1 What is JavaScript. How to use it?

JavaScript is a versatile, high-level programming language primarily used for adding interactive elements to websites and web applications. It is an essential part of web technology, alongside HTML and CSS, enabling dynamic content, user interaction, and complex web functionalities. JavaScript can be used on both the client-side (in the browser) and the server-side (with environments like Node.js).

**Key Features of JavaScript:**

1. **Dynamic Typing**: Variables can hold any type of data and their type can change at runtime.
2. **Object-Oriented**: Supports objects and prototypes, allowing for complex data structures and reusable code.
3. **Event-Driven**: Executes code in response to events like user actions (clicks, key presses).
4. **First-Class Functions**: Functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.

**How to Use JavaScript**

Using JavaScript typically involves embedding it into an HTML document or linking to an external JavaScript file. Here’s a basic overview of how to get started:

**1. Inline JavaScript**

You can write JavaScript directly within an HTML document using the **<script>** tag.

html

Copy code

<!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <title>JavaScript Example</title> </head> <body> <h1>Hello, World!</h1> <script> document.querySelector('h1').innerText = 'Hello, JavaScript!'; </script> </body> </html>

**2. Internal JavaScript**

Place your JavaScript code within a **<script>** tag in the HTML document’s **<head>** or **<body>** section.

html

Copy code

<!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <title>JavaScript Example</title> <script> function changeText() { document.querySelector('h1').innerText = 'Hello, JavaScript!'; } </script> </head> <body> <h1>Hello, World!</h1> <button onclick="changeText()">Change Text</button> </body> </html>

**3. External JavaScript**

Write your JavaScript in a separate file and link to it from your HTML document using the **<script>** tag.

**HTML File:**

html

Copy code

<!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <title>JavaScript Example</title> <script src="script.js"></script> </head> <body> <h1>Hello, World!</h1> <button onclick="changeText()">Change Text</button> </body> </html>

**External JavaScript File (script.js):**

javascript

Copy code

function changeText() { document.querySelector('h1').innerText = 'Hello, JavaScript!'; }

**Basic Concepts in JavaScript**

1. **Variables**: Used to store data values.

javascript

Copy code

let message = 'Hello, JavaScript!';

1. **Data Types**: Common types include strings, numbers, booleans, arrays, and objects.

javascript

Copy code

let number = 10; // Number let text = "Hello"; // String let isTrue = true; // Boolean let array = [1, 2, 3]; // Array let object = { name: "Alice", age: 25 }; // Object

1. **Functions**: Blocks of code designed to perform a particular task.

javascript

Copy code

function greet(name) { return 'Hello, ' + name; } console.log(greet('World'));

1. **Events**: JavaScript reacts to user actions like clicks, form submissions, or page loads.

html

Copy code

<button onclick="alert('Button clicked!')">Click Me</button>

1. **DOM Manipulation**: Interacting with and modifying the HTML structure and content.

javascript

Copy code

document.getElementById('myElement').style.color = 'blue';

**Advanced Usage**

As you become more familiar with JavaScript, you can explore more advanced topics such as:

* **Asynchronous Programming**: Using callbacks, promises, and async/await to handle asynchronous operations.
* **APIs and AJAX**: Interacting with web APIs and server data without refreshing the page.
* **Frameworks and Libraries**: Utilizing tools like React, Angular, or Vue.js to build complex web applications.

**Conclusion**

JavaScript is a powerful tool for web development, enabling developers to create interactive and dynamic web pages. By understanding its basics and progressively learning more advanced concepts, you can leverage JavaScript to build engaging and sophisticated web experiences.

Top of Form

2 How many type of Variable in JavaScript?

Answer :- There are three types of variable in javascript are 1)var 2)let and 3) const.

3 Define a Data Types in js?

Answer In JavaScript, data types categorize the various types of data that can be manipulated and stored in variables. JavaScript has a dynamic type system, which means variables can hold any type of data and the type can change at runtime. The primary data types in JavaScript are divided into two categories: **primitive** and **non-primitive** (or **reference** types).

Q.4 Write a mul Function Which will Work Properly When invoked With Following Syntax.

Answer:-   
To create a **mul** function that works correctly with a variety of invocation patterns, such as:

console.log(mul(2)(3)(4)); // 24

console.log(mul(5)(-1)(2)); // -10

You need to design a function that returns another function until the final calculation is triggered. One common approach to achieve this is using closures. Here's a solution:

function mul(a) {

return function(b) {

if (b !== undefined) {

return mul(a \* b); // Return a new function that continues the chain

} else {

return a; // When no argument is passed, return the accumulated value

}

}

}

// To make it work, we need to trigger the final calculation in a way that makes sense

console.log(mul(2)(3)(4)()); // 24

console.log(mul(5)(-1)(2)()); // -10

**Explanation:**

1. **Closure**: The **mul** function returns another function that takes an argument **b**. This inner function captures the current value of **a** and multiplies it by **b**.
2. **Recursion**: The inner function returns a new call to **mul** with the product of **a** and **b**, allowing the chain to continue.
3. **Termination**: When the inner function is called without any argument (**b** is **undefined**), it returns the accumulated product.

**Final Result:**

To finalize the computation, you call the function without any arguments at the end of the chain. This triggers the termination condition, returning the accumulated product.

You can also modify the function to avoid the need to call it with an empty invocation for a more natural syntax using a custom **toString** or **valueOf** method:

**Alternative Solution with Custom toString or valueOf:**

function mul(a) {

const fn = (b) => {

if (b !== undefined) {

return mul(a \* b);

} else {

return a;

}

};

fn.valueOf = () => a; // Use valueOf for implicit conversion

return fn;

}

console.log(mul(2)(3)(4)); // 24

console.log(mul(5)(-1)(2)); // -10

1. **Function Returning Functions**: The **mul** function still returns another function that captures the current value and multiplies it.
2. **Custom valueOf**: By defining a custom **valueOf** method on the returned function, we enable implicit conversion to the accumulated value when the final function is evaluated in a context where a primitive value is needed (e.g., **console.log**).

This approach allows for a more natural syntax without the need for an empty final call, enhancing usability and readability.

Q.5 What the deference between undefined and undeclare in JavaScript?

Answer:- In JavaScript, the terms "undefined" and "undeclared" refer to different states of variables, and it's important to understand the distinction between them.

A variable is considered "undefined" when it has been declared but has not been assigned a value. This can happen in several scenarios:

**Declared but not assigned:**

let x;

console.log(x); // Output: undefined

**Function parameters:**

function foo(y) {

console.log(y); // Output: undefined

}

foo();

**Object properties:**

let obj = {};

console.log(obj.prop); // Output: undefined

### Undeclared

A variable is considered "undeclared" when it has not been declared in the current scope (or any containing scope). Trying to access such a variable results in a **ReferenceError**.

**Accessing undeclared variable:**

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

**Strict mode enforcement:**

'use strict';

b = 10; // Output: ReferenceError: b is not defined

### Conclusion

Understanding the difference between "undefined" and "undeclared" is crucial for debugging and writing robust JavaScript code. An "undefined" variable is one that has been declared but not yet given a value, whereas an "undeclared" variable is one that has never been declared in any scope that can be accessed.

6) Using console.log() print out the following statement: The quote 'There is no exercise better for the heart than reaching down and lifting people up.' by John Holmes teaches us to help one another. Using console.log() print out the following quote by Mother Teresa:

Answer:- To print the specified statements using **console.log()**, you can use the following code:

console.log("The quote 'There is no exercise better for the heart than reaching down and lifting people up.' by John Holmes teaches us to help one another.");

console.log('Mother Teresa once said, "If you want to change the world, go home and love your family."');

Q.7 Check if typeof '10' is exactly equal to 10. If not make it exactly equal?

Answer:- To check if **typeof '10'** is exactly equal to **10**, and if not, make it exactly equal, you need to convert the string **'10'** to a number. Here's how you can do it:

let str = '10';

let num = 10;

// Check if the types and values are equal

if (typeof str !== typeof num) {

// Convert the string to a number

str = Number(str); // or str = +str;

}

console.log(str === num); // true

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

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

Q.8 Write a JavaScript Program to find the area of a triangle?

Answer:- bre = 20

            hei = 30

            var Area\_of\_retangle = bre\*hei

            document.write(Area\_of\_retangle)

Q.9 Write a JavaScript program to calculate days left until next Christmas?

Answer:- // Function to calculate the number of days left until the next Christmas

function daysUntilNextChristmas() {

// Get the current date

const today = new Date();

// Get the current year

const currentYear = today.getFullYear();

// Create a date object for this year's Christmas

const thisYearsChristmas = new Date(currentYear, 11, 25); // Months are zero-based, so 11 = December

// Check if today's date is after this year's Christmas

if (today > thisYearsChristmas) {

// If today is after Christmas, calculate days until next year's Christmas

const nextYearsChristmas = new Date(currentYear + 1, 11, 25);

const diffTime = nextYearsChristmas - today;

const diffDays = Math.ceil(diffTime / (1000 \* 60 \* 60 \* 24));

return diffDays;

} else {

// If today is before or on Christmas, calculate days until this year's Christmas

const diffTime = thisYearsChristmas - today;

const diffDays = Math.ceil(diffTime / (1000 \* 60 \* 60 \* 24));

return diffDays;

}

}

// Example usage:

const daysLeft = daysUntilNextChristmas();

console.log(`There are ${daysLeft} days left until next Christmas.`);

10 .What is Condition Statement?

Answer:- A condition statement, also known as a conditional statement, is a programming construct that allows you to execute different pieces of code based on whether a specific condition or set of conditions is true or false. Conditional statements enable decision-making in your code, allowing your program to perform different actions under different circumstances.

11. Find circumference of Rectangle formula : C = 4 \* a ?

Answer:- The formula \( C = 4 \times a \) is not the correct formula for the circumference (or perimeter) of a rectangle. The correct formula for the circumference (perimeter) of a rectangle is given by: \[ C = 2 \times (l + w) \] where: - \( l \) is the length of the rectangle. - \( w \) is the width of the rectangle. Here's how the formula works: 1. Add the length and the width together. 2. Multiply the sum by 2 to account for both pairs of opposite sides of the rectangle. ### Example Calculation Suppose you have a rectangle with: - Length \( l = 10 \) units - Width \( w = 5 \) units The circumference (perimeter) \( C \) would be calculated as follows: \[ C = 2 \times (10 + 5) = 2 \times 15 = 30 \text{ units} \] ### JavaScript Example to Calculate the Perimeter of a Rectangle Here's a simple JavaScript program that calculates the perimeter of a rectangle given its length and width: ```javascript function calculatePerimeter(length, width) { return 2 \* (length + width); } // Example usage: let length = 10; let width = 5; let perimeter = calculatePerimeter(length, width); console.log(`The perimeter of the rectangle is ${perimeter} units.`); ``` In this code: - The `calculatePerimeter` function takes two arguments: `length` and `width`. - It returns the perimeter using the formula \( 2 \times (length + width) \). - The example usage calculates the perimeter for a rectangle with a length of 10 units and a width of 5 units, and then logs the result.

12. WAP to convert years into days and days into years?

Answer:- // Function to convert years to days

function yearsToDays(years) {

const daysInYear = 365; // Approximate number of days in a year (ignoring leap years)

return years \* daysInYear;

}

// Function to convert days to years

function daysToYears(days) {

const daysInYear = 365; // Approximate number of days in a year (ignoring leap years)

return days / daysInYear;

}

// Example usage:

let years = 5;

let days = 1825;

let convertedDays = yearsToDays(years);

let convertedYears = daysToYears(days);

console.log(`${years} years is approximately ${convertedDays} days.`);

console.log(`${days} days is approximately ${convertedYears.toFixed(2)} years.`);

13. Convert temperature Fahrenheit to Celsius? (Conditional logic Question)

Answer:- ### JavaScript Program ```javascript // Function to convert Fahrenheit to Celsius function fahrenheitToCelsius(fahrenheit) { return (fahrenheit - 32) \* 5 / 9; } // Example usage with conditional logic let fahrenheitTemperature = 50; // Change this value to test different temperatures let celsiusTemperature = fahrenheitToCelsius(fahrenheitTemperature); console.log(`${fahrenheitTemperature}°F is equal to ${celsiusTemperature.toFixed(2)}°C.`); // Conditional logic to check if the temperature is above or below freezing point if (fahrenheitTemperature > 32) { console.log("The temperature is above freezing point."); } else if (fahrenheitTemperature < 32) { console.log("The temperature is below freezing point."); } else { console.log("The temperature is at the freezing point."); } ```

14. Write a JavaScript exercise to get the extension of a filename.?

Answer:- // Function to get the file extension

function getFileExtension(filename) {

// Find the last occurrence of the dot character

let lastDotIndex = filename.lastIndexOf('.');

// If no dot or the dot is at the first position (hidden files), return an empty string

if (lastDotIndex === -1 || lastDotIndex === 0) {

return '';

}

// Return the substring from the last dot to the end of the string

return filename.substring(lastDotIndex + 1);

}

// Example usage

let filenames = [

'document.txt',

'archive.tar.gz',

'image.jpeg',

'noextensionfile',

'.hiddenfile',

'script.js',

'data.backup.tar'

];

// Test the function with various filenames

filenames.forEach(filename => {

let extension = getFileExtension(filename);

console.log(`The file extension of "${filename}" is "${extension}"`);

});

15 . What is the result of the expression (5 > 3 && 2 < 4)?

Answer:-

The expression (5 > 3 && 2 < 4) involves two comparison operations combined with the logical AND operator (&&). Let's break it down step by step:

1. **Comparison Operations**:
   * 5 > 3: This checks if 5 is greater than 3. This is true.
   * 2 < 4: This checks if 2 is less than 4. This is also true.
2. **Logical AND (&&) Operator**:
   * The logical AND operator returns true only if both operands are true.
   * In this case, both 5 > 3 and 2 < 4 are true.

Since both comparisons are true, the overall expression (5 > 3 && 2 < 4) evaluates to true.

**Summary**

* 5 > 3 is true
* 2 < 4 is true
* true && true is true

Therefore, the result of the expression (5 > 3 && 2 < 4) is true.

16. What is the result of the expression (true && 1 && "hello")?

Answer:- The expression `(true && 1 && "hello")` involves the logical AND operator (`&&`) with different types of operands. To understand the result, it's important to know how the logical AND operator works in JavaScript:

1. \*\*Logical AND Operator\*\*:

- The logical AND operator (`&&`) evaluates expressions from left to right.

- It returns the first falsy value it encounters. If all values are truthy, it returns the last value.

Let's evaluate the expression step-by-step:

1. \*\*First operand\*\*: `true`

- `true` is a truthy value.

2. \*\*Second operand\*\*: `1`

- Since the first operand (`true`) is truthy, the evaluation continues.

- `1` is also a truthy value.

3. \*\*Third operand\*\*: `"hello"`

- Since both the first (`true`) and second (`1`) operands are truthy, the evaluation continues.

- `"hello"` is a truthy value.

Since all operands are truthy, the logical AND operator returns the last truthy value, which is `"hello"`.

### Summary

- `true` is truthy.

- `1` is truthy.

- `"hello"` is truthy.

- Since all operands are truthy, the result of the expression `(true && 1 && "hello")` is the last operand, which is `"hello"`.

Therefore, the result of the expression `(true && 1 && "hello")` is `"hello"`.

17 . What is the result of the expression true && false || false && true?

Answer:- To determine the result of the expression `true && false || false && true`, we need to evaluate it step by step, considering the precedence and associativity of logical operators in JavaScript.

### Operator Precedence

1. The logical AND (`&&`) operator has higher precedence than the logical OR (`||`) operator.

2. Operators with higher precedence are evaluated first.

### Expression Breakdown

Let's break down the expression step by step:

1. \*\*Evaluate the logical AND (`&&`) operations first\*\*:

- `true && false`: This evaluates to `false` because both operands need to be true for the result to be true.

- `false && true`: This also evaluates to `false` for the same reason.

