Module 8) JavaScript

JavaScript Introduction

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

JavaScript is a **high-level, interpreted programming language** that is mainly used in **web development**. It is one of the **core technologies of the World Wide Web**, along with **HTML** and **CSS**.

- **HTML (HyperText Markup Language):** Creates the structure of a web page (headings, paragraphs, images).
- **CSS (Cascading Style Sheets):** Styles the content (colors, fonts, layouts).
- **JavaScript:** Makes web pages **dynamic, interactive, and functional**.

Role of JavaScript in Web Development

1. **Adding Interactivity**
    - JavaScript makes web pages come alive.
    - Example: Buttons that perform actions, image sliders, dropdown menus, and pop-up alerts.
2. **Manipulating the DOM (Document Object Model)**
    - JavaScript can change content, styles, and structure **dynamically** without reloading the page.
    - Example: Clicking a button changes a heading's text instantly.
3. **Event Handling**
    - Responds to user actions like clicks, mouse movements, and keyboard input.
    - Example: A login form that checks if fields are empty when the user presses "Submit".
4. **Asynchronous Communication**

- With **AJAX** or **Fetch API**, JavaScript can send/receive data from a server without refreshing the page.
- Example: Live search suggestions, chat messages, or weather updates.

5. **Full-Stack Development**
   - JavaScript runs on both the **frontend** (in the browser) and **backend** (using Node.js).
   - Popular frameworks and libraries: **React, Angular, Vue** (frontend) and **Express.js** (backend).
6. **Cross-Platform Usage**
   - Can be used to build **mobile apps** (React Native, Ionic), **desktop apps** (Electron), and even **games**.

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

1. **Platform & Execution**
   - **JavaScript:** Runs mostly in browsers (frontend), also on servers with Node.js.
   - **Python:** Runs with an interpreter, mostly for backend, data science, and AI.
   - **Java:** Runs on JVM (compiled to bytecode), used in enterprise and mobile apps.
2. **Typing**
   - **JavaScript & Python:** Dynamically typed (no need to declare types).
   - **Java:** Statically typed (must declare data types).
3. **Syntax**
   - **JavaScript:** Uses { } and ;.
   - **Python:** Uses indentation.
   - **Java:** Uses { } and ; but is more strict.
4. **Usage**
   - **JavaScript:** Web development (interactive websites, frontend + backend).
   - **Python:** AI, machine learning, scripting, backend.
   - **Java:** Large-scale systems, Android apps, enterprise software.
5. **OOP Approach**

- **JavaScript:** Prototype-based OOP.
- **Python & Java:** Class-based OOP.

In short:

- **JavaScript** = Web interactivity.
- **Python** = AI, data, backend.
- **Java** = Enterprise, mobile apps.

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 is used in **HTML** to add JavaScript code to a webpage.
- It can be used for:
  1. Writing **inline JavaScript** (directly inside HTML).
  2. Linking an **external JavaScript file**.
- It is usually placed in the **<head>** or **<body>** section of the HTML document.

---

1. Inline JavaScript Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Inline Script Example</title>
  <script>
    alert("Hello, this is inline JavaScript!");
  </script>
</head>
<body>
  <h2>Welcome</h2>
</body>
</html>
```

---

2. Linking External JavaScript File

- To keep code organized, JavaScript is usually written in a **separate file** with the extension .js.
- You link it using the <script> tag with the **src attribute**.

Example:

index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>External Script Example</title>
  <!-- Linking external JavaScript -->
  <script src="script.js"></script>
</head>
<body>
  <h2 id="demo">Hello World</h2>
  <button onclick="changeText()">Click Me</button>
</body>
</html>
```

script.js

```
function changeText() {
  document.getElementById("demo").innerText = "Text changed using external JS!";
}
```

Key Points:

- <script> tag is used to **embed or include JavaScript** in HTML.
- Use src="filename.js" to link an external JS file.
- External JS helps in **reusability** and **clean code separation**.
- Best practice: Place <script> before the closing </body> tag for faster page loading.

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

- A **variable** is a container used to store data values.
- Example: numbers, strings, objects, etc.
- Variables allow us to **reuse and manipulate data** in programs.

Declaring Variables in JavaScript

JavaScript provides **three keywords** to declare variables: var, let, and const.

