INTRODUCTION

What is JavaScript? (JS)

- JavaScript is a high level, dynamic and interpreted programming language that is primarily used for client-side scripting on the web.
- JavaScript is often used to add interactive and dynamic effects to websites, web applications and mobile applications.
- It is also used for server side programming, desktop and mobile application development and game development.

Some common uses of JavaScript

- Creating interactive web pages
- Validating form data
- Animating images and text
- Creating dynamic user interfaces
- Fetching data from APIs
- Handling user events (eg. clicks, hover, scroll)
- Creating games and interactive simulations

HTML adds Structure to a web page, CSS styles it and JavaScript brings it to life by allowing users to interact with elements on the page, such as actions on clicking buttons, filling out forms, and showing animations.

Features of JavaScript

- All popular web browsers support JavaScript as they provide built-in execution environment.
- JavaScript follows the syntax and structure of the C programming language. Thus it is a structured programming language.
- JavaScript is a weakly typed language, where certain types are implicitly cast.
- JavaScript is an object oriented programming language.
- It is a light weighted and interpreted language.
- It is case-sensitive language
- JavaScript is supportable in several operating systems including Windows, MacOs etc
- It provides good control to the users over the web browsers.

Some popular JavaScript frameworks and libraries include

- React
- Angular
- Vue.js
- Ember.js

- JQuery

JavaScript is also used in various non-web contexts such as

- Node.js (server-side JavaScript)
- Desktop applications (eg. Electron, NW.js)
- Mobile applications (eg. React Native, angular mobile)
- Game Development (eg: Phaser)

How to Add JavaScript in HTML Document?

To add JavaScript in HTML document, several methods can be used. These methods include embedding JavaScript directly within the HTML file or linking an external JavaScript file.

Inline JavaScript

You can write JavaScript code directly inside the HTML element using the onclick, onmouseover, or other event handler attributes.
Example

```
<button onclick="alert('Button Clicked!')">
    Click Here
</button>
```

Internal JavaScript (Within <script> Tag)

You can write JavaScript code inside the <script> tag within the HTML file. This is known as internal JavaScript and is commonly placed inside the <head> or <body> section of the HTML document.

1. JavaScript Code Inside <head> Tag

Placing JavaScript within the **<head>** section of an HTML document ensures that the script is loaded and executed as the page loads. This is useful for scripts that need to be initialized before the page content is rendered.

Example

```
<head>
  <script>
    function myFun() {
      document.getElementById("demo")
        .innerHTML = "Content changed!";
    }
  </script>
</head>
```

2. JavaScript Code Inside <body> Tag

JavaScript can also be placed inside the <body> section of an HTML page. Typically, scripts placed at the end of the <body> load after the content, which can be useful if your script depends on the DOM being fully loaded.

Example

```
<body>
<h2>
    Add JavaScript Code
    inside Body Section
</h2>
<h3 id="demo" style="color:green;">
    Hello world
</h3>
<button type="button" onclick="myFun()">
    Click Here
</button>
<script>
    function myFun() {
        document.getElementById("demo")
            .innerHTML = "Content changed!";
    }

</script>
```

External JavaScript (Using External File)

For larger projects or when reusing scripts across multiple HTML files, you can place your JavaScript code in an external .js file. This file is then linked to your HTML document using the src attribute within a <script> tag.

