Assignment 4

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

Sol-> Hoisting in JavaScript is a behavior where variable and function declarations are moved to the top of their respective scopes during the compilation phase, before the code is executed. This can sometimes lead to unexpected results if developers are not aware of how hoisting works.

There are two types of hoisting in JavaScript:

1. Variable Hoisting:

When you declare a variable using the `var` keyword, the declaration is moved to the top of its scope, but the initialization (assignment of value) remains in place. This means that you can access the variable before it is declared, but it will have an initial value of `undefined`.

Example:

```

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

var x = 10;

console.log(x); // Output: 10

```

What actually happens during execution:

```

var x; // Variable declaration is hoisted

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

x = 10; // Initialization is not hoisted

console.log(x); // Output: 10

```

It's important to note that `let` and `const` declarations are also hoisted, but they are not initialized until the actual line of code where the declaration appears, resulting in a "Temporal Dead Zone" (TDZ) error if accessed before initialization.

2. Function Hoisting:

Function declarations are entirely hoisted, including the function body. This means you can call a function even before it is declared in the code.

Example:

```

sayHello(); // Output: "Hello, World!"

function sayHello() {

console.log("Hello, World!");

}

```

What actually happens during execution:

```

function sayHello() {

console.log("Hello, World!");

}

sayHello(); // Output: "Hello, World!"

```

However, it's important to note that function expressions (functions assigned to variables) are not hoisted like function declarations. Only the variable declaration itself is hoisted, not the function assignment.

Example:

```

sayHello(); // Error: sayHello is not a function

var sayHello = function() {

console.log("Hello, World!");

};

```

What actually happens during execution:

```

var sayHello; // Variable declaration is hoisted

sayHello(); // Error: sayHello is not a function

sayHello = function() {

console.log("Hello, World!");

};

```

To avoid confusion and unexpected behavior due to hoisting, it is generally recommended to declare variables at the top of their scope and to use `let` and `const` for better scoping behavior and to make the code more predictable. Similarly, it's good practice to define functions before calling them to ensure smooth execution.

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

Sol-> The Temporal Dead Zone (TDZ) is a behavior in JavaScript that occurs when using variables declared with the **let** and **const** keywords. It is a specific phase during the code execution where variables are hoisted to their respective scopes but are not accessible until they are formally declared in the code.

Working of temporal dead zone:

When a block scope(such as function,if statement ,loop) is entered,variable declared with let and const are hoisted to the top of the block, similar to variable hoisting with var.

However,unlike variable declared with var ,variable declared with let and const are not initialized with a default value (‘**undefined**”)during hoisting .Instead they remain uninitilized until there explicit decleration is encountered.

The TDZ is the period between the start of the block scope and the actual declaration of the variable.

During the TDZ, if you try to access or use a variable declared with **let** or **const**, a **ReferenceError** will be thrown. This is because the variable exists in the scope but is in an uninitialized state and not yet accessible.

Only after the explicit declaration of the variable is encountered in the code is it considered to be in the initialized state and can be accessed without any errors.

The variable declaration serves as a "gatekeeper" that signals the end of the TDZ for that particular variable.

Ex-

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

let x = 5;

if (true) {

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

let y = 10;

}

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

const z = 15;

In the example above, **x**, **y**, and **z** are declared with **let** and **const**. However, trying to access them before their actual declarations result in **ReferenceError** because they are in the TDZ. Only after their respective declarations are encountered can they be accessed without any errors.

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

Sol-> The main differences between `var` and `let` in JavaScript are related to their scoping behavior and hoisting. Here are the key distinctions:

1. Scoping:

- `var` variables are function-scoped or globally scoped, meaning they are accessible throughout the entire function or global scope in which they are defined.

- `let` variables are block-scoped, meaning they are only accessible within the block (enclosed by curly braces) where they are defined. Block scopes include if statements, loops, and functions.

2. Hoisting:

- Variables declared with `var` are hoisted to the top of their respective function or global scope during the compilation phase. This means you can access and use `var` variables before their actual declarations in the code. However, they are initialized with a default value of `undefined` until their assignment is encountered.

