JAVASCRIPT

**Definition of JavaScript**

JavaScript is a high-level, dynamic, and interpreted programming language primarily used for creating and enhancing interactive elements on websites. It is one of the core technologies of the World Wide Web, alongside HTML and CSS, and enables developers to create dynamic and responsive user interfaces. JavaScript can be used both on the client-side and server-side of web applications.

**JavaScript Syntax**

JavaScript syntax is the set of rules that define a correctly structured JavaScript program. Below are some of the key components and syntax rules:

1. **Variables**

var name = "John"; // Using var (older way)

let age = 30; // Using let (block-scoped variable)

const PI = 3.14; // Using const (constant value)

1. **Data Types**
   * **String**

let greeting = "Hello, World!";

* + **Number**

let count = 42;

* + **Boolean**

let isActive = true;

* + **Array**

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

* + **Object**

let person = {

firstName: "John",

lastName: "Doe",

age: 25

};

* + **Null and Undefined**

let value = null;

let notAssigned;

1. **Operators**
   * **Arithmetic Operators**

let sum = 10 + 5; // Addition

let diff = 10 - 5; // Subtraction

let product = 10 \* 5; // Multiplication

let quotient = 10 / 5; // Division

* + **Comparison Operators**

let isEqual = (10 == "10"); // true

let isStrictEqual = (10 === 10); // true

let isGreater = (10 > 5); // true

1. **Control Structures**
   * **Conditional Statements**

if (age > 18) {

console.log("Adult");

} else {

console.log("Minor");

}

* + **Loops**

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

console.log(i);

}

let j = 0;

while (j < 5) {

console.log(j);

j++;

}

1. **Functions**
   * **Function Declaration**

function greet(name) {

return "Hello, " + name;

}

* + **Function Expression**

const greet = function(name) {

return "Hello, " + name;

};

* + **Arrow Function**

const greet = (name) => "Hello, " + name;

1. **DOM Manipulation**
   * **Selecting Elements**

const element = document.getElementById("myElement");

* + **Modifying Elements**

element.textContent = "New Content";

**Uses of JavaScript**

JavaScript is versatile and can be used for a wide range of applications, including but not limited to:

1. **Client-Side Web Development**
   * Enhancing user interfaces with dynamic content and animations
   * Form validation
   * Creating interactive features like sliders, pop-ups, and tabs
   * AJAX requests for asynchronous data loading
2. **Server-Side Development**
   * Building server-side applications using Node.js
   * Handling HTTP requests and responses
   * Working with databases
3. **Mobile App Development**
   * Creating cross-platform mobile applications using frameworks like React Native, Ionic, and Cordova
4. **Desktop Application Development**
   * Building desktop applications with Electron
5. **Game Development**
   * Developing browser-based games using libraries like Phaser and Three.js
6. **Automation and Scripting**
   * Writing scripts for task automation
   * Automating web scraping and data extraction
7. **Data Visualization**
   * Creating interactive data visualizations with libraries like D3.js and Chart.js

JavaScript’s broad ecosystem, extensive libraries, and frameworks make it an essential language for modern web development and beyond.

### Linking JavaScript File: Definition, Syntax, and Uses

### Definition

Linking a JavaScript file refers to the process of connecting an external JavaScript file to an HTML document. This is typically done using the <script> tag in HTML, which allows the JavaScript code to be loaded and executed in the context of the webpage.

### Syntax

The syntax for linking an external JavaScript file in an HTML document is as follows:

<!DOCTYPE html>

<html>

<head>

<title>Linking JavaScript Example</title>

</head>

<body>

<h1>Hello, World!</h1>

<!-- Link to the external JavaScript file -->

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

</body>

</html>

In this example, script.js is the name of the external JavaScript file.

### Uses

Linking an external JavaScript file has several advantages:

1. **Code Organization**: Keeping JavaScript code in separate files helps in maintaining and organizing code better.
2. **Reusability**: External JavaScript files can be reused across multiple HTML documents.
3. **Performance**: Browsers can cache external JavaScript files, leading to faster load times on subsequent visits.