1. var

- Old way (before ES6).
- Function-scoped (limited to the function where declared).
- Can be **redeclared** and **updated**.

var name = "John";

var name = "Mike"; // allowed

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

---

2. let

- Introduced in ES6 (modern way).
- Block-scoped (works only inside { }).
- Can be **updated** but **not redeclared** in the same scope.

let age = 25;

age = 30; // allowed (update)

// let age = 40; ❌ not allowed (redeclare in same scope)

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

---

3. const

- Block-scoped.
- Must be assigned a value at the time of declaration.
- **Cannot be updated or redeclared**.

const pi = 3.14;

// pi = 3.14159; ❌ not allowed (cannot update)

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

Summary Table

Keyword	Scope	Redeclare	Update	Use Case
var	Function scope	✅ Yes	✅ Yes	Old code / legacy support
let	Block scope	❌ No	✅ Yes	Variables that change
const	Block scope	❌ No	❌ No	Fixed values / constants

In short:

- Use let for variables that may change.
- Use const for values that stay constant.
- Avoid var (used in older JavaScript).

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

Data Types in JavaScript

JavaScript has **two categories** of data types:

1. **Primitive Data Types** (basic, immutable values)
2. **Non-Primitive (Reference) Data Types** (objects, arrays, functions, etc.)

---

1. Primitive Data Types

These hold a **single value** and are stored directly in memory.

1. **String** → Represents text.

```
let name = "John";
let greeting = 'Hello';
```

2. **Number** → Represents both integers and decimals.

```
let age = 25;
let price = 99.99;
```

3. **Boolean** → Represents true or false.

```
let isStudent = true;
let isAdmin = false;
```

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

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

5. **Null** → Represents an intentional empty value.

```
let car = null;
```

6. **Symbol (ES6)** → Represents a unique identifier.

```
let id1 = Symbol("id");
let id2 = Symbol("id");
console.log(id1 === id2); // false (unique values)
```

7. **BigInt (ES11)** → Represents very large integers.

```
let bigNumber = 1234567890123456789012345678901234567890n;
```

---

2. Non-Primitive (Reference) Data Types

These hold **collections of values or more complex entities**.

1. **Object** → Stores key-value pairs.

```
let person = {
  name: "Alice",
  age: 22,
  isStudent: true
};
```

2. **Array** → Ordered collection (list) of values.

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

3. **Function** → A block of code designed to perform a task.

```
function greet() {
  return "Hello World!";
}
```

Summary

Category	Data Types
Primitive	String, Number, Boolean, Undefined, Null, Symbol, BigInt
Non-Primitive	Object, Array, Function

In short:

- **Primitive types** hold single, immutable values.
- **Non-primitive types** hold collections or complex data.

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

Difference Between undefined and null in JavaScript

1. Meaning

- **undefined:** A variable is declared but not assigned a value.
- **null:** Represents an *intentional empty* or *nothing* value.

---

2. Type

- **typeof undefined** → "undefined"
- **typeof null** → "object" (this is a historical bug in JavaScript, but still exists).

---

3. Usage

- **undefined:** Default value given by JavaScript when no value is assigned.

- **null:** Assigned by the programmer to indicate "no value".

---

4. Example

let a;

console.log(a); // undefined (no value assigned)

let b = null;

console.log(b); // null (explicitly empty)

Summary Table

Feature	U**ndefined**	**null**
Meaning	Variable declared but no value assigned	Assigned empty value
Set by	JavaScript (default)	Programmer
Type	"undefined"	"object" (special case)
Example	let x; → undefined	let y = null;

In short:

- undefined = value not assigned.
- null = intentional empty value.

JavaScript Operators

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

- **Arithmetic operators**
- **Assignment operators**

• Comparison operators

• Logical operators

Operators in JavaScript

Operators are special symbols used to **perform operations on values and variables**.

JavaScript has many types, but here we'll focus on:

- **Arithmetic Operators**
- **Assignment Operators**
- **Comparison Operators**
- **Logical Operators**

---

1. Arithmetic Operators

Used for mathematical calculations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*****	Multiplication	5 * 3	15
/	Division	10 / 2	5
%	Modulus (remainder)	10 % 3	1
******	Exponentiation	2 ** 3	8
++	Increment	let x=5; x++;	6
--	Decrement	let x=5; x--;	4

---

2. Assignment Operators

Used to assign values to variables.

