Module-01

Basic Java Script Instructions

Instructions & Statements in JavaScript

Java Script is a programming language used primarily for creating interactive websites. The instructions given to the browser through JavaScript code are executed in the form of **statements**.

Statements are the building blocks of any Java Script program. Understanding how this work will help you in writing efficient and effective JavaScript code.

1. Instructions in Java Script

- **Definition**: In Java Script, instructions are written commands that the browser or environment follows to perform a specific task.
- **Purpose**: These instructions enable the browser to manipulate data, interact with users, and control the flow of the program.
- **Common Instruction Types**:
    - **Variable declarations**: Used to define variables that store data.
    - **Function calls**: Instructions that invoke a function to execute specific actions.
    - **Conditional checks**: Direct the flow based on certain conditions (e.g., if, else).
    - **Loops**: Used to repeat code until a condition is met.

2. Statements in JavaScript

- **Definition**: A statement is a complete line of code that performs an action. In JavaScript, each statement usually ends with a semicolon (;).
- **Purpose**: Statements represent the instructions given to the browser that perform specific tasks.
- **Structure**: Each statement typically contains an expression followed by an action to be performed (e.g., assignment, function call).

3. Key Types of Statements

a. Variable Declaration Statements

- **Purpose**: Declare variables that hold data values.
- **Syntax**:

```
let x = 10; // Variable declaration with assignment

const name="John";//Declaring a constant variable
```

- **Examples**:
    - Declaring a variable to store a number:

```
let age = 30;
```

    - Declaring a string:

```
let userName = "Alice";
```

- **Explanation**: Variables are essential for storing data that can be used and manipulated later in the program.

b. Assignment Statements

- **Purpose**: Assign a value to a previously declared variable.
- **Syntax**:

```
x=20;//Reassigning the value of` x`
```

- **Examples**:

```
userName="Bob";//Re assign the value of `userName` age

= 35; // Updating the value of age
```

- **Explanation**: The assignment statement changes the value stored in a variable.

c. Function Declaration and Call Statements

- **Purpose**: Define reusable code blocks (functions) and invoke them to execute their logic.
- **Syntax**:

```
function

greet(){

console.log("Hello,World!");//Function body with a statement

}
```

greet();//Calling the function to execute the code inside it

Examples:

javascript

CopyEdit

```
function add(x, y) {
return x + y; // Returns the sum of `x` and `y`
}
let result = add(5, 10); // Calling the `add` function with arguments
console.log(result); // Output: 15
```

Explanation: Functions encapsulate reusable code, and the return statement can be used to send values back

Conditional Statements (If, Else, Else If)

- **Purpose**: Control the flow of the program based on conditions.
- **Syntax:**

```
if (x > 10) {
console.log("x is greater than 10");
} else {
console.log("x is not greater than 10");
}
```

- **Examples:**

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

- **Explanation**: Conditional statements allow you to make decisions and execute different blocks of code based on conditions.

Looping Statements (For, While, Do-While)

Purpose: Execute a block of code repeatedly until a condition is met.

• **Syntax:**

```
for (let i = 0; i < 5; i++) {
console.log(i); // Logs numbers from 0 to 4
}
```

• **Examples**:

```
let numbers = [1, 2, 3, 4];
for (let i = 0; i < numbers.length; i++) {
console.log(numbers[i]); // Prints each number in the array
 }
```

- **Explanation**: Loops help reduce code repetition, making it easier to perform tasks such as iterating through arrays or lists.

Return Statement

• **Purpose**: Return a value from a function to the caller.

• **Syntax:**

```
function sum(x, y) {
return x + y; // Returns the result of the addition
}
```

• **Examples**:

```
function square(number) {
return number * number; // Returns the square of a number
 }
 let result = square(5); // Result will be 25
```

- **Explanation:** The return statement allows a function to output a value, which can then be used in the program.

Expression Statements

- **Purpose**: Execute expressions that result in a value. These expressions are then treated as statements when executed.
- **Syntax**: x = x + 1; // An expression that increments the value of `x`

- **Examples**: let total = 5 * 4; // Expression that calculates multiplication
- **Explanation**: Expressions evaluate to a value, and when placed in a statement, they execute their logic.

