

30 JavaScript Coding Exercises

30 JavaScript Coding Exercises	1
Section 4: Intermediate to Advanced Concepts (Exercises 31-40)	2
Exercise 31: Error Handling with try...catch	2
Exercise 32: Understanding this Keyword Context	3
Exercise 33: Arrow Functions (=>)	4
Exercise 34: Object Destructuring	6
Exercise 35: Array Destructuring	8
Exercise 36: Spread Operator (...) - Arrays	9
Exercise 37: Spread Operator (...) - Objects	11
Exercise 38: Ternary Operator (Conditional Operator)	12
Exercise 39: Nullish Coalescing Operator (??)	13
Exercise 40: Optional Chaining (?.)	15
Section 5: More Advanced Concepts (Exercises 41-50)	17
Exercise 41: Asynchronous JavaScript - async/await	17
Exercise 42: Classes and Inheritance	19
Exercise 43: Static Methods and Properties	21
Exercise 44: Getters and Setters	23
Exercise 45: Array Method: some()	25
Exercise 46: Array Method: every()	26
Exercise 47: Array Method: find() and findIndex()	27
Exercise 48: Set Data Structure	29
Exercise 49: Map Data Structure	31
Exercise 50: localStorage (Basic Persistence)	33
Section 6: Problem Solving & Algorithms (Exercises 51-60)	35
Exercise 51: Palindrome Checker	35
Exercise 52: Anagram Checker	36
Exercise 53: FizzBuzz	38
Exercise 54: Remove Duplicates from an Array	39
Exercise 55: Count Character Occurrences	40
Exercise 56: Merge Two Sorted Arrays	41
Exercise 57: Find Missing Number in a Sequence	43
Exercise 58: Flatten an Array	44
Exercise 59: Implement a Simple Queue	46
Exercise 60: Implement a Simple Stack	49

This set of exercises continues your journey through JavaScript, covering more advanced concepts and practical problem-solving techniques. Keep practicing and

experimenting!

Section 4: Intermediate to Advanced Concepts (Exercises 31-40)

Exercise 31: Error Handling with try...catch

- **Skills Highlighted:** Error handling, try block, catch block, Error object, custom error messages.
- **Problem Statement:** Create a function `divide(a, b)` that divides `a` by `b`. Use a `try...catch` block to handle cases where `b` is 0. If `b` is 0, throw an Error with the message "Cannot divide by zero". Print the result or the error message.
- **Step-by-step Solution:**
 1. Define the `divide` function with `a` and `b` parameters.
 2. Inside the function, wrap the division logic in a try block.
 3. Within the try block, check if `b` is 0. If so, throw a new Error.
 4. If `b` is not 0, return `a / b`.
 5. Implement a catch block that receives the error.
 6. Inside the catch block, print the error message (e.g., `error.message`).
 7. Call the function with both valid and invalid denominators.

- **Code:**

```
// 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
```

- **Explanation:**

- The **try block** contains code that might throw an error.
- If an error occurs within the try block, execution immediately jumps to the **catch block**.
- The catch block receives the Error object (conventionally named error or err), which contains information about the error (like error.message).
- throw new Error(...) creates and throws a custom error, stopping the current function's execution.
- This pattern is crucial for gracefully handling unexpected situations and preventing your program from crashing.

Exercise 32: Understanding this Keyword Context

- **Skills Highlighted:** this keyword, object methods, function context.
- **Problem Statement:** Create an object calculator with a property value = 0. Add two methods: add(num) that adds num to value and returns this (for chaining), and getResult() that returns the current value. Demonstrate adding numbers using method chaining.
- **Step-by-step Solution:**
 1. Define the calculator object.
 2. Add value property.
 3. Add add(num) method: this.value += num; return this;
 4. Add getResult() method: return this.value;
 5. Chain calls: calculator.add(5).add(10).add(20).getResult().

- **Code:**

```
// 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

```

- **Explanation:**

- The `this` keyword in JavaScript refers to the object that is *currently executing* the function. Its value depends on *how* a function is called.
- In the `add` and `getResult` methods, when called as `calculator.add(...)` or `calculator.getResult()`, `this` correctly points to the `calculator` object itself.
- Returning `this` from a method allows you to "chain" multiple method calls together, making the code more fluid and readable.

Exercise 33: Arrow Functions (=>)

- **Skills Highlighted:** Arrow functions, concise syntax, lexical `this` binding.
- **Problem Statement:** Rewrite the `forEach` and `map` examples from previous exercises using arrow functions for their callbacks. Additionally, create a simple person object with a `greetDelayed` method that uses `setTimeout` and an arrow function to log a delayed greeting, demonstrating arrow functions' `this` binding.
- **Step-by-step Solution:**
 1. Rewrite a `forEach` loop with an arrow function.
 2. Rewrite a `map` operation with an arrow function.

3. Create a person object with name and greetDelayed method.
4. Inside greetDelayed, use setTimeout with an arrow function for the callback.
5. Observe how this behaves correctly with the arrow function in the delayed context.

