Exercise 1: Declare and Assign Variables

```
// Exercise 1: Declare and Assign Variables
let userName = "Alice"; // Declaring a string variable [cite: 7, 8]
const favoriteNumber = 7; // Declaring a number constant
console.log("User Name:", userName);
console.log("Favorite Number:", favoriteNumber);
```

Exercise 2: Basic Arithmetic Operations

```
// Exercise 2: Basic Arithmetic Operations
let num1 = 10;
let num2 = 5;
let sum = num1 + num2;
let difference = num1 - num2;
let product = num1 * num2;
let quotient = num1 / num2;
console.log("Sum:", sum);
console.log("Difference:", difference);
console.log("Product:", product);
console.log("Quotient:", quotient);
```

Exercise 3: String Concatenation

```
// Exercise 3: String Concatenation
let firstName = "John";
let lastName = "Doe";
// Method 1: Using the + operator
let fullName = firstName + " " + lastName;
console.log("Full Name (using +):", fullName);
// Method 2: Using template literals (recommended for complex strings)
let fullGreeting = `Hello, my name is ${firstName} ${lastName}!`;
console.log("Full Greeting (using template literal):", fullGreeting);
```

Exercise 4: Conditional Statement (If/Else)

```
// Exercise 4: Conditional Statement (If/Else)
let age = 20;
// Try changing this value to 16, 18, 25
if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}
let anotherAge = 16;
if (anotherAge >= 18) {
  console.log("You are an adult. (for anotherAge)");
} else {
  console.log("You are a minor. (for anotherAge)");
}
```

Exercise 5: Conditional Statement (If/Else If/Else)

```
// Exercise 5: Conditional Statement (If/Else If/Else)
let score = 85;
// Try changing this value (e.g., 95, 72, 55)
let grade;
if (score >= 90) {
  grade = "A";
} else if (score >= 80) {
  grade = "B";
} else if (score >= 70) {
  grade = "C";
} else if (score >= 60) {
  grade = "D";
} else {
  grade = "F";
}
console.log(`With a score of ${score}, your grade is: ${grade}`);
```

Exercise 6: For Loop (Basic Iteration)

```
// Exercise 6: For Loop (Basic Iteration)
```

```
console.log("Numbers from 1 to 10:");
for (let i = 1; i <= 10; i++) {
  console.log(i);
}
```

Exercise 7: While Loop (Conditional Iteration)

```
// Exercise 7: While Loop (Conditional Iteration)
let count = 5;
console.log("Countdown from 5:");
while (count >= 1) {
  console.log(count);
  count--; // Decrement count by 1
}
console.log("Blast off!");
```

Exercise 8: Simple Function Definition and Call

```
// Exercise 8: Simple Function Definition and Call
function greet(name) {
  console.log(`Hello, ${name}!`);
}
// Call the function
greet("Alice");
greet("Bob");
greet("Charlie");
```

Exercise 9: Function with Return Value

```
// Exercise 9: Function with Return Value
function addNumbers(a, b) {
  return a + b;
}
// Call the function and store its result
let result1 = addNumbers(5, 3);
console.log("Sum of 5 and 3:", result1); // Expected: 8
let result2 = addNumbers(100, 20);
```

```
console.log("Sum of 100 and 20:", result2);
// Expected: 120
```

Exercise 10: Introduction to Arrays

```
// Exercise 10: Introduction to Arrays
let fruits = ["Apple", "Banana", "Orange"];
console.log("Original fruits array:", fruits);
// Accessing elements by index (arrays are zero-indexed)
console.log("First fruit:", fruits[0]); // "Apple"
console.log("Second fruit:", fruits[1]);
// "Banana"
console.log("Third fruit:", fruits[2]); // "Orange"
// Accessing the last element using .length property
console.log("Last fruit:", fruits[fruits.length - 1]);
// "Orange"
// Adding an element to the end of the array
fruits.push("Grape");
console.log("Fruits array after adding 'Grape':", fruits);
console.log("New last fruit:", fruits[fruits.length - 1]); // "Grape"
console.log("Total number of fruits:", fruits.length); // 4
```

Exercise 11: Iterating Over an Array with For Loop

```
// Exercise 11: Iterating Over an Array with For Loop
let numbers = [10, 20, 30, 40, 50];
console.log("Numbers in the array:");
for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}
```

Exercise 12: Iterating Over an Array with For...of Loop

```
// Exercise 12: Iterating Over an Array with For...of Loop
let numbers = [10, 20, 30, 40, 50];
console.log("Numbers in the array (using for...of):");
for (let number of numbers) { // 'number' here directly gets each element's value
```

```
  console.log(number);
}
```

Exercise 13: Array Method: forEach()

```
// Exercise 13: Array Method: forEach()
let names = ["Alice", "Bob", "Charlie"];
console.log("Greetings:");
names.forEach(function(name) { // The function here is a "callback"
  console.log(`Hello, ${name}!`);
});
// Using arrow function syntax (more common in modern JS)
console.log("\nGreetings (using arrow function):");
names.forEach(name => {
  console.log(`Hi there, ${name}.`);
});
```

Exercise 14: Object Basics

```
// Exercise 14: Object Basics
let person = {
  name: "John Doe",
  age: 30,
  isStudent: false,
  "favorite color": "blue" // Property with a space in its name
};
console.log("Person's details:");
console.log("Name:", person.name); // Accessing with dot notation
console.log("Age:", person["age"]); // Accessing with bracket notation
console.log("Is Student:", person.isStudent);
// Accessing property with a space in its name (only bracket notation works)
console.log("Favorite Color:", person["favorite color"]);
// Modifying a property
person.age = 31;
console.log("Updated Age:", person.age); // Adding a new property
person.city = "New York";
console.log("City:", person.city);
```

```
console.log("Updated Person object:", person);
```

Exercise 15: Function with Object as Argument

```
// Exercise 15: Function with Object as Argument
function displayBookInfo(book) {
  console.log(`Book: ${book.title} by ${book.author}`);
}
// Optional: Using object destructuring in function parameters
function displayBookInfoDestructured({ title, author }) {
  console.log(`Book (destructured): ${title} by ${author}`);
}
let myBook = {
  title: "The Great Gatsby",
  author: "F. Scott Fitzgerald",
  year: 1925
};
let anotherBook = {
  title: "1984",
  author: "George Orwell",
  pages: 328
};
displayBookInfo(myBook);
displayBookInfo(anotherBook);
// Even if 'pages' is present, function only uses 'title' and 'author'
displayBookInfoDestructured(myBook);
```

Exercise 16: Array of Objects

```
// Exercise 16: Array of Objects
let students = [
  { name: "Alice", grade: 92 },
  { name: "Bob", grade: 78 },
  { name: "Charlie", grade: 85 },
  { name: "Diana", grade: 60 }
];
console.log("Student Grades:");
```

```
for (let student of students) {
  console.log(`${student.name}: ${student.grade}`);
}
// Using forEach
console.log("\nStudent Grades (using forEach):");
students.forEach(student => {
  console.log(`- ${student.name} got a ${student.grade}`);
});
```

Exercise 17: Basic String Methods

```
// Exercise 17: Basic String Methods
let message = "Hello JavaScript World";
console.log("Original Message:", message);
console.log("Length of message:", message.length); // 22
console.log("Uppercase:", message.toUpperCase()); // "HELLO JAVASCRIPT WORLD"
console.log("Lowercase:", message.toLowerCase());
// "hello javascript world"
// Finding the index of a substring
let jsIndex = message.indexOf("JavaScript");
console.log("Index of 'JavaScript':", jsIndex);
// 6 (index where 'J' starts)
// Extracting a substring using slice()
// slice(startIndex, endIndex - 1)
let extractedWord = message.slice(jsIndex, jsIndex + "JavaScript".length);
console.log("Extracted word:", extractedWord); // "JavaScript"
// Check if a string includes a substring
console.log("Does message include 'World'?", message.includes("World"));
// true
```

Exercise 18: Function to Reverse a String

```
// Exercise 18: Function to Reverse a String
function reverseString(str) {
  let reversedStr = "";
  for (let i = str.length - 1; i >= 0; i--) {
    reversedStr += str[i];
```

```
    // Append character to the reversed string
  }
  return reversedStr;
}
// Alternative using array methods (more advanced, but common)
function reverseStringMethod(str) {
  return str.split('').reverse().join('');
}
console.log("Reversed 'hello':", reverseString("hello")); // Expected: "olleh"
console.log("Reversed 'world':", reverseString("world")); // Expected: "dlrow"
console.log("Reversed 'JavaScript':", reverseString("JavaScript"));
// Expected: "tpircSavaJ"
console.log("Reversed 'hello' (method):", reverseStringMethod("hello"));
```

Exercise 19: Find the Largest Number in an Array

```
// Exercise 19: Find the Largest Number in an Array
function findLargestNumber(numbers) {
  if (numbers.length === 0) {
    return undefined;
    // Or throw an error, or return a specific value
  }
  let largest = numbers[0];
  // Assume the first element is the largest initially
  for (let i = 1; i < numbers.length; i++) { // Start from the second element
    if (numbers[i] > largest) {
      largest = numbers[i];
      // Update largest if current number is greater
    }
  }
  return largest;
}
console.log("Largest in [3, 8, 1, 12, 5]:", findLargestNumber([3, 8, 1, 12, 5]));
// Expected: 12
console.log("Largest in [100, 20, 30]:", findLargestNumber([100, 20, 30])); //
Expected: 100
console.log("Largest in [7]:", findLargestNumber([7]));
// Expected: 7
```

```
console.log("Largest in []:", findLargestNumber([])); // Expected: undefined (due to added check)
```

Exercise 20: Calculate the Sum of Array Elements

```
// Exercise 20: Calculate the Sum of Array Elements
function calculateSum(numbers) {
  let totalSum = 0;
  // Initialize sum to zero
  for (let i = 0; i < numbers.length; i++) {
    totalSum += numbers[i];
    // Add current number to totalSum
  }
  return totalSum;
}
// Alternative using for...of loop
function calculateSumForOf(numbers) {
  let totalSum = 0;
  for (let num of numbers) {
    totalSum += num;
  }
  return totalSum;
}
// Alternative using reduce() (more advanced)
function calculateSumReduce(numbers) {
  return numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
}
console.log("Sum of [1, 2, 3, 4, 5]:", calculateSum([1, 2, 3, 4, 5]));
// Expected: 15
console.log("Sum of [10, 20, 30]:", calculateSumForOf([10, 20, 30])); // Expected: 60
console.log("Sum of []:", calculateSum([]));
// Expected: 0
console.log("Sum of [7, 8, 9] (reduce):", calculateSumReduce([7, 8, 9])); // Expected: 24
```

Exercise 21: Array Method: filter()

```
// Exercise 21: Array Method: filter()
let data = [10, 25, 30, 45, 50, 65, 5, 80];
// Filter numbers greater than 40
let greaterThan40 = data.filter(function(number) {
  return number > 40;
});
console.log("Numbers greater than 40:", greaterThan40); // Expected: [45, 50, 65, 80]
// Using arrow function syntax
let evenNumbers = data.filter(number => number % 2 === 0);
console.log("Even numbers:", evenNumbers); // Expected: [10, 30, 50, 80]
// Filtering objects in an array
let products = [
  { name: "Laptop", price: 1200 },
  { name: "Mouse", price: 25 },
  { name: "Keyboard", price: 75 },
  { name: "Monitor", price: 300 }
];
let expensiveProducts = products.filter(product => product.price > 100);
console.log("Expensive products (> $100):", expensiveProducts);
// Expected: [{ name: "Laptop", price: 1200 }, { name: "Monitor", price: 300 }]
```

Exercise 22: Array Method: map()

```
// Exercise 22: Array Method: map()
let prices = [10, 20, 30, 45];
// Double each price
let doubledPrices = prices.map(function(price) {
  return price * 2;
});
console.log("Doubled prices:", doubledPrices);
// Expected: [20, 40, 60, 90]
// Using arrow function syntax
let pricesWithTax = prices.map(price => price * 1.05);
// Add 5% tax
console.log("Prices with 5% tax:", pricesWithTax); // Mapping an array of objects to
get specific properties
let users = [
  { id: 1, name: "Alice", email: "alice@example.com" },
```

```
  { id: 2, name: "Bob", email: "bob@example.com" },
  { id: 3, name: "Charlie", email: "charlie@example.com" }
];
let userNames = users.map(user => user.name);
console.log("User names:", userNames); // Expected: ["Alice", "Bob", "Charlie"]
```

Exercise 23: Array Method: reduce()

```
// Exercise 23: Array Method: reduce()
let items = [5, 10, 15, 20];
// Summing all numbers in an array
let sum = items.reduce(function(accumulator, currentValue) {
  console.log(`Accumulator: ${accumulator}, Current Value: ${currentValue}`);
  return accumulator + currentValue;
}, 0);
// 0 is the initial value for the accumulator
console.log("Sum of items:", sum);
// Expected: 50
// Using arrow function syntax (common)
let product = items.reduce((acc, val) => acc * val, 1);
// Initial value for product is 1
console.log("Product of items:", product);
// Expected: 5 * 10 * 15 * 20 = 15000
// Reducing an array of objects to a single value
let cart = [
  { item: "Shirt", price: 25 },
  { item: "Jeans", price: 60 },
  { item: "Socks", price: 10 }
];
let totalCartPrice = cart.reduce((total, currentItem) => total + currentItem.price, 0);
console.log("Total cart price:", totalCartPrice);
// Expected: 95
```

Exercise 24: Closures

```
// Exercise 24: Closures
function makeCounter() {
```

```
  let count = 0;
  // 'count' is in the outer function's scope
  return function() { // This inner function forms a closure
    count++;
    // It "remembers" and can access 'count' from its lexical environment
    return count;
  };
}
let counter1 = makeCounter(); // counter1 is now the inner function returned by
makeCounter()
console.log("Counter 1:");
console.log(counter1());
// Expected: 1
console.log(counter1()); // Expected: 2
console.log(counter1()); // Expected: 3
let counter2 = makeCounter();
// A new, independent counter
console.log("\nCounter 2:");
console.log(counter2()); // Expected: 1 (starts fresh)
console.log(counter1());
// Expected: 4 (counter1 continues independently)
```

Exercise 25: Basic Object-Oriented Programming (Constructor Function / Class)

```
// Exercise 25: Basic Object-Oriented Programming (Constructor Function)
// Using a Constructor Function (traditional ES5 way)
function Car(make, model) {
  this.make = make;
  this.model = model;
}
// Add a method to the Car's prototype
// This ensures all instances share the same method, saving memory
Car.prototype.displayInfo = function() {
  console.log(`This is a ${this.make} ${this.model}.`);
};
// Create instances
let car1 = new Car("Toyota", "Camry");
```

```
let car2 = new Car("Honda", "Civic");
car1.displayInfo();
// Expected: "This is a Toyota Camry."
car2.displayInfo(); // Expected: "This is a Honda Civic."
// --- Using ES6 Class Syntax (modern way, syntactic sugar over prototypes) ---
class Motorcycle {
  constructor(brand, type) {
    this.brand = brand;
    this.type = type;
  }
  displayDetails() {
    console.log(`This is a ${this.brand} ${this.type} motorcycle.`);
  }
}
let moto1 = new Motorcycle("Harley-Davidson", "Sportster");
let moto2 = new Motorcycle("Kawasaki", "Ninja");
moto1.displayDetails();
moto2.displayDetails();
```

Exercise 26: Asynchronous JavaScript - Callbacks (Simulated)

```
// Exercise 26: Asynchronous JavaScript - Callbacks (Simulated)
function fetchUserData(userId, callback) {
  console.log(`Fetching data for user ID: ${userId}...`);
  // Simulate an asynchronous operation (e.g., network request)
  setTimeout(() => {
    const user = {
      id: userId,
      name: `User ${userId}`,
      email: `user${userId}@example.com`
    };
    console.log(`Data for user ${userId} received.`);
    callback(user); // Execute the callback with the fetched data
  }, 2000);
  // Simulate a 2-second delay
}
// How to use it:
```

```
console.log("Starting data fetch for User 1.");
fetchUserData(1, function(user) {
  console.log("Processing fetched user data:");
  console.log("User Name:", user.name);
  console.log("User Email:", user.email);
});
console.log("\nStarting data fetch for User 2.");
fetchUserData(2, (user) => { // Using arrow function for callback
  console.log("Processing fetched user data for User 2:");
  console.log("User ID:", user.id);
});
console.log("Requests initiated. This message appears first because the fetch is
async.");
// The "Requests initiated..." message will appear immediately,
// before the "Data received" messages, demonstrating asynchronicity.
```

