MODULE 1

FULLSTACK DEVELOPMENT

Basic JavaScript Instructions, Statements, Comments, Variables, Data Types, Arrays, Strings, Functions, Methods & Objects, Decisions & Loops.

Introduction to JavaScript

◈ What is JavaScript?

JavaScript (JS) is a lightweight, interpreted, client-side scripting language used for making web pages interactive. It is used to create dynamic web applications, handle events, validate forms, and more.

◈ Features of JavaScript

- Client-Side Execution: Runs in the browser, reducing server load.
- Interactivity: Enables user interaction (e.g., form validation, animations).
- Event-Driven: Executes code based on user actions (e.g., clicks, keypresses).
- Cross-Platform: Works across all browsers and devices.
- Supports Object-Oriented Programming (OOP).

◈ History of JavaScript

- 1995: Developed by Brendan Eich at Netscape (originally named Mocha).
- 1996: Renamed to JavaScript and became part of Netscape Navigator.
- 1997: Standardized as ECMAScript (ES1) by ECMA International.
- 2009: ES5 introduced modern features.

- 2015: ES6 (ECMAScript 2015) introduced let, const, arrow functions, classes, and modules.
- Present: Regular updates (ES6, ES7, ES8, ESNext) continue improving JS.

Basic JavaScript Instructions

◈ Writing JavaScript

JavaScript can be written in three ways:

1. Inline JS (inside HTML elements)

```
<button onclick="alert('Hello, World!')">Click Me</button>
```

2. Internal JS (inside <script> tag in an HTML file)

```
<script>
   document.write("Hello, JavaScript!");
</script>
```

3. External JS (stored in a .js file)

```
// script.js
console.log("Hello, JavaScript!");

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

JavaScript Statements

JavaScript **statements** are instructions that the browser executes. A JavaScript program consists of a series of statements, which **control execution flow** and define the logic of the program.

Each statement is **executed in order**, from **top to bottom**, unless control statements (like loops or conditions) modify the flow.

◈ Types of JavaScript Statements

1. Declaration Statements
2. Assignment Statements
3. Expression Statements
4. Control Flow Statements
5. Looping Statements
6. Function Statements
7. Error Handling Statements

1 Declaration Statements

Declaration statements define variables, constants, or functions.

Example: Variable Declaration

```
var x;       // Declares a variable
let y = 10;  // Declares and assigns a value
const PI = 3.14; // Declares a constant
```

Example: Function Declaration

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

greet(); // Calls the function

2 Assignment Statements

Used to **assign values** to variables.

Example:

let a = 5; // Assigns 5 to a

let b = 10; // Assigns 10 to b

b += a; // Adds a to b (b = b + a → 15)

console.log(b);

3 Expression Statements

Expressions that produce a value.

Example:

```
let result = 10 + 5;  // Addition expression
console.log(result);  // Output: 15
```

4 Control Flow Statements

These statements control the execution flow.

Example: if...else

```
let age = 18;
if (age >= 18) {
   console.log("You can vote.");
} else {
   console.log("You cannot vote.");
}
```

Example: Switch Case

```
let grade = 'B';
switch (grade) {
   case 'A':
      console.log("Excellent!");
      break;
   case 'B':
      console.log("Good job!");
      break;
   case 'C':
      console.log("Needs improvement.");
      break;
   default:
      console.log("Invalid grade.");
}
```

5 Looping Statements

Used to execute a block of code multiple times.

Example: for Loop

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

Example: while Loop

```
let count = 3;
while (count > 0) {
   console.log("Countdown: " + count);
```

```
    count--;
}
```

6 Function Statements

Define reusable blocks of code.

Example: Function Definition & Call

```
function add(a, b) {
    return a + b;
}
console.log(add(5, 10)); // Output: 15
```

7 Error Handling Statements

Used to handle errors gracefully.

Example: try...catch

