Diploma In Software Engineering (FMD-1) (Developers Stack Academy) Assignment 1

Part A [60 marks]

1. **What is JavaScript, and how does it differ from Java?**

JavaScript is a scripting language for web development, adding dynamic behavior to web pages run on web browsers. It's dynamically typed and object-oriented. **Java** is a general-purpose, compiled language suitable for various applications. It's statically typed and fully object-oriented. Java's "Write Once, Run Anywhere" principle allows it to run on different platforms. Key differences include purpose, typing, object orientation, execution, and platform.

2. **Explain how var, let, and const differ in JavaScript?**
   var, let, and const are three keywords used to declare variables in JavaScript, each with its own distinct scope and behavior:

var: Declares variables with function scope or global scope if declared outside a function. Can be reassigned or redeclared within its scope. Generally considered less preferred due to its potential for hoisting and scope-related issues.

let: Declares variables with block scope, meaning their visibility is limited to the block where they are declared. It can be reassigned but cannot be redeclared within the same block. Provides better control over variable lifetimes and prevents unintended variable shadowing.

const: Declares variables with block scope and assigns a constant value that cannot be reassigned or redeclared. Once initialized, the value of a const variable remains unchanged throughout its scope. Often used for values that should not be modified, such as fundamental constants or object references.

3. **What are the arrow functions in JavaScript?**

Arrow functions are a concise syntax for defining functions in JavaScript. They were introduced in ES6 and offer a cleaner and more readable way to write functions, especially for short, anonymous functions.

Key characteristics:

Implicit return: If the function body consists of a single expression, the return value is implicitly returned without using the return keyword.

No, this binding: Arrow functions do not have their own this binding. Instead, they inherit the this value from the enclosing lexical scope. This can be useful for avoiding this binding issues in callback functions and object methods.

No arguments object: Arrow functions do not have their own arguments object. If you need to access arguments, you can use the rest parameter (...args).

4. What is the difference between null and undefined in JavaScript?

null and **undefined** are two special values in JavaScript that represent the absence of a value. However, they have distinct meanings and use cases:

null: When we intentionally set a variable to null, it indicates that we are deliberately assigning a value to represent the absence of a meaningful value. To indicate the absence of an object or value in data structures like arrays or objects. As a return value from functions that might not always find a result. To reset a variable to an empty state.

undefined: A variable declared but not initialized is automatically assigned the value undefined. To represent a variable that has been declared but hasn't been assigned a value yet. To indicate the absence of a property or method on an object. As a return value from functions that don't have a return statement.

Key differences: null is explicitly assigned, while undefined is implicitly assigned. null indicates the absence of a value by deliberate choice, while undefined indicates the absence of a value due to lack of initialization or access. null is often used for intentional absence or as a placeholder, while undefined is used for variables without values or missing properties.

Example:

```
let x = null; // Explicitly assigned null
let y; // Declared but not initialized (undefined)


console.log(x); // Output: null
console.log(y); // Output: undefined


let person = {
  name: "Alice"
};
console.log(person.age); // Output: undefined (age property doesn't exist)
```

```
function greet(name) {
  if (name === null) {
    console.log("No name provided");
  } else if (name === undefined) {
    console.log("Name is undefined");
  } else {
    console.log("Hello, " + name);
  }
}
greet(null); // Output: No name provided
greet(); // Output: Name is undefined
greet("Bob"); // Output: Hello, Bob
```

5. **What is hoisting in JavaScript?**

Hoisting in JavaScript is a mechanism where variable declarations are moved to the top of their scope (either function or global scope) before the code is executed. This means that you can reference a variable before it's declared, even though it's technically undefined at that point.

Variable declarations: Only variable declarations are hoisted, not variable assignments or function expressions.

Function declarations: Function declarations are also hoisted, which means you can call them before they are defined.

Function expressions: Function expressions are not hoisted, so you cannot call them before they are defined.

Scope: Hoisting occurs within the scope where the variable is declared (function or global scope).

Example:

```
console.log(x); // Output: undefined (x is hoisted, but not initialized)
var x = 10;
console.log(x); // Output: 10 (x is now initialized)
foo(); // Output: Hello
```

```
function foo() {
  console.log("Hello");
}
bar(); // Output: Error: bar is not a function (function expressions are not hoisted)
var bar = function() {
  console.log("World");
};
```

6. How does JavaScript handle scope (global, function, block scope)?

JavaScript scope refers to the visibility or accessibility of variables and functions within different parts of your code. There are three main types of scope in JavaScript:

1. Global Scope:

- Variables declared outside of any function or block have global scope.
- They can be accessed from anywhere within the JavaScript code, including functions.
- However, it's generally considered a best practice to avoid excessive use of global variables, as they can make your code harder to maintain and can lead to naming conflicts.

2. Function Scope:

- Variables declared within a function have function scope.
- They are only accessible within that function and its nested functions.
- Variables declared with `var` keyword have function scope.

