initialized at the time of declaration.

- Block-scoped like let.

Example:

```
const pi = 3.14;

console.log(pi); // Output: 3.14

// pi = 3.1416;  //  Error: Assignment to constant variable
```

Summary Table: var vs. let vs. const

Feature	**var**	**let**	**const**
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Yes (initialized as undefined)	Yes (but not initialized)	Yes (but not initialized)
Redeclaration	Allowed	Not allowed	Not allowed
Reassignment	Allowed	Allowed	Not allowed

Question 2: Explain the Different Data Types in JavaScript. Provide Examples for Each.

JavaScript has **8 primitive data types** and **1 non-primitive type (Objects)**.

1. Primitive Data Types (Immutable)

These are basic types that hold **single values**.

1. String

Used for storing text (written inside **quotes**).

```
let name = "Alice";

let greeting = 'Hello, World!';

console.log(name, greeting);
```

2. Number

Represents both **integers** and **floating-point numbers**.

```
let age = 25;

let price = 99.99;

console.log(age, price);
```

3. Boolean

Represents **true** or **false** values.

```
let isLoggedIn = true;
let isAdmin = false;
console.log(isLoggedIn, isAdmin);
```

4. Undefined

A variable that is declared but **not assigned** a value.

```
let user;
console.log(user); // Output: undefined
```

5. Null

A special value that **represents an empty or unknown value**.

```
let data = null;
console.log(data); // Output: null
```

6. Symbol (ES6)

Used for creating **unique identifiers** (useful in objects).

```
const sym1 = Symbol("id");
const sym2 = Symbol("id");
console.log(sym1 === sym2); // Output: false (Symbols are unique)
```

7. BigInt (ES11)

Used for **very large integers** beyond Number.MAX_SAFE_INTEGER.

```
let bigNum = 123456789012345678901234567890n;
console.log(bigNum);
```

2. Non-Primitive Data Type (Mutable)

8. Object

Used to store **key-value pairs**.

```
let person = {
  name: "John",
  age: 30,
  isStudent: false
};
```

console.log(person.name); // Output: John

Special Cases

- **Arrays** (special objects)

Functions (also objects in JavaScript)

let colors = ["Red", "Green", "Blue"]; // Array

console.log(colors[0]); // Output: Red

function greet() {

  return "Hello!";

}

console.log(greet()); // Output: Hello!

Question 3: What Is the Difference Between undefined and null in JavaScript?

Feature	**undefined**	**null**
Meaning	A variable is declared but has **no value assigned**	A variable is **explicitly set** to "nothing"
Type	undefined (primitive type)	object (typeof null returns "object" due to legacy reasons)
Assigned By	JavaScript (default)	Developer (explicitly)
Example	let x; console.log(x); // undefined	let y = null; console.log(y); // null

Example Code

let a;

console.log(a); // Output: undefined (variable declared but not assigned)

let b = null;

console.log(b); // Output: null (explicitly assigned)

Checking for null vs undefined

console.log(typeof undefined); // Output: "undefined"

console.log(typeof null); // Output: "object" (legacy bug in JavaScript)

Key Takeaways

- **undefined** means "not assigned a value".
- **null** is an intentional "empty value".
- Both are **falsy values**, meaning they behave as false in boolean contexts.

Example:

if (!undefined) console.log("undefined is falsy"); // Executes

if (!null) console.log("null is falsy"); // Executes

3)JavaScript Operators

Question 1: What Are the Different Types of Operators in JavaScript?

Operators in JavaScript are **symbols** that perform operations on variables and values. There are several types:

1. Arithmetic Operators (+, -, *, /, %, **, ++, --)

Used to perform **mathematical calculations**.

Operator	Description	Example	Output
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	6 * 2	12
/	Division	10 / 2	5
%	Modulus (Remainder)	10 % 3	1
**	Exponentiation	2 ** 3	8
++	Increment	let a = 5; a++;	6
--	Decrement	let b = 5; b--;	4

Example:

```
let x = 10;

let y = 3;

console.log(x + y);  // Output: 13

console.log(x % y);  // Output: 1
```

2. Assignment Operators (=, +=, -=, *=, /=, %=, **=)

Used to **assign values** to variables.

