QUESTIONS

## LEVEL EASY

### Demo 01

## 1.Implement the classic FizzBuzz problem using loops.

***Requirements:***

1. Write a function called `fizzBuzz` that takes a number n as a parameter.
2. The function should loop through numbers from 1 to n.
3. For each number:
    a. Print "Fizz" if it is divisible by 3.
    b. Print "Buzz" if it is divisible by 5.
    c. Print "FizzBuzz" if it is divisible by both 3 and 5.
    d. Print the number itself if it is not divisible by either.

### Steps to Solve the Exercise

1. **Define the Function:**
    a. Create a function called `fizzBuzz` that accepts a single parameter n.
2. **Implement the Loop:**
    a. Use a `for` loop to iterate from 1 to n.
3. **Conditional Logic:**
    a. Use `if`, `else if`, and `else` statements to determine what to print for each number.
4. **Test the Function:**
    a. Call the function with different values of n to ensure it works as expected.

### Solution

Here's how you can implement this in JavaScript:

```
javascript
// Step 1: Define the Function
function fizzBuzz(n) {
    // Step 2: Implement the Loop
```

```
    for (let i = 1; i <= n; i++) {
        // Step 3: Conditional Logic
        if (i % 3 === 0 && i % 5 === 0) {
            console.log("FizzBuzz");
        } else if (i % 3 === 0) {
            console.log("Fizz");
        } else if (i % 5 === 0) {
            console.log("Buzz");
        } else {
            console.log(i);
        }
    }
}

// Step 4: Test the Function
fizzBuzz(15);
```

## Explanation of the Code

1. **Function Definition:**
    a. The `fizzBuzz` function is defined to accept a number `n`.
2. **Loop Implementation:**
    a. A `for` loop runs from 1 to `n`, checking each number.
3. **Conditional Logic:**
    a. The `if` statements determine whether to print "Fizz", "Buzz", "FizzBuzz", or the number itself based on divisibility.
4. **Testing:**
    a. Calling `fizzBuzz(15)` will print the expected output for numbers 1 through 15.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `fizzbuzz`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/fizzbuzz.git
cd fizzbuzz
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the function.

Example content for `README.md`:

```
# FizzBuzz

This project implements the classic FizzBuzz problem using loops in
JavaScript.

## How to Use

Call the `fizzBuzz(n)` function, where `n` is the upper limit for the
counting.
For example, `fizzBuzz(15)` prints the FizzBuzz output for numbers 1
to 15.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add FizzBuzz implementation and README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.

## LEVEL MEDIUM

# Demo 02

## 2.Create a function that classifies a person's age group based on the following criteria:

- Child: 0-12 years
- Teenager: 13-19 years
- Adult: 20-64 years
- Senior: 65 years and above

***Requirements:***

1. Create a function called `classifyAge` that takes an age as an argument.
2. Use conditional statements to determine the age group.
3. If the input is invalid (not a number or negative), return an error message.
4. Test the function with various age inputs.

### Steps to Solve the Exercise

1. **Define the Function:**
    a. Create a function named `classifyAge` that accepts an `age` parameter.
2. **Input Validation:**
    a. Check if the input is a valid number and if it's non-negative.
3. **Implement Conditional Logic:**
    a. Use `if`, `else if`, and `else` statements to classify the age.
4. **Return the Classification:**
    a. Return the appropriate age group as a string.
5. **Test the Function:**
    a. Call the function with different age values and log the results.

### Solution

Here's how you can implement this in JavaScript:

```
Javascript
function classifyAge(age) {
    // Step 2: Input Validation
    if (typeof age !== 'number' || age < 0) {
        return "Error: Invalid input. Please enter a non-negative number for
age.";
    }
```

```
    // Step 3: Implement Conditional Logic
    let ageGroup;
    if (age <= 12) {
        ageGroup = "Child";
    } else if (age >= 13 && age <= 19) {
        ageGroup = "Teenager";
    } else if (age >= 20 && age <= 64) {
        ageGroup = "Adult";
    } else {
        ageGroup = "Senior";
    }

    // Step 4: Return the Classification
    return `You are classified as: ${ageGroup}`;
}

