1: Introduction to JavaScript

Overview of JavaScript

History and evolution:

JavaScript was invented by **Brendan Eich** in 1995.

It was developed for **Netscape 2**, and became the **ECMA-262** standard in 1997.

After Netscape handed JavaScript over to ECMA, the Mozilla foundation continued to develop JavaScript for the Firefox browser. Mozilla's latest version was 1.8.5. (Identical to ES5).

Internet Explorer (IE4) was the first browser to support ECMA-262 Edition 1 (ES1).

Here's an overview of its history and evolution:

Birth (1995): JavaScript was introduced in Netscape Navigator 2.0 as a client-side scripting language to enhance web interactivity.

Standardization (1997): Recognizing the need for standardization, Netscape submitted JavaScript to the European Computer Manufacturers Association (ECMA), resulting in the ECMAScript standard. ECMAScript 1 was released in June 1997.

Growth and DOM (Document Object Model): JavaScript's capabilities expanded as it gained support for interacting with HTML documents dynamically through the DOM. This period saw the emergence of DHTML (Dynamic HTML), where JavaScript played a pivotal role.

AJAX (Asynchronous JavaScript and XML): Around the mid-2000s, with the advent of AJAX, JavaScript's role expanded beyond client-side interactions to enabling asynchronous communication with servers. This allowed for more dynamic and responsive web applications.

Frameworks and Libraries: The rise of JavaScript frameworks and libraries such as jQuery, Prototype.js, and Dojo provided developers with powerful tools for simplifying DOM manipulation and creating rich web applications.

Node.js (2009): Node.js, created by Ryan Dahl, brought JavaScript to the server-side, enabling developers to build scalable, real-time applications using JavaScript both on the client and server sides.

ES6/ES2015 (2015): ECMAScript 6, also known as ES2015, introduced significant improvements to the language, including arrow functions, classes, promises, and let/const declarations, making JavaScript more powerful and expressive.

Modern JavaScript (2016-present): JavaScript continues to evolve rapidly with yearly releases of new ECMAScript versions, introducing features like async/await, destructuring, spread syntax, and more, enhancing developer productivity and code readability.

TypeScript: Developed by Microsoft, TypeScript is a superset of JavaScript that adds static typing to the language. It has gained popularity for large-scale applications, offering better tooling and catching errors at compile time.

WebAssembly (Wasm): While not strictly part of JavaScript, WebAssembly is a binary instruction format that enables high-performance execution of code on web browsers. JavaScript can interact with WebAssembly modules, allowing developers to run complex computations at near-native speeds in the browser.

JavaScript's evolution has been driven by the growing demands of web development, leading to a versatile and powerful language that powers a significant portion of the modern web.

Role in web development:

JavaScript plays a crucial role in web development, offering functionality and interactivity to websites and web applications. Here are some key aspects of JavaScript's role:

Client-Side Scripting: JavaScript is primarily used for client-side scripting, meaning it runs on the user's web browser. It enables dynamic updates to the content and behavior of web pages without requiring a full page reload. This capability enhances user experience by creating interactive elements like forms, animations, pop-ups, and dynamic content.

DOM Manipulation: JavaScript interacts with the Document Object Model (DOM) of a webpage, allowing developers to dynamically manipulate HTML elements, change their styles, attributes, and content. This enables developers to create responsive and interactive user interfaces.

Event Handling: JavaScript facilitates event-driven programming, where functions can be triggered by user actions such as clicks, mouse movements, keyboard inputs, or form submissions. This enables developers to create interactive and engaging user experiences.

AJAX and Fetch API: JavaScript enables Asynchronous JavaScript and XML (AJAX) requests, allowing web pages to fetch data from servers in the background without reloading the entire page. With the introduction of the Fetch API, making asynchronous requests has become even more streamlined and efficient.

Browser Compatibility: JavaScript is supported by all major web browsers, making it a universal language for client-side scripting. Modern JavaScript frameworks and libraries often include features to handle browser compatibility issues, ensuring a consistent experience across different browsers.

Single Page Applications (SPAs): JavaScript frameworks like React, Angular, and Vue.js enable the development of SPAs, where a single HTML page is dynamically updated in response to user interactions, without the need for page reloads. SPAs provide a seamless and fluid user experience similar to that of desktop or mobile applications.