Operator	Example	Meaning
=	x = 10	Assigns 10 to x
+=	x += 5	x = x + 5
-=	x -= 3	x = x - 3

***=**	x *= 2	x = x * 2
/=	x /= 2	x = x / 2
%=	x %= 2	x = x % 2

---

3. Comparison Operators

Used to compare two values, returns true or false.

Operator	Example	Result
==	5 == "5"	true (checks only value)
===	5 === "5"	false (checks value + type)
!=	5 != 4	true
!==	5 !== "5"	true
>	10 > 5	true
<	10 < 5	false
>=	10 >= 10	true
<=	5 <= 10	true

---

4. Logical Operators

Used to combine conditions, returns true or false.

Operator	Example	Result
&& (AND)	(5 > 2 && 10 > 5)	true
`		` (OR)
! (NOT)	!(5 > 2)	false

Example in Code:

```
let a = 10, b = 5;


// Arithmetic
console.log(a + b); // 15
console.log(a % b); // 0
```

```
// Assignment
a += 5;
console.log(a); // 15

// Comparison
console.log(a > b);  // true
console.log(a === "15"); // false

// Logical
console.log(a > b && b > 2); // true
```

In short:

- **Arithmetic** → Math operations.
- **Assignment** → Assign/update values.
- **Comparison** → Compare values (true/false).
- **Logical** → Combine conditions.

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

Difference Between == and === in JavaScript

1. == (Equality Operator)

- Compares **only values**, not data types.
- Performs **type conversion** if needed.

```
console.log(5 == "5");   // true  (string "5" is converted to number 5)
console.log(true == 1);  // true  (true is converted to 1)
console.log(null == undefined); // true
```

---

2. === (Strict Equality Operator)

- Compares **both value and type**.
- Does **not** perform type conversion.

```
console.log(5 === "5");  // false (number vs string)
console.log(true === 1); // false (boolean vs number)
console.log(null === undefined); // false
```

Summary Table

Operator	Compares	Type Conversion	Example	Result
==	Only values	✅ Yes	5 == "5"	true
===	Value + type	❌ No	5 === "5"	false

In short:

- Use **==** when you only care about value (but can cause bugs).
- Use **===** for safer comparison (recommended).

Control Flow (If-Else, Switch)

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

What is Control Flow in JavaScript?

- **Control flow** is the order in which statements are executed in a program.
- Normally, code runs **top to bottom**, line by line.
- With control flow statements (like **if-else, switch, loops**), we can change the execution path depending on conditions.

---

if-else Statement in JavaScript

The **if-else** statement is used to make decisions in code.

Syntax:

```
if (condition) {
  // code runs if condition is true
} else {
  // code runs 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.");
}
```

How it works:

- If age >= 18 is true, it prints **"You are eligible to vote."**
- Otherwise, it prints **"You are not eligible to vote."**

Key Points:

- if checks the condition.
- else runs only if the if condition is false.
- You can also use **else if** for multiple conditions.

Example:

```
let marks = 75;

if (marks >= 90) {
  console.log("Grade A");
} else if (marks >= 60) {
  console.log("Grade B");
```

```
} else {
  console.log("Grade C");
}
```

In short:

- **Control flow** decides the order of execution.
- **if-else** allows the program to take different paths based on conditions.

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

Switch Statement in JavaScript

- A **switch statement** is used to execute **one block of code** out of many options.
- It's often used as a cleaner alternative to multiple **if-else-if** statements.

Syntax:

```
switch (expression) {
  case value1:
    // code runs if expression === value1
    break;

  case value2:
    // code runs if expression === value2
    break;

  default:
    // code runs if no case matches
}
```

- expression is evaluated once.
- The result is compared with each case value using **strict comparison (===)**.

- break ends the switch (without it, execution “falls through” to the next case).
- default runs if no cases match.

Example:

```
let day = 3;


switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  case 4:
    console.log("Thursday");
    break;
  default:
    console.log("Invalid day");
}
```

Output: Wednesday

When to Use switch Instead of if-else?

- Use **if-else** when you need to check **complex conditions** (e.g., ranges, logical operators).
- Use **switch** when you need to compare **one value** against **multiple possible exact matches**.

Example use cases for **switch**:

- Menu selection
- Day of the week
- User role (admin, editor, viewer)
- Handling multiple fixed options

In short:

- **switch** is best for multiple exact matches.
- **if-else** is best for conditions with ranges or complex logic.

