ES6 Tasks

Advance JS Task for Test

Global Scope

- Declare a variable using `var` outside of any function or block.
- Declare a variable using `let` outside of any function or block.
- Declare a variable using `const` outside of any function or block.
- Log all three variables to the console.
- Are they accessible globally?

Function Scope

- Create a function and declare a variable using `var` inside the function.
- Declare a variable using `let` inside the function.
- Declare a variable using `const` inside the function.
- Try to log all three variables to the console outside the function.
- What do you observe?

Block Scope:

- Use an `if` statement and declare a variable using `var` inside the block.
- Declare a variable using `let` inside the block.
- Declare a variable using `const` inside the block.
- Try to log all three variables to the console outside the block.
- What do you observe?

Hoisting with `var`:

- Write code where you log a `var` variable before it is declared.
- What value do you get?

Hoisting with `let` and `const`:

- Write code where you log a `let` variable before it is declared.
- Write code where you log a `const` variable before it is declared.
- What kind of error do you get?

Re-declaration:

- Try to declare the same variable name twice using `var`.
- Try to declare the same variable name twice using `let`.
- Try to declare the same variable name twice using `const`.
- What happens in each case?

Re-assignment:

- Declare a variable using `var` and assign it a value. Then reassign it a new value.
- Declare a variable using `let` and assign it a value. Then reassign it a new value.
- Declare a variable using `const` and assign it a value. Then reassign it a new value.
- What happens in each case?

Temporal Dead Zone (TDZ):

- Declare a `let` variable inside a block but try to log it before the declaration.
- Declare a `const` variable inside a block but try to log it before the declaration.
- What error do you get? Why?

When to use `var`, `let`, and `const`:

- Write a piece of code to demonstrate a good use case for `var`.
- Write a piece of code to demonstrate a good use case for `let`.
- Write a piece of code to demonstrate a good use case for `const`.

String Interpolation:

- Create variables for a person's first name and last name.
- Use a template literal to create a full name string and log it to the console.

Multi-line Strings:

- Use a template literal to create a multi-line string (e.g., an address) and log it to the console.

Simple Expressions:

- Create variables for two numbers.
- Use a template literal to create a string that includes the sum of the numbers.
- Log the string to the console.

Function Calls:

- Create a function that takes two numbers and returns their product.
- Use a template literal to call this function inside a string and log the result to the console.

Creating a Tagged Template:

- Write a simple tag function that takes a template string and logs it.
- Use this tag function with a template literal.

Formatting:

- Write a tag function that formats a string by making it uppercase.
- Use this tag function with a template literal and log the result.

Conditional Logic:

- Create a variable for the current hour.
- Use a template literal to display a different message depending on whether it's morning (before 12 PM) or afternoon (after 12 PM).

Loops within Template Literals:

- Create an array of items (e.g., a shopping list).
- Use a template literal to generate an HTML list (`<ul>` and `<li>` elements) from the array and log it to the console.

Escaping Backticks:

- Create a string that includes a backtick character using a template literal.
- Log the string to the console.

Nested Template Literals:

- Create nested template literals to build a more complex string, such as a nested HTML structure (e.g., a table with rows and cells).
- Log the result to the console.

Simple Condition:

- Create a variable `age`.
- Use the ternary operator to assign a variable `canVote` the value "Yes" if `age` is 18 or older, and "No" otherwise.
- Log `canVote` to the console.

Even or Odd:

- Create a variable `number`.
- Use the ternary operator to assign a variable `evenOrOdd` the value "Even" if `number` is even, and "Odd" if it's odd.
- Log `evenOrOdd` to the console.

Grade Evaluation:

Create a variable `score`.

Use the ternary operator to assign a variable `grade` based on the following conditions:

- "A" if `score` is 90 or above.
- "B" if `score` is 80 or above.
- "C" if `score` is 70 or above.
- "D" if `score` is 60 or above.
- "F" otherwise.

Log `grade` to the console.

Login Status:

- Create a variable `isLoggedIn`.
- Use the ternary operator and logical operators to assign a variable `statusMessage` the value "Welcome back!" if `isLoggedIn` is `true`, and "Please log in" if `isLoggedIn` is `false`.
- Log `statusMessage` to the console.