3. Block Scope:

- Variables declared within a block (code enclosed in curly braces) have block scope.
- They are only accessible within that block and any nested blocks.
- Variables declared with `let` and `const` keywords have block scope.

```
var globalVariable = "I'm a global variable";


function myFunction() {
 var functionVariable = "I'm a function variable";
 let blockVariable = "I'm a block variable";
 if (true) {
  let blockScopeVariable = "I'm a block-scoped variable";
  console.log(blockScopeVariable); // Output: I'm a block-scoped variable
 }
 console.log(blockVariable); // Output: I'm a block variable
 console.log(functionVariable); // Output: I'm a function variable
}
myFunction();
console.log(globalVariable); // Output: I'm a global variable
console.log(functionVariable); // Error: functionVariable is not defined
console.log(blockVariable); // Error: blockVariable is not defined
```

7. What is the this keyword in JavaScript, and how is it used?

The this keyword in JavaScript refers to the object that is executing the current function. Its value depends on how the function is called. Here are the common ways this is used:

1. Method Calls:

- When a function is called as a method of an object, this refers to that object.
- For example:

JavaScript

```
const person = {
 name: "Alice",
 greet: function() {
  console.log("Hello, my name is " + this.name);
 }
};

person.greet(); // Output: Hello, my name is Alice
```

2. Constructor Functions:

- In constructor functions, this refers to the newly created object instance.
- For example:

JavaScript

```
function Person(name) {

  this.name = name;

}

const person1 = new Person("Bob");

console.log(person1.name); // Output: Bob
```

3. Regular Function Calls:

- When a function is called as a regular function (not as a method), this refers to the global object (usually window in a browser environment).
- For example:

JavaScript

```
function greet() {

  console.log("Hello, my name is " + this.name);

}

greet(); // Output: Hello, my name is undefined (if there's no global name property)
```

4. Call, Apply, and Bind Methods:

- These methods allow you to explicitly set the value of this for a function call.
- For example:

JavaScript

```
const person = { name: "Alice" };

const greet = function() {

  console.log("Hello, my name is " + this.name);

};

greet.call(person); // Output: Hello, my name is Alice
```

5. Arrow Functions:

- Arrow functions do not have their own this binding. They inherit the this value from the enclosing lexical scope.
- For example:

JavaScript

```
const person = { name: "Alice" };
const greet = () => {
  console.log("Hello, my name is " + this.name);
};
person.greet(); // Output: Hello, my name is Alice
```

8. **What are template literals in JavaScript?**

Template literals are a new feature introduced in ES6 that provide a more convenient way to create strings in JavaScript. They allow you to embed expressions within strings, making it easier to create dynamic content.

Syntax:

JavaScript

```
`Your text here ${expression}`
```

The backticks (` `) are used to define a template literal. Expressions enclosed in ${} are evaluated and their values are inserted into the string.

Examples:

- **Basic usage:**

JavaScript

```
const name = "Alice";
const greeting = `Hello, ${name}!`;
console.log(greeting); // Output: Hello, Alice!
```

- **Multiple expressions:**

JavaScript

```
const num1 = 10;
const num2 = 20;
const result = `The sum of ${num1} and ${num2} is ${num1 + num2}`;

console.log(result); // Output: The sum of 10 and 20 is 30
```

- **Multiline strings:**

JavaScript

```
const message = `This is a multiline

string.`;

console.log(message);
```

- **Tag functions:**

JavaScript

```
function greet(strings, ...values) {

  return strings[0] + values[0] + "!";

}

const name = "Bob";

const greeting = greet`Hello, ${name}`;

console.log(greeting); // Output: Hello, Bob!
```

9. **How do you handle errors in JavaScript using try...catch?**

The try...catch block is a fundamental error handling mechanism in JavaScript. It allows you to anticipate potential errors in your code and gracefully handle them, preventing your application from crashing.

Here's the basic structure of a try...catch block:

JavaScript

```
try {

  // Code that might throw an error

} catch (error) {

  // Code to handle the error

}
```

How it works:

1. **try block:** The code within the try block is executed. If an error occurs, the execution is immediately stopped, and the control is transferred to the catch block.

2. **catch block:** The catch block receives the error object as an argument. You can use this object to access information about the error, such as its message, stack trace, and name. The code within the catch block is executed to handle the error and prevent the application from crashing.

Example:

JavaScript

```
function divideNumbers(a, b) {
  try {
    if (b === 0) {
      throw new Error("Division by zero is not allowed");
    }
    return a / b;
  } catch (error) {
    console.error("An error occurred:", error.message);
    return null;
  }
}

const result = divideNumbers(10, 0);
console.log(result); // Output: An error occurred: Division by zero is not allowed
```

10. . Explain the difference between == and === in JavaScript.

In JavaScript, == and === are both comparison operators used to check if two values are equal, but they have different behaviors:

== (Loose equality):

- Performs type coercion, meaning it attempts to convert values to a common type before comparing them.
- If the types are different, JavaScript will try to convert them to a compatible type.
- For example, 1 == "1" will evaluate to true because the string "1" is converted to a number before comparison.

=== (Strict equality):

- Does not perform type coercion.
- Values must be of the same type and have the same value to be considered equal.
- For example, 1 === "1" will evaluate to false because the types are different, even though the values are the same.

Key differences:

- **Type coercion:** == performs type coercion, while === does not.
- **Strictness:** === is stricter and requires both values to be of the same type, while == allows for type conversions.
- **Recommended usage:** It's generally recommended to use === for most comparisons to avoid unexpected behavior caused by type coercion.

