**Assignment 4**

**Q.1 Explain Hoisting in JavaScript ?**

Hoisting in JavaScript is a behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase, even if they appear to be placed lower in the code. This means that you can use variables and call functions before they are actually declared in your code.

For example, you can call a function before its actual declaration:

| sayHello(); // Output: "Hello!" function sayHello() {  console.log("Hello!"); }  console.log(message); // Output: undefined var message = "Hello!"; |
| --- |

**Q.2 Explain Temporal Dead Zone?**

The Temporal Dead Zone (TDZ) is a behavior in JavaScript that occurs when using variables declared with the let and const keywords before they are actually declared in the code. It is a specific period within a block or function where variables exist but cannot be accessed or used.

During the TDZ, if you try to access a variable before its declaration, JavaScript throws a reference error. This is because the variable is in a "dead zone" where it has been registered in the scope but cannot be used until it is formally declared.

The TDZ acts as a safety mechanism to catch potential issues caused by accessing variables too early. It helps enforce the best practice of declaring variables before using them, which promotes cleaner and more predictable code.

Here's an example to illustrate the Temporal Dead Zone:

| console.log(message); // Output: ReferenceError: Cannot access 'message' before initialization let message = "Hello!"; |
| --- |

In this example, we try to access the message variable before its declaration. JavaScript throws a reference error because the variable is still in the TDZ and cannot be accessed until the line let message = "Hello!"; is reached.

**Q.3 Difference between var & let?**

In simple terms, the main difference between var and let in JavaScript is how they behave in terms of scope and hoisting.

Scope: Variables declared with var are function-scoped, while variables declared with let are block-scoped. This means that a variable declared with var is accessible throughout the entire function in which it is declared, while a variable declared with let is only accessible within the block (enclosed within curly braces) where it is defined.

Hoisting: Variables declared with var are hoisted to the top of their scope during the compilation phase, which means you can use them before they are physically written in your code. On the other hand, variables declared with let are also hoisted, but they remain in the "Temporal Dead Zone" until they are declared, causing a reference error if accessed before their declaration.

Here's an example to illustrate the difference:

| **console.log(x); // Output: undefined var x = 5;  console.log(y); // Output: ReferenceError: y is not defined let y = 10;** |
| --- |

**Q.4 What are the major features introduced in ECMAScript 6?**

ECMAScript 6, also known as ECMAScript 2015 or ES6, introduced several significant features and enhancements to JavaScript. Some of the major features introduced in ECMAScript 6 include:

1. let and const Declarations: The let and const keywords were introduced as block-scoped alternatives to var for declaring variables. let allows you to declare variables with block scope, while const is used for declaring constants that cannot be re-assigned.
2. Arrow Functions: Arrow functions provide a concise syntax for defining functions, using the => arrow notation. They have implicit return and lexical scoping of this, making them more expressive and convenient for writing functions.
3. Classes: ECMAScript 6 introduced a new syntax for defining classes in JavaScript, providing syntactical sugar over the existing prototype-based inheritance. The class keyword allows for more familiar class-oriented programming, with support for constructors, methods, and inheritance.
4. Enhanced Object Literals: Object literals were enhanced with new features in ES6. This includes shorthand syntax for defining object properties and methods, computed property names, and the ability to define getters and setters within object literals.
5. Template Literals: Template literals provide a more powerful way of working with strings. They allow for multi-line strings, string interpolation with placeholders using ${}, and easy expression embedding.
6. Destructuring Assignment: Destructuring assignment enables extracting values from arrays or objects into separate variables. It provides a concise and convenient way to assign values or access nested properties.
7. Spread Operator: The spread operator (...) allows expanding an iterable (like an array or string) into individual elements. It is useful for functions that accept variable arguments or for creating shallow copies of arrays and objects.
8. Modules: ECMAScript 6 introduced native support for modules, allowing developers to organize their code into reusable modules. Modules have their own scope and can be imported and exported using import and export statements.
9. Promises: Promises were introduced as a built-in way to handle asynchronous operations in a more structured manner. Promises simplify asynchronous programming by representing the eventual result or failure of an asynchronous operation.
10. These are just a few of the major features introduced in ECMAScript 6. ES6 brought significant improvements to the JavaScript language, making it more expressive, readable, and powerful for modern web development.

**Q.7 What’s difference between map & forEach?**

Both map() and forEach() are array methods in JavaScript used to iterate over the elements of an array. However, they differ in how they handle the returned values and their impact on the original array.

1. **Return Value:**

* map(): The map() method creates a new array by applying a provided function to each element of the original array. It returns a new array containing the results of applying the

function to each element. The original array remains unchanged.