// Step 5: Test the Function
console.log(classifyAge(10));  // "You are classified as: Child"
console.log(classifyAge(15));  // "You are classified as: Teenager"
console.log(classifyAge(30));  // "You are classified as: Adult"
console.log(classifyAge(70));  // "You are classified as: Senior"
console.log(classifyAge(-5));  // "Error: Invalid input. Please enter a non-
negative number for age."
console.log(classifyAge("twenty")); // "Error: Invalid input. Please enter a
non-negative number for age."
```

## Explanation of the Code

1. **Function Definition:**
    a. The `classifyAge` function takes `age` as an input parameter.
2. **Input Validation:**
    a. The function checks if the input is a valid number and non-negative. If not, it returns an error message.
3. **Conditional Logic:**
    a. A series of conditional statements classify the input age into appropriate groups.
4. **Return Value:**
    a. The function returns the classification result as a string.
5. **Testing the Function:**
    a. Various test cases validate the function's behavior for different inputs.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**

a. Go to GitHub and create a new repository. Name it something like `age-classification`.

2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/age-classification.git
cd age-classification
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the function.

Example content for `README.md`:

```
# Age Classification

This project contains a simple function that classifies a person's age group based on input.

## How to Use

Call the `classifyAge(age)` function with a non-negative integer to receive the age classification.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add age classification function and README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.

By following these steps, you'll have successfully created, tested, and uploaded a JavaScript project to GitHub!

## LEVEL MEDIUM

### Demo 03

## 3.Create a function that checks if a given string is a palindrome (reads the same forwards and backwards).

***Requirements:***

1. Create a function called `isPalindrome` that takes a string as an argument.
2. The function should return `true` if the string is a palindrome and `false` otherwise.
3. Ignore spaces, punctuation, and case sensitivity in the check.

### Steps to Solve the Exercise

1. **Define the Function:**
    a. Create a function named `isPalindrome` that accepts a string parameter.
2. **Normalize the String:**
    a. Remove spaces and punctuation, and convert the string to lowercase.
3. **Check for Palindrome:**
    a. Compare the normalized string to its reverse.
4. **Return the Result:**
    a. Return `true` if it's a palindrome; otherwise, return `false`.
5. **Test the Function:**
    a. Call the function with various strings and log the results.

### Solution

Here's how you can implement this in JavaScript:

```
javascript
function isPalindrome(str) {
    // Step 2: Normalize the string
    const normalizedStr = str
        .replace(/[^A-Za-z0-9]/g, '') // Remove non-alphanumeric characters
        .toLowerCase(); // Convert to lowercase
```

```
    // Step 3: Check for palindrome
    const reversedStr = normalizedStr.split('').reverse().join('');

    // Step 4: Return the result
    return normalizedStr === reversedStr;
}

// Step 5: Test the Function
console.log(isPalindrome("A man, a plan, a canal, Panama")); // true
console.log(isPalindrome("racecar")); // true
console.log(isPalindrome("hello")); // false
console.log(isPalindrome("No 'x' in Nixon")); // true
console.log(isPalindrome("12321")); // true
console.log(isPalindrome("This is not a palindrome")); // false
```

## Explanation of the Code

1. **Function Definition:**
    a. The `isPalindrome` function takes a string `str` as input.
2. **Normalization:**
    a. The string is cleaned using a regular expression that removes all non-alphanumeric characters and converts it to lowercase.
3. **Checking for Palindrome:**
    a. The normalized string is reversed, and the original normalized string is compared to the reversed version.
4. **Return Value:**
    a. The function returns `true` if the string is a palindrome; otherwise, it returns `false`.
5. **Testing the Function:**
    a. Several test cases validate the function's behavior for different inputs.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `palindrome-checker`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/palindrome-checker.git
cd palindrome-checker
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the function.

