1. JavaScript Introduction

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

Answer:
JavaScript is a **high-level, interpreted programming language** primarily used to make web pages **interactive and dynamic**. It runs directly in the web browser, allowing developers to manipulate webpage content, handle user inputs, and communicate with servers without reloading the page.

Role of JavaScript in Web Development:

1. **Client-Side Interactivity:**
   It adds interactive behavior like dropdown menus, sliders, pop-ups, and animations.

2. **DOM Manipulation:**
   JavaScript can dynamically change HTML and CSS elements (e.g., hiding or showing elements, changing styles).

3. **Form Validation:**
   It validates user input before sending data to the server (e.g., checking if an email field is filled correctly).

4. **Asynchronous Communication (AJAX):**
   It allows web pages to load or update content **without refreshing** the entire page.

5. **Frameworks and Libraries:**
   Libraries like **React, Angular, Vue.js** are built on JavaScript to create modern, efficient web applications.

6. **Server-Side Development:**
   With **Node.js**, JavaScript is also used on the **server-side**

to build backend systems.

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

Feature	JavaScript	Python	Java
Type	Interpreted scripting language	Interpreted high-level language	Compiled and strongly typed
Execution Environment	Runs mainly in browsers (client-side) and Node.js (server-side)	Runs on Python interpreter	Runs on JVM (Java Virtual Machine)
Syntax	C-like but loosely typed	Simple, indentation-based	Strictly typed and verbose
Typing	Dynamically typed	Dynamically typed	Statically typed
Use Case	Web development (frontend + backend)	Data science, AI, scripting, backend	Enterprise apps, Android development
Compilation	Interpreted at runtime by browser	Interpreted	Compiled into bytecode

In short:

- JavaScript is mainly used for **web interactivity**.
- Python is often used for **data analysis, AI, and scripting**.

- Java is used for **large-scale, enterprise, and mobile applications**.

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

Answer:
The <script> tag in HTML is used to **embed or reference JavaScript code** in a webpage. It can appear inside the <head> or <body> section of the HTML document.

1. Embedding JavaScript directly:

```
<!DOCTYPE html>
<html>
<head>
  <title>Inline JavaScript Example</title>
</head>
<body>
  <h2>Hello World!</h2>
  <script>
    alert("Welcome to JavaScript!");
  </script>
</body>
</html>
```

2. Linking an external JavaScript file:

You can place your JavaScript code in a separate file (e.g., script.js) and link it using the src attribute.

```
<!DOCTYPE html>
<html>
<head>
  <title>External JS Example</title>
  <script src="script.js"></script>
</head>
<body>
  <h2>External JavaScript Example</h2>
</body>
</html>
```

2. Variables and Data Types

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

Answer:
A **variable** in JavaScript is a **container used to store data values** such as numbers, strings, or objects.
Variables allow you to **reuse and manipulate data** throughout your program.

Declaration Methods:

1. **var** – (Old method)
   - Declares a variable globally or function-scoped.
   - Can be **re-declared** and **updated**.
   - Hoisted (moved to top of scope), which may cause unexpected behavior.

```
var name = "John";
var name = "Doe";   // Re-declaration allowed
console.log(name);  // Output: Doe
```

2. **let** – (Modern and recommended)
   - Declares a **block-scoped** variable.
   - Can be **updated** but **not re-declared** in the same scope.

```
let age = 25;
age = 30;          // Allowed
console.log(age);   // Output: 30
```

3. **const** – (Constant)

   - Declares a **block-scoped constant** whose value **cannot be changed** once assigned.

   - Must be **initialized** at declaration time.

```
const pi = 3.14159;
// pi = 3.14;  ❌ Error: Assignment to constant variable
console.log(pi);
```

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

Answer:
JavaScript supports **two main categories** of data types:
➡️ **Primitive types** (basic, immutable)
➡️ **Non-primitive types** (objects)

1. Primitive Data Types:

Type	Description	Example
Number	Represents numeric values	`let x = 10;`
String	Represents text values (inside quotes)	`let name = "Alice";`
Boolean	Represents `true` or `false`	`let isValid = true;`
Undefined	A variable declared but not assigned a value	`let a; // undefined`

Null	Represents an intentional “no value”	let data = null;
BigInt	Represents very large integers	let big = 12345678901234567890n;
Symbol	Represents unique and immutable identifiers	let id = Symbol("id");

