

Introduction to JavaScript

What is a Scripting language?

A scripting language is a type of programming language designed to automate tasks, control applications, or manipulate data within a specific environment. Scripting languages are often interpreted rather than compiled, meaning they are executed directly without requiring a separate compilation step.

Client-Side Scripting: Client-side scripting refers to code that is executed directly in the user's web browser. The focus is on creating a dynamic and interactive user experience.

Server-Side Scripting: Server-side scripting refers to code that runs on the web server before the content is sent to the user's browser. It is primarily used for handling backend operations, data processing, and database interactions.

What is JavaScript?

JavaScript is a high-level, interpreted, Client-Side Scripting language primarily used to create dynamic, interactive features on websites. It enables developers to add functionality such as animations, form validation, content updates, and interactive elements to web pages. JavaScript is a core technology of the web, along with **HTML** (structure) and **CSS** (styling).

JavaScript was initially created to “make web pages alive.” The programs in this language are called **scripts**. They can be written right in a web page's HTML and run automatically as the page loads. Scripts are provided and executed as plain text. They don't need special preparation or compilation to run. In this aspect, JavaScript is very different from another language called Java.

Why is it called JavaScript?

When JavaScript was created, it initially had another name: “LiveScript.” But Java was very popular at that time, so it was decided that positioning a new language as a “younger brother” of Java would help. But as it evolved, JavaScript became a fully independent language with its own specification called ECMAScript, and now it has no relation to Java at all.

Today, JavaScript can execute not only in the browser, but also on the server, or actually on any device that has a special program called the JavaScript engine. The browser has an embedded engine sometimes called a “JavaScript virtual machine.” Different engines have different “codenames.” For example: V8 – in Chrome, Opera and Edge, SpiderMonkey – in Firefox, Chakra for IE, JavaScriptCore, Nitro and SquirrelFish for Safari, etc.

Key Features of JavaScript

- **Client-Side Execution:** JavaScript runs in the user's web browser, allowing dynamic content and faster user interactions.
- **Lightweight and Easy to Learn:** The syntax is simple and beginner-friendly, making it one of the most popular programming languages.
- **Event-Driven:** Reacts to user actions (e.g., clicks, key presses) to make web pages interactive.
- **Object-Oriented:** Supports object-oriented principles, making it suitable for building complex applications.

- **Cross-Platform:** Works on all major browsers and operating systems.
- **Versatile:** Can be used for both front-end (browser) and back-end (server, with Node.js) development.

Uses of JavaScript

- **Web Development:** Add interactivity to web pages (e.g., form validation, dropdown menus). Create single-page applications (SPAs) with frameworks like React, Angular, or Vue.js.
- **Server-Side Development:** With Node.js, JavaScript can handle server-side logic, APIs, and database operations.
- **Mobile App Development:** Frameworks like React Native allow developers to build mobile apps using JavaScript.
- **Game Development:** Used to create browser-based games with libraries like Phaser.
- **Automation and Testing:** Automate tasks using tools like Puppeteer or create test scripts with libraries like Jest.

History of JavaScript

JavaScript has an interesting history, marked by rapid development and widespread adoption. Here's a detailed timeline of its evolution:

1. The Birth of JavaScript (1995)

- **Who:** JavaScript was created by **Brendan Eich**, a developer at Netscape Communications.
- **Why:** Netscape wanted a lightweight scripting language for web browsers to make websites more dynamic and interactive.
- **When:** JavaScript was developed in just **10 days** in May 1995.
- **Original Name:** It was initially called **Mocha**, then **LiveScript**, and finally renamed **JavaScript** to ride on the popularity of Java, despite being unrelated.

2. Early Adoption (1996-1997)

- In **1996**, JavaScript was submitted to **ECMA International**, a standards organization, to standardize the language.
- This resulted in the creation of **ECMAScript**, the official specification of JavaScript.

Key Milestone:

- **1997:** The first version of ECMAScript, **ECMAScript 1 (ES1)**, was released.

3. Browser Wars (1995–2000)

- During the late 1990s, there was intense competition between Netscape's **Navigator** and Microsoft's **Internet Explorer**.
- Both browsers implemented JavaScript differently, causing compatibility issues.
- Microsoft introduced its own version, called **JScript**, leading to fragmentation in web development.

4. Standardization and Growth (2000–2005)

- The release of **ECMAScript 3 (ES3)** in **1999** solidified the language's features, making it more robust.
- JavaScript gained popularity with the rise of **dynamic web pages**.
- Developers used JavaScript for features like form validation and simple interactivity.

5. The Ajax Revolution (2005–2010)

- **2005:** The introduction of **Ajax (Asynchronous JavaScript and XML)** revolutionized web development by enabling asynchronous communication with servers.
 - Websites could update parts of a page without requiring a full reload (e.g., Google Maps, Gmail).
- This period saw the rise of **JavaScript libraries** like:
 - **jQuery** (2006): Simplified DOM manipulation and event handling.
 - **Prototype.js**: Helped with cross-browser compatibility.

6. Modern JavaScript Frameworks and ES6 Era (2010–2015)

- The demand for more complex web applications led to the development of frameworks and libraries:
 - **AngularJS** (2010): First popular JavaScript framework for building Single Page Applications (SPAs).
 - **React** (2013): Introduced by Facebook, became a game-changer for building dynamic user interfaces.
 - **Vue.js** (2014): Lightweight and flexible, gaining popularity among developers.

ECMAScript 6 (ES6) – 2015:

- A major update to the JavaScript language, adding features like:
 - **Arrow functions:** `() => {}` for shorter syntax.
 - **let/const:** Block-scoped variables.
 - **Template literals:** Backticks (```) for easier string formatting.
 - **Classes:** Cleaner object-oriented programming syntax.
 - **Promises:** Simplified asynchronous programming.

7. The Node.js Revolution (2009–Present)

- **2009:** **Node.js** was introduced by **Ryan Dahl**. It enabled JavaScript to run on the server, making it a full-stack programming language.
 - Enabled building scalable back-end applications.
 - Fuelled the rise of the **MEAN/MERN stack** (MongoDB, Express.js, Angular/React, Node.js).

8. Recent Developments (2015–Present)

- **ECMAScript Updates:**
 - ECMAScript updates are now released annually (e.g., ES7, ES8, etc.).
 - Features like **async/await**, **modules**, and **optional chaining** have made JavaScript more powerful and developer-friendly.
- **Modern Frameworks:**
 - Libraries and frameworks like **React**, **Vue.js**, **Angular**, and **Svelte** dominate the front-end landscape.
- **Tooling and Ecosystem:**
 - Tools like **Webpack**, **Babel**, and **ESLint** streamline development.
 - **npm** (Node Package Manager) has become the largest package ecosystem for developers.

9. JavaScript Today: JavaScript is now the most popular programming language globally, powering nearly all modern web applications. It has grown from a simple scripting language to a versatile, full-fledged programming environment.

What In-Browser (Client-Side JavaScript) JavaScript Can Do: In-browser JavaScript is powerful and plays a significant role in enhancing user experience on web pages. Here are the main things it can do:

- **Manipulate the Web Page:** JavaScript can interact with and modify the **DOM (Document Object Model)**, allowing developers to create dynamic and interactive web pages.
- **Handle User Interactions:** JavaScript can respond to user actions like clicks, key presses, or form submissions using **event listeners**.
- **Validate and Process Data:** JavaScript can validate user inputs on the client side before sending them to the server, reducing server load.
- **Communicate with Servers:** JavaScript can send and receive data from a server without reloading the page using **AJAX** or **Fetch API**.
- **Work with Multimedia:** JavaScript can manipulate audio, video, and images directly within the browser.
- **Store Data Locally:** JavaScript can store and retrieve data in the browser using Cookies, Local Storage and Session Storage.
- **Create Animations:** JavaScript can animate elements on a page, often in combination with **CSS**.
- **Access Geolocation and Device APIs:** JavaScript can access certain device capabilities via browser APIs.

What In-Browser (Client-Side JavaScript) JavaScript Cannot Do: While JavaScript is powerful, its abilities are intentionally restricted for security and privacy reasons. Here are things it cannot do:

- **Access Local Files Directly:** JavaScript cannot access or read files stored on a user's local machine, except for files explicitly provided by the user (e.g., via a file input). Using APIs like the File System Access API, but this requires explicit user permission and is limited.
- **Directly Access the File System:** JavaScript cannot write to the user's file system (e.g., save or modify files directly on the user's disk). Can access devices via the WebRTC API (for cameras/microphones) or the WebUSB API with explicit permission.
- **Run System Commands:** JavaScript cannot execute system-level commands (e.g., opening a terminal, running shell commands).
- **Access Other Browser Tabs:** JavaScript cannot interact with or access the contents of other tabs or browser windows, ensuring privacy.
- **Bypass Same-Origin Policy:** JavaScript is restricted by the **same-origin policy**, meaning it cannot make requests to domains other than the one that served the web page. Cross-Origin Resource Sharing (CORS) allows controlled access with server permission.
- **Perform Background Tasks Without User Consent:** JavaScript cannot perform tasks in the background indefinitely unless explicitly allowed (e.g., via Web Workers or Service Workers).
- **Access Sensitive System Information:** JavaScript cannot access sensitive system details like: Installed software, User's file structure, Operating system-level details.
- **Work Outside the Browser Sandbox:** JavaScript operates in a **sandboxed environment**, meaning it cannot: Access the operating system directly, Modify browser settings.
- **Prevent Browser Security Restrictions:** JavaScript cannot bypass browser-enforced security features, such as: Pop-up blockers, Content Security Policies (CSP).

Such limitations do not exist if JavaScript is used outside of the browser, for example on a server.

Server-Side JavaScript: Runs on the server, using environments like **Node.js**, **Deno**, or server-side frameworks (e.g., Express.js for Node.js). It is used for backend operations, such as handling database queries, processing logic, and serving content to the client, and execution happens on the server before any content is sent to the browser.

Capabilities:

- Handle HTTP requests and responses.
- Perform database operations (e.g., CRUD operations with MongoDB, MySQL).
- Implement server-side logic like user authentication and authorization.
- Generate dynamic HTML to send to the client.
- Manage file operations on the server (e.g., reading and writing files).
- Work with APIs and integrate external services.
- Process complex data on the server to reduce the client's workload.

What makes JavaScript unique?

There are at least three great things about JavaScript:

- Full integration with HTML/CSS.
- Simple things are done simply.
- Supported by all major browsers and enabled by default.

Languages “over” JavaScript

The syntax of JavaScript does not suit everyone's needs. Different people want different features. That's to be expected, because projects and requirements are different for everyone. So, recently a plethora of new languages appeared, which are **transpiled** (converted) to JavaScript before they run in the browser.

