JavaScript Introduction

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

JavaScript is a high-level, interpreted programming language used to create dynamic and interactive content on web pages. It is one of the core technologies of the web, alongside HTML and CSS. JavaScript enables functionalities such as real-time updates, interactive forms, animations, and event handling, making web pages more engaging for users.

Role of JavaScript in Web Development:

1. **Client-Side Interactivity:**

   Enhances user experience by adding dynamic behavior like form validation, pop-ups, and animations.

2. **DOM Manipulation:**

   Allows developers to change the structure and content of a web page dynamically.

3. **Event Handling:**

   Listens and responds to user interactions like clicks, keypresses, and mouse movements.

4. **Asynchronous Communication:**

   Supports AJAX and Fetch API for loading data without reloading the page.

5. **Frameworks & Libraries:**

   JavaScript powers popular frameworks like React, Angular, and Vue.js for building complex web applications.

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

Feature	JavaScript	Python	Java
Type	Interpreted, Dynamic Typing	Interpreted, Dynamic Typing	Compiled, Static Typing

Feature	JavaScript	Python	Java
Execution	Runs in browsers (client-side) and servers (Node.js)	Runs on servers and desktops	Runs on JVM (Java Virtual Machine)
Syntax	Lightweight, event-driven	Readable and simple	Strict, verbose
Usage	Web development, interactive UI	Data science, AI, backend	Enterprise applications, Android development
Concurrency	Single-threaded (Event Loop)	Multi-threading supported	Multi-threading supported

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 is used to embed or link JavaScript code into a web page. It can be placed in the <head> or before the closing </body> tag.

Embedding JavaScript:

```
<!DOCTYPE html>
<html>
<head>
  <title>JavaScript Example</title>
  <script>
    alert("Hello, World!");
  </script>
</head>
<body>
  <h1>Welcome to JavaScript</h1>
</body>
```

```
</html>
```

Linking an External JavaScript File:

An external JavaScript file is linked using the src attribute inside the <script> tag. This keeps the HTML clean and promotes reusability.

```
<!DOCTYPE html>
<html>
<head>
  <title>External JavaScript</title>
  <script src="script.js"></script>  <!-- Linking an external file -->
</head>
<body>
  <h1>JavaScript Linked Externally</h1>
</body>
</html>
```

Best Practices for Using the <script> Tag:

1. **Place scripts at the bottom of the <body> tag** to improve page load speed.
2. Use the **defer** attribute to load the script after HTML parsing.
3. Use **external files** for better maintainability.

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 that can be referenced and manipulated throughout a program. They act as containers for values.

Declaring Variables in JavaScript:

1. **var** (Old way, function-scoped)
2. **let** (Modern way, block-scoped)
3. **const** (Constant, cannot be reassigned)

```
var name = "Alice";  // Can be re-declared and updated (not recommended)
let age = 25;       // Can be updated but not re-declared
```

Key Differences:

Feature	var	let	const
Scope	Function-scoped	Block-scoped	Block-scoped
Re-declaration	Allowed	Not allowed	Not allowed
Reassignment	Allowed	Allowed	Not allowed
Hoisting	Hoisted with undefined	Hoisted but not initialized	Hoisted but not initialized

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

JavaScript has **two main categories** of data types: **Primitive** and **Non-Primitive (Reference)**.

Primitive Data Types:

1. **String** → Text enclosed in quotes.

```
let name = "John";
```

2. **Number** → Any number (integer or float).

```
let age = 30;
```

3. **Boolean** → True or false values.

```
let isLoggedIn = true;
```

4. **Undefined** → A variable declared but not assigned a value.

```
let city;
console.log(city); // undefined
```

5. **Null** → An intentional empty value.

let car = null;

6. **Symbol** → Unique values, mainly for object properties.

let id = Symbol("uniqueID");

7. **BigInt** → Large numbers beyond Number.MAX_SAFE_INTEGER.

let bigNumber = 1234567890123456789012345678901234567890n;

Non-Primitive (Reference) Data Types:

1. **Object**

   → Key-value pairs.