- **Code:**

```
// 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();
```

- **Explanation:**

- **Concise Syntax:** Arrow functions offer a shorter way to write functions, especially for callbacks.
 - `param => expression` (for single parameter, single expression)
 - `(param1, param2) => { ... }` (for multiple parameters or block body)
- **Lexical this:** This is the most significant difference. Arrow functions do not have their own `this` context. Instead, they inherit this from the surrounding (lexical) scope where they are defined. This solves a common problem where `this` would change unexpectedly inside callbacks defined with regular functions, especially with `setTimeout` or event handlers.

Exercise 34: Object Destructuring

- **Skills Highlighted:** Object destructuring, extracting properties, default values, renaming properties.
- **Problem Statement:** Given an object `movie = { title: "Inception", director: "Christopher Nolan", year: 2010, rating: 8.8 }`.
 1. Use object destructuring to extract `title` and `director` into separate variables.
 2. Use object destructuring to extract `year` and `rating`, but rename `rating` to `imdbRating`.
 3. Try to extract a non-existent property `genre` and provide a default value of `"Sci-Fi"`.Print all extracted values.

- **Step-by-step Solution:**

1. Declare `movie` object.
2. Use `{ title, director } = movie;`
3. Use `{ year, rating: imdbRating } = movie;`
4. Use `{ genre = "Sci-Fi" } = movie;`
5. Print results.

- **Code:**

```
// 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

```

- **Explanation:**

- **Object Destructuring:** A convenient way to extract properties from objects and bind them to variables.
- `const { prop1, prop2 } = obj;` creates `prop1` and `prop2` variables with values from `obj.prop1` and `obj.prop2`.
- **Renaming:** `const { oldName: newName } = obj;` allows you to extract a property but assign it to a variable with a different name.
- **Default Values:** `const { prop = defaultValue } = obj;` assigns `defaultValue` if the property is undefined or not present in the object.
- Destructuring is frequently used in function parameters to directly access

specific properties of an object passed as an argument.

Exercise 35: Array Destructuring

- **Skills Highlighted:** Array destructuring, extracting elements, skipping elements, rest pattern.
- **Problem Statement:** Given an array `rgb = ["red", "green", "blue", "alpha"]`.
 1. Use array destructuring to extract the first two elements (`firstColor`, `secondColor`).
 2. Use array destructuring to extract the third element, skipping the first two.
 3. Use array destructuring to extract the first element and collect the rest into a new array `remainingColors`.
Print all extracted values.
- **Step-by-step Solution:**
 1. Declare `rgb` array.
 2. Use `const [firstColor, secondColor] = rgb;`
 3. Use `const [, thirdColor] = rgb;` (with commas to skip).
 4. Use `const [first, ...remainingColors] = rgb;` (rest pattern).
 5. Print results.

- **Code:**

```
// 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
```

- **Explanation:**

- **Array Destructuring:** A syntax for extracting elements from arrays and assigning them to variables.
- `const [elem1, elem2] = arr;` creates variables `elem1` and `elem2` from the first and second elements of `arr`.
- **Skipping Elements:** Use empty commas `,` to skip elements you don't want to extract.
- **Rest Pattern (...):** When used in array destructuring, it collects all remaining elements into a new array. It must be the last element in the destructuring pattern.
- A common trick is to use array destructuring for easily swapping variable values.

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

- **Skills Highlighted:** Spread operator, array copying, array concatenation, passing array elements as arguments.
- **Problem Statement:**
 1. Create an array `arr1 = [1, 2, 3]` and `arr2 = [4, 5, 6]`. Use the spread operator to create a new array `combinedArr` that contains all elements of `arr1` and `arr2`.
 2. Create a copy of `arr1` named `arr1Copy` using the spread operator.
 3. Create a function `sumAll(a, b, c)` that takes three arguments and sums them. Use the spread operator to pass the elements of `arr1` to `sumAll`.
- **Step-by-step Solution:**
 1. Declare `arr1` and `arr2`.
 2. `const combinedArr = [...arr1, ...arr2];`
 3. `const arr1Copy = [...arr1];`
 4. Define `sumAll(a, b, c)`.
 5. `console.log(sumAll(...arr1));`
- **Code:**

```
// 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
```

- **Explanation:**

- The **spread operator (...)** "expands" an iterable (like an array) into its individual elements.
- **Array Concatenation:** It's a clean way to combine arrays or insert elements.
- **Shallow Copying:** It creates a new array with the same elements. For arrays

of primitive values (numbers, strings), this is a deep copy. For arrays of objects, it's a shallow copy (the objects themselves are still referenced).

- **Function Arguments:** It allows you to pass an array's elements as separate arguments to a function, which is particularly useful for functions that expect multiple arguments (like `Math.max()` or `Math.min()`).