Modern tools make the transpilation very fast and transparent, actually allowing developers to code in another language and auto-converting it “under the hood”.

Examples of such languages:

- **CoffeeScript** is “syntactic sugar” for JavaScript. It introduces shorter syntax, allowing us to write clearer and more precise code. Usually, Ruby devs like it.
- **TypeScript** is concentrated on adding “strict data typing” to simplify the development and support of complex systems. It is developed by Microsoft.
- **Flow** also adds data typing, but in a different way. Developed by Facebook.
- **Dart** is a standalone language that has its own engine that runs in non-browser environments (like mobile apps), but also can be transpiled to JavaScript. Developed by Google.
- **Brython** is a Python transpiler to JavaScript that enables the writing of applications in pure Python without JavaScript.
- **Kotlin** is a modern, concise and safe programming language that can target the browser or Node.

Manuals and specifications:

Specification: [The ECMA-262 specification](#) contains the most in-depth, detailed and formalized information about JavaScript. It defines the language.

But being that formalized, it's difficult to understand at first. So, if you need the most trustworthy source of information about the language details, the specification is the right place. But it's not for everyday use.

A new specification version is released every year. Between these releases, the latest specification draft is at <https://tc39.es/ecma262/>.

To read about new bleeding-edge features, including those that are "almost standard" (so-called "stage 3"), see proposals at <https://github.com/tc39/proposals>.

Manuals: MDN (Mozilla) JavaScript Reference is the main manual with examples and other information. It's great to get in-depth information about individual language functions, methods etc.

You can find it at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>.

Code editors: There are two main types of code editors: IDEs and lightweight editors. Many people use one tool of each type.

- **IDE:** The term IDE (Integrated Development Environment) refers to a powerful editor with many features that usually operates on a "whole project." As the name suggests, it's not just an editor, but a full-scale "development environment."

There are following options:

- Visual Studio Code (cross-platform, free).
- WebStorm (cross-platform, paid).

- **Lightweight editors:** "Lightweight editors" are not as powerful as IDEs, but they're fast, elegant and simple. They are mainly used to open and edit a file instantly.

There are many options, for instance:

- Sublime Text (cross-platform, shareware).
- Notepad++ (Windows, free).
- Vim and Emacs are also cool if you know how to use them.

Developer console

Code is prone to errors. You will quite likely make errors... Oh, what am I talking about? You are *absolutely* going to make errors, at least if you're a human, not a robot

But in the browser, users don't see errors by default. So, if something goes wrong in the script, we won't see what's broken and can't fix it.

To see errors and get a lot of other useful information about scripts, "developer tools" have been embedded in browsers.

Most developers lean towards Chrome or Firefox for development because those browsers have the best developer tools. Other browsers also provide developer tools, sometimes with special features, but are

usually playing “catch-up” to Chrome or Firefox. So most developers have a “favorite” browser and switch to others if a problem is browser-specific.

Developer tools are potent; they have many features. To start, we’ll learn how to open them, look at errors, and run JavaScript commands.

Google Chrome: There’s an error in the JavaScript code on it. It’s hidden from a regular visitor’s eyes, so let’s open developer tools to see it.

Press F12 or, if you’re on Mac, then Cmd+Opt+J.

The developer tools will open on the Console tab by default.

Installation:

There are multiple ways to execute JavaScript code locally, depending on whether you’re working with:

- Browser-based JavaScript (frontend)
- Node.js-based JavaScript (backend or general scripting)

1. Using the Browser Console

The easiest and most instant method.

- Open Chrome (or any browser)
- Press F12 or Ctrl + Shift + I → Go to the **Console** tab (or Source tab -> Snippet)
- Type JavaScript directly

2. Using a .html File (With Embedded or Linked JS)

Run JavaScript as part of a webpage.

Create an index.html file:

3. Using Live Server in VS Code

For real-time preview of JavaScript with HTML.

1. Install the **Live Server** extension in VS Code
2. Right-click index.html → **Open with Live Server**
3. Write and run JavaScript inside HTML or via a separate .js file

4. Using Node.js in Terminal (Backend / Scripting JS)

Node allows running JavaScript outside the browser.

- Install Node.js (<https://nodejs.org/en>)
- Create a file like app.js:

```
console.log("Running in Node!");
```
- Run it using command prompt or PowerShell as:

```
node app.js
```

JavaScript Character: The JavaScript character set defines the characters that can be used in JavaScript code. These characters are part of the Unicode standard, which is a universal encoding system designed to support virtually all written languages and symbols.

Key Components of the JavaScript Character Set:

Letters (Alphabetic Characters):

- Uppercase letters: A-Z
- Lowercase letters: a-z
- Unicode allows the use of letters from other languages, like ç, é, or α.

Digits:

- 0-9 are valid for numbers or identifiers (e.g., variable names starting from the second character).

Whitespace Characters:

- Spaces and tabs help improve code readability.
- Examples:
 - Space (U+0020)
 - Tab (U+0009)
 - Line terminators like \n (newline) and \r (carriage return).

Constants, Keywords, and Variables in JavaScript:

1. Constants: Constants are variables whose values cannot be changed once assigned. They are defined using the const keyword.

```
const CONSTANT_NAME = value;
```

- Rules:
 - Must be initialized during declaration.
 - Cannot be reassigned.
 - If the value is an object or array, its properties or elements **can** be modified (but the reference cannot change).

2. Keywords

JavaScript keywords are reserved words used by the language to perform specific operations. They cannot be used as variable, function, or identifier names.

Some common JavaScript keywords:

- Variable declaration: var, let, const
- Control flow: if, else, switch, case, default
- Loops: for, while, do
- Functions: function, return
- Objects/Classes: class, extends, this, super
- Asynchronous operations: async, await, Promise

- Others: break, continue, try, catch, throw, typeof, delete

3. Variables

Variables store data that can be used and manipulated in the program. They are declared using the following keywords:

- var
- let
- const

Rules for Variable Names

1. Can include letters, digits, \$, and _.
2. Cannot begin with a digit.
3. Case-sensitive (variable and Variable are different).
4. Cannot use reserved JavaScript keywords.

Output Instruction in JavaScript:

In JavaScript, there are multiple ways to output information or display data to the user. Each method serves a specific purpose depending on the context, such as debugging, displaying messages in the browser, or showing alerts. Here's an explanation of the different output instructions:

1. console.log(): Outputs data to the **browser's console**. Commonly used for debugging or displaying messages during development.

Program: usingconsole.js

```
console.log("This introduction to Console application");  
console.log("Welcome to Archer Infotech Pune");  
console.log("Welcome to Deep Dive to JavaScript");  
console.log("Thank You");
```

2. alert(): Displays a **popup alert box** with a message. Used for simple notifications or warnings but not commonly used in modern web development due to its intrusive nature.

Program: usingalert.js (execute in Browser)

```
<!DOCTYPE html>  
<html>  
<head>  
  <title>Alert Example</title>  
</head>  
<body>  
  <script>  
    // Display an alert message  
    alert("Hello, welcome to JavaScript programming!");  
  </script>
```

```
</body>
```

```
</html>
```

3. document.write(): Writes directly into the HTML document. Often used to display output on the web page during development or testing but is rarely used in modern applications.

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
<head>
```

```
  <meta charset="UTF-8">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>document.write() Example</title>
```

```
</head>
```

```
<body>
```

```
  <script>
```

```
    document.write("Hello, World! This text is written using document.write().");
```

```
  </script>
```

```
</body>
```

```
</html>
```

4. innerHTML: The innerHTML property in JavaScript is used to get or set the HTML content of an element. It allows you to add, modify, or replace the content of an element dynamically.

Here's a detailed explanation with examples:

Program: innerhtml.html

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
<head>
```

```
  <meta charset="UTF-8">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>innerHTML Example</title>
```

```
</head>
```

```
<body>
```

```
  <div id="example">This is the original content.</div>
```

```
  <script>
```

```
    // Access the element
```

```
    const div = document.getElementById("example");
```

```
    // Replace its content
```

```
    div.innerHTML = "This is the <strong>new content</strong> added using innerHTML.";
```

```
  </script>
```

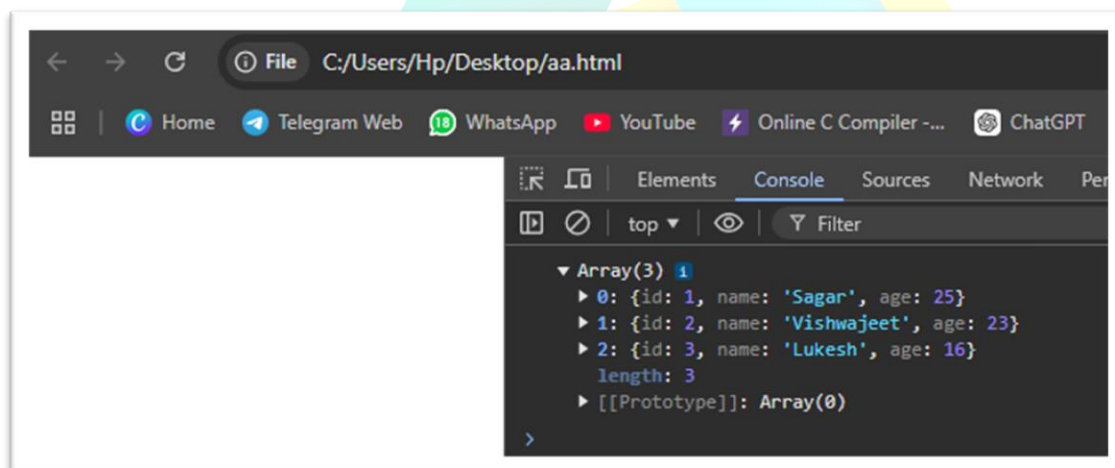
```
</body>
```

```
</html>
```

5. console.table(): Displays tabular data in the console. Useful for visualizing arrays or objects.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>console.table Example</title>
</head>
<body>
  <script>
    const users = [
      { id: 1, name: "Sagar", age: 25 },
      { id: 2, name: "Vishwajeet", age: 23 },
      { id: 3, name: "Lukesh", age: 16 }
    ];

    console.table(users);
  </script>
</body>
</html>
```



Sample Program to Change the page content dynamically

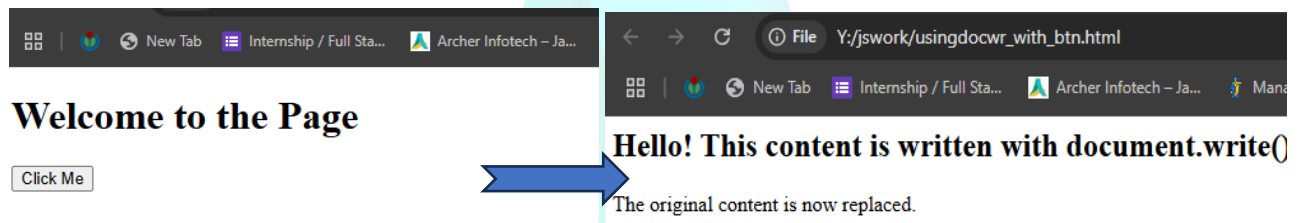
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```

<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document Write Example</title>
</head>
<body>
  <h1>Welcome to the Page</h1>
  <button onclick="writeContent()">Click Me</button>

  <script>
    function writeContent() {
      // This will overwrite the current document
      document.write("<h2>Hello! This content is written with document.write().</h2>");
      document.write("<p>The original content is now replaced.</p>");
    }
  </script>
</body>
</html>