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

2. **Array**

   → List of values.

let colors = ["red", "green", "blue"];

3. **Function**

   → A block of reusable code.

function greet() {

  return "Hello!";

}

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

Feature	undefined	null
Meaning	A variable is declared but **not assigned a value**.	A variable is assigned an **empty or intentional absence of value**.
Type	undefined (Primitive)	object (Primitive, but mistakenly classified as object)
Example	let x; console.log(x); // undefined	let y = null; console.log(y); // null

Easy Way to Remember:

- **undefined** → JavaScript **did not** assign a value.
- **null** → **You** assigned an empty value.

JavaScript Operators

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

1. **Arithmetic Operators (Used for mathematical calculations)**
    - **+ (Addition):**
    - **5 + 3 = 8**
    - **- (Subtraction):**
    - **10 - 4 = 6**
    - *** (Multiplication):**
    - **6 * 2 = 12**
    - **/ (Division):**
    - **10 / 2 = 5**
    - **% (Modulus - Remainder):**
    - **10 % 3 = 1**
    - **++ (Increment): let a = 5; a++; // a becomes 6**
    - **-- (Decrement): let b = 8; b--; // b becomes 7**

```
let a = 10;
let b = 5;
console.log(a + b);  // Output: 15
console.log(a % b);  // Output: 0
```

2. **Assignment Operators (Used to assign values)**
    - **= (Assigns value):**

      x = 5
    - **+= (Adds and assigns):**

x += 3 (same as x = x + 3)

- -= (Subtracts and assigns):

  x -= 2

- *= (Multiplies and assigns):

  x *= 4

- /= (Divides and assigns):

  x /= 2

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

3. Comparison Operators (Used to compare values)
   - == (Equal to - checks only value):

     5 == "5" // true

   - === (Strict equal - checks value & type):

     5 === "5" // false

   - != (Not equal - checks only value):

     10 != 5 // true

   - !== (Strict not equal - checks value & type):

     10 !== "10" // true

   - > (Greater than):

     8 > 5 // true

   - < (Less than):

     3 < 7 // true

   - >= (Greater than or equal):

     5 >= 5 // true

   - <= (Less than or equal):

     4 <= 6 // true

```
console.log(10 > 5);  // true
```

console.log(10 === "10"); // false

4. **Logical Operators (Used to combine conditions)**
    - **&& (AND - both conditions must be true): (5 > 3 && 8 > 6) // true**
    - **|| (OR - at least one condition must be true): (10 < 5 || 7 > 2) // true**
    - **! (NOT - reverses the condition): !(5 === 5) // false**

```
let age = 20;
console.log(age > 18 && age < 30);  // true
console.log(!(age > 18));  // false
```

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

1. **== (Loose Comparison)**
    - **Checks only the value, not the type.**
    - **Converts values to the same type before comparing.**
    - **Example:**

```
console.log(5 == "5");   // true (string "5" is converted to number 5)
console.log(0 == false); // true (false is converted to 0)
```

2. **=== (Strict Comparison)**
    - **Checks both value and type without conversion.**
    - **Example:**

```
console.log(5 === "5");   // false (number vs string)
console.log(0 === false); // false (number vs boolean)
```

Easy way to remember:

- **== → Compares only the value (loose comparison, type conversion happens).**
- **=== → Compares both value and type (strict comparison, no conversion).**

Control Flow (If-Else, Switch)

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

1. **Control flow** in JavaScript refers to the order in which statements are executed in a program.
2. By default, JavaScript runs code from top to bottom, but control structures like **if-else**, **loops**, and **switch statements** allow us to change this flow.

How if-else statements work:

- if checks a condition. If it's true, the code inside the block runs.
- else if allows checking multiple conditions.
- else runs if none of the conditions are true.

Example:

```
let age = 18;


if (age < 18) {
   console.log("You are a minor.");
} else if (age === 18) {
   console.log("You just became an adult!");
} else {
   console.log("You are an adult.");
}
```

Output: (if age is 18)

✅ **Use if-else when checking conditions that involve ranges (<, >, >=, etc.).**

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

1. A **switch statement** is used to perform **different actions** based on different values.
2. It checks a value against multiple **cases** and executes the matching block.
3. A break is used to stop execution after a case matches (otherwise, all cases after the match will run).
4. The default case runs if no other case matches.