Example:

console.log(10 == "10"); // Output: true (loose equality)

console.log(10 === "10"); // Output: false (strict equality)

console.log(null == undefined); // Output: true (loose equality)

console.log(null === undefined); // Output: false (strict equality)

11. What are JavaScript data types?

JavaScript data types define the kind of values that can be stored in variables. Here are the primitive data types in JavaScript:

1. Number:

- Represents numerical values, including integers and floating-point numbers.
- Examples: 10, 3.14, -5

2. String:

- Represents sequences of characters enclosed in quotes.
- Examples: "Hello, world", 'JavaScript'

3. Boolean:

- Represents logical values, either true or false.
- Used for conditional statements and logical operations.

4. Null:

- Represents the absence of a value.
- It is a special value that indicates an intentional absence of an object.

5. Undefined:

- Represents a variable that has been declared but not yet assigned a value.
- It is often used to indicate the absence of a defined value.

6. Symbol (ES6):

- Represents a unique identifier that cannot be changed or duplicated.
- Used for creating unique property keys on objects.

7. BigInt (ES2020):

- Represents integers of arbitrary size, allowing you to work with large numbers that exceed the range of the Number type.

In addition to primitive data types, JavaScript also has object data types:

1. Object:

- Represents a collection of key-value pairs, where keys are strings and values can be of any data type.
- Used to store complex data structures and create custom objects.

2. Array:

- Represents an ordered collection of elements, which can be of any data type.
- Used to store multiple values in a single variable.

3. Function:

- Represents a block of code that can be executed.
- Used to define reusable code blocks and create custom functions.

12. How can you convert a string to a number in JavaScript?

There are several ways to convert a string to a number in JavaScript:

1. Using the parseInt() function:

- This function parses a string and returns an integer.
- You can optionally specify the radix (base) of the number.

```
const stringNumber = "123";
const number = parseInt(stringNumber);

console.log(number); // Output: 123
```

2. Using the parseFloat() function:

- This function parses a string and returns a floating-point number.

```
const stringNumber = "3.14";

const number = parseFloat(stringNumber);

console.log(number); // Output: 3.14
```

3. Using the unary plus operator (+):

- This operator converts a string to a number.

```
const stringNumber = "10";

const number = +stringNumber;

console.log(number); // Output: 10
```

4. Using the Number() function:

- This function converts a value to a number.
- It can handle various input types, including strings, numbers, and booleans.

```
const stringNumber = "20";

const number = Number(stringNumber);

console.log(number); // Output: 20
```

5. Using the Number.parseInt() and Number.parseFloat() methods:

- These methods are equivalent to the parseInt() and parseFloat() functions.

```
const stringNumber = "42";

const number = Number.parseInt(stringNumber);

console.log(number); // Output: 42
```

13. What is the difference between map() and forEach()?

map() and **forEach()** are both higher-order functions in JavaScript used to iterate over arrays, but they serve different purposes:

map()

- **Purpose:** Creates a new array by applying a function to each element of the original array.
- **Returns:** A new array with the transformed elements.
- **Use cases:** When you need to create a new array based on the elements of the original array, such as filtering, mapping to different data types, or performing calculations.

forEach()

- **Purpose:** Executes a function for each element of an array.
- **Returns:** undefined.
- **Use cases:** When you need to perform side effects, such as modifying elements in place, logging values, or making API calls.

Example:

```
const numbers = [1, 2, 3, 4, 5];

// map() creates a new array of squared numbers

const squaredNumbers = numbers.map(number => number * number);

console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]

// forEach() modifies the original array in place

numbers.forEach((number, index) => {

  numbers[index] *= 2;

});

console.log(numbers); // Output: [2, 4, 6, 8, 10]
```

14. How do you create and manipulate objects in JavaScript?

There are two primary ways to create objects in JavaScript:

1. Object Literal Notation:

- The most common method, using curly braces {} to define properties and their values.

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York"
};
```

2. Constructor Function:

- A function that creates objects using the new keyword.

```
function Person(name, age, city) {
  this.name = name;
  this.age = age;
  this.city = city;
}
const person1 = new Person("Bob", 25, "Los Angeles");
```

Manipulating Objects:

Once you've created an object, you can access and modify its properties using dot notation or bracket notation:

1. Dot Notation:

- Use a dot to access or assign properties.

```
person.name = "Charlie";
console.log(person.age); // Output: 30
```

2. Bracket Notation:

- Use square brackets to access properties with dynamic keys or properties that contain special characters.

```
const key = "city";
console.log(person[key]); // Output: New York
```

```
person["address"] = "123 Main St";
```

Adding and Removing Properties:

- **Adding properties:**

```
person.occupation = "Engineer";
```

- **Removing properties:**

```
delete person.age;
```

Iterating Over Properties:

- Use the for...in loop to iterate over an object's properties.

```
for (let property in person) {
  console.log(property + ": " + person[property]);
}
```

Methods:

- Objects can have methods, which are functions defined within the object.