```
try {
    let num = 10 / 0; // Possible error
    console.log(num);
} catch (error) {
    console.log("An error occurred: " + error);
}
```


Summary

- **JavaScript statements** execute instructions in a program.
- **Declaration statements** define variables and functions.
- **Assignment statements** assign values.
- **Expression statements** compute values.
- **Control statements** manage program flow (if, switch).

- **Looping statements** repeat actions (for, while).
- **Function statements** define reusable code blocks.
- **Error handling statements** prevent crashes.

Variables

- A variable is a value assigned to an identifier, so you can reference and use it later in the program.
- This is because JavaScript is **loosely typed**, a concept you'll frequently hear about.
- A variable must be declared before you can use it.

We have 2 main ways to declare variables. The first is to use const:

const a = 0

The second way is to use let:

let a = 0

What's the difference?

- const defines a constant reference to a value. This means the reference cannot be changed. You cannot reassign a new value to it.
- Using let you can assign a new value to it.

For example, you cannot do this:

const a = 0

a = 1

Because you'll get an error: TypeError: Assignment to constant variable..

On the other hand, you can do it using let:

let a = 0

a = 1

const does not mean "constant" in the way some other languages like C mean. In particular, it does not mean the value cannot change - it means it cannot be reassigned. If the variable points to an object or an array (we'll see more about objects and arrays later) the content of the object or the array can freely change.

const variables must be initialized at the declaration time:

const a = 0

but let values can be initialized later:

let a

a = 0

You can declare multiple variables at once in the same statement:

const a = 1, b = 2

let c = 1, d = 2

But you cannot redeclare the same variable more than one time:

let a = 1

let a = 2

or you'd get a "duplicate declaration" error.

Advice is to always use const and only use let when you know you'll need to reassign a value to that variable. Why? Because the less power our code has, the better. If we know a value cannot be reassigned, it's one less source for bugs.

Now that we saw how to work with const and let, I want to mention var.

Until 2015, var was the only way we could declare a variable in JavaScript.

Today, a modern codebase will most likely just use const and let.

Data Types

What are Data Types in JavaScript

Data types define the kind of values a variable can hold in JavaScript.
JavaScript is dynamically typed, meaning we don't need to declare the data type explicitly; it is assigned automatically based on the value.

◈ JavaScript Data Types Categories

JavaScript has two main types of data:

1. Primitive Data Types (Stores a single value)
2. Non-Primitive (Reference) Data Types (Stores collections of values)

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

➢ Primitive Data Types

These are immutable (cannot be changed) and store single values.

1. String

Used to store text, enclosed in single ('), double ("), or backticks (`).

```
let name = "Deepika";
let greeting = 'Hello';
let message = `Welcome to JavaScript!`;
console.log(name, greeting, message);
```

2. Number

Holds both integer and floating-point values.

```
let age = 25;
let price = 99.99;
console.log(age, price);
```

3. Boolean

Represents true or false values.

```
let isStudent = true;
let hasPassed = false;
console.log(isStudent, hasPassed);
```

4. BigInt

Used for large numbers beyond Number.MAX_SAFE_INTEGER.

```
let bigNum = 9007199254740991n; // Add 'n' at the end for BigInt
console.log(bigNum);
```

5. Undefined

A variable is undefined when it is declared but not assigned a value.

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

6. Null

Represents an intentional absence of value.

```
let data = null;
console.log(data);
```

7. Symbol

Used to create unique identifiers.

```
let id1 = Symbol('id');
let id2 = Symbol('id');
console.log(id1 === id2); // Output: false (Symbols are always unique)
```

- **Non-Primitive (Reference) Data Types**

These store collections or functions.

1. Object

Used to store key-value pairs.

```
let student = {
   name: "Deepika",
   age: 22,
   course: "MERN Stack"
};
console.log(student.name, student.age);
```

2. Array

Stores multiple values in a single variable.

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

3. Function

A block of reusable code.

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

◈ Checking Data Type in JavaScript

Use the typeof operator to check a variable's type.

```
console.log(typeof "Hello");    // String
console.log(typeof 42);         // Number
console.log(typeof true);       // Boolean
console.log(typeof null);       // Object (a known JavaScript bug)
console.log(typeof undefined);  // Undefined
console.log(typeof {});         // Object
console.log(typeof []);         // Object (Arrays are objects)
console.log(typeof function(){}); // Function
```

JavaScript Arrays and Methods

◈ What is an Array in JavaScript?

An **array** is a special data structure in JavaScript that can hold **multiple values** in a single variable. It allows storing **ordered collections of elements**, where each element can be of any data type, including numbers, strings, objects, or even other arrays.

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

Arrays in JavaScript are **zero-indexed**, meaning the first element is at index 0, the second at 1, and so on.

◈ How to Create Arrays?

1. Using Array Literal (Recommended)

The most common way to create an array is by using **square brackets []**.

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

2. Using the new Array() Constructor

This approach is less commonly used but still valid.

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

3. Creating an Empty Array and Adding Elements

```
let cities = [];
cities[0] = "New York";
cities[1] = "London";
console.log(cities);
```

◈ Accessing Array Elements

Each item in an array has an **index** starting from 0. We can access elements using square brackets [].