Server-Side Development: With the introduction of Node.js, JavaScript can also be used for server-side development. Node.js allows developers to build scalable and high-performance server-side applications using JavaScript, enabling full-stack development with a single programming language.

Overall, JavaScript is essential for modern web development, empowering developers to create dynamic, interactive, and responsive web applications that deliver a rich user experience across various devices and platforms.

Setting Up the Environment

Introduction to the browser console:

The browser console is like a control panel for developers working on websites. It's a place where they can see what's happening behind the scenes and fix any issues. Here's what it does in simpler terms:

Finding Errors: If something isn't working on a website, developers can use the console to find out what's gone wrong. It tells them if there are any mistakes in the code.

Testing Code: Imagine if you're trying out a new recipe. The console is like a taste test for developers. They can quickly try out different bits of code to see if they work, just like tasting a small bit of the dish before serving it.

Checking Stuff: Developers can use the console to check how the webpage is built. They can see what's inside different parts of the page, like finding out what's in a box without having to open it.

Talking to the Website: It's like having a conversation with the website. Developers can tell it to do things or ask it questions, and the console lets them do that.

Spying on the Website: Sometimes, developers need to see what's happening in the background. The console lets them peek at all the messages the website is sending and receiving.

So, in simple words, the browser console is like a secret toolbox for developers to fix problems, test things out, and understand how websites work better.

Basic Syntax and Fundamentals

Variables (var, let, const)

var:
Function Scope: Variables declared with var are function-scoped, which means their scope is limited to the function they are declared in, or if not declared within a function, they become globally scoped.
Hoisting: Variables declared with var are hoisted to the top of their scope during the compilation phase, which means you can access them before they are declared (though their value will be undefined).
Reassignable: You can reassign and update the value of a var variable after it's declared.

let:
Block Scope: Variables declared with let are block-scoped, which means their scope is limited to the block (enclosed by curly braces) in which they are declared. This includes if statements, loops, and functions.
No Hoisting: Variables declared with let are not hoisted to the top of their scope. They are only accessible after they are declared.
Reassignable: Like var, you can reassign and update the value of a let variable after it's declared.

const:
Block Scope: Similar to let, variables declared with const are block-scoped.
No Hoisting: Like let, const variables are not hoisted to the top of their scope.
Immutable: Once a value is assigned to a const variable, it cannot be reassigned or updated. However, it's important to note that for objects and arrays, while the reference cannot be changed, the properties or elements of the object or array can still be modified.

In summary:
Use **var** if you need function scope or you're working with older code.
Use **let** if you need block scope and you may need to reassign the variable later.
Use **const** if you need block scope and you know the value won't change throughout your code.

Data types (strings, numbers, booleans, null, undefined)

In JavaScript, a data type is a classification that specifies what kind of value a variable can hold and what operations can be performed on it. JavaScript is a dynamically typed language, meaning variables can hold values of any type, and their type can change over time. Here are the main data types in JavaScript:

Number: Represents numeric values. It includes integers, floating-point numbers, and special numeric values like Infinity, -Infinity, and NaN (Not a Number).

String: Represents textual data. Strings are sequences of characters enclosed within single (' ') or double (" ") quotes.

Boolean: Represents logical values. It can be either true or false, representing the two possible states of logic.

Undefined: Represents a variable that has been declared but hasn't been assigned a value yet. It also represents the value returned by functions that don't explicitly return anything.

Null: Represents the intentional absence of any value or object. It's often used to indicate that a variable has been declared but isn't currently set to any value.

Operators (arithmetic, comparison, logical)

In JavaScript, operators are special symbols or keywords that perform operations on operands. Operands can be variables, literals, or expressions. Operators allow you to perform actions like arithmetic calculations, comparisons, assignments, and logical operations. Here's a brief overview of the types of operators commonly used in JavaScript:

Arithmetic Operators:

- Addition (+): Adds two numbers together or concatenates two strings.
- Subtraction (-): Subtracts one number from another.
- Multiplication (*): Multiplies two numbers.
- Division (/): Divides one number by another.
- Modulus (%): Returns the remainder of a division operation.
- Exponentiation ()**: Raises the left operand to the power of the right operand (ES6+).