### Example

#### HTML File (index.html)

<!DOCTYPE html>

<html>

<head>

<title>JavaScript Linking Example</title>

</head>

<body>

<h1 id="greeting">Hello, World!</h1>

<button id="changeTextButton">Change Text</button>

<!-- Link to the external JavaScript file -->

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

</body>

</html>

#### JavaScript File (script.js)

// Function to change the text content of the greeting

function changeGreeting() {

const greetingElement = document.getElementById("greeting");

greetingElement.textContent = "Hello, JavaScript!";

}

// Add an event listener to the button

const button = document.getElementById("changeTextButton");

button.addEventListener("click", changeGreeting);

### Explanation

1. **HTML File (index.html)**:
   * The <script src="script.js"></script> tag links to the external JavaScript file named script.js.
   * The HTML file contains an <h1> element with the id greeting and a button with the id changeTextButton.
2. **JavaScript File (script.js)**:
   * The changeGreeting function selects the <h1> element by its id and changes its text content to "Hello, JavaScript!".
   * An event listener is added to the button, which triggers the changeGreeting function when the button is clicked.

### Uses of Linking JavaScript Files

1. **Separation of Concerns**: By linking an external JavaScript file, you keep your HTML structure separate from your JavaScript logic. This makes your code cleaner and easier to manage.
2. **Maintainability**: With separate JavaScript files, it's easier to update and debug your scripts without having to sift through the HTML code.
3. **Reusability**: The same JavaScript file can be linked to multiple HTML files, allowing you to reuse functions and logic across different pages.
4. **Scalability**: As your web project grows, having separate JavaScript files for different functionalities makes it easier to scale and maintain the project.

This example demonstrates how linking an external JavaScript file enhances the functionality of a simple HTML page by allowing dynamic content changes based on user interaction.

**Values and Variables of JavaScript:**

### Values

In JavaScript, values are the data that variables store. JavaScript supports various types of values:

1. **Primitive Values**:
   * **Number**: Represents numerical values.

let age = 25;

let price = 19.99;

* + **String**: Represents textual data.

let name = "Alice";

let greeting = 'Hello, World!';

* + **Boolean**: Represents true or false.

let isActive = true;

let hasFinished = false;

* + **Undefined**: Represents a variable that has been declared but not assigned a value.

let score;

* + **Null**: Represents the intentional absence of any object value.

let data = null;

* + **Symbol**: Represents a unique and immutable value, often used as an object property identifier.

let sym = Symbol('description');

1. **Complex Values**:
   * **Object**: A collection of key-value pairs.

let person = {

firstName: "John",

lastName: "Doe",

age: 30

};

* + **Array**: An ordered list of values.

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

* + **Function**: A block of code designed to perform a particular task.

function greet() {

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

}

.

### Data Types in JavaScript:

A data type in JavaScript specifies the type of data that a variable can hold. JavaScript is a dynamically typed language, meaning variables can hold values of any data type and the type can change at runtime.

### Data Types

JavaScript data types can be broadly classified into two categories:

1. **Primitive Data Types**
2. **Complex (Non-Primitive) Data Types**

### 1. Primitive Data Types

#### Definition

Primitive data types are the most basic data types in JavaScript. They are immutable, meaning their values cannot be changed.

#### Types, Syntax, and Examples

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

let age = 30;

let price = 19.99;

* + Example:

let num1 = 5;

let num2 = 3.14;

let sum = num1 + num2; // 8.14

1. **String**
   * Represents a sequence of characters.
   * Syntax:

let name = "Alice";

let greeting = 'Hello';

* + Example:

let firstName = "John";

let lastName = "Doe";

let fullName = firstName + " " + lastName; // "John Doe"

1. **Boolean**
   * Represents a logical value, either true or false.
   * Syntax:

let isActive = true;

let hasFinished = false;

* + Example:

let isLoggedIn = true;

if (isLoggedIn) {

console.log("Welcome!"); // Outputs: Welcome!

