1. Question: Reverse a string without using the built-in reverse() method.

index.js index.html

index.js > reversestring

```javascript
function reversestring(str){
    let Reversed=''
    for(i = str.length - 1; i >= 0 ; i--){
        Reversed += str[i];
    }
    return Reversed;
}

let sir="Ali" //output 'ila'
let reversedstring = reversestring(sir);
console.log(reversedstring);
```

2. Question: Count the number of vowels in a given string.

index.js index.html

index.js > ...

```javascript

function sum(str){
   var exact = [''];
    var exact = str.match(/[a,e,i,o]/g);
 exact.reduce((acc,currentvalue) =>
acc+currentvalue,0)
return exact;
}

let car="audi";//    output['a','i'] ,2
let extract=sum(car);
console.log(extract);
```

3. Question: Convert the first letter of each word in a sentence to uppercase.

index.js index.html

index.js > ...

```javascript
function capitalizeWords(sentence) {
    let words = sentence.split(" ");

    let capitalizedWords = words.map(word => {
        return word.charAt(0).toUpperCase() + word.slice(1);
    });

    let capitalizedSentence = capitalizedWords.join(" ");

    return capitalizedSentence;
}

let sentence = "convert the first letter of each word in a sentence to uppercase";
let capitalizedSentence = capitalizeWords(sentence);
console.log(capitalizedSentence);
```

No definition found

Elements Console Sources Network Performance

top Filter Default levels No Issues

```
Convert The First Letter Of Each Word In A Sentence To Uppercase
```

index.js:15

>

index.js index.html

index.js > isPalindrome

```javascript
function isPalindrome(str) {
    let cleanStr = str.replace(/[^A-Za-z0-9]/g, '').toLowerCase();

    for (let i = 0; i < Math.floor(cleanStr.length / 2); i++) {
        if (cleanStr[i] !== cleanStr[cleanStr.length - 1 - i]) {
            return false;
        }
    }
    return true;
}
// Example usage:
let str1 = "A man, a plan, a canal, Panama";
console.log(isPalindrome(str1)); // Output: true

let str2 = "race a car";
console.log(isPalindrome(str2)); // Output: false
```

5. Question: Find the sum of all positive numbers in an array.

index.js index.html

index.js > ...

```javascript
function sumOfPositiveNumbers(arr) {
    let sum = 0;
    for (let num of arr) {
        if (num > 0) {
            sum += num;
        }
    }
    return sum;
}

// Example usage:
let array1 = [3, -2, 5, -1, 0, 7]; // Positive numbers: 3, 5, 7
console.log(sumOfPositiveNumbers(array1)); // Output: 15

let array2 = [-1, -5, -9]; // No positive numbers
console.log(sumOfPositiveNumbers(array2)); // Output: 0

let array3 = []; // Empty array
console.log(sumOfPositiveNumbers(array3)); // Output: 0
```

6. Question: Find the index of the first occurrence of a specific element in an array.

index.js index.html

index.js > ...

```javascript
function indexOfFirstOccurrence(arr, element) {
    return arr.indexOf(element);
}

// Example usage:
let array = [2, 5, 9, 2, 7, 5, 3];
let elementToFind1 = 5;
console.log(indexOfFirstOccurrence(array, elementToFind1)); // Output: 1 (index of first occurrence of 5)

let elementToFind2 = 10;
console.log(indexOfFirstOccurrence(array, elementToFind2)); // Output: -1 (element 10 is not in the array)
```

let elementToFind1: number

7. Question: Remove all duplicates from an array without built-in methods.

index.js index.html

index.js > removeDuplicates

```javascript
function removeDuplicates(arr) {
    let uniqueArray = [];
    for (let i = 0; i < arr.length; i++) {

        if (uniqueArray.indexOf(arr[i]) === -1) {

            uniqueArray.push(arr[i]);
        }
    }

    return uniqueArray;
}

// Example usage:
let array1 = [1, 2, 3, 2, 4, 5, 1, 6];
console.log(removeDuplicates(array1)); // Output: [1, 2, 3, 4, 5, 6]

let array2 = ["apple", "orange", "banana", "orange", "pear"];
console.log(removeDuplicates(array2)); // Output: ["apple", "orange", "banana", "pear"]
```