Discount Eligibility:

- Create variables `isMember` and `purchaseAmount`.
- Use the ternary operator and logical operators to assign a variable `discount` the value 10% of `purchaseAmount` if `isMember` is `true` and `purchaseAmount` is greater than 100, and 0 otherwise.
- Log `discount` to the console.

Determine Max Value:

- Create a function `maxValue(a, b)` that returns the larger of the two numbers using the ternary operator.
- Call the function with two numbers and log the result.

Greeting Message:

- Create a function `greet(name)` that returns a greeting message. If `name` is not provided (or is an empty string), it should return "Hello, guest!", otherwise, it should return "Hello, [name]!".
- Call the function with and without a name and log the result.

Mapping Values:

- Create an array of numbers.
- Use the `map` method with a ternary operator to create a new array where each number is doubled if it is even and tripled if it is odd.
- Log the new array to the console.

Filtering Values:

- Create an array of strings.
- Use the `filter` method with a ternary operator to create a new array that only includes strings with a length greater than 3.
- Log the new array to the console.

Copying an Array:

- Create an array `originalArray` with some elements.
- Use the spread operator to create a copy of `originalArray` called `copiedArray`.
- Log both arrays to the console to verify they are the same but not the same reference.

Merging Arrays:

- Create two arrays `array1` and `array2`.
- Use the spread operator to create a new array `mergedArray` that combines the elements of `array1` and `array2`.
- Log `mergedArray` to the console.
-

Adding Elements to an Array:

- Create an array `numbers` with some elements.
- Use the spread operator to add a new element at the beginning and at the end of the `numbers` array.
- Log the updated array to the console.

Copying an Object:

- Create an object `originalObject` with some key-value pairs.
- Use the spread operator to create a copy of `originalObject` called `copiedObject`.
- Log both objects to the console to verify they are the same but not the same reference.

Merging Objects:

- Create two objects `object1` and `object2` with some overlapping keys.
- Use the spread operator to create a new object `mergedObject` that combines the properties of `object1` and `object2`.
- Log `mergedObject` to the console and note which values are retained for the overlapping keys.

Updating Object Properties:

- Create an object `user` with properties `name`, `age`, and `email`.
- Use the spread operator to create a new object `updatedUser` that updates the `email` property and adds a new `address` property.
- Log the `updatedUser` object to the console.

Passing Array Elements as Arguments:

- Create a function `sum(a, b, c)` that returns the sum of three numbers.
- Create an array `numbers` with three elements.
- Use the spread operator to pass the elements of `numbers` as arguments to the `sum` function.
- Log the result to the console.

Combining Multiple Arrays:

- Create a function `combineArrays` that takes any number of arrays as arguments and returns a single array containing all elements.
- Use the spread operator inside the function to combine the arrays.
- Call the function with multiple arrays and log the result.

Rest Parameter with Spread Operator:

- Create a function `multiply` that takes a number and any number of additional arguments.
- Use the rest parameter to gather the additional arguments into an array and multiply each by the first argument.
- Return an array of the results.
- Call the function with appropriate arguments and log the result.

Spread Operator with Nested Structures:

- Create a nested array `nestedArray` and use the spread operator to create a shallow copy.
- Modify the inner arrays in the copied array.
- Log both the original and copied arrays to observe the effect of shallow copying.

Sum Function:

- Create a function `sum` that uses the rest operator to take any number of arguments.
- The function should return the sum of all its arguments.
- Call the function with different numbers of arguments and log the results.

Average Function:

- Create a function `average` that uses the rest operator to take any number of arguments.
- The function should return the average of all its arguments.
- Call the function with different numbers of arguments and log the results.

First and Rest:

- Create an array `numbers` with at least 5 elements.
- Use array destructuring with the rest operator to assign the first element to a variable `first` and the remaining elements to a variable `rest`.
- Log `first` and `rest` to the console.

Skip and Rest:

- Create an array `colors` with at least 5 elements.
- Use array destructuring with the rest operator to skip the first two elements and assign the remaining elements to a variable `remainingColors`.
- Log `remainingColors` to the console.

Basic Destructuring:

- Create an object `person` with properties `name`, `age`, `email`, and `address`.
- Use object destructuring with the rest operator to assign `name` and `email` to individual variables, and the remaining properties to a variable `rest`.
- Log the variables to the console.

Nested Destructuring:

- Create an object `student` with properties `id`, `name`, `grades`, and `info` (where `info` is another object with properties `age` and `major`).
- Use nested destructuring with the rest operator to extract `id`, `name`, and `major` to individual variables, and the remaining properties to a variable `rest`.
- Log the variables to the console.