Operator	Description	Example	Equivalent To
=	Assigns a value	x = 10	x = 10
+=	Adds and assigns	x += 5	x = x + 5
-=	Subtracts and assigns	x -= 3	x = x - 3

Operator	Description	Example	Equivalent To
*=	Multiplies and assigns	x *= 2	x = x * 2
/=	Divides and assigns	x /= 2	x = x / 2
%=	Modulus and assigns	x %= 3	x = x % 3
**=	Exponentiation and assigns	x **= 2	x = x ** 2

Example:

```
let num = 10;
num += 5; // Equivalent to num = num + 5;
console.log(num); // Output: 15
```

3. Comparison Operators (==, ===, !=, !==, >, <, >=, <=)

Used to **compare two values** and return true or false.

Operator	Description	Example	Output
==	Equal (loose comparison)	5 == "5"	true
===	Strictly Equal (checks type too)	5 === "5"	false
!=	Not Equal	10 != 5	true
!==	Strictly Not Equal	10 !== "10"	true
>	Greater Than	8 > 5	true
<	Less Than	3 < 7	true
>=	Greater Than or Equal	6 >= 6	true
<=	Less Than or Equal	4 <= 5	true

Example:

```
console.log(10 > 5);  // Output: true
console.log(5 == "5"); // Output: true (loose comparison)
console.log(5 === "5"); // Output: false (strict comparison
```

4. Logical Operators (&&, ||, !)

Used to **combine multiple conditions**.

Operator	Description	Example	Output
&&	Logical AND (Both must be true)	(5 > 2 && 10 > 5)	true
`		`	Logical OR (At least one must be true)
!	Logical NOT (Reverses true/false)	!(5 > 2)	false

Example:

```
let age = 20;

let hasID = true;

if (age >= 18 && hasID) {

   console.log("You can enter."); // Output: You can enter.

}
```

Question 2: What Is the Difference Between == and === in JavaScript?

Operator	Description	Type Checking	Example	Output
==	Loose equality (checks value only)	No	5 == "5"	true
===	Strict equality (checks value & type)	Yes	5 === "5"	false

Example:

```
console.log(5 == "5");  // Output: true (loose comparison)

console.log(5 === "5"); // Output: false (strict comparison)
```

Why Use === Instead of ==?

- == can lead to **unexpected results** due to **type coercion** (automatic type conversion).
- === ensures both **value and type** match.

Example of Type Coercion with ==

```
console.log(false == 0);  // Output: true (0 is treated as false)

console.log(null == undefined); // Output: true

console.log("10" == 10); // Output: true
```

Using === to Avoid Bugs

```
console.log(false === 0);  // Output: false

console.log(null === undefined); // Output: false

console.log("10" === 10); // Output: false
```

Key Takeaways

- Use **==** when you only care about values, not types.
- Use **===** for precise comparisons to avoid unexpected type coercion.

4)Control Flow (If-Else, Switch)

Question 1: What Is Control Flow in JavaScript? Explain How if-else Statements Work with an

Control flow in JavaScript **determines the order in which statements are executed** in a program. Normally, code runs **line by line (top to bottom)**, but using conditional statements like if-else, loops, and switch statements, we can **control the flow** of execution.

How if-else Statements Work

The if-else statement **executes different blocks of code based on a condition**.

Syntax:

```
if (condition) {
   // Code executes if condition is true
} else {
// Code executes if condition is false
}
```

Example:

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

Output:

You are eligible to vote.

if-else if Ladder

When multiple conditions need to be checked, we use an if-else if ladder.

Example:

```
let score = 85;
if (score >= 90) {
```

```
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else if (score >= 70) {
  console.log("Grade: C");
} else {
  console.log("Grade: F");
}
```

Output:

Grade: B

Question 2: Describe How switch Statements Work in JavaScript. When Should You Use a switch Statement Instead of if-else?

How switch Statements Work

A switch statement **compares a value against multiple cases** and executes the matching block. It is **useful when dealing with multiple possible values of a single variable**.

Syntax:

```
switch (expression) {
  case value1:
    // Code block for value1
    break;
  case value2:
    // Code block for value2
    break;
  default:
    // Code block if no case matches
}
```

- The expression is **evaluated once**.