```
let animals = ["Cat", "Dog", "Elephant"];
console.log(animals[0]); // Output: Cat
console.log(animals[2]); // Output: Elephant
```

If we access an index that doesn't exist, we get undefined.

```
console.log(animals[5]); // Output: undefined
```


Array Methods

JavaScript provides **built-in methods** to perform different operations on arrays.

1. Adding and Removing Elements

◈ push() – Add at the End

Adds an element at the end of the array.

```
let cars = ["BMW", "Audi"];
cars.push("Mercedes");
console.log(cars); // Output: ["BMW", "Audi", "Mercedes"]
```

◈ pop() – Remove from the End

Removes the last element from the array.

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

◈ unshift() – Add at the Beginning

Adds an element at the beginning of the array.

```
let nums = [2, 3, 4];
nums.unshift(1);
console.log(nums); // Output: [1, 2, 3, 4]
```

◈ shift() – Remove from the Beginning

Removes the first element from the array.

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

2. Slicing and Splicing Arrays

◈ slice(start, end) – Extract a Part of an Array

Returns a new array with elements from the given start to end index (excluding the end index).

```
let numbers = [10, 20, 30, 40, 50];
let slicedArray = numbers.slice(1, 3);
console.log(slicedArray); // Output: [20, 30]
```

◈ splice(start, deleteCount, item1, item2, ...) – Modify the Array

Can add or remove elements from an array at any position.

```
let fruits = ["Apple", "Banana", "Cherry"];
fruits.splice(1, 1, "Mango");
console.log(fruits); // Output: ["Apple", "Mango", "Cherry"]
```

- **1st parameter**: Start index (1)
- **2nd parameter**: Number of elements to delete (1)
- **Other parameters**: New elements to add ("Mango")

3. Searching in an Array

◈ indexOf(value) – Find Position

Returns the index of the first occurrence of the value, or -1 if not found.

```
let animals = ["Dog", "Cat", "Elephant"];
console.log(animals.indexOf("Cat")); // Output: 1
console.log(animals.indexOf("Tiger")); // Output: -1
```

◈ includes(value) – Check if Exists

Returns true if the element exists in the array, otherwise false.

```
let cities = ["Paris", "London", "New York"];
console.log(cities.includes("Paris")); // Output: true
console.log(cities.includes("Tokyo")); // Output: false
```

4. Combining and Transforming Arrays

◈ concat() – Merge Two Arrays

Joins two or more arrays into a new array.

```
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let mergedArray = arr1.concat(arr2);
console.log(mergedArray); // Output: [1, 2, 3, 4, 5, 6]
```

◈ join(separator) – Convert to String

Converts an array into a string with a specified separator.

```
let words = ["Hello", "World"];
console.log(words.join(" ")); // Output: "Hello World"
```

5. Sorting and Reversing Arrays

◈ sort() – Sort Elements

Sorts elements **alphabetically** (default).

```
let names = ["John", "Alice", "Bob"];
console.log(names.sort()); // Output: ["Alice", "Bob", "John"]
```

◈ Sorting Numbers (Using a Compare Function)

JavaScript's default sort() method does **not** correctly sort numbers, so we need a function.

```
let numbers = [40, 100, 1, 5, 25, 10];
numbers.sort((a, b) => a - b); // Ascending order
console.log(numbers); // Output: [1, 5, 10, 25, 40, 100]
```

◈ reverse() – Reverse Elements

Reverses the order of elements in an array.

```
let nums = [1, 2, 3, 4, 5];
console.log(nums.reverse()); // Output: [5, 4, 3, 2, 1]
```

6. Iterating Over an Array

◈ forEach() – Loop Through Elements

Executes a function for each element.

```
let cars = ["BMW", "Audi", "Mercedes"];
cars.forEach(car => console.log(car));
```

◈ map() – Create a New Array

Creates a new array by applying a function to each element.

```
let numbers = [1, 2, 3];
let squares = numbers.map(num => num * num);
console.log(squares); // Output: [1, 4, 9]
```

◈ filter() – Filter Values

Returns a new array with elements that satisfy a condition.

```
let ages = [18, 25, 30, 15, 20];
let adults = ages.filter(age => age >= 18);
console.log(adults); // Output: [18, 25, 30, 20]
```

◈ reduce() – Reduce to a Single Value

Calculates a single value (sum, product, etc.) from the array.