}

1. **Undefined**
   * Represents a variable that has been declared but not assigned a value.
   * Syntax:

let score;

console.log(score); // undefined

1. **Null**
   * Represents the intentional absence of any object value.
   * Syntax:

let data = null;

* + Example:

let user = null; // Represents no user

1. **Symbol**
   * Represents a unique and immutable identifier.
   * Syntax:

let sym = Symbol('description');

* + Example:

let uniqueId = Symbol('id');

### 2. Complex (Non-Primitive) Data Types

#### Definition

Complex data types can hold multiple values and are mutable, meaning their values can be changed.

#### Types, Syntax, and Examples

1. **Object**
   * Represents a collection of key-value pairs.
   * Syntax:

let person = {

firstName: "John",

lastName: "Doe",

age: 30

};

* + Example:

let car = {

make: "Toyota",

model: "Corolla",

year: 2020

};

console.log(car.make); // Outputs: Toyota

1. **Array**
   * Represents an ordered list of values.
   * Syntax:

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

* + Example:

let colors = ["red", "green", "blue"];

console.log(colors[0]); // Outputs: red

1. **Function**
   * Represents a block of code designed to perform a particular task.
   * Syntax:

function greet() {

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

}

* + Example:

function add(a, b) {

return a + b;

}

let result = add(5, 3); // 8

### Usage Examples

1. **Using Primitive Data Types**

let age = 25; // Number

let name = "Alice"; // String

let isStudent = true; // Boolean

let score; // Undefined

let value = null; // Null

let uniqueSymbol = Symbol('unique'); // Symbol

1. **Using Object**

let user = {

username: "john\_doe",

password: "12345",

email: "john@example.com"

};

console.log(user.username); // Outputs: john\_doe

1. **Using Array**

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

console.log(numbers[2]); // Outputs: 3

1. **Using Function**

function multiply(a, b) {

return a \* b;

}

let product = multiply(4, 5); // 20

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

### Conclusion

Understanding data types is fundamental in JavaScript programming. Primitive data types are the simplest forms of data, while complex data types allow for more intricate data structures. Properly using and manipulating these data types is essential for effective coding in JavaScript.

### Variables in JavaScript:

A variable in JavaScript is a named container for storing data values. Variables allow you to store and manipulate data within your programs. You can declare variables using var, let, or const.

### Declaring Variables

#### Using var

* var is function-scoped or globally-scoped.
* Can be re-declared and updated.

var name = "Alice";

name = "Bob"; // Reassigning value

var name = "Charlie"; // Redeclaring

#### Using let

* let is block-scoped.
* Can be updated but not re-declared within the same scope.

let age = 25;

age = 26; // Reassigning value

// let age = 27; // Error: Cannot redeclare 'age' in the same scope

#### Using const

* const is block-scoped.
* Cannot be updated or re-declared.
* Must be initialized at the time of declaration.

const birthYear = 1990;

// birthYear = 1991; // Error: Assignment to constant variable.

// const birthYear = 1992; // Error: Cannot redeclare 'birthYear' in the same scope

### Syntax

1. **var Syntax**

var variableName = value;

1. **let Syntax**

let variableName = value;

1. **const Syntax**

const variableName = value;

### Examples

#### Using var

function varExample() {

var city = "New York";

if (true) {

var city = "Los Angeles";

console.log(city); // Los Angeles

}

console.log(city); // Los Angeles (var is function-scoped)

}

varExample();

#### Using let

function letExample() {

let country = "USA";

if (true) {

let country = "Canada";

console.log(country); // Canada

}

console.log(country); // USA (let is block-scoped)

}

letExample();

#### Using const

const pi = 3.14159;

console.log(pi); // 3.14159

// Attempting to reassign or redeclare will result in an error

// pi = 3.14; // Error

// const pi = 3.14; // Error

### Variable Scope

1. **Global Scope**
   * Variables declared outside any function or block are in the global scope.
   * Accessible from anywhere in the code.

var globalVar = "I am global";

console.log(globalVar); // I am global

1. **Function Scope**
   * Variables declared with var inside a function are scoped to that function.

function testScope() {

var functionVar = "I am function-scoped";

console.log(functionVar); // I am function-scoped

}

testScope();

// console.log(functionVar); // Error: functionVar is not defined

1. **Block Scope**
   * Variables declared with let and const inside a block (e.g., within {}) are scoped to that block.

if (true) {

let blockVar = "I am block-scoped";

const anotherBlockVar = "I am also block-scoped";

console.log(blockVar); // I am block-scoped

console.log(anotherBlockVar); // I am also block-scoped

}

// console.log(blockVar); // Error: blockVar is not defined

// console.log(anotherBlockVar); // Error: anotherBlockVar is not defined

### Hoisting

JavaScript hoists variable declarations to the top of their scope. However, only the declarations are hoisted, not the initializations.