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

- **Skills Highlighted:** Spread operator, object copying, object merging.
- **Problem Statement:**
 1. Create an object `user = { name: "Jane", age: 28 }`. Create a *copy* of `user` named `userCopy` using the spread operator. Add a new property `city: "London"` to `userCopy`.
 2. Create an object `address = { street: "123 Main St", zip: "10001" }`. Use the spread operator to create a new object `userProfile` that merges `user` and `address`. If there are conflicting properties, the latter one wins.

- **Step-by-step Solution:**

1. Declare `user`.
2. `const userCopy = { ...user, city: "London" };`
3. Declare `address`.
4. `const userProfile = { ...user, ...address };`
5. Print results.

- **Code:**

```
// 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 }
```

- **Explanation:**

- The **spread operator (...)** for objects works similarly to arrays, expanding an object's properties into a new object.
- **Object Copying:** { ...obj } creates a shallow copy of an object.
- **Object Merging:** { ...obj1, ...obj2 } merges properties from obj1 and obj2 into a new object. If properties with the same key exist in both, the value from the *latter* object in the spread sequence will override the former.
- This is a very common and readable way to manage object state without direct mutation.

Exercise 38: Ternary Operator (Conditional Operator)

- **Skills Highlighted:** Ternary operator, concise conditional assignments.
- **Problem Statement:** Declare a variable temperature and set it to a number. Use the ternary operator to assign a string to a weatherStatus variable: if temperature is greater than 25, it should be "Hot"; otherwise, "Cold". Print weatherStatus.
- **Step-by-step Solution:**
 1. Declare temperature.
 2. const weatherStatus = (temperature > 25) ? "Hot" : "Cold";
 3. Print weatherStatus.
- **Code:**

```
// 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!
```

- **Explanation:**

- The **ternary operator** (also called the conditional operator) is a shorthand for a simple if-else statement.
- Syntax: condition ? expressionIfTrue : expressionIfFalse
- It's useful for assigning values conditionally in a single line, making the code more compact. However, for complex conditions, a full if-else statement is usually more readable.

Exercise 39: Nullish Coalescing Operator (??)

- **Skills Highlighted:** Nullish coalescing operator, default values, handling null and undefined.
- **Problem Statement:**
 1. Declare a variable `userName` and set it to null.
 2. Declare a variable `defaultName` = "Guest".
 3. Use the nullish coalescing operator to assign `userName` to `displayName`, falling back to `defaultName` if `userName` is null or undefined.
 4. Repeat with `userName` set to an empty string ("") and a 0. Observe the difference compared to `||`.
- **Step-by-step Solution:**
 1. `let userName = null;`

2. `let defaultName = "Guest";`
3. `const displayName = userName ?? defaultName;`
4. Test with `userName = ""`, `userName = 0`.
5. Compare with `||` (logical OR) operator.

- **Code:**

```
// 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)
```

- **Explanation:**

- The **nullish coalescing operator (??)** provides a default value only when the left-hand side operand is null or undefined.
- This is distinct from the **logical OR operator (||)**, which provides a default value for *any falsy* value (including 0, empty strings "", and false).
- ?? is useful when you want to treat 0, false, or "" as valid values and only provide a fallback if the variable is truly missing (null or undefined).

Exercise 40: Optional Chaining (?.)

- **Skills Highlighted:** Optional chaining, safe property access, preventing runtime errors.
- **Problem Statement:** Given a user object that might have nested address or address.street properties, which could be undefined. Use optional chaining to safely access and print user.address?.street and user.company?.name. If a property doesn't exist, it should return undefined without throwing an error.
- **Step-by-step Solution:**
 1. Declare a user object with name, email, and optionally address with street.
 2. Use user.address?.street to access the nested property safely.
 3. Use user.company?.name to access a potentially non-existent property chain.
 4. Print the results. Compare with direct access that would throw an error.
- **Code:**

```
// 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)
```

- **Explanation:**
 - **Optional Chaining (?.)** allows you to safely access properties of an object that might be null or undefined without causing a runtime error.
 - If any part of the chain before ?. is null or undefined, the expression immediately evaluates to undefined instead of throwing a TypeError.
 - This is incredibly useful for working with data where the structure is not guaranteed or when dealing with API responses where certain fields might be missing.
 - It can also be used for safe access to array elements (arr?.[index]) or methods (obj.method?.()).

Section 5: More Advanced Concepts (Exercises 41-50)

Exercise 41: Asynchronous JavaScript - async/await

- **Skills Highlighted:** async function, await keyword, Promises, cleaner asynchronous code.
- **Problem Statement:** Rewrite the fetchUserDataPromise function from Exercise 27 using async/await. Create a new function displayUserAsync(userId) that uses await to get the user data and then prints it. Handle potential errors using try...catch within the async function.
- **Step-by-step Solution:**
 1. Define an async function fetchUserDataAsync(userId) that wraps the Promise logic. It will still return a Promise.
 2. Define an async function displayUserAsync(userId).
 3. Inside displayUserAsync, use try...catch.
 4. Within try, const user = await fetchUserDataAsync(userId);
 5. Print user data.
 6. In catch, log the error.
 7. Call displayUserAsync for success and error cases.
- **Code:**

```
// 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.");
```

- **Explanation:**

Laurence Svekis Learn More <https://basescripts.com/>

- **async keyword:** Used to declare an asynchronous function. An async function implicitly returns a Promise.
- **await keyword:** Can *only* be used inside an async function. It pauses the execution of the async function until the Promise it's awaiting resolves (or rejects).
- If the Promise resolves, await returns the resolved value. If it rejects, await throws the error, which can then be caught by a try...catch block.
- async/await makes asynchronous code look and behave more like synchronous code, greatly improving readability and maintainability compared to deeply nested callbacks (callback hell).