Exercise 27: Asynchronous JavaScript - Promises (Basic)

```
// Exercise 27: Asynchronous JavaScript - Promises (Basic)
function fetchUserDataPromise(userId) {
  console.log(`(Promise) Fetching data for user ID: ${userId}...`);
  return new Promise((resolve, reject) => { // A Promise takes a function with resolve
and reject
    setTimeout(() => {
      if (userId === 0) {
        reject("User with ID 0 not found."); // Simulate an error
        return;
      }
      const user = {
        id: userId,
        name: `Promise User ${userId}`,
        status: "active"
      };
      console.log(`(Promise) Data for user ${userId} received.`);
      resolve(user); // Resolve the promise with the user data
    }, 2000);
  });
```

```
}
// Using the Promise:
console.log("--- Fetching User 1 (Success Case) ---");
fetchUserDataPromise(1)
  .then((user) => { // .then() is called when the promise resolves
    console.log("Success! User 1 Data:", user);
    console.log(`Resolved User 1 Name: ${user.name}`);
  })
  .catch((error) => { // .catch() is called when the promise rejects
    console.error("Error fetching User 1:", error);
  });
console.log("\n--- Fetching User 0 (Error Case) ---");
fetchUserDataPromise(0)
  .then((user) => {
    console.log("Success! User 0 Data:", user); // This block will NOT execute
  })
  .catch((error) => {
    console.error("Error fetching User 0:", error); // This block WILL execute
  });
console.log("Promise requests initiated. This message appears first.");
```

Exercise 28: Recursion - Factorial Calculation

```
// Exercise 28: Recursion - Factorial Calculation
function factorial(n) {
  // Base case: When to stop the recursion
  if (n === 0 || n === 1) {
    return 1;
  }
  // Recursive step: Call the function itself with a smaller problem
  else {
    return n * factorial(n - 1);
  }
}
console.log("Factorial of 0:", factorial(0)); // Expected: 1
console.log("Factorial of 1:", factorial(1)); // Expected: 1
console.log("Factorial of 5:", factorial(5));
// Expected: 120 (5 * 4 * 3 * 2 * 1)
```

```
console.log("Factorial of 7:", factorial(7));
// Expected: 5040
// console.log("Factorial of -1:", factorial(-1)); // This would lead to infinite recursion without proper handling
```

Exercise 29: Higher-Order Function - map() with Objects

```
// Exercise 29: Higher-Order Function - map() with Objects
let products = [
  { id: 1, name: "Laptop", price: 1200, category: "Electronics" },
  { id: 2, name: "Mouse", price: 25, category: "Electronics" },
  { id: 3, name: "Notebook", price: 15, category: "Stationery" },
  { id: 4, name: "Desk Chair", price: 250, category: "Furniture" }
];
// Add a priceWithTax property to each product
const TAX_RATE = 0.15;
let productsWithTax = products.map(product => {
  return {
    ...product, // Copies all existing properties from the original product object
    priceWithTax: product.price * (1 + TAX_RATE) // Adds the new property
  };
});
console.log("Products with Tax:", productsWithTax);
/* Expected Output Structure (approx):
[
  { id: 1, name: "Laptop", price: 1200, category: "Electronics", priceWithTax: 1380 },
  { id: 2, name: "Mouse", price: 25, category: "Electronics", priceWithTax: 28.75 },
  ...
] */
// Transform products into a simplified list for display
let productTitles = products.map(product => `${product.name} ($${product.price})`);
console.log("\nProduct Titles:", productTitles); // Expected: ["Laptop ($1200)", "Mouse ($25)", ...]
```

Exercise 30: Chaining Array Methods

```
// Exercise 30: Chaining Array Methods
let transactions = [
  { id: 1, amount: 100, type: 'credit', date: '2023-01-01' },
  { id: 2, amount: 50, type: 'debit', date: '2023-01-02' },
  { id: 3, amount: 200, type: 'credit', date: '2023-01-03' },
  { id: 4, amount: 30, type: 'debit', date: '2023-01-04' },
  { id: 5, amount: 150, type: 'credit', date: '2023-01-05' }
];
// Calculate the total amount of all credit transactions
let totalCreditAmount = transactions
  .filter(transaction => transaction.type === 'credit') // Step 1: Filter credit transactions
  .map(creditTransaction => creditTransaction.amount) // Step 2: Extract amounts
  .reduce((sum, amount) => sum + amount, 0);
// Step 3: Sum the amounts
console.log("Transactions:", transactions);
console.log("Total Credit Amount:", totalCreditAmount);
// Expected: 100 + 200 + 150 = 450
// Another example: Get names of users older than 25
let people = [
  { name: "Alice", age: 20 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 25 },
  { name: "Diana", age: 35 }
];
let namesOfAdults = people
  .filter(person => person.age > 25)
  .map(adult => adult.name);
console.log("\nNames of people older than 25:", namesOfAdults); // Expected: ["Bob",
"Diana"]
```

Exercise 31: Error Handling with try...catch

```
// Exercise 31: Error Handling with try...catch
function divide(a, b) {
  try {
    if (b === 0) {
      throw new Error("Cannot divide by zero.");
      // Throw an error if b is 0
```

```
    }
    return a / b;
    // Perform division if b is not 0
  } catch (error) {
    console.error("An error occurred:", error.message);
    // Catch and log the error message
    return NaN;
    // Return Not-a-Number or some other indicative value
  }
}
console.log("10 / 2 =", divide(10, 2));
// Expected: 5
console.log("7 / 0 =", divide(7, 0)); // Expected: An error message and NaN
console.log("15 / 3 =", divide(15, 3));
// Expected: 5
```

Exercise 32: Understanding this Keyword Context

```
// Exercise 32: Understanding 'this' Keyword Context
let calculator = {
  value: 0, // Initial value
  // Method to add a number to the current value
  add: function(num) {
    this.value += num;
    // 'this' refers to the 'calculator' object
    return this;
    // Return 'this' to allow method chaining
  },
  // Method to get the current result
  getResult: function() {
    return this.value;
    // 'this' refers to the 'calculator' object
  },
  // Example of 'this' context changing inside a regular function
  // (This will be clarified with arrow functions later)
  debugValueLater: function() {
    setTimeout(function() {
      // console.log("Value inside setTimeout (problematic 'this'):", this.value);
```

```
    // In strict mode (default for modules), 'this' here would be undefined.
    // In non-strict mode (old browsers), 'this' would be the global object
(window/global).
    // This highlights why arrow functions are often preferred for callbacks.
   }, 100);
  }
};
// Demonstrate method chaining
let finalResult = calculator.add(5).add(10).add(20).getResult();
console.log("Chained result:", finalResult);
// Expected: 35
// Reset and try another sequence
calculator.value = 0;
// Reset for a new calculation
let anotherResult = calculator.add(2).add(3).getResult();
console.log("Another result:", anotherResult);
// Expected: 5
```

Exercise 33: Arrow Functions (=>)

```
// Exercise 33: Arrow Functions (=>)
// Example 1: `forEach` with arrow function
let numbers = [1, 2, 3, 4, 5];
console.log("Numbers via forEach (arrow function):");
numbers.forEach(num => console.log(num * 2));
// Concise syntax for single expression
// Example 2: `map` with arrow function
let names = ["Alice", "Bob", "Charlie"];
let uppercasedNames = names.map(name => name.toUpperCase());
console.log("Uppercased names (arrow function):", uppercasedNames);
// Example 3: Arrow functions and `this` binding (lexical `this`)
let person = {
  name: "John Doe",
  // Regular function for method definition
  greetDelayed: function() {
    console.log(`Hello from ${this.name}!`);
    // 'this' correctly refers to 'person'
    // Using an arrow function for the setTimeout callback
```

```
    // Arrow functions do NOT bind their own 'this'.
    // They inherit 'this' from the enclosing (lexical) scope.
    setTimeout(() => {
      console.log(`Delayed greeting from ${this.name}.`); // 'this' still refers to 'person'
    }, 1000);
    // For comparison: if you used a regular function here, 'this' would be different
    setTimeout(function() {
      // console.log(`Problematic delayed greeting from ${this.name}.`);
      // 'this' would be 'window' or 'undefined' in strict mode
    }, 1200);
  }
};
person.greetDelayed();
```

Exercise 34: Object Destructuring

```
// Exercise 34: Object Destructuring
let movie = {
  title: "Inception",
  director: "Christopher Nolan",
  year: 2010,
  rating: 8.8
};
// 1. Basic destructuring
const { title, director } = movie;
console.log("Title:", title); // Expected: Inception
console.log("Director:", director);
// Expected: Christopher Nolan
// 2. Destructuring with renaming
const { year, rating: imdbRating } = movie;
console.log("Year:", year);
// Expected: 2010
console.log("IMDB Rating:", imdbRating); // Expected: 8.8 (using new variable name)
// 3. Destructuring with default values for non-existent properties
const { genre = "Sci-Fi", producer = "Unknown" } = movie;
console.log("Genre (with default):", genre); // Expected: Sci-Fi
console.log("Producer (with default):", producer);
// Expected: Unknown
```

```
// Destructuring in function parameters (common use case)
function displayMovieDetails({ title, director, year, runtime = "N/A" }) {
  console.log(`\nDetails: ${title} (${year}) by ${director}. Runtime: ${runtime}`);
}
displayMovieDetails(movie);
displayMovieDetails({ title: "Avatar", director: "James Cameron", year: 2009 });
// No runtime provided
```

Exercise 35: Array Destructuring

```
// Exercise 35: Array Destructuring
let rgb = ["red", "green", "blue", "alpha", "cyan"];
// 1. Basic destructuring
const [firstColor, secondColor] = rgb;
console.log("First color:", firstColor); // Expected: red
console.log("Second color:", secondColor);
// Expected: green
// 2. Skipping elements
const [, , thirdColor] = rgb;
// Skip first two elements with empty commas
console.log("Third color:", thirdColor);
// Expected: blue
// 3. Rest pattern: collects remaining elements into a new array
const [primaryColor, ...otherColors] = rgb;
console.log("Primary Color:", primaryColor); // Expected: red
console.log("Other Colors:", otherColors); // Expected: ["green", "blue", "alpha",
"cyan"]
// Destructuring with default values
const [color1, color2, color3, color4, color5 = "magenta"] = rgb;
console.log("Color 5 (with default):", color5); // Expected: magenta (if not enough
elements)
// Swapping variables easily with destructuring
let x = 10;
let y = 20;
[x, y] = [y, x]; // Swap x and y without a temporary variable
console.log(`\nSwapped: x = ${x}, y = ${y}`);
// Expected: x = 20, y = 10
```

Exercise 36: Spread Operator (...) - Arrays

```
// Exercise 36: Spread Operator (...) - Arrays
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
// 1. Combining arrays
let combinedArr = [...arr1, ...arr2];
console.log("Combined Array:", combinedArr);
// Expected: [1, 2, 3, 4, 5, 6]
let moreCombined = [0, ...arr1, 10, ...arr2, 7];
console.log("More Combined:", moreCombined);
// Expected: [0, 1, 2, 3, 10, 4, 5, 6, 7]
// 2. Copying arrays (shallow copy)
let arr1Copy = [...arr1];
console.log("Array 1 Copy:", arr1Copy); // Expected: [1, 2, 3]
// Verify it's a copy (modifying copy doesn't affect original)
arr1Copy.push(99);
console.log("Array 1 after copy modified:", arr1); // Expected: [1, 2, 3]
console.log("Array 1 Copy after modification:", arr1Copy);
// Expected: [1, 2, 3, 99]
// 3. Passing array elements as function arguments
function sumAll(a, b, c) {
  return a + b + c;
}
let numbersForSum = [10, 20, 30];
console.log("Sum of numbersForSum (using spread):", sumAll(...numbersForSum));
// Expected: 60
// Using Math.max() with spread
let grades = [85, 92, 78, 95, 88];
console.log("Max grade:", Math.max(...grades));
// Expected: 95
```

Exercise 37: Spread Operator (...) - Objects

```
// Exercise 37: Spread Operator (...) - Objects
let user = { name: "Jane", age: 28 };
// 1. Copying objects and adding new properties
```

```
let userCopy = { ...user, city: "London" };
// Creates a new object, copies properties, adds/overrides 'city'
console.log("Original User:", user);
// Expected: { name: "Jane", age: 28 }
console.log("User Copy:", userCopy);
// Expected: { name: "Jane", age: 28, city: "London" }
// 2. Merging objects
let address = { street: "123 Main St", zip: "10001" };
let contactInfo = { email: "jane@example.com", phone: "555-1234" };
let userProfile = { ...user, ...address, ...contactInfo, occupation: "Engineer" };
console.log("User Profile (merged):", userProfile);
/* Expected: {
  name: "Jane", age: 28, street: "123 Main St",
  zip: "10001", email: "jane@example.com", phone: "555-1234",
  occupation: "Engineer"
} */
// Handling conflicts (later properties override earlier ones)
let baseSettings = { theme: "dark", fontSize: 16 };
let userSettings = { fontSize: 18, notifications: true };
let finalSettings = { ...baseSettings, ...userSettings };
console.log("Final Settings (conflict resolved):", finalSettings); // Expected: { theme:
"dark", fontSize: 18, notifications: true }
```

Exercise 38: Ternary Operator (Conditional Operator)

```
// Exercise 38: Ternary Operator (Conditional Operator)
let temperature1 = 30;
let weatherStatus1 = (temperature1 > 25) ?
"Hot" : "Cold";
console.log(`Temperature: ${temperature1}°C, Status: ${weatherStatus1}`); //
Expected: Hot
let temperature2 = 18;
let weatherStatus2 = (temperature2 > 25) ? "Hot" : "Cold";
console.log(`Temperature: ${temperature2}°C, Status: ${weatherStatus2}`);
// Expected: Cold
// Another example: Check if a user is logged in
let isLoggedIn = true;
let message = isLoggedIn ? "Welcome back!" : "Please log in.";
```

```
console.log(message); // Expected: Welcome back!
// Nested ternary (use sparingly for readability)
let time = 14;
// 2 PM
let greeting = (time < 12) ? "Good morning!"
: (time < 18) ?
"Good afternoon!" : "Good evening!";
console.log(greeting); // Expected: Good afternoon!
```

Exercise 39: Nullish Coalescing Operator (??)

```
// Exercise 39: Nullish Coalescing Operator (??)
let userName1 = null;
let defaultName = "Guest";
const displayName1 = userName1 ?? defaultName; // Falls back if userName1 is null or
undefined
console.log("Display Name 1 (null):", displayName1);
// Expected: Guest
let userName2 = undefined;
const displayName2 = userName2 ?? defaultName;
console.log("Display Name 2 (undefined):", displayName2);
// Expected: Guest
let userName3 = "Alice";
const displayName3 = userName3 ?? defaultName;
console.log("Display Name 3 (value):", displayName3);
// Expected: Alice
// --- Difference between ?? and ||
// The logical OR (||) operator considers `false`, `0`, `""` (empty string), `null`,
`undefined` as "falsy".
// The nullish coalescing operator (??) only considers `null` and `undefined` as
"nullish".
let valueZero = 0;
let valueEmptyString = "";
let valueFalse = false;
// Using ||
console.log("\n--- Using || (Logical OR) ---");
console.log("valueZero || 'Default':", valueZero || "Default"); // Expected: Default (0 is
falsy)
```

```
console.log("valueEmptyString || 'Default':", valueEmptyString || "Default");
// Expected: Default ("" is falsy)
console.log("valueFalse || 'Default':", valueFalse || "Default");
// Expected: Default (false is falsy)
// Using ??
console.log("\n--- Using ?? (Nullish Coalescing) ---");
console.log("valueZero ?? 'Default':", valueZero ?? "Default"); // Expected: 0 (0 is not nullish)
console.log("valueEmptyString ?? 'Default':", valueEmptyString ?? "Default");
// Expected: "" ("" is not nullish)
console.log("valueFalse ?? 'Default':", valueFalse ?? "Default");
// Expected: false (false is not nullish)
```

