1. **FizzBuzz**

Write a function that prints numbers from 1 to n, but for multiples of 3, print "Fizz" instead of the number, and for multiples of 5, print "Buzz." For numbers that are multiples of both 3 and 5, print "FizzBuzz."

|  |
| --- |
| **Solution** |
| function fizzBuzz(n) {    for (let i = 1; i <= n; i++) {      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);      }    }  }  // Example usage:  fizzBuzz(15); |
| **Result** |
|  |

1. **Reverse a String**

Implement a function that reverses a given string with and without using built-in **reverse** methods.

|  |
| --- |
| **Solution** |
| //USING REVERSE METHOD  function reverseString(str) {    return str.split("").reverse().join("");  }  // Example usage:  const originalString = "Hello, World!";  const reversedString = reverseString(originalString);  console.log(reversedString); // Output: "!dlroW ,olleH"  /\* ------------------------------------------------------- \*/  //NOT USING REVERSE METHOD  function reverseString2(str) {    let reversed = "";    for (let i = str.length - 1; i >= 0; i--) {      reversed += str[i];    }    return reversed;  }  // Example usage:  const originalString2 = "Hello, World!";  const reversedString2 = reverseString(originalString);  console.log(reversedString2); // Output: "!dlroW ,olleH" |

1. **Palindrome Checker**

Write a function to determine if a given string is a palindrome (reads the same forwards and backwards) while ignoring spaces, punctuation, and capitalization.

|  |
| --- |
| **Result** |
| function isPalindrome(str) {    // Remove non-alphanumeric characters and convert to lowercase    const cleanStr = str.replace(/[^A-Za-z0-9]/g, "").toLowerCase();    // Reverse the cleaned string    const reversedStr = cleanStr.split("").reverse().join("");    // Check if the cleaned string and its reverse are equal    return cleanStr === reversedStr;  }  // Example usage:  const testString1 = "A man, a plan, a canal, Panama";  console.log(isPalindrome(testString1)); // Output: true  const testString2 = "racecar";  console.log(isPalindrome(testString2)); // Output: true  const testString3 = "Hello, World!";  console.log(isPalindrome(testString3)); // Output: false |

1. **Factorial**

Write a function to calculate the factorial of a given positive integer.

|  |
| --- |
| **Solution** |
| function factorial(n) {    if (n === 0 || n === 1) {      return 1; // Factorial of 0 and 1 is 1    } else {      return n \* factorial(n - 1);    }  }  // Example usage:  const number = 5;  const result = factorial(number);  console.log(`The factorial of ${number} is ${result}`); |
| **Result** |
|  |

1. **Anagram Checker**

Create a function that checks if two strings are anagrams of each other (contain the same characters in a different order). The word "secure" is an anagram of "rescue."

|  |
| --- |
| **Solution** |
| function areAnagrams(str1, str2) {      // Remove non-alphabetic characters, spaces, and convert both strings to lowercase      str1 = str1.replace(/[^a-z]/g, '').toLowerCase();      str2 = str2.replace(/[^a-z]/g, '').toLowerCase();        // Check if the lengths of the strings are equal      if (str1.length !== str2.length) {        return false;      }        // Sort the characters in both strings and compare them      const sortedStr1 = str1.split('').sort().join('');      const sortedStr2 = str2.split('').sort().join('');        return sortedStr1 === sortedStr2;    }      // Test cases    console.log(areAnagrams('secure', 'rescue'));             // Output: true    console.log(areAnagrams('Hello, World!', 'World, hello!')); // Output: true    console.log(areAnagrams('hello', 'world'));                 // Output: false |

1. **Array Manipulation**

Given an array of numbers, write functions to find the maximum and minimum values, calculate the sum, and find the average.