#### Example of Hoisting

console.log(hoistedVar); // undefined (declaration is hoisted, but not initialization)

var hoistedVar = "I am hoisted";

console.log(notHoistedVar); // Error: Cannot access 'notHoistedVar' before initialization

let notHoistedVar = "I am not hoisted";

### Conclusion

Variables in JavaScript are essential for storing and manipulating data. The choice between var, let, and const depends on the desired scope and whether the variable should be reassignable. Using let and const is recommended for modern JavaScript development due to their block-scoping and better error prevention.

### Operators in JavaScript

#### Definition

Operators are special symbols in JavaScript that perform operations on operands (values and variables). They are essential for performing tasks like arithmetic calculations, comparisons, logical operations, and more.

### Types of Operators

1. **Arithmetic Operators**
2. **Assignment Operators**
3. **Comparison Operators**
4. **Logical Operators**
5. **Bitwise Operators**
6. **String Operators**
7. **Conditional (Ternary) Operator**
8. **Type Operators**

### 1. Arithmetic Operators

#### Syntax and Uses

Arithmetic operators perform basic arithmetic on numbers (both literals and variables).

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| + | Addition | a + b |
| - | Subtraction | a - b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (Remainder) | a % b |
| ++ | Increment | a++ or ++a |
| -- | Decrement | a-- or --a |

#### Examples

\let a = 10;

let b = 5;

console.log(a + b); // 15

console.log(a - b); // 5

console.log(a \* b); // 50

console.log(a / b); // 2

console.log(a % b); // 0

a++;

console.log(a); // 11

b--;

console.log(b); // 4

### 2. Assignment Operators

#### Syntax and Uses

Assignment operators assign values to variables.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| = | Assignment | a = 5 |
| += | Addition assignment | a += 5 |
| -= | Subtraction assignment | a -= 5 |
| \*= | Multiplication assignment | a \*= 5 |
| /= | Division assignment | a /= 5 |
| %= | Modulus assignment | a %= 5 |

#### Examples

let x = 10;

x += 5; // x = x + 5

console.log(x); // 15

x -= 3; // x = x - 3

console.log(x); // 12

x \*= 2; // x = x \* 2

console.log(x); // 24

x /= 4; // x = x / 4

console.log(x); // 6

x %= 3; // x = x % 3

console.log(x); // 0

### 3. Comparison Operators

#### Syntax and Uses

Comparison operators compare two values and return a boolean (true or false).

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| == | Equal to | a == b |
| === | Strict equal to | a === b |
| != | Not equal to | a != b |
| !== | Strict not equal to | a !== b |
| > | Greater than | a > b |
| < | Less than | a < b |
| >= | Greater than or equal | a >= b |
| <= | Less than or equal | a <= b |

#### Examples

let a = 10;

let b = 5;

console.log(a == b); // false

console.log(a != b); // true

console.log(a > b); // true

console.log(a < b); // false

console.log(a >= b); // true

console.log(a <= b); // false

let c = '10';

console.log(a == c); // true (loose equality, type is not checked)

console.log(a === c); // false (strict equality, type is checked)

### 4. Logical Operators

#### Syntax and Uses

Logical operators perform logical operations and return a boolean.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| && | Logical AND | a && b |
| ` |  | ` |
| ! | Logical NOT | !a |

#### Examples

let a = true;

let b = false;

console.log(a && b); // false

console.log(a || b); // true

console.log(!a); // false

### 5. Bitwise Operators

#### Syntax and Uses

Bitwise operators perform bit-level operations on integers.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| & | AND | a & b |
| ` | ` | OR |
| ^ | XOR | a ^ b |
| ~ | NOT | ~a |
| << | Left shift | a << b |
| >> | Right shift | a >> b |
| >>> | Zero-fill right shift | a >>> b |