Example

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

    /* Filename: script.js*/

    function myFun () {
    document.getElementById('demo')
        .innerHTML = 'Content Changed'
    }
```

History

In 1993, Mosaic, the first popular web browser, came into existence. In 1994, Netscape was founded by **Marc Andreessen** . He realized that the web needed to become more dynamic. Thus, a 'glue language' was believed to be necessary to enhance HTML and make web designing easier for designers and part-time programmers.

Consequently, in 1995, the company recruited **Brendan Eich** with the intention of implementing and embedding the Scheme programming language into the browser. However, before Brendan could begin, the company collaborated with Sun Microsystems to integrate Java into its Navigator browser to compete with Microsoft over web technologies and platforms.

With the presence of both Java and the scripting language, Netscape decided to give the scripting language a name similar to Java's. This decision led to the creation of **"JavaScript"**.

JS versions

Version	Name	Release Year	Features
ES1	**ECMAScript 1**	**1997**	**Initial Release**
ES2	**ECMAScript 2**	**1998**	**Minor Editorial changes**
ES3	**ECMAScript 3**	**1999**	**Added:** • Regular Expression • try/catch • Exception Handling • switch case **and** do-while
ES4	**ECMAScript 4**		**Abandoned due to conflicts**
ES5	**ECMAScript 5**	**2009**	**Added:** • JavaScript "strict mode" • JSON **support** • JS getters and setters
ES6	**ECMAScript 2015**	**2015**	**Added:** • let **and** const • Class **declaration** • import and export • for..of loop • Arrow functions

ES7	**ECMAScript 2016**	**2016**	**Added:** • Block scope for variable • async/await • Array.includes function • Exponentiation Operator
ES8	**ECMAScript 2017**	**2017**	**Added:** • Object.values • Object.entries • Object.getOwnPropertiesDescriptors
ES9	**ECMAScript 2018**	**2018**	**Added:** • spread operator • rest parameters
ES10	**ECMAScript 2019**	**2019**	**Added:** • Array.flat() • Array.flatMap() • Array.sort is now stable
ES11	**ECMAScript 2020**	**2020**	**Added:** • BigInt primitive type • nullish coalescing operator
ES12	**ECMAScript 2021**	**2021**	**Added:** • String.replaceAll() Method • Promise.any() Method
ES13	**ECMAScript 2022**	**2022**	**Added:** • Top-level await • New class elements • Static block inside classes
ES14	**ECMAScript 2023**	**2023**	**Added:** • toSorted method • toReversed method • findLast, and findLastIndex methods on Array, prototype and TypedArray.prototypet

How to run JS?

1. Using a Web Browser

Modern web browsers (like Chrome, Firefox, Edge, or Safari) have built-in JavaScript engines.

- **Steps**:
    1. Open your browser.
    2. Open the **Developer Tools**:
        - Press Ctrl + Shift + J (Windows/Linux) or Cmd + Option + J (Mac) to open the Console.
    3. Type or paste your JavaScript code into the Console and press **Enter** to execute it.

```
console.log("Hello, World!");
```

2. **Using a** .html **File**

You can embed JavaScript inside an HTML file and run it in a browser.

- **Steps**:
    1. Create an HTML file (e.g., index.html).
    2. Write the following code:

```
<!DOCTYPE html>
<html>
<head>
   <title>Run JavaScript</title>
</head>
<body>
   <h1>Check the console for the message</h1>
   <script>
      console.log("Hello from JavaScript!");
   </script>
</body>
```

    3. Open the file in a browser and check the Console (open Developer Tools).

3. **Using Node.js**

Node.js allows you to run JavaScript outside of the browser.

- **Steps**:
    1. Download and install Node.js.
    2. Create a JavaScript file (e.g., app.js) with the following content:

```
console.log("Hello, Node.js!");
```

    3. Open a terminal or command prompt, navigate to the file's directory, and run:

       node app.js

4. **Using Online Editors**

Online platforms allow you to run JavaScript without installing anything.

- Examples:
    - CodePen
    - JSFiddle
    - PlayCode

5. Using Integrated Development Environments (IDEs)

Use IDEs like **Visual Studio Code**, **WebStorm**, or **Sublime Text** for writing and testing JavaScript.

- Steps:
    1. Write your JavaScript code in a .js file.
    2. Use **Node.js** or browser-based tools to execute it.

Example

```
// Print a message to the console
console.log("JavaScript is running!");

// Simple addition
let sum = 5 + 3;
console.log("The sum is: " + sum);
```

VARIABLE

Variable declaration keywords(let,var,const)

A variable is like a container that holds data that can be reused or updated later in the program. In JavaScript, variables are declared using the keywords var, let, or const.

Global Variables

Global variables in JavaScript are those declared outside of any function or block scope. They are accessible from anywhere within the script, including inside functions and blocks. Variables declared without the var, let, or const keywords (prior to ES6) inside a function automatically become global variables.
However, variables declared with var, let, or const inside a function are local to that function unless explicitly marked as global using window (in browser environments) or global (in Node.js).

Example

```
<Script>
var data=200, //global variable
function abc(){
}
</Script>
```

Key Characteristics of Global Variables:

- **Scope:** Accessible throughout the entire script, including inside functions and blocks.
- **Automatic Global Variables:** If a variable is declared inside a function without var, let, or const, it automatically becomes a global variable (a common source of bugs).

Local Variables

Local variables are defined within functions in JavaScript. They are confined to the scope of the function that defines them and cannot be accessed from outside. Attempting to access local variables outside their defining function results in an error.

Example