Exercise 40: Optional Chaining (?.)

```
// Exercise 40: Optional Chaining (?.)
let user1 = {
  name: "Alice",
  email: "alice@example.com",
  address: {
    street: "123 Main St",
    city: "Anytown"
  }
};
let user2 = {
  name: "Bob",
  email: "bob@example.com"
  // No address property
};
let user3 = {
  name: "Charlie",
  contact: {
    email: "charlie@example.com"
  }
};
// Safely access nested properties using optional chaining
console.log("User 1 Street:", user1.address?.street);
// Expected: 123 Main St
```

```
console.log("User 2 Street:", user2.address?.street); // Expected: undefined (no error)
console.log("User 1 Zip Code:", user1.address?.zipCode);
// Expected: undefined (property doesn't exist)
// Accessing a potentially non-existent nested object property
console.log("User 1 Company Name:", user1.company?.name);
// Expected: undefined (no error)
console.log("User 2 Company Name:", user2.company?.name);
// Expected: undefined (no error)
// Combining with Nullish Coalescing for a fallback
let user1City = user1.address?.city ?? "N/A";
let user2City = user2.address?.city ?? "N/A";
console.log("User 1 City:", user1City); // Expected: Anytown
console.log("User 2 City:", user2City);
// Expected: N/A
// Optional chaining with function calls
// If user3.contact is undefined, it won't try to call .getEmail()
const getEmail = (usr) => usr.contact?.getEmail?.();
// The method getEmail might not exist
user3.contact.getEmail = () => "charlie_from_method@example.com";
console.log("User 3 Email (from method):", getEmail(user3));
// Expected: charlie_from_method@example.com
user3.contact.getEmail = undefined; // Remove the method
console.log("User 3 Email (method removed):", getEmail(user3));
// Expected: undefined (no error)
```

Exercise 41: Asynchronous JavaScript - async/await

```
// Exercise 41: Asynchronous JavaScript - async/await
// Reusing the Promise-based function from Exercise 27
function fetchUserDataPromise(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (userId === 0) {
        reject("User with ID 0 not found.");
        return;
      }
      const user = {
        id: userId,
```

```
      name: `Async User ${userId}`,
      status: "active"
    };
    resolve(user);
  }, 1500); // Shorter delay for quicker demonstration
 });
}
// New function using async/await
async function displayUserAsync(userId) {
 console.log(`\n(async/await) Attempting to fetch user ${userId}...`);
 try {
  const user = await fetchUserDataPromise(userId);
  // Pause execution until the promise resolves
  console.log(`(async/await) Successfully fetched user ${userId}:`, user);
 } catch (error) {
  console.error(`(async/await) Error fetching user ${userId}:`, error);
 }
}
// Demonstrate usage
displayUserAsync(101);
// Success case
displayUserAsync(0); // Error case
displayUserAsync(102); // Another success case
console.log("Main script continues to run while async functions are awaiting.");
```

Exercise 42: Classes and Inheritance

```
// Exercise 42: Classes and Inheritance
// Parent Class
class Shape {
 constructor(color) {
  this.color = color;
 }
 displayColor() {
  console.log(`The shape color is ${this.color}.`);
 }
}
// Child Class inheriting from Shape
```

```
class Circle extends Shape {
  constructor(color, radius) {
    super(color);
    // Call the parent class's constructor
    this.radius = radius;
  }
  // Override the displayColor method from the parent class
  displayColor() {
    console.log(`The circle color is ${this.color}.`);
  }
  // New method specific to Circle
  calculateArea() {
    return Math.PI * this.radius * this.radius;
  }
}
// Child Class inheriting from Shape (another example)
class Rectangle extends Shape {
  constructor(color, width, height) {
    super(color);
    this.width = width;
    this.height = height;
  }
  calculateArea() {
    return this.width * this.height;
  }
  // Overriding a method and calling the super method
  displayColor() {
    super.displayColor();
    // Call the parent's displayColor method
    console.log(`This is a rectangle.`);
  }
}
// Create instances
let genericShape = new Shape("Red");
genericShape.displayColor(); // Expected: The shape color is Red.
let myCircle = new Circle("Blue", 5);
myCircle.displayColor(); // Expected: The circle color is Blue. (Overridden method)
console.log("Circle Area:", myCircle.calculateArea().toFixed(2));
// Expected: ~78.54
```

```
let myRectangle = new Rectangle("Green", 4, 6);
myRectangle.displayColor(); // Expected: The shape color is Green.
console.log("Rectangle Area:", myRectangle.calculateArea()); // Expected: 24
```

Exercise 43: Static Methods and Properties

```
// Exercise 43: Static Methods and Properties
class MathHelper {
  // Static property
  static PI = 3.14159;
  // Static method: can be called directly on the class, not instances
  static add(a, b) {
    return a + b;
  }
  static multiply(a, b) {
    return a * b;
  }
  // Instance method (requires an instance of MathHelper)
  instanceMethod() {
    console.log("This is an instance method.");
  }
}
// Accessing static property
console.log("MathHelper.PI:", MathHelper.PI); // Expected: 3.14159
// Calling static methods
console.log("MathHelper.add(5, 3):", MathHelper.add(5, 3));
// Expected: 8
console.log("MathHelper.multiply(4, 6):", MathHelper.multiply(4, 6)); // Expected: 24
// Trying to call a static method on an instance will throw an error
// let myHelper = new MathHelper();
// myHelper.add(1, 2); // TypeError: myHelper.add is not a function
// Calling an instance method (requires an instance)
let myHelper = new MathHelper();
myHelper.instanceMethod();
```

Exercise 44: Getters and Setters

```
// Exercise 44: Getters and Setters
class Product {
  constructor(name, initialPrice) {
    this.name = name;
    this._price = 0; // Conventionally, _ prefix indicates a private/protected property
    this.price = initialPrice;
    // Use the setter to initialize with validation
  }
  // Getter for 'price'
  get price() {
    console.log(`Getting price for ${this.name}...`);
    return this._price;
  }
  // Setter for 'price'
  set price(newPrice) {
    console.log(`Attempting to set price for ${this.name} to ${newPrice}...`);
    if (typeof newPrice === 'number' && newPrice >= 0) {
      this._price = newPrice;
      console.log(`Price set successfully to ${newPrice}.`);
    } else {
      console.error(`Error: Invalid price value: ${newPrice}. Price must be a non-negative
number.`);
    }
  }
  displayDetails() {
    console.log(`${this.name} - $${this.price.toFixed(2)}`);
  }
}
let laptop = new Product("Laptop", 1200);
laptop.displayDetails(); // Getting price... Laptop - $1200.00
console.log("Laptop price is:", laptop.price);
// Accessing as a property (invokes getter)
laptop.price = 1250; // Setting price (invokes setter)
laptop.displayDetails();
// Getting price... Laptop - $1250.00
laptop.price = -50; // Invalid price (invokes setter, logs error)
laptop.displayDetails();
// Still Laptop - $1250.00 (price not changed due to validation)
laptop.price = "one thousand";
```

```
// Invalid type (invokes setter, logs error)
laptop.displayDetails(); // Still Laptop - $1250.00
```

Exercise 45: Array Method: some()

```
// Exercise 45: Array Method: some()
let grades = [60, 75, 80, 90, 55, 62];
// Check if at least one grade is >= 90
let hasExcellentGrade = grades.some(grade => grade >= 90);
console.log("Are there any excellent grades (>= 90)?", hasExcellentGrade); //
Expected: true
let grades2 = [50, 60, 70, 80];
let hasExcellentGrade2 = grades2.some(grade => grade >= 90);
console.log("Are there any excellent grades (>= 90) in grades2?",
hasExcellentGrade2);
// Expected: false
// Check if any product is out of stock (using objects)
let products = [
  { name: "Milk", inStock: true },
  { name: "Bread", inStock: false },
  { name: "Eggs", inStock: true }
];
let anyOutOfStock = products.some(product => !product.inStock);
console.log("Is any product out of stock?", anyOutOfStock);
// Expected: true
let allInStockProducts = [
  { name: "Apples", inStock: true },
  { name: "Bananas", inStock: true }
];
let anyOutOfStock2 = allInStockProducts.some(product => !product.inStock);
console.log("Is any product out of stock (all in stock)?", anyOutOfStock2);
// Expected: false
```

Exercise 46: Array Method: every()

```
// Exercise 46: Array Method: every()
let ages = [22, 28, 35, 40, 19];
```

```
// Check if all ages are >= 18
let allAdults = ages.every(age => age >= 18);
console.log("Are all ages >= 18?", allAdults); // Expected: true
let ages2 = [17, 20, 25];
let allAdults2 = ages2.every(age => age >= 18);
console.log("Are all ages >= 18 in ages2?", allAdults2);
// Expected: false (due to 17)
// Check if all tasks are completed
let tasks = [
  { id: 1, completed: true },
  { id: 2, completed: true },
  { id: 3, completed: false }
];
let allTasksCompleted = tasks.every(task => task.completed);
console.log("Are all tasks completed?", allTasksCompleted);
// Expected: false
let allDoneTasks = [
  { id: 1, completed: true },
  { id: 2, completed: true }
];
let allTasksCompleted2 = allDoneTasks.every(task => task.completed);
console.log("Are all tasks completed (all done)?", allTasksCompleted2);
// Expected: true
```

Exercise 47: Array Method: find() and findIndex()

```
// Exercise 47: Array Method: find() and findIndex()
let users = [
  { id: 1, name: "Alice", active: true },
  { id: 2, name: "Bob", active: false },
  { id: 3, name: "Charlie", active: true },
  { id: 4, name: "Alice", active: false } // Another Alice
];
// 1. Using find() to get the first matching element
let userWithId2 = users.find(user => user.id === 2);
console.log("User with ID 2:", userWithId2); // Expected: { id: 2, name: "Bob", active:
false }
let activeUser = users.find(user => user.active === true);
```

```
console.log("First active user:", activeUser); // Expected: { id: 1, name: "Alice", active:
true }
// 2. Using findIndex() to get the index of the first matching element
let charlieIndex = users.findIndex(user => user.name === "Charlie");
console.log("Index of Charlie:", charlieIndex); // Expected: 2
let firstAliceIndex = users.findIndex(user => user.name === "Alice");
console.log("Index of first Alice:", firstAliceIndex);
// Expected: 0
// 3. Finding non-existent elements/indices
let nonExistentUser = users.find(user => user.id === 99);
console.log("Non-existent user:", nonExistentUser);
// Expected: undefined
let nonExistentIndex = users.findIndex(user => user.name === "David");
console.log("Index of non-existent user:", nonExistentIndex);
// Expected: -1
```

Exercise 48: Set Data Structure

```
// Exercise 48: Set Data Structure
const uniqueNumbers = new Set();
uniqueNumbers.add(1);
uniqueNumbers.add(2);
uniqueNumbers.add(3);
uniqueNumbers.add(2); // Adding 2 again has no effect as Sets only store unique
values
uniqueNumbers.add(4);
uniqueNumbers.add(1);
// Adding 1 again has no effect
console.log("Set after adding elements:", uniqueNumbers);
// Expected: Set { 1, 2, 3, 4 }
// 1. Size of the Set
console.log("Size of Set:", uniqueNumbers.size);
// Expected: 4
// 2. Check for existence
console.log("Does Set contain 3?", uniqueNumbers.has(3));
// Expected: true
console.log("Does Set contain 5?", uniqueNumbers.has(5)); // Expected: false
// 3. Remove an element
```

```
uniqueNumbers.delete(2);
console.log("Set after deleting 2:", uniqueNumbers); // Expected: Set { 1, 3, 4 }
console.log("Does Set contain 2 after deletion?", uniqueNumbers.has(2));
// Expected: false
// 4. Iterate over the Set
console.log("Elements in Set:");
for (let num of uniqueNumbers) {
  console.log(num);
}
// Convert array with duplicates to array with unique elements using Set
let numbersWithDuplicates = [1, 5, 2, 8, 5, 1, 9, 2];
let uniqueArray = [...new Set(numbersWithDuplicates)]; // Convert to Set, then spread back to array
console.log("Unique array from duplicates:", uniqueArray);
// Expected: [1, 5, 2, 8, 9]
```

Exercise 49: Map Data Structure

```
// Exercise 49: Map Data Structure
const userRoles = new Map();
userRoles.set("Alice", "Admin");
userRoles.set("Bob", "Editor");
userRoles.set("Charlie", "Viewer");
console.log("Map after adding elements:", userRoles);
// Expected: Map { 'Alice' => 'Admin', 'Bob' => 'Editor', 'Charlie' => 'Viewer' }
// 1. Get a value by key
console.log("Role of Alice:", userRoles.get("Alice"));
// Expected: Admin
// 2. Check if a key exists
console.log("Does David exist?", userRoles.has("David"));
// Expected: false
console.log("Does Alice exist?", userRoles.has("Alice")); // Expected: true
// 3. Update a value
userRoles.set("Bob", "Moderator");
console.log("Updated role of Bob:", userRoles.get("Bob")); // Expected: Moderator
// 4. Delete a key-value pair
userRoles.delete("Charlie");
console.log("Map after deleting Charlie:", userRoles);
```

```
// Expected: Map { 'Alice' => 'Admin', 'Bob' => 'Moderator' }
// 5. Iterate over the Map
console.log("\nIterating Map (entries):");
for (let [name, role] of userRoles) { // Directly destructure key and value
  console.log(`${name}'s role is ${role}`);
}
console.log("\nIterating Map (keys):");
for (let name of userRoles.keys()) {
  console.log(`User: ${name}`);
}
console.log("\nIterating Map (values):");
for (let role of userRoles.values()) {
  console.log(`Role: ${role}`);
}
```

Exercise 50: localStorage (Basic Persistence)

```
// Exercise 50: localStorage (Basic Persistence)
// Check if localStorage is available (it usually is in browsers)
if (typeof localStorage !== 'undefined') {
  console.log("localStorage is available.");
  // 1. Store a string
  localStorage.setItem("myUserName", "Sarah");
  console.log("Stored 'Sarah' in localStorage under 'myUserName'.");
  // 2. Retrieve a string
  let storedUserName = localStorage.getItem("myUserName");
  console.log("Retrieved 'myUserName':", storedUserName);
  // Expected: Sarah
  // 3. Store an object (must be stringified)
  let settingsObject = {
    theme: "dark",
    notifications: true,
    fontSize: 16
  };
  localStorage.setItem("userSettings", JSON.stringify(settingsObject));
  console.log("Stored settings object (stringified) under 'userSettings'.");
  // 4. Retrieve and parse the object
  let storedSettingsString = localStorage.getItem("userSettings");
```

```
  if (storedSettingsString) {
    let parsedSettings = JSON.parse(storedSettingsString);
    console.log("Retrieved and parsed 'userSettings':", parsedSettings);
    // Expected: { theme: 'dark', notifications: true, fontSize: 16 }
    console.log("Theme from settings:", parsedSettings.theme);
  } else {
    console.log("No 'userSettings' found in localStorage.");
  }
  // 5. Remove an item
  localStorage.removeItem("myUserName");
  console.log("Removed 'myUserName' from localStorage.");
  console.log("Attempting to retrieve 'myUserName' after removal:",
localStorage.getItem("myUserName")); // Expected: null
  // You can also clear all items (use with caution!)
  // localStorage.clear();
  // console.log("All localStorage cleared.");
} else {
  console.warn("localStorage is not available in this environment.");
}
```

Exercise 51: Palindrome Checker

```
// Exercise 51: Palindrome Checker
function isPalindrome(str) {
  // Step 1: Clean the string - convert to lowercase and remove non-alphanumeric
characters
  const cleanedStr = str.toLowerCase().replace(/[^a-z0-9]/g, '');
  // Step 2: Reverse the cleaned string
  const reversedStr = cleanedStr.split('').reverse().join('');
  // Step 3: Compare the cleaned string with its reversed version
  return cleanedStr === reversedStr;
}
console.log("'racecar' is a palindrome:", isPalindrome("racecar")); // Expected: true
console.log("'hello' is a palindrome:", isPalindrome("hello"));
// Expected: false
console.log("'Madam' is a palindrome:", isPalindrome("Madam")); // Expected: true
(case-insensitive)
console.log("'A man, a plan, a canal: Panama' is a palindrome:", isPalindrome("A man,
```

```
a plan, a canal: Panama"));
// Expected: true (ignores non-alphanumeric)
console.log("'' is a palindrome:", isPalindrome(""));
// Expected: true (empty string is a palindrome)
console.log("'A' is a palindrome:", isPalindrome("A"));
// Expected: true (single character is a palindrome)
```