```
const person = {
  name: "Alice",
  greet: function() {
    console.log("Hello, my name is " + this.name);
  }
};
person.greet();
```

15. How do you handle default parameters in JavaScript functions?

Default parameters in JavaScript allow you to specify default values for function arguments. If a value is not provided for a parameter when the function is called, the default value is used instead.

Syntax:

```
function myFunction(param1 = defaultValue1, param2 = defaultValue2) {
  // Function body
}
```

```
function greet(name = "World") {
  console.log("Hello, " + name + "!");
}
greet(); // Output: Hello, World!
greet("Alice"); // Output: Hello, Alice!
```

In this example, the greet function has a default parameter name with a value of "World". When the function is called without an argument, "World" is used. If an argument is provided, it overrides the default value.

Key points:

- Default parameters must be placed after any required parameters.
- You can use expressions or other functions as default values.
- If a default parameter is undefined, it will be treated as undefined.

Additional notes:

- **Rest parameters:** The ... syntax can be used to collect multiple arguments into an array. Default values for rest parameters are treated as empty arrays.
- **Destructuring assignment:** Destructuring assignment can be used to assign default values to function parameters.

16. What is a higher-order function in JavaScript?

A higher-order function in JavaScript is a function that takes one or more functions as arguments or returns a function as a result. This concept is fundamental to functional programming and allows for more flexible and expressive code.

characteristics:

- **Takes functions as arguments:** Higher-order functions can accept other functions as input parameters. This enables you to create functions that operate on other functions, such as applying transformations, filtering, or mapping.
- **Returns functions as results:** Higher-order functions can return new functions as their output. This allows you to create functions that generate other functions based on specific criteria or configurations.

Common examples of higher-order functions:

- **map():** Takes an array and a function as arguments, applies the function to each element of the array, and returns a new array with the transformed values.

- **filter():** Takes an array and a function as arguments, filters the elements of the array based on the function's return value, and returns a new array containing only the elements that pass the filter.
- **reduce():** Takes an array, an initial value, and a function as arguments, applies the function to each element of the array and the accumulated value, and returns a single value.
- **forEach():** Takes an array and a function as arguments, executes the function for each element of the array, but does not return a value.

```
function applyOperation(array, operation) {
  return array.map(operation);
}


const numbers = [1, 2, 3, 4, 5];
const squaredNumbers = applyOperation(numbers, (number) => number * number);
console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

17. Explain what function currying is in JavaScript?

Currying in JavaScript is a technique that transforms a function that takes multiple arguments into a series of functions that each take a single argument. It involves creating a new function that takes the first argument and returns a new function that takes the second argument, and so on. This process continues until all arguments have been provided.

Benefits of currying:

- **Improved code readability:** Currying can make code more readable by breaking down functions into smaller, more focused parts.
- **Partial application:** Currying allows you to create partially applied functions, which can be useful for creating reusable functions that take some arguments as constants and others as variables.
- **Functional programming:** Currying is a key concept in functional programming, enabling you to write more declarative and composable code.

```
function add(x, y) {
  return x + y;
}
```

```
const add5 = add.bind(null, 5);

console.log(add5(3)); // Output: 8
```

In this example, add5 is a curried version of the add function. It takes a single argument y and adds 5 to it, returning the result.

Another example:

JavaScript

```
function greet(salutation, name) {
  return `${salutation}, ${name}!`;
}


const greetMr = greet.bind(null, "Mr.");

console.log(greetMr("John")); // Output: Mr., John!
```

Here, greetMr is a curried version of the greet function that always uses the salutation "Mr.".

Currying can be implemented using various techniques:

- **bind() method:** This method creates a new function with a fixed this value and pre-specified arguments.
- **Higher-order functions:** You can create curried functions using higher-order functions like map, reduce, or custom functions.
- **Closures:** Closures can be used to capture and retain the values of variables from outer scopes, which is essential for currying.

18. **What is an immediately-invoked function expression (IIFE) in JavaScript?**

An immediately-invoked function expression (IIFE) is a JavaScript function that is defined and executed in a single statement. It's a common pattern used to create private scopes and avoid polluting the global namespace.

```
(function() {
  // Function body
})();
```

The parentheses around the function definition and the immediately following () invoke the function as soon as it's defined. This creates a new scope for the function's variables, preventing them from interfering with variables in the outer scope.

Key benefits of IIFEs:

- **Private scope:** Variables declared within an IIFE are private to that function, preventing naming conflicts and accidental modifications.
- **Modularity:** IIFEs can be used to encapsulate related code into self-contained modules, improving code organization and maintainability.
- **Immediate execution:** IIFEs are executed as soon as they are defined, making them useful for initializing variables or performing tasks that need to be executed immediately.

```
(function() {
  const secretMessage = "This is a secret message";
  console.log(secretMessage); // Output: This is a secret message
})();

console.log(secretMessage); // Output: ReferenceError: secretMessage is not defined
```

In this example, the secretMessage variable is defined and logged within the IIFE. Because the variable is declared within the function's scope, it is not accessible outside of the IIFE, preventing it from polluting the global namespace.

IIFEs are a powerful tool for organizing and managing JavaScript code. By understanding how they work, you can write more modular, maintainable, and secure code.

19. What are generators in JavaScript, and how do you use them?

Generators are a special type of function in JavaScript that allow you to pause and resume execution, creating iterators that can produce a sequence of values over time. They are particularly useful for handling asynchronous operations and creating custom iterators.