Loops (For, While, Do-While)

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

Loops in JavaScript

A **loop** is used to execute a block of code repeatedly as long as a condition is true.

The main types of loops in JavaScript are:

1. **for loop**
2. **while loop**
3. **do-while loop**

---

1. for loop

- Best when the number of iterations is **known in advance**.

Syntax:

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

Example:

```
for (let i = 1; i <= 5; i++) {
```

```
  console.log("Number: " + i);
}
```

Prints numbers 1 to 5.

---

2. while loop

- Best when the number of iterations is **not known** and depends on a condition.

Syntax:

```
while (condition) {
  // code to be executed
}
```

Example:

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

Prints numbers 1 to 5.

---

3. do-while loop

- Similar to while, but it **executes at least once** (because the condition is checked **after** execution).

Syntax:

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

Example:

```
let i = 1;
do {
  console.log("Number: " + i);
  i++;
```

} while (i <= 5);

Prints numbers 1 to 5 (runs at least once even if condition is false).

Summary Table

Loop Type	Condition Checked	Executes At Least Once?	Best Used For
for	Before each loop	❌ No	When number of iterations is known
while	Before each loop	❌ No	When condition must be true to start
do-while	After each loop	✅ Yes	When code should run at least once

---

In short:

- **for** → fixed repetitions.
- **while** → repeats while condition is true.
- **do-while** → executes at least once, then checks condition.

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

Difference Between while loop and do-while loop

1. Condition Checking

- **while loop:** Condition is checked **before** executing the loop body.
- **do-while loop:** Condition is checked **after** executing the loop body.

---

2. Minimum Execution

- **while loop:** May run **0 times** if condition is false initially.
- **do-while loop:** Runs **at least once**, even if condition is false.

---

3. Syntax & Example

while loop:

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

Output: nothing (condition false at start).

do-while loop:

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

Output: Hello (runs once before checking condition).

Summary Table

Feature	**while loop**	**do-while loop**
Condition checked	Before body	After body
Minimum executions	0 times	1 time
Use case	When loop should only run if condition is true	When loop must run at least once

---

In short:

- **while** → checks first, may not run.
- **do-while** → runs once, then checks.

Functions

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

What are Functions in JavaScript?

A **function** in JavaScript is a block of code designed to perform a specific task.

- It helps to **reuse code** (write once, use many times).
- Improves **readability** and **maintainability** of programs.

Syntax for Declaring a Function

```
function functionName(parameters) {
  // block of code
  return value; // optional
}
```

- **function** → keyword to declare a function.
- **functionName** → name you give to the function.
- **parameters** → values you pass into the function (optional).
- **return** → sends a value back (optional).

Syntax for Calling a Function

```
functionName(arguments);
```

- **arguments** → actual values passed to the function.

Example

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

```
// Function call
greet("Charmi");
greet("Alex");
```

Output:

```
Hello, Charmi!
Hello, Alex!
```

Key Points

- A function runs only when it is **called**.
- You can call it **multiple times** with different values.
- Functions can **return values** or just perform an action.

In short:
Functions = reusable blocks of code.
Declare → function myFunc() { ... }
Call → myFunc();

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

1. Function Declaration

A **function declaration** defines a named function using the function keyword.

- It is **hoisted** (can be called before it is defined).

Syntax:

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

greet(); // ✅ Works even if called before declaration
```

---

2. Function Expression

A **function expression** creates a function and assigns it to a variable.

- It is **not hoisted** (can only be called after it is defined).

Syntax:

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

greet(); // ✅ Works only if called after the definition
```

Key Differences

Feature	Function Declaration	Function Expression
Hoisting	Yes (can call before definition)	No (must define first)
Name	Always has a name	Can be named or anonymous
When to use	For general reusable functions	When assigning to variables, passing as arguments, or using callbacks

---

Example Showing the Difference

```
// Function Declaration
sayHello(); // ✅ Works (hoisted)
function sayHello() {
  console.log("Hello from declaration!");
}

// Function Expression
sayHi(); // ❌ Error (not hoisted)
const sayHi = function() {
  console.log("Hello from expression!");
};
```

In short:

- **Function Declaration** → hoisted, can be called before it's defined.
- **Function Expression** → not hoisted, must be defined before calling.