Exercise 52: Anagram Checker

```
// Exercise 52: Anagram Checker
function cleanAndSortString(str) {
  return str
    .toLowerCase() // Convert to lowercase
    .replace(/[^a-z0-9]/g, '') // Remove non-alphanumeric characters
    .split('') // Split into an array of characters
    .sort() // Sort the characters alphabetically
    .join(''); // Join back into a string
}
function areAnagrams(str1, str2) {
  // Anagrams must have the same length after cleaning
  if (str1.length !== str2.length) {
    return false;
  }
  return cleanAndSortString(str1) === cleanAndSortString(str2);
}
console.log("'listen' and 'silent' are anagrams:", areAnagrams("listen", "silent"));
// Expected: true
console.log("'Debit Card' and 'Bad Credit' are anagrams:", areAnagrams("Debit Card",
"Bad Credit"));
// Expected: true (ignores case and spaces)
console.log("'hello' and 'world' are anagrams:", areAnagrams("hello", "world"));
// Expected: false
console.log("'Anagram' and 'Nag A Ram' are anagrams:", areAnagrams("Anagram",
"Nag A Ram"));
// Expected: true
console.log("'' and '' are anagrams:", areAnagrams("", "")); // Expected: true
console.log("'a' and 'b' are anagrams:", areAnagrams("a", "b"));
// Expected: false
```

Exercise 53: FizzBuzz

```
// Exercise 53: FizzBuzz
function fizzBuzz(countTo) {
  console.log(`FizzBuzz up to ${countTo}:`);
  for (let i = 1; i <= countTo; i++) {
    let output = "";
    if (i % 3 === 0) { // Check if divisible by 3
      output += "Fizz";
    }
    if (i % 5 === 0) { // Check if divisible by 5
      output += "Buzz";
    }
    // If output is empty, it means it's not divisible by 3 or 5
    console.log(output || i);
    // Use || to print number if output is empty
  }
}
fizzBuzz(15);
// Will print up to 15 to demonstrate
```

Exercise 54: Remove Duplicates from an Array

```
// Exercise 54: Remove Duplicates from an Array
function removeDuplicates(arr) {
  // The most concise and often preferred way using Set
  return [...new Set(arr)];
}
// Alternative using filter and indexOf (less efficient for large arrays)
function removeDuplicatesLegacy(arr) {
  return arr.filter((item, index) => arr.indexOf(item) === index);
}
console.log("Remove duplicates from [1, 2, 2, 3, 4, 4, 5]:", removeDuplicates([1, 2, 2, 3,
4, 4, 5]));
// Expected: [1, 2, 3, 4, 5]
console.log("Remove duplicates from ['apple', 'banana', 'apple', 'orange']:",
```

```
removeDuplicates(['apple', 'banana', 'apple', 'orange']));
// Expected: ['apple', 'banana', 'orange']
console.log("Remove duplicates from [1, '1', 2, 1]:", removeDuplicates([1, '1', 2, 1]));
// Expected: [1, '1', 2] (Set distinguishes types)
console.log("Remove duplicates from []:", removeDuplicates([]));
// Expected: []
console.log("Remove duplicates from ['a', 'b', 'c']:", removeDuplicates(['a', 'b', 'c'])); //
Expected: ['a', 'b', 'c']
console.log("\nUsing legacy method:");
console.log("Remove duplicates from [1, 2, 2, 3, 4, 4, 5]:", removeDuplicatesLegacy([1,
2, 2, 3, 4, 4, 5]));
```

Exercise 55: Count Character Occurrences

```
// Exercise 55: Count Character Occurrences
function countChars(str) {
  const charCounts = {};
  // Initialize an empty object to store counts
  const cleanedStr = str.toLowerCase();
  // Convert to lowercase for case-insensitivity
  for (let i = 0; i < cleanedStr.length; i++) {
    const char = cleanedStr[i];
    // Only count alphanumeric characters (optional, but good practice for practical
use)
    if (/[a-z0-9]/.test(char)) {
      // If the character is already a key in charCounts, increment its value
      // Otherwise, add it as a new key with value 1
      charCounts[char] = (charCounts[char] || 0) + 1;
    }
  }
  return charCounts;
}
console.log("Counts for 'hello world':", countChars("hello world"));
// Expected: { h: 1, e: 1, l: 3, o: 2, w: 1, r: 1, d: 1 } (excluding space)
console.log("Counts for 'Programming is fun':", countChars("Programming is fun"));
// Expected: { p: 1, r: 2, o: 2, g: 2, a: 1, m: 2, i: 2, n: 2, s: 1, f: 1, u: 1 } (excluding space)
console.log("Counts for 'AAAaaa':", countChars("AAAaaa"));
// Expected: { a: 6 }
```

```
console.log("Counts for '123123':", countChars("123123"));
// Expected: { '1': 2, '2': 2, '3': 2 }
```

Exercise 56: Merge Two Sorted Arrays

```
// Exercise 56: Merge Two Sorted Arrays
function mergeSortedArrays(arr1, arr2) {
  const merged = [];
  let ptr1 = 0; // Pointer for arr1
  let ptr2 = 0;
  // Pointer for arr2
  // Compare elements from both arrays and add the smaller one to merged
  while (ptr1 < arr1.length && ptr2 < arr2.length) {
    if (arr1[ptr1] < arr2[ptr2]) {
      merged.push(arr1[ptr1]);
      ptr1++;
    } else {
      merged.push(arr2[ptr2]);
      ptr2++;
    }
  }
  // Add any remaining elements from arr1 (if any)
  while (ptr1 < arr1.length) {
    merged.push(arr1[ptr1]);
    ptr1++;
  }
  // Add any remaining elements from arr2 (if any)
  while (ptr2 < arr2.length) {
    merged.push(arr2[ptr2]);
    ptr2++;
  }
  return merged;
}
console.log("Merge [1, 3, 5] and [2, 4, 6]:", mergeSortedArrays([1, 3, 5], [2, 4, 6]));
// Expected: [1, 2, 3, 4, 5, 6]
console.log("Merge [10, 20] and [5, 15, 25]:", mergeSortedArrays([10, 20], [5, 15, 25]));
// Expected: [5, 10, 15, 20, 25]
console.log("Merge [1, 2] and []:", mergeSortedArrays([1, 2], []));
```

```
// Expected: [1, 2]
console.log("Merge [] and [7, 8]:", mergeSortedArrays([], [7, 8]));
// Expected: [7, 8]
```

Exercise 57: Find Missing Number in a Sequence

```
// Exercise 57: Find Missing Number in a Sequence
function findMissingNumber(arr) {
  const n = arr.length + 1;
  // If one number is missing, n is (array length + 1)
  // Calculate the expected sum of numbers from 1 to n
  // Formula for sum of an arithmetic series: n * (n + 1) / 2
  const expectedSum = n * (n + 1) / 2;
  // Calculate the actual sum of numbers in the given array
  let actualSum = 0;
  for (let i = 0; i < arr.length; i++) {
    actualSum += arr[i];
  }
  // Or using reduce: const actualSum = arr.reduce((sum, num) => sum + num, 0);
  // The missing number is the difference between the expected and actual sums
  return expectedSum - actualSum;
}
console.log("Missing number in [1, 2, 4, 5]:", findMissingNumber([1, 2, 4, 5]));
// Expected: 3
console.log("Missing number in [1, 3]:", findMissingNumber([1, 3]));
// Expected: 2 (n=3, sum=6, actual=4)
console.log("Missing number in [2, 1, 4, 5, 6, 3, 8]:", findMissingNumber([2, 1, 4, 5, 6, 3,
8]));
// Expected: 7 (n=8, sum=36, actual=29)
console.log("Missing number in [1]:", findMissingNumber([1]));
// Expected: undefined if array is empty (needs error handling)
console.log("Missing number in []:", findMissingNumber([]));
// Expected: 1 (n=1, sum=1, actual=0)
```

Exercise 58: Flatten an Array

```
// Exercise 58: Flatten an Array
```

```
function flattenArray(arr) {
  let flatArr = [];
  for (let i = 0; i < arr.length; i++) {
    if (Array.isArray(arr[i])) {
      // If the element is an array, recursively flatten it and concatenate
      flatArr = flatArr.concat(flattenArray(arr[i]));
    } else {
      // If it's not an array, just push it
      flatArr.push(arr[i]);
    }
  }
  return flatArr;
}
// Modern ES2019+ built-in method for comparison
function flattenArrayBuiltIn(arr, depth = 1) {
  return arr.flat(depth);
  // flat() can take a depth argument, or Infinity
}
const nestedArray1 = [1, [2, 3], 4];
console.log("Flatten [1, [2, 3], 4]:", flattenArray(nestedArray1)); // Expected: [1, 2, 3, 4]
const nestedArray2 = [1, [2, [3, 4]], 5, [6]];
console.log("Flatten [1, [2, [3, 4]], 5, [6]]:", flattenArray(nestedArray2)); // Expected: [1,
2, 3, 4, 5, 6]
const nestedArray3 = [[1, 2], [3, [4, 5]]];
console.log("Flatten [[1, 2], [3, [4, 5]]]:", flattenArray(nestedArray3)); // Expected: [1, 2,
3, 4, 5]
console.log("\nUsing built-in flat() method:");
console.log("Flatten [1, [2, 3], 4]:", flattenArrayBuiltIn(nestedArray1));
console.log("Flatten [1, [2, [3, 4]], 5, [6]] (depth 1):", flattenArrayBuiltIn(nestedArray2,
1));
console.log("Flatten [1, [2, [3, 4]], 5, [6]] (depth Infinity):",
flattenArrayBuiltIn(nestedArray2, Infinity));
```

Exercise 59: Implement a Simple Queue

```
// Exercise 59: Implement a Simple Queue
class Queue {
  constructor() {
```

```
  this.items = [];
  // The array to store queue elements
}
// Add an element to the back of the queue
enqueue(element) {
  this.items.push(element);
  console.log(`Enqueued: ${element}. Queue: [${this.items.join(', ')}]`);
}
// Remove and return the element from the front of the queue
dequeue() {
  if (this.isEmpty()) {
    console.log("Queue is empty, cannot dequeue.");
    return undefined;
  }
  const removedElement = this.items.shift();
  // Removes from the beginning
  console.log(`Dequeued: ${removedElement}. Queue: [${this.items.join(', ')}]`);
  return removedElement;
}
// Return the element at the front of the queue without removing it
peek() {
  if (this.isEmpty()) {
    console.log("Queue is empty, nothing to peek.");
    return undefined;
  }
  const frontElement = this.items[0];
  console.log(`Peeked: ${frontElement}.`);
  return frontElement;
}
// Check if the queue is empty
isEmpty() {
  return this.items.length === 0;
}
// Get the number of elements in the queue
size() {
  return this.items.length;
}
// For debugging/display
printQueue() {
```

```
    console.log(`Current Queue: [${this.items.join(', ')}] (Size: ${this.size()})`);
  }
}
// Demonstrate Queue usage
const myQueue = new Queue();
console.log("Is queue empty?", myQueue.isEmpty()); // Expected: true
myQueue.enqueue("Task 1");
myQueue.enqueue("Task 2");
myQueue.enqueue("Task 3");
myQueue.printQueue();
console.log("Queue size:", myQueue.size()); // Expected: 3
myQueue.peek(); // Expected: Peeked: Task 1.
myQueue.dequeue();
// Expected: Dequeued: Task 1. Queue: [Task 2, Task 3]
myQueue.peek(); // Expected: Peeked: Task 2.
myQueue.dequeue();
// Expected: Dequeued: Task 2. Queue: [Task 3]
myQueue.dequeue(); // Expected: Dequeued: Task 3. Queue: []
myQueue.dequeue();
// Expected: Queue is empty, cannot dequeue.
console.log("Is queue empty?", myQueue.isEmpty());
// Expected: true
```

Exercise 60: Implement a Simple Stack

```
// Exercise 60: Implement a Simple Stack
class Stack {
  constructor() {
    this.items = [];
    // The array to store stack elements
  }
  // Add an element to the top of the stack
  push(element) {
    this.items.push(element);
    console.log(`Pushed: ${element}. Stack: [${this.items.join(', ')}]`);
  }
  // Remove and return the element from the top of the stack
  pop() {
```

```
    if (this.isEmpty()) {
      console.log("Stack is empty, cannot pop.");
      return undefined;
    }
    const poppedElement = this.items.pop();
    // Removes from the end
    console.log(`Popped: ${poppedElement}. Stack: [${this.items.join(', ')}]`);
    return poppedElement;
  }
  // Return the element at the top of the stack without removing it
  peek() {
    if (this.isEmpty()) {
      console.log("Stack is empty, nothing to peek.");
      return undefined;
    }
    const topElement = this.items[this.items.length - 1];
    console.log(`Peeked: ${topElement}.`);
    return topElement;
  }
  // Check if the stack is empty
  isEmpty() {
    return this.items.length === 0;
  }
  // Get the number of elements in the stack
  size() {
    return this.items.length;
  }
  // For debugging/display
  printStack() {
    console.log(`Current Stack: [${this.items.join(', ')}] (Size: ${this.size()})`);
  }
}
// Demonstrate Stack usage
const myStack = new Stack();
console.log("Is stack empty?", myStack.isEmpty()); // Expected: true
myStack.push("Page 1");
myStack.push("Page 2");
myStack.push("Page 3");
myStack.printStack();
```

```
console.log("Stack size:", myStack.size()); // Expected: 3
myStack.peek(); // Expected: Peeked: Page 3.
myStack.pop();
// Expected: Popped: Page 3. Stack: [Page 1, Page 2]
myStack.peek(); // Expected: Peeked: Page 2.
myStack.pop();
// Expected: Popped: Page 2. Stack: [Page 1]
myStack.pop(); // Expected: Popped: Page 1. Stack: []
myStack.pop();
// Expected: Stack is empty, cannot pop.
console.log("Is stack empty?", myStack.isEmpty());
// Expected: true
```

Exercise 61: Promise.all()

```
// Exercise 61: Promise.all()
function fetchData(name, delay) {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`Finished fetching ${name} after ${delay}ms`);
      resolve(`Data from ${name}`);
    }, delay);
  });
}
console.log("Starting all data fetches...");
// Create an array of Promises
const allPromises = [
  fetchData("Service A", 2000), // Longest delay
  fetchData("Service B", 1000),
  fetchData("Service C", 1500)
];
// Use Promise.all to wait for all promises to resolve
Promise.all(allPromises)
  .then(results => {
    console.log("\nAll data received:");
    console.log(results); // Expected: ["Data from Service A", "Data from Service B",
"Data from Service C"]
  })
```

```
  .catch(error => {
    console.error("One of the fetches failed:", error);
  });
console.log("Meanwhile, other tasks can run in the main thread.");
```

Exercise 62: Basic Event Loop Understanding with setTimeout

```
// Exercise 62: Basic Event Loop Understanding with setTimeout
console.log("1. Start of script.");
setTimeout(() => {
  console.log("3. Inside setTimeout callback (0ms delay).");
}, 0);
// Scheduled to run as soon as possible after call stack is clear
setTimeout(() => {
  console.log("4. Inside setTimeout callback (100ms delay).");
}, 100);
// Scheduled to run after at least 100ms
console.log("2. End of script.");
// This loop simulates a long-running synchronous task
// It will block the main thread and prevent the 0ms timeout from executing
immediately
// even though its delay is 0ms.
let sum = 0;
for (let i = 0; i < 1000000000; i++) {
  sum += i;
}
console.log("5. Long synchronous task finished. Sum:", sum); // This will appear before
any timeouts
```

Exercise 63: Error Handling in async/await with try...catch

```
// Exercise 63: Error Handling in async/await with try...catch
// Reusing the Promise-based function from Exercise 27/41
function fetchUserDataPromise(userId) {
  return new Promise((resolve, reject) => {
```

```
    setTimeout(() => {
      if (userId === 0 || userId < 0) { // Simulate error for 0 or negative IDs
        reject(`Error: User with ID ${userId} not found or invalid.`);
        return;
      }
      const user = {
        id: userId,
        name: `User ${userId}`,
        status: "active"
      };
      resolve(user);
    }, 800); // Shorter delay for demonstration
  });
}
async function processUserData(userId) {
  console.log(`\nAttempting to process data for user ID: ${userId}...`);
  try {
    const user = await fetchUserDataPromise(userId);
    // Await the promise
    console.log(`Success! Data for user ${userId}:`, user);
  } catch (error) {
    // If fetchUserDataPromise rejects, the error is caught here
    console.error(`Failed to process data for user ID ${userId}:`, error);
  } finally {
    console.log(`Finished processing attempt for user ID: ${userId}.`);
  }
}
// Demonstrate usage:
processUserData(10);
// Success case
processUserData(0); // Error case
processUserData(15); // Another success case
processUserData(-1);
// Another error case
```