Filter Even Numbers:

- Create a function `filterEven` that uses the rest operator to take any number of arguments.
- The function should return a new array containing only the even numbers.
- Call the function with different numbers of arguments and log the results.

Combine and Sort Arrays:

- Create a function `combineAndSort` that uses the rest operator to take any number of arrays.
- The function should combine all the arrays into one and return the sorted result.
- Call the function with different arrays and log the results.

Basic Destructuring:

- Create an array `fruits` with the elements "apple", "banana", and "cherry".
- Use destructuring to assign the first element to a variable `firstFruit`, the second to `secondFruit`, and the third to `thirdFruit`.
- Log the variables to the console.

Skipping Elements:

- Create an array `colors` with the elements "red", "green", "blue", "yellow".
- Use destructuring to assign the first element to `primaryColor` and the third element to `tertiaryColor`.
- Log the variables to the console.

Rest Operator:

- Create an array `numbers` with the elements 1 through 5.
- Use destructuring to assign the first element to `firstNumber` and the rest of the elements to `remainingNumbers`.
- Log the variables to the console.

Basic Destructuring:

- Create an object `person` with properties `name`, `age`, and `city`.
- Use destructuring to assign the properties to variables `name`, `age`, and `city`.
- Log the variables to the console.

Renaming Variables:

- Create an object `car` with properties `make`, `model`, and `year`.
- Use destructuring to assign the properties to variables `carMake`, `carModel`, and `carYear`.
- Log the variables to the console.

Default Values:

- Create an object `settings` with properties `theme` and `language`.
- Use destructuring to assign the properties to variables `theme` and `language`, and provide a default value of "English" for `language`.
- Log the variables to the console.

Array of Arrays:

- Create an array `nestedArray` with the elements `[1, 2]`, `[3, 4]`, and `[5, 6]`.
- Use nested destructuring to assign the first elements of each sub-array to variables `a`, `b`, and `c`.
- Log the variables to the console.

Object within an Object:

- Create an object `profile` with properties `username`, `details` (which is another object with properties `email` and `address`).
- Use nested destructuring to assign `username`, `email`, and `address` to variables.
- Log the variables to the console.

Mix of Arrays and Objects:

- Create an object `data` with properties `id`, `info` (which is an array with elements `{name: "Alice"}` and `{age: 25}`).
- Use nested destructuring to assign `id`, `name`, and `age` to variables.
- Log the variables to the console.

Array Parameters:

- Create a function `printCoordinates` that takes an array `[x, y]` as a parameter.
- Use destructuring in the function parameter to extract `x` and `y`.
- Log `x` and `y` inside the function.
- Call the function with different coordinates.

Object Parameters:

- Create a function `displayUser` that takes an object `{name, age}` as a parameter.
- Use destructuring in the function parameter to extract `name` and `age`.
- Log `name` and `age` inside the function.
- Call the function with different user objects.

List Property Names:

- Create an object `book` with properties `title`, `author`, and `year`.
- Use `Object.keys()` to get an array of the property names of the `book` object.
- Log the array to the console.

Count Properties:

- Create an object `student` with properties `name`, `age`, `grade`, and `school`.
- Use `Object.keys()` to get an array of the property names and determine the number of properties in the `student` object.
- Log the number of properties to the console.

Iterate Over Keys:

Create an object `product` with properties `name`, `price`, and `category`.

Use `Object.keys()` to get an array of the property names and iterate over this array to log each property name and its corresponding value.

List Property Values:

- Create an object `movie` with properties `title`, `director`, `year`, and `genre`.
- Use `Object.values()` to get an array of the property values of the `movie` object.
- Log the array to the console.

Sum Values:

- Create an object `scores` with properties `math`, `science`, and `english`, each with numeric values.
- Use `Object.values()` to get an array of the property values and calculate the total sum of the values.
- Log the sum to the console.

Iterate Over Values:

- Create an object `user` with properties `username`, `email`, and `location`.
- Use `Object.values()` to get an array of the property values and iterate over this array to log each value.

List Entries:

- Create an object `car` with properties `make`, `model`, and `year`.
- Use `Object.entries()` to get an array of the key-value pairs of the `car` object.
- Log the array to the console.