```
let numbers = [1, 2, 3, 4];
let sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum); // Output: 10
```


JavaScript Strings

A **string** in JavaScript is a sequence of characters enclosed in **single (' ')**, **double (" ")**, or **backticks ( )**. Strings are **immutable**, meaning once a string is created, it cannot be changed. However, JavaScript provides several **built-in methods** to manipulate and work with strings. Below are the most commonly used **string methods** explained in detail.

1. toUpperCase() - Convert String to Uppercase

The toUpperCase() method converts all lowercase characters in a string to uppercase.

Syntax:

```
string.toUpperCase();
```

Example:

let text = "hello world";

console.log(text.toUpperCase()); // Output: "HELLO WORLD"

- Used to normalize text before comparison.
- Helps in formatting text for display.

2. toLowerCase() - Convert String to Lowercase

The toLowerCase() method converts all uppercase characters in a string to lowercase.

Example:

let text = "JavaScript";

console.log(text.toLowerCase()); // Output: "javascript"

- Helps in case-insensitive comparisons.
- Useful when storing usernames or emails.

3. length - Find String Length

The length property returns the number of characters in a string.

Example:

let msg = "Hello";

console.log(msg.length); // Output: 5

- Helps in **validating** user input length (e.g., passwords).
- Used in loops to process each character in a string.

4. trim() - Remove Whitespace

The trim() method removes extra spaces from both ends of a string.

Example:

let userInput = " Hello ";

console.log(userInput.trim()); // Output: "Hello"

- Used before storing user input to avoid unintended spaces.
- Helps in text validation (e.g., checking if input is empty).

5. slice(start, end) - Extract a Substring

The slice() method extracts part of a string and returns it as a new string.

Example:

let str = "JavaScript";

console.log(str.slice(0, 4)); // Output: "Java"

console.log(str.slice(-6)); // Output: "Script"

- Used for extracting a portion of a string (e.g., a username from an email).
- Supports negative indices to start counting from the end.

6. substring(start, end) - Extract a Substring (Alternative to slice)

The substring() method works similarly to slice(), but it does not support negative indices.

Example:

let str = "Hello World";

console.log(str.substring(0, 5)); // Output: "Hello"

console.log(str.substring(6)); // Output: "World"

- Often used for extracting parts of a string where negative indexing is not needed.

7. replace(oldValue, newValue) - Replace Text in a String

The replace() method replaces the first occurrence of a word or character in a string.

Example:

let text = "I love JavaScript";

console.log(text.replace("JavaScript", "Python")); // Output: "I love Python"

- Used to replace specific words dynamically.
- Often combined with regular expressions for advanced replacements.

8. replaceAll(oldValue, newValue) - Replace All Occurrences

The replaceAll() method replaces **all** occurrences of a specific substring in a string.

Example:

let sentence = "Apples are tasty. Apples are juicy.";

console.log(sentence.replaceAll("Apples", "Oranges"));

// Output: "Oranges are tasty. Oranges are juicy."

- Replaces multiple instances of a word in large text.

9. concat() - Merge Strings

The concat() method joins two or more strings.

Example:

```
let first = "Hello";
let second = "World";
console.log(first.concat(" ", second)); // Output: "Hello World"
```

- Used for constructing sentences dynamically.

10. split(separator) - Convert String to Array

The split() method splits a string into an array based on a given delimiter.

Example:

```
let words = "apple,banana,grape";
let arr = words.split(",");
console.log(arr); // Output: ["apple", "banana", "grape"]
```

- Converts comma-separated values (CSV) into an array.
- Useful in breaking sentences into words.

11. charAt(index) - Get Character at Position

The charAt() method returns the character at a given index.

Example:

```
let text = "JavaScript";
console.log(text.charAt(4)); // Output: "S"
```

- Used to fetch individual characters in a string.

12. indexOf(value) - Find First Occurrence

The indexOf() method returns the index of the **first occurrence** of a specified value.

Example:

let text = "Hello JavaScript";

console.log(text.indexOf("JavaScript")); // Output: 6

- Helps in checking if a word exists in a string.

13. lastIndexOf(value) - Find Last Occurrence

The lastIndexOf() method returns the index of the **last occurrence** of a specified value.

Example:

let text = "Hello JavaScript JavaScript";

console.log(text.lastIndexOf("JavaScript")); // Output: 17

- Useful when searching for words from the end of a string.

14. startsWith(value) - Check Start of String

The startsWith() method checks if a string starts with a specific value.

Example:

let message = "Hello World";

console.log(message.startsWith("Hello")); // Output: true

- Used to verify file extensions or domain names.

15. endsWith(value) - Check End of String

The endsWith() method checks if a string ends with a specific value.

Example:

let filename = "document.pdf";

console.log(filename.endsWith(".pdf")); // Output: true

- Useful for checking file types before uploading.

JavaScript Objects

An object in JavaScript is a data structure used to store related data collections. It stores data as key-value pairs, where each key is a unique identifier for the associated value. Objects are dynamic, which means the properties can be added, modified, or deleted at runtime.

There are two primary ways to create an object in JavaScript: Object Literal and Object Constructor.

1. Creation Using Object Literal

The object literal syntax allows you to define and initialize an object with curly braces {}, setting properties as key-value pairs.