Exercise 64: Generators (Basic)

```
// Exercise 64: Generators (Basic)
```

```
function* idGenerator() {
  let id = 1;
  while (true) { // This loop will run indefinitely until the generator is stopped
    yield id++;
    // Pause execution and return the current 'id', then increment it for the next call
  }
}
// Create a generator object
const myIdGenerator = idGenerator();
console.log("Generated IDs:");
console.log(myIdGenerator.next().value); // Expected: 1
console.log(myIdGenerator.next().value); // Expected: 2
console.log(myIdGenerator.next().value);
// Expected: 3
// You can create another independent generator
const anotherIdGenerator = idGenerator();
console.log(anotherIdGenerator.next().value);
// Expected: 1 (starts fresh)
console.log(myIdGenerator.next().value); // Expected: 4 (myIdGenerator continues)
```

Exercise 65: Iterators (Basic Custom)

```
// Exercise 65: Iterators (Basic Custom)
class MyRange {
  constructor(from, to) {
    this.from = from;
    this.to = to;
  }
  // This method makes the object "iterable"
  [Symbol.iterator]() {
    let current = this.from;
    // Keep track of the current number
    // The iterator object must have a 'next' method
    return {
      next: () => {
        if (current <= this.to) {
          // If there are more numbers, return the current one and advance
          return { done: false, value: current++ };
```

```
      } else {
        // If the range is exhausted, signal completion
        return { done: true };
      }
    }
  };
 }
}
console.log("Numbers in range 1 to 5:");
for (let num of new MyRange(1, 5)) {
 console.log(num);
 // Expected: 1, 2, 3, 4, 5
}
console.log("\nNumbers in range 7 to 10:");
for (let num of new MyRange(7, 10)) {
 console.log(num);
 // Expected: 7, 8, 9, 10
}
```

Exercise 66: JavaScript Modules (Basic import/export)

```
// Exercise 66: JavaScript Modules (Basic import/export)
// --- content of utils.js ---
// export function capitalize(str) {
//   if (!str) return '';
//   return str.charAt(0).toUpperCase() + str.slice(1).toLowerCase();
// }
// export const APP_NAME = "My Awesome App";
// export const PI_VALUE = 3.14159;
//
// Default export (only one per module)
// export default class Greeter {
//   constructor(name) {
//     this.name = name;
//   }
//   sayHello() {
//     console.log(`Hello, ${this.name}!`);
//   }
```

```
// }
// --- content of main.js ---
// import { capitalize, APP_NAME, PI_VALUE } from './utils.js';
// Named imports
// import MyGreeter from './utils.js'; // Default import (any name)
// console.log(`Application Name: ${APP_NAME}`);
// console.log(`Capitalized "hello": ${capitalize("hello")}`);
// console.log(`Value of PI: ${PI_VALUE}`);
// const greeterInstance = new MyGreeter("Module User");
// greeterInstance.sayHello();
console.log("This exercise describes modules conceptually.");
console.log("See the commented-out code for example `utils.js` and `main.js` file
structures.");
console.log("\nTo run this in a browser, you'd use a script tag like:");
console.log('<script type="module" src="main.js"></script>');
console.log("\nEach module runs in strict mode and has its own top-level scope.");
```

Exercise 67: Simple Event Emitter (Custom Implementation)

```
// Exercise 67: Simple Event Emitter (Custom Implementation)
class EventEmitter {
  constructor() {
    this.listeners = new Map();
    // Stores event names as keys and arrays of listener functions as values
  }
  // Register a listener for a specific event
  on(eventName, listener) {
    if (!this.listeners.has(eventName)) {
      this.listeners.set(eventName, []);
      // If event doesn't exist, create an empty array for its listeners
    }
    this.listeners.get(eventName).push(listener);
    // Add the listener to the array
    console.log(`Registered listener for '${eventName}'.`);
  }
  // Emit an event, calling all registered listeners
  emit(eventName, ...args) { // ...args allows passing any number of arguments to
```

```
listeners
    const eventListeners = this.listeners.get(eventName);
    if (eventListeners) {
      console.log(`Emitting event '${eventName}' with args: ${JSON.stringify(args)}`);
      eventListeners.forEach(listener => {
        try {
          listener(...args); // Call each listener with the provided arguments
        } catch (e) {
          console.error(`Error in listener for '${eventName}':`, e);
        }
      });
    } else {
      console.log(`No listeners registered for '${eventName}'.`);
    }
  }
  // Remove a specific listener for an event
  off(eventName, listenerToRemove) {
    const eventListeners = this.listeners.get(eventName);
    if (eventListeners) {
      const index = eventListeners.indexOf(listenerToRemove);
      if (index > -1) {
        eventListeners.splice(index, 1);
        // Remove the listener from the array
        console.log(`Unregistered listener for '${eventName}'.`);
      } else {
        console.log(`Listener not found for '${eventName}'.`);
      }
    } else {
      console.log(`No listeners registered for '${eventName}'.`);
    }
  }
}
// Demonstrate usage
const myEmitter = new EventEmitter();
// Define some listener functions
const greetListener = (name) => console.log(`Hello, ${name}!`);
const logDataListener = (data) => console.log(`Received data: ${data}`);
const celebrateListener = (count, message) => console.log(`🎉 Celebrating ${count}
times: ${message}`); // Register listeners
```

```
myEmitter.on("greet", greetListener);
myEmitter.on("dataLoaded", logDataListener);
myEmitter.on("celebrate", celebrateListener);
myEmitter.on("greet", (name) => console.log(`A secondary greeting to ${name}.`)); // Multiple listeners for same event
// Emit events
myEmitter.emit("greet", "Alice");
myEmitter.emit("dataLoaded", { id: 101, status: "completed" });
myEmitter.emit("celebrate", 3, "New Milestone!");
myEmitter.emit("unknownEvent", "This won't do anything.");
// No listeners for this
// Unregister a listener
myEmitter.off("greet", greetListener);
myEmitter.emit("greet", "Bob");
// greetListener won't be called now
myEmitter.off("greet", greetListener); // Trying to remove again (should log "Listener not found")
```

Exercise 68: WeakSet Data Structure

```
// Exercise 68: WeakSet Data Structure
// WeakSet can only store objects (not primitive values)
const weakSet = new WeakSet();
let user1 = { id: 1, name: "Alice" };
let user2 = { id: 2, name: "Bob" };
let user3 = { id: 3, name: "Charlie" }; // Add objects to the WeakSet
weakSet.add(user1);
weakSet.add(user2);
weakSet.add(user3);
console.log("WeakSet after adding objects.");
console.log("WeakSet has user1:", weakSet.has(user1)); // Expected: true
console.log("WeakSet has user2:", weakSet.has(user2));
// Expected: true
console.log("WeakSet has user4 (non-existent):", weakSet.has({ id: 4 }));
// Expected: false (new object)
// Delete an object from WeakSet
weakSet.delete(user2);
console.log("WeakSet has user2 after deletion:", weakSet.has(user2));
```

```
// Expected: false
// What happens if an object is no longer referenced elsewhere?
// Let's remove the strong reference to user3
user3 = null;
// user3 is now eligible for garbage collection
// The WeakSet will automatically remove user3 if it's garbage collected.
// However, we cannot directly observe this or iterate the WeakSet to prove it.
// The `has` method might still return true for a short while if GC hasn't run.
// There is no .size property or iteration methods on WeakSet.
// A common use case: keeping track of objects that have certain permissions or states,
// without preventing them from being garbage collected if they are no longer used elsewhere.
class Permissions {
  constructor() {
    this.adminUsers = new WeakSet();
  }
  grantAdmin(userObj) {
    this.adminUsers.add(userObj);
  }
  isAdmin(userObj) {
    return this.adminUsers.has(userObj);
  }
}
const permSystem = new Permissions();
let currentUser = { id: 10, name: "AdminUser" };
permSystem.grantAdmin(currentUser);
console.log("Is currentUser an admin?", permSystem.isAdmin(currentUser));
// Expected: true
currentUser = null; // AdminUser object becomes eligible for GC
// At some point later, it will be automatically removed from permSystem.adminUsers
// without us having to explicitly delete it.
```

Exercise 69: WeakMap Data Structure

```
// Exercise 69: WeakMap Data Structure
// WeakMap can only use objects as keys (not primitive values)
const weakMap = new WeakMap();
```

```
let obj1 = { name: "Config A" };
let obj2 = { name: "User Session" };
let obj3 = { name: "DOM Element Ref" }; // Set key-value pairs
weakMap.set(obj1, { version: 1.0, active: true });
weakMap.set(obj2, { sessionId: "xyz123", lastAccess: new Date() });
weakMap.set(obj3, "This is data for the DOM element");
console.log("WeakMap after setting entries.");
console.log("Data for obj1:", weakMap.get(obj1)); // Expected: { version: 1, active: true }
console.log("WeakMap has obj2:", weakMap.has(obj2));
// Expected: true
console.log("Data for obj3:", weakMap.get(obj3)); // Expected: This is data for the
DOM element
// Delete an entry
weakMap.delete(obj1);
console.log("Data for obj1 after deletion:", weakMap.get(obj1)); // Expected: undefined
// What happens if a key object is no longer referenced elsewhere?
obj2 = null; // obj2 (the key) is now eligible for garbage collection
// If obj2 is garbage collected, its entry in weakMap will also be automatically removed.
// Like WeakSet, WeakMap has no .size property and cannot be iterated.
// We cannot directly observe the entry's removal until GC runs.
// A common use case: associating private data with objects
const privateData = new WeakMap();
class User {
  constructor(name, initialPrivateInfo) {
    this.name = name;
    privateData.set(this, initialPrivateInfo);
    // Store private data in WeakMap using 'this' as key
  }
  getPrivateInfo() {
    return privateData.get(this);
  }
  updatePrivateInfo(newInfo) {
    privateData.set(this, newInfo);
  }
}
let user = new User("Jane Doe", { secretId: "abc", role: "admin" });
console.log("User private info:", user.getPrivateInfo());
user.updatePrivateInfo({ secretId: "def", role: "guest" });
console.log("User updated private info:", user.getPrivateInfo());
```

```
user = null;
// User object is now eligible for GC, and its associated private data in WeakMap will
also be removed.
```

Exercise 70: Set for Counting Unique Elements (Advanced)

```
// Exercise 70: Set for Counting Unique Elements (Advanced)
function countUniqueElements(arr) {
  const uniqueItems = new Set(arr);
  // Create a Set from the array, automatically handling uniqueness
  return uniqueItems.size;
  // The size property of the Set gives the count of unique elements
}
console.log("Unique count in [1, 2, 2, 3, 4, 4, 5]:", countUniqueElements([1, 2, 2, 3, 4, 4,
5]));
// Expected: 5
console.log("Unique count in ['apple', 'banana', 'apple', 'orange']:",
countUniqueElements(['apple', 'banana', 'apple', 'orange']));
// Expected: 3
console.log("Unique count in []:", countUniqueElements([])); // Expected: 0
console.log("Unique count in [1, 1, 1, 1]:", countUniqueElements([1, 1, 1, 1]));
// Expected: 1
console.log("Unique count in [1, '1', 2]:", countUniqueElements([1, '1', 2]));
// Expected: 3 (Set distinguishes number 1 from string '1')
```

Exercise 71: Selecting Elements by ID

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exercise 71</title>
</head>
<body>
    <p id="myParagraph">This is the original paragraph text.</p>
```

```
  <script>
    document.addEventListener('DOMContentLoaded', () => {
      // 1. Select the paragraph element by its ID
      const paragraph = document.getElementById('myParagraph');
      // 2. Check if the element exists before trying to modify it
      if (paragraph) {
        // 3. Change its text content
        paragraph.textContent = "Hello from JavaScript! The ID selector worked.";
        console.log("Paragraph text changed successfully!");
      } else {
        console.error("Element with ID 'myParagraph' not found!");
      }
    });
    console.log("Run this code in an HTML file to see the DOM manipulation.");
  </script>
</body>
</html>
```

Exercise 72: Selecting Elements by Class Name

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exercise 72</title>
</head>
<body>
  <ul>
    <li class="listItem">First Original Item</li>
    <li class="listItem">Second Original Item</li>
    <li class="listItem">Third Original Item</li>
  </ul>

  <script>
    document.addEventListener('DOMContentLoaded', () => {
      // 1. Select all elements with the class 'listItem'
```

```
        const listItems = document.getElementsByClassName('listItem');
        console.log(`Found ${listItems.length} elements with class 'listItem'.`);
        // 2. Iterate through the HTMLCollection (which is array-like, not a true array)
        for (let i = 0; i < listItems.length; i++) {
            listItems[i].textContent = `Item ${i + 1} - Updated by JS`;
        }
        // Alternative for iteration (converting to Array):
        // Array.from(listItems).forEach((item, index) => {
        //   item.textContent = `Item ${index + 1} - Updated by JS (using forEach)`;
        // });
        console.log("List item texts changed successfully!");
    });
    console.log("Run this code in an HTML file to see the DOM manipulation.");
  </script>
</body>
</html>
```

Exercise 73: Selecting Elements with querySelector and querySelectorAll

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exercise 73</title>
  <style>
    #container {
      border: 1px solid #ccc;
      padding: 15px;
      margin: 20px;
      border-radius: 8px;
      background-color: #f9f9f9;
    }
    p {
      margin-bottom: 10px;
    }
  </style>
```

```
</head>
<body>
  <div id="container">
    <h3>Container Content</h3>
    <p class="intro">This is the introductory paragraph within the container.</p>
    <p>First general paragraph.</p>
    <p>Second general paragraph.</p>
    <span class="intro">This span also has intro class, but shouldn't be selected by
'p.intro'</span>
  </div>

  <script>
    document.addEventListener('DOMContentLoaded', () => {
      // 1. Select the first element that matches the CSS selector
      const introParagraph = document.querySelector('#container .intro');
      if (introParagraph) {
        introParagraph.textContent = "The intro paragraph has been updated!";
        introParagraph.style.color = "blue";
        console.log("Intro paragraph updated using querySelector.");
      } else {
        console.error("Intro paragraph not found!");
      }
      // 2. Select all elements that match the CSS selector
      const allParagraphs = document.querySelectorAll('#container p');
      console.log(`Found ${allParagraphs.length} paragraphs inside container.`);
      // 3. Iterate through the NodeList (which behaves like an Array for iteration)
      allParagraphs.forEach((p, index) => {
        p.style.backgroundColor = (index % 2 === 0) ? '#e0ffe0' : '#f0fff0'; //
Alternate background
        p.style.padding = '5px';
        p.style.borderRadius = '5px';
        p.style.marginBottom = '5px';
        p.style.border = '1px solid #ccc';
      });
      console.log("All paragraphs inside container styled using querySelectorAll.");
    });
    console.log("Run this code in an HTML file to see the DOM manipulation.");
  </script>
</body>
```

```
</html>
```

Exercise 74: Modifying Element Attributes

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exercise 74</title>
</head>
<body>
    <img id="myImage"
src="https://placehold.co/100x100/FF0000/FFFFFF?text=Original" alt="An original
placeholder image."
    style="border: 2px solid grey; margin: 10px;">

    <script>
        document.addEventListener('DOMContentLoaded', () => {
            const myImage = document.getElementById('myImage');
            if (myImage) {
                console.log("Original Image Src:", myImage.getAttribute('src'));
                console.log("Original Image Alt:", myImage.getAttribute('alt'));
                // 1. Change its src attribute
                myImage.setAttribute('src',
'https://placehold.co/150x150/0000FF/FFFFFF?text=New+Image');
                console.log("Image src changed.");
                // 2. Change its alt attribute
                myImage.setAttribute('alt', 'A descriptive new image for the placeholder.');
                console.log("Image alt changed.");
                // 3. Add a title attribute
                myImage.setAttribute('title', 'Hover to see title! Click me!');
                console.log("Image title added.");
                // 4. Remove the alt attribute after 2 seconds
                setTimeout(() => {
                    myImage.removeAttribute('alt');
                    console.log("Image alt attribute removed after 2 seconds.");
                    console.log("Current Image Alt (after removal):",
```

```
myImage.getAttribute('alt'));
                // Will be null
            }, 2000);
        } else {
            console.error("Element with ID 'myImage' not found!");
        }
    });
    console.log("Run this code in an HTML file to see the DOM manipulation.");
  </script>
</body>
</html>
```

Exercise 75: Adding and Removing CSS Classes

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exercise 75</title>
  <style>
    #myBox {
      width: 150px;
      height: 150px;
      background-color: lightgrey;
      margin: 20px;
      display: flex;
      justify-content: center;
      align-items: center;
      font-size: 1.2em;
      transition: background-color 0.3s ease, border 0.3s ease;
      border: 2px solid transparent;
      border-radius: 8px;
    }
    .highlight {
      background-color: #ffd700; /* Gold */
      border-color: #da0037;    /* Dark Red */
      box-shadow: 0 0 10px rgba(255, 215, 0, 0.5);
```