```



Common Escape Sequences

Escape Sequence	Description	Example Output
\'	Single quote	'It\'s a nice day' → It's a nice day
\"	Double quote	"He said, \"Hello!\"" → He said, "Hello!"
\\	Backslash	"This is a backslash: \\" → This is a backslash: \
\n	New line	"Hello\nWorld" → Hello (new line) World
\t	Tab	"Hello\tWorld" → Hello World
\r	Carriage return	"Hello\rWorld" → (Depends on system)
\b	Backspace	"Hello\bWorld" → HellWorld
\f	Form feed (rarely used)	"Hello\fWorld" → Hello□World (depends on rendering)
\v	Vertical tab (rarely used)	"Hello\vWorld" → (Behavior varies)

```

console.log('Hello\nWorld'); // New line
console.log('It\'s a great day!'); // Escape single quote
console.log("He said, \"JavaScript is awesome!\""); // Escape double quote
console.log("Path: C:\\Program Files\\App"); // Escape backslash
console.log("Tab\tSpace"); // Tab space

```

Java Data types:

JavaScript has several data types divided into two main categories: **Primitive Data Types** and **Non-Primitive Data Types (Objects)**. Below is a detailed explanation of each data type with examples:

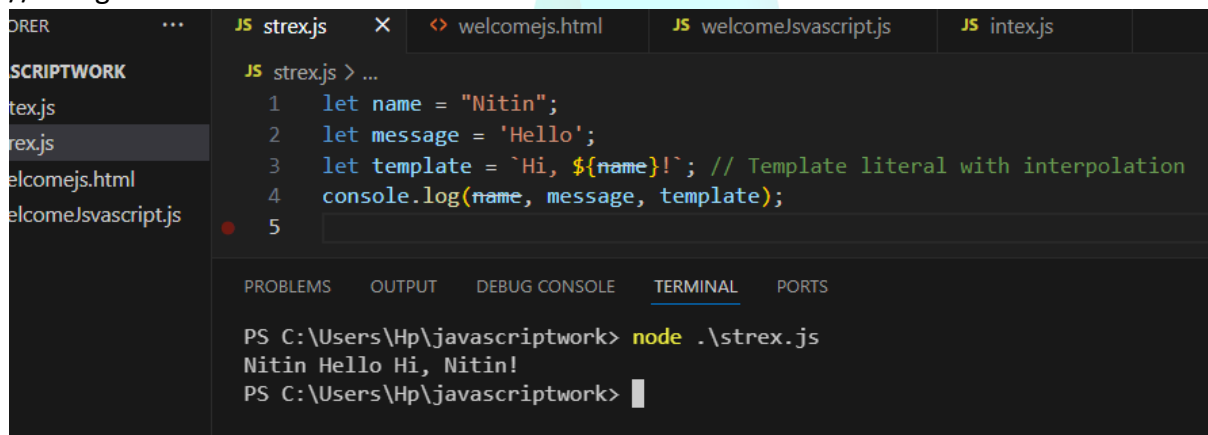
Primitive Data Types

1. Number: Represents both integer and floating-point numbers. Includes special values: Infinity, -Infinity, and NaN (Not-a-Number).

```
let age = 25;           // Integer
let price = 99.99;      // Floating-point number
let infiniteValue = Infinity; // Infinity
let invalidNumber = 'hello' / 2; // NaN
console.log(age, price, infiniteValue, invalidNumber);
```

2. String: Represents a sequence of characters enclosed in single quotes ('), double quotes ("), or backticks (`) for template literals. Strings are immutable.

// using node



The screenshot shows a VS Code editor with a file named 'strex.js' open. The code in the file is as follows:

```
1 let name = "Nitin";
2 let message = 'Hello';
3 let template = `Hi, ${name}!`; // Template literal with interpolation
4 console.log(name, message, template);
5
```

The terminal output shows the command 'node .\strex.js' being executed, resulting in the output: 'Nitin Hello Hi, Nitin!'.

// Using in browse – writing on Console:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>JavaScript Example</title>
</head>
<body>
  <h1>JavaScript Template Literals Example</h1>
  <p>Check the console for the output!</p>

  <script>
    let name = "Nitin";
    let message = 'Hello';
    let template = `Hi, ${name}!`; // Template literal with interpolation
    console.log(name, message, template);
  </script>
```

```
</body>
</html>
```

// Using in browse – writing on browser canvas:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>JavaScript Example</title>
</head>
<body>
  <h1>JavaScript Display Methods Example</h1>
  <div id="output"></div> <!-- Div to display data using innerHTML -->

  <script>
    let name = "Nitin";
    let message = 'Hello';
    let template = `Hi, ${name}!`; // Template literal with interpolation

    // Display using document.write()
    document.write(`<p>Using document.write: ${template}</p>`);

    // Display using alert()
    alert(`Using alert: ${template}`);

    // Display using innerHTML
    document.getElementById('output').innerHTML = `<p>Using innerHTML: ${template}</p>`;
  </script>
</body>
</html>
```

3. Boolean: Represents two values: true or false. Often used in conditional statements.

```
let isJavaScriptFun = true;
let isOld = false;
console.log(isJavaScriptFun, isOld);
```

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

```
var v1=45;
let v2;
console.log(v1 , v2);
```

Note:

- Use **let (Preferred): Block-scoped** (works only within { }), **Re-declaration not allowed** in the same scope, Use when the variable **value might change**
- Use **var (Old, avoid if possible): Function-scoped** (ignores block { }), Can be **re-declared** and **overwritten**, Use only if you're maintaining **older code**

5. Null: Represents an intentional absence of a value (manually set to null).

```
let emptyValue = null;  
console.log(emptyValue); // Output: null
```

Symbol (Introduced in ES6): Represents a unique identifier. Symbols are guaranteed to be unique.

The Symbol is a unique and immutable data type introduced in **ES6 (ECMAScript 2015)**. It is primarily used to create unique identifiers for object properties.

Key Characteristics of Symbol

1. **Uniqueness:** Every Symbol value is unique, even if it has the same description.
2. **Immutable:** Once created, a Symbol value cannot be changed.
3. **Usage:** Used as unique keys in objects to prevent property name collisions.

```
4. // Creating a Symbol  
5. let sym1 = Symbol("identifier");  
6. let sym2 = Symbol("identifier");  
7.  
8. console.log(sym1); // Symbol(identifier1)  
9. console.log(sym2); // Symbol(identifier2)  
10. console.log(sym1 === sym2); // false (each Symbol is unique)
```

7. BigInt (Introduced in ES11): Used to represent numbers larger than $2^{53} - 1$, which is the maximum safe integer in JavaScript. Denoted by 'n' at the end.

```
let largeNumber = 1234567890123456789012345678901234567890n; // 'n' denotes BigInt  
console.log(largeNumber);
```

2. Non-Primitive (or Complex) Data Types

1. Object data type: JavaScript, **objects** are a crucial **non-primitive data type**. Unlike primitive types (like numbers, strings, etc.), which hold a single value, **objects** are collections of key-value pairs. They allow you to store multiple values, methods, or other objects. It is reference type.

Characteristics of Objects

1. **Key-Value Pairs:** Objects store data as properties (keys) and their corresponding values.
2. **Mutable:** Objects can be modified after creation by adding, updating, or removing properties.
3. **Reference Type:** When assigned or compared, objects are referenced in memory, not copied.

Creating the Object:

1. **Using Object Literal Syntax:**

```
let Person = {  
  name : "Sourabh",  
  id : 12,  
  age : 22,  
  marks : 92.34  
}  
// values are accessed using keys  
console.log(Person.name);  
console.log(Person.marks);  
console.log(Person);
```

// Adding the function as a part and changing the values

```
let Person = {  
  name : "Sourabh",  
  id : 12,  
  age : 22,  
  marks : 92.34,  
  display : function () {  
    console.log("name: "+this.name+"\t Age: "+this.age+"\t id: "+this.id+"\t Marks: "+this.marks);  
  }  
}  
  
// values are accessed using keys  
console.log(Person.name);  
console.log(Person.marks);  
Person.display();  
  
// changing the values of keys inside object  
Person.marks = 94.34;  
Person.id = 101;  
  
Person.display();
```

2. Using Constructor Function:

*// Def of constructor - automatically creates entire construct for type and autodefine
// mentioned keys, then assigns the values passed*

```
function Person(name , age, id, marks) {  
  this.name = name;  
  this.age = age;  
  this.id = id;  
  this.marks = marks;  
}  
  
// Create Object  
let p1 = new Person("Anant", 21, 100, 98.34);  
  
// values are accessed using keys  
console.log(`name is ${p1.name}`);  
console.log('age:', p1.age);  
console.log("id is:" + p1.id);  
console.log("marks: ", p1.marks);
```

3. Using new Object() Syntax

```
// initially create the logical construct of object  
let student = new Object();  
// now choose the keys and assign the values, which will automatically insert the pairs in object.  
student.rno = 12;  
student.name = "Kiran";
```

```
student.marks = 85.23;  
console.log(`name: ${student.name} \t Roll No: ${student.rno} \t marks:${student.marks}`);
```

4. Using Class Syntax (Introduced in ES6):

```
// define class templet  
class student {  
    constructor(rno, name, marks) {  
        // assigning this.xxx will automatically creates the fields within class.  
        this.rno = rno;  
        this.name = name;  
        this.marks = marks;  
    }  
  
    display() {  
        console.log(`name: ${this.name} \t Roll No: ${this.rno} \t marks:${this.marks}`);  
        console.log("name:" + this["name"] + "\t Roll No:" + this["rno"] + "\t marks:" + this["marks"]);  
    }  
}  
  
let s1 = new student(11, "Akash", 98.34);  
console.log("name:" + s1["name"] + "\t Roll No:" + s1["rno"] + "\t marks:" + s1["marks"]);  
s1.display();
```

2. Arrays Data Type

In JavaScript, an **array** is a special type of object used to store multiple values in a single variable. Arrays are **ordered, iterable, and dynamic**, meaning they can grow and shrink in size. They allow you to store values of different data types, including numbers, strings, objects, and even other arrays.

