**JAVA SCRIPT HANDS ON PROGRAMS**

1. **Create a simple web page displaying "Hello, World!" using JavaScript.**

**Solution:**

**On Client Browser or Node.js Server Console**

The [console.log() method](https://www.geeksforgeeks.org/javascript-console-log-method/) prints the message to the browser console if we use JavaScript on client side, and on Server console if we use on Node.js. This approach is mainly used for debugging purposes and checking outputs while developing code. The below code is an example of server side printing as we do not have HTML tags here.

console.log('Hello World');  
**Output**

Hello World

**On Client Side**

Using the [document.write() method](https://www.geeksforgeeks.org/html-dom-write-method/" \t "_blank) in JavaScript allows you to display “Hello World” on the client side HTML document

<!DOCTYPE html>

<**html**>

<**head**>

<**title**></**title**>

</**head**>

<**body**>

<**script**>

document.write('Hello World');

</**script**>

</**body**>

</**html**>

**Pop-Up on Client Side**

The [alert() method](https://www.geeksforgeeks.org/html-dom-window-alert-method/) displays a popup alert box with the message. It’s useful for simple notifications or warnings but should be used sparingly as it disrupts user interaction.

<!DOCTYPE html>

<**html**>

<**head**>

<**title**></**title**>

</**head**>

<**body**>

<**script**>

alert('Hello World');

</**script**>

</**body**>

</**html**>

Each of the above methods has different ways of outputting the content. Though ‘document.write()’ is used when we want to print the content onto the document which is the HTML Document. Also ‘console.log()’ is mainly used when we are debugging JavaScript code and the ‘alert()’ is used to show an alert box on the browser window with some message or warning.

**NOTE: How to Add JavaScript in HTML Document?**

To add JavaScript in HTML document, several methods can be used. These methods include embedding JavaScript directly within the [HTML](https://www.geeksforgeeks.org/html-tutorial/) file or linking an external JavaScript file.

**Inline JavaScript**

You can write [JavaScript](https://www.geeksforgeeks.org/javascript/) code directly inside the [HTML element](https://www.geeksforgeeks.org/html-elements/) using the [onclick](https://www.geeksforgeeks.org/html-onclick-event-attribute/), [onmouseover](https://www.geeksforgeeks.org/html-onmouseover-event-attribute/" \t "_blank), or other event handler attributes.

{...}

<**button** onclick="alert('Button Clicked!')">

Click Here

</**button**>

{...}

**Internal JavaScript (Within <script> Tag)**

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

**1. JavaScript Code Inside <head> Tag**

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

{...}

<**head**>

<**script**>

**function** myFun() {

document.getElementById("demo")

.innerHTML = "Content changed!";

}

</**script**>

</**head**>

{...}

**2. JavaScript Code Inside <body> Tag**

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

{...}

<**body**>

<**h2**>

Add JavaScript Code

inside Body Section

</**h2**>

<**h3** id="demo" style="color:green;">

GeeksforGeeks

</**h3**>

<**button** type="button" onclick="myFun()">

Click Here

</**button**>

<**script**>

**function** myFun() {

document.getElementById("demo")

.innerHTML = "Content changed!";

}

</**script**>

</**body**>

{...}

**External JavaScript (Using External File)**

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

HTMLJavaScript

{...}

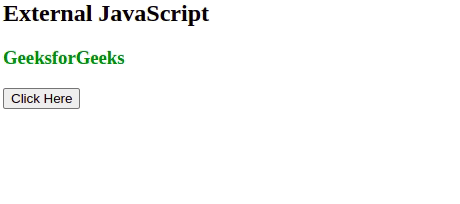
<**head**>

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

</**head**>

{...}

**Output:**



**Advantages of External JavaScript**

* **Faster Page Load Times:** Cached external JavaScript files don’t need to be reloaded every time the page is visited, which can speed up loading times.
* **Improved Readability and Maintenance:** Keeping HTML and JavaScript separate makes both easier to read and maintain.
* **Separation of Concerns:** By separating HTML (structure) and JavaScript (behavior), your code becomes cleaner and more modular.
* **Code Reusability:** One external JavaScript file can be linked to multiple HTML files, reducing redundancy and making updates easier.