|  |
| --- |
| **Solution** |
| function findMaximum(arr) {    if (arr.length === 0) {      return undefined; // Return undefined for an empty array    }    return Math.max(...arr);  }  function findMinimum(arr) {    if (arr.length === 0) {      return undefined; // Return undefined for an empty array    }    return Math.min(...arr);  }  function calculateSum(arr) {    return arr.reduce((sum, num) => sum + num, 0);  }  function calculateAverage(arr) {    if (arr.length === 0) {      return undefined; // Return undefined for an empty array    }    const sum = calculateSum(arr);    return sum / arr.length;  }  // Test the functions  const numbers = [3, 7, 2, 8, 5];  console.log("Maximum:", findMaximum(numbers)); // Output: Maximum: 8  console.log("Minimum:", findMinimum(numbers)); // Output: Minimum: 2  console.log("Sum:", calculateSum(numbers)); // Output: Sum: 25  console.log("Average:", calculateAverage(numbers)); // Output: Average: 5 |

1. **Array Filtering**

Implement a function that filters an array of objects based on a given condition (e.g., filter out all objects with ages less than 18).

|  |
| --- |
| **Solution** |
| const people = [    { name: "Alice", age: 25 },    { name: "Bob", age: 30 },    { name: "Charlie", age: 35 },    { name: "David", age: 40 },  ];  function filterPeopleByAge(peopleArray, ageThreshold) {    return peopleArray.filter((person) => person.age > ageThreshold);  }  // Example usage:  const filteredPeople = filterPeopleByAge(people, 30);  console.log(filteredPeople); |
| **Result** |
|  |

1. **Callback Practice**

Solve problems that require the use of callback functions, such as filtering an array based on a **callback condition**

|  |
| --- |
| **Result – Callback Condition** |
| //In the example usage, we filter even numbers from the numbers array using the isEven callback function.  function filterArray(arr, callback) {    const filteredArray = [];    for (const item of arr) {      if (callback(item)) {        filteredArray.push(item);      }    }    return filteredArray;  }  // Example usage: Filter even numbers from an array  const numbers = [1, 2, 3, 4, 5, 6];  const isEven = (number) => number % 2 === 0;  const evenNumbers = filterArray(numbers, isEven);  console.log(evenNumbers); // Output: [2, 4, 6] |

1. **Object Manipulation**

Create functions to add, delete, or update properties of an object dynamically.

|  |
| --- |
| **Solution** |
| // Function to add a property to an object  function addProperty(obj, key, value) {    obj[key] = value;  }  // Function to delete a property from an object  function deleteProperty(obj, key) {    if (obj.hasOwnProperty(key)) {      delete obj[key];    }  }  // Function to update a property of an object  function updateProperty(obj, key, newValue) {    if (obj.hasOwnProperty(key)) {      obj[key] = newValue;    }  }  // Example usage:  const person = {    name: "Alice",    age: 30,  };  console.log(person); // Original object  addProperty(person, "city", "New York"); // Add the "city" property  console.log(person);  deleteProperty(person, "age"); // Delete the "age" property  console.log(person);  updateProperty(person, "name", "Bob"); // Update the "name" property  console.log(person); |

1. **Event Handling**

Write code to handle DOM events, such as creating an event listener for a button click or keyboard input.

|  |
| --- |
| **Solution** |
| <!DOCTYPE html>  <html>    <head>      <title>Event Handling Example</title>    </head>    <body>      <h1>Event Handling</h1>      <button id="clickButton">Click Me</button>      <input type="text" id="textInput" placeholder="Type something" />      <p id="output">Event results will be displayed here.</p>      <script>        const clickButton = document.getElementById("clickButton");        const textInput = document.getElementById("textInput");        const output = document.getElementById("output");        textInput.addEventListener("keyup", handleTextInput); // Add a keyup event listener to the text input        clickButton.addEventListener("click", handleClick); // Add a click event listener to the button        // Function to handle button click event        function handleClick() {          output.textContent = "Button was clicked!";        }        // Function to handle text input event        function handleTextInput() {          const inputValue = textInput.value;          output.textContent = `You typed: ${inputValue}`;        }        </script>    </body>  </html> |
| **Results** |
|  |