Key characteristics of generators:

- **yield keyword:** Generators use the yield keyword to pause execution and return a value. When the generator is resumed, execution continues from the point where it was paused.
- **Iterators:** Generators can be used to create iterators, which are objects that provide a way to iterate over a sequence of values.
- **Asynchronous operations:** Generators are often used to handle asynchronous operations in a more synchronous-like manner, using techniques like async/await.

```
function* generatorFunction() {
  // Generator body
  yield value1;
  yield value2;
  // ...
}
function* countToFive() {
  for (let i = 1; i <= 5; i++) {
    yield i;
  }
}

const generator = countToFive();

console.log(generator.next()); // Output: { value: 1, done: false }
console.log(generator.next()); // Output: { value: 2, done: false }
console.log(generator.next()); // Output: { value: 3, done: false }
console.log(generator.next()); // Output: { value:   4, done: false }
console.log(generator.next()); // Output: { value: 5, done: false }
console.log(generator.next()); // Output: { value:   undefined, done: true }
```

Using generators with for...of:

```
function* generateNumbers() {
  yield 1;
  yield 2;
  yield 3;
}

for (let number of generateNumbers()) {
  console.log(number); // Output: 1, 2, 3
}
```

Key use cases of generators:

- **Asynchronous programming:** Generators can be used to create asynchronous workflows that are easier to read and maintain.
- **Custom iterators:** You can create custom iterators for various data structures or algorithms.
- **Lazy evaluation:** Generators can be used to lazily evaluate values, which can be beneficial for performance optimization.

20. What is memoization in JavaScript, and when would you use it?

Memoization is a technique in JavaScript that optimizes function calls by caching the results of previous function executions. When a function is called with the same arguments, the cached result is returned instead of re-calculating it. This can significantly improve performance, especially for functions that are called frequently with the same arguments.

When to use memoization:

- **Expensive functions:** Functions that perform complex calculations or involve heavy computations can benefit from memoization, as repeated calls with the same arguments can be avoided.
- **Pure functions:** Memoization is most effective for pure functions, which always return the same result for the same inputs and have no side effects.
- **Functions with frequent calls:** If a function is called multiple times with the same arguments, memoization can prevent redundant calculations.

```
function factorial(n) {
 if (n === 0) {
  return 1;
 } else {
  return n * factorial(n - 1);
 }
}


// Memoized version
function memoizedFactorial(n) {
 const cache = {};
 return function(n) {
  if (n in cache) {
   return cache[n];
  } else {
   const result = n * memoizedFactorial(n - 1);
   cache[n] = result;
   return result;
  }
 }(n);
}


// Test
console.log(memoizedFactorial(5)); // Output: 120
console.log(memoizedFactorial(5)); // Output: 120 (cached result)
```

In this example, the memoizedFactorial function uses a cache object to store previously calculated results. When the function is called with the same argument, the cached result is returned, avoiding the recursive calculations. This can significantly improve performance for large values of n.

Key points to remember:

- Memoization can be implemented manually or using libraries like lodash.
- It's important to choose functions that are suitable for memoization, such as pure functions with frequent calls.
- Memoization can introduce additional memory overhead, so it's important to balance performance gains with potential memory usage.

21. How can you remove duplicates from an array in JavaScript?

There are several ways to remove duplicates from an array in JavaScript:

1. Using a Set:

JavaScript

```
function removeDuplicates(arr) {
 return [...new Set(arr)];
}


const arrayWithDuplicates = [1, 2, 3, 2, 4, 1];

const uniqueArray = removeDuplicates(arrayWithDuplicates);

console.log(uniqueArray); // Output: [1, 2, 3, 4]
```

A Set in JavaScript is a collection of unique values. By creating a new Set from the array and then converting it back to an array using the spread operator (...), we effectively remove duplicates.

2. Using a Loop and Object:

JavaScript

```
function removeDuplicates(arr) {

  const seen = {};

  const uniqueArray = [];

  for (let i = 0; i < arr.length; i++) {

    if (!seen[arr[i]]) {

      seen[arr[i]] = true;

      uniqueArray.push(arr[i]);

    }

  }

  return uniqueArray;

}


const arrayWithDuplicates = [1, 2, 3, 2, 4, 1];

const uniqueArray = removeDuplicates(arrayWithDuplicates);

console.log(uniqueArray); // Output: [1, 2, 3, 4]
```

This method uses an object to keep track of the elements seen so far. If an element is not seen before, it is added to the unique array.

3. Using filter() and indexOf():

JavaScript

```
function removeDuplicates(arr) {
  return arr.filter((item, index) => arr.indexOf(item) === index);
}


const arrayWithDuplicates = [1, 2, 3, 2, 4, 1];
const uniqueArray = removeDuplicates(arrayWithDuplicates);
console.log(uniqueArray); // Output: [1, 2, 3, 4]
```

This method filters the array, keeping only elements whose index is equal to the first index of that element. This ensures that duplicates are removed.