**Asynchronous and Deferred JavaScript**

JavaScript can be loaded asynchronously or deferred to optimize page performance, especially for larger scripts. By default, JavaScript blocks the rendering of the HTML page until it is fully loaded, but using async or defer can help improve load times.

**1. async Attribute**

The async attribute loads the script asynchronously, meaning the script will be downloaded and executed as soon as it is available, without blocking the page.

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

**2. defer Attribute**

The defer attribute delays the execution of the script until the entire HTML document has been parsed. This is particularly useful for scripts that manipulate the DOM.

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

**How to Reference External JavaScript Files?**

We can reference an external script in three ways in javascript:

* **By using a full URL:**

src = "https://www.geeksforgeek.org/js/script.js"

* **By using a file path:**

src = "/js/script.js"

* **Without using any path:**

src = "script.js"

1. **Write a function to sum two numbers.**

**JavaScript Program to Add Two Numbers**

There are several methods that can be used to Add Two Numbers in JavaScript:

* [Using + Operator](https://www.geeksforgeeks.org/javascript-program-to-add-two-numbers/#using-operator)
* [Using function](https://www.geeksforgeeks.org/javascript-program-to-add-two-numbers/#using-function)
* [Using Arrow function](https://www.geeksforgeeks.org/javascript-program-to-add-two-numbers/#using-arrow-function)
* [Using Addition Assignment (+=) Operator](https://www.geeksforgeeks.org/javascript-program-to-add-two-numbers/#using-addition-assignment-operator)

**Using + Operator**

In this approach we add two numbers in JavaScript involves using the + operator to perform arithmetic addition on numeric variables, resulting in their sum.

**Syntax:**

x + y;

**Example:**In this example, we are adding two numeric values by using the + operator.

**let** num1 = 10;

**let** num2 = 10;

**let** sum = num1 + num2;

console.log("Sum :", sum);

**Output**

Sum: 20

**Using function**

In this approach, we are adding two numbers using a function in JavaScript involves defining a custom function that takes two parameters, adds them, and returns the result.

**Syntax:**

function additionFunction(a, b) {

return a + b;

}

**Example:**In this example we are using the above-explained approach.

**function** additionFunction(a, b) {

**return** a + b;

}

**let** num1 = 5;

**let** num2 = 10;

**let** sum = additionFunction(num1, num2);

console.log("Sum of given numbers is :", sum);

**Output**

Sum of given numbers is : 15

**Using Arrow function**

Adding two numbers using an [arrow function in JavaScript](https://www.geeksforgeeks.org/arrow-functions-in-javascript/) involves creating a concise function syntax that adds parameters and returns the sum.

**Syntax:**

let addition = (a, b) => a + b;

**Example:**In this example we are using arrow function to add two numbers.

**let** addition = (a, b) => a + b;

**let** num1 = 25;

**let** num2 = 25;

**let** sum = addition(num1, num2);

console.log("Sum of given numbers is :", sum);

**Output**

Sum of given numbers is : 50

**Using Addition Assignment (+=) Operator**

In this approach we use Addition Assignment (+=) operator in which operator Sums up left and right operand values and then assigns the result to the left operand.

**Syntax:**

Y += 1 gives Y = Y + 1

**Example:**In this example we are using the above-explained approach.

**let** num1 = 15;

**let** num2 = 10;

*// Equivalent to num1 = num1 + num2*

num1 += num2;

console.log("Sum of the given number is :", num1);

**Output**

Sum of the given number is : 25

1. **Convert a regular function to an arrow function.**

**Arrow functions in JavaScript**

An arrow function is a shorter syntax for writing functions in JavaScript. Introduced in ES6, arrow functions allow for a more concise and readable code, especially in cases of small functions. Unlike regular functions, arrow functions don’t have their own this, but instead, inherit it from the surrounding context.