8. Question: Sort the array in ascending and descending without built-in methods.

index.js index.html

index.js > bubbleSortDescending

```javascript
function bubbleSortDescending(arr) {
    let n = arr.length;


    for (let i = 0; i < n - 1; i++) {
        for (let j = 0; j < n - 1 - i; j++) {

            if (arr[j] < arr[j + 1]) {
                let temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    return arr;
}

// Example usage:
let array = [64, 34, 25, 12, 22, 11, 90];
console.log("Descending Order:", bubbleSortDescending(array.slice())); // Output: [90, 64, 34, 25, 22, 12, 11]
```

index.js index.html

index.js > bubbleSortAscending

```javascript
function bubbleSortAscending(arr) {
    let n = arr.length;

    for (let i = 0; i < n - 1; i++) {

        for (let j = 0; j < n - 1 - i; j++) {

            if (arr[j] > arr[j + 1]) {
                let temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    return arr;
}

let array = [64, 34, 25, 12, 22, 11, 90];
console.log("Ascending Order:", bubbleSortAscending(array.slice())); // Output: [11, 12, 22, 25, 34, 64, 90]
```

9. Question: Print all even numbers between 1 and 20 using a while loop.

index.js index.html

index.js

```javascript
for(i=0;i<=20;i++){
    if(i%2==0){
        console.log(i);
    };
};
//output: 0,2,4,6,8,10,12,14,16,18,20
```

10. Question: Calculate the factorial of a number using a do-while loop.

index.js index.html

index.js > factorialWithDoWhileLoop

```javascript
function factorialWithDoWhileLoop(n) {
    if (n < 0 || !Number.isInteger(n)) {
        return "Factorial is not defined for negative numbers or non-integers.";
    }

    let factorial = 1;
    let i = 1;

    do {
        factorial *= i;
        i++;
    } while (i <= n);

    return factorial;
}

let number = 5;
console.log(`Factorial of ${number} is:`, factorialWithDoWhileLoop(number)); // Output: Factorial of 5 is: 120
```

11. Question: Iterate through the properties of an object using a for-in loop.

index.js index.html

index.js > ...

```javascript
let person = {
    firstName: "John",
    lastName: "Doe",
    age: 30,
    email: "john.doe@example.com"
};

for (let key in person) {

    if (person.hasOwnProperty(key)) {
        console.log(`${key}: ${person[key]}`);
    }
}

//output:firstName: John
//lastName: Doe
//age: 30
//email: john.doe@example.com
```

12. Question: Loop through an array using a for-of loop and double each element.

index.js index.html

index.js > ...

```javascript
function doubleElements(arr) {

    let doubledArray = [];


    for (let element of arr) {
        doubledArray.push(element * 2);
    }

    return doubledArray;
}

let array = [1, 2, 3, 4, 5];
let doubledArray = doubleElements(array);
console.log("Original array:", array); // Output: Original array: [1, 2, 3, 4, 5]
console.log("Doubled array:", doubledArray); // Output: Doubled array: [2, 4, 6, 8, 10]
```

13. Question: Check if a number is even or odd and return a corresponding message.

index.js index.html

index.js > ...

```javascript
function doubleElements(arr) {

    let doubledArray = [];


    for (let element of arr) {
        doubledArray.push(element * 2);
    }

    return doubledArray;
}

let array = [1, 2, 3, 4, 5];
let doubledArray = doubleElements(array);
console.log("Original array:", array); // Output: Original array: [1, 2, 3, 4, 5]
console.log("Doubled array:", doubledArray); // Output: Doubled array: [2, 4, 6, 8, 10]
```