Now the expression simplifies to:

```javascript

false || false

```

2. \*\*Evaluate the logical OR (`||`) operation\*\*:

- `false || false`: This evaluates to `false` because at least one operand needs to be true for the result to be true.

### Summary

- `true && false` evaluates to `false`.

- `false && true` evaluates to `false`.

- `false || false` evaluates to `false`.

Therefore, the result of the expression `true && false || false && true` is `false`.

18 . What is a Loop and Switch Case in JavaScript define that ?

### Answer:- Loop in JavaScript

A **loop** in JavaScript is a programming construct that repeats a block of code as long as a specified condition is true. Loops are used to perform repetitive tasks, such as iterating over arrays or processing data until a certain condition is met.

### Switch Case in JavaScript

A **switch case** statement in JavaScript is a control structure that allows you to execute one block of code among many options. It evaluates an expression and matches the expression's value to a case label. When a match is found, the corresponding block of code is executed. If no match is found, the default case is executed (if defined).

19. What is the use of is Nan function?

Answer:- The `isNaN()` function in JavaScript is used to determine whether a value is NaN (Not-a-Number) or not. It returns `true` if the value is NaN, and `false` otherwise.

### Usage and Syntax:

```javascript

isNaN(value)

```

### Parameters:

- `value`: The value to be tested.

### Return Value:

- `true` if the value is NaN.

- `false` if the value is a number or can be converted to a number.

### Example:

```javascript

isNaN(NaN); // true

isNaN("Hello"); // true (cannot be converted to a number)

isNaN("123"); // false (can be converted to a number)

isNaN(123); // false (is a number)

```

### Use Cases:

1. \*\*Checking for Numeric Input Validation\*\*:

The `isNaN()` function is commonly used to validate user input, especially when expecting numeric values. It helps ensure that the input is indeed a number before performing further operations.

```javascript

let userInput = prompt("Enter a number:");

if (isNaN(userInput)) {

console.log("Invalid input. Please enter a valid number.");

} else {

console.log("You entered: " + userInput);

}

```

2. \*\*Handling Calculation Errors\*\*:

When performing arithmetic operations where the result might be uncertain or unexpected, `isNaN()` can be used to handle potential errors gracefully.

```javascript

let result = Math.sqrt(-1); // Square root of a negative number

if (isNaN(result)) {

console.log("Error: Invalid input for square root operation.");

} else {

console.log("Square root: " + result);

}

```

3. \*\*Detecting ParseFloat Errors\*\*:

When converting strings to numbers using `parseFloat()`, `isNaN()` can be used to check if the conversion was successful.

```javascript

let floatValue = parseFloat("3.14");

if (isNaN(floatValue)) {

console.log("Error: Unable to parse the floating-point number.");

} else {

console.log("Parsed floating-point number: " + floatValue);

}

```

### Note:

- `isNaN()` converts its argument to a number before testing. If the argument cannot be converted to a number, `isNaN()` returns `true`.

- It's important to note that `isNaN()` can return unexpected results for non-numeric values like empty strings (`""`), boolean values, and objects. To perform more precise numeric checks, consider using `Number.isNaN()` introduced in ECMAScript 6 (ES6).

20 . What is the difference between && and || in JavaScript?

Answer:- In JavaScript, `&&` and `||` are both logical operators, but they have different functionalities and behaviors.

### && (Logical AND) Operator:

- The `&&` operator returns `true` if both operands are `true`. Otherwise, it returns `false`.

- It evaluates expressions from left to right.

- If the left operand is `false`, the right operand is not evaluated because the overall result will be `false` regardless of the right operand's value. This is known as short-circuit evaluation.

- Example:

```javascript

true && true; // true

true && false; // false

false && true; // false

false && false; // false

```

### || (Logical OR) Operator:

- The `||` operator returns `true` if at least one of the operands is `true`. If both operands are `false`, it returns `false`.

- Like `&&`, it also evaluates expressions from left to right.

- If the left operand is `true`, the right operand is not evaluated because the overall result will be `true` regardless of the right operand's value. This is short-circuit evaluation.

- Example:

```javascript

true || true; // true

true || false; // true

false || true; // true

false || false; // false

```

### Summary:

- \*\*&& (Logical AND)\*\*:

- Returns `true` if both operands are `true`.

- Short-circuits if the left operand is `false`.

- \*\*|| (Logical OR)\*\*:

- Returns `true` if at least one of the operands is `true`.

- Short-circuits if the left operand is `true`.

### Use Cases:

- `&&` is typically used for conditions where all conditions must be true.

```javascript

if (age >= 18 && hasID) {

// Allow entry to the club

}

```

- `||` is used when at least one condition should be true.

```javascript

if (username === 'admin' || username === 'superuser') {

// Grant admin privileges

}

```

Understanding these operators and how they behave is crucial for writing concise and efficient JavaScript code.

21 . What is the use of Void (0)?

Answer:- In JavaScript, `void(0)` is an expression that evaluates to `undefined`. The `void` operator takes an operand and evaluates it, then returns `undefined`.

### Usage of `void(0)`:

1. \*\*Preventing Navigation\*\*:

One common use of `void(0)` is to prevent a page from navigating to a new URL when a link is clicked. This is achieved by setting the `href` attribute of the anchor (`<a>`) element to `javascript:void(0)`.

```html

<a href="javascript:void(0)">Click me</a>

```

This creates a "dead" link that does nothing when clicked.

2. \*\*Using in Bookmarklets\*\*:

In bookmarklets, `void(0)` is often used to prevent the browser from navigating to a new page when executing the bookmarklet. For example, a bookmarklet might execute some JavaScript code when clicked, and `void(0)` is appended to the end of the code to prevent the browser from loading a new page.

```javascript

javascript:(function() {

// JavaScript code here

alert("Hello, world!");

})();void(0);

```

In this example, `void(0)` ensures that the bookmarklet does not cause the browser to navigate to a new page after the code execution.

### Advantages of Using `void(0)`:

1. \*\*Avoiding Navigation\*\*: It prevents the browser from navigating to a new page or URL, which is useful when you want to execute JavaScript code without changing the current page.

2. \*\*Minimal Overhead\*\*: `void(0)` is a concise and clear way to achieve the goal of preventing navigation without introducing unnecessary code or side effects.

### Note:

- In modern JavaScript development, the use of `void(0)` for preventing navigation is considered somewhat outdated. Instead, developers often use event.preventDefault() in event handlers to prevent the default behavior of elements like links and form submissions.

22. Check Number Is Positive or Negative in JavaScript?

Answer:- To check if a number is positive or negative in JavaScript, you can use a simple conditional statement. Here's how you can do it:

```javascript

function checkNumber(number) {

if (number > 0) {

return "Positive";

} else if (number < 0) {

return "Negative";

} else {

return "Zero";

}

}

// Example usage:

console.log(checkNumber(5)); // Output: Positive

console.log(checkNumber(-2)); // Output: Negative

console.log(checkNumber(0)); // Output: Zero

```

In this code:

- The `checkNumber` function takes a `number` as input.

- It checks if the number is greater than 0. If it is, it returns "Positive".

- If the number is less than 0, it returns "Negative".

- If the number is exactly 0, it returns "Zero".

You can call this function with any number as an argument to determine if it's positive, negative, or zero.

23 . Find the Character Is Vowel or Not ?

Answer:- To determine whether a character is a vowel or not in JavaScript, you can create a function that checks if the character is one of the vowels (a, e, i, o, u) either in uppercase or lowercase. Here's how you can do it:

```javascript

function isVowel(character) {

// Convert the character to lowercase to handle both uppercase and lowercase vowels

character = character.toLowerCase();

// Check if the character is one of the vowels

if (character === 'a' || character === 'e' || character === 'i' || character === 'o' || character === 'u') {

return true;

} else {

return false;

}

}

// Example usage:

console.log(isVowel('a')); // Output: true

console.log(isVowel('E')); // Output: true

console.log(isVowel('z')); // Output: false

```

In this code:

- The `isVowel` function takes a `character` as input.

- It converts the character to lowercase using the `toLowerCase()` method to handle both uppercase and lowercase vowels.

- It checks if the character is equal to one of the vowels 'a', 'e', 'i', 'o', or 'u'.

- If the character matches any of the vowels, the function returns `true`; otherwise, it returns `false`.

You can call this function with any character as an argument to determine if it's a vowel or not.

24 . Write to check whether a number is negative, positive or zero?

Answer:- You can create a function that checks whether a number is negative, positive, or zero in JavaScript. Here's how you can do it:

```javascript

function checkNumber(number) {

if (number > 0) {

return "Positive";

} else if (number < 0) {

return "Negative";

} else {

return "Zero";

}

}

// Example usage:

console.log(checkNumber(5)); // Output: Positive

console.log(checkNumber(-2)); // Output: Negative

console.log(checkNumber(0)); // Output: Zero

```

In this code:

- The `checkNumber` function takes a `number` as input.

- It checks if the number is greater than 0. If it is, it returns "Positive".

- If the number is less than 0, it returns "Negative".

- If the number is exactly 0, it returns "Zero".

You can call this function with any number as an argument to determine if it's negative, positive, or zero.

25. Write to find number is even or odd using ternary operator in JS?

Answer:- You can use the ternary operator to determine whether a number is even or odd in JavaScript. Here's how you can do it:

```javascript

function checkEvenOrOdd(number) {

return number % 2 === 0 ? "Even" : "Odd";

}

// Example usage:

console.log(checkEvenOrOdd(5)); // Output: Odd

console.log(checkEvenOrOdd(8)); // Output: Even

console.log(checkEvenOrOdd(-3)); // Output: Odd

```

In this code:

- The `checkEvenOrOdd` function takes a `number` as input.

- It uses the ternary operator (`condition ? expressionIfTrue : expressionIfFalse`) to check if the remainder of dividing the number by 2 is equal to 0.

- If the remainder is 0, the number is even, so the function returns "Even".

- If the remainder is not 0, the number is odd, so the function returns "Odd".

You can call this function with any number as an argument to determine if it's even or odd.

26. Write find maximum number among 3 numbers using ternary operator in JS?

Answer:- You can find the maximum number among three numbers using the ternary operator in JavaScript. Here's how you can do it:

```javascript

function findMax(a, b, c) {

return a > b ? (a > c ? a : c) : (b > c ? b : c);

}

// Example usage:

console.log(findMax(5, 10, 3)); // Output: 10

console.log(findMax(-2, -8, -1)); // Output: -1

console.log(findMax(7, 7, 7)); // Output: 7

```

In this code:

- The `findMax` function takes three numbers (`a`, `b`, and `c`) as input.

- It uses nested ternary operators to compare the numbers:

- First, it checks if `a` is greater than `b`.

- If true, it then checks if `a` is greater than `c`. If true, it returns `a`; otherwise, it returns `c`.

- If false, it means `b` is greater than or equal to `a`, so it then checks if `b` is greater than `c`. If true, it returns `b`; otherwise, it returns `c`.

- The result is the maximum of the three numbers.

You can call this function with any three numbers as arguments to find the maximum among them.

27. Write to find minimum number among 3 numbers using ternary operator in JS?

Answer: - You can find the minimum number among three numbers using the ternary operator in JavaScript. Here's how you can do it:

```javascript

function findMin(a, b, c) {

return a < b ? (a < c ? a : c) : (b < c ? b : c);

}

// Example usage:

console.log(findMin(5, 10, 3)); // Output: 3

console.log(findMin(-2, -8, -1)); // Output: -8

console.log(findMin(7, 7, 7)); // Output: 7

```

In this code:

- The `findMin` function takes three numbers (`a`, `b`, and `c`) as input.

- It uses nested ternary operators to compare the numbers:

- First, it checks if `a` is less than `b`.

- If true, it then checks if `a` is less than `c`. If true, it returns `a`; otherwise, it returns `c`.

- If false, it means `b` is less than or equal to `a`, so it then checks if `b` is less than `c`. If true, it returns `b`; otherwise, it returns `c`.

- The result is the minimum of the three numbers.

You can call this function with any three numbers as arguments to find the minimum among them.

Q.28 Write to find the largest of three numbers in JS?

Answer:- To find the largest of three numbers in JavaScript, you can use a simple comparison approach. Here's how you can do it:

```javascript

function findLargest(a, b, c) {

if (a >= b && a >= c) {

return a;

} else if (b >= a && b >= c) {

return b;

} else {

return c;

}

}

// Example usage:

console.log(findLargest(5, 10, 3)); // Output: 10

console.log(findLargest(-2, -8, -1)); // Output: -1

console.log(findLargest(7, 7, 7)); // Output: 7

```

In this code:

- The `findLargest` function takes three numbers (`a`, `b`, and `c`) as input.

- It checks each number against the other two using a series of `if` statements:

- If `a` is greater than or equal to both `b` and `c`, then `a` is the largest.

- If `b` is greater than or equal to both `a` and `c`, then `b` is the largest.

- If neither of the above conditions is true, then `c` is the largest.

- The function returns the largest of the three numbers.

You can call this function with any three numbers as arguments to find the largest among them.

Q.29 Write to show i. Monday to Sunday using switch case in JS? ii. Vowel or Consonant using switch case in JS?

Answer:- Sure, here are examples of using switch-case statements in JavaScript to achieve the requested tasks:

### i. Displaying Days of the Week (Monday to Sunday) using Switch Case:

```javascript

function getDayName(dayNumber) {

switch (dayNumber) {

case 1:

return "Monday";

case 2:

return "Tuesday";

case 3:

return "Wednesday";

case 4:

return "Thursday";

case 5:

return "Friday";

case 6:

return "Saturday";

case 7:

return "Sunday";

default:

return "Invalid day number";

}

}

// Example usage:

for (let i = 1; i <= 7; i++) {

console.log(`Day ${i}: ${getDayName(i)}`);

}

```

### ii. Checking if a Character is Vowel or Consonant using Switch Case:

```javascript

function checkVowelOrConsonant(character) {

// Convert the character to lowercase for case-insensitive comparison

character = character.toLowerCase();

switch (character) {

case 'a':

case 'e':

case 'i':

case 'o':

case 'u':

return "Vowel";

default:

return "Consonant";

}

}

// Example usage:

console.log(checkVowelOrConsonant('a')); // Output: Vowel

console.log(checkVowelOrConsonant('b')); // Output: Consonant

console.log(checkVowelOrConsonant('e')); // Output: Vowel

console.log(checkVowelOrConsonant('z')); // Output: Consonant

```

### Explanation:

1. \*\*Days of the Week (Monday to Sunday)\*\*:

- The `getDayName` function takes a number representing the day of the week (1 for Monday, 2 for Tuesday, etc.) as input.

- It uses a switch-case statement to determine the corresponding day name.

- The function returns the name of the day.

- The example demonstrates how to call this function for each day of the week.

2. \*\*Vowel or Consonant\*\*:

- The `checkVowelOrConsonant` function takes a character as input.

- It converts the character to lowercase for case-insensitive comparison.

- It uses a switch-case statement to check if the character is a vowel ('a', 'e', 'i', 'o', 'u').

- If the character matches any of the vowels, it returns "Vowel"; otherwise, it returns "Consonant".

- The example demonstrates how to call this function with different characters to check if they are vowels or consonants.

Q.30 What are the looping structures in JavaScript? Any one Example?