* Arrow functions are written with the => symbol, which makes them compact.
* They don’t have their own [this](https://www.geeksforgeeks.org/javascript-this-keyword/). They inherit this from the surrounding context.
* For functions with a single expression, the return is implicit, making the code more concise.
* Arrow functions do not have access to the arguments object, which is available in regular functions.

**const** add = (a, b) => a + b;

console.log(add(5, 3));  
**Output**

8

* **‘add’** is an arrow function that takes two parameters a and b, and returns their sum.
* The arrow function’s concise syntax eliminates the need for the function keyword and curly braces for single-line expressions.

**1. Arrow Function without Parameters**

An arrow function without parameters is defined using empty parentheses (). This is useful when you need a function that doesn’t require any arguments.

**const** gfg = () => {

console.log( "Hi from GeekforGeeks!" );

}

gfg();  
**Output**

Hi from GeekforGeeks!

**2. Arrow Function with Single Parameters**

If your arrow function has a single parameter, you can omit the parentheses around it.

**const** square = x => x\*x;

console.log(square(4));  
**Output**

16

**3. Arrow Function with Multiple Parameters**

Arrow functions with multiple parameters, like **(param1, param2) => { }**, simplify writing concise function expressions in JavaScript, useful for functions requiring more than one argument.

**const** gfg = ( x, y, z ) => {

console.log( x + y + z )

}

gfg( 10, 20, 30 );  
**Output**

60

**4. Arrow Function with Default Parameters**

Arrow functions support default parameters, allowing predefined values if no argument is passed, making JavaScript function definitions more flexible and concise.

**const** gfg = ( x, y, z = 30 ) => {

console.log( x + " " + y + " " + z);

}

gfg( 10, 20 );  
**Output**

10 20 30

**5. Return Object Literals**

In JavaScript, returning object literals within functions is concise: **() => ({ key: value })** returns an object { key: value }, useful for immediate object creation and returning.

**const** makePerson = (firstName, lastName) =>

({first: firstName, last: lastName});

console.log(makePerson("Pankaj", "Bind"));

**Output**

{ first: 'Pankaj', last: 'Bind' }

**Async Arrow Functions**

Arrow functions can be made asynchronous by adding the async keyword before the parameter list.

const fetchData = async () => {  
 const data = await fetch('https://api.example.com/data');  
 return data.json();  
};

**Advantages of Arrow Functions**

* **Concise Syntax:** Arrow functions reduce the amount of code needed for function expressions.
* **Lexical *this*Binding:** Arrow functions automatically bind this to the surrounding context, eliminating common issues when dealing with callbacks.
* **Improved Readability:** For shorter functions, arrow syntax can make your code more readable.

**Limitations of Arrow Functions**

* **No prototype Property:**Arrow functions do not have the prototype property, so they cannot be used as constructors.
* **Cannot be Used with *new*:** Since they lack a prototype, they cannot be used with the new keyword to create instances.
* **Cannot be Generators:**Arrow functions cannot be used as generator functions (function\*) because they do not support the yield keyword.
* **Anonymous Nature:** Debugging can be harder because arrow functions are anonymous by default.
* **No Own this, arguments, super, or new.target:** Arrow functions do not have their own bindings for these properties, which can limit their use in some cases.

**Best Practices for Using Arrow Functions**

* Use arrow functions for callbacks and array methods to improve readability.
* Avoid arrow functions in object methods if this needs to refer to the object itself.
* Prefer arrow functions in functional programming patterns for their concise syntax.
* Use rest parameters for variadic functions when arguments are needed.