Convert Object to Array:

- Create an object `person` with properties `firstName`, `lastName`, and `age`.
- Use `Object.entries()` to convert the `person` object into an array of key-value pairs.
- Log the array to the console.

Iterate Over Entries:

- Create an object `settings` with properties `theme`, `notifications`, and `privacy`.
- Use `Object.entries()` to get an array of the key-value pairs and iterate over this array to log each key and value.

Filter Keys:

- Create an object `inventory` with properties `apples`, `bananas`, `oranges`, and `grapes`, each with numeric values.
- Use `Object.keys()` and `filter()` to get an array of keys where the value is greater than 10.
- Log the array to the console.

Transform Values:

- Create an object `temperatures` with properties `morning`, `afternoon`, and `evening`, each with numeric values.
- Use `Object.entries()` to get an array of key-value pairs, then use `map()` to convert the temperatures from Celsius to Fahrenheit.
- Convert the transformed array back to an object.
- Log the new object to the console.

Key-Value Swap:

- Create an object `roles` with properties `admin`, `editor`, and `viewer`, each with string values.
- Use `Object.entries()` to get an array of key-value pairs, then use `map()` to swap the keys and values.
- Convert the transformed array back to an object.
- Log the new object to the console.

Filter and Map:

- Create an array `numbers` with values from 1 to 10.
- Write a higher-order function `filterAndMap` that takes an array, a filter function, and a map function.
- Use this function to filter out even numbers and then square the remaining numbers.
- Log the resulting array to the console.

Sort and Reduce:

- Create an array `words` with the values "apple", "banana", "cherry", "date".
- Write a higher-order function `sortAndReduce` that takes an array, a sort function, and a reduce function.
- Use this function to sort the words alphabetically and then concatenate them into a single string.
- Log the resulting string to the console.

Simple Callback:

- Write a function `greet` that takes a name and a callback function.
- The `greet` function should call the callback function with a greeting message.
- Write a callback function `printGreeting` that logs the message to the console.
- Call the `greet` function with a name and the `printGreeting` callback.

Asynchronous Callback:

- Write a function `fetchData` that simulates fetching data from a server (use `setTimeout` to delay execution).
- The `fetchData` function should take a callback function and call it with the data after a delay.
- Write a callback function `displayData` that logs the data to the console.
- Call the `fetchData` function with the `displayData` callback.

Simple Arrow Function:

- Convert the following function to an arrow function:

```
function add(a, b) {
  return a + b;
}
```

- Log the result of calling the arrow function with arguments 3 and 5.

Arrow Function with Array Methods:

- Create an array `numbers` with values from 1 to 5.
- Use the `map` method and an arrow function to create a new array with each number squared.
- Log the resulting array to the console.

Variable Scope:

- Write a function `outer` that declares a variable `x` and assigns it a value.
- Inside `outer`, write another function `inner` that logs `x` to the console.
- Call the `inner` function from within `outer`.
- Call the `outer` function to see the result.

Closure:

- Write a function `createCounter` that returns another function.
- The returned function should increment and log a counter variable that is declared in `createCounter`.
- Create two counters and demonstrate that they maintain independent state.

Simple Default Parameters:

- Write a function `greet` that takes a name and a greeting message with a default value of "Hello".
- The function should log the greeting message and the name to the console.
- Call the function with and without the greeting message to see both cases.

Default Parameters with Other Arguments:

- Write a function `calculateArea` that takes `width` and `height` with default values of 10 and 5, respectively.
- The function should return the area.
- Call the function with and without arguments and log the results.

Square Numbers:

- Create an array `numbers` with values `[1, 2, 3, 4, 5]`.
- Use `map` to create a new array where each number is squared.
- Log the resulting array to the console.

Convert to Uppercase:

- Create an array `words` with values `["apple", "banana", "cherry"]`.
- Use `map` to create a new array where each word is converted to uppercase.
- Log the resulting array to the console.

Filter Even Numbers:

- Create an array `numbers` with values `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`.
- Use `filter` to create a new array containing only even numbers.
- Log the resulting array to the console.

Filter Long Words:

- Create an array `words` with values `["apple", "banana", "cherry", "date"]`.
- Use `filter` to create a new array containing only words with more than 5 characters.
- Log the resulting array to the console.

Log Numbers:

- Create an array `numbers` with values `[1, 2, 3, 4, 5]`.
- Use `forEach` to log each number to the console.