2. Non-Primitive (Reference) Data Type:

Type	Description	Example
Object	Collection of key-value pairs	let person = {name: "John", age: 30};
Array	Ordered collection of values	let fruits = ["apple", "banana", "cherry"];
Function	Block of code designed to perform a task	function greet() { console.log("Hello!"); }

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

Feature	undefined	null
Meaning	Automatically assigned by JavaScript when a variable is declared but not initialized	Manually assigned by the programmer to represent “no value”
Type	Type: undefined	Type: object
Usage	Represents a variable that **has not been given a value**	Represents **an intentional empty or nonexistent value**

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

In short:

- `undefined` → JavaScript **doesn’t know what the value is** yet.
- `null` → Developer **intentionally sets it to have no value.**

3. JavaScript Operators

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

Operators in JavaScript are **symbols** used to perform **operations on variables and values.**
There are several types of operators, but the most common ones are:

1. Arithmetic Operators

Used to perform **mathematical operations**.

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); // 13
console.log(x % y); // 1
```

2. Assignment Operators

Used to **assign values** to variables.

Operator	Description	Example	Equivalent To
=	Assigns value	x = 5	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
%=	Modulus and assign	x %= 3	x = x % 3

Example:

```
let a = 10;
a += 5; // a = a + 5 → 15
console.log(a);
```

3. Comparison Operators

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

Operator	Description	Example	Output
==	Equal to (value only)	5 == "5"	true
===	Equal to (value + type)	5 === "5"	false
!=	Not equal to (value only)	5 != "5"	false
!==	Not equal to (value + type)	5 !== "5"	true
>	Greater than	10 > 5	true
<	Less than	3 < 5	true
>=	Greater than or equal to	5 >= 5	true
<=	Less than or equal to	4 <= 5	true

Example:

```
let a = 10, b = "10";
console.log(a == b);  // true
console.log(a === b); // false
```

4. Logical Operators

Used to **combine or invert conditions.**

Operator	Description	Example	Output
&&	Logical AND (true if both are true)	(5 > 2 && 10 > 5)	true

`		`	Logical OR (true if one is true)
!	Logical NOT (inverts the value)	!(5 > 2)	false

Example:

```
let a = 5, b = 10;
console.log(a > 0 && b > 0);  // true
console.log(a > 10 || b > 5); // true
console.log(!(a > b));        // true
```

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

Operator	Name	Description	Example	Output
==	Equality Operator	Compares **only values**, not data types (performs type conversion).	5 == "5"	true
===	Strict Equality Operator	Compares **values and data types**, no type conversion.	5 === "5"	fals e

4. Control Flow (If-Else, Switch)

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

Answer:

What is Control Flow?

Control flow in JavaScript refers to the **order in which statements are executed** in a program.
Normally, JavaScript code runs **from top to bottom**, but control flow statements allow you to **make decisions**, **repeat actions**, or **jump** to different parts of the code based on certain conditions.

Examples of control flow statements include:

- if, else if, else
- switch
- for, while, do-while loops
- break, continue, return

How if-else Statements Work

The **if-else** statement is used to **execute a block of code only if a condition is true**.
If the condition is false, the else block (or else if block) can provide an alternative action.

Syntax:

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

Example:

```
let marks = 85;

if (marks >= 90) {
  console.log("Grade: A+");
} else if (marks >= 75) {
  console.log("Grade: A");
} else if (marks >= 50) {
  console.log("Grade: B");
} else {
  console.log("Grade: Fail");
}
```

Output:

```
 Grade: A
```

✅ **Explanation:**

- JavaScript checks each condition **from top to bottom**.
- As soon as one condition is true, it executes that block and **skips the rest**.
- If no condition is true, the `else` block executes.

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

Answer:

How switch Statements Work

A **switch** statement is used to **test a variable against multiple possible values.**
It is often cleaner and easier to read than using multiple if-else if conditions.

Syntax:

```
switch(expression) {
  case value1:
    // Code to execute if expression === value1
    break;
  case value2:
    // Code to execute if expression === value2
    break;
  default:
    // Code to execute 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;
  case 4:
    dayName = "Thursday";
    break;
  case 5:
    dayName = "Friday";
    break;
  case 6:
    dayName = "Saturday";
    break;
  case 7:
    dayName = "Sunday";
    break;
  default:
    dayName = "Invalid day";
}

console.log(dayName);
```

Output:

```
 Wednesday
```

✅ **Explanation:**

- The switch expression (day) is compared with each case value using **strict equality (===)**.
- When a match is found, that block runs.
- The break statement stops further execution — without it, JavaScript would continue to execute the following cases (**“fall-through” behavior**).
- The default case runs when no match is found.

When to Use switch Instead of if-else:

Situation	Recommended
When checking **one variable against many possible values**	✅ Use switch
When checking **different conditions or ranges**	✅ Use if-else
When readability is important (clear case handling)	✅ Use switch
When conditions are **complex or involve comparisons**	✅ Use if-else

Example:

- Use switch for menu selections, days of the week, user roles, etc.
- Use if-else for numeric comparisons, range checks, or logical expressions.

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

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

Answer:

A **loop** in JavaScript is used to **execute a block of code repeatedly** as long as a certain condition is true.
Loops help reduce repetitive code and make programs more efficient.

There are mainly **three types of loops** in JavaScript:
for, while, and do-while.

1. For Loop

Definition:
The **for loop** is used when the **number of iterations is known** beforehand.
It has three parts: **initialization**, **condition**, and **increment/decrement**.

Syntax:

```
for (initialization; condition; increment/decrement) {
  // code to be executed
}
```

Example:

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

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

✅ **Explanation:**

- The loop starts with `i = 1`.
- It runs while `i <= 5`.
- After each iteration, `i` increases by 1.

2. While Loop

Definition:
The **while loop** is used when the **number of iterations is not known** and depends on a condition.
It checks the condition **before** executing the code block.

Syntax:

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

Example:

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

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

✅ **Explanation:**

- The loop runs as long as the condition `i <= 5` is true.
- Once the condition becomes false, the loop stops.

3. Do-While Loop

Definition:
The **do-while loop** is similar to the while loop, but it **executes the code block at least once**, even if the condition is false.
The condition is checked **after** executing the loop body.

Syntax:

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

Example:

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

Output:

```
Value: 1
```

```
Value: 2
Value: 3
Value: 4
Value: 5
```

✅ **Explanation:**

- The code inside `do` runs once **before checking** the condition.
- Then it continues looping as long as `i <= 5`.

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

Feature	while loop	do-while loop
Condition Checking	Checked **before** the loop body executes.	Checked **after** the loop body executes.
Minimum Execution	May **not execute at all** if the condition is false initially.	**Executes at least once**, even if the condition is false.
Syntax	`while (condition) { ... }`	`do { ... } while (condition);`
Example	`javascript let x = 5; while (x < 5) { console.log(x); }` → No output	`javascript let x = 5; do { console.log(x); } while (x < 5);` → Output: `5`

6. Functions

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

Answer:

What is a Function?

A **function** in JavaScript is a **block of code designed to perform a specific task**.
Functions help make code **reusable**, **organized**, and **easier to maintain**.

A function can take input values (called **parameters**) and optionally return an output (called a **return value**).

Syntax for Declaring a Function:

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

Syntax for Calling a Function:

```
functionName(arguments);
```

Example:

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

greet("Dhruvil");  // Calling the function
```

Output:

Hello, Dhruvil!

✅ **Explanation:**

- function greet(name) declares a function with one parameter name.
- greet("Dhruvil") calls the function and passes "Dhruvil" as an argument.

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

Answer:

Feature	Function Declaration	Function Expression
Definition	Defines a named function using the function keyword.	Defines a function as part of an expression, often assigned to a variable.
Syntax	javascript function greet() { console.log("Hello!"); }	javascript let greet = function() { console.log("Hello!"); };
Hoisting	✅ **Hoisted** — can be called before it's defined.	❌ **Not hoisted** — cannot be called before definition.
Name	Must have a name.	Can be **anonymous** or **named**.
Example	javascript sayHi(); // Works function sayHi() { console.log("Hi!"); }	javascript greet(); // Error let greet = function() { console.log("Hi!"); };

✅ **In short:**

- **Function Declaration** → Hoisted, defined independently.
- **Function Expression** → Not hoisted, stored in a variable.

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

Answer:

1. Parameters

- **Parameters** are variables listed inside the parentheses `()` in a function definition.
- They act as **placeholders** for values that are passed into the function.

Example:

```
function add(a, b) {
  console.log(a + b);
}

add(5, 10); // Output: 15
```

✅ Here, `a` and `b` are **parameters**, and `5` and `10` are **arguments**.

2. Return Values

- A function can **return a value** to the caller using the `return` keyword.

- Once a return statement executes, the function **stops running** and sends the result back.