**When to Use – Arrow Functions vs. Regular Functions**

| **Aspect** | **Arrow Functions** | **Regular Functions** |
| --- | --- | --- |
| this Binding | Lexically binds this to the surrounding context | this is dynamically bound depending on how the function is called |
| **Syntax** | Shorter and more concise | Requires the function keyword |
| arguments Object | Does not have its own arguments object | Has its own arguments object |
| **Use as Constructor** | Cannot be used as a constructor | Can be used as a constructor with new |
| **Method Definitions** | Cannot be used as methods within objects | Can be used to define methods in objects |
| **Return Value** | Implicit return for single expressions | Must use return keywor |

1. **Create a counter function using closures.**

**How to use a closure to create a private counter in JavaScript?**

A **closure** is a combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function’s scope from an inner function.

**Example:**Here the inner forms closure with outer. The *str* variable can be called a private member whose value we can access, but not modify directly.

* Javascript

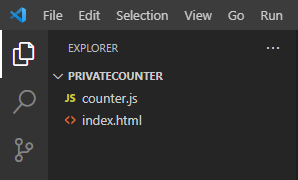
|  |
| --- |
| **function** outer() {          let str = "GeeksforGeeks";  **function** inner() {              console.log(str);          }  **return** inner;      }      const fun = outer();      fun(); |

**Output:**

GeeksforGeeks

The concept of a private counter means that publicly/globally we could not modify the counter variable directly. The below step-by-step guide will teach you how to implement a private counter with closure and understand it.

**Step 1:**Create *counter.js* and *index.html* files. You can give any allowed names to the files.



**Step 2:**First begin with the *index.html* file and create a front-end to see the counter. We would create a <div> to show the value of the counter and two buttons, one for incrementing and the other for decrementing the counter.

* index.html

|  |
| --- |
| <!DOCTYPE html>  <**html** lang="en">    <**head**>      <!-- counter.js becomes available      at execution time-->      <**script** src="counter.js"></**script**>  </**head**>    <**body**>      <**h1** style="color: blue;">          Private Counter using Closure      </**h1**>        <!-- This div displays the value of      private counter-->      <**div** id="counter\_div" style=          "margin-left: 5%; color: red;">            <!-- Default value of counter is zero-->          <**h2**>0</**h2**>      </**div**>        <!-- Buttons for incrementing and      decrementing the value of private counter-->      <**button** onclick="counterHandler(this)"          value="1">          Increment      </**button**>        <**button** onclick="counterHandler(this)"          value="0">          Decrement      </**button**>  </**body**>    </**html**> |

Here we are handling clicks on buttons using the *counterHandler* function which is in the JavaScript file. The value of the buttons helps us to distinguish which button was clicked. And have an *id*of the div so that we could use it to update it from JavaScript code.

**Step 3:**Let’snow work withthe *counter.js*file and implementbehind-the-scenes functions.

* counter.js

|  |
| --- |
| // Global function which would form  // closure with modify function  **function** counter() {      // Private counter variable    let count = 0;      // To increment the value of counter  **function** increment() {      count++;    }      // To decrement the value of counter  **function** decrement() {      count--;    }      // Modify function forms closure    // here which is used outside  **function** modify(val) {        // To check increment or decrement      // button has been clicked  **if** (val === "1") increment();  **else** **if** (val === "0") decrement();        // Return the counter  **return** count;    }      // Returning to make it available    // outside counter function  **return** modify;  }    // Storing the closure modify  const closure = counter();    // This function handles the button  // click, objButton to get value  **function** counterHandler(objButton) {      // Storing the value return by modify    let count = closure(objButton.value);      // Getting div by it's id    // and modifying its inner html    document.getElementById("counter\_div")      .innerHTML = "<h2>" + count + "</h2>";  } |

