# ****Introduction to JavaScript****

## ****1 What is JavaScript?****

### ****Overview of JavaScript****

JavaScript is a high-level, interpreted programming language that was originally created to add interactivity to web pages. It is widely used to enhance user experience on the web by enabling dynamic content updates, interactive forms, animations, and much more. JavaScript can be executed in web browsers (client-side), and it also powers server-side development through environments like Node.js.

Key Features of JavaScript:

* **Interactivity**: JavaScript allows web pages to be interactive by responding to user actions like clicks, key presses, and mouse movements.
* **Event-Driven**: JavaScript is designed to handle events like user actions, server responses, etc.
* **Versatile**: It works on both the front-end (browser) and back-end (server) of applications.
* **Dynamic Typing**: Variables are not tied to specific data types, allowing flexibility in programming.

### ****Role of JavaScript in Web Development (Client-Side vs Server-Side)****

* **Client-Side JavaScript**: This is the traditional use of JavaScript in web development. The JavaScript code is executed in the user's web browser, which allows immediate feedback to the user without needing to make requests to the server. Examples include form validation, dynamic content loading, interactive maps, and animations.
* **Server-Side JavaScript**: JavaScript can also be used to build server-side applications, primarily using Node.js, a runtime environment that allows JavaScript to be executed outside the browser. This allows developers to use JavaScript for both client-side and server-side code, creating more cohesive development processes. Examples include APIs, server-side rendering, and real-time communication.

### ****Brief History of JavaScript****

JavaScript was created in 1995 by Brendan Eich at Netscape Communications. Initially called "LiveScript," it was renamed to "JavaScript" to capitalize on the popularity of Java at the time, although they are not related. Over the years, JavaScript has evolved with the introduction of features like AJAX, ES6 (ECMAScript 2015), and frameworks like React, Angular, and Vue.js, allowing JavaScript to become one of the most widely used programming languages in web development.

### ****JavaScript and HTML/CSS Integration****

JavaScript works alongside HTML and CSS to create modern web applications. HTML provides the structure of a webpage, CSS handles the design and layout, and JavaScript makes the page interactive.

* **HTML (Hypertext Markup Language)**: Provides the skeleton of the webpage by defining elements like headings, paragraphs, links, and images.
* **CSS (Cascading Style Sheets)**: Defines the visual appearance of HTML elements, including their colors, fonts, layout, and spacing.
* **JavaScript**: Manipulates HTML and CSS elements dynamically by responding to user interactions, modifying content, or changing styles.

### ****Example of JavaScript Integrating with HTML and CSS****

<!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>

<style>

#message { color: blue; }

</style>

</head>

<body>

<button id="changeButton">Change Message</button>

<p id="message">Hello, World!</p>

<script>

document.getElementById("changeButton").onclick = function() {

document.getElementById("message").innerText = "Message Changed!";

document.getElementById("message").style.color = "red";

};

</script>

</body>

</html>

In the example above, JavaScript changes the text and color of the paragraph when the button is clicked.

## ****2 Setting up the Development Environment****

### ****Text Editors/IDEs (e.g., VS Code, Sublime Text)****

To write JavaScript code, you need a text editor or an integrated development environment (IDE). These tools help you write, edit, and organize your code efficiently.

* **Visual Studio Code (VS Code)**: A free, open-source code editor developed by Microsoft. It supports JavaScript syntax highlighting, debugging, and extensions for various libraries and frameworks.
* **Sublime Text**: A lightweight text editor with fast performance and an elegant user interface. It supports many languages, including JavaScript, and provides features like multi-caret editing and split views.
* **Other Editors**: Some other editors include Atom, Brackets, and Notepad++.

### ****Browsers and Developer Tools (Chrome DevTools, Firefox Developer Tools)****

Modern web browsers come with built-in developer tools that allow you to inspect your JavaScript code, debug issues, and view console outputs. Two popular browsers with excellent developer tools are:

* **Chrome DevTools**: A set of tools built directly into Google Chrome that allows you to inspect elements, view network requests, debug JavaScript code, and more.
* **Firefox Developer Tools**: Similar to Chrome’s tools, Firefox provides an extensive suite for inspecting and debugging web pages and JavaScript.

### ****The Console: Understanding the Developer Console****

The console is a vital tool for JavaScript developers, providing a space for logging information, running JavaScript code, and debugging. In Chrome DevTools or Firefox Developer Tools, you can open the console by pressing F12 or right-clicking on the webpage and selecting "Inspect" → "Console." Common commands in the console include:

* **console.log()**: Outputs messages to the console, which is useful for debugging.
* console.log('Hello, JavaScript!');
* **console.error()**: Outputs error messages.
* **console.warn()**: Outputs warnings.

## ****3 Basic Syntax****

### ****Comments: Single-line vs Multi-line Comments****

JavaScript supports two types of comments:

* **Single-line comment**: Begins with // and continues to the end of the line.
* // This is a single-line comment
* **Multi-line comment**: Begins with /\* and ends with \*/. This type can span multiple lines.
* /\*
* This is a multi-line comment.
* It can span several lines.
* \*/

### ****Case Sensitivity in JavaScript****

JavaScript is a case-sensitive language, meaning that Variable, variable, and VARIABLE are considered distinct identifiers.

let variable = 10;

let Variable = 20;

console.log(variable); // Outputs: 10

console.log(Variable); // Outputs: 20

### ****Semicolons in JavaScript****

Semicolons are used to terminate statements in JavaScript. However, JavaScript's automatic semicolon insertion (ASI) allows you to omit semicolons in most cases, though it's generally a good practice to include them for clarity.

let x = 5; // Semicolon is optional but recommended

### ****Basic Structure of a JavaScript Program****

A basic JavaScript program typically follows this structure:

1. Declare variables and constants.
2. Write functions to perform tasks.
3. Execute code with statements or event handlers.

Example:

// Declare a variable

let name = "Alice";

// Function to greet the user

function greet() {

console.log("Hello, " + name + "!");

}

// Call the function

greet();

# ****Data Types and Variables****

## ****1 Primitive Data Types****

### ****Numbers (Integer, Float)****

JavaScript has a single type of number, which can represent both integers and floating-point numbers.

* **Integer**: Whole numbers without decimals. Examples: 5, -12, 100
* **Float**: Numbers with decimals. Examples: 3.14, -0.5, 2.71

let integer = 10; // Integer

let float = 3.14; // Float

### ****Strings (Creating strings, String concatenation)****

Strings in JavaScript are sequences of characters enclosed in either single quotes (') or double quotes ("). Template literals (backticks `) are also used for embedding expressions inside strings.