Exercise 42: Classes and Inheritance

- **Skills Highlighted:** ES6 Classes, constructor, extends keyword, super(), method overriding.
- **Problem Statement:**
 1. Create a Shape class with a constructor that takes color and a method displayColor() that prints "The shape color is [color]".
 2. Create a Circle class that extends Shape. Its constructor should also take radius.
 3. Override the displayColor() method in Circle to print "The circle color is [color]". Add a new method calculateArea() to Circle that returns its area ($\pi * r^2$).
 4. Create instances of both Shape and Circle and call their methods.
- **Step-by-step Solution:**
 1. Define class Shape { ... }.
 2. Define class Circle extends Shape { ... }.
 3. In Circle's constructor, call super(color) and assign this.radius.
 4. Override displayColor in Circle.
 5. Add calculateArea to Circle.
 6. Create objects and test.

- **Code:**

```
// 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. \n This is a  
rectangle.  
console.log("Rectangle Area:", myRectangle.calculateArea()); // Expected: 24
```

- **Explanation:**

- **Classes (class keyword):** Syntactic sugar over JavaScript's prototype-based inheritance, making it look more like traditional object-oriented programming.
- **constructor:** A special method for creating and initializing an object created with a class.
- **Inheritance (extends keyword):** A class can inherit properties and methods from another class.
- **super():** In a child class's constructor, `super()` must be called *before* this is used. It calls the parent class's constructor. In methods, `super.methodName()` calls the parent's version of that method.
- **Method Overriding:** A child class can provide its own implementation of a method that is already defined in its parent class.

Exercise 43: Static Methods and Properties

- **Skills Highlighted:** Static methods, static properties, class-level functionality.
- **Problem Statement:** Create a `MathHelper` class. Add a static method `add(a, b)` that returns the sum of `a` and `b`. Add a static property `PI` with the value `3.14159`. Demonstrate calling the static method and accessing the static property directly on the class, without creating an instance.
- **Step-by-step Solution:**

1. Define class MathHelper { ... }.
2. Add static add(a, b) { ... }.
3. Add static PI = 3.14159; (or use static get PI() { return 3.14159; }).
4. Call MathHelper.add(x, y).
5. Access MathHelper.PI.

- **Code:**

// 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();
```

- **Explanation:**

- **Static methods and properties** belong to the class itself, not to instances of the class.
- They are defined using the static keyword.
- You access them directly on the class (e.g., `ClassName.staticMethod()`, `ClassName.staticProperty`).
- They are useful for utility functions that don't depend on the state of a specific object instance (e.g., helper functions, factory methods, constants).

Exercise 44: Getters and Setters

- **Skills Highlighted:** Getters, setters, property accessors, controlling property access.
- **Problem Statement:** Create a Product class with a `_price` property (conventionally indicating it's intended for internal use). Implement a getter `price` that returns `_price` and a setter `price` that validates if the new value is a positive number before setting `_price`. If not valid, log an error. Create an instance and test.

- **Step-by-step Solution:**

1. Define class Product { ... }.
2. In constructor, initialize `this._price`.
3. Define `get price()` { return `this._price`; }.
4. Define `set price(newPrice)` { ... }: add validation, then `this._price = newPrice`;
5. Create `new Product()`.
6. Access `product.price` (getter) and `product.price = value` (setter).

- **Code:**

```
// 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

```

- **Explanation:**

Laurence Svekis Learn More <https://basescripts.com/>

- **Getters (get propName()):** Methods that are accessed like properties. They allow you to control how a property's value is retrieved, enabling computations, formatting, or side effects upon reading.
- **Setters (set propName(value)):** Methods that are accessed like properties when assigned a value. They allow you to control how a property's value is modified, enabling validation, data transformation, or updating related properties.
- They provide a way to add logic around accessing and modifying object properties, without explicitly calling methods (e.g., `product.setPrice(100)` vs `product.price = 100`).

Exercise 45: Array Method: some()

- **Skills Highlighted:** `some()` method, checking for existence, early exit.
- **Problem Statement:** Given an array of numbers `grades = [60, 75, 80, 90, 55]`. Use the `some()` method to check if *at least one* grade is greater than or equal to 90. Print the boolean result.
- **Step-by-step Solution:**
 1. Declare `grades` array.
 2. Call `grades.some()`.
 3. Provide a callback that returns true if `grade >= 90`.
 4. Store and print the boolean result.
- **Code:**