JavaScript Execution Flow

- **Execution Order**: JavaScript statements are executed in a top-down order unless there are control structures like loops or conditionals that alter the flow.
- **Asynchronous Execution**: JavaScript can also handle asynchronous operations (e.g., API calls) through callback functions, Promises, and async/await syntax.

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

Example of Full JavaScript Code (Combined Use of Statements)

```
let userName = "John"; // Variable declaration
let age = 30; // Variable declaration

// Function Declaration
 function greetUser(name)
{ console.log("Hello, " + name);
}

greetUser(userName); // Function Call
 if (age > 18) { // Conditional statement
console.log("You are an adult.");
} else {
 console.log("You are a minor.");
}

 let numbers = [1, 2, 3, 4, 5];
f or (let i = 0; i < numbers.length; i++) { // Looping statement
console.log(numbers[i]); // Logs numbers 1 to 5 }
```

- **Output:**

Hello, John

You are an adult.

1

2

3

4

5

Comments in JavaScript

Comments are an essential part of any codebase, providing explanations, notes, or clarifications for both the developer and anyone reading the code.

- Comments help make the code more readable and maintainable, especially for larger projects.
- In JavaScript, there are two types of comments: **single-line comments and multi-line comments.**
- Single-Line Comment
- A single-line comment is used to add brief notes or explanations for a specific line of code.
- The comment starts with two forward slashes (//). Everything following these slashes on that line will be ignored by the JavaScript engine.

Syntax:

// This is a single-line comment

• Usage:

- Single-line comments are generally used for:

▪ Brief explanations about specific lines of code.

▪ Temporarily disabling a line of code.

▪ Adding simple notes for debugging or testing.

- **Example:**

let x = 10; // Declaring variable x with value 10

console.log(x); // This will log the value of x to the console

- In this example, the comments help explain the purpose of the code on each line, making it easier for someone to understand the logic.

- **Multi-Line Comment**
- A multi-line comment is used when a comment spans more than one line.
- It starts with /* and ends with */. Everything between these markers will be considered part of the comment and ignored by JavaScript.

• **Syntax**: /* This is a multi-line comment that spans multiple lines */

• **Usage**:

o Multi-line comments are ideal when:

▪ The explanation or note requires multiple lines of text.

▪ Large sections of code need to be temporarily disabled for testing or debugging purposes.

- Example: /* This function is used to calculate the area of a rectangle. The formula used is length * width. Length is provided as the first parameter and width as the second. */

```
function calculateArea(length, width) {
return length * width; // Return the product of length and width
}
```

In this example, the multi-line comment provides a detailed explanation of the function's purpose and how it works.

- **When to Use Comments**
- **Clarify Complex Code**: Use comments to explain parts of the code that may be difficult to understand at first glance.

• **Documentation**: Write comments to document the purpose and expected behavior of functions, classes, and sections of code.

• **To Do Notes**: Comments can be used to indicate tasks that need to be completed in the future or code that is temporarily left incomplete.

- **Disable Code**: During testing or debugging, you might use comments to disable certain lines of code temporarily without deleting them.
- **Best Practices for Comments**

• **Be concise:** Comments should be brief but informative. Avoid stating the obvious (e.g., x = 10; // Assigning 10 to x).

- **Use comments to explain "why" , not "what":** The code itself should show what is happening. Comments should explain why something is done in a specific way, especially when it's not immediately obvious.
- **Keep comments up to date:** Outdated comments can be misleading. Always update or remove comments if the code changes
- **Example of Proper Comment Usage**

```
// This function takes two parameters: length and width
// It calculates the area of the rectangle
function calculateArea(length, width) {
return length * width; // Multiply length by width to get area
 }
// Multi-line comment to explain a more complex function
/* This function calculates the total cost of items in a shopping cart. It takes an array of item objects, where each object contains the price of the item. The function returns the total cost of all items. */

 function calculateTotalCost(cartItems) {
let totalCost = 0; // Initialize total cost to 0
 for (let i = 0; i < cartItems.length; i++) {
totalCost += cartItems[i].price; // Add the price of each item to totalCost
}
return totalCost; // Return the total cost
}
```

In this example, single-line comments explain specific actions, while a multi-line comment provides a detailed description of the function's purpose.