Creating Arrays:

1. Using Array Literals (initialization of Array):

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

2. Using the new Array() Constructor:

```
let numbers = new Array(1, 2, 3, 4, 5);
```

3. Using Array.of() (Prevents Constructor Issues)

```
let numbers = Array.of(5); // Creates an array with one element [5]
```

4. Using Array.from() (Creates an Array from an Iterable)

```
let str = "hello";  
let charArray = Array.from(str); // ["h", "e", "l", "l", "o"]
```

Types of Values in JavaScript Arrays:

1. Primitive Data Types: Arrays can store primitive data types such as Number, String, Boolean etc.

```
let mixedArray = [42, "Hello", true, null, undefined, Symbol("id"), 9007199254740991n];
```

2. Objects and Complex Data Types: JavaScript arrays can store objects and functions

```
let objArray = [{ name: "Alice" }, { name: "Bob" }];  
let funcArray = [function () { return "Hello"; }, () => 42];
```

3. Nested Arrays (Multidimensional Arrays)

```
let matrix = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]
```

```
];  
console.log(matrix[1][2]); // Output: 6
```

Array Properties:

1. **length Property:** Returns the number of elements in the array.

```
let fruits = ["Apple", "Banana", "Cherry"];  
console.log(fruits.length); // Output: 3
```

2. **Arrays are Dynamic:** Arrays in JavaScript **can grow and shrink** dynamically.

```
let arr = [1, 2, 3];  
arr.push(4); // Adds 4  
arr.pop(); // Removes last element (4)
```

3. **Arrays are Zero-Indexed:** JavaScript arrays are **zero-based**, meaning indexing starts from 0:

```
let arr = ["A", "B", "C"];  
console.log(arr[0]); // Output: "A"  
console.log(arr[1]); // Output: "B"
```

Common Array Methods: JavaScript provides various methods to manipulate arrays

1. Adding and Removing Elements

- push(value): Adds to the end
- pop(): Removes from the end
- unshift(value): Adds to the beginning
- shift(): Removes from the beginning

```
let arr = [1, 2, 3];  
arr.push(4); // [1, 2, 3, 4]  
arr.pop(); // [1, 2, 3]  
arr.unshift(0); // [0, 1, 2, 3]  
arr.shift(); // [1, 2, 3]
```

2. Iterating Over Arrays

Using for() loop:

```
let arr = ["Nitin", "Lukesh", "Sagar", "Swapnil"];  
for(let i=0; i<arr.length; i++) {  
    console.log(i+ " - "+arr[i]);  
}
```

Using forEach():

```
let numbers = [1, 2, 3];  
numbers.forEach(num => console.log(num));
```

Program 1:

```
let arr = ["Nitin", "Lukesh", "Sagar", "Swapnil"]; // initialization  
console.log(arr); // will displays the entire array.
```

```
// As the array is indexed collection and counting starts from 0, you can access it as  
console.log("arr[0] - "+arr[0] );  
console.log("arr[1] - "+arr[1] );  
console.log("arr[2] - "+arr[2] );
```

```
// there is a property called length gives you number of array elements  
console.log("Element count in array: "+arr.length);
```

```
// Can be displayed using loop
console.log("Array element using for loop: ");
for(let i=0 ; i<arr.length ; i++) {
  console.log(i+ " - "+arr[i]);
}

// Array can be displayed using for each loop
console.log("Array element using forEach loop: ");
arr.forEach(n => console.log(n));

// adding element at end and removing it from end
arr.push("Kiran"); // insert at end
console.log("Array elements after push(): ");
console.log(arr);

arr.pop(); // removes from end, not compulsory to collect
console.log("Array elements after pop(): ");
console.log(arr);

let name = arr.pop(); // removed from end and collected in a variable
console.log("Array element popped: "+name);

// adding element at start and removing it from start
arr.unshift("swapnil"); // inserting 2 elements at start
arr.unshift("Keshav");
arr.unshift("Madhav");
console.log("Array elements after inserting at start: ");
console.log(arr);

arr.shift();
let str = arr.shift();
console.log("Array elements after inserting at start: ");
console.log(arr);
console.log("Array element shifted: "+str);
```

Program 2:

```
// creating the array using constructor
let ar1 = new Array(11,22,33,44,55);
console.log("Array Elements are: ");
console.log(ar1);

// Creating array using Array.from()
let str = "Good day";
let chararr = Array.from(str);
console.log("Array Elements are: ");
console.log(chararr);
```

```
//Creating array using Array.of()
let numbers = Array.of(2,5);
console.log("Array Elements are: ");
console.log(numbers);
```

```
// In JavaScript, Arrays may collect elements of different types also,
// Note that, Unlike C and C++, They are not collection of elements having same data type.
let mixedArray = [42, "Hello", true, null, undefined, Symbol("id"), 9007199254740991n];
console.log("Array Elements are: ");
console.log(mixedArray);
```

3. Functions data type: Functions are objects that can be executed. They allow code reuse and modular programming. Functions can be stored in variables, passed as arguments, and returned from other functions.

```
function test(name) {
    return "Hello, " + name + "!";
}
console.log(test("Vishwajeet")); // Output: Vishwajeet!
```

4. Date: The Date object is used to work with dates and times. It provides methods to retrieve, modify, and format date and time values.

```
let today = new Date();
console.log(today.toString()); // Output: Thu Jan 30 2025
```

There are some other Non-primitive / Advanced / Reference data types, which we will see in later chapters, Some of them are as listed below.

- **Maps:** A Map is a collection of key-value pairs where keys can be of any type. Unlike objects, maps maintain the order of their entries and allow any data type as a key.
- **Sets:** A Set is a collection of unique values (no duplicates). It is useful when storing unique elements efficiently.

Knowing the Data type of Variable:

Using typeof(): In JavaScript, typeof is an operator used to determine the type of a given value. It returns a string indicating the type of the operand.

```
// knowing the typeof literals
console.log(typeof 42); // "number"
console.log(typeof "hello"); // "string"
console.log(typeof true); // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null); // "object" (historical bug in JS)
console.log(typeof {}); // "object"
console.log(typeof []); // "object" (arrays are objects)
console.log(typeof function(){}); // "function"
console.log(typeof Symbol("id")); // "symbol"
console.log(typeof 10n); // "bigint"
```



```
// knowing the typeof variables
let x = 23.45;
console.log("type of x:"+typeof x);
let y = "hello";
console.log("type of x:"+typeof y);
let z = new Array(3,4,5);
console.log("type of x:"+typeof z);
```

Dynamic Typing in JavaScript:

JavaScript is a dynamically typed language, meaning a variable can hold any data type and change types during execution

```
let v1 = 10;    // Number
console.log("type of v1: "+typeof v1);

v1 = "Hello";  // String
console.log("type of v1: "+typeof v1);

v1 = true;     // Boolean
console.log("type of v1: "+typeof v1);
```

Type Conversion in JavaScript: Type conversion in JavaScript refers to the process of converting a value from one data type to another. JavaScript provides two types of type conversion.

1. Implicit Type Conversion (Type Coercion): JavaScript automatically converts data types when required. This typically happens in operations involving different types.

```
console.log(5 + "10"); // "510" (Number converted to String)
console.log("5" * 2);  // 10 (String converted to Number)
console.log("5" - 2);  // 3 (String converted to Number)
console.log(true + 1); // 2 (Boolean converted to Number)
console.log(false + 5); // 5 (Boolean converted to Number)
```

2. Explicit Type Conversion (Type Casting): JavaScript provides methods to manually convert data types.

String Conversion: Use String(), toString(), or template literals.

```
let num = 123;
console.log(String(num)); // "123"
console.log(num.toString()); // "123"
```

Number Conversion: Use Number(), parseInt(), or parseFloat().

```
console.log(Number("123")); // 123
console.log(parseInt("123.45")); // 123
console.log(parseFloat("123.45")); // 123.45
console.log(Number("abc")); // NaN (Not a Number)
```

Boolean Conversion: Values that are 0, "", null, undefined, and NaN convert to false, while others convert to true.

```
console.log(Boolean(0));    // false
console.log(Boolean(1));    // true
console.log(Boolean("Hello")); // true
console.log(Boolean(""));   // false
console.log(Boolean(null)); // false
```

Different ways of taking the input in JavaScript:

JavaScript provides various ways to handle data input depending on the context, such as user interaction in web applications or scripts running in a Node.js environment. Here are the primary ways of data input in JavaScript:

1. Input Through HTML Forms: In web development, input fields in HTML forms are the most common way to collect data from users. JavaScript can be used to access and process the input values.

```
<!DOCTYPE html>
<html>
<body>
  <form id="myForm">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name">
    <button type="button" onclick="getInput()">Submit</button>
  </form>

  <script>
    function getInput() {
      const name = document.getElementById("name").value; // Get input value
      console.log("Entered Value: "+name); // Output the value
      alert("You hvae entered: "+name);
    }
  </script>
</body>
</html>
```

2. Input Through prompt(): The prompt() method creates a pop-up dialog box to collect text input from the user. It pauses code execution until the user provides input or cancels the dialog.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Using Prompt</title>
<style>
  #msg1 {
    font-family: Arial, sans-serif;
    font-weight: bold;
    font-style: italic;
    font-size: 40px;
  }
</style>
</html>
```

```
        color: green;
    }
</style>
</head>
<body>
    <h2> Using the prompt to input the data in JavaScript </h2>

    <div id="msg" style="font-family: Arial, sans-serif; font-weight: bold; font-style: italic; font-size: 20px;
color: blue;"></div>

    <div id="msg1"></div>
    <script>

        // input using prompt
        let userName = prompt("What is your name?");
        console.log(`Hello, ${userName}!`); // output of console

        // display on page
        document.write(`Hello, ${userName}!`);

        document.getElementById("msg").innerHTML = `Hello, ${userName}!`;
        document.getElementById("msg1").innerHTML = `Hello, ${userName}!`;
    </script>
</body>
</html>
```

3. Input Through HTML Event Listeners: User input can also be captured by adding event listeners to input elements or other interactive HTML elements.