```
<Script>
function abc(){
var x=10; //local variable
}
</Script>
```

Key Characteristics of Local Variables:

- **Scope:** Limited to the function or block in which they are declared.
- **Function-Specific:** Each function can have its own local variables, even if they share the same name.

How to use variables

- The scope of a variable or function determines what code has access to it.
- Variables that are created inside a function are local variables, and local variables can only be referred to by the code within the function.
- Variables created outside of functions are global variables, and the code in all functions has access to all global variables.
- If you forget to code the var keyword in a variable declaration, the JavaScript engine assumes that the variable is global. This can cause debugging problems.
- In general, it's better to pass local variables from one function to another as parameters than it is to use global variables. That will make your code easier to understand with less chance of errors.

JavaScript is a dynamically typed language. So the type of variable is decided at run time. Therefore there is no need to explicitly define the type of a variable. We can declare variables in JavaScript in three ways.

var Keyword

The var keyword is used to declare a variable. It has a function-scoped or globally-scoped behaviour.

Example

```
var n=10;
var n=12; //Re-declaration allowed
```

let Keyword

The let keyword is introduced in ES6, has block scope and cannot be re-declared in the same scope.

Example

```
let n=10;
n=20; //value can be updated

let n=15; //cannot re declare
console.log(n)
```

Output

```
20
```

const Keyword

The const keyword declares variables that cannot be reassigned. It's block-scoped as well.

Example

```
const n=100;
n=200; //this will throw an error
console.log(n)
```

Output

```
100
```

Variable naming rules

- Name must start with a letter (a to z or A to Z), under score (_) or dollar ($) sign.
- After first letter we can use digits(0-9). For example value1.
- JavaScript variables are case sensitive. Variable names like `name` and `Name` are treated as different variables.
- Reserved keywords like let, const, class, if, etc., cannot be used as variable names

Variable Scope in JavaScript

Scope determines the accessibility of variables in your code. JavaScript supports the following types of scope

1. Global Scope

Variables declared outside any function or block are globally scoped. While var, let, and const can all have global scope when declared outside a function, their behavior differs:

- var is added to the window object in browsers.
- let and const do not attach to the window object, making them safer for modern usage.

Example

```
var globalVar = "I am global";
let globalLet = "I am also global";
const globalConst = "I am global too";
```

2. Function Scope

Variables declared inside a function are accessible only within that function. This applies to var, let, and const:

Example

```
function test() {
  var localVar = "I am local";
  let localLet = "I am also local";
  const localConst = "I am local too";
   }
  console.log(localVar); // Error: not defined
```

3. Block Scope

Variables declared with let or const inside a block (e.g., inside {}) are block-scoped, meaning they cannot be accessed outside the block. var, however, is not block-scoped and will leak outside the block.

Example

```
{
    let blockVar = "I am block-scoped";
    const blockConst = "I am block-scoped too";
  }
  console.log(blockVar); // Error: not defined
```

Hoisting

Hoisting refers to the behaviour where JavaScript moves the declarations of variables, functions, and classes to the top of their scope during the compilation phase. This can sometimes lead to surprising results, especially when using var, let, const, or function expressions.

- Hoisting applies to variable and function declarations.
- Initializations are not hoisted, they are only declarations.

- var variables are hoisted with undefined, while let and const are hoisted but remain in the Temporal Dead Zone.

1. Variable Hoisting with var

When you use var to declare a variable, the declaration is hoisted to the top, but its value is not assigned until the actual code execution.

Example

```
console.log(a); // undefined
var a = 5;
```

The declaration var a is hoisted to the top, but a is initialized with undefined. Hence, logging results in undefined.

2. Variable Hoisting with let and const

Variables declared with let and const are hoisted but are not initialized. Accessing them before declaration results in a ReferenceError. Because the variables declared with let and const remain under a Temporal Dead Zone till they are initialized.

Example

```
console.log(b); // ReferenceError: Cannot access 'b' before initialization
let b = 10;
```

The variable is hoisted, but it's in the Temporal Dead Zone (TDZ) until the declaration line is executed.

3. Function Declaration Hoisting

Function declarations are fully hoisted, meaning the function can be called before it's defined in the code.

Example

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

The function declaration is hoisted, and the entire function definition is available before its position in the code.

4. Function Expression Hoisting

Function expressions are not hoisted in the same way as function declarations. If you try to call them before their assignment, you'll get an error.

Example

```
hello(); // TypeError: hello is not a function
var hello = function() {
console.log("Hi!");
};
```

The variable hello is hoisted, but since it's a function expression, it's not initialized until the line is executed.

5. Hoisting with let and const in Functions

If you declare variables inside a function using let or const, they are hoisted only within that function scope and cannot be accessed before the declaration.

Example