* **Creating strings**:

let name = "John"; // Using double quotes

let greeting = 'Hello'; // Using single quotes

let message = `Hello, ${name}!`; // Template literals

* **String concatenation**: You can join strings using the + operator.

let fullName = "John" + " " + "Doe";

console.log(fullName); // Outputs: John Doe

### ****Booleans (true/false)****

Booleans represent a binary value, either true or false. They are commonly used in control flow statements like conditionals.

let isActive = true;

let isComplete = false;

### ****Undefined and Null****

* **Undefined**: A variable that has been declared but not assigned a value is automatically set to undefined.

let value;

console.log(value); // Outputs: undefined

* **Null**: Represents the intentional absence of a value. It's an object type, even though it signifies "nothing."

let noValue = null;

console.log(noValue); // Outputs: null

### ****Understanding NaN (Not a Number)****

NaN is a special value in JavaScript used to represent a value that is not a valid number, often as the result of an invalid arithmetic operation.

let invalidNumber = "Hello" \* 2;

console.log(invalidNumber); // Outputs: NaN

## ****2 Variables in JavaScript****

### ****Declaring variables using var****

JavaScript provides the var keyword to declare variables, but it has some limitations compared to the newer let and const.

Sure! Here's a more detailed and structured explanation with improved examples for var, let, and const:

**1. Scope of var (Function-scoped)**

* **Explanation**: Variables declared using var are function-scoped. This means they are accessible throughout the entire function where they are declared, even before the line of declaration (due to hoisting).
* **Key Point**: var does not respect block-scoping (e.g., if or for blocks).

**Example:**

function testVarScope() {

if (true) {

var message = "Inside block";

}

console.log(message); // Outputs: Inside block (accessible outside the block)

}

testVarScope();

**Breakdown:**

* var message is accessible outside the if block, even though it was declared inside it. This happens because var is function-scoped, so message is accessible throughout the entire testVarScope() function.

**2. Scope of let (Block-scoped)**

* **Explanation**: Variables declared using let are block-scoped, meaning they are only accessible within the block ({}) where they are defined.
* **Key Point**: let does not allow access outside the block in which it's declared.

**Example:**

function testLetScope() {

if (true) {

let message = "Inside block";

console.log(message); // Outputs: Inside block (inside the block)

}

console.log(message); // ReferenceError: message is not defined (outside the block)

}

testLetScope();

**Breakdown:**

* let message is only accessible within the if block, and trying to access it outside that block throws a ReferenceError because let is block-scoped.

**3. Scope of const (Block-scoped)**

* **Explanation**: Similar to let, variables declared with const are block-scoped. However, const requires an initialization at the time of declaration and prevents reassignment.
* **Key Point**: Once a const variable is assigned, its value cannot be changed.

**Example:**

function testConstScope() {

if (true) {

const message = "Inside block";

console.log(message); // Outputs: Inside block (inside the block)

}

console.log(message); // ReferenceError: message is not defined (outside the block)

}

testConstScope();

**Breakdown:**

* const message is block-scoped, so it is only accessible within the if block. Attempting to access it outside of the block results in a ReferenceError.

**4. Key Differences Between var, let, and const**

* **Hoisting**:
  + var is hoisted to the top of its function and is accessible even before its declaration (undefined).
  + let and const are hoisted but are not accessible until they are defined (they are in a "temporal dead zone").

**Example (Hoisting):**

function hoistingExample() {

console.log(aVar); // Outputs: undefined (hoisted)

// console.log(aLet); // ReferenceError: Cannot access 'aLet' before initialization

// console.log(aConst); // ReferenceError: Cannot access 'aConst' before initialization

var aVar = "var";

let aLet = "let";

const aConst = "const";

}

hoistingExample();

* **Reassignability**:
  + Variables declared with var and let can be reassigned.
  + Variables declared with const cannot be reassigned once they have been assigned a value.

**Example (Reassigning):**

function reassignmentExample() {

var aVar = "Hello";

let aLet = "World";

const aConst = "Immutable";

aVar = "Reassigned var"; // Reassignable

aLet = "Reassigned let"; // Reassignable

// aConst = "Reassigned const"; // Error: Assignment to constant variable

console.log(aVar); // Outputs: Reassigned var

console.log(aLet); // Outputs: Reassigned let

console.log(aConst); // Outputs: Immutable (cannot be reassigned)

}

reassignmentExample();

## ****3 Type Conversion****

### ****Implicit and explicit type conversion****

JavaScript can automatically convert one type to another, known as **implicit type conversion** or **type coercion**. You can also manually convert types using **explicit conversion**.

* **Implicit**:

let num = 10;

let str = "The number is " + num; // Implicit conversion of number to string

console.log(str); // Outputs: "The number is 10"

* **Explicit**:

let stringNum = "42";

let number = Number(stringNum); // Explicit conversion to number

console.log(number); // Outputs: 42

### ****Converting to Number, String, and Boolean****

* **Number**: Use Number() to convert a value to a number.

let str = "25";

let num = Number(str); // Converts string to number

* **String**: Use String() to convert a value to a string.

let num = 123;

let str = String(num); // Converts number to string

* **Boolean**: Use Boolean() to convert a value to a boolean.

let value = 0;

let boolValue = Boolean(value); // Converts 0 to false

### ****parseInt(), parseFloat(), toString()****

* **parseInt()**: Converts a string to an integer.

let str = "10.5";

let intVal = parseInt(str); // Outputs: 10

* **parseFloat()**: Converts a string to a floating-point number.

let str = "10.5";

let floatVal = parseFloat(str); // Outputs: 10.5

* **toString()**: Converts a number to a string.

let num = 42;

let numStr = num.toString(); // Outputs: "42"

## ****2.4 Type Checking****

### ****Using typeof operator****

The typeof operator is used to check the type of a variable or expression.

let number = 42;

console.log(typeof number); // Outputs: "number"

### ****NaN and isNaN() function****

* **NaN** is used to represent an invalid number.
* **isNaN()** is a function that checks whether a value is NaN.

let value = "abc";

console.log(isNaN(value)); // Outputs: true (since "abc" is not a valid number)

### ****Object data types: Array, Object, and Function****

In addition to primitive data types, JavaScript also has three primary object types:

* **Array**: A collection of ordered elements.

let arr = [1, 2, 3];

console.log(Array.isArray(arr)); // Outputs: true

* **Object**: A collection of key-value pairs.

let person = { name: "John", age: 25 };

console.log(person.name); // Outputs: John

* **Function**: A block of reusable code that can be invoked.

function greet() {

console.log("Hello!");

}

greet(); // Outputs: Hello!