```
// 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
```

- **Explanation:**

- The **some() method** checks if *at least one* element in an array satisfies the condition provided by the callback function.
- It returns true as soon as the callback returns true for any element (it "short-circuits"). If no element satisfies the condition, it returns false.

Exercise 46: Array Method: every()

- **Skills Highlighted:** every() method, checking all elements, early exit.
- **Problem Statement:** Given an array of ages ages = [22, 28, 35, 40]. Use the every() method to check if *all* ages are greater than or equal to 18. Print the boolean result.
- **Step-by-step Solution:**
 1. Declare ages array.
 2. Call ages.every().
 3. Provide a callback that returns true if age >= 18.
 4. Store and print the boolean result.
- **Code:**

```
// 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
```

- **Explanation:**

- The **every() method** checks if *all* elements in an array satisfy the condition provided by the callback function.
- It returns false as soon as the callback returns false for any element (it "short-circuits"). If all elements satisfy the condition, it returns true.

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

- **Skills Highlighted:** find() method, findIndex() method, searching arrays for specific elements/indices.
- **Problem Statement:** Given an array of users `users = [{id: 1, name: "Alice"}, {id: 2, name: "Bob"}, {id: 3, name: "Charlie"}]`.
 1. Use find() to get the user object with id: 2.
 2. Use findIndex() to get the index of the user with name: "Charlie".
 3. Try to find a user that doesn't exist and observe the result.
- **Step-by-step Solution:**

1. Declare users array.
2. `users.find(user => user.id === 2);`
3. `users.findIndex(user => user.name === "Charlie");`
4. Test with a non-existent search.

- **Code:**

```
// 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
```

- **Explanation:**

- **find():** Returns the **first element** in the array that satisfies the provided

- testing function. If no elements satisfy the condition, undefined is returned.
- **findIndex():** Returns the **index** of the first element in the array that satisfies the provided testing function. If no elements satisfy the condition, -1 is returned.
- Both methods are very efficient for searching, as they stop iterating as soon as a match is found.

Exercise 48: Set Data Structure

- **Skills Highlighted:** Set object, adding elements, checking for existence, removing duplicates, iteration.
- **Problem Statement:** Create a Set named uniqueNumbers. Add the numbers 1, 2, 3, 2, 4, 1 to it.
 1. Print the size of the Set.
 2. Check if uniqueNumbers contains 3.
 3. Remove 2 from the Set.
 4. Iterate over the Set and print its elements.
- **Step-by-step Solution:**
 1. `const uniqueNumbers = new Set();`
 2. Use `uniqueNumbers.add(...)`.
 3. `uniqueNumbers.size`.
 4. `uniqueNumbers.has(3)`.
 5. `uniqueNumbers.delete(2)`.
 6. Use `for...of` or `forEach` to iterate.

- **Code:**

```
// 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]

```

- **Explanation:**

- A **Set** is a collection of unique values. It can store any data type, whether primitive values or object references.
- **Uniqueness:** Duplicate values are ignored when added to a Set.
- Methods:
 - `new Set()`: Creates a new Set.
 - `set.add(value)`: Adds a new element to the Set.
 - `set.has(value)`: Returns a boolean indicating whether an element is present.
 - `set.delete(value)`: Removes an element.
 - `set.clear()`: Removes all elements.
 - `set.size`: A property that returns the number of elements in the Set.
- Sets are commonly used for efficiently removing duplicates from arrays or

checking for the presence of elements.

Exercise 49: Map Data Structure

- **Skills Highlighted:** Map object, key-value pairs, adding, getting, deleting, iterating.
- **Problem Statement:** Create a Map named userRoles. Add the following key-value pairs:
 - "Alice": "Admin"
 - "Bob": "Editor"
 - "Charlie": "Viewer"
 1. Get the role of "Alice".
 2. Check if "David" exists in the Map.
 3. Set a new role for "Bob" to "Moderator".
 4. Delete "Charlie" from the Map.
 5. Iterate over the Map and print all key-value pairs.
- **Step-by-step Solution:**
 1. `const userRoles = new Map();`
 2. `userRoles.set(key, value);`
 3. `userRoles.get("Alice");`
 4. `userRoles.has("David");`
 5. `userRoles.set("Bob", "Moderator");`
 6. `userRoles.delete("Charlie");`
 7. Use `for...of` with `map.entries()`, `map.keys()`, `map.values()`.

- **Code:**