```
        }
        /* Basic styling for buttons (optional, can use Tailwind/Bootstrap) */
        button {
            padding: 8px 15px;
            margin: 5px;
            border: none;
            border-radius: 5px;
            cursor: pointer;
            color: white;
            font-weight: bold;
        }
        #addHighlightBtn { background-color: #4CAF50; } /* Green */
        #removeHighlightBtn { background-color: #f44336; } /* Red */
        #toggleHighlightBtn { background-color: #008CBA; } /* Blue */
    </style>
</head>
<body>
    <button id="addHighlightBtn">Add Highlight</button>
    <button id="removeHighlightBtn">Remove Highlight</button>
    <button id="toggleHighlightBtn">Toggle Highlight</button>
    <div id="myBox">Interactive Box</div>

    <script>
        document.addEventListener('DOMContentLoaded', () => {
            const myBox = document.getElementById('myBox');
            const addBtn = document.getElementById('addHighlightBtn');
            const removeBtn = document.getElementById('removeHighlightBtn');
            const toggleBtn = document.getElementById('toggleHighlightBtn');

            if (myBox && addBtn && removeBtn && toggleBtn) {
                addBtn.addEventListener('click', () => {
                    myBox.classList.add('highlight');
                    console.log("Class 'highlight' added.");
                });
                removeBtn.addEventListener('click', () => {
                    myBox.classList.remove('highlight');
                    console.log("Class 'highlight' removed.");
                });
                toggleBtn.addEventListener('click', () => {
```

```
            myBox.classList.toggle('highlight');
            console.log("Class 'highlight' toggled.");
        });
    } else {
        console.error("One or more elements not found!");
    }
});
console.log("Run this code in an HTML file to see the DOM manipulation.");
</script>
</body>
</html>
```

Exercise 76: Creating and Appending Elements

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exercise 76</title>
    <style>
        #outputContainer {
            border: 1px dashed grey;
            padding: 10px;
            margin-top: 20px;
            border-radius: 5px;
            background-color: #f8f8f8;
        }
    </style>
</head>
<body>
    <div id="outputContainer">
        <h3>Content will be added below:</h3>
    </div>

    <script>
        document.addEventListener('DOMContentLoaded', () => {
            const outputContainer = document.getElementById('outputContainer');
```

```
        if (outputContainer) {
            // 1. Create a new <h2> element
            const newHeading = document.createElement('h2');
            newHeading.textContent = "Dynamically Created Heading";
            newHeading.style.color = "#2c3e50";
            newHeading.style.marginTop = "10px";
            // 2. Create a new paragraph element
            const newParagraph = document.createElement('p');
            newParagraph.textContent = "This paragraph was proudly added by
JavaScript!";
            newParagraph.style.fontStyle = "italic";
            newParagraph.style.color = "#34495e";
            newParagraph.style.marginBottom = "10px";
            // 3. Append both new elements to the outputContainer
            outputContainer.appendChild(newHeading);
            outputContainer.appendChild(newParagraph);
            console.log("New elements created and appended to outputContainer.");

            // Create and append a button that adds more content
            const addButton = document.createElement('button');
            addButton.textContent = "Add More Content";
            addButton.className = "bg-purple-600 text-white p-2 rounded-md mt-4";
/* Example class, for visual */
            outputContainer.appendChild(addButton);

            let clickCount = 0;
            addButton.addEventListener('click', () => {
                clickCount++;
                const dynamicPara = document.createElement('p');
                dynamicPara.textContent = `You clicked the button! This is dynamic
content #${clickCount}.`;
                dynamicPara.style.backgroundColor = '#f0f4f8';
                dynamicPara.style.padding = '5px';
                dynamicPara.style.borderRadius = '4px';
                dynamicPara.style.marginTop = '5px';
                outputContainer.insertBefore(dynamicPara, addButton); // Insert before
the button
            });
        } else {
```

```
            console.error("Element with ID 'outputContainer' not found!");
        }
    });
    console.log("Run this code in an HTML file to see the DOM manipulation.");
</script>
</body>
</html>
```

Exercise 77: Basic Event Listener (click)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exercise 77</title>
</head>
<body>
    <button id="myButton">Click Me</button>
    <p id="clickCountDisplay" style="margin-top: 10px; font-size: 1.2em;">Clicks: 0</p>

    <script>
        document.addEventListener('DOMContentLoaded', () => {
            const myButton = document.getElementById('myButton');
            const clickCountDisplay = document.getElementById('clickCountDisplay');
            let clickCount = 0;

            if (myButton && clickCountDisplay) {
                // Add a click event listener to the button
                myButton.addEventListener('click', () => {
                    clickCount++;
                    console.log("Button clicked! Total clicks:", clickCount);
                    clickCountDisplay.textContent = `Clicks: ${clickCount}`;
                    // Add a temporary visual feedback
                    myButton.style.transform = 'scale(0.98)';
                    setTimeout(() => {
                        myButton.style.transform = 'scale(1)';
                    }, 100);
```

```
            });
        } else {
            console.error("Button or display element not found!");
        }
    });
    console.log("Run this code in an HTML file and click the button to see console
output.");
  </script>
</body>
</html>
```

Exercise 78: Input Event Listener (input)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exercise 78</title>
  <style>
    input[type="text"] {
      padding: 8px;
      width: 250px;
      border: 1px solid #ccc;
      border-radius: 4px;
      margin-bottom: 10px;
    }
    #displayArea {
      font-size: 1.1em;
      color: #333;
    }
  </style>
</head>
<body>
  <div>
    <label for="myInput">Type something:</label><br>
    <input type="text" id="myInput">
    <p id="displayArea">You typed: <span style="font-weight: bold; color:
```

```
blue;"></span></p>
  </div>

  <script>
    document.addEventListener('DOMContentLoaded', () => {
      const myInput = document.getElementById('myInput');
      const displayArea = document.getElementById('displayArea');
      const typedTextSpan = displayArea ? displayArea.querySelector('span') : null; //
Span to update

      if (myInput && displayArea && typedTextSpan) {
        // Add an 'input' event listener to the input field
        myInput.addEventListener('input', (event) => {
          // event.target refers to the element that triggered the event (the input
field)
          // event.target.value gives the current value of the input field
          typedTextSpan.textContent = event.target.value;
          console.log("Input value:", event.target.value);
        });
      } else {
        console.error("Input or display elements not found!");
      }
    });
    console.log("Run this code in an HTML file and type into the input field to see live
updates.");
  </script>
</body>
</html>
```

Exercise 79: fetch API (Simple GET Request)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exercise 79</title>
</head>
```

```
<body>
    <h1>Fetch API Example</h1>
    <p>Check the browser's console for the fetched data.</p>

    <script>
        // A public API endpoint to fetch data from
        const API_URL = 'https://jsonplaceholder.typicode.com/posts/1';
        const INVALID_API_URL = 'https://jsonplaceholder.typicode.com/posts/99999999';
// To demonstrate error

        async function fetchAndDisplayPost(url) {
            console.log(`\nAttempting to fetch data from: ${url}`);
            try {
                const response = await fetch(url);
                // Initiate the fetch request

                // Check if the request was successful (status code 200-299)
                if (!response.ok) {
                    // If not successful, throw an error with the status text
                    throw new Error(`HTTP error! Status: ${response.status} -
${response.statusText}`);
                }

                const data = await response.json();
                // Parse the response body as JSON
                console.log("Successfully fetched data:");
                console.log(data);
                // Log the parsed JSON data

            } catch (error) {
                // Catch any network errors or errors thrown from the response.ok check
                console.error("Error fetching data:", error.message);
            }
        }

        // Call the function to fetch data
        fetchAndDisplayPost(API_URL);
        fetchAndDisplayPost(INVALID_API_URL);
        // To demonstrate error handling
```

```
    console.log("This message will appear before the fetch results, demonstrating asynchronicity.");
  </script>
</body>
</html>
```

Exercise 80: Basic Animation with setInterval

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exercise 80</title>
  <style>
    #animatedBox {
      width: 60px;
      height: 60px;
      background-color: #28a745; /* Green */
      border-radius: 8px;
      position: relative; /* Essential for 'left' property to work */
      left: 0px;
      top: 10px;
      transition: background-color 0.3s ease;
    }
    .controls {
      margin-top: 20px;
    }
    .control-btn {
      padding: 8px 16px;
      border-radius: 6px;
      font-size: 1em;
      cursor: pointer;
      border: none;
      margin-right: 10px;
      transition: background-color 0.2s ease;
    }
```

```
        .start-btn { background-color: #007bff; color: white; }
        .start-btn:hover { background-color: #0056b3; }
        .stop-btn { background-color: #dc3545; color: white; }
        .stop-btn:hover { background-color: #b02a3a; }
    </style>
</head>
<body>
    <div id="animatedBox"></div>
    <div class="controls">
        <button id="startButton" class="control-btn start-btn">Start Animation</button>
        <button id="stopButton" class="control-btn stop-btn">Stop Animation</button>
    </div>

    <script>
        document.addEventListener('DOMContentLoaded', () => {
            const animatedBox = document.getElementById('animatedBox');
            const startButton = document.getElementById('startButton');
            const stopButton = document.getElementById('stopButton');

            let position = 0;
            let animationIntervalId;
            const speed = 2; // Pixels per interval
            const maxPosition = window.innerWidth - 80; // Stop before going off screen
(adjust for box width)
            const intervalDelay = 10; // Milliseconds per update

            function animateBox() {
                if (animatedBox) {
                    animationIntervalId = setInterval(() => {
                        position += speed;
                        if (position > maxPosition) {
                            position = 0; // Reset to start
                        }
                        animatedBox.style.left = position + 'px';
                    }, intervalDelay);
                    console.log("Animation started.");
                } else {
                    console.error("Animated box not found!");
                }
```

```
        }

        function stopAnimation() {
            if (animationIntervalId) {
                clearInterval(animationIntervalId);
                animationIntervalId = null;
                console.log("Animation stopped.");
            }
        }

        if (startButton && stopButton) {
            startButton.addEventListener('click', () => {
                if (!animationIntervalId) { // Prevent starting multiple intervals
                    animateBox();
                }
            });
            stopButton.addEventListener('click', stopAnimation);
            // Optional: Stop animation if window resizes to recalculate maxPosition
            window.addEventListener('resize', stopAnimation);
        } else {
            console.error("Animation control buttons not found!");
        }
    });
    console.log("Run this code in an HTML file to see the animation.");
  </script>
</body>
</html>
```

Exercise 81: Binary Search

```
// Exercise 81: Binary Search
function binarySearch(arr, target) {
  let left = 0;
  let right = arr.length - 1;
  while (left <= right) {
    const mid = Math.floor((left + right) / 2);
    // Calculate the middle index
    // Check if the middle element is the target
```

```
    if (arr[mid] === target) {
      return mid;
      // Target found
    }
    // If the target is greater, ignore the left half
    if (arr[mid] < target) {
      left = mid + 1;
    }
    // If the target is smaller, ignore the right half
    else {
      right = mid - 1;
    }
  }
  return -1; // Target not found in the array
}
const sortedArray = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91];
console.log("Index of 12:", binarySearch(sortedArray, 12)); // Expected: 3
console.log("Index of 23:", binarySearch(sortedArray, 23));
// Expected: 5
console.log("Index of 2:", binarySearch(sortedArray, 2)); // Expected: 0
console.log("Index of 91:", binarySearch(sortedArray, 91));
// Expected: 9
console.log("Index of 7:", binarySearch(sortedArray, 7)); // Expected: -1 (not found)
console.log("Index of 100:", binarySearch(sortedArray, 100));
// Expected: -1 (not found)
console.log("Index of 38:", binarySearch(sortedArray, 38));
// Expected: 6
```

Exercise 82: Bubble Sort

```
// Exercise 82: Bubble Sort
function bubbleSort(arr) {
  const n = arr.length;
  let swapped; // Flag to optimize: if no swaps in an pass, array is sorted
  // Outer loop: iterate through the array from the beginning
  // After each pass, the largest unsorted element "bubbles" to its correct position at
the end
  for (let i = 0; i < n - 1; i++) {
```

```
    swapped = false;
    // Reset flag for each pass
    // Inner loop: compare adjacent elements and swap if they are in the wrong order
    // The last 'i' elements are already in place, so we don't need to check them
    for (let j = 0; j < n - 1 - i; j++) {
      if (arr[j] > arr[j + 1]) {
        // Swap arr[j] and arr[j + 1]
        [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]]; // ES6 array destructuring for swapping
        swapped = true;
        // A swap occurred in this pass
      }
    }
    // Optimization: If no two elements were swapped by inner loop, then array is sorted
    if (!swapped) {
      break;
    }
  }
  return arr; // Return the sorted array
}
const unsortedArray1 = [64, 34, 25, 12, 22, 11, 90];
console.log("Bubble Sort [64, 34, 25, 12, 22, 11, 90]:", bubbleSort([...unsortedArray1]));
// Use spread to avoid mutating original
const unsortedArray2 = [5, 1, 4, 2, 8];
console.log("Bubble Sort [5, 1, 4, 2, 8]:", bubbleSort([...unsortedArray2]));
const unsortedArray3 = [1, 2, 3, 4, 5];
// Already sorted
console.log("Bubble Sort [1, 2, 3, 4, 5]:", bubbleSort([...unsortedArray3]));
const unsortedArray4 = [9, 8, 7, 6, 5];
// Reverse sorted
console.log("Bubble Sort [9, 8, 7, 6, 5]:", bubbleSort([...unsortedArray4]));
```

Exercise 83: Selection Sort

```
// Exercise 83: Selection Sort
function selectionSort(arr) {
  const n = arr.length;
  // Outer loop: iterate through the array
  for (let i = 0; i < n - 1; i++) {
```

```
    // Assume the current element is the minimum
    let minIndex = i;
    // Inner loop: find the smallest element in the unsorted portion
    for (let j = i + 1; j < n; j++) {
      if (arr[j] < arr[minIndex]) {
        minIndex = j;
        // Update minIndex if a smaller element is found
      }
    }
    // If the smallest element is not at the current position 'i', swap them
    if (minIndex !== i) {
      [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]];
      // ES6 swap
    }
  }
  return arr;
  // Return the sorted array
}
const unsortedArray1 = [64, 25, 12, 22, 11];
console.log("Selection Sort [64, 25, 12, 22, 11]:", selectionSort([...unsortedArray1]));
const unsortedArray2 = [5, 1, 4, 2, 8];
console.log("Selection Sort [5, 1, 4, 2, 8]:", selectionSort([...unsortedArray2]));
const unsortedArray3 = [1, 2, 3, 4, 5];
// Already sorted
console.log("Selection Sort [1, 2, 3, 4, 5]:", selectionSort([...unsortedArray3]));
```

Exercise 84: Insertion Sort

```
// Exercise 84: Insertion Sort
function insertionSort(arr) {
  const n = arr.length;
  // Start from the second element (index 1) because the first element (index 0)
  // is considered the "sorted" part initially.
  for (let i = 1; i < n; i++) {
    let current = arr[i];
    // The element to be inserted into the sorted portion
    let j = i - 1;
    // Pointer to the last element of the sorted portion
```

```
    // Move elements of arr[0..i-1], that are greater than current,
    // to one position ahead of their current position
    while (j >= 0 && arr[j] > current) {
      arr[j + 1] = arr[j];
      // Shift element to the right
      j--;
    }
    // Place current element at its correct position in the sorted part
    arr[j + 1] = current;
  }
  return arr; // Return the sorted array
}
const unsortedArray1 = [12, 11, 13, 5, 6];
console.log("Insertion Sort [12, 11, 13, 5, 6]:", insertionSort([...unsortedArray1]));
const unsortedArray2 = [5, 1, 4, 2, 8];
console.log("Insertion Sort [5, 1, 4, 2, 8]:", insertionSort([...unsortedArray2]));
const unsortedArray3 = [1, 2, 3, 4, 5];
// Already sorted (best case)
console.log("Insertion Sort [1, 2, 3, 4, 5]:", insertionSort([...unsortedArray3]));
const unsortedArray4 = [9, 8, 7, 6, 5]; // Reverse sorted (worst case)
console.log("Insertion Sort [9, 8, 7, 6, 5]:", insertionSort([...unsortedArray4]));
```