# ****Operators****

## ****1 Arithmetic Operators****

Arithmetic operators perform mathematical operations on numbers.

* **Addition (+)**: Adds two values.
* **Subtraction (-)**: Subtracts one value from another.
* **Multiplication (\*)**: Multiplies two values.
* \*\*Division (/) \*\*: Divides one value by another.
* **Modulus (%)**: Returns the remainder of a division.
* **Increment (++)**: Increases a value by 1.
* **Decrement (--)**: Decreases a value by 1.
* **Exponentiation (**\*\*): Raises a number to the power of another (not part of ES6 but works in some older engines).

let num1 = 10, num2 = 2;

console.log(num1 + num2); // Addition: 12

console.log(num1 % num2); // Modulus: 0

## ****2 Assignment Operators****

Assignment operators assign values to variables.

* **Simple assignment (=)**: Assigns the value on the right to the variable on the left.

let x = 5;

* **Compound assignment operators**: Combine an arithmetic operation with assignment.

let x = 10;

x += 5; // x = x + 5, now x is 15

x \*= 2; // x = x \* 2, now x is 30

## ****3 Comparison Operators****

Comparison operators compare two values and return true or false.

* **Equality (==, ===)**: == compares values, === compares values and types.
* **Inequality (!=, !==)**: != compares values, !== compares values and types.
* **Greater than (>) and Less than (<)**: Compares two values to check which is larger or smaller.
* **Greater than or equal (>=), Less than or equal (<=)**: Compares two values for equality or greater/smaller.

console.log(5 === 5); // true

console.log(5 == '5'); // true

console.log(5 === '5'); // false

## ****4 Logical Operators****

Logical operators perform logical operations on boolean values.

* **AND (&&)**: Returns true if both operands are true.
* **OR (||)**: Returns true if either operand is true.
* **NOT (!)**: Inverts the boolean value.

console.log(true && false); // false

console.log(true || false); // true

console.log(!true); // false

## ****5 Ternary Operator****

The ternary operator is a shorthand for if statements. It evaluates a condition and returns one of two values based on the result.

### ****Syntax:****

condition ? value\_if\_true : value\_if\_false;

### ****Example:****

let age = 18;

let canVote = (age >= 18) ? "Yes" : "No";

console.log(canVote); // Outputs: Yes

**4: Control Flow and Functions**

**1: Conditional Statements**

1. **Using if, else if, else**
   * **Syntax:**

if (condition) {

// Executes this block if the condition is true

} else if (anotherCondition) {

// Executes this block if the first condition is false and another condition is true

} else {

// Executes this block if no previous conditions were true

}

* + **Example:**

let age = 18;

if (age < 18) {

console.log("You are a minor.");

} else if (age === 18) {

console.log("You just became an adult.");

} else {

console.log("You are an adult.");

}

1. **The switch Statement**
   * **Syntax:**

switch (expression) {

case value1:

// Executes this block if expression === value1

break;

case value2:

// Executes this block if expression === value2

break;

default:

// Executes this block if no case matches the expression

}

* + **Example:**

let day = 3;

switch (day) {

case 1:

console.log("Monday");

break;

case 2:

console.log("Tuesday");

break;

case 3:

console.log("Wednesday");

break;

default:

console.log("Invalid day");

}

1. **Ternary Operator**

Syntax:

condition ? expressionIfTrue : expressionIfFalse;

Example:

let age = 20;

let result = (age >= 18) ? "Adult" : "Minor";

console.log(result); // Outputs: Adult

**Lesson 4.2: Loops**

1. **for Loop**
   * **Syntax:**

for (let i = 0; i < condition; i++) {

// Executes for each iteration as long as condition is true

}

* + **Example:**

for (let i = 0; i < 5; i++) {

console.log(i); // Outputs: 0, 1, 2, 3, 4

}

1. **while Loop**
   * **Syntax:**

while (condition) {

// Executes as long as condition is true

}

* + **Example:**

let i = 0;

while (i < 5) {

console.log(i); // Outputs: 0, 1, 2, 3, 4

i++;

}

1. **do-while Loop**
   * **Syntax:**

do {

// Executes at least once before checking condition

} while (condition);

* + **Example:**

let i = 0;

do {

console.log(i); // Outputs: 0, 1, 2, 3, 4

i++;

} while (i < 5);

1. **Looping with break and continue**
   * **break**: Exits the loop entirely when a condition is met.
   * **continue**: Skips the current iteration and continues with the next iteration of the loop.
   * **Example using break:**

for (let i = 0; i < 10; i++) {

if (i === 5) {

break; // Exits the loop when i reaches 5

}

console.log(i); // Outputs: 0, 1, 2, 3, 4

}

* + **Example using continue:**

for (let i = 0; i < 5; i++) {

if (i === 3) {

continue; // Skips the current iteration when i equals 3

}

console.log(i); // Outputs: 0, 1, 2, 4

}

**3: Functions**

1. **Declaring Functions Using the function Keyword**
   * **Syntax:**

function functionName(parameters) {

// Function body

}

* + **Example:**

function greet(name) {

console.log("Hello, " + name);

}

greet("John"); // Outputs: Hello, John

1. **Function Parameters and Return Values**
   * **Syntax:**

function functionName(param1, param2) {

return param1 + param2;

}

* + **Example:**

function add(a, b) {

return a + b;

}

console.log(add(3, 4)); // Outputs: 7

1. **Function Scope and Variable Scope**
   * **Explanation**: A function has its own scope for variables, and any variables declared inside a function are not accessible outside it.
   * **Example:**

function testScope() {

let x = 10;

console.log(x); // Outputs: 10

}

console.log(x); // ReferenceError: x is not defined

1. **Function Expressions (Anonymous Functions)**
   * **Syntax:**

const func = function() {

// Anonymous function body

};

* + **Example:**

const multiply = function(a, b) {

return a \* b;

};

console.log(multiply(2, 3)); // Outputs: 6

**4: Recursion**

1. **What is Recursion?**
   * **Explanation**: Recursion occurs when a function calls itself to solve a smaller version of the problem until a base case is reached.
2. **Writing Simple Recursive Functions**
   * **Example:**

function factorial(n) {

if (n === 0) {

return 1;

}

return n \* factorial(n - 1);

}

console.log(factorial(5)); // Outputs: 120

**5: Working with Data Structures**

**1: Arrays**

1. **Declaring Arrays**
   * **Syntax:**

let arr = [element1, element2, element3];

* + **Example:**

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

1. **Accessing and Modifying Array Elements**
   * **Syntax:**

arr[index] = newValue; // Modify an element

let element = arr[index]; // Access an element

* + **Example:**

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

fruits[1] = "orange"; // Modify element at index 1

console.log(fruits[1]); // Outputs: orange

1. **Array Methods**

push(): Adds one or more elements to the end of an array.