14. Question: Find the maximum of three numbers using nested ternary operators.

index.js index.html

index.js > ...

```javascript
function maxOfThreeNumbers(a, b, c) {
    return a >= b ? (a >= c ? a : c) : (b >= c ? b : c);
}

let num1 = 10, num2 = 5, num3 = 8;
console.log(`Maximum of ${num1}, ${num2}, ${num3} is:`, maxOfThreeNumbers(num1, num2, num3));
// Output: Maximum of 10, 5, 8 is: 10

let num4 = 12, num5 = 15, num6 = 9;
console.log(`Maximum of ${num4}, ${num5}, ${num6} is:`, maxOfThreeNumbers(num4, num5, num6));
// Output: Maximum of 12, 15, 9 is: 15
```

15. Question: Determine if a year is a leap year or not.

index.js index.html

index.js > isLeapYear

```javascript
function isLeapYear(year) {
    // Check if the year is divisible by 4
    if (year % 4 === 0) {
        // Check if the year is divisible by 100 and not divisible by 400
        if (year % 100 === 0) {
            if (year % 400 === 0) {
                return true; // Divisible by 400 is a leap year
            } else {
                return false; // Divisible by 100 but not by 400 is not a leap year
            }
        } else {
            return true; // Divisible by 4 but not by 100 is a leap year
        }
    } else {
        return false; // Not divisible by 4 is not a leap year
    }
}

// Example usage:
let year1 = 2020;
console.log(`${year1} is a leap year:`, isLeapYear(year1)); // Output: 2020 is a leap year: true

let year2 = 2021;
console.log(`${year2} is a leap year:`, isLeapYear(year2)); // Output: 2021 is a leap year: false
```

Section 2.

1. Rewrite the following code using a ternary operator:
let result;
if (score >= 80) {
   result = "Pass";
} else {
   result = "Fail";
}

```
let result = (score >= 80) ? "Pass" : "Fail";
```

2. How does the optional chaining operator (?.) work, and how can it be used to access nested properties of an object?

How the optional chaining operator works:

1. **Accessing Properties**: Normally, when you access a property of an object that might be undefined or null, you'd get a TypeError if you try to access a property or method of it. For example:

```
let user = {
    name: "John",
    address: {
        city: "New York"
    }
};

console.log(user.address.city); // "New York"
console.log(user.address.street); // TypeError: Cannot read property 'street' of undef
```

2.Using Optional Chaining: The optional chaining operator `?.` allows you to safely access deeply nested properties. It short-circuits the evaluation if any part of the chain is null or undefined, returning `undefined` instead of throwing an error:

```
console.log(user.address?.city); // "New York"
console.log(user.address?.street); // undefined (no error)
console.log(user.phoneNumber?.personal?.mobile); // undefined (no error)
```

3.Using Optional Chaining with Methods:

Optional chaining can also be used to call methods on potentially undefined or null objects:

```
let user = {
    name: "John",
    getAddress() {
        return this.address?.city ?? "No address available";
    }
};

console.log(user.getAddress()); // "No address available" (since address is undefined)
```

3. Compare the for...in loop and the for...of loop in terms of their use cases and the types of values they iterate over.

For in loop is iterable through the properties of an object for of loop is iterate through the value of an Object.

4. Define a function calculateAverage that takes an array of numbers as an argument and returns the average value.

```
function calculateAverage(numbers) {
    // Check if the array is empty to avoid division by zero
    if (numbers.length === 0) {
        return 0; // Return 0 or handle this case according to your requirements
    }

    // Calculate the sum of all numbers in the array
    let sum = numbers.reduce((acc, num) => acc + num, 0);

    // Calculate the average by dividing the sum by the number of elements
    let average = sum / numbers.length;

    return average;
}
```

5. Explain the concept of "closures" in JavaScript and provide an example of their practical use.

In JavaScript, closures are an important concept that allows functions to retain access to variables from the outer scope (lexical environment) even after the outer function has finished executing. This means that a function defined inside another function (nested function) has access to its parent function's variables, even after the parent function has returned.

Key Points about Closures:

1. **Lexical Scope**: JavaScript uses lexical scoping, which means that functions are executed using the scope chain that was in effect when they were defined, not when they are executed.
2. **Access to Outer Variables**: A closure gives you access to an outer function's scope from an inner function even after the outer function has finished execution.
3. **Function and Scope Relationship**: Closures occur naturally in JavaScript because functions are first-class citizens, meaning they can be passed around like any other value.