Answer:- In JavaScript, there are several looping structures that allow you to execute a block of code repeatedly. Some of the most commonly used looping structures are:

1. \*\*for Loop\*\*: Executes a block of code a specified number of times.

```javascript

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

console.log(i);

}

```

2. \*\*while Loop\*\*: Executes a block of code while a specified condition is true.

```javascript

let i = 0;

while (i < 5) {

console.log(i);

i++;

}

```

3. \*\*do...while Loop\*\*: Similar to the while loop, but the block of code is executed at least once before the condition is checked.

```javascript

let i = 0;

do {

console.log(i);

i++;

} while (i < 5);

```

4. \*\*for...in Loop\*\*: Iterates over the properties of an object.

```javascript

const person = {name: 'John', age: 30};

for (let key in person) {

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

}

```

5. \*\*for...of Loop\*\*: Iterates over iterable objects like arrays, strings, etc.

```javascript

const array = [1, 2, 3, 4, 5];

for (let value of array) {

console.log(value);

}

```

6. \*\*forEach() Method\*\*: Executes a provided function once for each array element.

```javascript

const array = [1, 2, 3, 4, 5];

array.forEach(function(value) {

console.log(value);

});

```

### Example using for Loop:

```javascript

// Printing numbers from 1 to 5 using a for loop

for (let i = 1; i <= 5; i++) {

console.log(i);

}

```

In this example, the `for` loop is used to iterate from 1 to 5. The loop initializes `i` to 1, checks if `i` is less than or equal to 5, executes the code block (printing the value of `i`), and then increments `i` by 1 in each iteration.

Q.31 Write a print 972 to 897 using for loop in JS?

Answer:- To print numbers from 972 to 897 using a for loop in JavaScript, you can start the loop from 972 and decrement `i` until it reaches 897. Here's how you can do it:

```javascript

for (let i = 972; i >= 897; i--) {

console.log(i);

}

```

In this code:

- The `for` loop starts with `i` initialized to 972.

- The loop continues as long as `i` is greater than or equal to 897.

- In each iteration, `i` is decremented by 1.

- The `console.log(i)` statement inside the loop prints the current value of `i`.

When you run this code, it will print numbers from 972 down to 897, inclusive.

Q.32 Write to print factorial of given number?

Answer:- To calculate the factorial of a given number in JavaScript, you can use a loop or a recursive function. Here's how you can implement both approaches:

### Using a Loop:

```javascript

function factorialLoop(number) {

if (number < 0) {

return "Factorial is not defined for negative numbers.";

} else if (number === 0) {

return 1;

} else {

let result = 1;

for (let i = 1; i <= number; i++) {

result \*= i;

}

return result;

}

}

// Example usage:

console.log(factorialLoop(5)); // Output: 120 (5! = 5 \* 4 \* 3 \* 2 \* 1 = 120)

console.log(factorialLoop(0)); // Output: 1 (0! = 1)

console.log(factorialLoop(-5)); // Output: Factorial is not defined for negative numbers.

```

### Using Recursion:

```javascript

function factorialRecursion(number) {

if (number < 0) {

return "Factorial is not defined for negative numbers.";

} else if (number === 0) {

return 1;

} else {

return number \* factorialRecursion(number - 1);

}

}

// Example usage:

console.log(factorialRecursion(5)); // Output: 120 (5! = 5 \* 4 \* 3 \* 2 \* 1 = 120)

console.log(factorialRecursion(0)); // Output: 1 (0! = 1)

console.log(factorialRecursion(-5)); // Output: Factorial is not defined for negative numbers.

```

Both `factorialLoop` and `factorialRecursion` functions take a number as input and return the factorial of that number. They handle negative numbers by returning an appropriate message. If the number is 0, they return 1 (since 0! equals 1). Otherwise, they calculate the factorial using either a loop or recursion.

Q.33 Write to print Fibonacci series up to given numbers?

Answer:- To print the Fibonacci series up to a given number in JavaScript, you can use a loop to generate the Fibonacci sequence. Here's how you can do it:

```javascript

function fibonacciSeries(limit) {

// Initialize variables to store the first two numbers of the Fibonacci series

let a = 0, b = 1;

// Print the first two numbers of the Fibonacci series

console.log(a);

console.log(b);

// Generate the subsequent numbers of the Fibonacci series using a loop

while (a + b <= limit) {

// Calculate the next Fibonacci number

let next = a + b;

// Print the next Fibonacci number

console.log(next);

// Update variables for the next iteration

a = b;

b = next;

}

}

// Example usage:

fibonacciSeries(100); // Print Fibonacci series up to 100

```

In this code:

- The `fibonacciSeries` function takes a `limit` parameter, which specifies the maximum value up to which the Fibonacci series should be printed.

- The variables `a` and `b` are initialized to 0 and 1, respectively, representing the first two numbers of the Fibonacci series.

- The first two numbers of the Fibonacci series are printed using `console.log`.

- Inside the `while` loop, the next Fibonacci number is calculated by adding the values of `a` and `b`.

- If the next Fibonacci number is less than or equal to the `limit`, it is printed, and the variables `a` and `b` are updated for the next iteration.

- The loop continues until the next Fibonacci number exceeds the `limit`.

You can call the `fibonacciSeries` function with any positive number to print the Fibonacci series up to that number.

Q.34 Write to print number in reverse order e.g.: number = 64728 ---> reverse =82746 in JS?

Answer:- To reverse the digits of a given number in JavaScript, you can convert the number to a string, split it into an array of characters, reverse the array, and then join it back into a string before converting it back to a number. Here's how you can do it:

```javascript

function reverseNumber(number) {

// Convert the number to a string

let numberStr = number.toString();

// Split the string into an array of characters

let numberArray = numberStr.split('');

// Reverse the array of characters

let reversedArray = numberArray.reverse();

// Join the reversed array back into a string

let reversedStr = reversedArray.join('');

// Convert the reversed string back to a number

let reversedNumber = parseInt(reversedStr, 10);

return reversedNumber;

}

// Example usage:

let number = 64728;

let reversed = reverseNumber(number);

console.log(reversed); // Output: 82746

```

In this code:

1. \*\*Convert the number to a string\*\*: `number.toString()` converts the number to its string representation.

2. \*\*Split the string into an array of characters\*\*: `numberStr.split('')` splits the string into an array of individual characters.

3. \*\*Reverse the array of characters\*\*: `numberArray.reverse()` reverses the array in place.

4. \*\*Join the reversed array back into a string\*\*: `reversedArray.join('')` joins the characters of the reversed array back into a single string.

5. \*\*Convert the reversed string back to a number\*\*: `parseInt(reversedStr, 10)` converts the reversed string back into an integer.

This approach handles the reversal of digits for any integer number. You can call the `reverseNumber` function with any number to get its reversed form.

Q.35 Write a program make a summation of given number (E.g., 1523 Ans: - 11) in JS?

Answer:- To calculate the summation of the digits of a given number in JavaScript, you can convert the number to a string, split it into an array of digits, convert each digit back to a number, and then sum them up. Here's how you can do it:

```javascript

function sumOfDigits(number) {

// Convert the number to a string

let numberStr = number.toString();

// Split the string into an array of characters (digits)

let digitsArray = numberStr.split('');

// Convert each character back to a number and sum them up

let sum = digitsArray.reduce((acc, digit) => acc + parseInt(digit, 10), 0);

return sum;

}

// Example usage:

let number = 1523;

let summation = sumOfDigits(number);

console.log(summation); // Output: 11

```

In this code:

1. \*\*Convert the number to a string\*\*: `number.toString()` converts the number to its string representation.

2. \*\*Split the string into an array of characters\*\*: `numberStr.split('')` splits the string into an array of individual characters (each representing a digit).

3. \*\*Convert each character back to a number and sum them up\*\*: `digitsArray.reduce((acc, digit) => acc + parseInt(digit, 10), 0)` iterates over each character in the array, converts it back to a number using `parseInt(digit, 10)`, and accumulates the sum. The initial value of the accumulator `acc` is set to 0.

This approach will correctly calculate the summation of the digits for any integer number. You can call the `sumOfDigits` function with any number to get the sum of its digits.

Q.36 Write a program you have to make a summation of first and last Digit. (E.g., 1234 Ans: - 5) in JS?

Answer:- To find the summation of the first and last digits of a given number in JavaScript, you can convert the number to a string, extract the first and last characters, convert them back to numbers, and then add them. Here's how you can do it:

```javascript

function sumFirstAndLastDigit(number) {

// Convert the number to a string

let numberStr = number.toString();

// Get the first and last digits as characters

let firstDigit = numberStr.charAt(0);

let lastDigit = numberStr.charAt(numberStr.length - 1);

// Convert the characters back to numbers

let firstDigitNum = parseInt(firstDigit, 10);

let lastDigitNum = parseInt(lastDigit, 10);

// Calculate the sum of the first and last digits

let sum = firstDigitNum + lastDigitNum;

return sum;

}

// Example usage:

let number = 1234;

let summation = sumFirstAndLastDigit(number);

console.log(summation); // Output: 5

```

In this code:

1. \*\*Convert the number to a string\*\*: `number.toString()` converts the number to its string representation.

2. \*\*Get the first and last digits as characters\*\*:

- `numberStr.charAt(0)` retrieves the first character of the string.

- `numberStr.charAt(numberStr.length - 1)` retrieves the last character of the string.

3. \*\*Convert the characters back to numbers\*\*: `parseInt(firstDigit, 10)` and `parseInt(lastDigit, 10)` convert the characters back to numbers.

4. \*\*Calculate the sum of the first and last digits\*\*: Add the two numbers together.

You can call the `sumFirstAndLastDigit` function with any number to get the sum of its first and last digits.

Q.37 Use console.log() and escape characters to print the following pattern in JS? 1 1 1 1 1 2 1 2 4 8 3 1 3 9 27 4 1 4 16 64 5 1 5 25 125

Answer:- To print the desired pattern using `console.log()` and escape characters in JavaScript, you can use the newline character `\n` and tab character `\t` to format the output. Here's how you can do it:

```javascript

console.log("1\t1\t1\t1\t1");

console.log("2\t1\t2\t4\t8");

console.log("3\t1\t3\t9\t27");

console.log("4\t1\t4\t16\t64");

console.log("5\t1\t5\t25\t125");

```

If you want to generate this pattern programmatically, you can use a loop to calculate and print each row:

```javascript

for (let i = 1; i <= 5; i++) {

let row = `${i}\t1\t${i}\t${i \* i}\t${i \* i \* i}`;

console.log(row);

}

```

In this code:

1. The loop iterates from 1 to 5.

2. For each iteration, it constructs a string `row` with the required pattern using template literals and tab characters `\t`.

3. The string `row` is then printed using `console.log()`.

Both approaches will print the following pattern:

```

1 1 1 1 1

2 1 2 4 8

3 1 3 9 27

4 1 4 16 64

5 1 5 25 125

```

Q.38 Use pattern in console.log in JS? 1) 1 1 0 1 0 1 1 0 1 0 1 0 1 0 1 2) A B C D E F G H I J K L M N O 3) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 4) \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

Answer:- Here are the solutions to print each of the requested patterns using `console.log` in JavaScript:

### 1. Pattern of 1s and 0s

```javascript

let pattern1 = '';

for (let i = 1; i <= 5; i++) {

for (let j = 1; j <= i; j++) {

pattern1 += (j % 2 === 0) ? '0 ' : '1 ';

}

pattern1 += '\n';

}

console.log(pattern1);

```

### 2. Pattern of letters

```javascript

let pattern2 = '';

let charCode = 65; // ASCII code for 'A'

for (let i = 1; i <= 5; i++) {

for (let j = 1; j <= i; j++) {

pattern2 += String.fromCharCode(charCode) + ' ';

charCode++;

}

pattern2 += '\n';

}

console.log(pattern2);

```

### 3. Pattern of numbers

```javascript

let pattern3 = '';

let number = 1;

for (let i = 1; i <= 5; i++) {

for (let j = 1; j <= i; j++) {

pattern3 += number + ' ';

number++;

}

pattern3 += '\n';

}

console.log(pattern3);

```

### 4. Pattern of asterisks

```javascript

let pattern4 = '';

for (let i = 1; i <= 5; i++) {

for (let j = 1; j <= i; j++) {

pattern4 += '\* ';

}

pattern4 += '\n';

}

console.log(pattern4);

```

Each code block generates the desired pattern and then prints it using `console.log`. The patterns are built in a nested loop structure, where the outer loop controls the number of lines and the inner loop constructs each line. The `\n` character is used to create a new line after each row is constructed.

Q.39 Accept 3 numbers from user using while loop and check each numbers palindrome?

Answer:- To accept three numbers from the user and check if each number is a palindrome using a `while` loop, you can use the `prompt` function to get user input and a helper function to check if a number is a palindrome. Here's a complete solution:

```html

<!DOCTYPE html>

<html>

<body>

<script>

function isPalindrome(number) {

// Convert the number to a string

let str = number.toString();

// Get the reverse of the string

let reversedStr = str.split('').reverse().join('');

// Check if the original string is equal to the reversed string

return str === reversedStr;

}

let count = 0;

let numbers = [];

while (count < 3) {

let input = prompt("Enter a number:");

let number = parseInt(input, 10);

if (!isNaN(number)) {

numbers.push(number);

count++;

} else {

alert("Please enter a valid number.");

}

}

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

if (isPalindrome(numbers[i])) {

console.log(`The number ${numbers[i]} is a palindrome.`);

} else {

console.log(`The number ${numbers[i]} is not a palindrome.`);

}

}

</script>

</body>

</html>

```

### Explanation:

1. \*\*isPalindrome Function\*\*:

- The `isPalindrome` function checks if a given number is a palindrome. It converts the number to a string, reverses the string, and then checks if the reversed string is equal to the original string.

2. \*\*Main Script\*\*:

- The `count` variable keeps track of how many numbers have been entered.

- The `numbers` array stores the valid numbers entered by the user.

- The `while` loop runs until three valid numbers have been entered.

- Inside the loop, `prompt` is used to accept input from the user. The input is then parsed as an integer.

- If the input is a valid number, it is added to the `numbers` array and the `count` is incremented. If not, an alert is shown to prompt the user to enter a valid number.

- After three numbers have been entered, a `for` loop iterates over the `numbers` array, checking each number for the palindrome property using the `isPalindrome` function and printing the result using `console.log`.

### Note:

- The above script uses `prompt` and `alert`, which are blocking functions and suitable for simple user input in web browsers. For more complex input handling, a different approach involving HTML forms and event listeners would be more appropriate.

Q.40 Write a JavaScript Program to display the current day and time in the following format. Sample Output: Today is Friday. Current Time is 12 PM: 12 : 22 2 ?

Answer:- To display the current day and time in the specified format using JavaScript, you can use the `Date` object to get the current date and time. Here’s how you can implement it:

```html

<!DOCTYPE html>

<html>

<body>

<script>

function displayCurrentDayAndTime() {

// Create a new Date object

let now = new Date();

// Array of weekday names

let days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"];

// Get the current day

let dayName = days[now.getDay()];

// Get the current hour, minute, and second

let hours = now.getHours();

let minutes = now.getMinutes();

let seconds = now.getSeconds();

// Determine AM or PM suffix

let period = hours >= 12 ? 'PM' : 'AM';

// Convert hours from 24-hour format to 12-hour format

hours = hours % 12;

hours = hours ? hours : 12; // If hours is 0, set it to 12

// Add leading zeros to minutes and seconds if necessary

minutes = minutes < 10 ? '0' + minutes : minutes;

seconds = seconds < 10 ? '0' + seconds : seconds;

// Format the current time

let currentTime = `${hours} ${period} : ${minutes} : ${seconds}`;

// Display the output

console.log(`Today is ${dayName}.`);

console.log(`Current Time is ${currentTime}`);

}

// Call the function to display the current day and time

displayCurrentDayAndTime();

</script>

</body>

</html>

```

### Explanation:

1. \*\*Create a new `Date` object\*\*: `let now = new Date();` gets the current date and time.

2. \*\*Array of weekday names\*\*: An array `days` contains the names of the days of the week.

3. \*\*Get the current day\*\*: `let dayName = days[now.getDay()];` retrieves the name of the current day.

4. \*\*Get the current hour, minute, and second\*\*:

- `let hours = now.getHours();`

- `let minutes = now.getMinutes();`

- `let seconds = now.getSeconds();`

5. \*\*Determine AM or PM suffix\*\*: `let period = hours >= 12 ? 'PM' : 'AM';` sets the period based on the hour.

6. \*\*Convert hours to 12-hour format\*\*:

- `hours = hours % 12;` converts the hour from 24-hour to 12-hour format.