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

shift(): Removes the first element of an array.

unshift(): Adds one or more elements to the beginning of an array.

* + **Example:**

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

fruits.push("cherry"); // Adds cherry to the end

console.log(fruits); // Outputs: ["apple", "banana", "cherry"]

1. **Looping Through Arrays**
   * **for Loop**:

for (let i = 0; i < arr.length; i++) {

console.log(arr[i]);

}

* + **for-in Loop**:

for (let index in arr) {

console.log(arr[index]);

}

* + **for-of Loop**:

for (let value of arr) {

console.log(value);

}

**2: Objects**

1. **Declaring Objects Using {} Syntax**
   * **Syntax:**

let obj = { key1: value1, key2: value2 };

* + **Example:**

let person = { name: "John", age: 30 };

1. **Accessing and Modifying Object Properties**
   * **Example:**

console.log(person.name); // Outputs: John

person.age = 31; // Modify age property

console.log(person.age); // Outputs: 31

1. **for-in Loop with Objects**
   * **Example:**

for (let key in person) {

console.log(key + ": " + person[key]);

}

1. **Object Methods and the this Keyword**
   * **Example:**

let person = {

name: "John",

greet: function() {

console.log("Hello, " + this.name);

}

};

person.greet(); // Outputs: Hello, John

**Lesson 5.3: Array and Object Methods**

1. **Array Methods**
   * concat(): Merges two or more arrays.
   * slice(): Returns a shallow copy of a portion of an array.
   * splice(): Adds/removes elements from an array.
   * join(): Joins all elements of an array into a string.
   * reverse(): Reverses the order of elements in an array.
   * **Example:**

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

let newFruits = fruits.concat(["cherry", "date"]);

console.log(newFruits); // Outputs: ["apple", "banana", "cherry", "date"]

1. **Object Methods**
   * hasOwnProperty(): Checks if an object has a specific property.
   * Object.keys(): Returns an array of an object's keys.
   * Object.values(): Returns an array of an object's values.
   * **Example:**

let person = { name: "John", age: 30 };

console.log(person.hasOwnProperty("name")); // Outputs: true

console.log(Object.keys(person)); // Outputs: ["name", "age"]

console.log(Object.values(person)); // Outputs: ["John", 30]

**6: Error Handling**

**1: Error Handling**

1. **try, catch, finally**
   * **Explanation**: JavaScript provides the try, catch, and finally blocks to handle errors gracefully and manage control flow when an error occurs.
   * **Syntax**:

try {

// Code that may throw an error

} catch (error) {

// Code to handle the error

} finally {

// Code that will run regardless of an error occurring

}

* + **Example**:

try {

let result = 10 / 0;

if (!isFinite(result)) throw "Division by zero error!";

} catch (error) {

console.log("Error: " + error); // Outputs: Error: Division by zero error!

} finally {

console.log("This will always run.");

}

* + **Explanation**:
    - try: The block that contains the code that may throw an error.
    - catch: If an error occurs in the try block, control is transferred to the catch block, which handles the error.
    - finally: This block will always execute, regardless of whether an error was thrown or not, and is typically used for cleanup operations.

1. **Throwing Errors with throw**
   * **Explanation**: The throw statement allows you to create custom errors in JavaScript. You can throw an error manually by using throw followed by an error message or an error object.
   * **Syntax**:
   * throw new Error("Custom error message");
   * **Example**:

function checkAge(age) {

if (age < 18) {

throw new Error("Age must be 18 or older.");

} else {

console.log("You are allowed.");

}

}

try {

checkAge(16);

} catch (e) {

console.log(e.message); // Outputs: Age must be 18 or older.

}

1. **The Error Object**
   * **Explanation**: The Error object is the base object for all errors in JavaScript. It provides a message property that can describe the error, and a stack trace that shows where the error occurred.
   * **Example**:

try {

throw new Error("Something went wrong!");

} catch (error) {

console.log(error.message); // Outputs: Something went wrong!

console.log(error.stack); // Outputs the stack trace where the error occurred

}

**Code Example:**

function demoTryCatchFinally() {

try {

console.log("Start of try block");

// Intentionally causing an error (dividing by zero)

let result = 10 / 0;

if (!isFinite(result)) {

throw new Error("Cannot divide by zero!"); // Throwing custom error

}

console.log("End of try block");

} catch (error) {

console.log("Catch block executed: " + error.message); // Handling the error

} finally {

console.log("Finally block executed: This code runs no matter what.");

}

}

demoTryCatchFinally();

**Explanation:**

1. **try block**: This is where you place the code that you want to test for errors. In this case, we're intentionally causing a division by zero to trigger an error.
2. **catch block**: If any error occurs inside the try block, it is caught in the catch block. You can then handle the error (in this example, printing the error message).
3. **finally block**: This block is always executed, whether or not an error occurred in the try block. It is typically used for clean-up operations (like closing file handles, releasing resources, etc.).

**Sample Output:**

Start of try block

Catch block executed: Cannot divide by zero!

Finally block executed: This code runs no matter what.

**7: Working with the DOM (Document Object Model)**

**1: Introduction to the DOM**

1. **What is the DOM?**
   * **Explanation**: The **DOM (Document Object Model)** is a programming interface for web documents. It represents the structure of a web page as a tree of nodes, where each node corresponds to part of the page, such as elements, attributes, and text.
   * The DOM allows JavaScript to interact with HTML and CSS, enabling dynamic changes to content, structure, and styles of web pages.
   * **Example**: The HTML document:

<html>

<body>

<div id="content">Hello, World!</div>

</body>

</html>

The DOM structure will look something like:

Document

├── html

│ └── body

│ └── div (id="content")

│ └── text ("Hello, World!")