Practical Example of Closures:

Here's a simple example to illustrate how closures work and their practical use:

index.js index.html

index.js > outerFunction

```js
function outerFunction() {
    let outerVariable = "I am from the outer function";

    function innerFunction() {
        console.log(outerVariable);
    }

    return innerFunction;
}

let closureExample = outerFunction();


closureExample();
```

6. Create an object named student with properties name, age, and grades. Add a method calculateAverage that calculates the average of the grades.

```javascript
let student = {
    name: "John",
    age: 20,
    grades: [85, 90, 75, 95, 80],

    calculateAverage: function() {
        let sum = this.grades.reduce((acc, grade) => acc + grade, 0);
        let average = sum / this.grades.length;
        return average;
    }
};


console.log(student.calculateAverage()); // Output: 85 (assuming grades are [85, 90, 75, 95, 80])
```

7. How can you clone an object in JavaScript and also give one example each deep copy, shallow copy, and reference copy

In JavaScript, there are different ways to clone or copy objects, each with its implications for how deeply nested properties and references are handled. Here's how you can perform shallow copy, deep copy, and reference copy of objects:

1. Shallow Copy:

A shallow copy creates a new object and copies all top-level properties of the original object. However, if the original object contains nested objects or arrays, they are copied as references.

Example of shallow copy:

```javascript
// Original object
let originalObject = {
    name: 'John',
    age: 30,
    hobbies: ['reading', 'sports']
};

// Shallow copy using Object.assign()
let shallowCopy = Object.assign({}, originalObject);

// Modify the shallow copy
shallowCopy.name = 'Jane';
shallowCopy.hobbies.push('cooking'); // Modifies the original object's hobbies array too

console.log(originalObject); // Output: { name: 'John', age: 30, hobbies: ['reading', 'sports', 'cooking'] }
console.log(shallowCopy);    // Output: { name: 'Jane', age: 30, hobbies: ['reading', 'sports', 'cooking'] }
```

2. Deep Copy:

A deep copy creates a new object and recursively copies all nested properties and their nested properties, etc. This ensures that any nested objects or arrays are also copied rather than referenced.

```
// Original object with nested properties
let originalObject = {
    name: 'John',
    age: 30,
    address: {
        city: 'New York',
        country: 'USA'
    }
};

// Deep copy using JSON.parse and JSON.stringify
let deepCopy = JSON.parse(JSON.stringify(originalObject));

// Modify the deep copy
deepCopy.name = 'Jane';
deepCopy.address.city = 'Los Angeles';

console.log(originalObject); // Output: { name: 'John', age: 30, address: { city: 'New York', country: 'USA' } }
console.log(deepCopy);       // Output: { name: 'Jane', age: 30, address: { city: 'Los Angeles', country: 'USA'
```

3. Reference Copy:

A reference copy does not create a new object but instead creates a new reference to the same object. Changes made through the new reference will affect the original object and vice versa.

Example of Reference copy:

index.js index.html

index.js > ...

```javascript
// Original object
let originalObject = {
    name: 'John',
    age: 30
};

// Reference copy
let referenceCopy = originalObject;

// Modify the reference copy
referenceCopy.name = 'Jane';

console.log(originalObject); // Output: { name: 'Jane', age: 30 }
console.log(referenceCopy);  // Output: { name: 'Jane', age: 30 }
```

8. Write a loop that iterates over an array of numbers and logs whether each number is even or odd, using a ternary operator.

index.js index.html

index.js > ...

```javascript
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

for (let number of numbers) {
    let message = (number % 2 === 0) ? `${number} is even` : `${number} is odd`;
    console.log(message);
}

// output
// 1 is odd
// 2 is even
// 3 is odd
// 4 is even
// 5 is odd
// 6 is even
// 7 is odd
// 8 is even
// 9 is odd
// 10 is even
```

9. Describe the differences between the for loop, while loop, and do...while loop in JavaScript. When might you use each?