Example of a Switch Statement:

```
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("Start of the workweek.");
    break;
  case "Friday":
    console.log("Weekend is near!");
    break;
  case "Sunday":
    console.log("It's a relaxing day.");
    break;
  default:
    console.log("It's a regular day.");
}
```

Output: "Start of the workweek." (if day is "Monday").

- ⇨ **Use switch when comparing a single value against multiple specific options (case values).**
  Use if-else for conditions involving ranges, complex logic, or boolean expressions.

Loops (For, While, Do-While)

Question 1: Different Types of Loops in JavaScript

Loops in JavaScript help in executing a block of code multiple times. The main types of loops are:

1. **for Loop** → Used when the number of iterations is known.

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

2. **while Loop** → Runs as long as the condition is true.

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

3. **do-while Loop** → Similar to while, but executes the block **at least once** before checking the condition.

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

Question 2: Difference Between while and do-while Loop

1. **while Loop**
   - **Condition is checked first** before executing the loop.
   - If the condition is false at the beginning, the loop **does not run** at all.

```
let x = 10;
while (x < 5) {
  console.log(x);  // This will NOT run because x is already 10
}
```

2. **do-while Loop**
   - **Executes at least once**, even if the condition is fase.
   - Condition is checked **after** running the loop once.

```
let x = 10;
```

```
do {
   console.log(x);  // Runs once, even though x is already 10
} while (x < 5);
```

Easy way to remember:

- **while** → **Checks first**, runs only if condition is true.
- **do-while** → **Runs first**, checks condition afterwar.

Arrays

Question 1: What is an Array in JavaScript?

- **An array is a special variable that stores multiple values in a single variable**.
- Arrays help organize data and make it easy to access and modify.

How to Declare and Initialize an Array

1. **Using square brackets ([]) → Most common method:**

```
let fruits = ["Apple", "Banana", "Mango"];
let numbers = [10, 20, 30, 40];
```

2. **Using new Array() (less common)**

```
let cars = new Array("Toyota", "Honda", "Ford");
```

Accessing Elements in an Array

- Each item has an index (starting from 0).

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

Question 2: Methods push(), pop(), shift(), and unshift()

1. **push()**

   → Adds an item at the end of the array.

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

2. **pop()** →

   Removes the last item from the array.

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

3. **shift()**

   → Removes the first item from the array.

```
let numbers = [10, 20, 30];
numbers.shift();  // Removes 10
console.log(numbers);  // Output: [20, 30]
```

4. **unshift()**

   → Adds an item at the beginning of the array.

```
numbers.unshift(5);  // Adds 5 at the beginning
console.log(numbers);  // Output: [5, 20, 30]
```

- push() → Add to end
- pop() → Remove from end
- shift() → Remove from start
- unshift() → Add to start

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**.
- It is used to store multiple related data items **together**, like a person's name, age, and city.

How to Declare an Object

```
let person = {
  name: "John",
```

```
  age: 25,
  city: "New York"
};
```

Objects vs. Arrays

Feature	Object	Array
Structure	Key-value pairs	Indexed list of values
Access	object.key	array[index]
Use case	When data has named properties	When data is a list

Example:

```
let car = { brand: "Toyota", model: "Camry", year: 2022 };  // Object
let numbers = [10, 20, 30, 40];  // Array
```

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

1. Dot Notation (.)

- Easiest and most commonly used.
- Works when property names **don't have spaces or special characters**.
- **Example:**

```
let person = { name: "John", age: 25 };
console.log(person.name);  // Output: John

// Updating a property
person.age = 30;
console.log(person.age);  // Output: 30
```

2. Bracket Notation ([])

- Used when property names **have spaces or special characters**.
- Also useful when accessing properties dynamically.
- **Example:**

```
let person = { "full name": "John Doe", age: 25 };
console.log(person["full name"]);  // Output: John Doe

// Updating a property
person["age"] = 30;
console.log(person["age"]);  // Output: 30
```

JavaScript Events

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

1. **JavaScript events** are actions that happen in a web page, such as clicking a button, typing in a form, or moving the mouse.
2. **Events allow webpages to be interactive** by responding to user actions.
3. Examples of events:
    - click (When a user clicks a button)
    - mouseover (When a user hovers over an element)
    - keydown (When a user presses a key)

What is an Event Listener?