Comparison Operators:

- Equal (==): Returns true if two operands are equal in value.
- Strict Equal (===): Returns true if two operands are equal in value and data type.
- Not Equal (!=): Returns true if two operands are not equal in value.
- Strict Not Equal (!==): Returns true if two operands are not equal in value or data type.
- Greater Than (>): Returns true if the left operand is greater than the right operand.
- Greater Than or Equal (>=): Returns true if the left operand is greater than or equal to the right operand.
- Less Than (<): Returns true if the left operand is less than the right operand.
- Less Than or Equal (<=): Returns true if the left operand is less than or equal to the right operand.

Logical Operators:
Logical AND (&&): Returns true if both operands are true.
Logical OR (||): Returns true if at least one of the operands is true.
Logical NOT (!): Returns true if the operand is false and vice versa.

Control Structures

Conditionals (if, else if, else, switch)

In JavaScript, a conditional is a programming construct that allows you to execute different blocks of code based on whether a specified condition evaluates to true or false. This is achieved using conditional statements, which are often referred to as "if statements" or "if-else statements".

if Statement: The if statement is used to execute a block of code if a specified condition is true. If the condition is false, the block of code is skipped.

if-else Statement: The if-else statement is used to execute one block of code if a specified condition is true and another block of code if the condition is false.

if-else if-else Statement: The if-else if-else statement allows you to check multiple conditions and execute different blocks of code based on the outcome of those conditions.

Ternary Operator: The ternary operator (condition ? expr1 : expr2) provides a compact way to write simple conditional statements. It evaluates the condition and returns expr1 if the condition is true, otherwise it returns expr2.

switch Statement: The switch statement is used to perform different actions based on different conditions. It evaluates an expression and matches the expression's value to a case label, executing the associated block of code.

Loops (for, while, do-while)

In JavaScript, loops are used to repeatedly execute a block of code until a specified condition is met. There are several types of loops available in JavaScript:

for Loop: A for loop is used when you know in advance how many times you want to execute a block of code.

while Loop: A while loop is used when you want to execute a block of code as long as a condition is true. The condition is evaluated before executing the block.

do...while Loop: A do...while loop is similar to a while loop, but the condition is evaluated after executing the block. This guarantees that the block of code is executed at least once.

for...in Loop: A for...in loop is used to iterate over the properties of an object. It enumerates the properties of an object in arbitrary order.

for...of Loop: A for...of loop is used to iterate over the values of an iterable object, such as arrays or strings.

2: Functions and Scope

Functions

Function declaration and expression

Function Declaration:

Function declarations are the traditional way of declaring a function in JavaScript. Here's how it works:

- Syntax: The syntax starts with the function keyword, followed by the name of the function, then a pair of parentheses (), and finally curly braces {} to enclose the code block.
- Naming: we can give a name to the function (demo), which you'll use to call it later in your code.
- Usage: Once declared, we can call this function anywhere in our code by simply using its name followed by parentheses, like demo()

Function Expression:

Function expressions are another way to create functions in JavaScript, where we assign a function to a variable.

- Syntax: Instead of using the function keyword followed by the function name, we create an anonymous function (a function without a name) and assign it to a variable (demo in this case).
- Usage: The variable greet now holds the function, and we can call it using the variable name followed by parentheses, just like with function declarations (demo()).

Arrow functions

- Arrow functions were introduced in ES6.
- Arrow functions allow us to write shorter function syntax:
  const myFunction = (a, b) => a + b;
- It gets shorter! If the function has only one statement, and the statement returns a value, we can remove the brackets and the return keyword
- If we have parameters, we pass them inside the parentheses.
- In fact, if we have only one parameter, we can skip the parentheses as well.

Parameters and arguments

Parameters:

- Parameters are variables listed as a part of the function declaration.
- They act as placeholders for the values that the function will receive when it's called.
- Parameters are like variables defined in the function signature that represent the inputs the function expects.
- They are declared inside the parentheses () of the function declaration.

Example:

```
function demo(name) {
   console.log("Hello, " + name + "!");
}
```

Arguments:

- Arguments are the actual values passed to the function when it's called.
- They are the values that are assigned to the parameters when the function is invoked.
- They are specified inside the parentheses () when calling the function.