- **for Loop**:
  - Use when the number of iterations is known.
  - Provides clear initialization, condition, and iteration steps within the loop header.
  - Typically used for iterating over arrays or performing a fixed number of iterations.
- **while Loop**:
  - Use when the number of iterations is not known beforehand and depends on a condition.
  - The condition is checked before each iteration.
  - Useful for situations where you need to repeatedly execute code as long as a condition is true.
- **do...while Loop**:
  - Use when you want to ensure that the loop block executes at least once, regardless of the initial condition.
  - The condition is checked after each iteration.

- Useful when you need to execute a block of code and then check a condition to decide if the loop should continue.

10. Provide an example of using optional chaining within a loop to access a potentially missing property of an object.

index.js index.html

index.js > user > address

```javascript
// Example object with nested structure
let user = {
    name: 'John',
    address: {
        city: 'New York'
    },
    orders: [
        { id: 1, product: 'Laptop', price: 1200 },
        { id: 2, product: 'Headphones', price: 100 }
    ]
};

// Loop over orders array and access city property using optional chaining
for (let order of user.orders) {
    let cityName = order.address?.city ?? 'Unknown'; // Optional chaining to access city
    console.log(`Order ID ${order.id}: Customer ${user.name} lives in ${cityName}`);
}

// output
// Order ID 1: Customer John lives in Unknown
// Order ID 2: Customer John lives in Unknown
```

11. Write a for...in loop that iterates over the properties of an object and logs each property name and value.

index.js index.html

index.js > ...

```javascript

let person = {
    firstName: 'John',
    lastName: 'Doe',
    age: 30,
    city: 'New York'
};


for (let key in person) {
    if (person.hasOwnProperty(key)) {
        console.log(`${key}: ${person[key]}`);
    }
}

```

12. Explain the use of the break and continue statements within loops. Provide scenarios where each might be used.

1. break Statement:

The break statement is used to terminate the execution of a loop prematurely, regardless of whether the loop condition is still true or there are remaining iterations. When break is encountered inside a loop, the loop stops executing and control passes to the statement immediately following the loop.

2. continue Statement:

The continue statement is used to skip the current iteration of a loop and continue with the next iteration. Unlike break, continue does not terminate the loop but rather skips the remaining code inside the loop for the current iteration and proceeds with the next iteration.

13. Write a function calculateTax that calculates and returns the tax amount based on a given income. Use a ternary operator to determine the tax rate.

index.js index.html

index.js > calculateTax

```javascript
function calculateTax(income) {
    // Ternary operator to determine tax rate based on income
    let taxRate = income <= 50000 ? 0.1 : income <= 100000 ? 0.2 : 0.3;

    // Calculate tax amount
    let taxAmount = income * taxRate;

    // Return tax amount
    return taxAmount;
}

// Example usage:
let income = 60000;
let tax = calculateTax(income);
console.log(`Tax amount for income $${income} is $${tax}`);
```

14. Create an object car with properties make, model, and a method startEngine that logs a message. Instantiate the object and call the method.

index.js index.html

index.js > ...

```javascript
// Define the car object
let car = {
    make: 'Toyota',
    model: 'Camry',
    startEngine: function() {
        console.log(`Starting the engine of ${this.make} ${this.model}`);
    }
};

// Instantiate the object and call the method
car.startEngine();
```

15. Explain the differences between regular functions and arrow functions in terms of scope, this binding, and their use as methods.

- **Regular functions** have their own `arguments` and `this` binding that is determined dynamically.
- **Arrow functions** do not have their own `arguments` and lexically bind `this` from the surrounding code.
- Use regular functions when `this` binding needs to be dynamic or when defining object methods. Use arrow functions for concise and predictable `this` binding in lexical contexts and callbacks.