```
let obj = {
   name: "Sourav",
   age: 23,
   job: "Developer"
};
console.log(obj);
```

Output

{ name: 'Sourav', age: 23, job: 'Developer' }

2.**Creation Using new Object() Constructor**

```
let obj = new Object();
obj.name= "Sourav",
obj.age= 23,
obj.job= "Developer"
console.log(obj);
```

Output

{ name: 'Sourav', age: 23, job: 'Developer' }

Basic Operations on JavaScript Objects

1. Accessing Object Properties

You can access an object's properties using either dot notation or bracket notation

```
let obj = { name: "Sourav", age: 23 };

// Using Dot Notation
console.log(obj.name);
// Using Bracket Notation
console.log(obj["age"]);
```

Output

Sourav

23

2. Modifying Object Properties

Properties in an object can be modified by reassigning their values.

```
let obj = { name: "Sourav", age: 22 };
console.log(obj);
obj.age = 23;
console.log(obj);
```

Output

{ name: 'Sourav', age: 22 }

{ name: 'Sourav', age: 23 }

3. Adding Properties to an Object

You can dynamically add new properties to an object using dot or bracket notation.

```
let obj = { model: "Tesla" };
obj.color = "Red";
console.log(obj);
```

Output

{ model: 'Tesla', color: 'Red' }

4. Removing Properties from an Object

The delete operator removes properties from an object.

```
let obj = { model: "Tesla", color: "Red" };
delete obj.color;
console.log(obj);
```

Output

```
{ model: 'Tesla' }
```

5. Iterating Through Object Properties

Use for…in loop to iterate through the properties of an object.

```
let obj = { name: "Sourav", age: 23 };
for (let key in obj) {
    console.log(key + ": " + obj[key]);
}
```

Output

```
name: Sourav
age: 23
```

6. Object Length

You can find the number of properties in an object using Object.keys().

```
let obj = { name: "Sourav", age: 23 };
console.log(Object.keys(obj).length);
```

Output

2

Recognizing a JavaScript Object

To check if a value is an object, use typeof and verify it’s not null.

let obj = { name: "Sourav" };

console.log(typeof obj === "object" && obj !== null);

Output

True

Methods and Objects

1. Defining an Object with Methods

An object in JavaScript is a collection of properties and methods. A method is a function inside an object. The students object contains properties (name, age) and a method (greet). The this keyword refers to the current object and is used to access its properties inside methods.

```
const students = {
    name: "Deepika",
    age: 25,
    greet: function() {
        return `Hello, my name is ${this.name}.`;
    }
};
console.log(students.greet());
```

Output: Hello, my name is Deepika.

2. Using this Keyword in Object Methods

The this keyword refers to the object on which the method is called. It allows dynamic access to the object's properties. In the car1 object, this.brand and this.model retrieve values from the object itself.

```
const car1 = {
    brand: "Toyota",
    model: "Camry",
    getCarInfo: function() {
        return `Car: ${this.brand} ${this.model}`;
    }
};
console.log(car1.getCarInfo());
```

Output: Car: Toyota Camry

3. Shorthand Method Notation in ES6

ES6 introduced shorthand syntax for defining methods inside objects. Instead of writing greet: function() {}, we can use greet() {} for better readability.

```
const user1 = {
    name: "Alice",
    greet() {
        return `Hi, I'm ${this.name}`;
    }
};
console.log(user1.greet());
```

Output: Hi, I'm Alice

4. Method Chaining

Method chaining allows multiple method calls on the same object in a single statement. It works by returning the object (this) at the end of each method.