Variables in JavaScript

Variables are fundamental to any programming language, and in JavaScript, they are used to store data that can be referenced and manipulated throughout the program.

- Variables allow you to store values like numbers, strings, objects, and more. In JavaScript, you can declare variables using three different keywords**: var, let, and const.**

Declaration of Variables

- To declare a variable in JavaScript, we use one of the three keywords: var, let, or const.

• **Syntax:**

let variableName = value; // Declares a block-scoped variable

const variableName = value; // Declares a constant variable

var variableName = value; // Declares a function-scoped variable

Example:

let name = "John"; // Block-scoped variable

const pi = 3.14; // Constant variable

var age = 25; // Function-scoped variable

var (Function-Scoped Variable)

• var is the traditional way of declaring variables in JavaScript.

• Scope: A variable declared with var is function-scoped, meaning it is accessible within the function where it is declared. If declared outside of any function, the variable is globally scoped.

- Hoisting: Variables declared with var are "hoisted" to the top of their scope, meaning the declaration is moved to the top during execution, but the initialization stays in place.

Example: var name = "Alice"; // Declared with var

```
function greet() {
var message = "Hello, " + name; // Accessible inside the function
console.log(message);
}
greet();
console.log(message); // Error: message is not defined outside the function
```

- In this example, message is accessible within the function greet() but not outside.

let (Block-Scoped Variable)

let is a modern way to declare variables that **are block-scoped.**

- **Scope**: A variable declared with let is only accessible within the block (denoted by curly braces {}) where it is declared, such as inside loops, conditionals, or functions.
- **Hoisting**: Like var, variables declared with let are also hoisted to the top of their block, but they cannot be accessed until after the declaration line. This prevents issues caused by hoisting in var.
- **Example:**

```
let city = "Paris"; // Declared with let
if (true) {
let city = "London"; // This city is scoped to the if block
console.log(city); // Output: London
}
console.log(city); // Output: Paris (original city remains unchanged)
```

In this example, the city variable inside the if block does not affect the city variable outside of it, because the variable is **block-scoped.**

const (Constant Variable)

- const is used to declare constant variables whose values cannot be reassigned after they are initialized.
- Scope: Like let, const is also **block-scoped.**
- **Reassignment**: A variable declared with const cannot be reassigned, meaning its value remains constant. However, if the value is an object or an array, its properties or elements can still be modified.

Example:

```
const country = "USA"; // Constant variable
country = "Canada"; // Error: Assignment to constant variable.
const numbers = [1, 2, 3];
numbers.push(4); // Allowed: Modifying the array's contents
console.log(numbers); // Output: [1, 2, 3, 4]
```

In this example, **the reassignment of the country variable results in an error, but modifying the contents of the numbers array is allowed because arrays are reference types.**

Scope in JavaScript

Scope refers to the visibility and lifetime of a variable. It determines where the variable can be accessed and modified.

- **Global Scope:** If a variable is declared outside any function or block, it has a global scope and can be accessed anywhere in the program.
- **Function Scope**: Variables declared with var inside a function are only accessible within that function.
- **Block Scope:** Variables declared with let or const inside a block (e.g., if statements, loops) are only accessible within that block.

- **Hoisting in JavaScript**
- Hoisting refers to the behavior where variable declarations are moved to the top of their scope during the compilation phase, before the code is executed.
- **With var:** The declaration is hoisted, but the initialization remains in place.

```
console.log(name); // undefined
var name = "Alice";
console.log(name); // Alice
```

- **With let and const**: The declaration is hoisted, but not the initialization. Accessing the variable before initialization results in a ReferenceError.

```
console.log(name); // ReferenceError: Cannot access 'name' before initialization
let name = “Bob"
```

Data Types in JavaScript

- In JavaScript, data types are categorized into two main groups:
- **Primitive Types** and **Non-Primitive** Types.
- Each type serves a specific purpose, and understanding these data types is fundamental to writing effective JavaScript code.
- **Primitive Data Types**
- Primitive types are the most basic types in JavaScript. They represent single values and are immutable (cannot be changed after creation).
- These types include:

1**. String:** A sequence of characters used to represent text.

o Example: let greeting = "Hello, World!";

2**. Number**: Represents both integer and floating-point numbers.

o Example: let score = 95; // Integer let price = 19.99; // Floating-point number

3. **Boolean**: Represents a logical value of either true or false.

o Example: let isAvailable = true; // Boolean value

let isValid = false; // Boolean value

4. Undefined: A variable that has been declared but not yet assigned a value has the value undefined.

- o **Example**: let message; // Undefined since no value is assigned yet

5. **Null:** Represents the intentional absence of any value or object. It is a primitive value that is not the same as undefined.