Section 03

1. **Map Transformation:**
- Q: Given an array of integers, use the `map` method to square each element and return a new array with the squared values.

index.js index.html

index.js > ...

```
// Example array of integers
let numbers = [1, 2, 3, 4, 5];

// Using map to square each element
let squaredNumbers = numbers.map(num => num * num);

// Output the transformed array
console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

2. **Filter and Map Combination:**
- Q: Take an array of strings, filter out the ones with a length less than 5, and then capitalize the remaining strings using the `map` method.

index.js index.html

index.js > ...

```javascript
// Example array of strings
let words = ["apple", "banana", "grape", "orange", "kiwi", "melon"];

// Filter and map combination
let filteredAndCapitalized = words
    .filter(word => word.length >= 5) // Filter strings with length >= 5
    .map(word => word.toUpperCase()); // Capitalize the remaining strings

// Output the transformed array
console.log(filteredAndCapitalized); // Output: ["BANANA", "ORANGE", "MELON"]
```

3. **Sorting Objects:**
- Q: Given an array of objects with a 'price' property, use the `sort` method to arrange them in descending order based on their prices.

index.js index.html

index.js > ...

```javascript
// Example array of objects with 'price' property
let products = [
    { name: 'Laptop', price: 1200 },
    { name: 'Headphones', price: 100 },
    { name: 'Smartphone', price: 800 },
    { name: 'Tablet', price: 600 }
];

// Sort products array in descending order based on 'price'
products.sort((a, b) => b.price - a.price);

// Output the sorted array
console.log(products);
```

4. **Reduce for Aggregation:**
- Q: Use the `reduce` method to find the total sum of all even numbers in an array of integers.

index.js index.html

index.js > ...

```javascript
/// Example array of integers
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Use reduce to find sum of even numbers
let sumOfEvens = numbers.reduce((accumulator, current) => {
    // Check if
    if (current
        return accumulator + current; // Add to accumulator if even
    } else {
        return accumulator; // Otherwise, return accumulator unchanged
    }
}, 0); // Initial value of accumulator is 0

// Output the sum of even numbers
console.log(sumOfEvens); // Output: 30 (2 + 4 + 6 + 8 + 10 = 30)
```

(parameter) accumulator: number

5. **Find and Modify:**
- Q: Given an array of objects with 'id' properties, use the `find` method to locate an object with a specific 'id' and update its 'status' property to 'completed'.

```javascript
// Example array of objects with 'id' properties
let tasks = [
    { id: 1, description: 'Task 1', status: 'pending' },
    { id: 2, description: 'Task 2', status: 'in progress' },
    { id: 3, description: 'Task 3', status: 'pending' },
    { id: 4, description: 'Task 4', status: 'in progress' }
];

// Function to update status of task with specific id to 'completed'
function updateStatusById(tasksArray, idToUpdate) {
    // Find the task object with matching id
    let taskToUpdate = tasksArray.find(task => task.id === idToUpdate);

    // Check if taskToUpdate is defined (i.e., found)
    if (taskToUpdate) {
        // Update status to 'completed'
        taskToUpdate.status = 'completed';
        console.log(`Task with id ${idToUpdate} updated to 'completed'.`);
    } else {
        console.log(`Task with id ${idToUpdate} not found.`);
    }

    // Return the updated tasks array (optional)
    return tasksArray;
}

// Example usage: Update task with id 3 to 'completed'
tasks = updateStatusById(tasks, 3);

// Output the updated tasks array
console.log(tasks);
```

6. **Chaining Methods:**
- Q: Create a chain of array methods to find the average of all positive numbers in an array of mixed integers and return the result rounded to two decimal places.

index.js index.html

index.js > averagePositive

```javascript
// Example array of mixed integers
let numbers = [5, -3, 10, -2, 8, -1, 6];

// Chain of array methods to find average of positive numbers
let averagePositive = numbers
    .filter(num => num > 0)         // Filter positive numbers
    .reduce((sum, num, index, array) => {
        sum += num;                 // Accumulate sum of positive numbers
        if (index === array.length - 1) {
            return sum / array.length;  // Return average at the end of the array
        } else {
            return sum;               // Pass sum to the next iteration
        }
    }, 0)                           // Initial value of sum for reduce
    .toFixed(2);                    // Round to two decimal places

// Output the average of positive numbers
console.log(averagePositive);       // Output: "6.33"
```

7. **Conditional Filtering:**
- Q: Implement a function that takes an array of objects with 'age' properties and returns an array of those who are adults (age 18 and above) using the `filter` method.

index.js index.html

index.js > ...