```
function test() {
    console.log(x); // ReferenceError: Cannot access 'x' before initialization
    let x = 50;
    }
    test();
```

The let variable x is hoisted, but the TDZ means it cannot be accessed before it's initialized.

6. Re-declaring Variables with var

With var, you can redeclare a variable within the same scope. This is a unique behavior compared to let and const.

Example

```
var a = 10;
 var a = 20; // No error
 console.log(a); // 20
```

With var, the second declaration overwrites the first one without throwing an error.

7. Accessing Variables Declared Later in Loops

Hoisting impacts how you access variables inside loops. With var, variables are hoisted outside the loop, causing unexpected behavior.

Example

```
for (var i = 0; i < 3; i++) {
setTimeout(function() {
   console.log(i); // 3, 3, 3
 }, 100);
 }
```

The var i is hoisted, and all setTimeout functions share the same i reference, which results in the value 3 after the loop finishes.

8. Using Hoisted Functions with Parameters

Functions can be hoisted with their parameters, but any parameters passed to the function are still determined by the invocation, not by the hoisting.

Example

```
test(10); // 10
function test(num) {
console.log(num);
}
```

The entire function, including its parameters, is hoisted and available for use before the function's declaration in the code.

9. Hoisting in Nested Functions

Hoisting works within nested functions as well. Variables declared with var inside a function are hoisted to the top of that function scope.

Example

```
function outer() {
console.log(a); // undefined
var a = 5;
function inner() {
   console.log(b); // undefined
   var b = 10;
 }
 inner();
}
```

- Both a and b are hoisted within their respective scopes (outer and inner functions), but their values are not set until the code execution reaches the initialization lines.

Conclusion

Hoisting is a key concept to understand in JavaScript, especially when working with variables and functions. It can sometimes lead to confusing behavior, but once you grasp its rules and limitations, you'll be able to write more predictable and effective code. Be sure to use let and const for better scope management, and always be mindful of hoisting's impact on your code structure.

DATA TYPES

JavaScript supports various datatypes, which can be broadly categorized into primitive and non-primitive types.

Primitive Datatypes

Primitive datatypes represent single values and are immutable.

1. **Number:** Represents numeric values (integers and decimals).

Example

```
let n=42;
let pi=3.14;
```

2. String: Represents text enclosed in single or double quotes.

Example

```
let s= ”Hello, world”;
```

3. Boolean: Represents a logical value (true or false).

Example

```
let bool =true;
```

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

Example

```
let notAssigned;
console.log(notAssigned);
```

Output

```
Undefined
```

5. Null: Represents an intentional absence of any value.

Example

```
let empty=null;
```

6. **Symbol:** Represents unique and immutable values, often used as object keys.

Example

```
let sym=Symbol('unique');
```

7. BigInt: Represents integers larger than Number.MAX_SAFE_INTEGER.

Example

```
let bigNumber=123456789012345678901234567890n;
```

Non-Primitive Datatypes

Non-primitive types are objects and can store collections of data or more complex entities.

1. Object: Represents key-value pairs.

Example

```
let obj={
name: "Amit",
age: 25,
}
```

2. Array: Represents an ordered list of values.

Example

```
let a=[“red”, “green”, ”blue”];
```

3. Function: Represents reusable blocks of code.

Example

```
function fun() {
Console.log(“Hello”);
}
```

TYPECASTING

Typecasting in JavaScript (also known as **type conversion**) is the process of converting a value from one data type to another. JavaScript is a dynamically typed language, meaning variables can hold any data type and type conversion can happen **implicitly** (automatic) or **explicitly** (manually by the developer).

1. Implicit Typecasting (Type Coercion)

JavaScript automatically converts data types in certain situations, like during string concatenation or mathematical operations.

1. String Coercion:
    - When a number is added to a string, JavaScript converts the number to a string.

    Example

    ```
    let result = "The value is " + 42; // Implicitly converts 42 to "42"
    console.log(result); // Output: "The value is 42"
    ```

2. Numeric Coercion:
    - When a string contains a numeric value and is used in a numeric operation, JavaScript converts it to a number.

    Example

    ```
    let result = "5" * 2; // Implicitly converts "5" to 5
    console.log(result); // Output: 103.
    ```

3. Boolean Coercion:
    - Non-boolean values are converted to true or false in logical operations.

Example