```
const calculator = {
    value: 0,
    add(num) {
        this.value += num;
        return this;
    },
    subtract(num) {
        this.value -= num;
        return this;
    },
    getResult() {
        console.log(this.value);
        return this;
    }
};
calculator.add(10).subtract(5).getResult();
```

Output: 5

5. Preventing Object Modification with Object.freeze()

The Object.freeze() method prevents adding, removing, or modifying properties of an object. The person object remains unchanged even after attempting modifications.

const person = { name: "Deepika", age: 25 };

Object.freeze(person);

person.age = 30; // No effect

person.city = "Mysore"; // No effect

console.log(person);

Output: { name: "Deepika", age: 25 }

6. Allowing Modifications but Preventing New Properties with Object.seal()

The Object.seal() method allows modifying existing properties but prevents adding or deleting properties. Here, cars.model can be changed, but cars.year cannot be added.

const cars = { brand: "Toyota", model: "2022" };

Object.seal(cars);

cars.model = "Corolla"; // Allowed

cars.year = 2023; // Not allowed

delete cars.brand; // Not allowed

console.log(cars);

Output: { brand: "Toyota", model: "Corolla" }

7. Checking Property Existence using hasOwnProperty()

The hasOwnProperty() method checks if a property exists in an object. It returns true if the property exists and false otherwise.

```
const user = { name: "John", age: 30 };

console.log(user.hasOwnProperty("name")); // true

console.log(user.hasOwnProperty("gender")); // false
```

Output:
true
false

8. Comparing Objects using Object.is()

The Object.is() method is similar to === but correctly handles special cases like NaN. It returns true for values that are strictly equal.

```
console.log(Object.is(25, 25)); // true

console.log(Object.is(NaN, NaN)); // true

console.log(Object.is({}, {})); // false (Different references)
```

Output:
true
true
false

9. Converting Arrays to Objects using Object.fromEntries()

The Object.fromEntries() method transforms a list of key-value pairs (array) into an object. It is the reverse of Object.entries().

```
const entries = [["title", "JavaScript Guide"], ["author", "Deepika"]];

const bookObj = Object.fromEntries(entries);

console.log(bookObj);
```

Output: { title: "JavaScript Guide", author: "Deepika" }

Decision Making in JavaScript

1.if Statement

Executes a block of code only if the given condition is true.

```
let Age = 18;
if (Age >= 18) {
    console.log("You are eligible to vote.");
}
```

Output: "You are eligible to vote."

2. if-else Statement

Executes one block of code if the condition is true, otherwise executes the else block.

```
let num = 10;
if (num % 2 == 0) {
    console.log("Even number");
} else {
    console.log("Odd number");
}
```

Output: "Even number"

3. if-else-if Statement

Used when there are multiple conditions to check.

```
let marks = 85;
if (marks >= 90) {
    console.log("Grade: A+");
} else if (marks >= 75) {
    console.log("Grade: A");
} else {
    console.log("Grade: B");
}
```

Output: "Grade: A"

Ternary Operator (? :)

What is a Ternary Operator

The ternary operator is a conditional operator that takes three operands: **a condition, a value** to be returned if the condition is **true**, and a value to be returned if the condition is **false**. It evaluates the condition and returns one of the two specified values based on whether the condition is true or false.

Syntax of Ternary Operator:

The Ternary operator can be in the form

variable = Expression1 ***?*** *Expression2* ***:*** *Expression3****;***

ex

```
let age = 20;
let result = (age >= 18) ? "Adult" : "Minor";
console.log(result);
```

Output: "Adult"

It can be visualized into an if-else statement as:

```
if(Expression1)
{
   variable = Expression2;
}
else
{
   variable = Expression3;
}
```

Working of Ternary Operator :

The Ternary operator can be in the form

variable = Expression1 **?** Expression2 **:** Expression3**;**

The working of the Ternary operator is as follows:

- **Step 1: Expression1** is the condition to be evaluated.
- **Step 2A:** If the condition(**Expression1**) is True then **Expression2** will be executed.
- **Step 2B**: If the condition(**Expression1**) is false then **Expression3** will be executed.
- **Step 3:** Results will be returned

switch Statement

The **JavaScript switch statement** evaluates an expression and executes a block of code based on matching cases. It provides an alternative to long if-else chains, improving readability and maintainability, especially when handling multiple conditional branches.

Ex

```
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

How Switch Statement Works

- **Evaluation**: The expression inside the switch the statement is evaluated once.
- **Comparison**: The value of the expression is compared with each case label (using strict equality ===).
- **Execution**: If a match is found, the corresponding code block following the matching case the label is executed. If no match is found, the execution jumps to the default case (if present) or continues with the next statement after the switch block.
- **Break Statement**: After executing a code block, the break statement terminates the switch statement, preventing execution from falling through to subsequent cases. If break is omitted, execution will continue to the next case (known as "fall-through").
- **Default Case**: The default case is optional. If no match is found, the code block under default is executed.