Example content for `README.md`:

```
# Palindrome Checker

This project contains a function that checks if a given string is a
palindrome.

## How to Use

Call the `isPalindrome(str)` function with a string input to receive a
boolean indicating if it is a palindrome.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add palindrome checker function and README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files

## LEVEL MEDIUM

# Demo 04

## 4. Create a function that generates the Fibonacci sequence up to a specified number.

***Requirements:***

1. Create a function called `fibonacci` that takes a positive integer n as an argument.
2. The function should return an array containing the Fibonacci sequence up to n.
3. If n is less than 1, return an error message.

### Steps to Solve the Exercise

1. **Define the Function:**
    a. Create a function named `fibonacci` that accepts a parameter n.
2. **Input Validation:**
    a. Check if n is a positive integer. If not, return an error message.
3. **Generate the Fibonacci Sequence:**
    a. Initialize an array to hold the sequence and calculate Fibonacci numbers up to n.
4. **Return the Result:**
    a. Return the array containing the Fibonacci sequence.
5. **Test the Function:**
    a. Call the function with various inputs and log the results.

### Solution

Here's how you can implement this in JavaScript:

```
javascript
Copy code
function fibonacci(n) {
    // Step 2: Input Validation
    if (n < 1 || !Number.isInteger(n)) {
        return "Error: Please enter a positive integer.";
    }

    // Step 3: Generate the Fibonacci Sequence
    const sequence = [];
    let a = 0, b = 1;
```

```
    for (let i = 1; i <= n; i++) {
        sequence.push(a);
        const next = a + b;
        a = b;
        b = next;
    }

    // Step 4: Return the Result
    return sequence;
}

// Step 5: Test the Function
console.log(fibonacci(1));  // [0]
console.log(fibonacci(5));  // [0, 1, 1, 2, 3]
console.log(fibonacci(10)); // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
console.log(fibonacci(0));  // "Error: Please enter a positive integer."
console.log(fibonacci(-5)); // "Error: Please enter a positive integer."
console.log(fibonacci(5.5)); // "Error: Please enter a positive integer."
```

## Explanation of the Code

1. **Function Definition:**
    a. The `fibonacci` function takes a single parameter n.
2. **Input Validation:**
    a. The function checks if n is a positive integer. If not, it returns an error message.
3. **Generating the Fibonacci Sequence:**
    a. The function uses a loop to calculate Fibonacci numbers, storing them in the `sequence` array.
4. **Return Value:**
    a. The function returns the array containing the Fibonacci sequence.
5. **Testing the Function:**
    a. Various test cases validate the function's behavior for different inputs.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `fibonacci-sequence-generator`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
bash
Copy code
```

```
git clone https://github.com/yourusername/fibonacci-sequence-generator.git
cd fibonacci-sequence-generator
```

3. **Create the Files:**
    a. Create the necessary files:

```
bash
Copy code
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the function.

Example content for `README.md`:

```
markdown
Copy code
# Fibonacci Sequence Generator

This project contains a function that generates the Fibonacci sequence up to
a specified number.

## How to Use

Call the `fibonacci(n)` function with a positive integer to receive an array
containing the Fibonacci sequence.
```

6. **Stage and Commit Changes:**

```
bash
Copy code
git add .
git commit -m "Add Fibonacci sequence generator function and README"
```

7. **Push to GitHub:**

```
bash
Copy code
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.

## LEVEL DIFFICULT

## Demo 05

## 5. Implement a debounce function that limits the rate at which a function can be executed. This is particularly useful for optimizing performance in scenarios like handling user input events.

**Requirements:**

1. Create a function called `debounce` that takes two parameters:
    a. `func`: The function to debounce.
    b. `delay`: The number of milliseconds to wait before invoking `func`.
2. The `debounce` function should return a new function that, when executed, will only allow `func` to be called after `delay` milliseconds have passed since the last call.
3. If the returned function is called again before the `delay` period has expired, reset the timer.

### Steps to Solve the Exercise

1. **Define the Debounce Function:**
    a. Create a function named `debounce` that takes `func` and `delay` as parameters.
2. **Set Up a Timer:**
    a. Use a variable to keep track of the timeout.
3. **Return a New Function:**
    a. Inside `debounce`, return a new function that will clear the previous timeout and set a new one.
4. **Invoke the Function:**
    a. When the timer completes, call the original function with the appropriate context and arguments.
5. **Test the Debounce Function:**
    a. Call the debounced function multiple times in quick succession and log the results to see how it behaves.