- `hours = hours ? hours : 12;` ensures that 0 hours is displayed as 12.

7. \*\*Add leading zeros to minutes and seconds if necessary\*\*:

- `minutes = minutes < 10 ? '0' + minutes : minutes;`

- `seconds = seconds < 10 ? '0' + seconds : seconds;`

8. \*\*Format the current time\*\*: Constructs a string `currentTime` in the desired format.

9. \*\*Display the output\*\*:

- `console.log(`Today is ${dayName}.`);`

- `console.log(`Current Time is ${currentTime}`);`

The script displays the current day and time in the format "Today is Friday. Current Time is 12 PM: 12 : 22".

Q.41 Write a JavaScript program to get the current date?

Answer:- To get the current date in JavaScript and display it in a specific format, you can use the `Date` object. Here's a simple example that displays the current date in the format "MM/DD/YYYY":

```html

<!DOCTYPE html>

<html>

<body>

<script>

function getCurrentDate() {

// Create a new Date object

let now = new Date();

// Get the current day, month, and year

let day = now.getDate();

let month = now.getMonth() + 1; // Months are zero-based, so add 1

let year = now.getFullYear();

// Add leading zeros to day and month if necessary

day = day < 10 ? '0' + day : day;

month = month < 10 ? '0' + month : month;

// Format the date as MM/DD/YYYY

let currentDate = `${month}/${day}/${year}`;

// Display the current date

console.log(`Current Date: ${currentDate}`);

}

// Call the function to display the current date

getCurrentDate();

</script>

</body>

</html>

```

### Explanation:

1. \*\*Create a new `Date` object\*\*: `let now = new Date();` gets the current date and time.

2. \*\*Get the current day, month, and year\*\*:

- `let day = now.getDate();` retrieves the day of the month.

- `let month = now.getMonth() + 1;` retrieves the month (adding 1 because months are zero-based).

- `let year = now.getFullYear();` retrieves the full year.

3. \*\*Add leading zeros to day and month if necessary\*\*:

- `day = day < 10 ? '0' + day : day;` adds a leading zero if the day is less than 10.

- `month = month < 10 ? '0' + month : month;` adds a leading zero if the month is less than 10.

4. \*\*Format the date as MM/DD/YYYY\*\*: Constructs a string `currentDate` in the desired format.

5. \*\*Display the current date\*\*: Uses `console.log` to print the formatted date.

When you run this script, it will print the current date in the format "MM/DD/YYYY". You can also modify the format as needed, for example, to "YYYY-MM-DD" by adjusting the string template accordingly:

```javascript

let currentDate = `${year}-${month}-${day}`;

console.log(`Current Date: ${currentDate}`);

```

Q.42 Write a JavaScript program to compare two objects?

Answer:- To compare two objects in JavaScript, you need to check if they have the same properties and values. A deep comparison is required, meaning you need to recursively compare each property. Here's a function to do that:

```javascript

function isObject(obj) {

return obj != null && typeof obj === 'object';

}

function deepEqual(obj1, obj2) {

// If both are the same object reference, they are equal

if (obj1 === obj2) {

return true;

}

// If either is not an object, they are not equal

if (!isObject(obj1) || !isObject(obj2)) {

return false;

}

// Get the keys of both objects

const keys1 = Object.keys(obj1);

const keys2 = Object.keys(obj2);

// If they have a different number of keys, they are not equal

if (keys1.length !== keys2.length) {

return false;

}

// Compare each key and value

for (let key of keys1) {

if (!keys2.includes(key) || !deepEqual(obj1[key], obj2[key])) {

return false;

}

}

// If all keys and values are equal, the objects are equal

return true;

}

// Example usage:

const obj1 = {

name: 'John',

age: 30,

details: {

hobbies: ['reading', 'gaming'],

address: {

city: 'New York',

zip: '10001'

}

}

};

const obj2 = {

name: 'John',

age: 30,

details: {

hobbies: ['reading', 'gaming'],

address: {

city: 'New York',

zip: '10001'

}

}

};

console.log(deepEqual(obj1, obj2)); // Output: true

const obj3 = {

name: 'John',

age: 31,

details: {

hobbies: ['reading', 'gaming'],

address: {

city: 'New York',

zip: '10001'

}

}

};

console.log(deepEqual(obj1, obj3)); // Output: false

```

### Explanation:

1. \*\*isObject Function\*\*:

- `isObject(obj)` checks if a value is an object and not null.

2. \*\*deepEqual Function\*\*:

- The function starts by checking if `obj1` and `obj2` are the same reference. If so, they are equal.

- If either of the values is not an object, they are compared directly.

- If both values are objects, their keys are compared. If they have different numbers of keys, they are not equal.

- Each key in `obj1` is checked to see if it exists in `obj2` and if their corresponding values are equal (recursively).

- If all keys and values match, the objects are considered equal.

### Usage:

- The `deepEqual` function can be used to compare any two objects, even nested ones. It will return `true` if they are deeply equal and `false` otherwise.

Q.43 Write a JavaScript program to convert an array of objects into CSV string?

Answer:- To convert an array of objects into a CSV string in JavaScript, you can follow these steps:

1. Extract the headers (keys) from the objects.

2. Create a CSV string by joining the headers and the values of each object.

Here's how you can do it:

```javascript

function arrayToCSV(data) {

if (!data.length) {

return '';

}

// Extract the headers

const headers = Object.keys(data[0]);

// Create the CSV rows

const rows = data.map(obj =>

headers.map(header => JSON.stringify(obj[header] || '')).join(',')

);

// Combine headers and rows

const csvContent = [headers.join(','), ...rows].join('\n');

return csvContent;

}

// Example usage:

const data = [

{ name: "John", age: 30, city: "New York" },

{ name: "Jane", age: 25, city: "Los Angeles" },

{ name: "Mike", age: 35, city: "Chicago" }

];

const csvString = arrayToCSV(data);

console.log(csvString);

```

### Explanation:

1. \*\*Check if the data array is empty\*\*:

- `if (!data.length) { return ''; }` ensures that if the array is empty, an empty string is returned.

2. \*\*Extract the headers\*\*:

- `const headers = Object.keys(data[0]);` retrieves the keys from the first object in the array to use as headers.

3. \*\*Create the CSV rows\*\*:

- `const rows = data.map(obj => headers.map(header => JSON.stringify(obj[header] || '')).join(','));`

- For each object in the array, it maps each header to the corresponding value in the object. `JSON.stringify` is used to handle cases where values may contain commas, quotes, or other special characters. `obj[header] || ''` ensures that `undefined` values are replaced with empty strings.

4. \*\*Combine headers and rows\*\*:

- `const csvContent = [headers.join(','), ...rows].join('\n');`

- Joins the headers and the rows with new line characters to form the complete CSV string.

### Example Output:

The resulting CSV string will look like this:

```

name,age,city

"John",30,"New York"

"Jane",25,"Los Angeles"

"Mike",35,"Chicago"

```

This function can handle arrays of objects with varying structures and special characters within the values.

Q.44 Write a JavaScript program to capitalize first letter of a string?

Answer:- You can capitalize the first letter of a string in JavaScript by using string manipulation techniques. Here's a function to achieve that:

```javascript

function capitalizeFirstLetter(str) {

// Check if the input string is empty

if (!str) {

return '';

}

// Capitalize the first letter and concatenate with the rest of the string

return str.charAt(0).toUpperCase() + str.slice(1);

}

// Example usage:

console.log(capitalizeFirstLetter("hello")); // Output: "Hello"

console.log(capitalizeFirstLetter("javascript")); // Output: "Javascript"

console.log(capitalizeFirstLetter("")); // Output: ""

```

### Explanation:

1. \*\*Check for empty string\*\*:

- `if (!str)` checks if the input string is empty. If it is, the function returns an empty string.

2. \*\*Capitalize the first letter\*\*:

- `str.charAt(0).toUpperCase()` extracts the first character of the string and converts it to uppercase.

- `str.slice(1)` extracts the rest of the string starting from the second character.

3. \*\*Return the modified string\*\*:

- The capitalized first letter is concatenated with the rest of the string and returned.

This function handles empty strings gracefully and capitalizes the first letter of non-empty strings.

Q. 45 Write a JavaScript program to determine if a variable is array?

Answer:- To determine if a variable is an array in JavaScript, you can use the `Array.isArray()` method. Here's how you can do it:

```javascript

function isArray(variable) {

return Array.isArray(variable);

}

// Example usage:

console.log(isArray([])); // Output: true

console.log(isArray([1, 2, 3])); // Output: true

console.log(isArray({})); // Output: false

console.log(isArray("string")); // Output: false

console.log(isArray(123)); // Output: false

```

### Explanation:

1. \*\*Using `Array.isArray()`\*\*:

- The `Array.isArray()` method checks if a value is an array.

- It returns `true` if the value is an array and `false` otherwise.

2. \*\*Defining the `isArray` function\*\*:

- The `isArray` function simply returns the result of `Array.isArray(variable)`.

- This function abstracts the usage of `Array.isArray()` and makes the code more readable.

3. \*\*Example usage\*\*:

- The function is called with different types of variables to demonstrate its behavior.

- Arrays return `true`, while other types return `false`.

This approach provides a reliable and built-in way to determine if a variable is an array in JavaScript.

Q.46 Write a JavaScript program to clone an array?

Answer:- To clone an array in JavaScript, you can use various methods such as the spread operator (`...`), the `slice()` method, or the `concat()` method. Here are examples of each approach:

1. \*\*Using the spread operator (`...`)\*\*:

```javascript

function cloneArray(arr) {

return [...arr];

}

// Example usage:

const originalArray = [1, 2, 3];

const clonedArray = cloneArray(originalArray);

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

```

2. \*\*Using the `slice()` method\*\*:

```javascript

function cloneArray(arr) {

return arr.slice();

}

// Example usage:

const originalArray = [1, 2, 3];

const clonedArray = cloneArray(originalArray);

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

```

3. \*\*Using the `concat()` method\*\*:

```javascript

function cloneArray(arr) {

return [].concat(arr);

}

// Example usage:

const originalArray = [1, 2, 3];

const clonedArray = cloneArray(originalArray);

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

```

### Explanation:

- \*\*Spread Operator (`...`)\*\*: This operator spreads the elements of the array into a new array.

- \*\*`slice()` Method\*\*: This method creates a new array containing a copy of the elements from the original array.

- \*\*`concat()` Method\*\*: This method concatenates the original array with an empty array, effectively creating a copy.

All three approaches create a new array that is a shallow copy of the original array. This means that modifying elements in the cloned array will not affect the original array, but modifying nested arrays or objects will affect both the original and cloned arrays if they share the same reference.

Q.47 What is the drawback of declaring methods directly in JavaScript objects?

Answer:- One drawback of declaring methods directly in JavaScript objects is that it creates a new function object for each instance of the object. This can lead to memory inefficiency, especially if the method performs complex operations or is frequently called.

When a method is declared directly in an object literal or added to an object dynamically, each instance of that object will have its own copy of the method. This means that memory will be allocated for each instance's method, even though the functionality is identical across instances.

Consider the following example:

```javascript

// Object constructor function

function Person(name, age) {

this.name = name;

this.age = age;

this.sayHello = function() {

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

};

}

// Create two instances of the Person object

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

const person2 = new Person('Bob', 25);

// Both instances have their own copy of the sayHello method

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

person2.sayHello(); // Output: Hello, my name is Bob and I am 25 years old.

```

In this example, each `Person` object created with the constructor function has its own `sayHello` method. If there are many instances of `Person`, each with its own method, it can lead to increased memory usage.

To mitigate this drawback, you can define methods on the prototype of the constructor function instead. This way, all instances of the object share the same method, reducing memory usage.

Q.48 Print the length of the string on the browser console using console.log()?

Answer:- To print the length of a string on the browser console using `console.log()`, you can simply pass the `length` property of the string to `console.log()`. Here's how you can do it:

```javascript

const str = "Hello, World!";

console.log(str.length);

```

When you execute this code in a browser environment, it will output the length of the string `"Hello, World!"` to the console.

For example, if you run the above code, it will print:

```

13

```

This indicates that the length of the string `"Hello, World!"` is 13 characters.

Q.49 Change all the string characters to capital letters using toUpperCase() method?

Answer:- To change all the characters of a string to capital letters in JavaScript, you can use the `toUpperCase()` method. This method converts all the characters in a string to uppercase and returns the resulting string. Here's how you can use it:

```javascript

const str = "hello, world!";

const upperCaseStr = str.toUpperCase();

console.log(upperCaseStr);

```

This code will print `"HELLO, WORLD!"` to the console.

The `toUpperCase()` method does not modify the original string; it returns a new string with all characters converted to uppercase. If you want to store the result, you need to assign it to a variable, as shown in the example.

Q.50 What is the drawback of declaring methods directly in JavaScript objects?

Answer:- One drawback of declaring methods directly in JavaScript objects is that it creates a new function object for each instance of the object. This can lead to memory inefficiency, especially if the method performs complex operations or is frequently called.

When a method is declared directly in an object literal or added to an object dynamically, each instance of that object will have its own copy of the method. This means that memory will be allocated for each instance's method, even though the functionality is identical across instances.

Consider the following example:

```javascript

// Object constructor function

function Person(name, age) {

this.name = name;

this.age = age;

this.sayHello = function() {

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

};

}

// Create two instances of the Person object

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

const person2 = new Person('Bob', 25);

// Both instances have their own copy of the sayHello method

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

person2.sayHello(); // Output: Hello, my name is Bob and I am 25 years old.

```

In this example, each `Person` object created with the constructor function has its own `sayHello` method. If there are many instances of `Person`, each with its own method, it can lead to increased memory usage.

To mitigate this drawback, you can define methods on the prototype of the constructor function instead. This way, all instances of the object share the same method, reducing memory usage.

Q.51 Write a JavaScript program to get the current date. Expected Output : mm-dd-yyyy, mm/dd/yyyy or dd-mm-yyyy, dd/mm/yyyy?

Answer:- You can get the current date in JavaScript using the `Date` object and then format it according to your desired format. Here's a JavaScript program that gets the current date and formats it in the specified formats (mm-dd-yyyy, mm/dd/yyyy, dd-mm-yyyy, dd/mm/yyyy):

```javascript

function getCurrentDate(format = 'mm-dd-yyyy') {

const now = new Date();

let month = now.getMonth() + 1;

let day = now.getDate();

const year = now.getFullYear();

// Add leading zeros to month and day if necessary

month = month < 10 ? '0' + month : month;

day = day < 10 ? '0' + day : day;

if (format === 'mm-dd-yyyy') {

return `${month}-${day}-${year}`;

} else if (format === 'mm/dd/yyyy') {

return `${month}/${day}/${year}`;

} else if (format === 'dd-mm-yyyy') {

return `${day}-${month}-${year}`;

} else if (format === 'dd/mm/yyyy') {

return `${day}/${month}/${year}`;

} else {

return 'Invalid format specified';

}

}

// Example usage:

console.log(getCurrentDate('mm-dd-yyyy')); // Output: "05-30-2024"

console.log(getCurrentDate('mm/dd/yyyy')); // Output: "05/30/2024"

console.log(getCurrentDate('dd-mm-yyyy')); // Output: "30-05-2024"

console.log(getCurrentDate('dd/mm/yyyy')); // Output: "30/05/2024"

```

In this program:

- We use the `Date` object to get the current date.

- We extract the month, day, and year components.

- Depending on the `format` parameter provided to the `getCurrentDate` function, we format the date accordingly.

- We return the formatted date string.

You can call `getCurrentDate()` with different format options to get the date in various formats as specified.

Q.52 Use indexOf to determine the position of the first occurrence of a in 30 Days Of JavaScript?

Answer:- To use the `indexOf` method to determine the position of the first occurrence of a character (such as 'a') in a string, you can simply call `indexOf('a')` on the string. Here's how you can do it:

```javascript

const text = "30 Days Of JavaScript";

const position = text.indexOf('a');

console.log("Position of the first occurrence of 'a':", position);

```

This code will search for the first occurrence of the character 'a' in the string "30 Days Of JavaScript" and print its position (index) to the console.