- o **Example:** let person = null; // Represents no value or object

6. **Symbol:** A unique and immutable value primarily used to add unique property keys to objects.

- **o Example**: let uniqueSymbol = Symbol("id"); // Symbol with a description

7. **BigInt**: A type that can represent integers with arbitrary precision. It is useful for working with large numbers that exceed the Number type's limit.

- **o Example:**

let largeNumber = 1234567890123456789012345678901234567890n; // BigInt

- Note: Primitive data types are compared by their values. This means that two variables with the same value of a primitive type will be considered equal.

Non-Primitive Data Types

- Non-primitive data types, also called reference types, are more complex than primitive types. They store references to the data rather than the actual data. Modifying an object or array will affect all references to it.

1. **Object**: Represents a collection of key-value pairs. Keys are strings (or Symbols), and values can be any data type (including other objects).

o **Example:** let person = { name: "John", age: 30 }; // Object with properties

2. **Array**: A special type of object used to store a collection of elements (values), usually ordered.

o Example:

let numbers = [1, 2, 3, 4, 5]; // Array of numbers

let fruits = ["apple", "banana", "cherry"]; // Array of strings

3. **Function**: A function is a block of code designed to perform a particular task, and it is also considered a non-primitive type.

o **Example**: function greet(name) {

return "Hello, " + name;

```
}
console.log(greet("Alice")); // Output: Hello, Alice
```

4**. Date**: Used to represent date and time.

- **Example:**

```
let currentDate = new Date(); // Creates a new Date object
console.log(currentDate); // Outputs the current date and time
```

- **Note:** Non-primitive data types are compared by reference, meaning if two objects or arrays refer to the same location in memory, they are considered equal.

Understanding Type Behavior

Primitive types are immutable: Once you assign a primitive value to a variable, that value cannot be changed.

- For example, if you assign a string to a variable, you cannot change the characters of that string without creating a new string.

```
let name = "Alice";
name[0] = "B"; // This does not change the string 'name'
console.log(name); // Output: Alice
```

- **Non-primitive types are mutable**: You can modify the contents of an object or array after they have been created.

```
let person = { name: "John", age: 25 };
person.age = 26; // The age property is updated
console.log(person.age); // Output: 26
```

Feature	Primitive Types	Non-Primitive Types
1. Definition	Represents a single value.	Represents a reference to an object or collection of values.
2. Immutability	Immutable: Once created, their value cannot be changed.	Mutable: Their contents can be modified.
3. Storage	Stored by value.	Stored by reference (memory location).
4. Types	Includes `string`, `number`, `boolean`, `undefined`, `null`, `symbol`, `bigint`.	Includes `object`, `array`, `function`, `date`.
5. Comparison	Compared by value: Two variables with the same value are considered equal.	Compared by reference: Two objects/arrays are equal only if they refer to the same location in memory.
6. Memory Efficiency	More memory-efficient, as they only store a simple value.	Less memory-efficient, as they store references to complex structures.
7. Assignment	Direct assignment copies the value.	Assignment copies the reference, not the actual object/array.
8. Example	`let name = "John";`	`let person = { name: "John" };`
9. Type Conversion	Implicit type coercion is common (automatic conversion).	Explicit type conversion may be required for objects and arrays.
10. Use Cases	Best for storing simple data (e.g., numbers, strings, booleans).	Best for storing complex data like collections (arrays) and objects.

Arrays in JavaScript

Arrays in JavaScript are used to store collections of elements.