```
// 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} `);
}
```

- **Explanation:**

- A **Map** is a collection of keyed data items, similar to objects, but it allows keys of *any* type (not just strings or Symbols).
- **Ordered:** Elements are iterated in the order they were inserted.
- **Methods:**
 - `new Map()`: Creates a new Map.
 - `map.set(key, value)`: Stores a key-value pair.
 - `map.get(key)`: Returns the value associated with the key, or undefined if the key doesn't exist.
 - `map.has(key)`: Returns a boolean indicating whether the key exists.
 - `map.delete(key)`: Removes the entry by key.

- `map.clear()`: Removes all entries.
 - `map.size`: A property that returns the number of entries.
- Maps are particularly useful when you need to use non-string keys (like objects or functions) or when the order of elements is important.

Exercise 50: localStorage (Basic Persistence)

- **Skills Highlighted:** localStorage, storing data, retrieving data, JSON conversion.
- **Problem Statement:**
 1. Store your `userName` (from Exercise 1) in localStorage under the key `"myUserName"`.
 2. Retrieve `userName` from localStorage and print it.
 3. Store an object `{ setting: "dark", notifications: true }` in localStorage under the key `"userSettings"`. Remember that localStorage only stores strings.
 4. Retrieve `userSettings` and parse it back into an object, then print it.
 5. Remove `"myUserName"` from localStorage.
- **Step-by-step Solution:**
 1. `localStorage.setItem("myUserName", "Your Name");`
 2. `localStorage.getItem("myUserName");`
 3. `JSON.stringify()` the object before `setItem()`.
 4. `JSON.parse()` the retrieved string after `getItem()`.
 5. `localStorage.removeItem("myUserName");`
- **Code:**

```
// 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.");
}

```

- **Explanation:**

- **localStorage:** A web API that allows web applications to store data persistently in the browser, with no expiration date. Data stored in localStorage remains until explicitly deleted by the user or the application.
- **Key-Value Store:** It works like a simple dictionary, storing data as key-value pairs.
- **String-only:** localStorage can *only* store strings.
 - To store numbers, booleans, or objects, you must convert them to a JSON

- string using `JSON.stringify()` before storing.
- When retrieving, you must parse the JSON string back into a JavaScript object using `JSON.parse()`.
 - **`localStorage.setItem(key, value)`**: Stores a value.
 - **`localStorage.getItem(key)`**: Retrieves a value. Returns null if the key doesn't exist.
 - **`localStorage.removeItem(key)`**: Deletes a specific key-value pair.
 - **`localStorage.clear()`**: Deletes all key-value pairs for the current origin.
 - **Note**: `localStorage` is synchronous and can block the main thread for large amounts of data. For more complex persistent storage, IndexedDB is often preferred.

Section 6: Problem Solving & Algorithms (Exercises 51-60)

Exercise 51: Palindrome Checker

- **Skills Highlighted**: String manipulation, loops, conditional logic, `toLowerCase()`, `replace()`.
- **Problem Statement**: Create a function `isPalindrome(str)` that takes a string and returns true if it's a palindrome (reads the same forwards and backwards, ignoring case and non-alphanumeric characters), and false otherwise.
- **Step-by-step Solution**:
 1. Define `isPalindrome(str)`.
 2. Convert the string to lowercase and remove non-alphanumeric characters (e.g., regex `/[^a-z0-9]/g`).
 3. Compare the cleaned string with its reversed version.
 4. Return the boolean result.

- **Code:**

```
// 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)
```

- **Explanation:**

- The core idea is to normalize the string (remove spaces, punctuation, make it lowercase) and then compare it to its reversed self.
- `toLowerCase()`: Converts the string to all lowercase characters.
- `replace(/[^a-z0-9]/g, "")`: Uses a regular expression to remove any character that is NOT (^) a lowercase letter (a-z) or a digit (0-9). The `g` flag ensures all occurrences are replaced.
- `split("").reverse().join("")`: A common pattern for reversing a string.

Exercise 52: Anagram Checker

- **Skills Highlighted:** String manipulation, sorting, conditional logic, `toLowerCase()`, `replace()`, `split()`, `sort()`, `join()`.
- **Problem Statement:** Create a function `areAnagrams(str1, str2)` that takes two strings and returns true if they are anagrams of each other (contain the same characters with the same counts, ignoring case and spaces), and false otherwise.
- **Step-by-step Solution:**
 1. Define `areAnagrams(str1, str2)`.
 2. Create a helper function `cleanAndSort(str)`:
 - Convert `str` to lowercase.
 - Remove spaces and other non-alphanumeric characters.
 - Split into an array of characters.
 - Sort the character array.
 - Join back into a string.
 3. Apply `cleanAndSort` to both `str1` and `str2`.

4. Compare the resulting sorted strings.

- **Code:**

// 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
```

- **Explanation:**

- The core idea for checking anagrams is that if two strings are anagrams, their sorted character representations will be identical.
- The cleanAndSortString helper function normalizes the input strings by making them lowercase, removing non-alphanumeric characters, splitting them into character arrays, sorting the arrays, and then joining them back into strings.

- Comparing these normalized and sorted strings efficiently determines if they are anagrams.

Exercise 53: FizzBuzz

- **Skills Highlighted:** Loops, modulo operator (%), multiple conditional checks.
- **Problem Statement:** Write a program that prints numbers from 1 to 100. For multiples of 3, print "Fizz" instead of the number. For multiples of 5, print "Buzz". For numbers that are multiples of both 3 and 5, print "FizzBuzz". Otherwise, print the number.
- **Step-by-step Solution:**
 1. Use a for loop from $i = 1$ to 100.
 2. Inside the loop, use if-else if-else statements.
 3. Check $(i \% 3 === 0 \ \&\& \ i \% 5 === 0)$ first (most specific condition).
 4. Then check $(i \% 3 === 0)$.
 5. Then check $(i \% 5 === 0)$.
 6. Finally, else print i .