If 'a' is found in the string, it will output the index of its first occurrence. If 'a' is not found, it will return -1.

For example, if 'a' is found at index 8 in the string "30 Days Of JavaScript", the output will be:

```

Position of the first occurrence of 'a': 8

```

Q,53 Use lastIndexOf to determine the position of the last occurrence of a in 30 Days Of JavaScript?

Answer:- To use the `lastIndexOf` method to determine the position of the last occurrence of a character (such as 'a') in a string, you can call `lastIndexOf('a')` on the string. Here's how you can do it:

```javascript

const text = "30 Days Of JavaScript";

const position = text.lastIndexOf('a');

console.log("Position of the last occurrence of 'a':", position);

```

This code will search for the last occurrence of the character 'a' in the string "30 Days Of JavaScript" and print its position (index) to the console.

If 'a' is found in the string, it will output the index of its last occurrence. If 'a' is not found, it will return -1.

For example, if 'a' is found at index 20 in the string "30 Days Of JavaScript", the output will be:

```

Position of the last occurrence of 'a': 20

```

Q.54 Form Validtion in JS?

Answer:- Form validation in JavaScript is the process of ensuring that user input in HTML forms meets certain criteria or constraints before it is submitted to the server. This is commonly done to prevent users from entering invalid data and to provide feedback to users about the correctness of their input.

Here are the general steps for form validation in JavaScript:

1. \*\*Accessing Form Elements\*\*: Use JavaScript to access the form and its input fields using DOM manipulation methods such as `getElementById`, `querySelector`, or `getElementsByName`.

2. \*\*Defining Validation Rules\*\*: Determine the validation rules for each input field. These rules define what constitutes valid input for each field. Common validation rules include:

- Required fields: Check if the input field is empty.

- Format validation: Validate input format (e.g., email format, phone number format).

- Length validation: Validate input length (e.g., minimum and maximum length).

- Numeric validation: Validate if input is numeric.

- Custom validation: Implement custom validation logic as needed.

3. \*\*Handling Form Submission\*\*: Attach an event listener to the form's submit event. When the form is submitted, the event listener function should perform the following tasks:

- Prevent the default form submission behavior using `event.preventDefault()` to prevent the form from being submitted to the server.

- Validate each input field according to the defined validation rules.

- Display error messages for invalid input fields.

- If all input fields are valid, allow the form submission to proceed. Otherwise, prevent the form submission and prompt the user to correct the errors.

4. \*\*Displaying Error Messages\*\*: Use JavaScript to dynamically create or modify elements to display error messages next to the input fields with invalid input. Error messages should provide clear instructions on how to correct the errors.

5. \*\*Feedback to Users\*\*: Provide feedback to users indicating whether their input is valid or invalid. This can include changing the appearance of input fields (e.g., changing border color) or displaying error messages.

6. \*\*Testing\*\*: Test the form validation thoroughly to ensure that it functions correctly under various scenarios and edge cases.

Here's a simplified example demonstrating form validation in JavaScript:

```html

<!DOCTYPE html>

<html>

<head>

<title>Form Validation</title>

<style>

.error {

color: red;

}

</style>

</head>

<body>

<form id="myForm">

<label for="username">Username:</label>

<input type="text" id="username" name="username"><span id="usernameError" class="error"></span><br>

<label for="email">Email:</label>

<input type="email" id="email" name="email"><span id="emailError" class="error"></span><br>

<input type="submit" value="Submit">

</form>

<script>

document.getElementById('myForm').addEventListener('submit', function(event) {

// Prevent form submission

event.preventDefault();

// Get form input values

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

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

// Validate username

if (username.trim() === '') {

document.getElementById('usernameError').textContent = 'Username is required';

} else {

document.getElementById('usernameError').textContent = '';

}

// Validate email

if (email.trim() === '') {

document.getElementById('emailError').textContent = 'Email is required';

} else if (!isValidEmail(email)) {

document.getElementById('emailError').textContent = 'Invalid email format';

} else {

document.getElementById('emailError').textContent = '';

}

// If all validations pass, submit the form

if (username.trim() !== '' && isValidEmail(email)) {

this.submit();

}

});

// Email validation function

function isValidEmail(email) {

const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

return emailRegex.test(email);

}

</script>

</body>

</html>

```

In this example, we perform basic validation for a username and an email address. When the form is submitted, JavaScript checks if the username and email fields are empty and if the email address has a valid format. Error messages are displayed next to the input fields with invalid input.

Q.55 Form in Email, number, Password, Validation?

Answer:- To create a form with validation for email, number, and password fields in HTML and JavaScript, you can follow these steps:

1. \*\*Create the HTML Form\*\*:

- Define a `<form>` element with input fields for email, number, and password.

- Add appropriate attributes such as `type`, `id`, `name`, and `placeholder` to the input fields.

- Optionally, include elements for displaying validation messages.

2. \*\*Define Validation Rules\*\*:

- Determine the validation rules for each input field.

- Use JavaScript to define functions for validating email format, number format, and password strength.

3. \*\*Handle Form Submission\*\*:

- Attach an event listener to the form's `submit` event.

- Validate the input fields when the form is submitted.

- Prevent the form from being submitted to the server if validation fails.

4. \*\*Display Validation Messages\*\*:

- Use JavaScript to dynamically update the content or visibility of elements that display validation messages.

- Show error messages next to the input fields with invalid input.

Here's an example of a form with validation for email, number, and password fields:

```html

<!DOCTYPE html>

<html>

<head>

<title>Form Validation</title>

<style>

.error {

color: red;

font-size: 0.8em;

}

</style>

</head>

<body>

<form id="myForm">

<label for="email">Email:</label>

<input type="email" id="email" name="email" placeholder="Enter your email"><span id="emailError" class="error"></span><br>

<label for="number">Number:</label>

<input type="number" id="number" name="number" placeholder="Enter a number"><span id="numberError" class="error"></span><br>

<label for="password">Password:</label>

<input type="password" id="password" name="password" placeholder="Enter a password"><span id="passwordError" class="error"></span><br>

<input type="submit" value="Submit">

</form>

<script>

document.getElementById('myForm').addEventListener('submit', function(event) {

// Prevent form submission

event.preventDefault();

// Get form input values

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

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

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

// Validate email

if (email.trim() === '') {

document.getElementById('emailError').textContent = 'Email is required';

} else if (!isValidEmail(email)) {

document.getElementById('emailError').textContent = 'Invalid email format';

} else {

document.getElementById('emailError').textContent = '';

}

// Validate number

if (number.trim() === '') {

document.getElementById('numberError').textContent = 'Number is required';

} else if (!isValidNumber(number)) {

document.getElementById('numberError').textContent = 'Invalid number format';

} else {

document.getElementById('numberError').textContent = '';

}

// Validate password

if (password.trim() === '') {

document.getElementById('passwordError').textContent = 'Password is required';

} else if (password.length < 8) {

document.getElementById('passwordError').textContent = 'Password must be at least 8 characters long';

} else {

document.getElementById('passwordError').textContent = '';

}

// If all validations pass, submit the form

if (isValidEmail(email) && isValidNumber(number) && password.length >= 8) {

this.submit();

}

});

// Email validation function

function isValidEmail(email) {

const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

return emailRegex.test(email);

}

// Number validation function

function isValidNumber(number) {

// You can add custom validation logic here (e.g., check if it's a valid number format)

return !isNaN(number);

}

</script>

</body>

</html>

```

In this example:

- Email, number, and password input fields are defined within a `<form>` element.

- JavaScript functions are used to validate each input field according to defined rules.

- Validation messages are displayed next to each input field with invalid input.

- The form is prevented from being submitted to the server if validation fails.

You can customize the validation rules and error messages based on your requirements.

Q.56 Dynamic Form Validation in JS?

Answer:- Dynamic form validation in JavaScript involves continuously validating form input as the user interacts with the form. This ensures that immediate feedback is provided to the user, improving the user experience and reducing errors.

Here's a general approach to implement dynamic form validation in JavaScript:

1. \*\*Attach Event Listeners\*\*: Attach event listeners to form input fields to listen for user interactions such as typing, focus, blur, etc.

2. \*\*Validate Input on Events\*\*: When the user interacts with an input field, validate its value based on predefined validation rules. This can be done in response to events like `input`, `focus`, `blur`, etc.

3. \*\*Display Validation Feedback\*\*: Provide immediate feedback to the user by dynamically updating the UI to indicate whether the input is valid or invalid. This can include changing the color of the input field, displaying error messages, or showing checkmarks for valid input.

4. \*\*Update Validation on Form Submission\*\*: Additionally, perform validation on form submission to ensure that all fields are valid before allowing the form to be submitted to the server.

5. \*\*Handle Edge Cases\*\*: Consider edge cases such as empty fields, special characters, or specific input formats that may need custom validation logic.

Here's a simplified example demonstrating dynamic form validation in JavaScript:

```html

<!DOCTYPE html>

<html>

<head>

<title>Dynamic Form Validation</title>

<style>

.error {

color: red;

}

.valid {

color: green;

}

</style>

</head>

<body>

<form id="myForm">

<label for="username">Username:</label>

<input type="text" id="username" name="username">

<span id="usernameError" class="error"></span><br>

<label for="email">Email:</label>

<input type="email" id="email" name="email">

<span id="emailError" class="error"></span><br>

<label for="password">Password:</label>

<input type="password" id="password" name="password">

<span id="passwordError" class="error"></span><br>

<input type="submit" value="Submit">

</form>

<script>

const form = document.getElementById('myForm');

// Validate username on input

document.getElementById('username').addEventListener('input', function() {

const username = this.value.trim();

const usernameError = document.getElementById('usernameError');

if (username === '') {

usernameError.textContent = 'Username is required';

} else {

usernameError.textContent = '';

}

});

// Validate email on input

document.getElementById('email').addEventListener('input', function() {

const email = this.value.trim();

const emailError = document.getElementById('emailError');

if (email === '') {

emailError.textContent = 'Email is required';

} else if (!isValidEmail(email)) {

emailError.textContent = 'Invalid email format';

} else {

emailError.textContent = '';

}

});

// Validate password on input

document.getElementById('password').addEventListener('input', function() {

const password = this.value.trim();

const passwordError = document.getElementById('passwordError');

if (password === '') {

passwordError.textContent = 'Password is required';

} else if (password.length < 8) {

passwordError.textContent = 'Password must be at least 8 characters long';

} else {

passwordError.textContent = '';

}

});

// Form submission validation

form.addEventListener('submit', function(event) {

event.preventDefault();

// Perform final validation before submission

const username = document.getElementById('username').value.trim();

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

const password = document.getElementById('password').value.trim();

if (username === '') {

document.getElementById('usernameError').textContent = 'Username is required';

}

if (email === '') {

document.getElementById('emailError').textContent = 'Email is required';

} else if (!isValidEmail(email)) {

document.getElementById('emailError').textContent = 'Invalid email format';

}

if (password === '') {

document.getElementById('passwordError').textContent = 'Password is required';

} else if (password.length < 8) {

document.getElementById('passwordError').textContent = 'Password must be at least 8 characters long';

}

// If all validations pass, submit the form

if (username !== '' && isValidEmail(email) && password !== '' && password.length >= 8) {

this.submit();

}

});

// Email validation function

function isValidEmail(email) {

const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

return emailRegex.test(email);

}

</script>

</body>

</html>

```

In this example:

- Event listeners are attached to input fields to listen for input events.

- Validation functions are called dynamically as the user interacts with the form.

- Feedback is provided to the user in real-time by updating error messages next to each input field.

- Final validation is performed on form submission to ensure all fields are valid before allowing the form to be submitted to the server.

Q.57 how many type of JS Event? How to use it ?

Answer:- JavaScript events are actions that occur as a result of user interactions with a web page or as a result of other occurrences, such as the page loading or the browser window being resized. There are various types of JavaScript events, categorized based on their triggers. Here are some common types of JavaScript events and how to use them:

1. \*\*Mouse Events\*\*:

- `click`: Occurs when the user clicks on an element.

- `mouseover`: Occurs when the mouse pointer moves over an element.

- `mouseout`: Occurs when the mouse pointer moves out of an element.

- `mousedown`: Occurs when the mouse button is pressed down over an element.

- `mouseup`: Occurs when the mouse button is released over an element.

Example usage:

```javascript

document.getElementById('myButton').addEventListener('click', function() {

// Do something when the button is clicked

});

```

2. \*\*Keyboard Events\*\*:

- `keydown`: Occurs when a key is pressed down.

- `keypress`: Occurs when a key is pressed and released.

- `keyup`: Occurs when a key is released.

Example usage:

```javascript

document.addEventListener('keydown', function(event) {

// Do something when a key is pressed down

});

```

3. \*\*Form Events\*\*:

- `submit`: Occurs when a form is submitted.

- `change`: Occurs when the value of a form element changes.

- `input`: Occurs when the value of an input element changes (similar to `change` but fires immediately).

Example usage:

```javascript

document.getElementById('myForm').addEventListener('submit', function(event) {

// Prevent form submission

event.preventDefault();

// Validate form input

// ...

});

```

4. \*\*Window Events\*\*:

- `load`: Occurs when the page and all its resources have finished loading.

- `resize`: Occurs when the browser window is resized.

- `scroll`: Occurs when the user scrolls the page.

Example usage:

```javascript

window.addEventListener('load', function() {

// Do something when the page is fully loaded

});

```

5. \*\*Focus Events\*\*:

- `focus`: Occurs when an element gains focus.

- `blur`: Occurs when an element loses focus.

Example usage:

```javascript

document.getElementById('myInput').addEventListener('focus', function() {

// Do something when the input element gains focus

});

```

6. \*\*Custom Events\*\*:

- Developers can also create custom events using the `CustomEvent` constructor and dispatch them on DOM elements.

Example usage:

```javascript

// Create a custom event

const event = new CustomEvent('myCustomEvent', { detail: { data: 'example' } });

// Dispatch the custom event on an element

document.getElementById('myElement').dispatchEvent(event);

// Listen for the custom event

document.getElementById('myElement').addEventListener('myCustomEvent', function(event) {

console.log(event.detail.data); // Output: "example"

});

```

These are just a few examples of the many types of JavaScript events available. Each event type has its own set of properties and methods that can be used to interact with it. Event listeners are attached to DOM elements using the `addEventListener` method, allowing you to execute JavaScript code in response to specific events.

Q.59 What is Bom vs Dom in JS?

Answer:- BOM (Browser Object Model) and DOM (Document Object Model) are both essential components of client-side web development in JavaScript, but they serve different purposes.

1. \*\*DOM (Document Object Model)\*\*:

- The DOM represents the structure of an HTML document as a tree-like structure.

- It provides a way for JavaScript to interact with the content of a web page dynamically.

- With the DOM, you can access, modify, add, or delete HTML elements and their attributes, styles, and content.

- The DOM is focused on the document structure and content, allowing developers to manipulate the elements on a web page.

- Example DOM operations include selecting elements, changing text or attributes, adding event listeners, and creating new elements.

2. \*\*BOM (Browser Object Model)\*\*:

- The BOM represents everything else provided by the browser, including browser-specific objects and methods.

- It provides interfaces for interacting with the browser itself, such as manipulating the browser window, managing history, handling cookies, and displaying alerts or prompts.

- Unlike the DOM, which focuses on the document content, the BOM focuses on the browser environment.

- BOM objects and methods are not standardized across browsers and may vary in functionality and support.

- Example BOM operations include controlling the browser window size and position, navigating history, managing cookies, and accessing browser-specific features like geolocation and local storage.

In summary, the DOM is primarily concerned with representing the structure and content of an HTML document and provides APIs for manipulating it, while the BOM deals with browser-specific functionality and interactions beyond the document content, such as controlling the browser window and managing browser history. Both DOM and BOM are integral parts of client-side web development in JavaScript, enabling rich and interactive web applications.