- An **event listener** waits for an event to happen and then runs a function when the event occurs.
- It helps in handling user interactions.

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

1. **addEventListener() is used to attach an event to an element.**
2. **It takes two main arguments:**
    - The **event type** (e.g., "click", "mouseover")
    - The **function** to run when the event happens

Example: Button Click Event

```
// Select the button
```

```
let button = document.getElementById("myButton");


// Add a click event listener
button.addEventListener("click", function() {
  alert("Button Clicked!");
});
```

✅ **Why use addEventListener() instead of onclick?**

- It allows adding multiple event handlers to the same element.
- It makes the code cleaner and more flexible

DOM Manipulation

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

1. **DOM (Document Object Model)** is a **tree-like structure** that represents an HTML page in the browser.
2. It allows JavaScript to **read, modify, and update** webpage elements dynamically.
3. **JavaScript can interact with the DOM** to:
    - Change text, images, and styles.
    - Add or remove HTML elements.
    - Handle user events like clicks and keystrokes.

Example: Changing Text with JavaScript

```
document.getElementById("heading").innerText = "Hello, JavaScript!";
```

Question 2: Methods to Select Elements from the DOM

1. **getElementById()**

    → Selects **one** element by its **ID**.

```
let heading = document.getElementById("title");
heading.style.color = "red";  // Changes text color to red
```

2. **getElementsByClassName()**

→ Selects **multiple** elements by **class name** (returns a list).

```
let items = document.getElementsByClassName("item");
items[0].style.color = "blue";  // Changes first item's color
```

3. **querySelector()**

→ Selects **the first matching** element (can use ID, class, or tag).

```
let button = document.querySelector(".btn");
button.style.backgroundColor = "green";  // Changes button color
```

Use getElementById() for a single element.
Use getElementsByClassName() when selecting multiple elements.
Use querySelector() for flexible selection with CSS selectors.

Question 1: setTimeout() and setInterval() in JavaScript

1. **setTimeout()** → Runs a function **once** after a delay.
2. **setInterval()** → Runs a function **repeatedly** at fixed intervals.

How They Are Used for Timing Events

- **Use setTimeout()** to delay an action (e.g., showing a message after 2 seconds).
- **Use setInterval()** to repeat an action (e.g., updating a clock every second).

Example of setTimeout()

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

Example of setInterval()

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

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

```
setTimeout(function() {
```

```
  alert("This alert appears after 2 seconds!");
}, 2000);
```

JavaScript Error Handling

Question 1: What is Error Handling in JavaScript?

- **Error handling** in JavaScript helps detect and manage errors **without crashing the program**.
- It ensures the program **runs smoothly**, even if something goes wrong.

try, catch, and finally Blocks

1. **try Block**

   → Code that might cause an error goes here.

2. **catch Block**

   → If an error occurs, this block handles it.

3. **finally Block**

   → Runs **always**, whether there's an error or not.

Example: Handling Errors in JavaScript

```
try {
  let x = 5;
  console.log(y);  // 'y' is not defined → Causes an error
} catch (error) {
  console.log("An error occurred: " + error.message);
} finally {
  console.log("This will always run.");
}
```

Output:

An error occurred: y is not defined

This will always run.

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

1. **Prevents Program Crashes**

   → Instead of stopping execution, errors are handled gracefully.

2. **Improves User Experience**

   → Users see friendly error messages instead of a broken page.

3. **Debugging & Maintenance**

   → Helps developers find and fix issues faster.

4. **Handles Unexpected Issues**

   → Prevents bugs from causing security risks