## Solution

Here's how you can implement this in JavaScript:

```
javascript
function debounce(func, delay) {
    let timer;

    return function (...args) {
        const context = this;

        // Clear the existing timer
        clearTimeout(timer);

        // Set a new timer
        timer = setTimeout(() => {
            func.apply(context, args); // Call the original function
        }, delay);
    };
}

// Step 5: Test the Debounce Function
function logMessage() {
    console.log("Debounced function executed!");
}

// Create a debounced version of the logMessage function
const debouncedLogMessage = debounce(logMessage, 2000);

// Simulate rapid calls
debouncedLogMessage(); // Call 1
debouncedLogMessage(); // Call 2
debouncedLogMessage(); // Call 3

// Wait 3 seconds before calling again to see the effect
setTimeout(debouncedLogMessage, 3000); // Should execute after 3 seconds
```

## Explanation of the Code

1. **Function Definition:**
    a. The `debounce` function accepts two parameters: `func` (the function to debounce) and `delay` (the time in milliseconds to wait before executing).
2. **Timer Variable:**
    a. A variable `timer` is used to keep track of the timeout.

3. **Returning a New Function:**
    a. The returned function clears the existing timeout whenever it is called and sets a new one. This way, the original function only executes after the specified delay.
4. **Context and Arguments:**
    a. The original function is called using `func.apply(context, args)` to maintain the correct `this` context and pass any arguments.
5. **Testing the Function:**
    a. The test simulates rapid calls to the debounced function to demonstrate that it only executes after the specified delay.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `debounce-function`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/debounce-function.git
cd debounce-function
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the debounce function.

Example content for `README.md`:

```
# Debounce Function

This project contains a debounce function that limits the rate at which a
function can be executed.

## How to Use

Call the `debounce(func, delay)` function to create a debounced version of
`func`. This debounced function will only execute `func` after `delay`
```

```
milliseconds have passed since the last call.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add debounce function and README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.

## LEVEL DIFFICULT

### Demo 06

## 6. Implement a throttle function that limits the number of times a function can be called over time. This is useful for optimizing performance, particularly for events that trigger frequently, like scrolling or resizing.

***Requirements:***

1. Create a function called `throttle` that takes two parameters:
    a. `func`: The function to throttle.
    b. `limit`: The number of milliseconds to wait before allowing `func` to be called again.
2. The `throttle` function should return a new function that can only invoke `func` once every `limit` milliseconds.
3. Ensure that the last invocation of the throttled function is called after the final event.

### Steps to Solve the Exercise

1. **Define the Throttle Function:**
    a. Create a function named `throttle` that accepts `func` and `limit` as parameters.
2. **Track Time and Last Invocation:**
    a. Use variables to track the last time the function was invoked and a flag to indicate if the function is currently in execution.

3. **Return a New Function:**
    a. Inside `throttle`, return a new function that checks if enough time has passed since the last invocation before calling `func`.
4. **Invoke the Function:**
    a. Call the original function with the appropriate context and arguments, especially on the final invocation.
5. **Test the Throttle Function:**
    a. Call the throttled function multiple times in quick succession and log the results to see how it behaves.

## Solution

Here's how you can implement this in JavaScript:

```
function throttle(func, limit) {
    let lastFunc;
    let lastRan;

    return function (...args) {
        const context = this;

        if (!lastRan) {
            func.apply(context, args);
            lastRan = Date.now();
        } else {
            clearTimeout(lastFunc);
            lastFunc = setTimeout(function () {
                if (Date.now() - lastRan >= limit) {
                    func.apply(context, args);
                    lastRan = Date.now();
                }
            }, limit - (Date.now() - lastRan));
        }
    };
}

// Step 5: Test the Throttle Function
function logMessage() {
    console.log("Throttled function executed at", new Date().toISOString());
}

// Create a throttled version of the logMessage function
const throttledLogMessage = throttle(logMessage, 2000);

// Simulate rapid calls
```

```
setInterval(throttledLogMessage, 500); // Call every 500 ms
```

## Explanation of the Code

1. **Function Definition:**
    a. The `throttle` function accepts two parameters: `func` (the function to throttle) and `limit` (the time in milliseconds to wait before executing again).
2. **Tracking State:**
    a. The `lastFunc` variable is used to store the timeout, and `lastRan` keeps track of the last time `func` was invoked.
3. **Returning a New Function:**
    a. The returned function checks if enough time has passed since the last invocation. If so, it calls the original function; otherwise, it sets a timeout.
4. **Context and Arguments:**
    a. The original function is called using `func.apply(context, args)` to maintain the correct `this` context and pass any arguments.
5. **Testing the Function:**
    a. The test simulates rapid calls to the throttled function using `setInterval`, demonstrating that it only executes after the specified delay.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `throttle-function`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/throttle-function.git
cd throttle-function
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the throttle function.

Example content for `README.md`:

```
# Throttle Function

This project contains a throttle function that limits the number of times a
function can be called over time.