- It is then **compared** to each case value.
- If a match is found, the **corresponding code block executes**.
- The break statement **stops execution** after a match is found.
- The default case runs if **no match** is found.

Example of switch:

```
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("Weekend is coming!");
    break;
  default:
    console.log("It's a regular day.");
}
```

Output:

Second day of the week.

When to Use switch Instead of if-else?

Feature	**if-else**	**switch**
Best For	Complex conditions with relational operators (>, <, >=, etc.)	Multiple fixed values of a single variable
Readability	Becomes complex with many conditions	Cleaner and more structured for multiple cases
Performance	Can be slower for many conditions (checks each if)	Faster for fixed values (direct lookup)

Use switch When:

You are checking a **single variable** against multiple possible **fixed values**.
The number of cases is **large**, making if-else messy.

Use if-else When:

Conditions involve **relational operators** (>, <, !=, etc.).
The decision depends on **complex logical expressions**.

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

Question 1: Explain the Different Types of Loops in JavaScript (for, while, do-while) with Examples.

Loops in JavaScript **execute a block of code multiple times** until a specified condition is met. The three main types of loops are:

1. for Loop

Used when the number of iterations is known beforehand.

Syntax:

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

- **Initialization**: Sets up a loop variable.
- **Condition**: The loop runs while this condition is true.
- **Update**: Changes the loop variable after each iteration.

Example: Print numbers 1 to 5

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

Output:

1

2

3

4

5

2. while Loop

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

Syntax:

```
while (condition) {
  // Code executes while condition is true
}
```

Example: Print numbers 1 to 5

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

Output:

1

2

3

4

5

3. do-while Loop

Executes at least once, then checks the condition.

Syntax:

```
do {
  // Code executes at least once
} while (condition);
```

Example: Print numbers 1 to 5

```
let i = 1;
do {
  console.log(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**
Execution	May never execute if condition is false initially	Executes **at least once**, even if the condition is false
Condition Check	Checked **before** executing the loop body	Checked **after** executing the loop body
Use Case	When the loop should only run if the condition is met from the start	When the loop **must execute at least once** before checking

Example: When do-while Executes at Least Once

```
let num = 10;
while (num < 5) {
   console.log("Inside while loop"); // Will NOT execute
}
do {
   console.log("Inside do-while loop"); // Executes at least once
} while (num < 5);
```

Output:

```
Inside do-while loop
```

6)Functions

Question 1: What Are Functions in JavaScript? Explain the Syntax for Declaring and Calling a Function.

A function in JavaScript is a block of reusable code that performs a specific task. Functions help make code modular, reusable, and organized.

Syntax for Declaring a Function

A function is declared using the function keyword, followed by a name, parentheses (), and curly braces {} containing the function body.

```
function functionName() {
  // Code to execute
}
```

Calling (Executing) a Function

To execute a function, use its name followed by parentheses:

```
functionName();
```

Example: Simple Function

```
function greet() {
  console.log("Hello, World!");
}
greet();  // Function call
```

Output:

Hello, World!

Question 2: What Is the Difference Between a Function Declaration and a Function Expression?

1. Function Declaration

A function declaration is a function that is hoisted (available throughout the script, even before declaration).

Syntax:

```
function greet() {
```

```
  console.log("Hello!");
}
```

Example:

```
sayHello(); //  Works due to hoisting
function sayHello() {
  console.log("Hello!");
}
sayHello(); //  Also works
```

Output:

```
Hello!
Hello!
```

2. Function Expression

A function expression is stored in a variable and not hoisted.

Syntax:

```
const functionName = function() {
  // Function body
};
```

Example:

```
// sayHi();  Error: Cannot access before initialization
const sayHi = function() {
  console.log("Hi!");
};
sayHi(); //  Works
```

Output:

```
Hi!
```

Key Differences Between Function Declaration & Expression

Feature	Function Declaration	Function Expression
Hoisting	Hoisted (can be called before declaration)	Not hoisted (must be declared first)
Usage	Best for standard, named functions	Useful for anonymous or callback functions

Question 3: Discuss the Concept of Parameters and Return Values in Functions.

1. Function Parameters

- **Parameters** allow **input values** to be passed into a function.
- **Arguments** are the actual values passed when calling a function.

Example: Function with Parameters