Exercise 85: Memoization (Simple Factorial)

```
// Exercise 85: Memoization (Simple Factorial)
// Using a Map for cache for better key flexibility (though object keys work for
numbers)
const factorialCache = new Map();
function memoizedFactorial(n) {
  // Base case for recursion
  if (n === 0 || n === 1) {
    return 1;
  }
  // Check if the result is already in the cache
  if (factorialCache.has(n)) {
    console.log(`Getting factorial(${n}) from cache.`);
    return factorialCache.get(n);
  }
```

```
  // If not in cache, compute the result recursively
  console.log(`Computing factorial(${n})...`);
  const result = n * memoizedFactorial(n - 1);
  // Store the computed result in the cache before returning
  factorialCache.set(n, result);
  return result;
}
console.log("--- First set of calls ---");
console.log("Factorial(5):", memoizedFactorial(5)); // Will compute 5!, 4!, 3!, 2!, 1!, 0!
console.log("\n--- Second set of calls (demonstrating caching) ---");
console.log("Factorial(3):", memoizedFactorial(3)); // Should use cache for 3!, 2!, 1!, 0!
console.log("Factorial(6):", memoizedFactorial(6)); // Will compute 6!, then use cache
for 5! and below
console.log("Factorial(5):", memoizedFactorial(5));
// Should use cache directly
```

Exercise 86: Simple Debounce Function

```
// Exercise 86: Simple Debounce Function
function debounce(func, delay) {
  let timeoutId;
  // This variable will store the timer ID across calls
  // Return a new function (the debounced version)
  return function(...args) { // ...args captures all arguments passed to the debounced
function
    const context = this;
    // Capture the 'this' context of the call
    // Clear any existing timer.
    // If the debounced function is called again before the delay
    // elapses, the previous timer is cancelled, and a new one is set.
    clearTimeout(timeoutId);
    // Set a new timer
    timeoutId = setTimeout(() => {
      // Execute the original function after the delay
      // Use .apply to correctly pass 'this' context and arguments
      func.apply(context, args);
    }, delay);
  };
```

```
}
// Example Usage: Simulate a search input
function performSearch(query) {
  console.log(`Performing search for: "${query}"...`);
}
// Create a debounced version of performSearch with a 500ms delay
const debouncedSearch = debounce(performSearch, 500);
console.log("Simulating rapid typing in a search bar...");
debouncedSearch("a");
debouncedSearch("ap");
debouncedSearch("app");
setTimeout(() => debouncedSearch("appl"), 100);
setTimeout(() => debouncedSearch("apple"), 200);
// This will be the only one that triggers performSearch
setTimeout(() => console.log("--- Finished typing simulation ---"), 1000);
// Another example: Button click that only triggers once
let clickCount = 0;
function handleClick() {
  clickCount++;
  console.log(`Button clicked! (Actual click count: ${clickCount})`);
}
const debouncedClick = debounce(handleClick, 1000);
console.log("\nSimulating rapid button clicks...");
debouncedClick(); // Calls debounce, sets timer
setTimeout(debouncedClick, 100); // Clears previous, sets new timer
setTimeout(debouncedClick, 200);
// Clears previous, sets new timer
setTimeout(debouncedClick, 300); // Clears previous, sets new timer
// After 1000ms from the LAST call (at 300ms), handleClick will fire once.
setTimeout(() => console.log("--- Finished click simulation ---"), 1500);
```

Exercise 87: Simple Throttling Function

```
// Exercise 87: Simple Throttling Function
function throttle(func, delay) {
  let inThrottle;
  // Flag to indicate if we are currently in a throttled state
  let lastFn;
```

```
  // Stores a reference to the setTimeout callback
  let lastTime;
  // Stores the timestamp of the last execution
  // Return a new function (the throttled version)
  return function(...args) {
    const context = this;
    // If not currently throttled, execute immediately
    if (!inThrottle) {
      func.apply(context, args);
      lastTime = Date.now();
      inThrottle = true; // Enter throttled state
    } else {
      // If currently throttled, clear any pending execution
      clearTimeout(lastFn);
      // Schedule a new execution after the remaining delay has passed
      // Math.max ensures the delay is at least 0, preventing negative delays
      lastFn = setTimeout(() => {
        // Check if enough time has passed since the last execution
        if ((Date.now() - lastTime) >= delay) {
          func.apply(context, args);
          lastTime = Date.now();
          inThrottle = false; // Exit throttled state (important for subsequent immediate
executions)
        }
      }, Math.max(delay - (Date.now() - lastTime), 0));
    }
  };
}
// Example Usage: Simulate a scroll event
function handleScroll(event) {
  console.log(`Scrolling detected! Timestamp: ${new Date().toLocaleTimeString()}`);
  // In a real scenario, you'd do DOM updates here, e.g., update scroll position display
}
// Create a throttled version of handleScroll with a 1000ms (1 second) delay
const throttledScroll = throttle(handleScroll, 1000);
console.log("Simulating rapid scroll events (will only log every 1 second)...");
throttledScroll(); // Should trigger immediately
setTimeout(throttledScroll, 100);
setTimeout(throttledScroll, 200);
```

```
setTimeout(throttledScroll, 300);
setTimeout(throttledScroll, 1050); // Should trigger after 1 sec
setTimeout(throttledScroll, 1100);
setTimeout(throttledScroll, 2200);
// Should trigger again after another 1 sec from previous
setTimeout(() => console.log("--- Finished scroll simulation ---"), 3000);
```

Exercise 88: Linked List (Basic Implementation)

```
// Exercise 88: Linked List (Basic Implementation)
// Represents a single node in the linked list
class Node {
  constructor(value) {
    this.value = value;
    this.next = null; // Pointer to the next node
  }
}
// Manages the linked list
class LinkedList {
  constructor() {
    this.head = null;
    // The first node in the list
    this.tail = null;
    // The last node in the list
    this.size = 0;
    // Number of elements in the list
  }
  // Adds a new node to the end of the list
  add(value) {
    const newNode = new Node(value);
    if (!this.head) {
      // If the list is empty, the new node is both the head and the tail
      this.head = newNode;
      this.tail = newNode;
    } else {
      // Otherwise, append the new node to the end and update the tail
      this.tail.next = newNode;
      this.tail = newNode;
```

```
  }
  this.size++;
  console.log(`Added "${value}". Current size: ${this.size}`);
 }
 // Prints all values in the list
 print() {
  if (!this.head) {
   console.log("Linked List is empty.");
   return;
  }
  let current = this.head;
  let result = [];
  while (current) {
   result.push(current.value);
   current = current.next;
   // Move to the next node
  }
  console.log(`Linked List: ${result.join(" -> ")}`);
 }
 // Finds if a value exists in the list
 find(value) {
  if (!this.head) {
   return false;
  }
  let current = this.head;
  while (current) {
   if (current.value === value) {
    return true;
    // Value found
   }
   current = current.next;
  }
  return false; // Value not found
 }
 // Removes the first occurrence of a value
 remove(value) {
  if (!this.head) {
   console.log("Cannot remove: List is empty.");
   return false;
```

```
    }
    // If the head needs to be removed
    if (this.head.value === value) {
      this.head = this.head.next;
      if (!this.head) { // If list becomes empty
        this.tail = null;
      }
      this.size--;
      console.log(`Removed "${value}". Current size: ${this.size}`);
      return true;
    }
    let current = this.head;
    while (current.next && current.next.value !== value) {
      current = current.next;
    }
    if (current.next) { // Found the node to remove
      if (current.next === this.tail) { // If removing the tail
        this.tail = current;
      }
      current.next = current.next.next;
      this.size--;
      console.log(`Removed "${value}". Current size: ${this.size}`);
      return true;
    }
    console.log(`"${value}" not found for removal.`);
    return false;
  }
}
// Demonstrate Linked List usage
const myList = new LinkedList();
myList.print(); // Expected: Linked List is empty.
myList.add("A");
myList.add("B");
myList.add("C");
myList.print(); // Expected: Linked List: A -> B -> C
console.log("Does 'B' exist?", myList.find("B"));
// Expected: true
console.log("Does 'D' exist?", myList.find("D")); // Expected: false
myList.remove("B"); // Expected: Removed "B". Current size: 2
```

```
myList.print();
// Expected: Linked List: A -> C
myList.remove("A"); // Expected: Removed "A". Current size: 1
myList.print();
// Expected: Linked List: C
myList.remove("C"); // Expected: Removed "C". Current size: 0
myList.print(); // Expected: Linked List is empty.
myList.remove("X"); // Expected: Cannot remove: List is empty.
```

Exercise 89: Tree Traversal (Depth-First Search - Preorder)

```
// Exercise 89: Tree Traversal (Depth-First Search - Preorder)
// Represents a single node in the binary tree
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null; // Pointer to the left child node
    this.right = null;
    // Pointer to the right child node
  }
}
// Performs a Depth-First Search (DFS) in Preorder traversal
// Preorder: Visit Root -> Traverse Left -> Traverse Right
function preorderTraversal(node) {
  if (node === null) {
    return;
    // Base case: if node is null, stop recursion
  }
  // 1. Visit the current node (Root)
  console.log(node.value);
  // 2. Traverse the left subtree
  preorderTraversal(node.left);
  // 3. Traverse the right subtree
  preorderTraversal(node.right);
}
// Example Tree Structure:
/* 10
```

```
      /  \
     5    15
    / \    \
   2   7    20 */
const root = new TreeNode(10);
root.left = new TreeNode(5);
root.right = new TreeNode(15);
root.left.left = new TreeNode(2);
root.left.right = new TreeNode(7);
root.right.right = new TreeNode(20);
console.log("Preorder Traversal (Root -> Left -> Right):");
preorderTraversal(root); // Expected: 10, 5, 2, 7, 15, 20
// Another example tree
/* A
     / \
    B   C
   /
  D */
const root2 = new TreeNode('A');
root2.left = new TreeNode('B');
root2.right = new TreeNode('C');
root2.left.left = new TreeNode('D');
console.log("\nPreorder Traversal for another tree:");
preorderTraversal(root2);
// Expected: A, B, D, C
```

Exercise 90: Tree Traversal (Breadth-First Search - Level Order)

```
// Exercise 90: Tree Traversal (Breadth-First Search - Level Order)
// Reusing TreeNode class from Exercise 89
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}
```

```
// Performs a Breadth-First Search (BFS) / Level-Order Traversal
function levelOrderTraversal(root) {
  if (root === null) {
    return;
    // Base case: empty tree
  }
  // Use a queue to keep track of nodes to visit
  // (A simple array can act as a queue using push/shift)
  const queue = [];
  queue.push(root);
  console.log("Level Order Traversal (BFS):");
  while (queue.length > 0) {
    const currentNode = queue.shift();
    // Dequeue the first node (FIFO)
    console.log(currentNode.value);
    // Visit the current node
    // Enqueue its left child if it exists
    if (currentNode.left !== null) {
      queue.push(currentNode.left);
    }
    // Enqueue its right child if it exists
    if (currentNode.right !== null) {
      queue.push(currentNode.right);
    }
  }
}
// Example Tree Structure (same as Ex 89):
/* 10
       /  \
      5    15
     / \    \
    2   7    20 */
const root = new TreeNode(10);
root.left = new TreeNode(5);
root.right = new TreeNode(15);
root.left.left = new TreeNode(2);
root.left.right = new TreeNode(7);
root.right.right = new TreeNode(20);
levelOrderTraversal(root); // Expected: 10, 5, 15, 2, 7, 20
```

```
// Another example tree
/* A
     / \
    B   C
   /
  D */
const root2 = new TreeNode('A');
root2.left = new TreeNode('B');
root2.right = new TreeNode('C');
root2.left.left = new TreeNode('D');
console.log("\nLevel Order Traversal for another tree:");
levelOrderTraversal(root2);
// Expected: A, B, C, D
```

Exercise 91: Currying a Function

```
// Exercise 91: Currying a Function
// The original function we want to curry
function sum(a, b, c) {
  return a + b + c;
}
// The curry function
function curry(func) {
  // Returns a new function that handles the currying logic
  return function curried(...args) {
    // If the number of arguments received is enough for the original function,
    // execute the original function with these arguments.
    if (args.length >= func.length) { // func.length gives the number of expected
arguments
      return func.apply(this, args);
    } else {
      // If not enough arguments, return another function that expects more arguments.
      // This new function will concatenate the previously received arguments with the
new ones.
      return function(...nextArgs) {
        return curried.apply(this, args.concat(nextArgs));
      };
    }
```

```
  };
}
// Demonstrate currying
const curriedSum = curry(sum);
console.log("Curried Sum (one by one):", curriedSum(1)(2)(3));
// Expected: 6
console.log("Curried Sum (two then one):", curriedSum(1, 2)(3));
// Expected: 6
console.log("Curried Sum (all at once):", curriedSum(1, 2, 3));
// Expected: 6
// Example with another function
function multiply(a, b, c, d) {
  return a * b * c * d;
}
const curriedMultiply = curry(multiply);
console.log("Curried Multiply:", curriedMultiply(2)(3)(4)(5)); // Expected: 120
console.log("Curried Multiply:", curriedMultiply(2, 3)(4, 5));
// Expected: 120
```

Exercise 92: Partial Application

```
// Exercise 92: Partial Application
function greet(greeting, name, punctuation) {
  console.log(`${greeting}, ${name}${punctuation}`);
}
// --- Method 1: Using .bind() for partial application ---
// .bind(thisArg, arg1, arg2, ...)
// - The first argument is the 'this' context (null or undefined here for global functions)
// - Subsequent arguments are prepended to the arguments of the original function
const sayHelloWithBind = greet.bind(null, "Hello", "!");
// Pre-fills greeting and punctuation
console.log("--- Using .bind() ---");
sayHelloWithBind("Alice"); // Expected: Hello, Alice!
sayHelloWithBind("Bob"); // Expected: Hello, Bob!
// --- Method 2: Manual Partial Application using a Closure ---
// Create a higher-order function that returns a new function
// The inner function closes over the pre-filled arguments
function createGreeting(greeting, punctuation) {
```

```
  return function(name) { // This is the partially applied function
    greet(greeting, name, punctuation);
  };
}
const sayHiExclamatory = createGreeting("Hi", "!");
const sayGoodMorningPeriod = createGreeting("Good morning", ".");
console.log("\n--- Using Closure ---");
sayHiExclamatory("Charlie");
// Expected: Hi, Charlie!
sayGoodMorningPeriod("Diana"); // Expected: Good morning, Diana.
// Another example of partial application with bind
function calculateDiscount(price, discountPercentage) {
  return price * (1 - discountPercentage);
}
const applyTenPercentDiscount = calculateDiscount.bind(null, 0.10); // Pre-fills
discountPercentage
console.log("\nPrice after 10% discount on 100:", applyTenPercentDiscount(100));
// Expected: 90
console.log("Price after 10% discount on 250:", applyTenPercentDiscount(250));
// Expected: 225
```

Exercise 93: Function Composition

```
// Exercise 93: Function Composition
// Simple functions to compose
const add2 = num => num + 2;
const multiply3 = num => num * 3;
const square = num => num * num;
const negate = num => -num;
// The compose function (applies functions from right to left)
function compose(...funcs) {
  // Returns a new function that takes an initial argument
  return function(initialArg) {
    // Use reduceRight to apply functions from right to left
    // The accumulator (acc) starts with initialArg
    // For each function (fn), it's called with the current accumulator's value
    return funcs.reduceRight((acc, fn) => fn(acc), initialArg);
  };
```

```
}
// The pipe function (alternative, applies functions from left to right)
function pipe(...funcs) {
  return function(initialArg) {
    return funcs.reduce((acc, fn) => fn(acc), initialArg);
  };
}
// Demonstrate composition
const composedFunc = compose(square, multiply3, add2);
// (5 + 2) => 7
// (7 * 3) => 21
// (21 * 21) => 441
console.log("compose(square, multiply3, add2)(5):", composedFunc(5));
// Expected: 441
const composedAndNegated = compose(negate, square, add2); // (3 + 2) => 5
// (5 * 5) => 25
// (-25) => -25
console.log("compose(negate, square, add2)(3):", composedAndNegated(3));
// Expected: -25
// Demonstrate pipe
const pipedFunc = pipe(add2, multiply3, square);
// (5 + 2) => 7
// (7 * 3) => 21
// (21 * 21) => 441
console.log("pipe(add2, multiply3, square)(5):", pipedFunc(5));
// Expected: 441 (same result as compose in this specific case, but order differs)
```