1. **The Structure of the DOM (Document Tree)**
   * **Explanation**: The DOM is structured as a tree, with each element, attribute, and piece of text represented as a node. The root of the tree is the document node, and each HTML element is a child of the root or other elements.
   * **Example**: A sample document tree for the following HTML:

<div>

<p>Hello</p>

<p>World</p>

</div>

Would look like this:

Document

└── div

├── p ("Hello")

└── p ("World")

1. **Accessing DOM Elements with getElementById(), getElementsByClassName(), getElementsByTagName()**
   * **getElementById()**: Access an element by its unique id attribute.
     + **Syntax**:

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

* + - **Example**:

let divElement = document.getElementById("content");

console.log(divElement); // Accesses the element with id="content"

* + **getElementsByClassName()**: Access all elements with a specific class.
    - **Syntax**:

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

* + - **Example**:

let items = document.getElementsByClassName("list-item");

console.log(items); // Accesses all elements with class="list-item"

* + **getElementsByTagName()**: Access all elements of a specific tag.
    - **Syntax**:

let elements = document.getElementsByTagName("tagName");

* + - **Example**:

let paragraphs = document.getElementsByTagName("p");

console.log(paragraphs); // Accesses all <p> elements in the document

**2: Manipulating the DOM**

1. **Modifying Element Content Using innerHTML, textContent**
   * **innerHTML**: Allows you to change the content of an element, including HTML tags.
     + **Example**:

let div = document.getElementById("content");

div.innerHTML = "<p>New content with <strong>HTML</strong> tags</p>";

* + **textContent**: Changes the text content of an element, without any HTML tags.
    - **Example**:

let div = document.getElementById("content");

div.textContent = "New plain text content without HTML tags.";

1. **Changing Element Styles Dynamically**
   * **Explanation**: You can modify the CSS styles of DOM elements directly via JavaScript by accessing the style property.
   * **Example**:

let div = document.getElementById("content");

div.style.backgroundColor = "lightblue"; // Changes background color

div.style.fontSize = "20px"; // Changes font size

1. **Handling Events (onclick, onmouseover, etc.)**
   * **Explanation**: DOM elements can trigger actions when certain events occur, such as mouse clicks, hovering, key presses, etc. You can attach event handlers to elements.
   * **Example**:

onclick:

let button = document.getElementById("myButton");

button.onclick = function() {

alert("Button clicked!");

};

onmouseover:

let div = document.getElementById("hoverArea");

div.onmouseover = function() {

div.style.backgroundColor = "yellow"; // Changes background color when hovered

};

1. **Event Listeners (addEventListener())**
   * **Explanation**: The addEventListener() method is a more flexible way to attach events to elements. Unlike event attributes like onclick, it allows multiple events to be bound to the same element and supports various event types.
   * **Syntax**:

element.addEventListener(event, callback);

* + **Example**:

let button = document.getElementById("myButton");

button.addEventListener("click", function() {

alert("Button clicked with addEventListener!");

});

**HTML + JavaScript**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Simple Event Handling</title>

</head>

<body>

<h1>Click the button to change the text!</h1>

<button id="myButton">Click Me</button>

<p id="myParagraph">This is the original text.</p>

<script>

// Get the button and paragraph elements

const button = document.getElementById('myButton');

const paragraph = document.getElementById('myParagraph');

// Define the event handler function

function changeText() {

paragraph.textContent = "The text has been changed!";

}

// Add an event listener to the button for the 'click' event

button.addEventListener('click', changeText);

</script>

</body>

</html>

**Explanation:**

1. **HTML Structure**:
   * A heading (<h1>), a button (<button>), and a paragraph (<p>) are defined in the HTML.
   * The button has an ID of myButton and the paragraph has an ID of myParagraph.
2. **JavaScript**:
   * We use document.getElementById to get references to the button and paragraph elements.
   * A function changeText is defined, which changes the text content of the paragraph when called.
   * We use addEventListener to listen for the click event on the button. When the button is clicked, the changeText function is executed, changing the paragraph's text.

**How it works:**

* Initially, the paragraph says "This is the original text."
* When you click the button, the paragraph text changes to "The text has been changed!"

**Output:**

When you open this page in a browser, clicking the "Click Me" button will change the paragraph's text.

**Example: Changing the background color of a paragraph when the mouse hovers over it**

**HTML + JavaScript**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Mouseover Event Example</title>

</head>

<body>

<h1>Hover over the paragraph to change its background color!</h1>

<p id="myParagraph">Hover over this text to change its background color.</p>

<script>

// Get the paragraph element

const paragraph = document.getElementById('myParagraph');

// Add an event listener to the paragraph for the 'mouseover' event

paragraph.addEventListener('mouseover', function() {

// Change the background color of the paragraph when mouse is over it

paragraph.style.backgroundColor = 'lightblue';

});

// Add an event listener to remove the background color when mouse leaves the paragraph

paragraph.addEventListener('mouseout', function() {

// Reset the background color when the mouse leaves the paragraph

paragraph.style.backgroundColor = '';

});

</script>

</body>

</html>

**Explanation:**

1. **HTML Structure**:
   * A heading (<h1>) and a paragraph (<p>) are defined in the HTML. The paragraph has an ID of myParagraph.
2. **JavaScript**:
   * We access the paragraph element using document.getElementById.
   * An event listener for the **mouseover** event is added to the paragraph. This event triggers when the mouse moves over the paragraph, and we change the background color to lightblue.
   * An event listener for the **mouseout** event is added to reset the background color when the mouse leaves the paragraph.

**How it works:**

* When you hover the mouse over the paragraph, the background color changes to light blue.
* When you move the mouse away from the paragraph, the background color is reset to its original state.

**Output:**

When you hover over the paragraph text, its background color will change. Moving the mouse away will reset the background color.

Here’s a list of some commonly used events categorized by type:

**1. Mouse Events:**

These events are triggered by mouse actions.

* **click**: Fired when the user clicks on an element.
* **dblclick**: Fired when the user double-clicks on an element.
* **mousedown**: Fired when the mouse button is pressed down over an element.
* **mouseup**: Fired when the mouse button is released over an element.
* **mousemove**: Fired when the mouse pointer is moved within an element.
* **mouseover**: Fired when the mouse pointer enters an element.
* **mouseout**: Fired when the mouse pointer leaves an element.
* **mouseenter**: Fired when the mouse pointer enters an element (does not bubble).
* **mouseleave**: Fired when the mouse pointer leaves an element (does not bubble).