## How to Use

Call the `throttle(func, limit)` function to create a throttled version of
`func`. This throttled function will only execute `func` once every `limit`
milliseconds.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add throttle function and README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.

## LEVEL DIFFICULT

## Demo 07

## 7. Create a set of functions that can perform array manipulations based on user-defined criteria.

***Requirements:***

1. Create a function expression called `arrayManipulator` that takes two parameters:
    a. `arr`: An array of numbers.
    b. `operation`: A string that specifies the operation to perform on the array. It can be one of the following:
        i. `"sum"`: Return the sum of the array elements.
        ii. `"average"`: Return the average of the array elements.
        iii. `"max"`: Return the maximum value in the array.
        iv. `"min"`: Return the minimum value in the array.

2. If the array is empty or the operation is invalid, return an appropriate error message.
3. Use higher-order functions (e.g., `reduce`, `map`) where appropriate.

## Steps to Solve the Exercise

1. **Define the Function Expression:**
    a. Create a function expression named `arrayManipulator`.
2. **Implement Conditional Logic:**
    a. Use conditional statements or a switch statement to determine which operation to perform based on the `operation` parameter.
3. **Perform Calculations:**
    a. Use array methods like `reduce` to calculate the sum and average, and use `Math.max` and `Math.min` for maximum and minimum values.
4. **Return the Result:**
    a. Return the result of the operation or an appropriate error message.
5. **Test the Function:**
    a. Call the function with various inputs and log the results.

## Solution

Here's how you can implement this in JavaScript:

```
const arrayManipulator = function(arr, operation) {
    // Step 2: Check if array is empty
    if (arr.length === 0) {
        return "Error: The array is empty.";
    }

    // Step 3: Implement Conditional Logic
    switch (operation) {
        case "sum":
            return arr.reduce((acc, num) => acc + num, 0);
        case "average":
            return arr.reduce((acc, num) => acc + num, 0) / arr.length;
        case "max":
            return Math.max(...arr);
        case "min":
            return Math.min(...arr);
        default:
            return "Error: Invalid operation. Please use sum, average, max,
or min.";
    }
};

// Step 5: Test the Function
```

```
console.log(arrayManipulator([1, 2, 3, 4, 5], "sum"));       // 15
console.log(arrayManipulator([1, 2, 3, 4, 5], "average"));   // 3
console.log(arrayManipulator([1, 2, 3, 4, 5], "max"));       // 5
console.log(arrayManipulator([1, 2, 3, 4, 5], "min"));       // 1
console.log(arrayManipulator([], "sum"));                     // "Error: The
array is empty."
console.log(arrayManipulator([1, 2, 3], "median"));          // "Error:
Invalid operation. Please use sum, average, max, or min."
```

## Explanation of the Code

1. **Function Expression:**
    a. The `arrayManipulator` function is defined as a function expression, allowing it to be assigned to a variable.
2. **Empty Array Check:**
    a. The function first checks if the input array is empty and returns an error message if true.
3. **Conditional Logic:**
    a. A `switch` statement determines which operation to perform based on the `operation` parameter.
4. **Calculations:**
    a. The `reduce` method is used for summing and averaging, while `Math.max` and `Math.min` are used to find the maximum and minimum values.
5. **Testing the Function:**
    a. Several test cases validate the function's behavior for different inputs.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `array-manipulator`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/array-manipulator.git
cd array-manipulator
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.

5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the function.