```
<!DOCTYPE html>
<html>
<body>
    <input type="text" id="userInput" placeholder="Type something...">
    <button id="submitButton">Submit</button>

    <script>
        var inputField = document.getElementById("userInput");
        var button = document.getElementById("submitButton");

        button.addEventListener("click", function () {
            var input = inputField.value;
            alert("You Entered: "+input);
        });
    </script>
</body>
</html>
```

Another example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Uppercase Form</title>
  <script>
    function convertToUpperCase(input) {
      input.value = input.value.toUpperCase();
    }

    function handleSubmit(event) {
      event.preventDefault(); // prevent actual form submission
      const input1 = document.getElementById("input1").value;
      const input2 = document.getElementById("input2").value;
      alert("Input 1: " + input1 + "\nInput 2: " + input2);
    }
  </script>
</head>
<body>
  <h2>Input Form</h2>
  <form onsubmit="handleSubmit(event)">
    <label for="input1">First Input:</label><br>
    <input type="text" id="input1" name="input1" onblur="convertToUpperCase(this)"><br><br>

    <label for="input2">Second Input:</label><br>
    <input type="text" id="input2" name="input2" onblur="convertToUpperCase(this)"><br><br>

    <button type="submit">Submit</button>
  </form>
</body>
</html>
```

4. Input Through Keyboard Events: Data can be collected as the user types, using keyboard events like keydown, keyup, or input.

```
<!DOCTYPE html>
<html>
<body>
  <input type="text" id="textInput" placeholder="Start typing...">
  <p id="output"></p>

  <script>
    var inputField = document.getElementById("textInput");
    var output = document.getElementById("output");

    inputField.addEventListener("input", function(event) {
      output.textContent = "You typed: " + event.target.value;
    });
  </script>
</body>
```

</html>

5. Input Through File Upload: Users can upload files using <input type="file"> elements. JavaScript can process these files using the FileReader API.

```
<!DOCTYPE html>
<html>
<body>
  <input type="file" id="fileInput">

  <script>
    const fileInput = document.getElementById("fileInput");
    fileInput.addEventListener("change", (event) => {
      const file = event.target.files[0];
      if (file) {
        console.log(`File name: ${file.name}`);
        console.log(`File size: ${file.size} bytes`);
      }
    });
  </script>
</body>
</html>
```

6. Input Through URL Parameters: Data can be passed via the URL's query string. JavaScript can extract and use these parameters.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Prefill Form from URL</title>
  <script>
    // Function to prefill the form inputs from URL parameters
    function prefillForm() {
      const params = new URLSearchParams(window.location.search);

      // Get parameters
      const name = params.get('name');
      const course = params.get('course');

      // Set values to inputs if parameters are present
      if (name) document.getElementById('name').value = name;
      if (course) document.getElementById('course').value = course;
      alert(`Data: ${name} and ${course}`);
    }

    window.onload = prefillForm; // Call on page load
  </script>
</head>
<body>
```

```
<h2>Student Registration</h2>
<form>
  <label for="name">Name:</label><br>
  <input type="text" id="name" name="name"><br><br>

  <label for="course">Course:</label><br>
  <input type="text" id="course" name="course"><br><br>

  <button type="submit">Submit</button>
</form>
</body>
</html>
```

7. Input Through Drag-and-Drop: JavaScript allows users to drag and drop elements or files into a target area. This can be used to collect input, such as uploading files.

```
<!DOCTYPE html>
<html>
<body>
  <div id="dropZone" style="border: 2px dashed #ccc; width: 200px; height: 100px;">
    Drop files here
  </div>

  <script>
    const dropZone = document.getElementById("dropZone");

    dropZone.addEventListener("dragover", (event) => {
      event.preventDefault();
    });

    dropZone.addEventListener("drop", (event) => {
      event.preventDefault();
      const files = event.dataTransfer.files;
      document.write(`You dropped ${files.length} file(s).`);
    });
  </script>
</body>
</html>
```

8. Input Through Command-Line Arguments (Node.js): In a Node.js environment, input can be captured via command-line arguments using the `process.argv` array.

```
// Run in Node.js
// Command: node script.js <val1> <val2>
// const args = process.argv.slice(); first two elements are node path and script path with name
const args = process.argv.slice(2); // eliminates first two elements
console.log(`Hello, ${args[0]}!`);
console.log(`Hello, ${args[1]}!`);
console.log(`Hello, ${args[2]}!`);
console.log(`Hello, ${args[3]}!`);
```


9. Input Through Web APIs: Data input can also come from external sources, such as web APIs, by using `fetch()` or other HTTP libraries.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Input via Web API</title>
  <script>
    function getUserInput() {
      // Using the prompt() Web API to get user input
      const name = prompt("Enter your name:");
      const course = prompt("Enter your course:");

      if (name && course) {
        alert("Welcome, " + name + "!\\nYou are enrolled in " + course + ".");
      } else {
        alert("Please provide valid inputs.");
      }
    }
  </script>
</head>
<body onload="getUserInput()">
  <h2>Web API Input Demo</h2>
  <p>Check the popup prompt when page loads.</p>
</body>
</html>
```

Example: Read and Display JSON Data

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Read JSON Data</title>
  <script>
    // Sample JSON data (like you'd get from an API)
    const studentData = {
      "name": "Yogesh",
      "course": "Java Full Stack",
      "duration": "6 months"
    };

    function showData() {
      // Accessing JSON values using dot notation
      const name = studentData.name;
      const course = studentData.course;
      const duration = studentData.duration;

      // Display in a paragraph
```

```
const output = `
  <strong>Name:</strong> ${name}<br>
  <strong>Course:</strong> ${course}<br>
  <strong>Duration:</strong> ${duration}
`;
document.getElementById("display").innerHTML = output;
}

window.onload = showData; // Call function when page loads
</script>
</head>
<body>
  <h2>Student Information</h2>
  <div id="display">Loading...</div>
</body>
</html>

//-----
Example: Fetch Data from JSONPlaceholder API
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Fetch Data from API</title>
<script>
  // Fetching posts from JSONPlaceholder API
  async function loadPosts() {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts');

      // Check if the fetch was successful
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }

      const posts = await response.json(); // Parse JSON response

      // Display the first 5 posts
      let output = '<h3>Latest Posts:</h3>';
      posts.slice(0, 5).forEach(post => {
        output += `
          <div>
            <h4>${post.title}</h4>
            <p>${post.body}</p>
          </div>
        `;
      });
    }
  }
  loadPosts();
</script>
</html>
```

```
document.getElementById("display").innerHTML = output;

} catch (error) {
  console.error("Error fetching data:", error);
}
}

window.onload = loadPosts; // Load posts when page loads
</script>
</head>
<body>
  <h2>Posts from JSONPlaceholder</h2>
  <div id="display">Loading posts...</div>
</body>
</html>
```

10. Input Through Sensors or Devices

- For advanced applications, inputs can be collected from devices like cameras, microphones, or GPS using browser APIs.

Operators in JavaScript:

Operators are used to process the data. JavaScript provides a wide range of operators to perform different types of operations. These operators can be categorized as follows:

1. **Arithmetic Operators** (+, -, *, /, %, **, ++, --)
2. **Assignment Operators** (=, +=, -=, *=, /=, %=, **=)
3. **Comparison Operators** (==, ===, !=, !==, <, <=, >, >=)
4. **Logical Operators** (&&, ||, !)
5. **Bitwise Operators** (&, |, ^, ~, <<, >>, >>>)
6. **Ternary (Conditional) Operator**(? :)
7. **Type Operators**(typeof, instanceof)
8. **Spread and Rest Operators** (ES6) (...)
9. **Nullish Coalescing Operator** (ES11)(??)
10. **Optional Chaining Operator** (ES11)(?.)

1. Arithmetic Operators: Arithmetic operators are used to perform mathematical calculations on numbers. JavaScript provides several arithmetic operators:

*// Arithmetic Operators +, -, *, /, %, **, ++, --*

```
// using +, -, *
let x=3;
let y=5.3;
let z = (10 + x)*y;
console.log("Answer z = "+z);
```

// using / - Returns a floating-point number if the division is not exact.

```
console.log("10/4 = "+10/4); // It will break the rule int <-> int -> int
console.log("10/5 = "+10/5); // it will gives the mathematical answer
console.log(`\n y/x is ${y/x}`);
console.log("123/10 is "+(123/10)); // gives the fractional value
console.log("3/10 is "+(3/10)); // gives the fractional value
console.log("3/0 is "+(3/0)); // If you divide by 0, JavaScript returns Infinity or -Infinity instead of throwing an error.
```

```
// using % - The modulus operator % returns the remainder after dividing the left operand by the right operand.
console.log("12%5: "+ (12%5) ); // will gives you mod after perfect division
```

```
console.log("-12%5: "+ (-12%5) );
console.log("12%-5: "+ (12%-5) ); // % can be operated on -ve values, sign of ans is sign of Numerator
console.log("-12%-5: "+ (-12%-5) );
```

```
console.log("12.4 % 5 : "+(12.4%5)); // % operator can be operated on fractional values
```

```
// Using ** operator: gives x to the power y (x^y)
let base = 2;
let power = 3;
let result = base ** power;
console.log("base**power: "+result); // Output: 8
```

```
//-----
```

```
// Using pre-post Operators i.e. ++ and --
```

```
let a = 5;
```

```
// Prefix Increment
console.log("Prefix Increment:");
console.log("++a = " + (++a)); // Increments 'a' first, then uses it. Output: 6
console.log("a = " + a); // Output: 6 (the updated value of 'a')
```

```
// Resetting the value of 'a'
a = 5;
```

```
// Postfix Increment
console.log("\nPostfix Increment:");
console.log("a++ = " + (a++)); // Uses 'a' first, then increments it. Output: 5
console.log("a = " + a); // Output: 6 (after the increment)
```

```
// Resetting the value of 'a'
a = 5;
```

```
// Prefix Decrement
console.log("\nPrefix Decrement:");
console.log("--a = " + (--a)); // Decrements 'a' first, then uses it. Output: 4
```

```
console.log("a = " + a); // Output: 4 (the updated value of 'a')
```

```
// Resetting the value of 'a'
```

```
a = 5;
```

```
// Postfix Decrement
```

```
console.log("\nPostfix Decrement:");
```

```
console.log("a-- = " + (a--)); // Uses 'a' first, then decrements it. Output: 5
```

```
console.log("a = " + a); // Output: 4 (after the decrement)
```

```
a = 5;
```

```
let b = 10;
```

```
let z = ++a + b--;
```

```
console.log("a = " + a + "\t b=" + b + "\t z = " + z); // executed in sequence pre -> calc -> assign -> post
```

2. Assignment Operators: In JavaScript, assignment operators are used to assign values to variables. They come in several forms, ranging from simple to compound operators. Here's an overview with examples:

- `=`: Assigns a value.
- `+=`: Adds a value to the variable.
- `-=`: Subtracts a value from the variable.
- `*=`: Multiplies the variable by a value.
- `/=`: Divides the variable by a value.
- `%=`: Assigns the remainder of division.
- `**=`: Raises the variable to a power.