Q.60 Array vs object defences in JS?

Answer:- In JavaScript, arrays and objects are two fundamental data structures, each with its own characteristics and use cases.

\*\*Arrays\*\*:

- Arrays are ordered collections of elements, indexed by non-negative integers starting from 0.

- They can store elements of any data type, including other arrays and objects.

- Arrays are versatile and commonly used for storing lists of similar items or sequential data.

- Arrays offer various built-in methods for manipulation and iteration, such as `push`, `pop`, `shift`, `unshift`, `splice`, `forEach`, `map`, `filter`, etc.

- Accessing elements in an array by index is fast, as arrays are optimized for random access.

- Arrays are suitable for situations where data needs to be accessed and processed in a sequential or ordered manner.

Example:

```javascript

const fruits = ['apple', 'banana', 'orange'];

console.log(fruits[0]); // Output: 'apple'

console.log(fruits.length); // Output: 3

```

\*\*Objects\*\*:

- Objects are collections of key-value pairs, where each key is a unique string (or symbol in ES6) and each value can be of any data type, including other objects and arrays.

- Objects represent unordered data structures, as the order of key-value pairs may not be preserved.

- They are versatile and used for storing structured data and modeling real-world entities.

- Objects are particularly useful for representing complex data with named properties.

- Accessing properties in an object by key is efficient, as objects are optimized for property lookup.

- Objects can have methods, which are functions stored as object properties.

- Objects can also be used to create custom data structures and implement various design patterns.

Example:

```javascript

const person = {

name: 'John',

age: 30,

address: {

city: 'New York',

country: 'USA'

},

greet: function() {

console.log(`Hello, my name is ${this.name}`);

}

};

console.log(person.name); // Output: 'John'

console.log(person.address.city); // Output: 'New York'

person.greet(); // Output: 'Hello, my name is John'

```

In summary, arrays are suitable for storing ordered collections of similar items, while objects are ideal for representing structured data with named properties. Both arrays and objects are powerful tools in JavaScript, and choosing the appropriate data structure depends on the specific requirements of the task at hand.

Q.61 Split the string into an array using split() Method?

Answer:- The `split()` method in JavaScript is used to split a string into an array of substrings based on a specified separator. Here's how you can use it:

```javascript

const str = "Hello,World,JavaScript";

const arr = str.split(',');

console.log(arr); // Output: ["Hello", "World", "JavaScript"]

```

In this example:

- We have a string `str` containing the text "Hello,World,JavaScript".

- We use the `split(',')` method to split the string into an array of substrings using the comma `,` as the separator.

- The resulting array `arr` contains the substrings "Hello", "World", and "JavaScript".

You can use any string as the separator for splitting the original string. If the separator is not specified, the entire string will be treated as a single element in the resulting array.

Q.62 Check if the string contains a word Script using includes() method?

Answer:- You can use the `includes()` method in JavaScript to check if a string contains a specific substring or word. Here's how you can use it to check if a string contains the word "Script":

```javascript

const str = "JavaScript is a scripting language.";

const wordToCheck = "Script";

if (str.includes(wordToCheck)) {

console.log(`The string contains the word '${wordToCheck}'.`);

} else {

console.log(`The string does not contain the word '${wordToCheck}'.`);

}

```

In this example:

- We have a string `str` containing the text "JavaScript is a scripting language."

- We use the `includes()` method to check if the string contains the word "Script".

- If the word "Script" is found within the string, `includes()` returns `true`, and we print a message indicating that the word is present.

- If the word "Script" is not found within the string, `includes()` returns `false`, and we print a message indicating that the word is not present.

So, the output of the above code will be:

```

The string contains the word 'Script'.

```

If you want the search to be case-insensitive, you can convert both the string and the word to lowercase or uppercase before using `includes()`.

Q.63 Change all the string characters to lowercase letters using toLowerCase() Method.

Answer:- You can use the `toLowerCase()` method in JavaScript to convert all characters in a string to lowercase. Here's how you can use it:

```javascript

const str = "Hello, World!";

const lowerCaseStr = str.toLowerCase();

console.log(lowerCaseStr); // Output: "hello, world!"

```

In this example:

- We have a string `str` containing the text "Hello, World!".

- We use the `toLowerCase()` method to convert all characters in the string to lowercase.

- The result is stored in the variable `lowerCaseStr`, which contains the lowercase version of the original string.

- When we log `lowerCaseStr` to the console, it prints "hello, world!", where all characters are in lowercase.

Q.64 What is Character at index 15 in ’30 Days of JavaScript’ string? Use charAt() method.

Answer:- You can use the `charAt()` method in JavaScript to retrieve the character at a specific index in a string. Here's how you can use it to find the character at index 15 in the string "30 Days of JavaScript":

```javascript

const str = '30 Days of JavaScript';

const charAtIndex15 = str.charAt(15);

console.log(charAtIndex15); // Output: "S"

```

In this example:

- We have a string `str` containing the text "30 Days of JavaScript".

- We use the `charAt(15)` method to retrieve the character at index 15 in the string.

- The character at index 15 in the string "30 Days of JavaScript" is "S", which is stored in the variable `charAtIndex15`.

- When we log `charAtIndex15` to the console, it prints "S".

Q.65 copy to one string to another string in JS?

Answer:- You can copy the contents of one string to another string in JavaScript using simple assignment or concatenation. Here are two common methods:

1. \*\*Using Simple Assignment\*\*:

```javascript

const originalString = 'Hello, World!';

const copiedString = originalString;

console.log(copiedString); // Output: "Hello, World!"

```

2. \*\*Using Concatenation\*\*:

```javascript

const originalString = 'Hello, World!';

const copiedString = '' + originalString; // Concatenate an empty string with the original string

console.log(copiedString); // Output: "Hello, World!"

```

Both of these methods create a copy of the original string and assign it to a new variable. The copied string will contain the same characters as the original string.

Q.65 copy to one string to another string in JS?

Answer:- In JavaScript, you can copy one string to another string simply by using the assignment operator. Here is an example:

```javascript

// Original string

let originalString = "Hello, world!";

// Copy the string to another variable

let copiedString = originalString;

// Output the copied string

console.log(copiedString); // Output: Hello, world!

```

In this example, `originalString` contains the original string, and `copiedString` is assigned the value of `originalString`. Both variables will contain the same string value, but they are separate variables, so changes to one will not affect the other.

If you need to demonstrate a more explicit method of copying strings (though unnecessary in JavaScript), you could create a new string object:

```javascript

// Original string

let originalString = "Hello, world!";

// Create a new string object explicitly (uncommon and not recommended)

let copiedString = new String(originalString);

// Output the copied string

console.log(copiedString); // Output: Hello, world!

```

However, this is rarely used in practice because the first method is simpler and more idiomatic in JavaScript.

Q.66 Find the length of a string without using libraryFunction?

Answer:- To find the length of a string without using the built-in `length` property in JavaScript, you can iterate through the string and count the number of characters manually. Here's an example of how you can do this:

```javascript

function getStringLength(str) {

let length = 0;

while (true) {

if (str[length] === undefined) {

break;

}

length++;

}

return length;

}

// Test the function

let myString = "Hello, world!";

console.log(getStringLength(myString)); // Output: 13

```

In this example, the `getStringLength` function initializes a `length` variable to 0 and increments it until it encounters an `undefined` value, which indicates the end of the string. This way, it effectively counts the number of characters in the string without using the `length` property.

• What is JavaScript?

Answer:- JavaScript is a versatile, high-level, interpreted programming language primarily used for enhancing web pages and creating interactive web applications. It is one of the core technologies of the World Wide Web, alongside HTML and CSS. Here are some key points about JavaScript:

1. \*\*Client-Side Language\*\*: JavaScript is primarily used on the client side (i.e., in the browser) to create dynamic and interactive user interfaces. It allows developers to manipulate the Document Object Model (DOM) to change the content, structure, and style of web pages without reloading them.

2. \*\*Versatility\*\*: While traditionally a client-side language, JavaScript can also be used on the server side thanks to environments like Node.js. This makes it possible to write full-stack applications using JavaScript alone.

3. \*\*Interpreted Language\*\*: JavaScript code is executed line by line by the web browser’s JavaScript engine, without the need for prior compilation.

4. \*\*Event-Driven\*\*: JavaScript excels in handling events (like user inputs, clicks, and other interactions), which is crucial for creating responsive and interactive web applications.

5. \*\*Object-Oriented\*\*: JavaScript supports object-oriented programming principles, allowing developers to create reusable code with objects and classes.

6. \*\*Prototypal Inheritance\*\*: Unlike classical inheritance in languages like Java, JavaScript uses prototypal inheritance, where objects inherit directly from other objects.

7. \*\*Widely Supported\*\*: All modern web browsers come with built-in JavaScript engines, ensuring broad support and compatibility across different platforms and devices.

8. \*\*Standardization\*\*: JavaScript is standardized by the ECMAScript specification, with regular updates (e.g., ES5, ES6/ES2015, and subsequent versions) that add new features and improvements to the language.

9. \*\*Extensive Ecosystem\*\*: JavaScript has a rich ecosystem of libraries and frameworks, such as React, Angular, Vue.js, and jQuery, which simplify and enhance web development.

10. \*\*Community and Resources\*\*: JavaScript has a large, active community and extensive resources, including documentation, tutorials, and forums, making it easier for developers to learn and troubleshoot.

Overall, JavaScript plays a crucial role in modern web development, enabling developers to create rich, interactive, and dynamic web applications.

• What is the use of isNaN function?

Answer:- The `isNaN` function in JavaScript is used to determine whether a value is `NaN` (Not-a-Number) or not. `NaN` is a special value in JavaScript that represents a value which is not a legal number.

### Syntax

```javascript

isNaN(value)

```

- `value`: The value to be tested.

### Description

- The `isNaN` function attempts to convert the given value to a number and then checks if it is `NaN`.

- It returns `true` if the value is `NaN`, and `false` otherwise.

### Use Cases

1. \*\*Checking Non-Numeric Values\*\*: You can use `isNaN` to check if a value is not a number.

2. \*\*Form Validation\*\*: In web forms, it can be used to validate if the input values are numeric or not.

### Examples

1. \*\*Basic Usage\*\*:

```javascript

console.log(isNaN(123)); // false (123 is a number)

console.log(isNaN('123')); // false ('123' can be converted to a number)

console.log(isNaN('Hello')); // true ('Hello' cannot be converted to a number)

console.log(isNaN(true)); // false (true is converted to 1)

console.log(isNaN(false)); // false (false is converted to 0)

console.log(isNaN(NaN)); // true (NaN is not a number)

```

2. \*\*Edge Cases\*\*:

```javascript

console.log(isNaN(undefined)); // true (undefined is not a number)

console.log(isNaN(null)); // false (null is converted to 0)

console.log(isNaN('')); // false (empty string is converted to 0)

console.log(isNaN([])); // false (empty array is converted to 0)

console.log(isNaN([1,2])); // true (array with multiple values cannot be converted to a single number)

```

### Important Note

- The `isNaN` function coerces the value to a number before checking if it is `NaN`. This can sometimes lead to unexpected results due to type coercion.

- To avoid issues with type coercion, you can use `Number.isNaN`, which does not coerce the value:

```javascript

console.log(Number.isNaN(NaN)); // true

console.log(Number.isNaN('Hello')); // false (because 'Hello' is not coerced to NaN)

```

### Summary

The `isNaN` function is a useful tool for checking whether a value is not a number, especially when validating inputs or processing data that may not always be numeric. However, be aware of its behavior with type coercion and consider using `Number.isNaN` for more precise checks.

• What is negative Infinity?

Answer:- In JavaScript, `negative Infinity` (or `-Infinity`) is a special numeric value that represents negative infinity. It is a property of the global `Number` object and can also be produced by dividing a negative number by zero.

### Characteristics of `negative Infinity`

1. \*\*Value Type\*\*: `-Infinity` is of type `number`.

2. \*\*Mathematical Concept\*\*: It represents a value that is lower than any other number.

3. \*\*Operations\*\*:

- Any positive number divided by zero results in `Infinity`.

- Any negative number divided by zero results in `-Infinity`.

- Multiplying `-Infinity` by any positive number results in `-Infinity`.

- Multiplying `-Infinity` by any negative number results in `Infinity`.

4. \*\*Comparison\*\*: `-Infinity` is considered smaller than any other numeric value.

### Examples

1. \*\*Basic Usage\*\*:

```javascript

console.log(-Infinity); // Output: -Infinity

console.log(typeof -Infinity); // Output: "number"

```

2. \*\*Operations with `-Infinity`\*\*:

```javascript

console.log(-1 / 0); // Output: -Infinity

console.log(-Infinity < 0); // Output: true

console.log(-Infinity < -1000);// Output: true

console.log(-Infinity \* 2); // Output: -Infinity

console.log(-Infinity \* -1); // Output: Infinity

```

3. \*\*Comparison\*\*:

```javascript

console.log(-Infinity < Number.MIN\_VALUE); // Output: true

console.log(-Infinity < -Number.MAX\_VALUE); // Output: true

```

### Practical Use Cases

1. \*\*Error Handling\*\*: In algorithms involving large or infinite limits, `-Infinity` can be used to initialize variables for comparison.

```javascript

let minValue = -Infinity;

let values = [100, 50, 200, -300, 400];

for (let value of values) {

if (value > minValue) {

minValue = value;

}

}

console.log(minValue); // Output: 400

```

2. \*\*Boundary Conditions\*\*: `-Infinity` can be used to represent lower bounds in mathematical computations or simulations.

### Summary

`negative Infinity` (`-Infinity`) in JavaScript is a special numeric value used to represent the concept of infinity in the negative direction. It is useful in various mathematical and computational scenarios, particularly when dealing with limits, error handling, and boundary conditions.

• Which company developed JavaScript?

Answer:- JavaScript was developed by Netscape Communications Corporation. The language was created by Brendan Eich in 1995 while he was working at Netscape. It was initially designed to be a lightweight scripting language for the Netscape Navigator web browser, enabling dynamic and interactive elements on web pages.

### Brief History

1. \*\*Development\*\*: Brendan Eich developed JavaScript in just 10 days in May 1995. It was originally called "Mocha," then "LiveScript," and finally renamed "JavaScript" to capitalize on the popularity of Java at the time.

2. \*\*Release\*\*: JavaScript was first introduced in Netscape Navigator 2.0 in September 1995.

3. \*\*Standardization\*\*: To ensure the language's widespread adoption and consistency across different browsers, JavaScript was submitted to ECMA International for standardization. The standardized version is known as ECMAScript. The first edition of ECMAScript was published in 1997.

### Key Points

- \*\*Netscape Communications Corporation\*\*: The company responsible for the initial development and release of JavaScript.

- \*\*Brendan Eich\*\*: The programmer who created JavaScript.

- \*\*Standardization\*\*: JavaScript was standardized under the name ECMAScript by ECMA International.

JavaScript has since become one of the most widely used programming languages in the world, essential for web development alongside HTML and CSS.

• What are undeclared and undefined variables?

Answer:- In JavaScript, understanding the difference between undeclared and undefined variables is important for effective debugging and coding practices. Here’s a detailed explanation:

### Undeclared Variables

\*\*Undeclared variables\*\* are those that have not been declared in any scope (using `var`, `let`, or `const`) before they are used. Attempting to use an undeclared variable results in a reference error.

\*\*Example\*\*:

```javascript

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

function foo() {

b = 10; // `b` is not declared with var, let, or const

}

foo();

console.log(b); // 10, b becomes a global variable due to lack of declaration (non-strict mode)

```

In strict mode (`'use strict';`), assigning a value to an undeclared variable will throw an error.

```javascript

'use strict';

function foo() {

b = 10; // ReferenceError: b is not defined

}

foo();

```

### Undefined Variables

\*\*Undefined variables\*\* are variables that have been declared but have not been assigned a value. The default value of such variables is `undefined`.