* forEach(): The forEach() method iterates over the array and executes a provided function for each element. It does not return anything. It is primarily used for performing a side effect (e.g., logging, modifying external variables) on each element of the array.

1. **Modifying the Original Array:**

* map(): The map() method does not modify the original array. Instead, it creates a new array with the transformed values based on the provided function.
* forEach(): The forEach() method does not create a new array. It only iterates over the existing elements in the array and executes the provided function for each element. It does not modify the original array.

| **const numbers = [1, 2, 3];  const squaredNumbers = numbers.map((num) => num \* num); console.log(squaredNumbers); // Output: [1, 4, 9] console.log(numbers); // Output: [1, 2, 3] (original array remains unchanged)  numbers.forEach((num) => {  console.log(num \* num); }); // Output: // 1 // 4 // 9** |
| --- |

**Q.8 How can you destructure objects and arrays in ES6?**

In ES6 (ECMAScript 2015) and later versions of JavaScript, you can use destructuring assignment to extract values from objects and arrays into separate variables. Destructuring provides a concise and convenient way to access specific elements without explicitly accessing them through indexing or property names.

**Destructuring Objects:**

To destructure an object, you use curly braces {} and provide the variable names that correspond to the object's property names you want to extract.

| **const person = {  name: "John",  age: 30,  city: "New York" };  // Destructuring assignment const { name, age, city } = person;  console.log(name); // Output: "John" console.log(age); // Output: 30 console.log(city); // Output: "New York"** |
| --- |

In this example, the object person is destructured into individual variables name, age, and city, which directly correspond to the property names in the object. The values are assigned to the respective variables.

**Destructuring Arrays:**

To destructure an array, you use square brackets [] and provide the variable names to extract values in the same order as they appear in the array.

| **const colors = ["red", "green", "blue"];  // Destructuring assignment const [firstColor, secondColor, thirdColor] = colors;  console.log(firstColor); // Output: "red" console.log(secondColor); // Output: "green" console.log(thirdColor); // Output: "blue"** |
| --- |

**Default Values and Rest Pattern:**

**Destructuring also allows you to provide default values for variables in case the corresponding property or element does not exist. Additionally, the rest pattern (...) can be used to capture the remaining elements into a new array or object**

| **const person = {  name: "John",  age: 30 };  const { name, age, city = "New York" } = person; console.log(city); // Output: "New York" (default value)  const numbers = [1, 2, 3, 4, 5]; const [firstNumber, secondNumber, ...restNumbers] = numbers; console.log(firstNumber); // Output: 1 console.log(secondNumber); // Output: 2 console.log(restNumbers); // Output: [3, 4, 5]** |
| --- |

**Q.9 How can you define default parameter values in ES6 functions?**

In ES6 (ECMAScript 2015) and later versions of JavaScript, you can define default parameter values for function parameters. Default parameters allow you to specify fallback values that will be used if the corresponding arguments are not provided or are explicitly set to undefined when invoking the function.

Here's the syntax for defining default parameter values in ES6 functions:

| function functionName(parameter1 = defaultValue1, parameter2 = defaultValue2) {  // Function code } |
| --- |

In this syntax:

parameter1 and parameter2 are the function parameters.

defaultValue1 and defaultValue2 are the default values assigned to the parameters if no arguments or undefined are passed.

**Q.10 What is the purpose of the spread operator (...) in ES6?**

In ES6 (ECMAScript 2015) and later versions of JavaScript, the spread operator (...) serves multiple purposes and provides a concise way to work with arrays, objects, and function arguments.

**Array Spread:**

The spread operator can be used to spread the elements of an existing array into a new array. It allows you to copy an array or combine multiple arrays easily.

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

In this example, the spread operator ... is used to spread the elements of array1 and array2 into the combinedArray.

**Object Spread:**

The spread operator can also be used to create a new object by spreading the properties of an existing object into a new object. It allows you to clone an object or merge multiple objects easily.

| const object1 = { a: 1, b: 2 }; const object2 = { c: 3, d: 4 };  const combinedObject = { ...object1, ...object2 }; console.log(combinedObject); // Output: { a: 1, b: 2, c: 3, d: 4 } |
| --- |

In this example, the spread operator is used to spread the properties of object1 and object2 into the combinedObject.

**Function Arguments:**

The spread operator can be used to pass multiple arguments to a function by spreading the elements of an array or iterable as individual arguments.

| **function sum(a, b, c) {  return a + b + c; }  const numbers = [1, 2, 3]; const result = sum(...numbers); console.log(result); // Output: 6** |
| --- |

In this example, the spread operator ... is used to spread the elements of the numbers array as individual arguments to the sum() function.