```
console.log(Boolean(0)); // Output: false
console.log(Boolean("Hello")); // Output: true
```

2. Explici

Explicit typecasting is done using JavaScript methods or operators to ensure a value is converted to a specific data type.

Convert to String

1. Using String**()** function:

   Example

   ```
   let num = 42;
   let str = String(num); // Converts 42 to "42"
   console.log(typeof str); // Output: "string"
   ```

2. Using **.**toString() method:

   Example

   ```
   let num = 42;
   let str = num.toString(); // Converts 42 to "42"
   console.log(typeof str); // Output: "string"
   ```

Convert to Number

1. Using Number() function:

   Example

   ```
   let str = "42";
   let num = Number(str); // Converts "42" to 42
   console.log(typeof num); // Output: "number"
   ```

2. Using Unary + operator:

   Example

   ```
   let str = "42";
   let num = +str; // Converts "42" to 42
   console.log(typeof num); // Output: "number"
   ```

3. Using parseInt() or parseFloat()**:**
    - parseInt() converts a string to an integer.
    - parseFloat() converts a string to a floating-point number.

Example

```
let intVal = parseInt("42px"); // Output: 42
let floatVal = parseFloat("3.14"); // Output: 3.14
```

Convert to Boolean

1. Using Boolean() function:

Example

```
let val = 0;
console.log(Boolean(val)); // Output: false
val = "Hello";
console.log(Boolean(val)); // Output: true
```

2. Double NOT **!!** operator:

Example

```
let val = 0;
console.log(!!val); // Output: false
val = "World";
console.log(!!val); // Output: true
```

Typecasting Examples

1. String to Number Conversion:

```
let str = "123";
console.log(Number(str)); // Output: 123
console.log(+str);        // Output: 123
console.log(parseInt(str)); // Output: 123
```

2. Number to String Conversion:

```
let num = 456;
console.log(String(num));    // Output: "456"
console.log(num.toString()); // Output: "456"
```

3. Boolean to Number Conversion:

```
console.log(Number(true));  // Output: 1
console.log(Number(false)); // Output: 0
```

4. **Number to Boolean Conversion:**

```
console.log(Boolean(0));    // Output: false
console.log(Boolean(123));  // Output: true
```

Truthy and Falsy Values

When converting to a boolean, JavaScript treats some values as **truthy** (convert to true) and others as **falsy** (convert to false).

Falsy Values:

- false
- 0
- "" (empty string)
- null
- undefined
- NaN

Truthy Values:

- Any value that is not falsy (e.g., non-empty strings, non-zero numbers, objects).

Example:

```
console.log(Boolean(0));       // false
console.log(Boolean("Hello")); // true
console.log(Boolean({}));      // true
```

Common Pitfalls

1. Using == instead of ===:
    - == performs type coercion, which can lead to unexpected results.

Example

```
console.log("5" == 5);  // true (because "5" is coerced to 5)
console.log("5" === 5); // false (strict comparison, no coercion)
```

2. NaN (Not-a-Number):
    - NaN is the result of invalid number conversions.

Example

```
let result = Number("abc");
console.log(result); // Output: NaN
console.log(isNaN(result)); // true
```

- Use **implicit typecasting** when JavaScript's default behavior is acceptable.
- Use **explicit typecasting** for clarity and to avoid unexpected results.
- Be cautious of potential issues with **type coercion** when comparing values. Use === for strict comparisons.

DATA STRUCTURES

In JavaScript, **data structures** are used to organize and store data efficiently. JavaScript provides several built-in data structures, which can be categorized into two types:

1. Primitive Data Structures

These are the basic, immutable data types in JavaScript.

1. **String**
    - Used to represent text.

Example

```
let name = "Alice";
let greeting = "Hello, " + name;
```

2. Non-Primitive Data Structures

These are objects and collections that can hold multiple values and are mutable.

1. Object

- A collection of key-value pairs.
- Keys are strings (or symbols), and values can be of any type.

Example:

```
let person = {
name: "John",
age: 30,
isStudent: false
};
console.log(person.name); // Output: John
```

2. Array

- An ordered collection of elements, indexed numerically.
- Arrays can store any type of data.

Example:

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

3. Set

- A collection of unique values.
- Does not allow duplicate elements.

Example:

```
let uniqueNumbers = new Set([1, 2, 3, 3, 4]);
console.log(uniqueNumbers); // Output: Set { 1, 2, 3, 4 }
```

OPERATORS

JavaScript operators are symbols or keywords used to perform operations on values and variables. They are the building blocks of JavaScript expressions and can manipulate data in various ways.

There are various operators supported by JavaScript.

1. JavaScript Arithmetic Operators

Arithmetic Operators perform mathematical calculations like addition, subtraction, multiplication, etc.

Example