Example:

```
function multiply(x, y) {
  return x * y; // returns the product
}

let result = multiply(4, 3);
console.log(result);
```

Output:

```
12
```

✅ **Explanation:**

- `return x * y;` sends the result (12) back to the variable `result`.

7. Arrays

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

Answer:

What is an Array?

An **array** in JavaScript is a **special type of object** used to **store multiple values in a single variable**.
It can hold **different data types** such as numbers, strings, booleans, or even other arrays and objects.

Each value in an array is called an **element**, and every element has an **index** (position), starting from **0**.

Declaring and Initializing an Array

There are two main ways to declare and initialize arrays in JavaScript:

1. Using Array Literal (Recommended)

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

2. Using the Array Constructor

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

✅ **Example:**

```
let colors = ["Red", "Green", "Blue"];
console.log(colors[0]); // Output: Red
console.log(colors[2]); // Output: Blue
```

✅ **Explanation:**

- colors[0] accesses the **first element** ("Red").

- `colors[2]` accesses the **third element** ("Blue").

Arrays can also hold mixed data types:

```
let data = ["John", 25, true, { city: "Delhi" }];
console.log(data);
```

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

Answer:

JavaScript provides several built-in **array methods** to add or remove elements easily.
Here are four commonly used ones 👇

1. push()

- **Purpose:** Adds one or more elements **to the end** of an array.
- **Returns:** The **new length** of the array.

Example:

```
let fruits = ["Apple", "Banana"];
fruits.push("Mango");
console.log(fruits); // Output: ["Apple", "Banana", "Mango"]
```

2. pop()

- **Purpose:** Removes the **last element** from an array.
- **Returns:** The **removed element**.

Example:

```
let fruits = ["Apple", "Banana", "Mango"];
let lastFruit = fruits.pop();
console.log(fruits);    // Output: ["Apple", "Banana"]
console.log(lastFruit); // Output: Mango
```

3. shift()

- **Purpose:** Removes the **first element** from an array.
- **Returns:** The **removed element**.
- **Note:** The remaining elements are **re-indexed**.

Example:

```
let fruits = ["Apple", "Banana", "Mango"];
let firstFruit = fruits.shift();
console.log(fruits);    // Output: ["Banana", "Mango"]
console.log(firstFruit); // Output: Apple
```

4. unshift()

- **Purpose:** Adds one or more elements **to the beginning** of an array.
- **Returns:** The **new length** of the array.

Example:

```
let fruits = ["Banana", "Mango"];
fruits.unshift("Apple");
console.log(fruits); // Output: ["Apple", "Banana", "Mango"]
```

8. Objects

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

Answer:

What is an Object?

An **object** in JavaScript is a **collection of key-value pairs**.
Each key (also called a **property name**) is a **string**, and each value can be **any data type** — a string, number, boolean, array, function, or even another object.

Objects are used to **store and organize data** in a structured way.

Example:

```
let student = {
  name: "John",
  age: 21,
  course: "Computer Science",
  isPassed: true
};
```

✅ Explanation:

- name, age, course, and isPassed are **keys** (or **properties**).
- "John", 21, "Computer Science", and true are their **values**.

Difference Between Objects and Arrays

Feature	Objects	Arrays
Structure	Stores data as **key-value pairs**	Stores data as **ordered list of elements**
Access Method	Accessed using **property names (keys)**	Accessed using **index numbers**

When to Use	When you need to describe an **entity** with named properties	When you need to store **a list of similar items**
Example	{name: "John", age: 21}	["John", 21]

✅ **Example Comparison:**

```
// Object
let car = { brand: "Toyota", model: "Camry", year: 2022 };

// Array
let carArray = ["Toyota", "Camry", 2022];

console.log(car.brand);      // Access by key -> Toyota
console.log(carArray[0]);   // Access by index -> Toyota
```

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

Answer:

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

1. Dot Notation (.)

- Easiest and most common method.
- Use the **dot (.)** followed by the property name.

Example:

```
let person = {
  name: "Alice",
  age: 25,
  city: "Mumbai"
};

// Access properties
console.log(person.name); // Output: Alice
console.log(person.age);  // Output: 25

// Update property
person.city = "Delhi";
console.log(person.city); // Output: Delhi
```

✅ **Note:** Dot notation **cannot** be used if the property name has spaces or special characters.

2. Bracket Notation ([ ])

- Property name is written as a **string inside square brackets**.
- Useful when the key name is stored in a variable or contains spaces.