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

Parameters in Functions

- **Parameters** are variables listed inside the function's parentheses.
- They act as **placeholders** for values that will be passed when the function is called.

Example with Parameters:

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

greet("Charmi");  // "Charmi" is an argument
greet("Alex");
```

Output:

```
Hello, Charmi!
Hello, Alex!
```

Return Values in Functions

- A function can **return a value** using the return keyword.
- The returned value can be stored in a variable or used directly.

Example with Return Value:

```
function add(a, b) {
  return a + b;  // returns the sum
}
```

```
let result = add(5, 3);

console.log(result); // 8
```

Here:

- a and b are **parameters**.
- return a + b; sends back the result.
- result stores the returned value.

Key Points

- **Parameters** = input placeholders inside the function.
- **Arguments** = actual values passed when calling.
- **Return value** = output given back by the function.

In short:
Parameters allow functions to take **inputs**, and return allows functions to give **outputs**.

Arrays

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

An **array** is a special type of object used to store **multiple values** in a single variable.

- Values in an array are called **elements**.
- Each element has an **index** (starting from 0).
- Arrays can hold any data type: numbers, strings, objects, even other arrays.

Declaring and Initializing Arrays

1. Using Square Brackets [] (most common)

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

```
console.log(fruits);      // ["Apple", "Banana", "Mango"]
console.log(fruits[0]);   // Apple (first element)
```

---

2. Using new Array() Constructor

```
let numbers = new Array(10, 20, 30, 40);
console.log(numbers);     // [10, 20, 30, 40]
```

Note: Using new Array(5) creates an array with 5 **empty slots**, not numbers.

---

3. Empty Array then Assigning Values

```
let colors = [];
colors[0] = "Red";
colors[1] = "Green";
colors[2] = "Blue";

console.log(colors);      // ["Red", "Green", "Blue"]
```

Example with Mixed Data Types

```
let mixed = [100, "Hello", true, { name: "Charmi" }, [1, 2, 3]];
console.log(mixed);
```

Arrays can hold **different types of values** together.

Key Points

- Arrays are **zero-indexed** (first element at index 0).
- Can store **multiple values** in one variable.
- Declared using [] (preferred) or new Array().

In short:

An **array** = a collection of values in a single variable.
Example: let arr = [1, 2, 3];

An **array** is a special type of object used to store **multiple values** in a single variable.

- Values in an array are called **elements**.
- Each element has an **index** (starting from 0).
- Arrays can hold any data type: numbers, strings, objects, even other arrays.

Declaring and Initializing Arrays

1. Using Square Brackets [] (most common)

```
let fruits = ["Apple", "Banana", "Mango"];
console.log(fruits);      // ["Apple", "Banana", "Mango"]
console.log(fruits[0]);   // Apple (first element)
```

---

2. Using new Array() Constructor

```
let numbers = new Array(10, 20, 30, 40);
console.log(numbers);     // [10, 20, 30, 40]
```

Note: Using new Array(5) creates an array with 5 **empty slots**, not numbers.

---

3. Empty Array then Assigning Values

```
let colors = [];
colors[0] = "Red";
colors[1] = "Green";
colors[2] = "Blue";

console.log(colors);      // ["Red", "Green", "Blue"]
```

Example with Mixed Data Types

```
let mixed = [100, "Hello", true, { name: "Charmi" }, [1, 2, 3]];
console.log(mixed);
```

Arrays can hold **different types of values** together.

Key Points

- Arrays are **zero-indexed** (first element at index 0).
- Can store **multiple values** in one variable.
- Declared using [] (preferred) or new Array().

In short:
An **array** = a collection of values in a single variable.
Example: let arr = [1, 2, 3];

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

1. push()

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

Example:

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

fruits.push("Mango");

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

---

2. pop()

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

Example:

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

let removed = fruits.pop();

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

console.log(removed);  // "Mango"
```

---

3. shift()

- Removes the **first** element from the array.
- Returns the removed element.

Example:

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

---

4. unshift()

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

Example:

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

Summary Table

Method	Action	Returns
push()	Adds element(s) to the end	New array length
pop()	Removes last element	Removed element
shift()	Removes first element	Removed element
unshift()	Adds element(s) to beginning	New array length

In short:

- 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** is a collection of **key-value pairs** (also called properties).
- Keys (also called properties) are **strings**, and values can be **any data type** including arrays, functions, or other objects.
- Objects are used to **represent real-world entities** and store structured data.