- They allow you to group multiple values into a single variable and access each element using an index.
- **Key Features of Arrays**

1**. Indexed Elements**: Each element in an array is assigned a numeric index starting from 0.

o **Example**: let colors = ["red", "green", "blue"]; console.log(colors[0]); // Output: red

2. **Dynamic Sizing:** Arrays in JavaScript can grow or shrink in size dynamically; you don't need to declare a fixed size.

o **Example:** let numbers = [1, 2, 3]; numbers.push(4); // Adds 4 to the array console.log(numbers); // Output: [1, 2, 3, 4]

3. **Heterogeneous Elements**: Arrays can hold values of different data types.

o **Example:**

let mixed = [42, "hello", true];

console.log(mixed); // Output: [42, "hello", true]

4. **Mutable**: You can modify elements in an array directly.

o **Example**:

let fruits = ["apple", "banana", "cherry"];

fruits[1] = "orange"; // Replaces "banana" with "orange"

console.log(fruits); // Output: ["apple", "orange", "cherry"]

Common Array Methods

- JavaScript provides several built-in methods to manipulate arrays.

1. Adding Elements:

- **push():** Adds an element to the end of the array.

let arr = [1, 2]; arr.push(3);

console.log(arr); // Output: [1, 2, 3]

- unshift(): Adds an element to the beginning of the array.

arr.unshift(0)

console.log(arr); // Output: [0, 1, 2, 3]

2. **Removing Elements:**

o pop(): Removes the last element of the array.

- arr.pop(); console.log(arr); // Output: [0, 1, 2]
- o shift(): Removes the first element of the array. arr.shift(); console.log(arr); // Output: [1, 2]

3. Finding Elements:

- indexOf(): Returns the index of the first occurrence of an element. Returns -1 if not found.

let fruits = ["apple", "banana", "cherry"];

console.log(fruits.indexOf("banana")); // Output: 1

4. **Iterating Through Arrays:**

o Using for loop:

```
for (let i = 0; i < fruits.length; i++){
console.log(fruits[i]);
}
```

o Using forEach():

```
fruits.forEach((fruit) => console.log(fruit));
```

Copying Elements

- slice(): Creates a shallow copy of a portion of an array.
- let newFruits = fruits.slice(0, 2); console.log(newFruits); // Output: ["apple", "banana"]
- **Transforming Elements:**
- map(): Creates a new array by transforming each element of the original array.

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

Filtering Elements

- filter(): Creates a new array containing elements that pass a test.

```
let nums = [1, 2, 3, 4, 5];
 let even = nums.filter((num) => num % 2 === 0);
console.log(even); // Output: [2, 4]
```

- **Accessing Array Elements**
- Using Index: Use the bracket notation with the element's index.

```
let animals = ["dog", "cat", "elephant"];
console.log(animals[1]); // Output: cat
```

- **Negative Indexing**: JavaScript does not support negative indexing, but you can achieve it using custom logic.

```
let last = animals[animals.length - 1]; // Access the last element
console.log(last); // Output: elephant
```

- **Multi-Dimensional Arrays**
- JavaScript supports arrays within arrays, known as multi-dimensional arrays.

• **Example:**

```
let matrix = [
[1, 2, 3],
[4, 5, 6],
[7, 8, 9],
];
console.log(matrix[1][2]); // Output: 6 (2nd row, 3rd column)
```

Strings in JavaScript

- Strings are sequences of characters used to store and manipulate text. They are enclosed in single quotes ('), double quotes ("), or backticks (`) for template literals.

Key Features of Strings

1**. String Declaration**:

- Using single or double quotes:

```
let name = "John";
let city = 'New York';
```

- **Using backticks for template literals:**

```
let greeting = `Hello, ${name} !`;
console.log(greeting); // Output: Hello, John!
```

2. String Properties:

- length: Returns the number of characters in a string.

```
let message = "JavaScript";
console.log(message.length); // Output: 10
```

3. Common String Methods:

toUpperCase(): Converts the string to uppercase.

```
console.log("hello".toUpperCase()); // Output: HELLO
```

toLowerCase(): Converts the string to lowercase.

```
console.log("WORLD".toLowerCase());//Output: world
```

- **slice**(start, end): Extracts a section of the string. let text = "JavaScript";

```
console.log(text.slice(0, 5)); // Output: Java
```

- replace(search, replaceWith): Replaces part of the string.

```
let sentence = "Hello, World!";
console.log(sentence.replace("World", "JavaScript")); // Output: Hello, JavaScript!
```

Additional concepts of functions

1. Function Declaration:

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

2. Parameters and Arguments:

- **Parameters**: Variables defined in the function declaration.
- **Arguments**: Values passed to the function when it is called.

