JavaScript Introduction

Question 1: What is JavaScript? Explain the role of JavaScript in web development.
Answer:
JavaScript is a **high-level, dynamic, interpreted programming language** primarily used for creating interactive and dynamic content on web pages. It is a core technology of the web, alongside HTML and CSS.

Role of JavaScript in Web Development

1. Interactivity: Adds features like sliders, pop ups, and animations.
2. Dynamic Content: Updates web pages in real-time without reloading (e.g., using AJAX).
3. Form Validation: Validates user inputs on the client side.
4. Event Handling: Reacts to user actions like clicks or keypresses.
5. DOM Manipulation: Dynamically changes HTML and CSS elements.
6. API Integration: Fetches or sends data to servers.
7. Web Applications: Builds complex apps using frameworks like React or Angular.

Question 2: How is JavaScript different from other programming languages like Python orJava?
Answer:
JavaScript is primarily distinguished from languages like Python and Java by its specific focus on web browser interaction, meaning it's designed to add dynamic and interactive features directly to web pages, while Python and Java are more versatile and can be used for a wider range of applications, including back-end development and data analysis, where JavaScript is less optimized; additionally, JavaScript is primarily used for client-side scripting, directly manipulating a web page's elements through the Document Object Model (DOM), unlike the more general-purpose nature of Python and Java.

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 within a webpage.
Inline JavaScript:

```
<script>
alert('Hello, world!');
</script>
```

External JavaScript:

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

Variables and Data Types

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

Answer:
In JavaScript, **variables** are containers for storing data values. They allow developers to label data and reuse it throughout a program.
VAR:
var name = "John";
console.log(name); // Outputs: John

Reassignment:Allowed
Declaration:Can be redeclared

LET:
let age = 25;
console.log(age); // Outputs: 25

age = 30;
console.log(age); // Outputs: 30

Reassignment:Allowed
Declaration:Cannot be redeclared

const pi = 3.14;
console.log(pi); // Outputs: 3.14
// pi = 3.15; // Error: Assignment to constant variable

Reassignment:Not Allowed
Declaration:Cannot be redeclared

Question 2: Explain the different data types in JavaScript. Provide examples for each.
Answer:
JavaScript supports several data types, which can be categorized into **primitive types** and **non-primitive types (objects)**.

Primitive Data Types

1. **String**
   let name = "John";
   console.log(name); // Outputs: John
2. **Number**
   let age = 25;
   let pi = 3.14;
   console.log(age, pi); // Outputs: 25 3.14
3. **BigInt**
   let bigNumber = 1234567890123456789012345678901234567890n;
   console.log(bigNumber); // Outputs: 1234567890123456789012345678901234567890n

4. **Boolean**
```
let isStudent = true;
console.log(isStudent); // Outputs: true
```
5. **Undefined**
```
let x;
console.log(x); // Outputs: undefined
```
6. **Null**
```
let y = null;
console.log(y); // Outputs: null
```
7. **Symbol**
```
let id = Symbol("uniqueId");
console.log(id); // Outputs: Symbol(uniqueId)
```

Non-Primitive Data Types

1. **Object**
```
let person = {
  name: "John",
  age: 30
};
console.log(person.name); // Outputs: John
```
2. **Array**
```
let colors = ["red", "green", "blue"];
console.log(colors[0]); // Outputs: red
```
3. **Function**
```
function greet() {
  console.log("Hello, World!");
}
greet(); // Outputs: Hello, World!
```
4. **Date**
```
let today = new Date();
console.log(today); // Outputs: Current date and time
```

Type Checking

Use `typeof` to check the type of a variable.

```
console.log(typeof "Hello"); // Outputs: string
console.log(typeof 42);      // Outputs: number
console.log(typeof true);    // Outputs: boolean
console.log(typeof null);    // Outputs: object (quirk in JavaScript)
```

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

- Use `undefined` for uninitialized variables or when a value is unknown.
- Use `null` when you want to intentionally assign "no value."