**2. Keyboard Events:**

These events are triggered by keyboard actions.

* **keydown**: Fired when a key is pressed down.
* **keypress**: Fired when a key is pressed and held (deprecated in favor of keydown and keyup).
* **keyup**: Fired when a key is released.

**3. Form Events:**

These events are used with form elements (like <input>, <textarea>, <select>, etc.).

* **submit**: Fired when a form is submitted.
* **reset**: Fired when a form is reset.
* **focus**: Fired when an element (input field) gains focus.
* **blur**: Fired when an element (input field) loses focus.
* **change**: Fired when the value of an input field is changed.
* **input**: Fired when the value of an input field is modified (e.g., typing in a text field).
* **select**: Fired when some text in a <textarea> or <input> field is selected.

**4. Window Events:**

These events are related to the browser window or document.

* **load**: Fired when the page has finished loading.
* **resize**: Fired when the window is resized.
* **scroll**: Fired when the page is scrolled.
* **unload**: Fired when the document or window is about to be unloaded.
* **beforeunload**: Fired when the page is about to be unloaded (e.g., closing the tab).

**5. Touch Events (for mobile devices):**

These events are triggered by touch interactions on touchscreen devices.

* **touchstart**: Fired when the user touches the screen.
* **touchmove**: Fired when the user moves their finger across the screen.
* **touchend**: Fired when the user releases their finger from the screen.
* **touchcancel**: Fired when the touch event is canceled (e.g., moving the finger off the screen).

**6. Clipboard Events:**

These events are triggered by actions related to the clipboard.

* **copy**: Fired when content is copied to the clipboard.
* **cut**: Fired when content is cut to the clipboard.
* **paste**: Fired when content is pasted from the clipboard.

**7. Focus Events:**

These events are related to focus changes on elements (like inputs or links).

* **focus**: Fired when an element gains focus (e.g., clicking on an input field).
* **blur**: Fired when an element loses focus (e.g., clicking away from an input field).

**3: DOM Traversal**

1. **Navigating DOM Nodes: parentNode, childNodes, nextSibling, previousSibling**
   * **parentNode**: Access the parent node of an element.
     + **Example**:

let paragraph = document.getElementById("content");

console.log(paragraph.parentNode); // Gets the parent of the paragraph element

* + **childNodes**: Access all child nodes (including text nodes and elements) of an element.
    - **Example**:

let div = document.getElementById("content");

console.log(div.childNodes); // Gets all child nodes, including text nodes

* + **nextSibling**: Access the next sibling of an element (i.e., the next node at the same level).
    - **Example**:

let firstChild = document.getElementById("first");

let nextElement = firstChild.nextSibling; // Accesses the next sibling of the first element

* + **previousSibling**: Access the previous sibling of an element (i.e., the previous node at the same level).
    - **Example**:

let secondChild = document.getElementById("second");

let previousElement = secondChild.previousSibling; // Accesses the previous sibling

1. **Inserting New Elements Using createElement(), appendChild(), removeChild()**
   * **createElement()**: Create a new element node.
     + **Syntax**:

let newElement = document.createElement("tagName");

* + - **Example**:

let newParagraph = document.createElement("p");

newParagraph.textContent = "This is a new paragraph.";

* + **appendChild()**: Appends a new child node to an existing node.
    - **Example**:

let parentDiv = document.getElementById("parent");

parentDiv.appendChild(newParagraph); // Adds the new paragraph to the parent div

* + **removeChild()**: Removes a specified child node from an element.
    - **Example**:

let parentDiv = document.getElementById("parent");

parentDiv.removeChild(newParagraph); // Removes the newly added paragraph

**8: Basic Object-Oriented Programming (OOP) Concepts**

**1: Introduction to OOP**

1. **What is Object-Oriented Programming (OOP)?**
   * **Explanation**: **Object-Oriented Programming (OOP)** is a programming paradigm based on the concept of objects. These objects are instances of classes and encapsulate both data (properties) and behavior (methods). The key goal of OOP is to structure code in a way that mimics real-world entities, making it easier to organize, manage, and extend applications.
   * **Key Principles of OOP**:
     + **Encapsulation**: Bundling data and the methods that operate on the data into a single unit (object).
     + **Abstraction**: Hiding complex implementation details and exposing only the necessary parts.
     + **Inheritance**: Deriving new classes from existing ones, allowing shared behavior and properties.
     + **Polymorphism**: Objects of different classes can be treated as objects of a common superclass.
   * **Example**: In the real world, an object like a **Car** can have properties such as color, model, and year, and methods such as startEngine() or drive(). In OOP, these can be modeled as classes and objects.
2. **Concepts of OOP: Classes, Objects, Methods, Properties**
   * **Class**: A blueprint or template for creating objects. It defines properties and methods that the created objects will have.
     + **Example**:

class Car {

constructor(make, model, year) {

this.make = make;

this.model = model;

this.year = year;

}

drive() {

console.log(`${this.make} ${this.model} is driving!`);

}

}

* + **Object**: An instance of a class. When a class is used to create an object, the object contains the properties and methods defined in the class.
    - **Example**:

const myCar = new Car("Toyota", "Corolla", 2022);

myCar.drive(); // Outputs: Toyota Corolla is driving!

* + **Properties**: Attributes or data held by an object. In the example above, make, model, and year are properties of the Car class.
  + **Methods**: Functions that are associated with an object, typically used to perform actions or computations based on the object's properties.
    - **Example**: The drive() method in the Car class is a method that outputs a message when called.

**Lesson 10.2: Constructor Functions**

1. **Creating Objects with Constructor Functions**
   * **Explanation**: In JavaScript, a constructor function is used to create objects. It is a special type of function that is invoked with the new keyword to create an instance of an object.
   * **Syntax**:

function Car(make, model, year) {

this.make = make;

this.model = model;

this.year = year;

this.drive = function() {

console.log(`${this.make} ${this.model} is driving!`);

};

}

const myCar = new Car("Honda", "Civic", 2023);

myCar.drive(); // Outputs: Honda Civic is driving!

* + **Explanation**:
    - The constructor function Car() defines properties (make, model, year) and a method (drive).
    - The new keyword is used to create an object, and this refers to the newly created object.