\*\*Example\*\*:

```javascript

var x;

console.log(x); // undefined

let y;

console.log(y); // undefined

const z;

console.log(z); // SyntaxError: Missing initializer in const declaration

```

In the example, `x` and `y` are declared but not initialized, so they are `undefined`. However, note that `const` variables must be initialized at the time of declaration, otherwise a syntax error occurs.

### Summary of Differences

1. \*\*Undeclared Variables\*\*:

- Not declared with `var`, `let`, or `const`.

- Accessing an undeclared variable throws a `ReferenceError`.

- In non-strict mode, assigning a value to an undeclared variable makes it a global variable.

- In strict mode, assigning to an undeclared variable throws a `ReferenceError`.

2. \*\*Undefined Variables\*\*:

- Declared with `var`, `let`, or `const` but not assigned a value.

- The default value is `undefined`.

- Can be explicitly set to `undefined`.

### Code Example to Demonstrate Both Concepts

```javascript

// Undeclared variable

try {

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

} catch (e) {

console.log(e.message);

}

// Declaring and using an undefined variable

let b;

console.log(b); // undefined

// Assigning value to an undeclared variable (non-strict mode)

function example() {

c = 5; // `c` becomes a global variable

}

example();

console.log(c); // 5

// Using 'use strict' to catch undeclared variables

'use strict';

function strictExample() {

try {

d = 10; // ReferenceError: d is not defined

} catch (e) {

console.log(e.message);

}

}

strictExample();

```

Understanding the distinction between undeclared and undefined variables is crucial for avoiding common JavaScript pitfalls and writing robust, error-free code.

• Write the code for adding new elements dynamically?

Answer:- Adding new elements dynamically to the DOM (Document Object Model) can be achieved using JavaScript. Here's a step-by-step guide and example code to demonstrate how you can create and append new elements dynamically:

### Example: Adding a New List Item to an Unordered List

This example demonstrates how to add new items to an unordered list (`<ul>`).

1. \*\*HTML Structure\*\*: Create a simple HTML structure with a button and an unordered list.

2. \*\*JavaScript\*\*: Write JavaScript to handle the button click event, create a new list item, and append it to the list.

### HTML

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Dynamic Elements Example</title>

</head>

<body>

<h1>Dynamic Elements Example</h1>

<button id="addButton">Add Item</button>

<ul id="itemList">

<li>Item 1</li>

<li>Item 2</li>

</ul>

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

</body>

</html>

```

### JavaScript (script.js)

```javascript

// Function to add a new list item

function addNewItem() {

// Get the unordered list (ul) element

const ul = document.getElementById('itemList');

// Create a new list item (li) element

const li = document.createElement('li');

// Create a text node with the content for the new list item

const textNode = document.createTextNode('New Item');

// Append the text node to the new list item

li.appendChild(textNode);

// Append the new list item to the unordered list

ul.appendChild(li);

}

// Get the button element

const addButton = document.getElementById('addButton');

// Attach an event listener to the button to call addNewItem function on click

addButton.addEventListener('click', addNewItem);

```

### Explanation

1. \*\*HTML Setup\*\*:

- An unordered list (`<ul>`) with the ID `itemList`.

- A button with the ID `addButton` to trigger the addition of new items.

2. \*\*JavaScript Code\*\*:

- \*\*Function `addNewItem`\*\*: This function performs the following steps:

- Selects the unordered list element using `document.getElementById('itemList')`.

- Creates a new list item (`<li>`) using `document.createElement('li')`.

- Creates a text node containing the text "New Item" using `document.createTextNode('New Item')`.

- Appends the text node to the new list item.

- Appends the new list item to the unordered list.

- \*\*Event Listener\*\*: Adds an event listener to the button which calls the `addNewItem` function whenever the button is clicked.

### Result

Each time the "Add Item" button is clicked, a new list item with the text "New Item" is appended to the unordered list.

You can further enhance this example by adding input fields for dynamic text entry, styling for the new elements, or more complex structures like forms and tables.

• What is the difference between ViewState and SessionState?

Answer:- In web development, particularly in the context of ASP.NET, `ViewState` and `SessionState` are used to manage state across web requests, but they serve different purposes and operate differently. Here’s a detailed comparison between the two:

### ViewState

1. \*\*Scope\*\*: `ViewState` is used to preserve the state of a web page (or controls on the page) between postbacks. It is specific to a single page.

2. \*\*Storage Location\*\*: The state information is stored in a hidden field on the page itself as a base64-encoded string. This data travels back and forth between the server and the client with each request and response.

3. \*\*Lifetime\*\*: The state is maintained only for the duration of the page’s lifecycle. Once the user navigates away from the page, the `ViewState` is lost.

4. \*\*Usage\*\*: Typically used to maintain control values, user inputs, and other page-specific data during postbacks within the same page.

5. \*\*Performance\*\*: Since `ViewState` data is sent with every request and response, it can affect page load times and network bandwidth if not managed properly.

6. \*\*Security\*\*: While the data is encoded, it is not encrypted by default. Sensitive data should not be stored in `ViewState` unless encryption is applied.

### Example of ViewState

```csharp

// Storing data in ViewState

ViewState["MyData"] = "Some data";

// Retrieving data from ViewState

string myData = ViewState["MyData"] as string;

```

### SessionState

1. \*\*Scope\*\*: `SessionState` is used to preserve user data across multiple requests and pages within a web application. It is specific to a user session.

2. \*\*Storage Location\*\*: The state information is stored on the server. The client is only given a session identifier (usually in a cookie) to retrieve the stored data.

3. \*\*Lifetime\*\*: The state is maintained for the duration of the user’s session, which can persist across multiple pages and visits until the session expires or is explicitly abandoned.

4. \*\*Usage\*\*: Typically used to store user-specific data that needs to be accessed across different pages, such as user authentication details, shopping cart contents, and user preferences.

5. \*\*Performance\*\*: Since `SessionState` is stored on the server, it does not affect the size of the page sent to the client. However, excessive use can impact server memory and performance.

6. \*\*Security\*\*: `SessionState` is generally more secure than `ViewState` because the data is stored on the server. However, proper security measures should still be implemented to protect session data.

### Example of SessionState

```csharp

// Storing data in SessionState

Session["UserData"] = "Some user-specific data";

// Retrieving data from SessionState

string userData = Session["UserData"] as string;

```

### Summary

- \*\*ViewState\*\*:

- Used for preserving state within a single page.

- Data stored in a hidden field on the page.

- Lifetime is limited to a single page lifecycle.

- Affects page load and network performance.

- \*\*SessionState\*\*:

- Used for preserving state across multiple pages and requests within a session.

- Data stored on the server.

- Lifetime extends across user sessions.

- Does not affect page load performance but impacts server memory.

Choosing between `ViewState` and `SessionState` depends on the scope and duration of the data you need to preserve. Use `ViewState` for page-specific state management during postbacks, and `SessionState` for user-specific data that needs to persist across multiple pages and visits.

• What is === operator?

Answer:- In JavaScript, the `===` operator is known as the strict equality operator. It is used to compare two values for equality, but unlike the `==` operator (loose equality), it does not perform type coercion. This means that for `===` to return `true`, the values being compared must be of the same type and have the same value.

### Characteristics of `===` (Strict Equality) Operator

1. \*\*Type Comparison\*\*: Both operands must be of the same type.

2. \*\*Value Comparison\*\*: Both operands must have the same value.

### Examples

1. \*\*Same Type and Value\*\*:

```javascript

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

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

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

```

2. \*\*Different Types\*\*:

```javascript

console.log(5 === '5'); // false (number vs string)

console.log(true === 1); // false (boolean vs number)

console.log(null === undefined); // false (null vs undefined)

```

3. \*\*Different Values\*\*:

```javascript

console.log(5 === 10); // false

console.log('hello' === 'world'); // false

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

```

### Comparison with `==` (Loose Equality)

The `==` operator performs type coercion, meaning it converts the operands to the same type before making the comparison. This can lead to unexpected results.

#### Examples:

- \*\*Type Coercion with `==`\*\*:

```javascript

console.log(5 == '5'); // true (string '5' is converted to number 5)

console.log(true == 1); // true (boolean true is converted to number 1)

console.log(null == undefined); // true (both are considered equal)

```

- \*\*No Type Coercion with `===`\*\*:

```javascript

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

console.log(true === 1); // false

console.log(null === undefined); // false

```

### When to Use `===`

- \*\*Best Practice\*\*: It is generally recommended to use `===` for comparisons to avoid unexpected results due to type coercion.

- \*\*Strict Type Checking\*\*: Ensures that the comparison is more predictable and only returns true when both the type and value are exactly the same.

### Summary

The `===` operator in JavaScript is used for strict equality comparisons, meaning both the type and value of the operands must be the same for the comparison to return true. This differs from the `==` operator, which performs type coercion before compare

son. Using `===` is a best practice to ensure accurate and predictable comparisons in your code.

• How can the style/class of an element be changed?

Answer:- You can change the style or class of an HTML element using JavaScript. Here are the common methods to achieve this:

### Changing Style

#### Using JavaScript:

1. \*\*Inline Style\*\*:

```javascript

// Change background color of an element with id "myElement"

document.getElementById("myElement").style.backgroundColor = "red";

```

2. \*\*Using `classList`\*\* (for adding/removing classes):

```javascript

// Add a class to an element with id "myElement"

document.getElementById("myElement").classList.add("newClass");

// Remove a class from an element with id "myElement"

document.getElementById("myElement").classList.remove("oldClass");

// Toggle a class on an element with id "myElement"

document.getElementById("myElement").classList.toggle("active");

```

### Changing Class

#### Using JavaScript:

1. \*\*Changing Class Directly\*\*:

```javascript

// Change the class of an element with id "myElement"

document.getElementById("myElement").className = "newClass";

```

2. \*\*Using `classList`\*\* (similar to changing style):

```javascript

// Add a class to an element with id "myElement"

document.getElementById("myElement").classList.add("newClass");

// Remove a class from an element with id "myElement"

document.getElementById("myElement").classList.remove("oldClass");

// Toggle a class on an element with id "myElement"

document.getElementById("myElement").classList.toggle("active");

```

### Example

Here's a complete example demonstrating how to change the style and class of an element using JavaScript:

#### HTML:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Change Style/Class Example</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<div id="myElement">Element with ID "myElement"</div>

<button onclick="changeStyle()">Change Style</button>

<button onclick="changeClass()">Change Class</button>

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

</body>

</html>

```

#### CSS (styles.css):

```css

/\* CSS for the initial style \*/

#myElement {

background-color: lightblue;

padding: 10px;

margin-top: 10px;

}

.newStyle {

background-color: lightgreen;

font-weight: bold;

}

.newClass {

color: red;

}

```

#### JavaScript (script.js):

```javascript

function changeStyle() {

// Change background color and add a new style

document.getElementById("myElement").style.backgroundColor = "lightgreen";

document.getElementById("myElement").classList.add("newStyle");

}

function changeClass() {

// Change class

document.getElementById("myElement").className = "newClass";

}

```

In this example, clicking the "Change Style" button will change the background color of the element with ID "myElement" and add a new style. Clicking the "Change Class" button will change the class of the element. You can adjust the styles and classes according to your requirements.

• How to read and write a file using JavaScript?

Answer:- In JavaScript, you can read and write files using different methods depending on the environment in which your code is running. Below are examples for both web browsers and Node.js.

### Reading and Writing Files in a Web Browser (Client-side JavaScript)

In a web browser, JavaScript does not have direct access to the file system for security reasons. However, you can interact with files through user interactions like file input elements or using Web APIs like the File and FileReader APIs.

#### Reading a File (using FileReader API)

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>File Reader Example</title>

</head>

<body>

<input type="file" id="fileInput">

<div id="fileContent"></div>

<script>

const fileInput = document.getElementById('fileInput');

const fileContentDisplay = document.getElementById('fileContent');

fileInput.addEventListener('change', function(event) {

const file = event.target.files[0];

const reader = new FileReader();

reader.onload = function(event) {

const fileContents = event.target.result;

fileContentDisplay.textContent = fileContents;

};

reader.readAsText(file);

});

</script>

</body>

</html>

```

#### Writing to a File (Not Possible in a Web Browser)

In a web browser, JavaScript running in a webpage cannot directly write to files on the user's computer for security reasons. If you need to save data generated by your web application, you typically use techniques like storing data in cookies, using local storage, or sending it to a server for storage.

### Reading and Writing Files in Node.js (Server-side JavaScript)

In Node.js, JavaScript has access to the file system through built-in modules like `fs`.

#### Reading a File (using `fs` module)

```javascript

const fs = require('fs');

fs.readFile('example.txt', 'utf8', function(err, data) {

if (err) {

console.error(err);

return;

}

console.log(data);

});

```

#### Writing to a File (using `fs` module)

```javascript

const fs = require('fs');

const content = 'This is the content that will be written to the file.';

fs.writeFile('example.txt', content, function(err) {

if (err) {

console.error(err);

return;

}

console.log('File written successfully.');

});

```

### Summary

- \*\*Web Browser (Client-side JavaScript)\*\*:

- Reading: Use File and FileReader APIs, typically in conjunction with user input.

- Writing: Not directly possible due to security restrictions.

- \*\*Node.js (Server-side JavaScript)\*\*:

- Reading: Use the `fs` module's `readFile` function.

- Writing: Use the `fs` module's `writeFile` function.

Choose the appropriate approach based on whether your JavaScript code is running in a web browser or in a Node.js environment.

• What are all the looping structures in JavaScript?

Answer:- JavaScript provides several looping structures to iterate over data or execute a block of code repeatedly. Here are the main looping structures in JavaScript:

1. \*\*for Loop\*\*: Executes a block of code a specified number of times.

```javascript

for (initialization; condition; update) {

// code to be executed

}

```

2. \*\*while Loop\*\*: Executes a block of code while a specified condition is true.

```javascript

while (condition) {

// code to be executed

}

```

3. \*\*do...while Loop\*\*: Similar to a while loop, but the code block is executed at least once before the condition is tested.

```javascript

do {

// code to be executed

} while (condition);

```

4. \*\*for...in Loop\*\*: Iterates over the enumerable properties of an object.

```javascript

for (variable in object) {

// code to be executed

}

```

5. \*\*for...of Loop\*\*: Iterates over the iterable objects like arrays, strings, maps, sets, etc.

```javascript

for (variable of iterable) {

// code to be executed

}

```

### Example Usage:

#### for Loop:

```javascript

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

console.log(i);

}

```

#### while Loop:

```javascript

let i = 0;

while (i < 5) {

console.log(i);

i++;

}

```

#### do...while Loop:

```javascript

let i = 0;

do {

console.log(i);

i++;

} while (i < 5);

```

#### for...in Loop:

```javascript

const person = {name: 'John', age: 30, city: 'New York'};

for (let key in person) {

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

}

```

#### for...of Loop:

```javascript

const fruits = ['apple', 'banana', 'cherry'];

for (let fruit of fruits) {

console.log(fruit);

}

```

### Summary:

- \*\*for Loop\*\*: Use when you know the number of iterations beforehand.

- \*\*while Loop\*\*: Use when the number of iterations is unknown and depends on a condition.

- \*\*do...while Loop\*\*: Use when you want to ensure that the loop body executes at least once.

- \*\*for...in Loop\*\*: Use to iterate over object properties.

- \*\*for...of Loop\*\*: Use to iterate over iterable objects like arrays, strings, etc.

Choose the looping structure that best fits your specific use case and coding style.

• How can you convert the string of any base to an integer in JavaScript?

Answer:- You can convert a string representing a number in any base to an integer in JavaScript using the `parseInt()` function. The `parseInt()` function parses a string and returns an integer based on the specified radix (base). Here's how you can use it:

```javascript

// Convert a binary string to an integer (base 2)

const binaryString = "1010";

const binaryInteger = parseInt(binaryString, 2);

console.log(binaryInteger); // Output: 10

// Convert an octal string to an integer (base 8)

const octalString = "12";

const octalInteger = parseInt(octalString, 8);

console.log(octalInteger); // Output: 10

// Convert a hexadecimal string to an integer (base 16)

const hexadecimalString = "A";

const hexadecimalInteger = parseInt(hexadecimalString, 16);

console.log(hexadecimalInteger); // Output: 10

```

In the `parseInt()` function:

- The first argument is the string to be parsed.

- The second argument is the radix (base) of the number represented by the string. It can be any integer between 2 and 36. If omitted or 0, JavaScript assumes the following:

- If the string begins with `"0x"` or `"0X"`, it is parsed as a hexadecimal number (base 16).

- If the string begins with `"0"`, it is parsed as an octal number (base 8).

- Otherwise, it is parsed as a decimal number (base 10).

### Note:

- `parseInt()` only parses the integer part of the string. Any non-numeric characters after the integer part are ignored.

- If the first character cannot be converted to a number (e.g., `"A"` in a decimal conversion), `parseInt()` returns `NaN` (Not-a-Number).

### Example:

```javascript

const invalidString = "Hello123";

const invalidInteger = parseInt(invalidString, 10);

console.log(invalidInteger); // Output: NaN

```

### Summary:

- Use `parseInt()` with the appropriate radix to convert a string representing a number in any base to an integer in JavaScript.

- Specify the base as the second argument to `parseInt()`.

• What is the function of the delete operator?

Answer:- In JavaScript, the `delete` operator is used to remove a property from an object. It can also be used to remove elements from an array. Here are the key points about the `delete` operator:

### Syntax:

```javascript

delete object.property; // Delete a property from an object

delete object[index]; // Delete an element from an array

```

### Functionality:

1. \*\*Removing Properties from Objects\*\*:

- When applied to an object property, `delete` removes the specified property from the object.