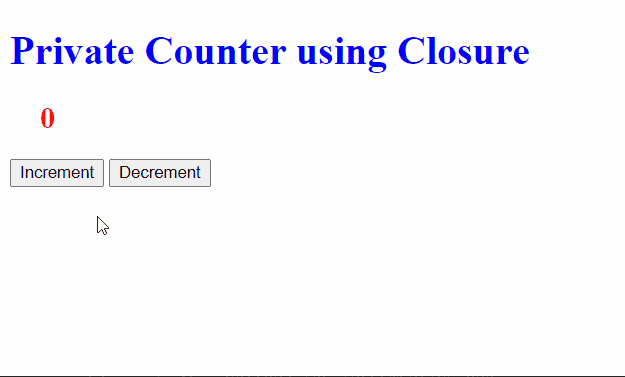
When the buttons get clicked *counterHandler* is called, we get the value of the button from *objButton* Object. If the value is one (1) then to be incremented else to be decremented the value of the counter which is *count* variable.

In the *counter* function, we have a counter variable*count*,*increment* function to increment, and *decrement*function to decrement the value by one.

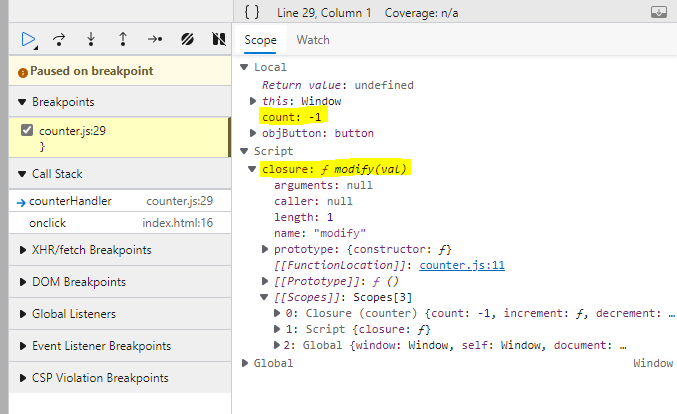
The*modify* function is returned as closure to the global when the *counter* function is called and we store its instance in *closure* constant.

Finally, we modify the div content to the value of the counter returned by closure using its *innerHTML* property.

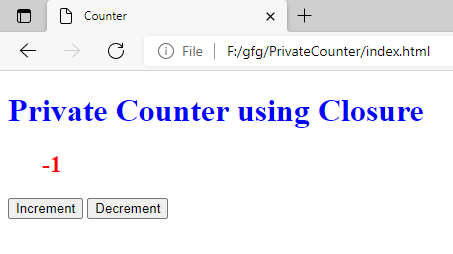
**Step 3:** Copy the full path of the HTML *index.html* file and paste it into any browser. After loading the HTML file in the browser, you would see something similar to it. Now play withthe *increment* and *decrement* buttons and observe the value of counter changing.



Now let’s see what happens in the backend using Developers Tool when the button is clicked. In the below image we had made the counter value equals -1. To see how the scope is defined and related to it.



*Scope in Developers Tool*



*Counter value on the frontend*

This is how you could implement a private counter variable using closures in JavaScript. Closures provide a way to achieve the functionality of data encapsulation in JavaScript.

1. **Define an object representing a car with properties and a method.**

**Creating objects in JavaScript**

An**object in JavaScript**is a collection of key-value pairs, where keys are strings (properties) and values can be any data type. Objects can be created using object literals, constructors, or classes. Properties are defined with key-value pairs, and methods are functions defined within the object, enabling encapsulation and organization of data and behavior.

These are the following 4 ways:

**Table of Content**

* [Creating object with a constructor](https://www.geeksforgeeks.org/creating-objects-in-javascript-4-different-ways/#creating-object-with-a-constructor)
* [Using object literals](https://www.geeksforgeeks.org/creating-objects-in-javascript-4-different-ways/#using-object-literals)
* [Creating object with Object.create() method](https://www.geeksforgeeks.org/creating-objects-in-javascript-4-different-ways/#creating-object-with-objectcreate-method)
* [Using es6 classes](https://www.geeksforgeeks.org/creating-objects-in-javascript-4-different-ways/#using-es6-classes)