1. **this Keyword and How It is Used in Constructor Functions**
   * **Explanation**: In JavaScript, the this keyword refers to the current object that is being created or worked with. Inside a constructor function, this refers to the new object being instantiated.
   * **Example**:

function Person(name, age) {

this.name = name;

this.age = age;

this.greet = function() {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

};

}

const person1 = new Person("Alice", 30);

person1.greet(); // Outputs: Hello, my name is Alice and I am 30 years old.

* + **Explanation**: In the constructor function Person(), this.name and this.age refer to the properties of the person1 object, and this.greet refers to the method of the person1 object.

1. **Prototypes in JavaScript**
   * **Explanation**: Every JavaScript function, including constructor functions, has a prototype property. The prototype is an object that is shared by all instances of that function's objects. You can add methods to the prototype to be shared among all instances, saving memory.
   * **Example**:

function Car(make, model) {

this.make = make;

this.model = model;

}

Car.prototype.displayDetails = function() {

console.log(`Car make: ${this.make}, model: ${this.model}`);

};

const myCar = new Car("Ford", "Mustang");

myCar.displayDetails(); // Outputs: Car make: Ford, model: Mustang

* + **Explanation**: Here, displayDetails() is added to the Car prototype. This method is shared across all instances of Car, and it doesn't need to be defined in the constructor function for each instance.

**Lesson 10.3: Inheritance using Prototypes**

1. **Creating Inheritance Using Prototypes**
   * **Explanation**: In JavaScript, inheritance allows one class (or function) to inherit properties and methods from another class (or function). This can be done by setting the prototype of one constructor function to an instance of another constructor function.
   * **Example**:

function Animal(name) {

this.name = name;

}

Animal.prototype.speak = function() {

console.log(`${this.name} makes a noise.`);

};

function Dog(name) {

Animal.call(this, name); // Inherit properties from Animal

}

Dog.prototype = Object.create(Animal.prototype); // Inherit methods from Animal

Dog.prototype.constructor = Dog; // Fix constructor reference

const dog1 = new Dog("Buddy");

dog1.speak(); // Outputs: Buddy makes a noise.

* + **Explanation**: The Dog constructor function inherits from the Animal constructor function using call() to inherit properties, and Object.create(Animal.prototype) to inherit methods.

1. **Using Object.create() Method for Inheritance**
   * **Explanation**: The Object.create() method creates a new object and sets its prototype to a specified object. This is a cleaner and more modern approach to setting up inheritance in JavaScript.
   * **Example**:

const animal = {

speak: function() {

console.log(`${this.name} makes a noise.`);

}

};

const dog = Object.create(animal);

dog.name = "Buddy";

dog.speak(); // Outputs: Buddy makes a noise.

* + **Explanation**: In this example, dog is an object created using Object.create(animal), making dog inherit from animal. The dog object has its own name property, and it uses the speak() method from animal.

**9. JavaScript Regular Expressions (RegEx)**

A **regular expression** (RegEx) is a sequence of characters that define a search pattern. They are used for matching text in strings, validating data, replacing parts of strings, and much more. In JavaScript, regular expressions are used in string methods like match(), replace(), test(), and split().

**Basics of Regular Expressions**

A regular expression is enclosed between **forward slashes** (/). For example, /abc/ is a regular expression that matches the string "abc".

**Common Syntax and Metacharacters**

**1. Literal Characters**

* **Explanation**: These are the actual characters you want to search for.
  + **Example**: /hello/ matches the string "hello" in a larger text.

**2. Metacharacters**

* **^**: Matches the beginning of a string.
  + **Example**: /^abc/ matches strings that start with "abc".
* **$**: Matches the end of a string.
  + **Example**: /abc$/ matches strings that end with "abc".
* **.**: Matches any single character except newline characters.
  + **Example**: /a.c/ matches "abc", "axc", "a1c", etc.
* **[]**: Denotes a character class, matching any single character within the brackets.
  + **Example**: /[aeiou]/ matches any vowel.
* **[^]**: Denotes a negated character class, matching any character except those inside the brackets.
  + **Example**: /[^0-9]/ matches any non-digit character.
* **|**: Represents a logical OR between two patterns.
  + **Example**: /cat|dog/ matches either "cat" or "dog".
* **()**: Groups patterns together.
  + **Example**: /a(bc|de)/ matches "abc" or "ade".

**Quantifiers**

Quantifiers are used to specify how many times an element in the regular expression should be matched.

* **\***: Matches 0 or more occurrences of the preceding element.
  + **Example**: /ab\*c/ matches "ac", "abc", "abbc", etc.
* **+**: Matches 1 or more occurrences of the preceding element.
  + **Example**: /ab+c/ matches "abc", "abbc", but not "ac".
* **?**: Matches 0 or 1 occurrence of the preceding element.
  + **Example**: /ab?c/ matches "abc" and "ac", but not "abbc".
* **{n}**: Matches exactly n occurrences of the preceding element.
  + **Example**: /a{3}/ matches "aaa".
* **{n,}**: Matches n or more occurrences of the preceding element.
  + **Example**: /a{2,}/ matches "aa", "aaa", "aaaa", etc.
* **{n,m}**: Matches between n and m occurrences of the preceding element.
  + **Example**: /a{2,4}/ matches "aa", "aaa", and "aaaa", but not "a".

**Special Characters**

* **\d**: Matches any digit (equivalent to [0-9]).
  + **Example**: /\d/ matches any digit like "1", "5", etc.
* **\D**: Matches any non-digit character.
  + **Example**: /\D/ matches any character that is not a digit.
* **\w**: Matches any word character (alphanumeric + underscore, equivalent to [A-Za-z0-9\_]).
  + **Example**: /\w/ matches any alphanumeric character or underscore.
* **\W**: Matches any non-word character.
  + **Example**: /\W/ matches any character that is not alphanumeric or underscore.
* **\s**: Matches any whitespace character (spaces, tabs, line breaks).
  + **Example**: /\s/ matches a space or a tab character.
* **\S**: Matches any non-whitespace character.
  + **Example**: /\S/ matches any character that is not a space or tab.
* **\b**: Matches a word boundary (position between a word character and a non-word character).
  + **Example**: /\bword\b/ matches "word" but not "sword".
* **\B**: Matches a non-word boundary.
  + **Example**: /\Bend/ matches "bend" but not "the end".