Example content for `README.md`:

```
# Array Manipulator

This project contains a function that performs various operations on an array of numbers.

## How to Use

Call the `arrayManipulator(arr, operation)` function with an array and a string specifying the operation (sum, average, max, min) to receive the result.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add array manipulator function and README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.

# LEVEL DIFFICULT

## Demo 08

# 8. Create a module that fetches user data from an API and processes the data using a callback function. Use an IIFE to encapsulate the fetching logic.

***Requirements:***

1. Create an IIFE that fetches user data from a placeholder API (e.g., https://jsonplaceholder.typicode.com/users).
2. The IIFE should accept a callback function that processes the fetched data.
3. Handle errors appropriately.
4. Use asynchronous programming (Promises) to fetch the data.

## Steps to Solve the Exercise

1. **Create the IIFE:**
    a. Define an Immediately Invoked Function Expression that fetches user data.
2. **Fetch Data:**
    a. Use the Fetch API to retrieve user data from the API.
3. **Process Data with Callback:**
    a. Call the provided callback function with the fetched data.
4. **Handle Errors:**
    a. Include error handling for the fetch operation.
5. **Test the Module:**
    a. Create a callback function to log or manipulate the fetched data and invoke the IIFE.

## Solution

Here's how you can implement this in JavaScript:

```
// IIFE to fetch user data
(async function(fetchUsers) {
    try {
        const response = await
fetch('https://jsonplaceholder.typicode.com/users');
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        const data = await response.json();
        fetchUsers(data); // Call the callback with the fetched data
    } catch (error) {
        console.error('Error fetching data:', error);
    }
})(function(users) {
    // Callback function to process the fetched user data
    console.log('Fetched Users:', users);
    users.forEach(user => {
        console.log(`Name: ${user.name}, Email: ${user.email}`);
    });
```

```
});
```

## Explanation of the Code

1. **IIFE Definition:**
    a. The IIFE is defined as an async function that immediately invokes itself.
2. **Fetching Data:**
    a. The `fetch` method is used to retrieve user data from the placeholder API.
3. **Processing Data:**
    a. Once the data is fetched and parsed, the callback function is called with the fetched data as an argument.
4. **Error Handling:**
    a. Errors are caught and logged to the console.
5. **Callback Function:**
    a. The callback function logs the user data and formats it for output.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `async-data-fetcher`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/async-data-fetcher.git
cd async-data-fetcher
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the module.

Example content for `README.md`:

```
# Asynchronous Data Fetcher

This project contains an IIFE that fetches user data from a placeholder API
```

```
and processes it using a callback function.

## How to Use

The IIFE fetches user data from `https://jsonplaceholder.typicode.com/users`
and logs the results to the console. You can modify the callback function to
process the data as needed.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add async data fetcher with IIFE and README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.

# LEVEL MEDIUM

## Demo 09

## 9.Create a module that transforms an array of objects using arrow functions to demonstrate a method chaining and functional programming.

***Requirements:***

1. Create an array of user objects, each containing `name`, `age`, and `email`.
2. Implement the following functionalities using arrow functions:
    a. Filter users who are adults (age >= 18).
    b. Map the filtered users to a new array that contains only their names and emails.
    c. Sort the resulting array alphabetically by name.
3. Log the final array of transformed user data.

## Steps to Solve the Exercise

1. **Create the User Array:**
    a. Define an array of user objects with `name`, `age`, and `email` properties.
2. **Filter the Users:**
    a. Use the `filter` method with an arrow function to filter out users who are adults.
3. **Map to New Array:**
    a. Use the `map` method with an arrow function to create a new array containing only the names and emails of the filtered users.
4. **Sort the Array:**
    a. Use the `sort` method with an arrow function to sort the array by names.
5. **Log the Result:**
    a. Output the final array to the console.

## Solution

Here's how you can implement this in JavaScript:

```
javascript
// Step 1: Create the User Array
const users = [
    { name: "Alice", age: 25, email: "alice@example.com" },
    { name: "Bob", age: 17, email: "bob@example.com" },
    { name: "Charlie", age: 30, email: "charlie@example.com" },
    { name: "David", age: 15, email: "david@example.com" },
    { name: "Eve", age: 22, email: "eve@example.com" }
];

// Step 2, 3, and 4: Filter, Map, and Sort
const transformedUsers = users
    .filter(user => user.age >= 18) // Filter adults
    .map(({ name, email }) => ({ name, email })) // Map to new array
with name and email
    .sort((a, b) => a.name.localeCompare(b.name)); // Sort by name

// Step 5: Log the Result
console.log(transformedUsers);
```

## Explanation of the Code

1. **User Array:**
    a. An array of user objects is defined, each containing `name`, `age`, and `email`.
2. **Filtering Users:**
    a. The `filter` method is used to retain only users whose age is 18 or older, utilizing an arrow function for clarity.
3. **Mapping to New Array:**
    a. The `map` method creates a new array of objects that contain only the `name` and `email` properties.
4. **Sorting:**
    a. The `sort` method is used with a locale-aware comparison function to sort users by their names.
5. **Logging the Result:**
    a. The transformed user data is logged to the console, showing only adults and their emails.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `data-transformation`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/data-transformation.git
cd data-transformation
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the module.