```
let x = 10; // Assigns 10 to x
```

```
console.log(x); // Output: 10
```

```
// Assignment can be done in the shorthand manner as shown
```

```
x = 5;
```

```
x += 3; // Equivalent to x = x + 3
```

```
console.log(x); // Output: 8
```

```
x = 10;
```

```
x -= 4; // Equivalent to x = x - 4
```

```
console.log(x); // Output: 6
```

```
x = 17;
```

```
x %= 5; // Equivalent to x = x % 5
```

```
console.log(x); // Output: 2
```

```
x = 3;
```

```
x **= 2; // Equivalent to x = x ** 2
```

```
console.log(x); // Output: 9
```

3. Comparison Operators: Comparison operators in JavaScript are used to compare two values. They return a boolean value (true or false) based on the result of the comparison.

- `==`: Equal to (values are equal).
- `===`: Strict equal to (values and types are equal).
- `!=`: Not equal to (values are not equal).
- `!==`: Strict not equal to (values and types are not equal).
- `>`: Greater than.
- `<`: Less than.
- `>=`: Greater than or equal to.
- `<=`: Less than or equal to.

```
let a = 5;
let b = "5";
console.log(a == b); // Output: true (because values are equal, even though types are different) Compares two
values for equality, but does not check for type.
console.log(a === b); // Output: false (because types are different) Compares both the value and the type of the
two operands.
console.log(a != b); // Output: false (because values are equal, even though types are different)
console.log(`ans: ${a !== b}`); // Output: true (because types are different)

console.log(a > b); // Output: false
console.log(a < b); // Output: false
console.log(a >= b); // Output: true
console.log(a <= b); // Output: true
```

4. Logical Operators: logical operators are used to combine or manipulate boolean values (true and false). They are often used in conditional statements and loops to make decisions based on multiple conditions.

```
// Using &&, || and !
// Using &&
let a = true;
let b = false;
let result = a && b; // false (because b is false)
console.log(result); // Output: false

let x = 5;
let y = 10;
if (x > 0 && y > 0) {
  console.log("Both x and y are positive."); // Output: "Both x and y are positive."
}

// Using ||
x = 0;
y = 10;
if (x > 0 || y > 0) {
  console.log("At least one of x or y is positive."); // Output: "At least one of x or y is positive."
}
```



```
// Using !
let isLoggedIn = false;
if (!isLoggedIn) {
  console.log("Please log in."); // Output: "Please log in."
}
```

5. Bitwise Operators: bitwise operators are used to perform operations on the binary representations of numbers. They work at the bit level, manipulating the individual bits of integers. These operators are often used in low-level programming, such as cryptography, compression, and hardware control.

1. Bitwise AND (&)
2. Bitwise OR (|)
3. Bitwise XOR (^)
4. Bitwise NOT (~)
5. Left Shift (<<)
6. Right Shift (>>)
7. Unsigned Right Shift (>>>)

```
// Using bitwise operators
// using Bitwise AND(&): Returns 1 if both bits are 1; otherwise, returns 0.
let a = 5; // Binary: 0101
let b = 3; // Binary: 0011
let result = a & b; // Binary: 0001 (Decimal: 1)
console.log(result); // Output: 1
```

```
// Using Bitwise OR(|): Returns 1 if at least one of the bits is 1; otherwise, returns 0.
result = a | b; // Binary: 0111 (Decimal: 7)
console.log(result); // Output: 7
```

```
// Using Bitwise XOR(^): Returns 1 if the bits are different; otherwise, returns 0.
result = a ^ b; // Binary: 0110 (Decimal: 6)
console.log(result); // Output: 6
```

```
// Using Bitwise NOT(~): Inverts all the bits of the operand (i.e., 1 becomes 0 and 0 becomes 1).
result = ~a; // Binary: 1010 (Decimal: -6)
console.log(result); // Output: -6
```

```
// Using Left Shift (<<): Shifts the bits of the number to the left and fills in with 0s on the right.
// Equivalent to multiplying the number by 2^n, where n is the number of shifts.
result = a << 1; // Binary: 1010 (Decimal: 10)
console.log(result); // Output: 10
```

```
// Using Right Shift (>>): Shifts the bits of the number to the right and fills in with the sign bit (0 for positive, 1 for negative) on the left.
// Equivalent to dividing the number by 2^n and truncating the result.
a = 10;
result = a >> 1; // Binary: 0101 (Decimal: 5)
console.log(result); // Output: 5
```

```
// Using Unsigned Right Shift (>>>): Shifts the bits of the number to the right and fills in with 0s on the left, regardless of the sign.  
// This operator treats the number as an unsigned 32-bit integer.  
let c = -10; // Binary: 11111111111111111111111111110110 (32-bit representation)  
result = c >>> 1; // Binary: 0111111111111111111111111111011 (Decimal: 2147483643)  
console.log(result); // Output: 2147483643
```

6. Ternary (Conditional) Operator: The **conditional (ternary) operator** is a concise way to perform conditional logic. It is often used as a shorthand for the if-else statement. The operator takes three operands, hence the name "ternary."

Syntax: condition ? expressionIfTrue : expressionIfFalse

```
// Using ?: - ternary / if-then-else / conditional operator  
let age = 20;  
let message = age >= 18 ? "You are an adult" : "You are a minor";  
console.log(message); // Output: "You are an adult"
```

```
// find max from three  
let a = 10;  
let b = 20;  
let c = 15;  
let max = (a > b) ? (a > c ? a : c) : (b > c ? b : c);  
console.log("The maximum number is:", max); // Output: 20
```

7. Type Operators (typeof, instanceof):

- **typeof** → Determines the type of a value or variable.
- **instanceof** → Checks if an object is an instance of a specific class or constructor.

```
// Using typeof  
console.log(typeof 42); // Output: "number"  
console.log(typeof "Hello"); // Output: "string"
```

```
// Using instanceof  
class Car {}  
let myCar = new Car();  
console.log(myCar instanceof Car); // Output: true  
console.log(myCar instanceof Object); // Output: true (because all objects inherit from Object)  
console.log([] instanceof Array); // Output: true  
console.log(42 instanceof Number); // Output: false (primitives are not instances)
```

8. Spread (...) and Rest (...) Operators in JavaScript (ES6): JavaScript ES6 introduced the spread (...) and rest (...) operators, both represented by three dots (...). While they look the same, they serve different purposes depending on how they are used.

- **Spread (...) Operator:** The spread operator is used to expand elements of an array, object, or string into individual elements.

```
// Using spread(...)  
const numbers = [1, 2, 3];  
const copy = [...numbers]; // Creates a new copy of the array
```

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

```
const arr1 = [1, 2, 3];  
const arr2 = [4, 5, 6];
```

```
const merged = [...arr1, ...arr2]; // copy both in 3rd  
console.log(merged); // Output: [1, 2, 3, 4, 5, 6]
```

```
const word = "Hello";  
const letters = [...word]; // copies all character in new array  
console.log(letters); // Output: ['H', 'e', 'l', 'l', 'o']
```

```
const person = { name: "vinayak", age: 23 };  
const updatedPerson = { ...person, city: "New York" };
```

```
console.log(updatedPerson); // Output: { name: 'John', age: 30, city: 'New York' }
```

- **Rest (...) Operator:** The rest operator is used to gather multiple values into an array. It is mainly used in function parameters.

```
// Using rest  
// Using to collect variable length argument  
function sum(...numbers) {  
  let tot = 0;  
  for(let i=0 ; i< numbers.length ; i++ ) {  
    tot+=numbers[i];  
  }  
  return tot;  
}  
console.log(sum(1, 2, 3, 4)); // Output: 10  
console.log(sum(5, 10)); // Output: 15
```

```
// Using to collect remaining elements  
const [first, second, ...rest] = [10, 20, 30, 40, 50];
```

```
console.log(first); // Output: 10  
console.log(second); // Output: 20  
console.log(rest); // Output: [30, 40, 50]
```

9. Nullish Coalescing Operator (??) in JavaScript (ES11): It is used to provide a default value when a variable is **null or undefined**. Not allowed to use ?? with && or ||.

```
// Using Nullish Coalescing Operator  
// Providing a Default Value  
let name = null;  
let defaultName = "Guest";  
let result = name ?? defaultName;  
console.log(result); // Output: "Guest"
```

```
// Difference Between ?? and ||
let age = 0;
let defaultAge = 18;
console.log(age || defaultAge); // Output: 18 (because 0 is falsy)
console.log(age ?? defaultAge); // Output: 0 (because 0 is NOT null or undefined)
let newage;
console.log(newage ?? defaultAge); // Output: 18 (because newage is undefined)
```

```
// Using ?? with Function Parameters
```

```
function fun(user) {
  let name = user ?? "Stranger";
  console.log(`Hello, ${name}!`);
}
```

```
fun("Sourabh"); // Output: "Hello, Alice!"
fun(null); // Output: "Hello, Stranger!"
fun(undefined); // Output: "Hello, Stranger!"
```

```
// Chaining ??
let value = null;
let fallback1 = undefined;
let fallback2 = "Available";
console.log(value ?? fallback1 ?? fallback2); // Output: "Available"
```

10. Optional Chaining Operator (ES11)(?.): It allows safe access to deeply nested object properties without throwing an error if a property is null or undefined. If the property/method exists, it returns the value. If the property/method is null or undefined, it **returns undefined instead of throwing an error**.

```
// Optional Chaining Operator (?.)
// Accessing Nested Object Properties
let user = {
  name: "Ashish",
  address: { city: "New York", pin: 411045 }
};
console.log(user.name); // output: "Ashish"
console.log(user.address?.city); // Output: "New York"
console.log(user.address?.phone); // Output: undefined (No error)
```

```
// Accessing Nested Arrays
let students = {
  names: ["Amol", "Nisha"]
};
console.log(students.names?.[0]); // Output: "Amol"
console.log(students.names?.[3]); // Output: undefined
console.log(students.grades?.[1]); // Output: undefined (No error)
```

```
// Calling Methods with Optional Chaining
let person = {
```

```
sayHello: function() {  
  return "Hello!";  
}  
};  
console.log(person.sayHello?.()); // Output: "Hello!"  
console.log(person.sayBye?.()); // Output: undefined (No error)  
//Using ?. with null or undefined  
let car = null;  
console.log(car?.model); // Output: undefined (No error)
```

Truthy and Falsy in short with example and what is difference between ?? and ||:

What are Truthy and Falsy Values?

JavaScript evaluates **any value** in a boolean context (like inside an if condition).

Values are classified as either:

- **Truthy** → treated as true
- **Falsy** → treated as false

Falsy Values (Only 7)

These **7 values** are **falsy** in JavaScript:

Value	Meaning
false	Boolean false
0	Number zero
-0	Negative zero
""	Empty string
null	Absence of value
undefined	Variable not assigned
NaN	Not-a-Number

Anything **not on this list is Truthy**.

Truthy Values

All other values are **truthy**, such as:

- "Hello" (non-empty string)
- 1, -1, 3.14 (non-zero numbers)
- [] (empty array)
- {} (empty object)
- function() {} (functions)

Example: Truthy & Falsy in Action