Feature	undefined	null
Meaning	Variables Declared but no assigned value	Intentional absence of a value
Type	undefined(primitive type)	null(primitive type, but typeof null is “object”)
Default	Automatically assigned by Javascript	Must be explicitly assigned by the developer
Usage	Indicates a variable or property isn’t initialized	Used to intentionally clear a variable or represent “no value”.

JavaScript Operators

Question 1: What are the different types of operators in JavaScript? Explain with examples.
Answer:
• Arithmetic operators
These are used to perform mathematical calculations.

Operator	Description	Example	Output
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	2 * 3	6
/	Division	9 / 3	3
%	Modulus (remainder)	7 % 2	1
**	Exponentiation	3 ** 2	9

Example Code:

let a = 10, b = 3;

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

console.log(a % b); // 1

console.log(b ** 2); // 9

• Assignment operators
These are used to assign values to variables.

Operator	Description	Example	Output
=	Assignment	x = 5	x = 5
+=	Add and Assign	x += 3(x=x+3)	x = 8
-=	Subtract and Assign	x -= 2(x=x-2)	x = 6
*=	Multiply and Assign	x *= 2 (x=x*2)	x = 12
/=	Divide and assign	x /= 2 (x=x/2)	x = 6
%=	Modulus and assign	x %= 3 (x=x%3)	x = 0

Example Code:

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

• Comparison operators
These are used to compare two values and return a boolean (true or false).

Operator	Description	Example	Output
==	Equal to (loose equality)	5 == '5'	true
===	Equal to (strict equality)	5 === '5'	false
!=	Not equal to	5 != '5'	false
!==	Not equal (strict)	5 !== '5'	true
<	Less than	3 < 5	true
>	Greater than	10 > 6	true
<=	Less than or equal to	7 <= 7	true
>=	Greater than or equal to	8 >= 9	false

Example Code:
Copy code

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

console.log(num === "10"); // false
• Logical operators
These are used to perform logical operations on boolean values.

1. Logical AND (&&)

- Description: Returns true if both operands are true. Otherwise, it returns false.
- Syntax: condition1 && condition2

Truth Table for &&:

Condition 1	Condition 2	Result
true	true	true
true	false	false
False	true	false
false	false	false

Example:

```
let isSunny = true;
let isWeekend = true;
console.log(isSunny && isWeekend); // Output: true

let isRaining = false;
console.log(isSunny && isRaining); // Output: false
```

2. Logical OR (| |)

- Description: Returns true if at least one operand is true. If both operands are false, it returns false.
- Syntax: condition1 || condition2

Truth Table for | |:

Condition 1	Condition 2	Result
true	true	true
true	false	true
False	true	true
false	false	false

Example:

```
let isHoliday = false;
let isWeekend = true;
console.log(isHoliday || isWeekend); // Output: true
```

```
let isWeekday = false;
console.log(isHoliday || isWeekday); // Output: false
```

3. Logical NOT (!)
- Description: Reverses the boolean value. If the operand is true, it returns false, and vice versa.
- Syntax: !condition

Truth Table for !:

Condition	Result
true	false
false	true

Example:
```
let isSunny = true;
console.log(!isSunny); // Output: false

let isRainy = false;
console.log(!isRainy); // Output: true
```

Question 2: What is the difference between == and === in JavaScript?
Answer:
1. Double Equals (==):
Compares values only but not types.
2. Triple Equals (===):
Compares both value and type.