Log Word Lengths:

- Create an array `words` with values `["apple", "banana", "cherry"]`.
- Use `forEach` to log the length of each word to the console.

Sum of Numbers:

- Create an array `numbers` with values `[1, 2, 3, 4, 5]`.
- Use `reduce` to calculate the sum of the numbers.
- Log the result to the console.

Concatenate Strings:

- Create an array `words` with values `["Hello", "world", "this", "is", "JavaScript"]`.
- Use `reduce` to concatenate all the words into a single string, separated by spaces.
- Log the result to the console.

Check for Even Number:

- Create an array `numbers` with values `[1, 3, 5, 7, 8]`.
- Use `some` to check if there is at least one even number in the array.
- Log the result to the console.

Check for Long Word:

- Create an array `words` with values `["apple", "banana", "cherry", "date"]`.
- Use `some` to check if there is at least one word with more than 5 characters.
- Log the result to the console.

Check All Even Numbers:

- Create an array `numbers` with values `[2, 4, 6, 8, 10]`.
- Use `every` to check if all numbers in the array are even.
- Log the result to the console.

Check All Long Words:

- Create an array `words` with values `["elephant", "giraffe", "hippopotamus"]`.
- Use `every` to check if all words in the array have more than 5 characters.
- Log the result to the console.

Find First Even Number:

- Create an array `numbers` with values `[1, 3, 5, 7, 8]`.
- Use `find` to get the first even number in the array.
- Log the result to the console.

Find Long Word:

- Create an array `words` with values `["apple", "banana", "cherry", "date"]`.
- Use `find` to get the first word with more than 5 characters.
- Log the result to the console.

Find Index of First Even Number:

- Create an array `numbers` with values `[1, 3, 5, 7, 8]`.
- Use `findIndex` to get the index of the first even number in the array.
- Log the result to the console.

Find Index of Long Word:

- Create an array `words` with values `["apple", "banana", "cherry", "date"]`.
- Use `findIndex` to get the index of the first word with more than 5 characters.
- Log the result to the console.

Simple Promise:

- Write a function `delay` that returns a promise which resolves after a given number of milliseconds.
- Use the `delay` function to log "Hello, world!" to the console after a delay of 2 seconds.

Promise Chain:

- Write a function `fetchData` that returns a promise which resolves with some data (e.g., a simple object).
- Chain a `.then()` method to the `fetchData` promise to log the data to the console.

Error Handling:

- Write a function `fetchUserData` that returns a promise which resolves with user data (e.g., an object with `name` and `age` properties).
- Modify the function to reject the promise with an error message if the user data is missing an `age` property.
- Use a `.catch()` method to handle the error and log an appropriate message to the console.

Simulate Network Request:

- Write a function `getWeather` that simulates a network request to fetch weather data (use `setTimeout`).
- The function should return a promise that resolves with weather data after 1 second.
- Simulate an error scenario where the promise rejects with an error message.
- Use `.then()` and `.catch()` to handle both success and error cases, logging appropriate messages to the console.

Simple `async` Function:

- Write an `async` function `sayHello` that uses `await` to call the `delay` function (from Task 1) and logs "Hello, world!" to the console after a delay of 2 seconds.

Fetch Data with `async`/`await`:

- Write an `async` function `getUserData` that uses `await` to call the `fetchUserData` function (from Task 2).
- Use a `try`/`catch` block to handle potential errors and log appropriate messages to the console.

Fetch and Process Data:

- Write a function `fetchUser` that returns a promise which resolves with user data (e.g., an object with `name` and `age` properties).
- Write a function `fetchPosts` that returns a promise which resolves with an array of posts for a given user.
- Write an `async` function `getUserAndPosts` that uses `await` to fetch user data and then their posts, logging both to the console.

Error Handling in `async`/`await`:

- Modify the `fetchUser` function to reject the promise with an error if the user data is not found.
- Write an `async` function `getUserInfo` that uses `await` to call the `fetchUser` function and handles potential errors with a `try`/`catch` block, logging appropriate messages to the console.

Simulate API Calls:

- Write a function `apiCall` that simulates an API call and returns a promise which resolves with data after a random delay (use `setTimeout` and `Math.random()`).
- Write an `async` function `getData` that uses `await` to call `apiCall` three times in sequence, logging each result to the console.
- Use `try`/`catch` to handle any errors that may occur during the API calls.