Example:

```
let person = {
  name: "Charmi",
  age: 23,
  isStudent: true
};

console.log(person.name); // "Charmi"
console.log(person.age);  // 23
```

Difference Between Objects and Arrays

Feature	Object	Array
Structure	Key-value pairs	Ordered list of values
Access	By key (obj.key or obj["key"])	By index (arr[0])
Use Case	Represent entities with properties	Store collections of items
Example	{name: "Alice", age: 20}	[10, 20, 30, 40]

Key Points

- Objects = unordered **named properties**.
- Arrays = ordered **indexed elements**.

- Arrays are a type of object in JavaScript, but with **numeric indexes** and special methods like push, pop.

In short:

- **Objects** = store data as **properties** (key-value pairs).
- **Arrays** = store data as **lists** (ordered by index).

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

Accessing and Updating Object Properties in JavaScript

There are **two main ways** to access and update object properties:

1. **Dot Notation**
2. **Bracket Notation**

---

1. Dot Notation

- Use the **dot (.)** followed by the property name.
- Works only when the property name is a valid identifier (no spaces, doesn't start with a number).

Example:

```
let person = {
  name: "Charmi",
  age: 23
};

// Access property
console.log(person.name); // "Charmi"

// Update property
person.age = 26;
console.log(person.age);  // 26
```

```
// Add new property
person.city = "Ahmedabad";
console.log(person.city); // "Ahmedabad"
```

---

2. Bracket Notation

- Use **square brackets [ ]** with the property name as a **string**.
- Useful when property names have **spaces, special characters, or dynamic keys**.

Example:

```
let person = {
  "first name": "Charmi",
  age: 25
};

// Access property
console.log(person["first name"]); // "Charmi"

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

// Add new property dynamically
let key = "city";
person[key] = "Ahmedabad";
console.log(person.city);         // "Ahmedabad"
```

Key Points

Feature	Dot Notation	Bracket Notation
Syntax	obj.key	obj["key"]

When to use	Normal property names	Special characters, spaces, or dynamic keys
Access & Update	obj.key = value	obj["key"] = value

In short:

- **Dot notation** → simple and common.
- **Bracket notation** → flexible for dynamic or special property names.

JavaScript Events

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

- **Events** are actions or occurrences that happen in the browser, usually triggered by the user or the browser itself.
- Examples of events:
  - User clicks a button → click
  - User moves the mouse → mousemove
  - User types in an input field → keydown, keyup
  - Page finishes loading → load
- Events allow web pages to be **interactive**.

🔹 **Role of Event Listeners**

- An **event listener** is a function that **waits for a specific event** to occur on an element and then executes code.
- It **"listens"** for the event and responds when it happens.

Syntax:

element.addEventListener(event, function, useCapture);

- event → the type of event to listen for (click, mouseover, etc.)
- function → the function to run when the event occurs
- useCapture → optional, usually false

Example:

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


<script>
let button = document.getElementById("myBtn");


// Add event listener
button.addEventListener("click", function() {
  alert("Button was clicked!");
});
</script>
```

How it works:

- The browser **listens** for a click event on the button.
- When clicked, the function runs and shows the alert.

Key Points:

- **Events** = actions happening in the browser.
- **Event listeners** = functions that respond to events.
- Using addEventListener is **preferred** over inline HTML events (onclick) because it **keeps JavaScript separate** from HTML.

In short:

- Events = triggers (click, hover, type).
- Event listeners = “watchers” that execute code when events happen.

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

How addEventListener() Works in JavaScript

- The addEventListener() method **attaches an event handler** to an HTML element.
- It allows the element to **"listen"** for a specific event and **execute a function** when that event occurs.
- Advantages over inline events:
    - You can attach **multiple events** to the same element.
    - Keeps JavaScript **separate from HTML**.

Syntax:

element.addEventListener(event, function, useCapture);

Parameter	Description
event	Type of event (e.g., "click", "mouseover")
function	Function to execute when the event occurs
useCapture	Optional, true or false (usually false)

Example:

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


<script>
let button = document.getElementById("myBtn");


// Attach a click event listener
button.addEventListener("click", function() {
  alert("Button clicked!");
});
</script>
```

Explanation:

1. getElementById("myBtn") selects the button element.
2. addEventListener("click", function(){...}) attaches a click event to it.
3. When the button is clicked, the function runs and shows an alert.