```
let name1 = "" || "Guest";  
console.log(name1); // "Guest" ("" is falsy)  
let name2 = "" ?? "Guest";  
console.log(name2); // "" (not null or undefined)
```

- Use || to replace **any falsy** value.
- Use ?? to replace only **null or undefined**.

Precedence And Associativity of Operators In JavaScript:

operator precedence determines the order in which operators are evaluated in an expression, and operator associativity determines the order in which operators of the same precedence level are processed.

1. Operator Precedence in JavaScript

Operators with higher precedence are evaluated before operators with lower precedence. Below is a table of common JavaScript operators ordered by **precedence (from highest to lowest)**:

2. Operator Associativity in JavaScript

When multiple operators of **the same precedence level** appear in an expression, **associativity** determines the order of evaluation.

JavaScript Operator Precedence and Associativity Table

Precedence	Operator	Description	Associativity
20	()	Grouping (Parentheses)	N/A
19	. []	Member Access, Array Indexing	Left-to-Right
18	()	Function Call	Left-to-Right
17	new MyFunction()	new (with arguments)	Right-to-Left
16	new MyFunction	new (without arguments)	Right-to-Left
15	x++ x--	Postfix Increment/Decrement	N/A
14	++x --x	Prefix Increment/Decrement	Right-to-Left
14	+ - ! ~ typeof void delete	Unary Operators	Right-to-Left
13	**	Exponentiation	Right-to-Left
12	* / %	Multiplication, Division, Modulus	Left-to-Right
11	+ -	Addition, Subtraction	Left-to-Right
10	<< >> >>>	Bitwise Shift	Left-to-Right
9	< <= > >= in instanceof	Comparison Operators	Left-to-Right
8	== != === !==	Equality Operators	Left-to-Right
7	&	Bitwise AND	Left-to-Right
6	^	Bitwise XOR	Left-to-Right
5		Bitwise OR	Left-to-Right
4	&&	Logical AND	Left-to-Right
3	??	logical OR, nullish coalescing	Left-to-Right
2	? :	Ternary (Conditional)	Right-to-Left
1	` += -= *= /= %= <= >= >>= &= ^= **= `	Assignment Operators	Right-to-left
0	,	Comma Operator (Sequencing)	Left-to-Right

// JavaScript Operator Precedence and Associativity

// Mixed Operators

*let result = 2 + 3 * 4; // * having higher Precedence than +
console.log(result); // 14 (multiplication happens first)*

// Using Parentheses

*result = (2 + 3) * 4; // () having higher Precedence than any other
console.log(result);*

// Associativity of Assignment (=) - It is Right-to-Left

```
let a, b, c;  
a = b = c = 10;  
console.log(a, b, c); // Output: 10 10 10
```

```
// Combining with Arithmetic  
let x = 5;  
x += 3 * 2; // Equivalent to: x = x + (3 * 2)  
console.log(x); // Output: 11
```

```
// associativity is Left to Right  
console.log(10 - 5 - 2); // Output: 3 coz as associativity is Left To Right 10-5 calculated first and then 5-2  
console.log(10 - (5 - 2)); // Output: 7 coz Precedence goes to () first fives 3 and then 10 - 3
```

```
console.log(30 / 2 * 5); // Output: 75 - As same Precedence, refer the Precedence which is Left To Right, so  
calculate 30/2 first and then 15 * 5
```

Control Statements in JavaScript: Control statements are essential for flow control in JavaScript.

- **Conditionals:** if, if...else, Nesting of if...else, Ladder (if...else if...else), switch - for decision-making.
- **Loops:** for, while, do...while, for...in, for...of - for repetition.
- **Jump Statements:** break, continue, return, labels - for flow control.

Conditionals Statements: Conditional statements control the flow of execution by allowing a program to make decisions based on conditions. These statements execute different blocks of code depending on whether a given condition evaluates to true or false.

1. if Statement: The if statement executes a block of code only if a specified condition is true. If the condition is false, the code inside the block is skipped.

```
if (condition) {  
    // Executes if condition is true  
}
```

If we want to input the data in node, the simple option is, input the data as a command line argument. This is applicable only when you are using the JavaScript out of browser, If you are using the In-Browser JavaScript then prompt is suitable option for you.

When you prefer off-Browser JavaScript, and entering the data as a command line arguments, the data is in form of String and that we need to convert into Number type so that the comparison will be done and for that we need to use Number(<>).

```
const args = process.argv.slice(2);  
let x = Number(args[0]);  
let y = Number(args[1]);  
console.log(typeof(x));  
console.log(typeof(y));  
console.log("x: "+x+" \t y: "+y);
```

```
// find max using if()  
if(x>y){  
    console.log(`x=${x} is max`);
```



```
}  
if(y>x){  
  console.log(`y=${y} is max`);  
}
```

2. if...else Statement

The if...else statement executes one block of code if the condition is true, and another block if the condition is false.

```
if (condition) {  
  // Executes if condition is true  
} else {  
  // Executes if condition is false  
}
```

Example:

```
const args = process.argv.slice(2);  
let x = Number(args[0]);  
let y = Number(args[1]);  
console.log(typeof(x));  
console.log(typeof(y));  
console.log("x: "+x+" \t y: "+y);
```

// find max using if()

```
if(x>y){  
  console.log(`x=${x} is max`);  
}else {  
  console.log(`y=${y} is max`);  
}
```

3. Using Nesting of if() ... else: In the nesting If and/or if..else can be part of if and/or else ..

```
const args = process.argv.slice(2);  
let x = Number(args[0]);  
let y = Number(args[1]);  
let z = Number(args[2]);
```

```
if(x>y) {  
  if(x>z) {  
    console.log(`x = ${x} is Max`);  
  }  
  else {  
    console.log(`z = ${z} is Max`);  
  }  
}  
else {  
  if(y>z) {  
    console.log(`y = ${y} is Max`);  
  }  
}
```

```
else {  
    console.log(`z = ${z} is Max`);  
}  
}
```

4. if...else if...else Statement

The if...else if...else statement allows multiple conditions to be checked. It executes the first if block that evaluates to true. If none of the conditions are true, the else block runs.

```
if (condition1) {  
    // Code to execute if condition1 is true  
} else if (condition2) {  
    // Code to execute if condition2 is true  
} else {  
    // Code to execute if none of the conditions are true  
}
```

Example:

```
let score = 85;  
if (score >= 90) {  
    console.log("Grade: A");  
} else if (score >= 80) {  
    console.log("Grade: B");  
} else {  
    console.log("Grade: C");  
}
```

Carefully note that, we are writing the above code in node, considering that we are using it on server side, but can be written as a part of In-Browser JavaScript as shown below

```
<!DOCTYPE html>  
<head>  
    <meta charset="utf-8">  
    <title>JavaScript If Statement</title>  
</head>  
<body>  
    <script>  
        let now = new Date();  
        let dayOfWeek = now.getDay(); // Sunday - Saturday : 0 - 6  
        document.write("Day of week Today: "+dayOfWeek);  
        if(dayOfWeek == 5) {  
            document.write("Have a nice weekend!");  
        }  
    </script>  
</body>  
</html>
```

When we are using node, simple way to take the input data is using command line argument. But when we are going for In-Browse JavaScript the prompt is simple option for us. So above program can be written as

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title>JavaScript If Statement</title>
</head>
<body>
  <script>
    //let dayOfWeek = Number(prompt("Enter the Day Number: "));
    let dayOfWeek = parseInt(prompt("Enter the Day Number: "));

    document.write("Day of week Today: "+dayOfWeek);
    if(dayOfWeek == 5) {
      document.write("Have a nice weekend!");
    }
  </script>
</body>
</html>
```

Program to add two numbers in browser:

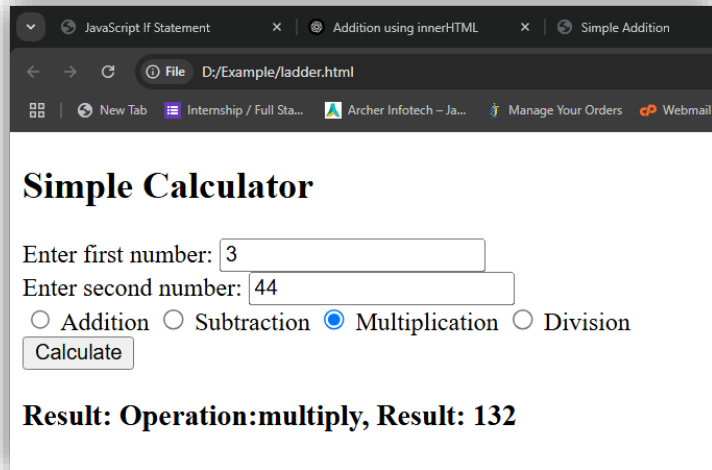
```
<!DOCTYPE html>
<head>
  <title>Simple Addition</title>
</head>
<body>
  <h2>Simple Addition</h2>
  <label for="num1">Enter first number:</label>
  <input type="number" id="num1">
  <br>
  <label for="num2">Enter second number:</label>
  <input type="number" id="num2">
  <br>
  <button onclick="calculateSum()">Add</button>
  <h3>Result: <span id="result"></span></h3>

  <script>
    function calculateSum() {
      let number1 = document.getElementById("num1").value;
      let number2 = document.getElementById("num2").value;
      let sum = parseFloat(number1) + parseFloat(number2);
      document.getElementById("result").innerHTML = sum;
    }
  </script>
</body>
</html>
```

Program to perform arithmetic operations:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Simple Calculator</title>
</head>
<body>
  <h2>Simple Calculator</h2>
  <label for="num1">Enter first number:</label>
  <input type="number" id="num1">
  <br>
  <label for="num2">Enter second number:</label>
  <input type="number" id="num2">
  <br>
  <input type="radio" name="operation" value="add" checked> Addition
  <input type="radio" name="operation" value="subtract"> Subtraction
  <input type="radio" name="operation" value="multiply"> Multiplication
  <input type="radio" name="operation" value="divide"> Division
  <br>
  <button onclick="calculateResult()">Calculate</button>
  <h3>Result: <span id="result"></span></h3>
<script>
  function calculateResult() {
    let number1 = parseFloat(document.getElementById("num1").value);
    let number2 = parseFloat(document.getElementById("num2").value);
    let operation = document.querySelector('input[name="operation"]:checked').value;

    let result;
    if (operation === "add") {
      result = number1 + number2;
    } else if (operation === "subtract") {
      result = number1 - number2;
    } else if (operation === "multiply") {
      result = number1 * number2;
    } else if (operation === "divide") {
      result = number2 !== 0 ? number1 / number2 : "Cannot divide by zero";
    }
    document.getElementById("result").innerHTML = `Operation:${operation}, Result: ${result}`;
  }
</script>
</body>
</html>
```



Using the Switch...Case Statement

The switch...case statement is an alternative to the if...else if...else statement, which does almost the same thing. The switch...case statement tests a variable or expression against a series of values until it finds a match, and then executes the block of code corresponding to that match. It's syntax is:

```
switch(x)
{
  case value1:
    // Code to be executed if x === value1
    break;
  case value2:
    // Code to be executed if x === value2
    break;
  ...
  default:
    // Code to be executed if x is different from all values
}
```

Consider the following example, which display the name of the day of the week.