```javascript

const obj = {name: 'John', age: 30};

delete obj.age; // Removes the 'age' property from the object

```

2. \*\*Removing Elements from Arrays\*\*:

- When applied to an array element using its index, `delete` sets the element to `undefined`, but it does not change the length of the array or remove the element completely.

```javascript

const arr = ['apple', 'banana', 'cherry'];

delete arr[1]; // Sets the element at index 1 to undefined

console.log(arr); // Output: ['apple', undefined, 'cherry']

```

### Limitations:

- \*\*Not Suitable for Removing Array Elements\*\*: While `delete` can be used to remove elements from an array, it's not the recommended way because it doesn't actually remove the element; it only sets it to `undefined`. The `splice()` method is generally preferred for removing elements from arrays as it adjusts the array length properly.

### Example:

```javascript

const obj = {name: 'John', age: 30};

console.log(obj); // Output: {name: 'John', age: 30}

// Deleting the 'age' property from the object

delete obj.age;

console.log(obj); // Output: {name: 'John'}

```

### Summary:

- The `delete` operator is used to remove properties from objects and elements from arrays in JavaScript.

- When applied to an object property, `delete` removes the property from the object.

- When applied to an array element, `delete` sets the element to `undefined`, but it doesn't change the length of the array or remove the element completely.

- It's not recommended to use `delete` to remove array elements because it doesn't adjust the array length properly; use `splice()` instead.

• What are all the types of Pop up boxes available in JavaScript?

Answer:- In JavaScript, there are three main types of pop-up boxes or dialog boxes that you can use to interact with users: `alert`, `confirm`, and `prompt`. These dialog boxes are commonly used to display messages, obtain user confirmation, and prompt for input. Here's a brief overview of each:

1. \*\*Alert Box (`alert`)\*\*:

- The `alert` dialog box is used to display an alert message to the user.

- It typically contains a message and an OK button.

- It's commonly used to provide information or notifications to the user.

```javascript

alert("This is an alert message.");

```

2. \*\*Confirm Box (`confirm`)\*\*:

- The `confirm` dialog box is used to get a yes/no confirmation from the user.

- It contains a message, an OK button, and a Cancel button.

- It returns `true` if the user clicks OK and `false` if the user clicks Cancel.

```javascript

const result = confirm("Are you sure you want to delete this item?");

if (result) {

// Delete the item

} else {

// Cancel the operation

}

```

3. \*\*Prompt Box (`prompt`)\*\*:

- The `prompt` dialog box is used to prompt the user to enter some input.

- It contains a message, an input field for the user to enter text, an OK button, and a Cancel button.

- It returns the text entered by the user if the OK button is clicked, or `null` if the Cancel button is clicked.

```javascript

const name = prompt("Please enter your name:", "John Doe");

if (name !== null) {

// Process the user input

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

} else {

// User clicked Cancel

console.log("Operation canceled.");

}

```

### Example Usage:

```javascript

// Alert box

alert("Welcome to our website!");

// Confirm box

const result = confirm("Are you sure you want to submit this form?");

if (result) {

// Submit the form

} else {

// Cancel the submission

}

// Prompt box

const age = prompt("Please enter your age:");

if (age !== null) {

console.log("You entered: " + age);

} else {

console.log("User canceled the prompt.");

}

```

These pop-up boxes provide simple ways to interact with users and gather input or feedback in JavaScript applications.

• What is the use of Void (0)?

Answer:- In JavaScript, `void(0)` is an expression that evaluates to `undefined`. The `void` operator in JavaScript is used to evaluate an expression and then return `undefined`. It is typically used to create a hyperlink that does nothing when clicked, also known as a "void link" or "javascript:void(0)".

### Usage of `void(0)`:

1. \*\*Creating Void Links\*\*:

- The most common use of `void(0)` is to create hyperlinks that do not perform any action when clicked. This is often used in scenarios where you want to create a placeholder link or prevent the page from scrolling to the top when clicked.

```html

<a href="javascript:void(0)">Click me</a>

```

2. \*\*Preventing Default Actions\*\*:

- It can be used in event handlers to prevent the default action of an event, such as submitting a form or following a link.

```javascript

document.getElementById('myLink').onclick = function(event) {

event.preventDefault(); // Prevent the default action of clicking the link

// Perform other actions

};

```

3. \*\*Returning Undefined\*\*:

- It can be used in JavaScript code to explicitly return `undefined` from a function or expression.

```javascript

function myFunction() {

return void(0); // Explicitly return undefined

}

```

### Why Use `void(0)`?

Using `void(0)` instead of `undefined` directly serves as a clear indicator that the intention is to evaluate an expression and return `undefined`. It is commonly used for consistency and readability in code, especially in scenarios involving event handling and hyperlink behavior.

### Example:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Void Example</title>

</head>

<body>

<a href="javascript:void(0)">Click me</a>

</body>

</html>

```

### Summary:

- `void(0)` is an expression that evaluates to `undefined` in JavaScript.

- It is commonly used to create hyperlinks that do not perform any action when clicked, prevent default actions in event handlers, or explicitly return `undefined` from functions.

- Using `void(0)` provides clarity and consistency in code, especially in scenarios involving event handling and hyperlink behavior.

• How can a page be forced to load another page in JavaScript?

Answer:- In JavaScript, you can force a page to load another page by changing the value of the `window.location` object. There are different ways to achieve this depending on the requirement. Here are a few common methods:

### 1. Changing the URL directly:

```javascript

window.location.href = 'https://example.com'; // Absolute URL

```

This will load the specified URL in the current window.

### 2. Changing the URL without reloading the page:

```javascript

window.location.assign('https://example.com'); // Absolute URL

```

This method behaves the same as changing `window.location.href`, but it also adds the new URL to the browser's history, allowing the user to navigate back to the previous page.

### 3. Opening the URL in a new tab or window:

```javascript

window.open('https://example.com'); // Absolute URL

```

This will open the specified URL in a new tab or window, depending on the browser's settings.

### 4. Using a relative URL:

```javascript

window.location.href = 'page2.html'; // Relative URL

```

This will load a page named "page2.html" in the same directory as the current page.

### Example Usage:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Redirect Example</title>

</head>

<body>

<button onclick="redirectToPage()">Go to Example</button>

<script>

function redirectToPage() {

// Change the URL to the specified page

window.location.href = 'https://example.com';

}

</script>

</body>

</html>

```

In this example, clicking the button will trigger the `redirectToPage` function, which changes the current page's URL to "https://example.com", effectively loading that page in the current window.

### Note:

- When using these methods to change the page URL, the browser will typically perform a navigation action, which may trigger browser events like `unload` on the current page and `load` on the new page.

- Changing the page URL using JavaScript may be subject to browser security restrictions, especially when navigating to external domains.

• What are the disadvantages of using innerHTML in JavaScript?

Answer:- While `innerHTML` in JavaScript is a powerful and convenient way to manipulate the HTML content of elements, it does have some disadvantages compared to alternative methods like using the DOM API directly. Here are some common drawbacks of using `innerHTML`:

1. \*\*Security Risks\*\*:

- Using `innerHTML` to insert HTML content from untrusted sources can expose your application to cross-site scripting (XSS) attacks. If the inserted content contains malicious scripts, they can execute within the context of your webpage.

2. \*\*Performance Overhead\*\*:

- Manipulating `innerHTML` can be less efficient compared to directly manipulating the DOM elements using methods like `createElement`, `appendChild`, `removeChild`, etc. When you use `innerHTML`, the browser has to parse and re-render the entire HTML content of the affected element, which can be slower, especially for large documents.

3. \*\*Loss of Event Handlers and Data\*\*:

- When you set `innerHTML` to replace the content of an element, any event handlers attached to the existing content will be lost. Similarly, any JavaScript data associated with the elements, such as properties or references, will also be lost.

4. \*\*Potential Markup Validation Issues\*\*:

- When using `innerHTML`, it's easy to inadvertently introduce invalid markup, such as mismatched tags or improperly nested elements, especially when inserting complex HTML strings. This can lead to unexpected rendering issues and make your HTML harder to maintain.

5. \*\*Limited Support for SVG and Custom Elements\*\*:

- `innerHTML` may not work as expected when dealing with SVG elements or custom elements (created using Web Components). Some browsers may not correctly handle SVG markup inserted via `innerHTML`, and custom elements may lose their functionality or data-binding when their HTML content is replaced using `innerHTML`.

### Example:

```javascript

// Example of using innerHTML to insert HTML content

document.getElementById('myElement').innerHTML = '<p>New content</p>';

```

### Recommendations:

- \*\*Sanitize User Input\*\*: If you must use `innerHTML` with dynamic content, make sure to properly sanitize the input to prevent XSS attacks.

- \*\*Consider Alternative Methods\*\*: Depending on your use case, consider using the DOM API methods like `createElement`, `appendChild`, etc., for better performance and security.

- \*\*Use Frameworks or Libraries\*\*: Frameworks like React, Vue.js, or libraries like jQuery provide safer and more efficient ways to manipulate the DOM compared to raw `innerHTML`.

While `innerHTML` can be convenient for certain tasks, it's important to be aware of its limitations and use it judiciously, especially in security-sensitive applications.

• Create password field with show hide functionalities

Answer:- You can create a password field with show/hide functionality using HTML, CSS, and JavaScript. Here's a simple example:

HTML:

```html

<input type="password" id="passwordInput">

<button id="toggleButton">Show</button>

```

CSS:

```css

#passwordInput {

padding: 10px;

margin-right: 5px;

}

#toggleButton {

padding: 10px;

cursor: pointer;

}

```

JavaScript:

```javascript

const passwordInput = document.getElementById('passwordInput');

const toggleButton = document.getElementById('toggleButton');

toggleButton.addEventListener('click', function() {

if (passwordInput.type === 'password') {

passwordInput.type = 'text';

toggleButton.textContent = 'Hide';

} else {

passwordInput.type = 'password';

toggleButton.textContent = 'Show';

}

});

```

This code creates a password input field and a button. When the button is clicked, it toggles between showing and hiding the password.

Here's a breakdown of how it works:

- We select the password input field and the toggle button using `document.getElementById`.

- We add an event listener to the toggle button. When clicked, it checks the type of the password input field. If it's currently set to "password", it changes it to "text" to reveal the password and updates the button text to "Hide". If it's already set to "text", it changes it back to "password" to hide the password and updates the button text to "Show".

You can further style the button and input field according to your preferences using CSS.

• Create basic math operation in JS

**Answer:- Certainly! Below is a simple JavaScript code that demonstrates basic math operations: addition, subtraction, multiplication, and division.**

**```html**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Basic Math Operations</title>**

**<script>**

**// Function to perform addition**

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

**return a + b;**

**}**

**// Function to perform subtraction**

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

**return a - b;**

**}**

**// Function to perform multiplication**

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

**return a \* b;**

**}**

**// Function to perform division**

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

**if (b === 0) {**

**return 'Cannot divide by zero';**

**}**

**return a / b;**

**}**

**// Function to display the results**

**function displayResults() {**

**let a = parseFloat(document.getElementById('num1').value);**

**let b = parseFloat(document.getElementById('num2').value);**

**document.getElementById('addition').innerHTML = 'Addition: ' + add(a, b);**

**document.getElementById('subtraction').innerHTML = 'Subtraction: ' + subtract(a, b);**

**document.getElementById('multiplication').innerHTML = 'Multiplication: ' + multiply(a, b);**

**document.getElementById('division').innerHTML = 'Division: ' + divide(a, b);**

**}**

**</script>**

**</head>**

**<body>**

**<h1>Basic Math Operations</h1>**

**<p>Enter two numbers:</p>**

**<input type="text" id="num1" placeholder="First number">**

**<input type="text" id="num2" placeholder="Second number">**

**<button onclick="displayResults()">Calculate</button>**

**<h2>Results:</h2>**

**<p id="addition"></p>**

**<p id="subtraction"></p>**

**<p id="multiplication"></p>**

**<p id="division"></p>**

**</body>**

**</html>**

**```**

**### Explanation:**

**1. \*\*HTML Structure\*\*:**

**- Input fields to take two numbers.**

**- A button to trigger the calculation.**

**- Paragraphs to display the results of the operations.**

**2. \*\*JavaScript Functions\*\*:**

**- `add(a, b)`: Returns the sum of `a` and `b`.**

**- `subtract(a, b)`: Returns the difference between `a` and `b`.**

**- `multiply(a, b)`: Returns the product of `a` and `b`.**

**- `divide(a, b)`: Returns the quotient of `a` divided by `b`, with a check to prevent division by zero.**

**3. \*\*Display Function\*\*:**

**- `displayResults()`: This function reads the values from the input fields, performs the operations using the defined functions, and updates the HTML to display the results.**

**You can save this code in an HTML file and open it in a web browser to see it in action.**

• Create a slider using JavaScript

Answer:- Sure! Here's a simple example of how to create a slider using JavaScript, HTML, and CSS.

### HTML:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Slider Example</title>

<style>

/\* CSS for the slider \*/

.slider-container {

width: 50%;

margin: 50px auto;

text-align: center;

}

.slider {

width: 100%;

}

.value {

margin-top: 20px;

font-size: 24px;

}

</style>

</head>

<body>

<div class="slider-container">

<input type="range" min="0" max="100" value="50" class="slider" id="mySlider">

<div class="value" id="sliderValue">50</div>

</div>

<script>

// JavaScript to handle the slider

const slider = document.getElementById('mySlider');

const sliderValue = document.getElementById('sliderValue');

// Update the displayed value when the slider is moved

slider.oninput = function() {

sliderValue.textContent = this.value;

};

</script>

</body>

</html>

```

### Explanation:

1. \*\*HTML Structure\*\*:

- A `div` with class `slider-container` to center the slider and its value display.

- An `input` element of type `range` to create the slider.

- A `div` element to display the current value of the slider.

2. \*\*CSS Styling\*\*:

- `.slider-container`: Sets the width of the container and centers it using margin.

- `.slider`: Makes the slider take the full width of the container.

- `.value`: Styles the value display below the slider.

3. \*\*JavaScript Functionality\*\*:

- Gets references to the slider (`mySlider`) and the value display (`sliderValue`) elements.

- Updates the displayed value (`sliderValue`) whenever the slider's value changes (`oninput` event).

You can save this code in an HTML file and open it in a web browser to see the slider in action. The displayed value will update dynamically as you move the slider.