- **Code:**

```
// 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
```

- **Explanation:**
 - The **modulo operator (%)** returns the remainder of a division. If number %

divisor === 0, it means number is a multiple of divisor.

- The order of if-else if-else conditions is crucial. The most specific condition (multiples of both 3 and 5) must be checked first, otherwise, numbers like 15 would only print "Fizz" or "Buzz".
- The provided solution uses a slightly more elegant approach: build the output string and if it's empty, fall back to the number using the || operator (which takes the first truthy value).

Exercise 54: Remove Duplicates from an Array

- **Skills Highlighted:** Array manipulation, Set (preferred), filter() (alternative), loops.
- **Problem Statement:** Create a function removeDuplicates(arr) that takes an array and returns a new array with all duplicate elements removed. The order of the remaining unique elements should be preserved.
- **Step-by-step Solution (using Set):**
 1. Define removeDuplicates(arr).
 2. Create a new Set from the input array new Set(arr). This automatically handles uniqueness and preserves insertion order.
 3. Convert the Set back into an array using the spread operator [...set].
 4. Return the new array.
- **Code:**

```
// 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]));
```

- **Explanation:**

- The most efficient and modern way to remove duplicates while preserving order is to convert the array to a Set (which automatically handles uniqueness) and then convert it back to an array using the spread operator.
- The filter() and indexOf() method works by keeping an element only if its first occurrence is at its current index, meaning it's the first time it has appeared. While functional, it's less efficient for large arrays due to indexOf scanning the array multiple times.

Exercise 55: Count Character Occurrences

- **Skills Highlighted:** String iteration, objects (hash maps), counting, conditional logic.
- **Problem Statement:** Create a function countChars(str) that takes a string and returns an object where keys are characters and values are their counts (case-insensitive).
- **Step-by-step Solution:**
 1. Define countChars(str).
 2. Initialize an empty object charCounts = {}.
 3. Convert the string to lowercase.
 4. Loop through each character of the string.
 5. For each character:
 - If it's already a key in charCounts, increment its value.
 - Otherwise, add it as a new key with value 1.
 6. Return charCounts.
- **Code:**

```
// 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 }

```

- **Explanation:**

- This problem uses an object (often called a **hash map** or dictionary) to store the counts. The characters are used as keys, and their counts are the values.
- `charCounts[char] = (charCounts[char] || 0) + 1;` is a common idiom:
 - `charCounts[char] || 0`: If `charCounts[char]` is truthy (i.e., already exists and has a count), use its value. Otherwise (if it's undefined or 0), use 0.
 - Then `+ 1` increments the count.

Exercise 56: Merge Two Sorted Arrays

- **Skills Highlighted:** Arrays, loops, comparison, `push()`, two-pointer approach (efficient).
- **Problem Statement:** Create a function `mergeSortedArrays(arr1, arr2)` that takes two *sorted* arrays of numbers and returns a *single sorted array* containing all elements from both. Do not use `concat()` followed by `sort()`.
- **Step-by-step Solution:**
 1. Define `mergeSortedArrays(arr1, arr2)`.

2. Initialize an empty array merged = [].
3. Initialize two pointers, ptr1 = 0 (for arr1) and ptr2 = 0 (for arr2).
4. Use a while loop that continues as long as both ptr1 and ptr2 are within their respective array bounds.
5. Inside the loop, compare arr1[ptr1] and arr2[ptr2]:
 - If arr1[ptr1] is smaller, push arr1[ptr1] to merged and increment ptr1.
 - Else, push arr2[ptr2] to merged and increment ptr2.
6. After the main loop, one of the arrays might have remaining elements. Add any remaining elements from arr1 (if ptr1 < arr1.length) and arr2 (if ptr2 < arr2.length) to merged.
7. Return merged.

- **Code:**

// 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]

```

- **Explanation:**

- This problem uses a **two-pointer approach**, which is a common and efficient technique for problems involving sorted arrays.
- By maintaining separate pointers for each input array, you avoid unnecessary traversals and comparisons.
- The while loops ensure that all elements from both arrays are included in the final merged array, even if one array is exhausted before the other.

Exercise 57: Find Missing Number in a Sequence

- **Skills Highlighted:** Arrays, arithmetic series, sum calculation, loops.
- **Problem Statement:** Create a function `findMissingNumber(arr)` that takes an array of distinct numbers from 1 to n (inclusive, but with one number missing). The array is not necessarily sorted. Return the missing number. Assume the input array will always be valid (contains $n-1$ distinct numbers, one missing).
- **Step-by-step Solution:**
 1. Define `findMissingNumber(arr)`.
 2. Determine n : $n = arr.length + 1$.
 3. Calculate the expected sum of numbers from 1 to n using the arithmetic series formula: $sum = n * (n + 1) / 2$.
 4. Calculate the actual sum of elements in the given `arr`.
 5. The missing number is $expectedSum - actualSum$.
 6. Return the result.
- **Code:**
// 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)

```

- **Explanation:**

- This problem leverages the property of an **arithmetic series**. The sum of integers from 1 to n can be calculated directly.
- By comparing the expected sum (if no number were missing) with the actual sum of the given numbers, you can easily find the missing one.
- This approach is more efficient than sorting the array or using a hash set for large n.