Example:

```
demo("John")
```

In this function call, "John" is the argument passed to the name parameter.

In summary, parameters are the placeholders defined in the function declaration, while arguments are the actual values passed to the function when it's called.

Return values

The return value of a function is the value that the function sends back to the code that called it after it completes its task. It's how a function communicates the result of its execution to the rest of the program.

- When a function produces a result, it uses the return keyword followed by the value or expression to be returned.
- If a function doesn't explicitly return anything, it implicitly returns undefined.
- The returned value can be of any data type: numbers, strings, arrays, objects, or even other functions.

Example:

```
function add(a, b) {
   return a + b
}

const result = add(3, 5);
console.log(result)
```

In this example, the add function returns the sum of its two parameters a and b, and the returned value (8) is stored in the variable result.

Scope and Closures

Global vs. local scope

Global Scope:

- Variables declared outside of any function, at the top level of our code, have global scope.
- They can be accessed and modified from anywhere in your code, including inside functions.
- Global variables are accessible to all parts of our program, which can be convenient but also risky if not managed properly, as they can lead to unexpected changes and bugs.
- In a browser environment, global variables are properties of the window object.

Example of a global variable:

```
let globalVariable = 10;

function myFunction() {
   console.log(globalVariable); // Output: 10
}
```

Local Scope:

- Variables declared inside a function have local scope.
- They can only be accessed and modified within the function in which they are declared.
- Local variables are confined to the scope of the function they are declared in, which helps encapsulate data and prevent unintended interactions between different parts of our code.

Example of a local variable:

```
function myFunction() {
   let localVariable = 20;
   console.log(localVariable); // Output: 20
}

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

Key Differences:

- Visibility: Global variables are visible to all parts of your code, while local variables are only visible within the function where they are declared.
- Lifetime: Global variables exist for the entire duration of your program, while local variables are created and destroyed each time their enclosing function is called.
- Avoiding Conflicts: Using local variables helps avoid naming conflicts and unintended interactions between different parts of your code.

In summary, global variables have broad accessibility but can lead to unintended side effects, while local variables are confined to specific functions, promoting encapsulation and preventing unintended modifications. It's generally recommended to minimize the use of global variables to maintain better code organization and reduce potential bugs.

Block scope (let and const)

Block scope, introduced with let and const in JavaScript, refers to the scope of variables defined within a block of code. A block is defined by curly braces {} and can be found in places like loops, conditionals, or standalone code blocks.

Explanation:

- Variables declared with let and const are scoped to the block in which they are defined.
- This means they are only accessible within the block they are declared in, and not outside of it.
- It helps in avoiding unintended side effects and variable collisions by limiting the scope of variables to the block where they are needed.

Example:

```
{
   let message = "Hello" // Variable message is scoped to this block
   console.log(message) // Output: Hello
}

console.log(message) // Error: message is not defined
```

In this example, message is scoped to the block where it's declared with let. Trying to access it outside of that block results in an error because it's not accessible globally.

Key Points:

- let and const provide block-level scoping, unlike var, which has function-level scoping.
- Variables declared with let or const are not hoisted to the top of their enclosing function or global scope.
- This allows for better control over variable usage and prevents issues like accidental variable overwriting.

In essence, block scope with let and const allows for better organization of code and reduces the risk of unintended behavior by restricting the visibility of variables to the blocks where they are needed.

Closures and their applications

Closures in JavaScript are a powerful and often misunderstood concept. In simple terms, a closure is a function that has access to its own scope, the scope in which it was defined, and the scope of its outer function, even after the outer function has finished executing.

Explanation:

- When a function is defined within another function, the inner function has access to the variables and parameters of the outer function, even after the outer function has returned.
- This happens because the inner function maintains a reference to the variables of its outer function, creating a closure.
- Closures allow functions to "remember" the environment in which they were created, capturing and preserving the state of their surrounding context.

Example:

```
function outerFunction() {
    let message = "Hello";

    function innerFunction() {
        console.log(message); // innerFunction has access to the message variable
    }

    return innerFunction;
}

const myClosure = outerFunction(); // Call outerFunction, which returns innerFunction
myClosure(); // Call the returned innerFunction
```

In this example, innerFunction is a closure because it retains access to the message variable even after outerFunction has finished executing. When myClosure is called, it prints "Hello", demonstrating that the closure has preserved the state of its outer function's scope.

Applications:

Data Privacy: Closures can be used to create private variables and functions, accessible only within the scope of the outer function.

Function Factories: Closures can be used to create functions with preset values or configurations, commonly referred to as function factories.

3: Objects and Arrays

Objects

Object literals and properties

In JavaScript, an object is a collection of related data and functionality. These are defined using key-value pairs, where the key is a string (also called a property name) and the value can be any type of data, including other objects and functions.

An object literal is a simple way to create an object using curly braces {}. Inside the braces, you define the properties of the object.

Properties are the key-value pairs defined within an object. Each property has a name (the key) and a value.

Example:

```
let person = {
    name: "John",
    age: 30,
    city: "New York"
};
```

Properties

Properties are the key-value pairs defined within an object. Each property has a name (the key) and a value.

Example:

In the person object above, name, age, and city are properties.

The name property has a value of "John".

The age property has a value of 30.

The city property has a value of "New York".

Methods

An object method in JavaScript is a function that is a property of an object. These methods can perform actions using the object's properties and are defined similarly to regular properties but with function values.

we can define a method by assigning a function to a property of an object.

In ES6, we can define methods more concisely without using the function keyword.

this keyword

The this keyword in JavaScript refers to the object that is executing the current function. Its value depends on how the function is called.

Inside an Object Method: this refers to the object itself.

Inside a Regular Function (Non-Strict Mode): this refers to the global object (window in browsers).

Inside a Regular Function (Strict Mode): this is undefined.

Constructors and prototypes

Constructor

A constructor is a special function used to create and initialize objects. It's like a blueprint that sets up properties and initial values for new objects.

Prototype

A prototype is a way to share methods and properties among all objects created from the same constructor. Instead of each object having its own copy of a method, the method is stored on the prototype and shared.

Arrays

Array methods (push, pop, shift, unshift, map, filter, reduce)

JavaScript provides a variety of methods to manipulate arrays. Here are some of the most commonly used ones:

push()

Adds one or more elements to the end of an array and returns the new length of the array.

pop()

Removes the last element from an array and returns that element.

shift()

Removes the first element from an array and returns that element. This method changes the length of the array.

unshift()

Adds one or more elements to the beginning of an array and returns the new length of the array.

map()

Creates a new array with the results of calling a provided function on every element in the calling array.

filter()

Creates a new array with all elements that pass the test implemented by the provided function.

reduce()

Executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

Iterating over arrays (for loop, forEach)

Iterating over arrays means going through each element of an array and performing some action, like printing each element, modifying them, or performing calculations. There are different ways to iterate over arrays in JavaScript, each with its own benefits.

for Loop

A for loop is a traditional way to iterate over elements in an array. It provides full control over the iteration process.

Example:

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

for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```

fruits.length gets the number of elements in the fruits array.

for (let i = 0; i < fruits.length; i++) loops through the array from the first element (index 0) to the last.

forEach Method

The forEach method provides a more modern and cleaner way to iterate over arrays in JavaScript. It executes a provided function once for each array element.

Example:

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


fruits.forEach(function(fruit) {
    console.log(fruit);
});
```

fruits.forEach(function(fruit) { ... }) iterates over each element in the fruits array.

fruit represents each element of the array in each iteration.

Multidimensional arrays

A multidimensional array in JavaScript is an array that contains other arrays. This creates a matrix-like structure where data is stored in rows and columns.

Characteristics:

- Structure: An array of arrays, forming rows and columns of elements.
- Usage: Used to represent data in a grid-like format, such as matrices, tables, or grids.
- Access: Elements are accessed using multiple indices (e.g., array[rowIndex][columnIndex]).
- Iteration: Requires nested loops for iteration over rows and columns.

Benefits:

- Organized Data: Allows for structured organization of data.
- Mathematical Operations: Suitable for mathematical operations like matrix operations.
- Representation: Ideal for representing grids, maps, or any data organized in a table-like format.