```javascript
// Example array of objects with 'age' properties
let persons = [
    { name: 'Alice', age: 25 },
    { name: 'Bob', age: 17 },
    { name: 'Charlie', age: 20 },
    { name: 'David', age: 16 },
    { name: 'Eve', age: 18 }
];

// Function to filter adults (age >= 18)
function filterAdults(personsArray) {
    return personsArray.filter(person => person.age >= 18);
}

// Example usage: Filter adults from the persons array
let adults = filterAdults(persons);

// Output the array of adults
console.log(adults);
```

8. **Advanced Sorting:**
- Q: Sort an array of strings based on their lengths in ascending order. If two strings have the same length, maintain their relative order in the sorted array.

index.js index.html

index.js > ...

```javascript
// Example array of strings
let strings = ["apple", "banana", "pear", "kiwi", "orange", "grape"];

// Custom sort function based on string lengths
strings.sort((a, b) => {
    // Compare lengths of strings
    if (a.length < b.length) {
        return -1; // 'a' should come before 'b'
    } else if (a.length > b.length) {
        return 1; // 'b' should come before 'a'
    } else {
        return 0; // Maintain relative order if lengths are equal
    }
});

// Output the sorted array of strings
console.log(strings);
```

9. **Nested Array Operations:**
- Q: Given an array of arrays containing numbers, use a combination of array methods to flatten the structure and then calculate the sum of all the numbers.

index.js index.html

index.js > ...

```javascript
// Example array of arrays containing numbers
let arrays = [[1, 2, 3], [4, 5], [6, 7, 8]];

// Use flatMap and reduce to flatten and sum the numbers
let sum = arrays
    .flatMap(innerArray => innerArray) // Flatten the array
    .reduce((total, num) => total + num, 0); // Calculate the sum

// Output the total sum
console.log(sum); // Output: 36 (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36)
```

10. **Error Handling with Find:**
- Q: Modify the `find` method to handle the scenario where the desired element is not found, returning a custom default object instead.

index.js index.html

index.js > ...

```javascript
// Example array of objects
let users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 3, name: 'Charlie' }
];

// Function to find user by ID or return a default object
function findUserById(usersArray, idToFind) {
    // Use find method to locate user by ID
    let foundUser = usersArray.find(user => user.id === idToFind);

    // Check if user is found
    if (foundUser) {
        return foundUser; // Return found user object
    } else {
        // Return default object if user is not found
        return { id: 0, name: 'Unknown User' };
    }
}

// Example usage: Find user by ID or return default object
let user = findUserById(users, 2);
console.log(user); // Output: { id: 2, name: 'Bob' }

let nonExistentUser = findUserById(users, 10);
console.log(nonExistentUser); // Output: { id: 0, name: 'Unknown User' }
```

11. **Map Method:**
- Q: How does the `map` method work in JavaScript, and can you provide an example of when you might use it to manipulate an array of objects?

The `map` method in JavaScript is used to iterate over an array and transform each element in the array according to a callback function. It creates a new array populated with the results of calling the callback function on each element of the original array. Here's how it works and an example of its usage with array of objects

Example: Manipulating an Array of Objects

Suppose you have an array of objects representing products, and you want to create a new array where each product's price is increased by 10%:

index.js index.html

index.js > ...

```javascript
// Example array of objects representing products
let products = [
    { id: 1, name: 'Laptop', price: 1200 },
    { id: 2, name: 'Smartphone', price: 800 },
    { id: 3, name: 'Tablet', price: 600 }
];

// Use map method to increase price of each product by 10%
let updatedPrices = products.map(product => {
    return {
        id: product.id,
        name: product.name,
        price: product.price * 1.1  // Increase price by 10%
    };
});

// Output the updated array of objects
console.log(updatedPrices);
```