**Flags in Regular Expressions**

Flags modify the behavior of the regular expression. They are added after the closing delimiter of the regex.

* **g**: Global search, finds all matches in the string, not just the first one.
  + **Example**: /a/g will match all "a"s in a string.
* **i**: Case-insensitive search.
  + **Example**: /hello/i will match "hello", "Hello", "HELLO", etc.
* **m**: Multiline search. It makes ^ and $ match the start and end of each line, not just the string.
  + **Example**: /^abc/m will match "abc" at the beginning of any line in a multi-line string.

**Regular Expression Methods in JavaScript**

1. **test()**: Tests if a pattern exists in a string.
   * **Syntax**: regex.test(string)
   * **Example**:

const regex = /hello/;

console.log(regex.test("hello world")); // true

console.log(regex.test("hi world")); // false

1. **exec()**: Executes a search for a match in a string and returns an array of matches, or null if no match is found.
   * **Syntax**: regex.exec(string)
   * **Example**:

const regex = /hello/;

console.log(regex.exec("hello world")); // ["hello"]

console.log(regex.exec("hi world")); // null

1. **match()**: Retrieves the matches of a regular expression in a string. It returns an array of matches or null.
   * **Syntax**: string.match(regex)
   * **Example**:

const text = "The quick brown fox";

console.log(text.match(/\b\w{5}\b/g)); // ["quick", "brown"]

1. **replace()**: Replaces matched text in a string with a specified replacement.
   * **Syntax**: string.replace(regex, replacement)
   * **Example**:

const text = "The quick brown fox";

const result = text.replace(/\b\w{5}\b/g, "\*\*\*\*\*");

console.log(result); // "The \*\*\*\*\* \*\*\*\*\* fox"

1. **split()**: Splits a string into an array of substrings based on a regular expression.
   * **Syntax**: string.split(regex)
   * **Example**:

const text = "apple,orange,banana";

console.log(text.split(/,/)); // ["apple", "orange", "banana"]

**Examples of Regular Expressions**

1. **Validating an Email Address**
   * **Regex**: /^[a-zA-Z0-9.\_-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/
   * **Explanation**: This regex matches valid email formats (e.g., test@example.com).
2. **Matching a Phone Number**
   * **Regex**: /^\+?[0-9]{1,4}?[-.\s]?[0-9]{1,4}[-.\s]?[0-9]{1,4}[-.\s]?[0-9]{1,9}$/
   * **Explanation**: This regex matches international phone numbers in various formats.

**Example 1: Validating an Email Address**

This program checks if an email address is valid using a regular expression.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Email Validation</title>

</head>

<body>

<h1>Enter an Email Address</h1>

<input type="text" id="emailInput" placeholder="Enter email">

<button id="validateBtn">Validate Email</button>

<p id="result"></p>

<script>

// Function to validate email using regular expression

function validateEmail() {

const email = document.getElementById('emailInput').value;

const regex = /^[a-zA-Z0-9.\_-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/; // Regular expression for email validation

const resultElement = document.getElementById('result');

if (regex.test(email)) {

resultElement.textContent = "Valid email address!";

resultElement.style.color = "green";

} else {

resultElement.textContent = "Invalid email address!";

resultElement.style.color = "red";

}

}

// Adding event listener to validate email when button is clicked

document.getElementById('validateBtn').addEventListener('click', validateEmail);

</script>

</body>

</html>

**Explanation:**

* The regular expression /^[a-zA-Z0-9.\_-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/ checks for a basic email format:
  + ^[a-zA-Z0-9.\_-]+: This part ensures the email's username can contain letters, digits, dots, underscores, and hyphens.
  + @: Ensures the @ symbol is present.
  + [a-zA-Z0-9.-]+: The domain part, which can contain letters, digits, dots, and hyphens.
  + \.[a-zA-Z]{2,6}$: The top-level domain (like .com, .org), which must be between 2 and 6 letters long.

When the button is clicked, the script will check if the email address entered matches the regular expression and show a message accordingly.

**Example 2: Extracting Numbers from a String**

This program extracts all numbers from a given string using a regular expression.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Extract Numbers</title>

</head>

<body>

<h1>Enter a String to Extract Numbers</h1>

<input type="text" id="inputString" placeholder="Enter text with numbers">

<button id="extractBtn">Extract Numbers</button>

<p id="result"></p>

<script>

// Function to extract numbers from the input string

function extractNumbers() {

const inputText = document.getElementById('inputString').value;

const regex = /\d+/g; // Regular expression to match one or more digits

const numbers = inputText.match(regex);

const resultElement = document.getElementById('result');

if (numbers) {

resultElement.textContent = "Extracted numbers: " + numbers.join(", ");

resultElement.style.color = "green";

} else {

resultElement.textContent = "No numbers found!";

resultElement.style.color = "red";

}

}

// Adding event listener to extract numbers when button is clicked

document.getElementById('extractBtn').addEventListener('click', extractNumbers);

</script>

</body>

</html>

**Explanation:**

* The regular expression /\d+/g looks for sequences of digits (\d+), where \d matches any digit, and + ensures that we match one or more digits. The g flag ensures the regular expression searches the entire string, not just the first match.
* The match method returns an array of matched numbers in the string. If no matches are found, it returns null.
* When the button is clicked, the script extracts and displays the numbers found in the input string.

**Example 3: Replacing Text Using Regular Expressions**

This program demonstrates how to replace all instances of a word in a string using a regular expression.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Replace Text</title>

</head>

<body>

<h1>Replace Text Example</h1>

<input type="text" id="inputText" placeholder="Enter some text">

<button id="replaceBtn">Replace 'apple' with 'orange'</button>

<p id="result"></p>

<script>

// Function to replace the word "apple" with "orange"

function replaceText() {

const inputText = document.getElementById('inputText').value;

const regex = /apple/g; // Regular expression to match "apple"

const resultText = inputText.replace(regex, 'orange'); // Replace all "apple" with "orange"

const resultElement = document.getElementById('result');

resultElement.textContent = "Updated Text: " + resultText;

}

// Adding event listener to replace text when button is clicked

document.getElementById('replaceBtn').addEventListener('click', replaceText);

</script>

</body>

</html>

**Explanation:**

* The regular expression /apple/g is used to match the word "apple" globally in the string.
* The replace method replaces all occurrences of "apple" with the word "orange".
* When the button is clicked, the script will show the updated text with all occurrences of "apple" replaced by "orange".