Key Points:

- addEventListener allows **dynamic and multiple event handling**.
- Event handler can be an **anonymous function** or a **named function**.

Example with a named function:

```
function showMessage() {
  alert("Hello!");
}

button.addEventListener("click", showMessage);
```

In short:

- addEventListener() = a way to **listen and respond to events** on HTML elements.
- Preferred over inline events for **cleaner, flexible code**.

DOM Manipulation

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

- **DOM (Document Object Model)** is a **programmatic representation of an HTML document**.
- It represents the **page as a tree structure** where:
  - **Nodes** = elements, text, attributes
  - **Branches** = relationships between elements (parent, child, siblings)
- The DOM allows JavaScript to **access, modify, and manipulate HTML elements dynamically**.

Example of DOM Structure:

For this HTML:

```
<!DOCTYPE html>
<html>
  <body>
    <h1 id="title">Hello World</h1>
    <p>Welcome to JavaScript!</p>
  </body>
</html>
```

The DOM tree looks like:

```
document
 └─ html
    └─ body
       ├─ h1#title
       └─ p
```

How JavaScript Interacts with the DOM

- **Access Elements:** Find HTML elements using methods like:
    - getElementById(), getElementsByClassName(), querySelector(), querySelectorAll()
- **Modify Content:** Change text or HTML inside elements using innerText or innerHTML.
- **Change Styles:** Modify CSS using style property.
- **Add/Remove Elements:** Use methods like appendChild(), removeChild().
- **Handle Events:** Attach event listeners using addEventListener().

Example: Interacting with the DOM

```
<h1 id="title">Hello World</h1>
<button id="btn">Change Text</button>

<script>
```

```
let heading = document.getElementById("title");
let button = document.getElementById("btn");

button.addEventListener("click", function() {
  heading.innerText = "Hello JavaScript!";
  heading.style.color = "blue";
});
</script>
```

Explanation:

- getElementById("title") → Access the <h1> element.
- innerText → Change the text.
- style.color → Change the CSS.
- addEventListener("click", ...) → React to user clicking the button.

Key Points:

- DOM = interface to **represent HTML as a tree of objects**.
- JavaScript interacts with DOM to **dynamically change content, structure, and style**.
- Makes webpages **interactive**.

In short:

- **DOM** = structured representation of HTML.
- **JavaScript** = tool to manipulate that structure dynamically.

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

1. getElementById()

- Selects a **single element** with the specified id.
- Returns **one element object**.

- Fast and widely used.

Syntax:

let element = document.getElementById("elementId");

Example:

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

<script>

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

heading.innerText = "Hello JavaScript!";

</script>

---

2. getElementsByClassName()

- Selects **all elements** with the specified class name.
- Returns an **HTMLCollection** (like an array, but not exactly).
- Can access elements using **index**.

Syntax:

let elements = document.getElementsByClassName("className");

Example:

Paragraph 1

Paragraph 2

<script>

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

paragraphs[0].style.color = "red"; // First paragraph

paragraphs[1].style.color = "blue"; // Second paragraph

</script>

---

3. querySelector()

- Selects the **first element** that matches a **CSS selector**.
- Very flexible: can use **id (#id), class (.class), tag name, attribute selectors**.

Syntax:

let element = document.querySelector("CSS selector");

Example:

Paragraph 1

Paragraph 2

<script>

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

firstParagraph.style.fontWeight = "bold"; // Only first element

</script>

Note: To select **all matching elements**, use querySelectorAll() (returns a NodeList).

Summary Table

Method	Returns	Selects	Notes
getElementById("id")	Single element	Element with specific id	Fast, only one element
getElementsByClassName("class")	HTMLCollection	All elements with class	Access by index
querySelector("selector")	Single element	First element matching CSS selector	Flexible, works with id, class, tag
querySelectorAll("selector")	NodeList	All elements matching CSS selector	Use for multiple elements

---

In short:

- **getElementById** → fastest, single element by ID
- **getElementsByClassName** → multiple elements by class
- **querySelector** → first element using CSS selector

JavaScript Timing Events (setTimeout, setInterval)

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

1. setTimeout()

- **Purpose:** Executes a function **once** after a specified delay (in milliseconds).
- Useful for **delayed actions** like showing messages, animations, or timeouts.