```
function greet(name) {  // name is a parameter
   console.log("Hello, " + name + "!");
}
greet("Alice");  // "Alice" is an argument
```

Output:

```
Hello, Alice!
```

2. Return Values

- The return statement **sends back a value** from a function.
- A function without return outputs undefined.

Example: Function with Return Value

```
function add(a, b) {
   return a + b;  // Returns sum of a and b
}
let result = add(5, 3);
console.log(result);
```

Output:

```
8
```

7)Arrays

Question 1: What Is an Array in JavaScript? How Do You Declare and Initialize an Array?

An array in JavaScript is a collection of elements stored in a single variable. It can hold multiple values, including numbers, strings, objects, and even other arrays.

Declaring and Initializing an Array

1. Using Square Brackets [] (Recommended)

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

2. Using new Array() (Less Common)

```
let colors = new Array("Red", "Green", "Blue");
```

3. Empty Array Initialization

```
let emptyArray = [];  // Empty array
```

Accessing Array Elements

Array elements are accessed using **indexing** (starting from 0).

```
console.log(fruits[0]); // "Apple"

console.log(fruits[1]); // "Banana"
```

Modifying Array Elements

```
fruits[1] = "Mango";

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

Question 2: Explain the Methods push(), pop(), shift(), and unshift() Used in Arrays.

1. push() – Add Element to the End

Adds an element at the **end** of the array.
Returns the new array length.

Example:

```
let numbers = [1, 2, 3];

numbers.push(4);

console.log(numbers); // [1, 2, 3, 4]
```

2. pop() – Remove Element from the End

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

Example:

```
let numbers = [1, 2, 3];

let lastElement = numbers.pop();

console.log(numbers); // [1, 2]

console.log(lastElement); // 3
```

3. shift() – Remove Element from the Beginning

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

Example:

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

let firstFruit = fruits.shift();

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

console.log(firstFruit); // "Apple"
```

4. unshift() – Add Element to the Beginning

Adds an element at the **beginning** of the array.
Returns the new array length.

Example:

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

fruits.unshift("Apple");

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

8)Objects

Question 1: What Is an Object in JavaScript? How Are Objects Different from Arrays?

An object in JavaScript is a collection of key-value pairs used to store structured data. Each key (or property) is a string and is mapped to a value, which can be any data type, including functions.

Example of an Object

```
let person = {
  name: "John",
  age: 30,
  city: "New York"
};
```

Objects vs. Arrays

Feature	Objects	Arrays
Data Storage	Key-value pairs	Indexed elements
Access Method	Dot . or Bracket [] notation	Index (array[0])
Best For	Representing structured data (e.g., a person)	Lists or collections (e.g., numbers, names)

Example of an Array vs. Object

```
// Array (ordered list)
let fruits = ["Apple", "Banana", "Cherry"];
// Object (key-value pairs)
let fruitInfo = {
  name: "Apple",
  color: "Red",
  price: 10
};
```

Question 2: How to Access and Update Object Properties Using Dot Notation and Bracket Notation?

1. Accessing Properties

Using Dot Notation (Recommended)

```
console.log(person.name); // "John"
console.log(person.age);  // 30
```

Using Bracket Notation

```
console.log(person["name"]); // "John"
console.log(person["age"]);  // 30
```

Bracket notation is useful when the property name is dynamic or has spaces.

```
let key = "city";
console.log(person[key]); // "New York"
```

2. Updating Properties

Using Dot Notation

```
person.age = 35;
console.log(person.age); // 35
```

Using Bracket Notation

```
person["city"] = "Los Angeles";
console.log(person.city); // "Los Angeles"
```

3. Adding New Properties

```
person.country = "USA";       // Dot notation
person["gender"] = "Male";    // Bracket notation
console.log(person);
/* Output:
{
   name: "John",
   age: 35,
   city: "Los Angeles",
```