Operator	Compares	Example	Output
==	Values only	5 == “5”	true
===	Values and datatypes	5 === “5”	false

Example:
```
let a = "5";
let b = 5;

console.log(a == b);  // true (values are equal after coercion)
console.log(a === b); // false (different types: string vs number)
```

Control Flow (If-Else, Switch)

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

Answer:Control flow refers to the order in which the code in a program is executed. JavaScript executes code from top to bottom by default, but control flow structures like conditional statements, loops, and functions allow you to alter the execution sequence based on certain conditions or logic.
if-else Statement in JavaScript:
The if-else statement is a control flow structure that allows you to execute a block of code if a specific condition is true. If the condition is false, an alternative block of code (in the else block) is executed.
Syntax:

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute 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.");
}
```

Question 2: Describe how switch statements work in JavaScript. When should you use a switch statement instead of if-else
Answer:A switch statement allows you to compare one value against multiple possible options (cases) and execute code based on the match. It is useful when you need to test a variable or expression against many specific values.
Syntax:

```
switch (expression) {
    case value1:
        // Code if expression === value1
        break;
    case value2:
        // Code if expression === value2
        break;
    default:
        // Code if no cases match
}
```

Example:

```
let color = "red";

switch (color) {
    case "red":
        console.log("You selected Red.");
```

```
        break;
    case "blue":
        console.log("You selected Blue.");
        break;
    case "green":
        console.log("You selected Green.");
        break;
    default:
        console.log("Color not recognized.");
}
```

Output:
You selected Red.

When to Use:

- Use switch for fixed values or discrete cases (e.g., matching colors, days of the week).
- Use if-else for ranges or complex conditions (e.g., x > 10 && x < 20).

Loops (For, While, Do-While)

Question 1: Explain the different types of loops in JavaScript (for, while, do-while). Provide basic examples of each.
Answer:
1. for Loop:The for loop is used when the number of iterations is known beforehand.
Syntax:

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

Example:

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

2. while Loop:The while loop runs as long as the condition is true.
Syntax:

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

Example:

```
let i = 1;
while (i <= 5) {
    console.log(i);  // Prints numbers from 1 to 5
    i++;  // Increment the value of i
}
```

3. do-while Loop:The do-while loop is similar to the while loop, but it guarantees at least one iteration. This is because the condition is checked after the code inside the loop runs.
Syntax:

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

Example:

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

Question 2: What is the difference between a while loop and a do-while loop?
Answer:
while loop checks the condition before each iteration and may not run at all if the condition is false initially.
do-while loop guarantees at least one execution of the code, as it checks the condition after each iteration.

Functions

Question 1: What are functions in JavaScript? Explain the syntax for declaring and calling a function.
Answer:
1.Functions
A function in JavaScript is a block of reusable code that performs a specific task. Functions allow you to encapsulate code so you can reuse it without rewriting the same code multiple times
Syntax:

```
function functionName(parameters) {
    // Code to execute
}
functionName(arguments);
```

Example:

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

2. Function Expression
A function can also be assigned to a variable. This is called a function expression
Syntax:

```
const functionName = function(parameters) {
    // Code to execute
};
```

Example:

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

greet("Bob");  // Output: Hello, Bob!.
```

3. Arrow Functions

Arrow functions are a more concise way to write functions

Syntax:

```
const functionName = (parameters) => {
   // Code to execute
};
```

Example:

```
const greet = (name) => {
   console.log("Hello, " + name + "!");
};
greet("David");  // Output: Hello, David!
```

If the function has only one parameter, you can omit the parentheses.

```
const greet = name => console.log("Hello, " + name + "!");
greet("Charlie");  // Output: Hello, Charlie!
```

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

Answer:

Function Declaration	Function Expression
Defines a function with the function keyword.	Defines a function as part of an expression (e.g., assigning to a variable).
Hoisted: Can be called before the declaration in the code.	Not hoisted: Cannot be called before the function is defined.
Always has a name (the name used in the declaration).	Can be anonymous (i.e., without a name) or named.

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

Answer:

1. Parameters

Parameters are placeholders used in the function definition. They allow you to pass data into a function when you call it. A function can accept zero or more parameters, and these parameters act as variables that can hold the values passed when the function is invoked.

Syntax:

```
function functionName(parameter1, parameter2, ...) {
   // Code to execute
}
```

Example:

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

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

2. Return Values

A return value is the value that a function sends back to the place where it was called. The return statement is used to send this value out of the function. If a function doesn't explicitly return anything, it returns undefined by default.