4. Using reduce() and includes():

JavaScript

```
function removeDuplicates(arr) {
  return arr.reduce((unique, item) => (unique.includes(item) ? unique : [...unique, item]), []);
}


const arrayWithDuplicates = [1, 2, 3, 2, 4, 1];
const uniqueArray = removeDuplicates(arrayWithDuplicates);
console.log(uniqueArray); // Output: [1, 2, 3, 4]
```

This method uses reduce to iterate over the array and build a new unique array. If the current element is already in the unique array, it's skipped. Otherwise, it's added to the unique array.

5. Using a library:

If you're working with a library like Lodash or Ramda, they often provide built-in functions for removing duplicates from an array. For example, in Lodash, you can use the uniq function.

Choose the method that best suits your needs and coding style. The Set method is often considered the most concise and efficient option.

22. What are the various ways to merge arrays in JavaScript?

There are several ways to merge arrays in JavaScript:

1. Concatenation using the concat() method:

JavaScript

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];

const mergedArray = array1.concat(array2);
console.log(mergedArray); // Output: [1, 2, 3, 4, 5, 6]
```

The concat() method creates a new array by combining the elements of the original arrays. It does not modify the original arrays.

2. Spread operator (...):

JavaScript

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];

const mergedArray = [...array1, ...array2];
console.log(mergedArray); // Output:   [1, 2, 3, 4, 5, 6]
```

The spread operator allows you to spread the elements of an array into another array. It's a concise and readable way to merge arrays.

3. Using push() and concat():

JavaScript

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];

array1.push(...array2);
console.log(array1); // Output: [1, 2, 3, 4, 5, 6]
```

This method modifies the original array1 by adding the elements of array2 to it.

4. Using a loop:

JavaScript

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const mergedArray = [];
for (let i = 0; i < array1.length; i++) {
  mergedArray.push(array1[i]);
}
for (let i = 0; i < array2.length; i++) {
  mergedArray.push(array2[i]);
}

console.log(mergedArray); // Output:   [1, 2, 3, 4, 5, 6]
```

This method manually iterates over both arrays and pushes their elements into a new array.

23. How do you validate user input in JavaScript?

Validating user input in JavaScript is essential to ensure data integrity and prevent errors. Here are some common techniques:

1. Regular Expressions:

- Use regular expressions to match input against specific patterns.
- For example, to validate an email address:

JavaScript

```
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
if (!emailRegex.test(email)) {
  console.error("Invalid email format");
}
```

2. Type Checking:

- Use typeof to check the data type of the input.
- For example, to ensure a number is entered:

JavaScript

```
if (typeof number !== "number") {
  console.error("Please enter a valid number");
}
```

3. Range Checking:

- Use comparison operators to check if a number is within a specific range.
- For example, to validate an age:

JavaScript

```
if (age < 0 || age > 120) {
  console.error("Invalid age");
}
```

4. Length Checking:

- Use length property to check the length of strings or arrays.
- For example, to validate a password length:

JavaScript

```
if (password.length < 8) {
  console.error("Password must be at least 8 characters long");
}
```

5. Custom Validation Functions:

- Create custom functions to perform specific validation checks.
- For example, to validate a phone number:

JavaScript

```
function validatePhoneNumber(phoneNumber) {
  // Your custom validation logic here
  if (!/^\d{10}$/.test(phoneNumber)) {
    console.error("Invalid phone number format");
  }
}
```

6. Client-Side and Server-Side Validation:

- Perform validation on the client-side using JavaScript to provide immediate feedback to the user.

- However, always validate input on the server-side as well to prevent malicious attacks and ensure data integrity.

24. What is the typeof operator in JavaScript, and what are its possible return values?

The typeof operator in JavaScript is used to determine the data type of a value. It returns a string that indicates the type of the value.

Here are the possible return values of the typeof operator:

- **"number"**: For numbers (e.g., 10, 3.14).
- **"string"**: For strings (e.g., "Hello, world", 'JavaScript').
- **"boolean"**: For boolean values (true, false).
- **"object"**: For objects, arrays, functions, dates, and null values.
- **"undefined"**: For variables that have not been assigned a value.
- **"function"**: For functions.
- **"symbol"**: For symbols (introduced in ES6).
- **"bigint"**: For BigInt values (introduced in ES2020).

```
console.log(typeof 10); // Output: "number"

console.log(typeof "Hello"); // Output: "string"

console.log(typeof true); // Output: "boolean"

console.log(typeof null); // Output: "object"

console.log(typeof undefined); // Output:    "undefined"

console.log(typeof    function() {}); // Output: "function"

console.log(typeof Symbol("unique")); // Output: "symbol"

console.log(typeof BigInt(123456789012345678901234567890)); // Output: "bigint"
```

25. How can you check if a variable is an array in JavaScript?

There are several ways to check if a variable is an array in JavaScript:

1. Using the Array.isArray() method:

```
const myArray = [1, 2, 3];
```

```
console.log(Array.isArray(myArray)); // Output: true
```

This is the most reliable and recommended method for checking if a variable is an array.

2. Using the constructor property:

```
const myArray = [1, 2, 3];

console.log(myArray.constructor === Array); // Output: true
```

This method checks if the constructor property of the variable is equal to the Array constructor. However, this method can be unreliable in certain cases, especially when dealing with objects that have been modified to have a different constructor property.