```
const sum = 5 + 3; // Addition
const diff = 10 - 2; // Subtraction
const p = 4 * 2; // Multiplication
const q = 8 / 2; // Division
console.log(sum, diff, p, q);

Output
  8 8 8 4
```

- + adds two numbers.
- – subtracts the second number from the first.
- * multiplies two numbers.
- / divides the first number by the second.

2. JavaScript Assignment Operators

Assignment operators are used to assign values to variables. They can also perform operations like addition or multiplication before assigning the value.

Example

```
let n = 10;
n += 5;
n *= 2;
console.log(n);
```

- = assigns a value to a variable.
- += adds and assigns the result to the variable.
- *= multiplies and assigns the result to the variable.

3. JavaScript Comparison Operators

Comparison operators compare two values and return a boolean (true or false). They are useful for making decisions in conditional statements.

Example

```
console.log(10 > 5);
    console.log(10==="10");
    Output
    True
    false
```

- > checks if the left value is greater than the right.
- === checks for strict equality (both type and value).
- Other operators include <, <=, >=, and !==.

4. JavaScript Logical Operators

Logical operators are mainly used to perform the logical operations that determine the equality or difference between the values.
Example

```
const a = true, b = false;
    console.log(a && b); // Logical AND
    console.log(a || b); // Logical OR

    Output
    false
    true
```

- && returns true if both operands are true.
- || returns true if at least one operand is true.
- ! negates the boolean value.

5. JavaScript Bitwise Operators

Bitwise operators perform operations on binary representations of numbers.

Example

```
const res = 5 & 1; // Bitwise AND

console.log(res);

Output

  1
```

- & performs a bitwise AND.
- | performs a bitwise OR.
- ^ performs a bitwise XOR.
- ~ performs a bitwise NOT.

6. JavaScript Ternary Operator

The ternary operator is a shorthand for conditional statements. It takes three operands.

Example

```
const age = 18;
const status = age >= 18 ? "Adult" : "Minor";
console.log(status);

Output
Adult
```

condition ? expression1 : expression2 evaluates expression1 if the condition is true, otherwise evaluates expression2.

7. JavaScript Comma Operator

Comma Operator (,) mainly evaluates its operands from left to right sequentially and returns the value of the rightmost operand.

Example

```
let n1, n2

const res = (n1 = 1, n2 = 2, n1 + n2);
console.log(res);

Output

3
```

- Each expression is evaluated from left to right.
- The final result of the expression is the rightmost value.

8. JavaScript Unary Operators

Unary operators operate on a single operand (e.g., increment, decrement).

Example

```
let x = 5;
console.log(++x); // Pre-increment
console.log(x--); // Post-decrement (Output: 6, then x becomes 5)
Output
6
6
```

- ++ increments the value by 1.
- — decrements the value by 1.
- typeof returns the type of a variable.

9. JavaScript Relational Operators

JavaScript **Relational operators** are used to compare its operands and determine the relationship between them. They return a Boolean value (true or false) based on the comparison result.

Example

```
const obj = { length: 10 };
      console.log("length" in obj);
      console.log([] instanceof Array);

     Output
     true
     true
```

- **in** checks if a property exists in an object.
- **instanceof** checks if an object is an instance of a constructor.

10. JavaScript BigInt Operators

BigInt operators allow calculations with numbers beyond the safe integer range.

Example

```
const big1 = 12345678901234567890123456789 0n;
     const big2 = 98765432109876543210987654321 0n;
     console.log(big1+big2);

    Output
    111111111011111111011111111100n
```

- Operations like addition, subtraction, and multiplication work with BigInt.
- Use **n** suffix to denote BigInt literals.

11. JavaScript String Operators

JavaScript String Operators include concatenation (+) and concatenation assignment (+=), used to join strings or combine strings with other data types.

Example

```
const s = "Hello" + " " + "World";
console.log(s);
```

Output

```
Hello World
```

- + concatenates strings.
- += appends to an existing string.

12. JavaScript Chaining Operator (?.)

The optional chaining operator allows safe access to deeply nested properties without throwing errors if the property doesn't exist.

Example

```
const obj = { name: "Aman", address: { city: "Delhi" } };
console.log(obj.address?.city);
console.log(obj.contact?.phone);
```

Output

```
Delhi
undefined
```

- **?.** safely accesses a property or method.
- Returns undefined if the property doesn't exist.