**Creating object with a constructor**

One of the easiest ways to instantiate an object is in JavaScript. Constructor is nothing but a function and with the help of a new keyword, the constructor function allows to creation of multiple objects of the same flavor as shown below:

**Example:**This example shows the implementation of the above-explained approach.

*// Simple function*

**function** vehicle(name, maker, engine) {

**this**.name = name;

**this**.maker = maker;

**this**.engine = engine;

}

*// New keyword to create an object*

**let** car = **new** vehicle('GT', 'BMW', '1998cc');

*// Property accessors*

console.log(car.name);

console.log(car.maker);

console.log(car['engine']);

**Output**

GT

BMW

1998cc

**Explanation:**

* **Class Components:** A class in OOP has parameters and member functions, encapsulating attributes and behaviors.
* **Constructor Function:**In a class method, parameters (e.g., name, maker, engine) use this to differentiate between class attributes and argument values.
* **Object Creation:** Create an object (e.g., obj) from the class, initialize it, and call its methods to use the class functionality.

**Using object literals**

Using object literals to create objects in JavaScript involves defining an object directly with key-value pairs inside curly braces {}. This method is concise and straightforward, allowing you to quickly create objects with properties and methods, enhancing code readability.

**Example:**This example shows the implementation of the above-explained approach.

*// Creating js objects with object literal*

**let** car = {

name: 'GT',

maker: 'BMW',

engine: '1998cc'

};

*// Property accessor*

console.log(car.name); *//dot notation*

console.log(car['maker']); *//bracket notation*

**Output**

GT

BMW

**Explanation:**

* In the above code example we Created a car object using an object literal with properties like name, maker, and engine.
* Used dot notation and bracket notation to access properties of the car object.

Now let’s see how we can add more properties to an already defined object:

**let** car = {

name: 'GT',

maker: 'BMW',

engine: '1998cc'

};

*// Adding property to the object*

car.brakesType = 'All Disc';

console.log(car);

**Output**

{ name: 'GT', maker: 'BMW', engine: '1998cc', brakesType: 'All Disc' }

 Methods can also be part of the object while creation or can be added later like properties as shown below:

*// Adding methods to the car object*

**let** car = {

name : 'GT',

maker : 'BMW',

engine : '1998cc',

start : **function**(){

console.log('Starting the engine...');

}

};

car.start();

*// Adding method stop() later to the object*

car.stop = **function**() {

console.log('Applying Brake...');

}

car.stop();

**Output**

Starting the engine...

Applying Brake...

**Explanation:**In the above code start method was added to the car object and later called by the *car.start()* and also the stop method was added too after the object was already declared.

**Creating object with Object.create() Method**

The Object.create() method in JavaScript creates a new object using an existing object as its prototype. This approach allows the new object to inherit properties and methods from the prototype object, enabling inheritance-like behavior. It’s useful for creating objects with shared behaviors while maintaining unique properties.

**Example:**This example shows the implementation of the above-explained approach.

**const** coder = {

isStudying : **false**,

printIntroduction : **function**(){

console.log(`My name is **${this**.name**}**. Am I studying?: **${this**.isStudying**}**`);

}

};

**const** me = Object.create(coder);

me.name = 'Mukul';

me.isStudying = **true**;

me.printIntroduction();

**Output**

My name is Mukul. Am I studying?: true

**Using es6 classes**

Using ES6 classes to create objects in JavaScript involves defining a class with a constructor to initialize properties and methods. Objects are then instantiated from the class using the new keyword, offering a more structured and OOP-like approach to object creation.

**Example:**This example the Vehicle class creates objects with name, maker, and engine properties. car1 is an instance with name set to GT.

*// Using es6 classes*

**class** Vehicle {

**constructor**(name, maker, engine) {

**this**.name = name;

**this**.maker = maker;

**this**.engine = engine;

}

}

**let** car1 = **new** Vehicle('GT', 'BMW', '1998cc');

console.log(car1.name); *//GT*

**Output**

GT