1. **DOM Manipulation**

Create a small web page and write JavaScript code to manipulate the DOM dynamically, such as adding, removing, or updating elements.

|  |
| --- |
| **Solution** |
| <!DOCTYPE html>  <html>    <head>      <title>DOM Manipulation Example</title>    </head>    <body>      <h1>To-Do List</h1>      <input type="text" id="taskInput" placeholder="Add a new task" />      <button id="addTaskButton">Add Task</button>      <ul id="taskList">        <!-- Tasks will be added here -->      </ul>      <script>        // Function to add a new task        function addTask() {          const taskInput = document.getElementById("taskInput");          const taskText = taskInput.value.trim();          const taskList = document.getElementById("taskList");          const addTaskButton = document.getElementById("addTaskButton");          if (taskText !== "") {            const taskItem = document.createElement("li");            taskItem.textContent = taskText;            // Add a delete button to each task            const deleteButton = document.createElement("button");            deleteButton.textContent = "Delete";            deleteButton.addEventListener("click", () => {              taskItem.remove();            });            taskItem.appendChild(deleteButton);            taskList.appendChild(taskItem);            // Clear the input field            taskInput.value = "";          }        }        // Add task when the button is clicked        addTaskButton.addEventListener("click", addTask);        // Add task when Enter key is pressed in the input field        taskInput.addEventListener("keyup", (event) => {          if (event.key === "Enter") {            addTask();          }        });      </script>    </body>  </html> |
| **Result** |
|  |

1. **Promise Handling**

Write code that demonstrates how to use Promises to handle asynchronous operations, such as fetching data from an API and handling errors.

|  |
| --- |
| **Result** |
| // Function to fetch data from an API using Promises  function fetchDataFromAPI() {    const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';    return new Promise((resolve, reject) => {      fetch(apiUrl)        .then((response) => {          if (!response.ok) {            throw new Error('Network response was not ok');          }          return response.json();        })        .then((data) => {          resolve(data);        })        .catch((error) => {          reject(error);        });    });  }  // Example usage: Fetch data from the API  fetchDataFromAPI()    .then((result) => {      console.log('API Data:', result);    })    .catch((error) => {      console.error('API Error:', error);    }); |

1. We define a function **fetchDataFromAPI** that returns a **Promise**.
2. Inside the **Promise**, we use the **fetch** function to make an API request to a sample API (JSONPlaceholder).
3. We check the **response status**, and if it's not "ok," we throw an **error**. If any errors occur during the fetch or JSON parsing, they are caught and rejected with an error.
4. We use the **.json()** method to parse the response body as JSON and resolve the Promise with the data.
5. We call the **fetchDataFromAPI** function, and after it resolves, we handle the result data in the **.then** block.
6. If there is an **error** during the API request or parsing, it is caught in the **.catch** block, where we can log and handle the error appropriately.

This code demonstrates how to use Promises to handle asynchronous operations, including error handling, when fetching data from an API. It's a common pattern for handling asynchronous operations in modern JavaScript applications.

1. **Asynchronous Programming Async/Await**

Solve problems related to asynchronous programming, such as implementing async/await with Promise-based functions

|  |
| --- |
| **Result** |
|  |
| **Solution** |
| // Simulated function to fetch user data from an API (returns a Promise)  function fetchUserData(userId) {    return new Promise((resolve, reject) => {      setTimeout(() => {        if (userId === 1) {          resolve({ id: 1, username: "john\_doe" });        } else {          reject(new Error("User not found"));        }      }, 1000); // Simulate a 1-second delay    });  }  // Asynchronous function that uses async/await to fetch user data  async function getUserInfo() {    try {      // Use await to wait for the Promise to resolve      const user = await fetchUserData(1);      console.log("User Data:", user);      // You can perform more operations here      // For example, fetch additional data or process the user object    } catch (error) {      console.error("Error:", error.message);    }  }  // Call the asynchronous function  getUserInfo(); |