3. Function Expression:

- **Assigning a function to a variable:**

```
const add = function (a, b) {
return a + b;
};
console.log(add(2, 3)); // Output: 5
```

4. Arrow Functions:

- **Shorter syntax for writing functions:**

```
const multiply = (x, y) => x * y;
console.log(multiply(4, 5)); // Output: 20
```

5. **Default Parameters**:

- Setting default values for parameters:

```
function greet(name = "Guest") {
return `Hello, ${name} !`; }
console.log(greet()); // Output: Hello, Guest!
```

Methods in JavaScript

- String Methods
- Array methods
- **Object Methods:**

- Custom methods can be added to objects:

```
let car = { brand: "Toyota",
start: function () {
return `${this.brand} is starting!`;
},
};
console.log(car.start()); // Output: Toyota is starting!
```

Aspect	Functions	Methods
Definition	A reusable block of code that can be called independently.	A function associated with an object. Operates on its data.
Invocation	Called by name directly: `functionName()`.	Called using the object: `object.methodName()`.
Scope	General purpose, not tied to any specific object.	Operates on and is tied to the object it belongs to.
Declaration	Defined using `function` or `arrow function` syntax.	Defined as a property of an object.
Data Access	Cannot access object properties directly unless passed.	Accesses properties of its associated object using `this`.
Usage Context	Used for general calculations or logic.	Used to perform actions specific to an object.
Example		
	``javascript	```javascript
	function greet() {	let car = {
	return "Hello!";	brand: "Toyota",
	}	start: function () {
	console.log(greet()); // Output: Hello!	return `${this.brand} starts!`;
	``	},
		};
		console.log(car.start()); // Output: Toyota starts!

Objects in JavaScript

Objects are collections of key-value pairs that allow you to store related data and methods together.

- Key Features:

1. Object Declaration:

```
let car = { make: "Toyota", model: "Camry", year: 2020 };
```

2. **Accessing Properties**:

- **Dot notation**:
- console.log(car.make); // Output: Toyota
- **Bracket notation**:

```
console.log(car["model"]); // Output: Camry
```

3. **Adding/Updating Properties**:

```
car.color = "Red"; // Adds a new property
car.year = 2022; // Updates an existing property
console.log(car); // Output: { make: 'Toyota', model: 'Camry', year: 2022, color:
    'Red' }
```

4. **Deleting Properties**:

```
delete car.color;
console.log(car); // Output: { make: 'Toyota', model: 'Camry', year: 2022 }
```

Decisions in JavaScript

Decision-making constructs allow you to execute code based on conditions.

- **if-else Statement**:
- Syntax:

```
let age = 18;
if (age >= 18) {
console.log("You can vote.");
```

```
} else {
console.log("You cannot vote."); } // Output: You can vote
```

- **else if Statement**:

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

3. **switch Statement:**

- **Syntax:**

```
let day = 3;
switch (day) {
case 1 : console.log("Monday");
break;
case 2: console.log("Tuesday");
break;
case 3: console.log("Wednesday");
break;
default: console.log("Invalid day"); }
// Output: Wednesday
```

Loops in JavaScript

- Loops allow you to execute a block of code repeatedly.
- **Types of Loops:**

1. for Loop:

- Used when the number of iterations is known.

```
for (let i = 0; i < 5; i++) {
console.log(i); // Output: 0, 1, 2, 3, 4 }
```

while Loop:

- Executes as long as the condition is true.

```
let i = 0; while (i < 5) {
console.log(i); // Output: 0, 1, 2, 3, 4
i++;
}
```

do-while Loop:

Executes at least once, then checks the condition.

```
let i = 0;
do { 23
console.log(i); // Output: 0, 1, 2, 3, 4
 i++; }
while (i < 5);
```

- **for...of Loop:**
- Iterates over iterable objects like arrays.

```
let fruits = ["apple", "banana", "cherry"];
for (let fruit of fruits) {
console.log(fruit); // Output: apple, banana, cherry }
```

- **for...in Loop:**
- Iterates over the keys of an object.

```
 for (let key in car) {
console.log(key, car[key]); // Output: make Toyota, model Camry, year 2020
```