Syntax:

```
function functionName(parameters) {
    return value;
}
```

Example:

```
function multiply(a, b) {
    return a * b;
}

let result = multiply(5, 4);
console.log(result);  // Output: 20
```

Arrays

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

Answer:

An array in JavaScript is a special variable that can hold multiple values at the same time. Instead of creating separate variables for each value, you can store them in a single array. It is like a list that you can access by index (position).

How to Declare and Initialize an Array:

Declare an Array: Use square brackets [].

Initialize an Array: Add values inside the square brackets, separated by commas.

```
// Declaring and initializing an array
let fruits = ["Apple", "Banana", "Cherry"];

// Accessing array elements
console.log(fruits[0]); // Outputs: Apple
console.log(fruits[1]); // Outputs: Banana

// Adding an element to the array
fruits.push("Orange");
```

console.log(fruits); // Outputs: ["Apple", "Banana", "Cherry", "Orange"]

Question 2: Explain the methods push(), pop(), shift(), and unshift() used in arrays
Answer:
- push()
What it does: 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("Cherry");
console.log(fruits); // ["Apple", "Banana", "Cherry"]
- pop()
What it does: Removes the last element from an array.
Returns: The element that was removed.
Example:
let fruits = ["Apple", "Banana", "Cherry"];
let removed = fruits.pop();
console.log(removed); // "Cherry"
console.log(fruits); // ["Apple", "Banana"]
- shift()
What it does: Removes the first element from an array.
Returns: The element that was removed.
Example:-
let fruits = ["Apple", "Banana", "Cherry"];
let removed = fruits.shift();
console.log(removed); // "Apple"
console.log(fruits); // ["Banana", "Cherry"]
- unshift()
What it does: Adds one or more elements to the beginning of an array.
Returns: The new length of the array.
Example:-
let fruits = ["Banana", "Cherry"];
fruits.unshift("Apple");
console.log(fruits); // ["Apple", "Banana", "Cherry"]

Object

Question 1: What is an object in JavaScript? How are objects different from arrays?
Answer:
An object in JavaScript is a collection of properties, where each property is a key-value pair.
Objects are used to represent real-world entities or structured data.
Syntax:
let objectName = {
    key1: value1,

```
    key2: value2,
    key3: value3
};
```
Example:
```
let person = {
    name: "John",
    age: 30,
    profession: "Engineer"
};

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

Difference between object and array:

Objects	Array
Unordered collection of Key-value pairs.	Ordered collection of elements(indexed)
Accessing properties using keys	Access elements using indices(numeric)
Defined with {}.	Defined with [].

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

Answer:

1. Dot Notation

Syntax:
```
objectName.propertyName
```
Example:
```
let person = {
    name: "John",
    age: 30
};

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

// Update property
person.age = 35;
console.log(person.age);  // Output: 35
```

2.Bracket Notation

Syntax:
```
objectName["propertyName"]
```

Example:

```
let person = {
    name: "John",
    age: 30,
    "favorite color": "blue"  // Property with a space
};

// Access property
console.log(person["name"]);  // Output: John
console.log(person["favorite color"]);  // Output: blue

// Update property
person["age"] = 40;
console.log(person["age"]);  // Output: 40
```

JavaScript Events

Question 1: What are JavaScript events? Explain the role of event listeners.
Answer:
JavaScript events are actions or occurrences that happen in the browser, often triggered by the user or the system. These can include clicks, mouse movements, key presses, or even the loading of a web page. Events allow JavaScript to interact with the user and make web pages dynamic.

User-triggered events: `click`, `mouseover`, `keydown`, `keyup`

System-triggered events: `load`, `resize`, `error`

Example:

```
<button onclick="alert('Button clicked!')">Click Me</button>
```

An **event listener** is a JavaScript feature that listens for a specified event on an element and executes a callback function when the event occurs. Event listeners allow you to separate JavaScript logic from HTML, improving code readability and maintainability.

Syntax:

```
element.addEventListener(event, callback);
```

Example:

HTML:

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

Javascript:

```
// Select the button element
let button = document.getElementById("myButton");

// Add an event listener for the 'click' event
button.addEventListener("click", function() {
    alert("Button was clicked!");
```

});
Question 2: How does the addEventListener() method work in JavaScript? Provide an example.
Answer:
The addEventListener() method in JavaScript allows you to attach an event handler to a DOM element. It listens for a specified event (like click, mouseover, keydown, etc.) and executes a callback function when that event occurs.
Syntax:
element.addEventListener(event, callback, useCapture);
Example:
HTML:
<button id="myButton">Click Me</button>
JavaScript:
// Select the button element
let button = document.getElementById("myButton");

// Define a function to handle the click event
function handleClick() {
    alert("Button was clicked!");
}

// Attach the event listener to the button
button.addEventListener("click", handleClick);

DOM Manipulation

Question 1: What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?
Answer:
The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of an HTML or XML document as a tree-like structure, where each element (tags, attributes, and text) is treated as an object that JavaScript can manipulate.
The DOM allows developers to:

1. Access: Retrieve elements and their properties.
2. Modify: Change content, styles, or structure dynamically.
3. Add/Remove: Insert new elements or remove existing ones.
4. React: Respond to user actions through events.

JavaScript interacts with the DOM by using the document object, which provides methods and properties to navigate and manipulate the DOM tree.
1. Accessing DOM Elements
By ID: `document.getElementById()`
By Class: `document.getElementsByClassName()`
By Tag: `document.getElementsByTagName()`
By Selector: `document.querySelector()` or `document.querySelectorAll()`

```
let header = document.getElementById("header"); // Access an element by its ID
console.log(header.textContent); // Print the content of the element
```
2. Modifying DOM Elements
```
let paragraph = document.querySelector("p");
paragraph.textContent = "New content!"; // Update the text inside <p>
paragraph.style.color = "blue"; // Change the text color to blue
```
3. Adding or Removing Elements

Adding:
```
let newElement = document.createElement("div"); // Create a new <div>
newElement.textContent = "Hello, DOM!";
document.body.appendChild(newElement); // Add the new <div> to the body
```
Removing:
```
let element = document.getElementById("removeMe");
element.remove(); // Remove the element
```
4. Handling Events

JavaScript can listen for user interactions (like clicks, key presses, etc.) and execute code in response.

Example: Event Handling:
```
document.getElementById("myButton").addEventListener("click", () => {
    alert("Button was clicked!");
});
```
5. Traversing the DOM

You can navigate through parent, child, and sibling elements in the DOM tree.

Example: Navigating the DOM:
```
let parent = document.getElementById("child").parentElement; // Get parent
let firstChild = document.getElementById("parent").firstElementChild; // Get first child
console.log(firstChild.textContent);
```

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

Answer:

1. getElementById()

Syntax:
```
document.getElementById(id);
```
Example:
```
<h1 id="header">Welcome to JavaScript</h1>
<script>
    let header = document.getElementById("header");
    console.log(header.textContent); // Output: Welcome to JavaScript
</script>
```
2. getElementsByClassName()

Syntax:
```
document.getElementsByClassName(className);
```

Example:

```
<div class="item">Item 1</div>
<div class="item">Item 2</div>
<div class="item">Item 3</div>
<script>
   let items = document.getElementsByClassName("item");
   console.log(items[0].textContent); // Output: Item 1
   for (let item of items) {
      console.log(item.textContent);
   }
</script>
```

3. querySelector()

Syntax:

```
document.querySelector(selector);
```

Example:

```
<p id="para">Paragraph with ID</p>
<p class="para">Paragraph with class</p>
<p>Generic paragraph</p>
<script>
   let firstPara = document.querySelector("#para"); // Select by ID
   console.log(firstPara.textContent); // Output: Paragraph with ID

   let classPara = document.querySelector(".para"); // Select by class
   console.log(classPara.textContent); // Output: Paragraph with class
</script>
```

JavaScript Timing Events (setTimeout, setInterval)

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

Answer:

1. setTimeout()

- Executes a function once after a specified delay (in milliseconds).
- Used for delayed execution of code.

Syntax:

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

Example:

```
console.log("Before setTimeout");
setTimeout(() => {
   console.log("Executed after 2 seconds");
```

```
}, 2000);
console.log("After setTimeout");
Output:
Before setTimeout
After setTimeout
Executed after 2 seconds
2. setInterval()
Executes a function repeatedly at a specified time interval (in milliseconds).
Used for repeated execution of code.
Syntax:
setInterval(function, delay, ...args);
Example:
let count = 0;
let intervalId = setInterval(() => {
    count++;
    console.log(`Count: ${count}`);
    if (count === 5) {
        clearInterval(intervalId); // Stops the interval after 5 iterations
    }
}, 1000);
Output:
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
1.With setTimeout() (Delayed Actions):
Example:
setTimeout(() => {
  alert("Time's up!");
}, 5000); // After 5 seconds
2.With setInterval() (Repeated Actions):
Example:
setInterval(() => {
  const now = new Date();
  console.log(now.toLocaleTimeString());
}, 1000); // Updates every second
Question 2: Provide an example of how to use setTimeout() to delay an action by 2 seconds
Answer:
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>setTimeout Example</title>
</head>
<body>
    <h1>JavaScript setTimeout Example</h1>
    <p id="message">Wait for it...</p>

    <script>
        // Select the paragraph element
        const messageElement = document.getElementById("message");

        // Use setTimeout to change the text after 2 seconds
        setTimeout(() => {
            messageElement.textContent = "Action executed after 2 seconds!";
        }, 2000);
    </script>
</body>
</html>
```

JavaScript Error Handling

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

Answer:

Error handling is the process of catching and managing runtime errors to prevent the application from crashing and to provide meaningful feedback to users or developers.

JavaScript provides the try-catch-finally construct for error handling:

1. try Block
   - Contains code that may throw an error.
   - If an error occurs, the code execution in the try block stops, and control moves to the catch block.
2. catch Block
   - Contains code to handle the error.
   - Provides access to the error object, which contains details about the error.
3. finally Block
   - Contains code that executes regardless of whether an error occurred or not.
   - Often used for cleanup tasks (e.g., closing a file or releasing resources).

Syntax:

```
try {
    // Code that may throw an error
} catch (error) {
    // Code to handle the error
} finally {
    // Code that always executes
}
```

Example:

```
function divideNumbers(a, b) {
    try {
        if (b === 0) {
            throw new Error("Division by zero is not allowed.");
        }
        const result = a / b;
        console.log(`Result: ${result}`);
    } catch (error) {
        console.error(`Error: ${error.message}`);
    } finally {
        console.log("Execution completed.");
    }
}

// Test cases
divideNumbers(10, 2); // Valid division
divideNumbers(10, 0); // Division by zero
```

Output:
For divideNumbers(10, 2):
Result: 5
Execution completed.
For divideNumbers(10, 0):
Error: Division by zero is not allowed.
Execution completed.
Question 2: Why is error handling important in JavaScript applications?
Answer:
Prevents Crashes: Ensures the application doesn't break entirely when errors occur.
Improves User Experience: Displays friendly error messages instead of confusing ones.
Simplifies Debugging: Provides detailed error messages for easier troubleshooting.
Handles Edge Cases Gracefully: Allows fallback solutions or alternative functionality.
Enhances Security: Prevents sensitive information from being exposed in errors.
Ensures Robustness: Keeps the application reliable in unexpected conditions.
Supports Logging and Monitoring: Tracks issues in real-time for faster fixes.