1. We have a **fetchUserData** function that returns a Promise. This function simulates fetching user data from an API asynchronously. It resolves with user data if the **userId** is 1 and rejects with an error if the **userId** is anything else.
2. We define an asynchronous function called **getUserInfo**. Inside this function, we use async/await to make the asynchronous **fetchUserData** call appear more synchronous.
3. Using **await**, we wait for the **fetchUserData(1)** Promise to resolve. When it resolves successfully, the user data is stored in the user variable.
4. We use a **try...catch** block to **handle any errors** that may occur during the Promise execution. If there's an error (e.g., if the user is not found), it's caught and logged as an error message.
5. You can perform additional operations within the getUserInfo function after fetching the user data. This allows you to process the data or make further API calls as needed.
6. Finally, we call the getUserInfo function to initiate the asynchronous operation.

Using **async/await** with Promise-based functions simplifies asynchronous code and makes it more readable by allowing you to write asynchronous code that resembles synchronous code.

1. **Asynchronous Programming: multiple operations**

Solve problems related to asynchronous programming, such as handling multiple asynchronous operations in sequence or parallel.

|  |
| --- |
| **Solution** |
| // Simulated functions to fetch data from APIs (return Promises)  function fetchDataFromAPI1() {    return new Promise((resolve) => {      setTimeout(() => {        resolve({ data1: "Data from API 1" });      }, 2000); // Simulate a 2-second delay    });  }  function fetchDataFromAPI2() {    return new Promise((resolve) => {      setTimeout(() => {        resolve({ data2: "Data from API 2" });      }, 1500); // Simulate a 1.5-second delay    });  }  // Sequential approach (fetch data one after another)  async function fetchDataSequentially() {    try {      const result1 = await fetchDataFromAPI1();      console.log("Result 1:", result1);      const result2 = await fetchDataFromAPI2();      console.log("Result 2:", result2);    } catch (error) {      console.error("Error:", error.message);    }  }  // Parallel approach (fetch data simultaneously)  async function fetchDataInParallel() {    try {      const [result1, result2] = await Promise.all([fetchDataFromAPI1(), fetchDataFromAPI2()]);      console.log("Result 1:", result1);      console.log("Result 2:", result2);    } catch (error) {      console.error("Error:", error.message);    }  }  // Call the functions  // Uncomment one approach at a time to test it  // fetchDataSequentially(); // Sequential approach  fetchDataInParallel(); // Parallel approach |
| **Result** |
|  |

1. **Promises and Fetch API**

Write code to make API requests using the Fetch API and handle the responses using Promises.

|  |
| --- |
| **Result** |
|  |
| **Solution** |
| // Function to fetch data from an API and return a Promise  function fetchDataFromAPI() {    const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';    return fetch(apiUrl)      .then((response) => {        if (!response.ok) {          throw new Error('Network response was not ok');        }        return response.json();      })      .then((data) => {        return data;      })      .catch((error) => {        console.error('Error:', error);      });  }  // Call the fetchDataFromAPI function  fetchDataFromAPI()    .then((result) => {      console.log('API Data:', result);    })    .catch((error) => {      console.error('API Error:', error);    }); |

1. We define a function **fetchDataFromAPI** that makes a **GET** request to a sample API (JSONPlaceholder). This function returns a **Promise**.
2. Inside the **fetch** function, we check the response status. If the response is not "ok" (e.g., 404 or 500 status), we throw an **error**.
3. We use the **.json()** method to parse the **response body as JSON** and return it as a **Promise**.
4. If any **errors** occur during the fetch or JSON parsing, they are caught and logged in the **catch** block.
5. We call the **fetchDataFromAPI** function, and after it resolves, we handle the result data in the **.then** block.
6. If there is an **error** during the API request or parsing, it is caught in the **.catch** block.

This code demonstrates how to use **Promises** and the Fetch API to make API requests and handle responses. It's a common pattern for asynchronous operations in modern JavaScript applications.