Example:

```
let person = {
  name: "Alice",
  age: 25,
  "home city": "Mumbai"
};

// Access properties
console.log(person["name"]);      // Output: Alice
console.log(person["home city"]); // Output: Mumbai

// Update property
person["age"] = 26;
console.log(person["age"]); // Output: 26

// Access using variable as key
let key = "name";
```

```
console.log(person[key]); // Output: Alice
```

9. JavaScript Events

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

Answer:

What are JavaScript Events?

In JavaScript, an **event** is an action or occurrence that happens in the browser or on a web page, which the program can **respond to**.
Events are used to make web pages **interactive and dynamic**.

Common Examples of Events:

Event	Description
onclick	Occurs when a user clicks an element
onchange	Occurs when an input value changes
onmouseover	Occurs when the mouse pointer hovers over an element
onkeydown	Occurs when a key is pressed
onload	Occurs when the page or image has finished loading

Example:

```
<button onclick="showMessage()">Click Me</button>

<script>
```

```
function showMessage() {
  alert("Button was clicked!");
}
</script>
```

✅ **Explanation:**

- When the user **clicks** the button, the **onclick** event occurs.
- The function showMessage() is **triggered** by the event.

What is an Event Listener?

An **event listener** is a JavaScript function that **waits for a specific event** to occur on an element and then **executes code** in response.

Event listeners:

- Provide a **cleaner** and more **flexible** way to handle events.
- Allow **multiple events** or **multiple functions** on the same element.

Example of Event Listener:

```
<button id="btn">Click Here</button>

<script>
document.getElementById("btn").addEventListener("click",
function() {
  alert("Event listener triggered!");
});
</script>
```

✅ **Explanation:**

- The **addEventListener()** method attaches an event listener to the button.
- When the button is clicked, the **anonymous function** runs.

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

Answer:

The **addEventListener()** method is used to attach an event handler to an HTML element **without overwriting existing events**.

Syntax:

element.addEventListener(event, function, useCapture);

Parameters:

Parameter	Description
event	The name of the event (e.g., "click", "mouseover", "keydown")
function	The function to execute when the event occurs
useCapture *(optional)*	A boolean value (true or false) that defines event flow (default is false)

Example 1: Using a Named Function

```
<button id="greetBtn">Say Hello</button>

<script>
function greet() {
  alert("Hello, welcome!");
}

document.getElementById("greetBtn").addEventListener("click
", greet);
</script>
```

✅ **Explanation:**

- The event "click" is attached to the button with id="greetBtn".
- When clicked, it triggers the greet() function.

Example 2: Using an Anonymous Function

```
<button id="colorBtn">Change Background</button>

<script>
document.getElementById("colorBtn").addEventListener("click
", function() {
  document.body.style.backgroundColor = "lightblue";
});
</script>
```

✅ **Explanation:**

- When the button is clicked, the background color of the page changes.

- No need to define a separate named function.

Advantages of addEventListener():

1. You can attach **multiple listeners** to the same event.
2. It separates **HTML and JavaScript**, improving code clarity.
3. Works for **both HTML and dynamically created elements**.

✅ Example of Multiple Listeners on Same Element:

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

<script>
let btn = document.getElementById("multiBtn");

btn.addEventListener("click", function() {
  alert("First listener!");
});

btn.addEventListener("click", function() {
  console.log("Second listener executed!");
});
</script>
```

✅ **Output:** Both messages execute on a single click — proving multiple listeners can coexist.

10. DOM Manipulation

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

Answer:
The **DOM (Document Object Model)** is a programming interface for web documents. It represents the structure of an HTML or XML document as a **tree of objects**, where each element, attribute, and piece of text becomes a **node**.

In simple terms, the DOM allows JavaScript to **access, modify, add, or delete** elements and content on a web page dynamically — without reloading the entire page.

How JavaScript interacts with the DOM:

1. **Accessing Elements:** JavaScript can access HTML elements using DOM methods like getElementById() or querySelector().

2. **Modifying Content:** It can change text, attributes, or styles of elements using properties like innerHTML, textContent, or style.

3. **Creating and Deleting Elements:** JavaScript can create new elements with createElement() and add them using appendChild() or remove them using removeChild().

4. **Handling Events:** JavaScript can respond to user actions (like clicks, typing, or hovering) using event listeners.

Example:

```
<p id="demo">Hello World!</p>

<script>
  document.getElementById("demo").innerHTML = "Hello
JavaScript!";
</script>
```

In this example, JavaScript accesses the paragraph element and changes its text content dynamically.

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

Answer:

1. **getElementById(id)**
    - Used to select a single HTML element using its **unique ID**.
    - Returns one element or null if no element is found.
      Example:

```
const heading = document.getElementById("title");
heading.style.color = "blue";
```

2. **getElementsByClassName(className)**
    - Selects all elements that share a specific **class name**.
    - Returns an **HTMLCollection** (similar to an array but not exactly).
      Example:

```
const items = document.getElementsByClassName("list-item");
items[0].style.fontWeight = "bold";
```

3. **querySelector(selector)**

- Returns the **first element** that matches a given CSS selector (e.g., id, class, or tag).
- More flexible since it uses CSS-style selection.
  Example:

```
const paragraph = document.querySelector(".intro");
```

4. `paragraph.style.fontSize = "18px";`

11. JavaScript Timing Events

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

Answer:
JavaScript provides two main timing functions — **setTimeout()** and **setInterval()** — that allow code to be executed after a certain amount of time or repeatedly at regular intervals. These functions are part of the **Window object** and are commonly used for **timing events**, animations, and delayed actions.

1. **setTimeout(function, delay)**

   ○ Executes a specified function **once** after a given delay (in milliseconds).

   ○ Commonly used to create **delays** or run code **after a certain time**.
   Syntax:

```
setTimeout(functionName, milliseconds);
```

Example:

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

2. **setInterval(function, interval)**

   ○ Executes a specified function **repeatedly** at given time intervals (in milliseconds).

   ○ Commonly used for **animations**, **updating clocks**, or **repeating tasks**.

Syntax:

```
setInterval(functionName, milliseconds);
```

Example:

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

3.

Usage in Timing Events:

- These functions allow JavaScript to handle **asynchronous operations**, meaning code can run **after a delay** or **at intervals** without blocking other tasks.
- Useful for tasks like:
    - Displaying notifications after some time
    - Creating slideshows or animations
    - Refreshing data periodically

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

Answer:

Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>setTimeout Example</title>
</head>
<body>
```

```
  <h2 id="message">Wait for it...</h2>

  <script>
    setTimeout(function() {
      document.getElementById("message").innerHTML = "Hello
after 2 seconds!";
    }, 2000);
  </script>
</body>
</html>
```

Explanation:

- The setTimeout() function delays the execution of the given code for **2000 milliseconds (2 seconds)**.

- After 2 seconds, the text inside the <h2> element changes from **"Wait for it..."** to **"Hello after 2 seconds!"**

12. JavaScript Error Handling

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

Answer:
Error handling in JavaScript is the process of detecting and managing runtime errors that occur while executing code.
Instead of stopping the entire program when an error occurs, JavaScript provides a structured way to **catch and handle** these errors gracefully using the **try...catch...finally** statement.

Syntax:

```
try {
  // Code that may cause an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code that will always run (optional)
}
```

Explanation of blocks:

1. **try block:**
    - Contains code that might throw an error.
2. **catch block:**
    - Executes if an error occurs in the try block.
    - It can access the error object that provides details about the error.
3. **finally block:**

    - (Optional) Executes **always**, whether an error occurs or not.
    - Useful for cleanup actions (like closing files or resetting variables).

Example:

```
try {
  let num = 10;
  console.log(num / 0); // No error
  console.log(x); // ReferenceError: x is not defined
}
catch (error) {
  console.log("An error occurred: " + error.message);
}
finally {
  console.log("Execution completed.");
}
```

Output:

```
Infinity
An error occurred: x is not defined
Execution completed.
```

Explanation:

- The code in the `try` block runs until an error occurs.
- When the `ReferenceError` occurs, control moves to the `catch` block.
- The `finally` block runs at the end, regardless of the error.

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

Answer:
Error handling is **important** in JavaScript applications because it ensures that the program can deal with unexpected situations without crashing or producing incorrect results.

Key reasons:

1. **Prevents program crashes:**
   Errors are caught and handled, preventing the entire program from stopping abruptly.

2. **Improves user experience:**
   Users can see meaningful error messages instead of blank screens or crashes.

3. **Helps in debugging:**
   Developers can identify and fix errors quickly using error messages.

4. **Ensures application stability:**
   Proper error handling keeps the app running smoothly even when issues occur.

5. **Handles unpredictable inputs or responses:**
   Especially important in web apps that interact with users, servers, or APIs.

Example:
If a web app fails to load data from a server, error handling can show a friendly message like *"Unable to load data. Please try again later."* instead of breaking the entire page.