```
  country: "USA",
  gender: "Male"
}
*/
```

4. Deleting Properties

```
delete person.age;
console.log(person);
/* Output:
{
  name: "John",
  city: "Los Angeles",
  country: "USA",
  gender: "Male"
}
*/
```

When to Use Dot vs. Bracket Notation?

Case	Dot Notation	Bracket Notation
Property name is known and valid	Yes	Yes
Property name has spaces/special characters	No	Yes (obj["first name"])
Property name is dynamic (from a variable)	No	Yes

9)JavaScript Events

Question 1: What Are JavaScript Events? Explain the Role of Event Listeners.

What Are JavaScript Events?

JavaScript **events** are actions or occurrences that happen in the browser, such as:

- Clicking a button (click)
- Pressing a key (keydown)
- Moving the mouse (mousemove)
- Submitting a form (submit)

Events allow JavaScript to interact with users dynamically.

Role of Event Listeners

An event listener is a function that waits for an event to occur and executes a specific action when the event happens.

Without an event listener, an event does nothing.
With an event listener, JavaScript can respond to user actions.

Example of an Event Listener

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

Explanation: -

1. The addEventListener() method is used to listen for a "click" event.
2. When the button is clicked, the function **executes**, displaying an alert.

Question 2: How Does the addEventListener() Method Work in JavaScript?

Syntax of addEventListener()

element.addEventListener(event, function, useCapture);

Parameter	**Description**
event	The event type (e.g., "click", "mouseover", "keydown")
function	The function to execute when the event occurs
useCapture	(Optional) true for capturing phase, false for bubbling (default)

Example: addEventListener() in Action

1. Click Event Listener

```
<button id="myButton">Click Me</button>
<script>
document.getElementById("myButton").addEventListener("click", function() {
  console.log("Button was clicked!");
});
</script>
```

When the button is clicked, "Button was clicked!" is logged in the console.

2. Mouseover Event Listener

```
document.getElementById("myButton").addEventListener("mouseover", function() {
  console.log("Mouse is over the button!");
});
```

When the mouse hovers over the button, "Mouse is over the button!" appears in the console.

3. Keypress Event Listener

```
document.addEventListener("keydown", function(event) {
  console.log("Key pressed:", event.key);
});
```

This logs the **key** pressed on the keyboard.

Removing an Event Listener

To remove an event listener, use removeEventListener().

Example:

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

```
document.getElementById("myButton").addEventListener("click", sayHello);
document.getElementById("myButton").removeEventListener("click", sayHello);
```

The "click" event is **removed**, so clicking the button does nothing.

10)DOM Manipulation

Question 1: What Is the DOM (Document Object Model) in JavaScript? How Does JavaScript Interact with the DOM?

The Document Object Model (DOM) is a tree-like structure that represents the elements of an HTML page. It allows JavaScript to interact with and modify web pages dynamically.

When a web page loads, the browser creates a DOM representation of the page. Each HTML element becomes a node in the DOM tree.

Example: HTML and Its DOM Structure

```
<!DOCTYPE html>
<html>
<head>
  <title>My Page</title>
</head>
<body>
  <h1 id="title">Hello, World!</h1>
  <p class="text">This is a paragraph.</p>
</body>
</html>
```

In the DOM, the <html> tag is the **root node**, <body> is its child, and so on.

How Does JavaScript Interact with the DOM?

JavaScript can **access, modify, and manipulate** HTML elements using the DOM API.

Common DOM Interactions

1. **Access Elements** (getElementById, querySelector)
2. **Modify Content** (innerHTML, textContent)
3. **Change Styles** (style.property)
4. **Handle Events** (addEventListener)

Example: JavaScript Modifying the DOM

```
document.getElementById("title").textContent = "Hello, JavaScript!";
document.body.style.backgroundColor = "lightblue";
```

This **changes** the <h1> text and **modifies** the background color.

Question 2: Explain getElementById(), getElementsByClassName(), and querySelector() Used to Select Elements from the DOM.

1. getElementById() – Select by ID

Finds a single element by its id.
Returns: The element **(or null if not found).**

Example:

<h1 id="title">Hello</h1>

<script>

let heading = document.getElementById("title");

console.log(heading.textContent); // "Hello"

</script>

Use when selecting a unique element (e.g., a header, button, or form).

2. getElementsByClassName() – Select by Class Name

Finds multiple elements by their class.
Returns: A **collection** (like an array) of elements.

Example:

Paragraph 1

Paragraph 2

<script>

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

console.log(paragraphs[0].textContent); // "Paragraph 1"

</script>

Use when selecting multiple elements (e.g., list items, sections).

3. querySelector() – Select by CSS Selector

Finds the first matching element using a **CSS selector**.
More flexible than getElementById().

Example:

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

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


<script>

let firstParagraph = document.querySelector(".text");

console.log(firstParagraph.textContent); // "First paragraph"

</script>
```

Use when selecting elements with CSS-like precision (#id, .class, tag).