3. Using the instanceof operator:

```
const myArray = [1, 2, 3];

console.log(myArray instanceof Array); // Output: true
```

This method checks if the variable is an instance of the Array object. However, this method can also be unreliable in certain cases, especially when dealing with objects that have been modified to have a different prototype chain.

4. Using the toString() method:

```
const myArray = [1, 2, 3];

console.log(myArray.toString() === "[object Array]"); // Output: true
```

This method checks if the string representation of the variable is equal to "[object Array]". However, this method can be unreliable as it relies on the internal implementation of the toString() method.

26. Explain what a Symbol is in JavaScript and provide an example of its use.

Symbols in JavaScript are unique identifiers that can be used as object property keys. They are different from strings in that they cannot be duplicated or modified. This makes them useful for creating private properties or symbols that are guaranteed to be unique within an application.

Creating a Symbol:

JavaScript

```
const mySymbol = Symbol("description");
```

This creates a new symbol with the optional description "description". The description is only for debugging purposes and does not affect the symbol's uniqueness.

Using Symbols as Object Properties:

JavaScript

```
const person = {
  [Symbol("name")]: "Alice",
  [Symbol("age")]: 30
};


console.log(person[Symbol("name")]); // Output: Alice
```

In this example, we're using symbols as property keys to create private properties that cannot be accessed directly using a string.

Symbol.for() and Symbol.keyFor():

- Symbol.for(key): This function returns a symbol with the given key. If the symbol already exists, it returns the same symbol. This is useful for creating global symbols that can be shared across different modules.
- Symbol.keyFor(symbol): This function returns the key associated with a given symbol, if it was created using Symbol.for().

Example:

JavaScript

```
const mySymbol = Symbol.for("mySymbol");
console.log(Symbol.keyFor(mySymbol)); // Output: "mySymbol"
```

27. How does the for...in loop differ from for...of in JavaScript?

for...in and **for...of** are both loop constructs used to iterate over objects and arrays in JavaScript, but they have different purposes and behaviors:

for...in:

- **Iterates over object properties:** Primarily used to iterate over the properties of an object.
- **Returns property names:** In each iteration, the loop variable is assigned the name of the current property.
- **Includes inherited properties:** Iterates over both own and inherited properties of an object.

for...of:

- **Iterates over iterable objects:** Used to iterate over iterable objects, such as arrays, strings, maps, sets, and generator functions.
- **Returns values:** In each iteration, the loop variable is assigned the value of the current element.
- **Does not include inherited properties:** Only iterates over the own properties of an object.

```
const person = {
  name: "Alice",
  age: 30
};
const numbers = [1, 2, 3];
// Using for...in to iterate over object properties
for (let property in person) {
  console.log(property + ": " + person[property]);
}
// Using for...of to iterate over array elements
for (let number of numbers) {
console.log(number);
}
```

28. What are rest and spread operators in JavaScript, and how are they used?

Rest and spread operators are powerful features introduced in ES6 that allow you to work with arrays and functions in a more flexible and concise way.

Rest Operator:

- **Syntax:** ...args
- **Purpose:** Collects multiple arguments into an array.
- **Usage:**
    - In function parameters:

```
function sum(...numbers) {
  return numbers.reduce((acc, cur) => acc + cur, 0);
}
```

  - To extract elements from an array:

```
const numbers = [1, 2, 3, 4, 5];
const [first, second, ...rest] = numbers;
console.log(first, second, rest); // Output: 1 2 [3, 4, 5]
```

Spread Operator:

- **Syntax:** ...array
- **Purpose:** Expands an array into individual elements.
- **Usage:**
  - To create new arrays:

const array1 = [1, 2, 3]; const array2 = [4, 5, 6]; const mergedArray = [...array1, ...array2]; console.log(mergedArray); // Output: [1, 2, 3, 4, 5, 6] - To copy objects:javascript const person = { name: "Alice", age: 30 }; const copiedPerson = { ...person }; console.log(copiedPerson); // Output: { name: "Alice", age: 30 } ` ` `

Key points:

- The rest operator can only be used at the end of a parameter list.
- The spread operator can be used anywhere an array is expected.
- Both operators can be used to create more concise and readable code.

Examples:

```
// Using rest operator
function multiply(...numbers) {
  return numbers.reduce((acc, cur) => acc * cur, 1);
}
console.log(multiply(2, 3, 4)); // Output: 24


// Using spread operator
const numbers = [1, 2, 3];
const doubledNumbers = numbers.map(number => number * 2);
```

```
        console.log(doubledNumbers); // Output: [2, 4, 6]
```

29. How can you flatten a multi-dimensional array in JavaScript?

There are several ways to flatten a multi-dimensional array in JavaScript:

1. Using reduce() and concat():

```
function flattenArray(arr) {
    return arr.reduce((acc, val) => acc.concat(Array.isArray(val) ? flattenArray(val) : [val]), []);
}


const multiDimensionalArray = [[1, 2], [3, 4, 5], [6]];
const flattenedArray = flattenArray(multiDimensionalArray);
console.log(flattenedArray); // Output: [1, 2, 3, 4, 5, 6]
```

This method recursively flattens the array by using reduce() to concatenate the elements of each subarray.