JavaScript Loops

Loops in JavaScript are used to reduce repetitive tasks by repeatedly executing a block of code as long as a specified condition is true. This makes code more concise and efficient.

Suppose we want to print 'Hello World' five times. Instead of manually writing the print statement repeatedly, we can use a loop to automate the task and execute it based on the given condition.

```
for (let i = 0; i < 5; i++) {
    console.log("Hello World!");
}
```

Output

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Let's now discuss the different types of loops available in JavaScript

1. JavaScript for Loop

The for loop repeats a block of code a specific number of times. It contains initialization, condition, and increment/decrement in one line.

Syntax

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

```
for (let i = 1; i <= 3; i++) {
    console.log("Count:", i);
}
```

Output

Count: 1

Count: 2

Count: 3

In this example

- Initializes the counter variable (let i = 1).
- Tests the condition (i <= 3); runs while true.
- Executes the loop body and increments the counter (i++).

2. JavaScript while Loop

The while loop executes as long as the condition is true. It can be thought of as a repeating if statement.

Syntax

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

```
let i = 0;
while (i < 3) {
   console.log("Number:", i);
   i++;
}
```


Output

Number: 0

Number: 1

Number: 2

In this example

- Initializes the variable (let i = 0).
- Runs the loop body while the condition (i < 3) is true.
- Increments the counter after each iteration (i++).

3. JavaScript do-while Loop

The do-while loop is similar to while loop except it executes the code block at least once before checking the condition.

Syntax

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

```
let i = 0;

do {

   console.log("Iteration:", i);

   i++;

} while (i < 3);
```

Output

Iteration: 0

Iteration: 1

Iteration: 2

In this example:

- Executes the code block first.
- Checks the condition (i < 3) after each iteration.

4. JavaScript for-in Loop

The for…in loop is used to iterate over the properties of an object. It only iterate over keys of an object which have their enumerable property set to “true”.

Syntax

```
for (let key in object) {
   // Code to execute
}

const obj = { name: "Ashish", age: 25 };
```

```
for (let key in obj) {
    console.log(key, ":", obj[key]);
}
```

Output

name : Ashish

age : 25

In this example:

- Iterates over the keys of the person object.
- Accesses both keys and values.

5. JavaScript for-of Loop

The for…of loop is used to iterate over iterable objects like arrays, strings, or sets. It directly iterate the value and has more concise syntax than for loop.

Syntax

```
for (let value of iterable) {
    // Code to execute
}
```

```
let a = [1, 2, 3, 4, 5];
for (let val of a) {
    console.log(val);
}
```

Output

1

2

3

4

Choosing the Right Loop

- Use for loop when the number of iterations is known.
- Use while loop when the condition depends on dynamic factors.
- Use do-while loop to ensure the block executes at least once.
- Use for…in loop to iterate over object properties.
- Use for…of loop for iterating through iterable objects.

Break and Continue in loops

In JavaScript, break and continue are used to control loop execution. break immediately terminates a loop when a condition is met, while continue Skips the current iteration and proceeds to the next loop iteration. Additionally, JavaScript supports labels, which can be used with break and continue to control nested loops efficiently.

The break Statement

The break statement is used to exit a loop when a certain condition is satisfied. It is commonly used when searching for a specific value in an array or when an early exit from a loop is required.

Syntax

break;

Using break in a for loop

```
for (let i = 0; i < 10; i++) {
   if (i === 5) {
      console.log("Breaking the loop at", i);
      break;
   }
   console.log(i);
}
```

Output

0

1

2

3

4

Breaking the loop at 5

In this example

- The loop starts from i = 0 and iterates up to i < 10.
- When i reaches 5, the break statement executes, terminating the loop immediately.
- Numbers 0 to 4 are printed before the loop stops.

Using break in a while Loop

```
let n = 1;
while (n <= 10) {
   if (n === 6) {
      console.log("Loop stopped at", n);
      break;
   }
   console.log(n);
   n++;
}
```

Output

1

2

3

4

5

Loop stopped at 6

In this example

- The loop starts from n = 1 and runs as long as n <= 10.
- When n reaches 6, the break statement is executed, stopping the loop.
- The numbers 1 to 5 are printed before the loop terminates.

The Continue statement

The continue statement skips the current iteration of the loop and moves to the next iteration without terminating the loop.