4. querySelectorAll() – Select Multiple Elements by CSS Selector

Finds all matching elements using a CSS selector.
Returns: A **NodeList** (like an array).

Example:

```
let allParagraphs = document.querySelectorAll(".text");

console.log(allParagraphs.length); // 2
```

11)Javascript Timing Events (setTimeout,setTimeinterval)

Question 1: Explain setTimeout() and setInterval() Functions in JavaScript. How Are They Used for Timing Events?

JavaScript provides timing functions to schedule code execution after a delay or at regular intervals.

1. setTimeout() – Delays Execution

Runs a function once after a specified time.
Time is given in milliseconds (1 sec = 1000 ms).

Syntax:

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

Example: Run Code After 3 Seconds

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

Use setTimeout() when you need a one-time delay.

2. setInterval() – Repeats Execution

Runs a function repeatedly at fixed intervals.
Keeps executing until stopped using clearInterval().

Syntax:

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

Example: Run Code Every 2 Seconds

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

Use setInterval() when you need continuous execution (e.g., updating a clock).

Stopping setTimeout() and setInterval()

- clearTimeout(timeoutID) → Stops a setTimeout().
- clearInterval(intervalID) → Stops a setInterval().

Example: Stop setInterval() After 5 Seconds

```
let count = 0;
let interval = setInterval(() => {
  console.log("Repeating action...");
  count++;
  if (count === 3) {
    clearInterval(interval);
    console.log("Stopped after 3 repetitions.");
  }
}, 1000);
```

Question 2: Example of setTimeout() to Delay an Action by 2 Seconds

Example: Show Alert After 2 Seconds

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

Example: Change Text After 2 Seconds

```
<h1 id="message">Wait for it...</h1>
<script>
setTimeout(() => {
  document.getElementById("message").textContent = "Text changed after 2 seconds!";
}, 2000);
</script>
```

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 allows us to detect, handle, and recover from errors without crashing the entire program. JavaScript provides the try...catch...finally mechanism to gracefully handle errors.

1. try Block – Code That Might Cause an Error

Encapsulates code that might throw an error.
If an error occurs, execution **jumps to the catch block` .**

2. catch Block – Handle the Error

Executes only if an error occurs in the try block.
Receives the **error object** (e.g., error.message).

3. finally Block – Always Executes

Runs regardless of errors, useful for cleanup (e.g., closing connections).

Example: Handling an Undefined Variable Error

```
try {
  console.log(unknownVariable); // This causes an error
} catch (error) {
  console.log("An error occurred:", error.message); // Catches the error
} finally {
  console.log("This will always run."); // Executes no matter what
}
```

Output:

An error occurred: unknownVariable is not defined

This will always run.

Best Practice: Use try...catch to **handle errors without breaking the script.**

Question 2: Why Is Error Handling Important in JavaScript Applications?

1. Prevents Application Crashes

- Without error handling, JavaScript stops execution on an error.
- Handling errors ensures the application **continues running smoothly**.

2. Improves User Experience

- Instead of showing **a broken page**, show a **friendly message**.
- Example:
- try {
- fetch("invalid-url");
- } catch (error) {
- alert("Oops! Something went wrong.");
- }

3. Debugging and Logging Errors

- Errors can be logged for **troubleshooting**.
- Example:
- try {
- let data = JSON.parse("invalid json");
- } catch (error) {
- console.error("JSON Parse Error:", error.message);
- }

4. Handles Asynchronous Errors in Promise and async/await

- Errors in **API calls, database operations, or file handling** need proper handling.
- Example:
- async function fetchData() {
- try {
- let response = await fetch("invalid-url");
- } catch (error) {
- console.error("API Error:", error.message);
- }
- }

fetchData();