Asynchronous JavaScript

Callbacks

A callback is a function passed as an argument to another function,This technique allows a function to call another functionA callback function can run after another function has finished

Using a callback, you could call the calculator function (myCalculator) with a callback (myCallback), and let the calculator function run the callback after the calculation is finished:

Ex:- function myDisplayer(some) {

```
  document.getElementById("demo").innerHTML = some;
}


function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}


myCalculator(5, 5, myDisplayer);
```

Promises

A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. In simple terms, it's like a placeholder for a result that is not immediately available but will be at some point in the future.

Key Points:

- Pending: The initial state. The operation is not yet complete.
- Fulfilled: The operation completed successfully, and the promise has a value.
- Rejected: The operation failed, and the promise has a reason for the failure (an error).

How to Use Promises

Creating a Promise

You create a promise using the Promise constructor, which takes a function with two parameters: resolve (to indicate success) and reject (to indicate failure).

```
let myPromise = new Promise(function(resolve, reject) {
    // Simulating an asynchronous operation (like a network request)
    setTimeout(function() {
        let success = true; // Simulate success or failure
        if (success) {
            resolve("Operation was successful!");
        } else {
            reject("Operation failed.");
        }
    }, 1000);
});
```

Handling a Promise

You handle the result of a promise using .then() for success and .catch() for errors.

```
myPromise
    .then(function(result) {
        console.log(result); // This runs if the promise is resolved
    })
    .catch(function(error) {
        console.error(error); // This runs if the promise is rejected
    });
```

Simple Example

Here’s a simple example of using a promise to simulate a task that takes some time to complete:

```
function delay(ms) {
    return new Promise(function(resolve) {
        setTimeout(resolve, ms);
    });
}

delay(2000).then(function() {
    console.log("2 seconds have passed");
});
```

In this example, the delay function returns a promise that resolves after a specified number of milliseconds. The then method is used to run a function once the promise resolves (after the delay).

Summary

Promise: An object representing a future value or error.

States: Pending, Fulfilled, Rejected.

Usage: Create with new Promise(), handle with .then() and .catch().

Promises make it easier to work with asynchronous operations in a clean and manageable way, avoiding deeply nested callbacks (known as "callback hell").

Async/Await

async and await in JavaScript

async: A keyword used to declare a function that always returns a Promise. It allows you to write asynchronous code.

```
async function myFunction() { }
```

await: A keyword used inside an async function to pause the execution until a Promise is resolved. It makes asynchronous code look synchronous.

```
async function myFunction() {
  let result = await somePromise;
}
```

These keywords make handling asynchronous operations easier and code more readable.

Try/Catch/finally

try: Run some code that might have an error.

```
try {
  // Code that might throw an error
}
```

catch: Handle the error if it happens.

```
try {
  // Code that might cause an error
} catch (error) {
  // Code to handle the error
}
```

finally: Always run this code, no matter what.

```
try {
  // Code that might cause an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code that runs no matter what
}
```

This helps you manage errors and ensure some code always runs.

Event Handling

Event Listeners in JavaScript

Event Listener: A way to run code when something happens on a web page, like a click or a key press.

How to Use Event Listeners

Select an Element: Choose the HTML element you want to listen for events on

```
let button = document.querySelector('button');
```

Add an Event Listener: Attach a function to run when the event happens.

```
button.addEventListener('click', function() {
  console.log('Button was clicked!');
});
```

This makes your web page interactive by responding to user actions.

Event propagation (bubbling and capturing)

Event propagation describes how events travel through the DOM when an event occurs. It has two main phases: capturing and bubbling.

Event Capturing (Trickling)

Events move from the root of the document down to the target element.

It starts at the top (document) and goes down to the element that triggered the event.

Event Bubbling

After reaching the target element, events bubble up from the target element back to the root.

It starts at the element that triggered the event and goes up to the top (document).

Preventing default actions

Default Action: The browser's built-in behavior when an event occurs (e.g., clicking a link navigates to a new page).

Prevent Default: A way to stop the browser’s built-in behavior from happening.

How to Prevent Default Actions

Use the event.preventDefault() method inside an event listener to stop the default action.

Example

Imagine you have a form and you want to prevent it from submitting when the submit button is clicked:

```
<form id="myForm">
  <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

```
let form = document.getElementById('myForm');

form.addEventListener('submit', function(event) {
  event.preventDefault(); // Prevents the form from submitting
  console.log('Form submission prevented');
});
```