Exercise 94: Singleton Pattern

```
// Exercise 94: Singleton Pattern
class Logger {
  // Private static field to hold the single instance
  static #instance = null;
  // Private constructor to prevent direct instantiation
  constructor() {
    if (Logger.#instance) {
      // If an instance already exists, throw an error to prevent direct calls
      throw new Error("Logger instance already exists. Use Logger.getInstance()
```

```
instead.");
    }
    this.logs = []; // To store log messages
    console.log("Logger: Initializing new instance...");
  }
  // Static method to get the single instance of the Logger
  static getInstance() {
    if (!Logger.#instance) {
      // If no instance exists, create one
      Logger.#instance = new Logger();
    }
    return Logger.#instance; // Return the existing instance
  }
  // Method to log a message
  log(message) {
    const timestamp = new Date().toISOString();
    const logEntry = `${timestamp} - ${message}`;
    this.logs.push(logEntry);
    console.log(`LOG: ${logEntry}`);
  }
  // Method to get all logs
  getLogs() {
    return this.logs;
  }
}
// Demonstrate the Singleton pattern
console.log("Attempting to get Logger instances...");
const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();
// This should return the same instance as logger1
console.log("\nAre logger1 and logger2 the same instance?", logger1 === logger2);
// Expected: true
logger1.log("Application started.");
logger2.log("User logged in.");
console.log("\nAll logs from logger1:", logger1.getLogs());
console.log("All logs from logger2:", logger2.getLogs());
// Should be the same logs as logger1
// Try to create a new instance directly (should throw an error)
try {
```

```
  const logger3 = new Logger();
} catch (e) {
  console.error("\nCaught an error when trying to instantiate Logger directly:",
e.message);
}
```

Exercise 95: Factory Pattern (Simple)

```
// Exercise 95: Factory Pattern (Simple)
// 1. Define product classes (or constructor functions)
class Car {
  constructor(options) {
    this.brand = options.brand ||
    "Generic Car";
    this.model = options.model || "Model X";
    this.doors = options.doors || 4;
  }
  drive() {
    console.log(`Driving the ${this.brand} ${this.model} (Car).`);
  }
}
class Motorcycle {
  constructor(options) {
    this.brand = options.brand || "Generic Moto";
    this.model = options.model || "Model Y";
    this.engineCC = options.engineCC || 250;
  }
  ride() {
    console.log(`Riding the ${this.brand} ${this.model} (Motorcycle).`);
  }
}
class Bicycle {
  constructor(options) {
    this.brand = options.brand ||
    "Generic Bike";
    this.model = options.model || "Model Z";
    this.gears = options.gears || 1;
  }
```

```
  pedal() {
    console.log(`Pedaling the ${this.brand} ${this.model} (Bicycle).`);
  }
}
// 2. Implement the Factory class
class VehicleFactory {
  createVehicle(type, options) {
    switch (type.toLowerCase()) {
      case 'car':
        return new Car(options);
      case 'motorcycle':
        return new Motorcycle(options);
      case 'bicycle':
        return new Bicycle(options);
      default:
        throw new Error(`Unknown vehicle type: ${type}`);
    }
  }
}
// Demonstrate the Factory usage
const factory = new VehicleFactory();
const myCar = factory.createVehicle('car', { brand: 'Toyota', model: 'Camry', doors: 4 });
const myMotorcycle = factory.createVehicle('motorcycle', { brand: 'Honda', model:
'CBR500R', engineCC: 500 });
const myBicycle = factory.createVehicle('bicycle', { brand: 'Schwinn', model: 'Cruiser',
gears: 7 });
console.log("Created Vehicles:");
console.log(myCar);
myCar.drive();
console.log(myMotorcycle);
myMotorcycle.ride();
console.log(myBicycle);
myBicycle.pedal();
try {
  const unknownVehicle = factory.createVehicle('boat', { brand: 'SeaRay' });
} catch (e) {
  console.error("\nCaught error for unknown vehicle type:", e.message);
}
```

Exercise 96: Observer Pattern (Using Custom Event Emitter)

```
// Exercise 96: Observer Pattern (Using Custom Event Emitter)
// Reusing the EventEmitter class from Exercise 67
class EventEmitter {
  constructor() {
    this.listeners = new Map();
  }
  on(eventName, listener) {
    if (!this.listeners.has(eventName)) {
      this.listeners.set(eventName, []);
    }
    this.listeners.get(eventName).push(listener);
    // console.log(`[EventEmitter] Registered listener for '${eventName}'.`);
  }
  emit(eventName, ...args) {
    const eventListeners = this.listeners.get(eventName);
    if (eventListeners) {
      // console.log(`[EventEmitter] Emitting event '${eventName}' with args:
${JSON.stringify(args)}`);
      eventListeners.forEach(listener => {
        try {
          listener(...args);
        } catch (e) {
          console.error(`[EventEmitter] Error in listener for '${eventName}':`, e);
        }
      });
    } else {
      // console.log(`[EventEmitter] No listeners registered for '${eventName}'.`);
    }
  }
  off(eventName, listenerToRemove) {
    const eventListeners = this.listeners.get(eventName);
    if (eventListeners) {
      const index = eventListeners.indexOf(listenerToRemove);
      if (index > -1) {
        eventListeners.splice(index, 1);
```

```
      // console.log(`[EventEmitter] Unregistered listener for '${eventName}'.`);
     } else {
      // console.log(`[EventEmitter] Listener not found for '${eventName}'.`);
     }
    } else {
     // console.log(`[EventEmitter] No listeners registered for '${eventName}'.`);
    }
  }
}
// The Subject (Publisher) that holds state and notifies observers
class Subject {
  constructor(initialState) {
    this.emitter = new EventEmitter();
    this.state = initialState;
    console.log(`Subject initialized with state: ${this.state}`);
  }
  // Method for observers to subscribe
  subscribe(listener) {
    this.emitter.on('stateChange', listener);
    console.log("Observer subscribed to 'stateChange' event.");
  }
  // Method for observers to unsubscribe
  unsubscribe(listener) {
    this.emitter.off('stateChange', listener);
    console.log("Observer unsubscribed from 'stateChange' event.");
  }
  // Method to change the subject's state and notify observers
  changeState(newState) {
    if (this.state !== newState) {
      this.state = newState;
      console.log(`\nSubject state changed to: ${this.state}`);
      this.emitter.emit('stateChange', this.state); // Emit the event with the new state
    } else {
      console.log(`\nState is already '${this.state}', no change.`);
    }
  }
  getCurrentState() {
    return this.state;
  }
```

```
}
// Observer functions (can be part of classes/objects too)
const displayLogger = (newState) => console.log(`Logger: State updated to ->
${newState}`);
const alertUser = (newState) => {
  if (newState === 'error') {
    console.warn(`ALERT: An error state was detected! Current state: ${newState}`);
  }
};
const simpleReporter = (newState) => console.log(`Reporter: New state is
${newState}.`);
// Demonstrate the Observer pattern
const dataSubject = new Subject("initial_status");
// Observers subscribe to the subject
dataSubject.subscribe(displayLogger);
dataSubject.subscribe(alertUser);
dataSubject.subscribe(simpleReporter);
// Change the subject's state, which will notify all subscribed observers
dataSubject.changeState("loading");
dataSubject.changeState("data_fetched");
dataSubject.changeState("error");
// This will trigger the alertUser observer
dataSubject.changeState("error"); // No change, no notification
// Unsubscribe an observer
dataSubject.unsubscribe(displayLogger);
dataSubject.changeState("resolved");
// displayLogger will no longer be notified
```

Exercise 97: Mixin Pattern

```
// Exercise 97: Mixin Pattern
// 1. Define the Mixin (a simple object containing methods to be mixed in)
const LoggerMixin = {
  // The 'this' context within this method will refer to the object
  // (or class instance) that the mixin is applied to.
  log(message) {
    // Use this.constructor.name to identify the class instance
    console.log(`[${this.constructor.name} LOG]: ${message}`);
```

```
  },
  // You can add more methods or properties here
  logError(errorMsg) {
    console.error(`[${this.constructor.name} ERROR]: ${errorMsg}`);
  }
};
// 2. Define classes that will consume the mixin
class User {
  constructor(name) {
    this.name = name;
  }
  greet() {
    this.log(`Hello, I am ${this.name}.`);
    // Uses the mixed-in log method
  }
}
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }
  displayPrice() {
    this.log(`The price of ${this.name} is $${this.price}.`);
    // Uses the mixed-in log method
  }
}
// 3. Apply the Mixin to the prototypes of the classes
// Object.assign copies properties from source objects to a target object.
// By assigning to the .prototype, all instances of User and Product will inherit these
methods.
Object.assign(User.prototype, LoggerMixin);
Object.assign(Product.prototype, LoggerMixin);
// Demonstrate usage
const user = new User("Alice");
user.greet(); // Expected: [User LOG]: Hello, I am Alice.
user.log("User specific activity.");
// Directly calling the mixed-in method
user.logError("Failed to fetch user data.");
// Using another mixed-in method
```

```
const product = new Product("Laptop", 1200);
product.displayPrice();
// Expected: [Product LOG]: The price of Laptop is $1200.
product.log("Product added to cart.");
// Directly calling the mixed-in method
```

Exercise 98: Module Pattern (Revealing Module Pattern)

```
// Exercise 98: Module Pattern (Revealing Module Pattern)
const ShoppingCart = (function() { // Immediately Invoked Function Expression (IIFE)
  // Private variables (not accessible from outside the IIFE)
  let items = [];
  let nextItemId = 1;

  // Private helper function
  function _findItemIndex(itemName) {
    return items.findIndex(item => item.name === itemName);
  }

  // Public methods that are exposed
  function addItem(name, price, quantity = 1) {
    const existingIndex = _findItemIndex(name);
    if (existingIndex > -1) {
      items[existingIndex].quantity += quantity;
      console.log(`Updated quantity for "${name}".`);
    } else {
      items.push({ id: nextItemId++, name, price, quantity });
      console.log(`Added "${name}" to cart.`);
    }
  }

  function removeItem(name) {
    const index = _findItemIndex(name);
    if (index > -1) {
      items.splice(index, 1);
      console.log(`Removed "${name}" from cart.`);
    } else {
```

```
    console.log(`"${name}" not found in cart.`);
  }
 }

 function getTotal() {
  return items.reduce((total, item) => total + (item.price * item.quantity), 0);
 }

 function getCartItems() {
  // Return a copy to prevent external modification of the private 'items' array
  return [...items];
 }

 function _clearCart() { // This could be a private helper not exposed
  items = [];
  nextItemId = 1;
  console.log("Cart cleared (private method).");
 }

 // The "revealing" part: expose public methods
 return {
  addItem: addItem,
  removeItem: removeItem,
  getTotal: getTotal,
  getItems: getCartItems, // Renaming for public interface
  // clear: _clearCart // Uncomment to expose a 'clear' method
 };
})(); // The IIFE is immediately executed, and its return value is assigned to
ShoppingCart

// Demonstrate using the ShoppingCart module
console.log("--- Shopping Cart Actions ---");
ShoppingCart.addItem("Laptop", 1200);
ShoppingCart.addItem("Mouse", 25, 2);
ShoppingCart.addItem("Keyboard", 75);
ShoppingCart.addItem("Laptop", 1200); // Add laptop again to update quantity
console.log("Current cart items:", ShoppingCart.getItems());
console.log("Cart total:", ShoppingCart.getTotal()); // Expected: 1200*2 + 25*2 + 75 =
2400 + 50 + 75 = 2525
```

```
ShoppingCart.removeItem("Mouse");
console.log("Cart total after removing Mouse:", ShoppingCart.getTotal()); // Expected:
2400 + 75 = 2475
console.log("Current cart items:", ShoppingCart.getItems());

// Trying to access private variables/methods directly (will fail)
// console.log(ShoppingCart.items); // Undefined
// ShoppingCart._findItemIndex("Laptop");
// TypeError
```

Exercise 99: Higher-Order Components (Conceptual)

```
// Exercise 99: Higher-Order Components (Conceptual)
// 1. Definition of a "Component" (as a simple function for this conceptual example)
// In React, this would be a React component (functional or class-based).
const UserProfileComponent = (props) => {
  console.log(`Rendering UserProfile for: ${props.userName}, Status:
${props.status}`);
  return `<div><h1>User Profile for ${props.userName}</h1><p>Status:
${props.status}</p></div>`;
};
// 2. The Higher-Order Component (HOC)
// A HOC is a function that takes a component (WrappedComponent)
// and returns a new component with enhanced functionality.
function withLoading(WrappedComponent) {
  // The HOC returns a new functional component
  return function EnhancedComponent(props) {
    if (props.isLoading) {
      console.log("Displaying loading state...");
      return "<div>Loading...</div>"; // Render loading UI
    } else if (props.error) {
      console.log("Displaying error state...");
      return `<div>Error: ${props.error.message || 'Something went wrong.'}</div>`; //
Render error UI
    } else {
      // If not loading and no error, render the original component with its props
      console.log("Rendering wrapped component.");
      // In a real React app, you'd use <WrappedComponent {...props} />
```

```
    // Here, we just call the function as a stand-in.
    return WrappedComponent(props);
   }
  };
}
// 3. Applying the HOC
// Create an enhanced version of UserProfileComponent
const EnhancedUserProfile = withLoading(UserProfileComponent);
// Demonstrate usage of the Enhanced Component
console.log("--- First Render (Loading) ---");
console.log(EnhancedUserProfile({ isLoading: true }));
// Expected: <div>Loading...</div>
console.log("\n--- Second Render (Data Loaded) ---");
console.log(EnhancedUserProfile({ isLoading: false, userName: "Alice", status: "Active"
}));
// Expected: <div><h1>User Profile for Alice</h1><p>Status: Active</p></div>
console.log("\n--- Third Render (Error) ---");
console.log(EnhancedUserProfile({ isLoading: false, error: new Error("Network failed")
}));
// Expected: <div>Error: Network failed</div>
console.log("\n--- Fourth Render (No Data, No Error) ---");
console.log(EnhancedUserProfile({ isLoading: false, userName: "Bob", status:
"Inactive" }));
```

Exercise 100: Web Workers (Conceptual)

```
// Exercise 100: Web Workers (Conceptual)
console.log("--- Web Workers Conceptual Example ---");
console.log("Web Workers allow JavaScript code to run in a background thread,
separate from the main execution thread.");
console.log("This prevents heavy computations from blocking the UI (main thread),
keeping the page responsive.");
console.log("They do NOT have direct access to the DOM or window object.");
console.log("Communication between the main thread and a worker happens via
messages.");
// --- Conceptual main.js code ---
console.log("\n--- Main Thread (simulated) ---");
console.log("Main thread: Starting a heavy calculation in a Web Worker...");
```

```
// Imagine worker.js contains the heavy computation logic
// const myWorker = new Worker('worker.js');
// In a real browser, this would create the worker
// Simulate worker communication (since we can't spawn actual workers here)
function simulateWorkerCommunication(workerFile, dataToPost) {
  console.log("Main thread: Sending data to simulated worker:", dataToPost);
  // Simulate the worker's onmessage
  setTimeout(() => {
    console.log("Simulated Worker: Received message from main thread.");
    // Simulate the heavy calculation
    const result = performHeavyCalculation(dataToPost.data); // Calls the heavy
function defined below
    console.log("Simulated Worker: Sending result back to main thread.");
    // Simulate the worker's postMessage
    setTimeout(() => {
      console.log("Main thread: Received result from simulated worker:", result);
      console.log("Main thread: UI remains responsive during heavy calculation.");
    }, 50); // Small delay to simulate message passing overhead
  }, 100);
}
// --- Heavy Calculation (would typically be in worker.js) ---
function performHeavyCalculation(num) {
  console.log("Worker: Performing heavy calculation...");
  let sum = 0;
  // This loop will block the worker thread, but not the main browser UI thread
  for (let i = 0; i < num; i++) {
    sum += Math.sqrt(i) * Math.log(i + 1);
    // A somewhat CPU-intensive calculation
  }
  return `Calculated sum of ${sum.toFixed(2)} for ${num} iterations.`;
}
// Start the simulated heavy computation
simulateWorkerCommunication("worker.js", { type: 'startCalculation', data: 10000000
});
// A large number for computation
console.log("Main thread: This message appears immediately, demonstrating
non-blocking behavior.");
console.log("Main thread: User can interact with the page while calculation is in
progress.");
```

```
// Benefits of Web Workers:
console.log("\nBenefits of Web Workers:");
console.log("- **Improved UI Responsiveness:** Prevents the main thread from freezing during long-running tasks.");
console.log("- **Parallel Execution:** Achieves a form of multi-threading in the browser.");
console.log("- **Better Performance:** Distributes workload across threads.");
```