Syntax:

setTimeout(function, delay);

- function → the function to execute
- delay → time in milliseconds (1000 ms = 1 second)

Example:

```
setTimeout(function() {
  alert("Hello after 3 seconds!");
}, 3000); // 3000 ms = 3 seconds
```

The function runs **once** after 3 seconds.

---

2. setInterval()

- **Purpose:** Executes a function **repeatedly** at a fixed time interval (in milliseconds).
- Useful for **clocks, timers, animations, or polling data**.

Syntax:

setInterval(function, interval);

- function → function to execute
- interval → time in milliseconds between executions

Example:

```
let count = 1;
let intervalId = setInterval(function() {
  console.log("Count: " + count);
  count++;
  if(count > 5) {
    clearInterval(intervalId); // stop the interval after 5 times
```

```
  }
}, 1000);
```

Prints Count: 1 to Count: 5 at **1-second intervals**.

Key Points

Function	Executes	Use Case
setTimeout()	Once	Delayed message, timeout
setInterval()	Repeatedly	Clock, timer, repeated actions

- **Stopping a timer:**
    - clearTimeout(timeoutId) → stops a timeout
    - clearInterval(intervalId) → stops an interval

In short:

- **setTimeout** → delay once
- **setInterval** → repeat at intervals
- Both control timing events in JavaScript.

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

```
console.log("Action will happen in 2 seconds...");

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

Explanation:

setTimeout() waits for 2000 milliseconds (2 seconds).

After the delay, it executes the function inside, printing the message.

The first console.log runs immediately; the second runs after 2 seconds.

Output (timeline):

Immediately: Action will happen in 2 seconds...

After 2 seconds: This message appears after 2 seconds!

JavaScript Error Handling

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

- **Error handling** is the process of responding to **runtime errors** in a program without stopping the entire execution.
- Helps make programs **robust** and **prevent crashes**.

---

try, catch, and finally

1. **try block**
    - Contains the code that **may throw an error**.
2. **catch block**
    - Executes if an error occurs in the try block.
    - Receives an **error object** containing details about the error.
3. **finally block** (optional)
    - Executes **always**, whether an error occurred or not.
    - Useful for **cleanup actions** like closing files, stopping timers, etc.

Syntax:

try {

```
  // code that may throw an error
} catch(error) {
  // code to handle the error
} finally {
  // code that always runs
}
```

Example:

```
try {
  let result = riskyOperation(); // may throw an error
  console.log("Result:", result);
} catch(error) {
  console.log("An error occurred:", error.message);
} finally {
  console.log("This block runs always, error or not.");
}

// Example function that throws an error
function riskyOperation() {
  throw new Error("Something went wrong!");
}
```

Output:

```
An error occurred: Something went wrong!
This block runs always, error or not.
```

Key Points:

- try → run code safely.
- catch → handle errors gracefully.
- finally → always runs, for cleanup tasks.

- Helps **prevent program crashes** and improves reliability.

In short:
Error handling = safely dealing with runtime errors using try-catch-finally.

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

1. **Prevents Program Crashes**
    - Without error handling, a runtime error stops the whole script.
    - Proper handling ensures the application **keeps running smoothly** even if an error occurs.
2. **Improves User Experience**
    - Users see friendly messages instead of broken pages or console errors.
    - Example: Showing “Unable to load data” instead of a blank page.
3. **Helps Debugging**
    - Error objects provide **information about the error** (message, type, stack trace).
    - Makes it easier for developers to **find and fix issues**.
4. **Maintains Data Integrity**
    - Ensures **critical operations** (like saving data, processing forms) are completed safely.
    - Prevents **corrupted data** or inconsistent application states.
5. **Supports Asynchronous Operations**
    - In modern JS apps using APIs or timers, errors can occur at any time.
    - Error handling ensures **asynchronous tasks** are managed safely.

Example:

```
try {
  let data = JSON.parse("Invalid JSON"); // Will throw an error
} catch(error) {
```

```
  console.log("Failed to parse data:", error.message);
}
console.log("App continues running...");
```

Output:

```
Failed to parse data: Unexpected token I in JSON at position 0
App continues running...
```

In short:

- **Error handling** ensures your JavaScript application is **robust, user-friendly, and reliable**.
- Without it, errors can **crash apps, frustrate users, and cause data loss**.