#### Examples

let a = 5; // 0101 in binary

let b = 3; // 0011 in binary

console.log(a & b); // 1 (0001 in binary)

console.log(a | b); // 7 (0111 in binary)

console.log(a ^ b); // 6 (0110 in binary)

console.log(~a); // -6 (1111...1010 in binary)

console.log(a << 1); // 10 (1010 in binary)

console.log(a >> 1); // 2 (0010 in binary)

console.log(a >>> 1);// 2 (0010 in binary)

### 6. String Operators

#### Syntax and Uses

String operators are used to concatenate strings.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| + | Concatenation | a + b |
| += | Concatenation assignment | a += b |

#### Examples

let str1 = "Hello, ";

let str2 = "world!";

console.log(str1 + str2); // "Hello, world!"

let greeting = "Hello, ";

greeting += "world!";

console.log(greeting); // "Hello, world!"

### 7. Conditional (Ternary) Operator

#### Syntax and Uses

The ternary operator is a shorthand for the if-else statement.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| ? : | Ternary | condition ? expr1 : expr2 |

#### Examples

let age = 20;

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

console.log(isAdult); // "Yes"

### 8. Type Operators

#### Syntax and Uses

Type operators are used to identify the type of a variable or to convert data types.

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| Typeof | Returns the type | typeof variable |
| Instanceof | Checks if an object is an instance of a specific class or constructor | object instanceof class |

#### Examples

let name = "Alice";

console.log(typeof name); // "string"

let age = 30;

console.log(typeof age); // "number"

let isAdult = true;

console.log(typeof isAdult); // "boolean"

let person = {

firstName: "John",

lastName: "Doe"

};

console.log(typeof person); // "object"

let arr = [1, 2, 3];

console.log(arr instanceof Array); // true

console.log(person instanceof Object); // true

### Conclusion

Operators in JavaScript are essential tools that perform various operations on values and variables. They help in calculations, comparisons, logical operations, string manipulations, and more, making them fundamental for any JavaScript programmer.

### Operator Precedence in JavaScript

Operator precedence determines the order in which operators are evaluated in expressions. Operators with higher precedence are evaluated before operators with lower precedence. When operators have the same precedence, their associativity determines the order of evaluation.

### Operator Precedence Table

Here's a table showing the precedence of various operators in JavaScript, from highest to lowest:

| **Precedence** | **Operator Type** | **Operators** | **Associativity** |
| --- | --- | --- | --- |
| 21 | Grouping | (...) | n/a |
| 20 | Member Access | obj.prop | Left-to-right |
| 20 | Computed Member Access | obj[prop] | Left-to-right |
| 20 | Function Call | func(...args) | Left-to-right |
| 20 | Optional Chaining | obj?.prop, obj?.[prop], func?.(...args) | Left-to-right |
| 19 | New (with argument list) | new Constructor(...args) | n/a |
| 18 | Increment/Decrement | ++, -- | Right-to-left |
| 17 | Logical NOT | ! | Right-to-left |
| 17 | Bitwise NOT | ~ | Right-to-left |
| 17 | Unary Plus | + | Right-to-left |
| 17 | Unary Negation | - | Right-to-left |
| 17 | Typeof | typeof | Right-to-left |
| 17 | Void | void | Right-to-left |
| 17 | Delete | delete | Right-to-left |
| 16 | Exponentiation | \*\* | Right-to-left |
| 15 | Multiplication | \* | Left-to-right |
| 15 | Division | / | Left-to-right |
| 15 | Remainder | % | Left-to-right |
| 14 | Addition | + | Left-to-right |
| 14 | Subtraction | - | Left-to-right |
| 13 | Bitwise Left Shift | << | Left-to-right |
| 13 | Bitwise Right Shift | >> | Left-to-right |
| 13 | Bitwise Unsigned Right Shift | >>> | Left-to-right |
| 12 | Less Than | < | Left-to-right |
| 12 | Less Than or Equal | <= | Left-to-right |
| 12 | Greater Than | > | Left-to-right |
| 12 | Greater Than or Equal | >= | Left-to-right |
| 12 | In | in | Left-to-right |
| 12 | Instanceof | instanceof | Left-to-right |
| 11 | Equality | == | Left-to-right |
| 11 | Inequality | != | Left-to-right |
| 11 | Strict Equality | === | Left-to-right |
| 11 | Strict Inequality | !== | Left-to-right |
| 10 | Bitwise AND | & | Left-to-right |
| 9 | Bitwise XOR | ^ | Left-to-right |
| 8 | Bitwise OR | ` | ` |
| 7 | Logical AND | && | Left-to-right |
| 6 | Logical OR | ` |  |
| 5 | Nullish Coalescing | ?? | Left-to-right |
| 4 | Conditional (Ternary) | ? : | Right-to-left |
| 3 | Assignment | =, +=, -=, \*=, /=, %= | Right-to-left |
|  |  | <<=, >>=, >>>=, &=, ^=, ` | =` |
| 2 | Yield | yield, yield\* | Right-to-left |
| 1 | Comma | , | Left-to-right |