2. Using a stack:

```
function flattenArray(arr) {
 const stack = [...arr];
 const result = [];

 while (stack.length > 0) {
  const current = stack.pop();
  if (Array.isArray(current)) {
   stack.push(...current);
  } else {
   result.unshift(current);
  }
 }
return result;
}
```

```
const multiDimensionalArray = [[1, 2], [3, 4, 5], [6]];

const flattenedArray = flattenArray(multiDimensionalArray);

console.log(flattenedArray); // Output: [1, 2, 3, 4, 5, 6]
```

This method uses a stack to keep track of the elements to be processed. It iterates over the stack, flattening subarrays and pushing the elements onto the result array.

3. Using flatMap():

```
function flattenArray(arr) {

  return arr.flatMap(item => (Array.isArray(item) ? flattenArray(item) : [item]));

}


const multiDimensionalArray = [[1, 2], [3, 4, 5], [6]];

const flattenedArray = flattenArray(multiDimensionalArray);

console.log(flattenedArray); // Output: [1, 2, 3, 4, 5, 6]
```

This method is a more concise way to achieve the same result as the reduce() and concat() method. It uses flatMap() to apply the flattening logic to each element of the array.

4. Using a library:

If you're using a library like Lodash or Ramda, they often provide built-in functions for flattening arrays. For example, in Lodash, you can use the flattenDeep function.

Choose the method that best suits your needs and coding style. The flatMap() method is often considered the most concise and efficient option.

30. What is the difference between let and const in terms of reassignment and scope?

let and **const** are both used to declare variables in JavaScript, but they differ in terms of reassignment and scope:

Reassignment:

- **let:** Variables declared with let can be reassigned, meaning their value can be changed after they are declared.
- **const:** Variables declared with const cannot be reassigned, meaning their value is fixed once they are initialized.

Scope:

- **let** and **const:** Both let and const have block scope, meaning their visibility is limited to the block (code enclosed within curly braces) where they are declared. This is different from var, which has function or global scope.

```
function myFunction() {
 let x = 10;
 const y = 20;

 x = 30; // This is allowed with let
 // y = 40; // This would cause an error with const

 if (true) {
  let z = 50;
  console.log(z); // Output: 50 (z is block-scoped)
 }

 console.log(x); // Output: 30
 console.log(y); // Output: 20
 // console.log(z); // ReferenceError: z is not defined (z is block-scoped)
}
```

31. What is NaN in JavaScript, and how can you check if a value is NaN?

NaN stands for "Not a Number" in JavaScript. It's a special value that represents an invalid number or a result of an operation that cannot produce a valid number.

Common scenarios that result in NaN:

- Arithmetic operations with non-numeric values: For example, 10 / "hello" or parseInt("abc").
- Math operations that result in an invalid result: For example, Math.sqrt(-1).
- Comparisons between incompatible types: For example, NaN === NaN.

Checking if a value is NaN:

- isNaN() function: This function returns true if the value is NaN, and false otherwise.
- Number.isNaN() function: This function is more reliable than isNaN() because it correctly handles the case where a value is explicitly NaN**.**

Example:

JavaScript

```
const result1 = 10 / "hello";
const result2 = Math.sqrt(-1);
const result3 = NaN;

console.log(isNaN(result1)); // Output: true
console.log(isNaN(result2)); // Output: true
console.log(isNaN(result3)); // Output: true

console.log(Number.isNaN(result1)); // Output: true
console.log(Number.isNaN(result2)); // Output: true
console.log(Number.isNaN(result3)); // Output: true
```

Part B [40 marks]

Index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>To-Do List</title>
  <link rel="stylesheet" href="style.css">
</head>
```

```
<body>
  <h1>To-Do List</h1>
  <div class="task-input">
    <input type="text" id="taskInput" placeholder="Enter a task">
    <button id="addTaskBtn">Add Task</button>
  </div>
  <ul id="taskList"></ul>
  <script src="script.js"></script>
</body>
</html>
```

style.css

```
ul {
  list-style: none;
  padding: 0;
}

li {
  margin-bottom: 10px;
}

.completed {
  text-decoration: line-through;
  color: gray;
}
```

Script.js

```
const taskInput = document.getElementById('taskInput');

const addTaskBtn = document.getElementById('addTaskBtn');

const taskList = document.getElementById('taskList');

let tasks = [];

addTaskBtn.addEventListener('click', () => {

  const taskText = taskInput.value;

  if (taskText !== '') {

    tasks.push(taskText);

    renderTasks();

    taskInput.value = '';

  }

});

function renderTasks() {

  taskList.innerHTML = '';

  tasks.forEach((task, index) => {

    const li = document.createElement('li');

    li.textContent = task;

    const checkbox = document.createElement('input');

    checkbox.type = 'checkbox';

    checkbox.addEventListener('change', () => {

      if (checkbox.checked) {

        li.classList.add('completed');

      } else {

        li.classList.remove('completed');

      }

    });

    const deleteBtn = document.createElement('button');

    deleteBtn.textContent = 'Delete';
    deleteBtn.addEventListener('click', () => {
      tasks.splice(index, 1);
      renderTasks();
    });

    li.appendChild(checkbox);
    li.appendChild(document.createTextNode(task));
    li.appendChild(deleteBtn);

    taskList.appendChild(li);
  });
}
```