Syntax

continue;

Using continue in a for Loop

```
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        continue; // Skips even numbers
    }
    console.log(i);
}
```

Output

1

3

5

7

9

In this example

- The loop iterates from i = 0 to i < 10.
- If i is an even number (i % 2 === 0), the continue statement is executed, skipping that iteration.
- Only odd numbers are printed.

Using continue in a while Loop

```
let i = 0;
while (i < 10) {
   i++;
   if (i % 3 === 0) {
      continue; // Skips multiples of 3
   }
   console.log(i);
}
```

Output

1

2

4

5

7

8

10

In this example

- The loop starts from i = 0 and runs while i < 10.
- If i is a multiple of 3, the continue statement skips that iteration.
- Numbers 3, 6, and 9 are not printed.

Functions in JavaScript

Functions in JavaScript are reusable blocks of code designed to perform specific tasks. They allow you to organize, reuse, and modularize code. It can take inputs, perform actions, and return outputs.

```
function sum(x, y) {
   return x + y;
}
console.log(sum(6, 9));
```

Output

15

Function Syntax and Working

A function definition is sometimes also termed a function declaration or function statement. Below are the rules for creating a function in JavaScript:

- Begin with the keyword **function** followed by,
- A user-defined function name (In the above example, the name is **sum**)
- A list of parameters enclosed within parentheses and separated by commas (In the above example, parameters are **x** and **y**)
- A list of statements composing the body of the function enclosed within curly braces {} (In the above example, the statement is “return x + y”).

Return Statement

In some situations, we want to return some values from a function after performing some operations. In such cases, we make use of the return. This is an optional statement. In the above function, “sum()” returns the sum of two as a result.

Function Parameters

Parameters are input passed to a function. In the above example, sum() takes two parameters, x and y.

Calling Functions

After defining a function, the next step is to call them to make use of the function. We can call a function by using the function name separated by the value of parameters enclosed between the parenthesis.

```
// Function Definition
function welcomeMsg(name) {
   return ("Hello " + name + " welcome ");
}

let nameVal = "User";

// calling the function
console.log(welcomeMsg(nameVal));
```

Output

Hello User welcome

Why Functions?

- Functions can be used multiple times, reducing redundancy.
- Break down complex problems into manageable pieces.
- Manage complexity by hiding implementation details.
- Can call themselves to solve problems recursively.

Function Invocation

The function code you have written will be executed whenever it is called.

- Triggered by an event (e.g., a button click by a user).
- When explicitly called from JavaScript code.
- Automatically executed, such as in self-invoking functions.

Function Expression

It is similar to a function declaration without the function name. Function expressions can be stored in a variable assignment.

Syntax:

```
let val= function(paramA, paramB) {
   // Set of statements
}
```

Ex,

```
const mul = function (x, y) {
   return x * y;
};
console.log(mul(4, 5));
```

Output

20

Arrow Functions

Arrow functions are a concise syntax for writing functions, introduced in ES6, and they do not bind their own this context.

Syntax:

let function_name = (argument1, argument2 ,..) => expression

Ex,

```
const a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];
const a2 = a.map(function (s) {
   return s.length;
});
console.log("Normal way ", a2);

//Arrow function
const a3 = a.map((s) => s.length);
console.log("Using Arrow Function ", a3);
```

Output

Normal way [8, 6, 7, 9]

Using Arrow Function [8, 6, 7, 9]

Callback Functions

A callback function is passed as an argument to another function and is executed after the completion of that function.

```
function num(n, callback) {
   return callback(n);
}
const double = (n) => n * 2;
console.log(num(5, double));
```

Output

10

Anonymous Functions

Anonymous functions are functions without a name. They are often used as arguments to other functions.

```
let show = function() {
   console.log('Anonymous function');
};
show();
```

Nested Functions

Functions defined within other functions are called nested functions. They have access to the variables of their parent function.

```
function outerFun(a) {
   function innerFun(b) {
      return a + b;
   }
   return innerFun;
}
const addTen = outerFun(10);
console.log(addTen(5));
```

Output

15

Key Characteristics of Functions

- **Parameters and Arguments:** Functions can accept parameters (placeholders) and be called with arguments (values).
- **Return Values:** Functions can return a value using the return keyword.
- **Default Parameters:** Default values can be assigned to function parameters.

Advantages of Functions in JavaScript

- **Reusability:** Write code once and use it multiple times.
- **Modularity:** Break complex problems into smaller, manageable pieces.
- **Improved Readability:** Functions make code easier to understand.
- **Maintainability:** Changes can be made in one place without affecting the entire codebase.