```
<!DOCTYPE html>
<head>
  <title>JavaScript Switch Case Statement</title>
</head>
<body>
  <script>
    let d = new Date();
    switch(d.getDay()) {
      case 0:
        document.write("Today is Sunday.");
        break;
      case 1:
        document.write("Today is Monday.");
        break;
      case 2:
        document.write("Today is Tuesday.");
```

```
        break;
    case 3:
        document.write("Today is Wednesday.");
        break;
    case 4:
        document.write("Today is Thursday.");
        break;
    case 5:
        document.write("Today is Friday.");
        break;
    case 6:
        document.write("Today is Saturday.");
        break;
    default:
        document.write("No information available for that day.");
        break;
}
</script>
</body>
</html>
```

Some Facts about Switch():

- If break is omitted, execution continues to the next case. (Fall-through Behaviour)
- Using Multiple Cases Together should run the same code.
- JavaScript allows switch statements to work with boolean expressions.

```
let num = 10;
switch (true) { // switch (no<0) – Direct condition not allowed
    case (num < 0):
        console.log("Negative number");
        break;
    case (num === 0):
        console.log("Zero");
        break;
    case (num > 0):
        console.log("Positive number");
        break;
    default:
        console.log("Not a number");
}
```

- JavaScript allows switch with Strings

```
let browser = "Chrome";
switch (browser) {
    case "Chrome":
    case "Firefox":
    case "Edge":
        console.log("Supported browser.");
        break;
    case "Safari":
```

```
        console.log("Partially supported.");
        break;
    default:
        console.log("Unknown browser.");
    }
}
```

Using Loops in JavaScript:

Loops are used to avoid the code repetitions in the code. There are three different loops in JavaScript for(), while() and do...while(). Some of them can be used in modern ways. Let's start...!!

The while() Loop: This is the simplest looping statement provided by JavaScript. The while loop loops through a block of code as long as the specified condition evaluates to true. As soon as the condition fails, the loop is stopped. The generic syntax of the while loop is:

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

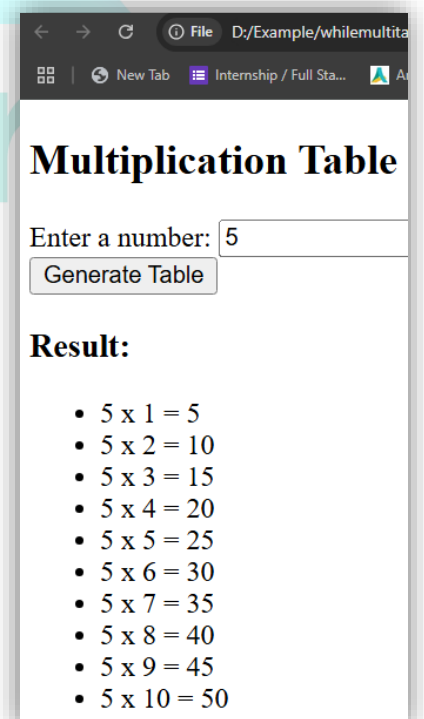
Example:

```
<!DOCTYPE html>
<body>
    <script>
        let i = 1;
        while(i <= 5) {
            document.write("<p>The number is " + i + "</p>");
            i++;
        }
    </script>
</body>
</html>
```

Example: Input number and display the multiplication table of that number

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Multiplication Table</title>
</head>
<body>
    <h2>Multiplication Table</h2>
    <label for="num">Enter a number:</label>
    <input type="number" id="num">
    <br>
    <button onclick="generateTable()">Generate Table</button>
    <h3>Result:</h3>
    <ul id="table"></ul>

    <script>
        function generateTable() {
```




```

    let number = parseInt(document.getElementById("num").value);
    let tableElement = document.getElementById("table");
    tableElement.innerHTML = "";

    let i = 1;
    while (i <= 10) {
        let listItem = document.createElement("li");
        listItem.textContent = `${number} x ${i} = ${number * i}`;
        tableElement.appendChild(listItem);
        i++;
    }
}
</script>
</body>
</html>

```

The do...while Loop

The do-while loop is a variant of the while loop, which evaluates the condition at the end of each loop iteration. With a do-while loop the block of code executed once, and then the condition is evaluated, if the condition is true, the statement is repeated as long as the specified condition evaluated to is true. The generic syntax of the do-while loop is:

```

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

```

Example:

```

<!DOCTYPE html>
<body>
    <script>
        let i = 1;
        do {
            document.write("<p>The number is " + i + "</p>");
            i++;
        }
        while(i <= 5);
    </script>
</body>
</html>

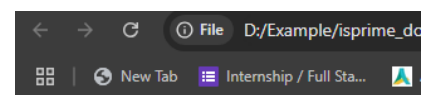
```

Example to check prime number.

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Prime Number Check</title>
</head>
<body>
    <h2>Prime Number Checker</h2>

```



Prime Number Checker

Enter a number:

Result: 21 is not a Prime Number

```
<label for="num">Enter a number:</label>
<input type="number" id="num">
<br>
<button onclick="checkPrime()">Check</button>
<h3>Result: <span id="result"></span></h3>
<script>
  function checkPrime() {
    let number = parseInt(document.getElementById("num").value);
    let isPrime = number > 1;
    let divisor = 2;

    do {
      if (number % divisor === 0 && number !== divisor) {
        isPrime = false;
        break;
      }
      divisor++;
    } while (divisor < number);

    document.getElementById("result").innerHTML = isPrime ? `${number} is a Prime Number` : `${number} is
not a Prime Number`;
  }
</script>
</body>
</html>
```

for Loop in JavaScript – In-Depth Explanation: A for loop in JavaScript is a control flow statement that repeatedly executes a block of code as long as a specified condition is true.

Syntax:

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

Example:

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

You can use a for loop to iterate through arrays.

```
let fruits = ["Apple", "Banana", "Cherry", "Mango"];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

The continue statement **skips the current iteration** and moves to the next.

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue; // Skips when i = 3
  }
  console.log(i);
}
```

The break statement **exits the loop immediately**.

```
for (let i = 1; i <= 5; i++) {  
  if (i === 3) {  
    break; // Stops loop when i = 3  
  }  
  console.log(i);  
}
```

Nested for Loop: A for loop inside another for loop.

```
for (let i = 1; i <= 3; i++) {  
  for (let j = 1; j <= 2; j++) {  
    console.log(`i = ${i}, j = ${j}`);  
  }  
}
```

An infinite loop **never stops** unless interrupted.

```
for (;;) {  
  console.log("This is an infinite loop");  
}
```

for...of: The **Modern Alternative**, for...of loop is simpler when working with arrays.

```
let colors = ["Red", "Green", "Blue"];  
for (let color of colors) {  
  console.log(color);  
}
```

for...in: Using for...in for Iterating Over Objects

```
let person = { name: "Sager", age: 23, city: "Pune" };  
for (let key in person) {  
  console.log(key + ": " + person[key]);  
}
```

forEach() in JavaScript : The forEach() method in JavaScript is used to iterate over arrays and execute a provided function once for each array element.

Syntax of forEach():

```
array.forEach( function(element, index, array) {  
  // Code to execute  
});
```

- **element** – The current element of the array.
- **index (optional)** – The index of the current element.
- **array (optional)** – The entire array being iterated.

Example:

```
let numbers = [11, 22, 33];  
numbers.forEach(function(num, index, arr) {  
  console.log("Element: " + num + ", Index: " + index + ", Array: " + arr);  
});
```

Can be used to display the only number as,

```
let numbers = [10, 20, 30, 40];  
numbers.forEach(function(num) {  
  console.log(num);  
});
```

Using index in forEach()

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

```
fruits.forEach(function(fruit, index) {  
  console.log(`Index ${index}: ${fruit}`);  
});
```

Using forEach() with Arrow Functions

```
let colors = ["Red", "Green", "Blue"];  
colors.forEach((color, index) => console.log(`Color ${index + 1}: ${color}`));
```

forEach() with Objects Inside Arrays

```
let students = [  
  { name: "John", age: 20 },  
  { name: "Sara", age: 22 },  
  { name: "Mike", age: 19 }  
];  
students.forEach((student) => {  
  console.log(`${student.name} is ${student.age} years old`);  
});
```

Unconditional control statements in JavaScript: Unconditional control statements in JavaScript are statements that alter the flow of execution without any condition. These include:

1. break Statement: Used to exit a loop or a switch statement prematurely. Execution continues after the loop or switch block.

```
for (let i = 0; i < 5; i++) {  
  if (i === 3) {  
    break; // Exits the loop when i is 3  
  }  
  console.log(i);  
}  
// Output: 0 1 2
```

2. continue Statement: Skips the current iteration of a loop and moves to the next iteration.

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

3. return Statement

- Exits a function and optionally returns a value.

```
function greet(name) {  
  return "Hello, " + name; // Exits the function and returns a value  
}  
console.log(greet("John"));  
// Output: Hello, John
```

Labelled break and continue in JavaScript:

1. Labelled break Statement: The break statement, when used with a label, allows you to exit not just the innermost loop but any specific loop.

```
outerLoop: for (let i = 0; i < 3; i++) {  
  for (let j = 0; j < 3; j++) {  
    if (i === 1 && j === 1) {  
      break outerLoop; // Breaks out of the outer loop  
    }  
    console.log(`i = ${i}, j = ${j}`);  
  }  
}  
console.log("Loop exited");  
// Output:  
// i = 0, j = 0  
// i = 0, j = 1  
// i = 0, j = 2  
// i = 1, j = 0  
// Loop exited
```

2. Labelled continue Statement: The continue statement with a label allows skipping an iteration of a **specific outer loop**, instead of just the innermost loop.

```
outerLoop: for (let i = 0; i < 3; i++) {  
  for (let j = 0; j < 3; j++) {  
    if (i === 1 && j === 1) {  
      continue outerLoop; // Skips the rest of the outer loop iteration  
    }  
    console.log(`i = ${i}, j = ${j}`);  
  }  
}  
console.log("Loop finished");  
// Output:  
// i = 0, j = 0  
// i = 0, j = 1  
// i = 0, j = 2  
// i = 1, j = 0  
// i = 2, j = 0  
// i = 2, j = 1  
// i = 2, j = 2  
// Loop finished
```