Example content for `README.md`:

```
# Data Transformation

This project demonstrates the use of arrow functions in JavaScript to
transform an array of user objects.

## How to Use

The module filters users who are adults, maps the data to a new
structure, and sorts the results by name.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add data transformation with arrow functions and
README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.

## LEVEL DIFFICULT

## Demo 10

## 10.Create a module that provides advanced mathematical operations using functions. The module should include currying and a way to chain operations.

***Requirements:***

1. Create a function `add` that returns a curried function to add two numbers.
2. Create a function `subtract` that returns a curried function to subtract two numbers.

3. Create a higher-order function `createOperation` that takes a mathematical operation (e.g., "add", "subtract") and returns the corresponding curried function.
4. Allow chaining operations. For example, `createOperation("add")(5)(3)` should return 8.
5. Handle invalid operations gracefully.

## Steps to Solve the Exercise

1. **Create the Curried Functions:**
    a. Define the add and `subtract` functions using closures to create curried versions.
2. **Implement the Higher-Order Function:**
    a. Define `createOperation` to return the appropriate curried function based on the operation provided.
3. **Error Handling:**
    a. Return an error message if the operation is invalid.
4. **Test the Functions:**
    a. Call the functions to verify they work as intended, including chaining.

## Solution

Here's how you can implement this in JavaScript:

```
javascript
// Step 1: Create the Curried Functions
const add = (a) => (b) => a + b;
const subtract = (a) => (b) => a - b;

// Step 2: Implement the Higher-Order Function
const createOperation = (operation) => {
    switch (operation) {
        case "add":
            return add;
        case "subtract":
            return subtract;
        default:
            return () => "Error: Invalid operation. Please use 'add'
or 'subtract'.";
    }
};

// Step 4: Test the Functions
```

```
console.log(createOperation("add")(5)(3));          // 8
console.log(createOperation("subtract")(10)(4));    // 6
console.log(createOperation("multiply"));             // "Error:
Invalid operation. Please use 'add' or 'subtract'."
```

## Explanation of the Code

1. **Curried Functions:**
    a. The `add` function returns another function that takes the second argument, effectively allowing for partial application.
2. **Higher-Order Function:**
    a. The `createOperation` function uses a `switch` statement to return the appropriate curried function based on the input operation. If the operation is invalid, it returns an error message.
3. **Testing:**
    a. Various test cases are included to verify the functionality of both valid operations and error handling.

## Steps to Load the Code to GitHub

1. **Create a New Repository:**
    a. Go to GitHub and create a new repository. Name it something like `advanced-math-operations`.
2. **Clone the Repository:**
    a. Open your terminal and run:

```
git clone https://github.com/yourusername/advanced-math-operations.git
cd advanced-math-operations
```

3. **Create the Files:**
    a. Create the necessary files:

```
touch index.js
touch README.md
```

4. **Add Code to index.js:**
    a. Open `index.js` in your preferred code editor and copy the solution code into it.
5. **Add a README:**
    a. Open `README.md` and write a brief description of the project, including how to use the functions.

Example content for `README.md`:

```
# Advanced Math Operations

This project demonstrates advanced JavaScript concepts such as
currying and higher-order functions.

## How to Use

- Use `createOperation("add")` to get a curried add function.
- Use `createOperation("subtract")` to get a curried subtract
function.
- Example: `createOperation("add")(5)(3)` returns `8`.
```

6. **Stage and Commit Changes:**

```
git add .
git commit -m "Add advanced math operations and README"
```

7. **Push to GitHub:**

```
git push origin main
```

8. **View Your Repository:**
    a. Go back to GitHub and refresh the page to see your new files.