### Examples

#### Example 1: Arithmetic and Assignment

let a = 10;

let b = 5;

let result = a + b \* 2; // Multiplication has higher precedence than addition

console.log(result); // 20 (5 \* 2 = 10; 10 + 10 = 20)

result = (a + b) \* 2; // Parentheses have the highest precedence

console.log(result); // 30 (10 + 5 = 15; 15 \* 2 = 30)

#### Example 2: Logical Operators

let x = true;

let y = false;

let z = true;

let result = x && y || z; // AND has higher precedence than OR

console.log(result); // true (x && y = false; false || z = true)

result = (x && y) || z; // Parentheses change the order of evaluation

console.log(result); // true (x && y = false; false || z = true)

#### Example 3: Conditional (Ternary) Operator

let age = 18;

let canVote = (age >= 18) ? "Yes" : "No"; // Ternary operator has lower precedence than comparison

console.log(canVote); // "Yes"

let score = 85;

let grade = (score >= 90) ? "A" : (score >= 80) ? "B" : "C";

console.log(grade); // "B"

### Conclusion

Understanding operator precedence is crucial for writing correct and efficient JavaScript code. Using parentheses can help to make the intended order of operations explicit and improve code readability.

**Variable Functions in JavaScript :**

var name = "John"; // Old way, avoid using if possible

let age = 25; // Block-scoped variable

const pi = 3.14; // Constant value, cannot be changed

**Data Types :**

**let number = 5; // Number**

**let name = "Alice"; // String**

**let isStudent = true; // Boolean**

**let colors = ["red", "blue", "green"]; // Array**

**let person = { // Object**

**firstName: "John",**

**lastName: "Doe",**

**age: 30**

**};**

**Operators :**

**// Arithmetic**

**let sum = 10 + 5;**

**let product = 10 \* 5;**

**// Comparison**

**let isEqual = (10 == "10"); // true, because == checks value, not type**

**let isStrictEqual = (10 === "10"); // false, because === checks value and type**

**// Logical**

**let isAdult = (age > 18) && (age < 65);**

**// Assignment**

**let count = 10;**

**count += 5; // Equivalent to count = count + 5**

1. **Functions :**

**// Traditional Function**

**function greet(name) {**

**return "Hello, " + name;**

**}**

**// Arrow Function**

**const greetArrow = (name) => {**

**return "Hello, " + name;**

**}**

**// Arrow Function with Implicit Return**

**const greetShort = name => "Hello, " + name;**

**console.log(greet("Alice"));**

**console.log(greetArrow("Bob"));**

**console.log(greetShort("Charlie"));**

**Events :**

**<button onclick="showMessage()">Click Me</button>**

**<script>**

**function showMessage() {**

**alert("Button was clicked!");**

**}**

**</script>**

**DOM Example :**

**<!DOCTYPE html>**

**<html>**

**<body>**

**<p id="text">Hello, World!</p>**

**<button onclick="changeText()">Change Text</button>**

**<script>**

**function changeText() {**

**document.getElementById("text").innerHTML = "Text changed!";**

**}**

**</script>**

**</body>**

**</html>**