Exercise 58: Flatten an Array

- **Skills Highlighted:** Recursion, array iteration, concat(), Array.isArray(), flat() (modern alternative).
- **Problem Statement:** Create a function flattenArray(arr) that takes an array possibly containing nested arrays and returns a new single-level array with all elements flattened. Do not use the built-in flat() method.
- **Step-by-step Solution (Recursive):**
 1. Define flattenArray(arr).
 2. Initialize an empty array flatArr = [].
 3. Loop through each element in arr.
 4. If the element is an array (Array.isArray(element)), recursively call flattenArray(element) and use concat() to add its flattened result to flatArr.
 5. Else (if it's not an array), push() the element directly to flatArr.
 6. Return flatArr.

- **Code:**

```
// 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));
```

- **Explanation:**

- This is a classic problem for demonstrating **recursion**.
- The **base case** implicitly occurs when an element is not an array; it's simply added to the flatArr.
- The **recursive step** is when an element *is* an array; the function calls itself on that nested array to flatten it further.
- `Array.isArray()` is used to reliably check if a variable is an array.
- `concat()` is used to merge arrays without mutating the original.
- The built-in `flat()` method (ES2019+) provides a much simpler solution for this specific problem, but implementing it manually helps understand recursion.

Exercise 59: Implement a Simple Queue

- **Skills Highlighted:** Data structures (Queue), array methods (push, shift), basic OOP/class.
- **Problem Statement:** Implement a Queue class with the following methods:
 - `enqueue(element)`: Adds an element to the back of the queue.
 - `dequeue()`: Removes and returns the element from the front of the queue. Returns undefined if the queue is empty.
 - `peek()`: Returns the element at the front of the queue without removing it. Returns undefined if empty.
 - `isEmpty()`: Returns true if the queue is empty, false otherwise.
 - `size()`: Returns the number of elements in the queue.
- **Step-by-step Solution:**
 1. Define class Queue { ... }.

2. In constructor, initialize `this.items = []`;
3. `enqueue: this.items.push(element)`;
4. `dequeue: this.items.shift()`;
5. `peek: this.items[0]`;
6. `isEmpty: this.items.length === 0`;
7. `size: this.items.length`;

- **Code:**

// 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

```

- **Explanation:**
 - A **Queue** is a **FIFO** (First-In, First-Out) data structure. Elements are added to the back and removed from the front, like a line of people.
 - The core operations are:
 - **enqueue:** Adds an element. `push()` adds to the end of an array.
 - **dequeue:** Removes the oldest element. `shift()` removes from the beginning of an array.
 - **peek:** Looks at the front element without removing it.
 - JavaScript arrays conveniently provide `push()` and `shift()` methods, making it straightforward to implement a queue. Be aware that `shift()` can be less performant for very large arrays as it re-indexes all subsequent elements. For production-grade queues with extremely large datasets, other implementations (e.g., linked lists) might be preferred.

Exercise 60: Implement a Simple Stack

- **Skills Highlighted:** Data structures (Stack), array methods (`push`, `pop`), basic OOP/class.
- **Problem Statement:** Implement a Stack class with the following methods:
 - `push(element)`: Adds an element to the top of the stack.
 - `pop()`: Removes and returns the element from the top of the stack. Returns undefined if the stack is empty.
 - `peek()`: Returns the element at the top of the stack without removing it. Returns undefined if empty.
 - `isEmpty()`: Returns true if the stack is empty, false otherwise.
 - `size()`: Returns the number of elements in the stack.
- **Step-by-step Solution:**
 1. Define class `Stack { ... }`.
 2. In constructor, initialize `this.items = []`;
 3. `push`: `this.items.push(element)`;
 4. `pop`: `this.items.pop()`;
 5. `peek`: `this.items[this.items.length - 1]`;
 6. `isEmpty`: `this.items.length === 0`;
 7. `size`: `this.items.length`;
- **Code:**

```
// 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

```

- **Explanation:**

- A **Stack** is a **LIFO** (Last-In, First-Out) data structure. Elements are added and removed from the "top" (the same end), like a stack of plates.
- The core operations are:
 - **push:** Adds an element to the top. `push()` adds to the end of an array.
 - **pop:** Removes the most recently added element. `pop()` removes from the end of an array.
 - **peek:** Looks at the top element without removing it.
- JavaScript arrays conveniently provide `push()` and `pop()` methods, making it straightforward to implement a stack. These operations are generally efficient

as they operate on the end of the array.