- Variables declared with `let` are also hoisted, but they remain in the Temporal Dead Zone (TDZ) until their explicit declaration is encountered in the code. Accessing `let` variables before their declaration will result in a `ReferenceError`.

3. Redeclaration:

- `var` allows you to redeclare the same variable within the same scope without generating an error. The subsequent declaration simply overrides the previous value.

- `let` does not allow redeclaration of the same variable within the same block scope. Attempting to redeclare a `let` variable will result in a `SyntaxError`.

4. Variable Initialization:

- `var` variables are initialized with a default value of `undefined` during hoisting, if not explicitly assigned a value.

- `let` variables are not initialized during hoisting. They remain uninitialized until their explicit assignment is encountered in the code.

5. Scope Pollution:

- Due to function-level scoping, variables declared with `var` can lead to unintentional scope pollution. If `var` variables are declared inside loops or conditional statements, they are accessible outside those blocks, which can potentially cause bugs or unintended behavior.

- `let` variables, being block-scoped, are less prone to scope pollution as they are limited to the specific block where they are defined.

Given these differences, it is generally recommended to use `let` and adhere to block scoping whenever possible, as it provides better control over variable access and reduces the risk of unintended side effects caused by hoisting and scope pollution.

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

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

1. Block-Scoped Variables: The `let` and `const` keywords were introduced to declare block-scoped variables, allowing variables to be limited in scope to the block where they are defined (such as inside a loop or if statement), improving code clarity and reducing scope-related issues.

2. Arrow Functions: Arrow functions provide a more concise syntax for creating functions, with a simplified syntax and lexical scoping of `this`. They are especially useful for writing shorter and more readable code for functions that do not need their own `this` binding or the `arguments` object.

3. Template Literals: Template literals allow for more flexible and expressive string interpolation and multi-line strings. They are denoted by backticks (`) and can contain embedded expressions using `${}` syntax.

4. Enhanced Object Literals: Object literals gained new features, including shorthand property and method definitions, computed property names, and the ability to define properties and methods without the need for the `function` keyword.

5. Destructuring Assignment: Destructuring assignment provides an easy way to extract values from arrays or objects into individual variables, simplifying the process of accessing and assigning multiple values.

6. Default Parameters: Default parameter values can now be assigned directly in function declarations, allowing functions to have default values for parameters if they are not explicitly provided.

7. Rest and Spread Operators: The rest (`...`) and spread (`...`) operators were introduced. The rest operator allows gathering multiple function arguments into an array, while the spread operator allows spreading elements of an array or object into individual elements.

8. Classes: ECMAScript 6 introduced class syntax for creating objects with a more traditional object-oriented programming (OOP) approach. Classes provide a clearer and more structured way to define and instantiate objects, with support for constructors, inheritance, and static methods.

9. Modules: ECMAScript 6 added native support for modules, allowing developers to define modules with separate files and export/import functionality to encapsulate and organize code, improving modularity and code reuse.

10. Promises: Promises provide a standardized way to handle asynchronous operations, making it easier to write and manage asynchronous code. Promises allow for better control flow, error handling, and avoiding callback hell.

**Q.5** What is the difference between **let** and **const** ?

Sol-> The main difference between `let` and `const` in JavaScript lies in their mutability:

1. Mutability:

- Variables declared with `let` can be reassigned new values. This means that you can assign a different value to a `let` variable at any point in your code.

- Variables declared with `const`, on the other hand, are constants and cannot be reassigned once they are assigned a value. Once a `const` variable is assigned, its value remains constant throughout the code execution.

Here's an example to illustrate the difference:

```

let x = 5;

x = 10; // Valid - Reassigning value

const y = 15;

y = 20; // Invalid - Cannot reassign value to a const variable

```

In the example above, `x` is declared with `let`, allowing it to be reassigned a new value (`10`). However, `y` is declared with `const`, and attempting to reassign a new value to it (`20`) will result in an error.

It's important to note that although `const` variables are immutable (cannot be reassigned), their contents can still be mutable. This means that for complex data types like objects and arrays, you can modify the properties or elements of a `const` variable. The immutability of `const` applies only to the variable reference itself.

```

const person = {

name: 'John',

age: 30

};

person.age = 35; // Valid - Modifying object property

const numbers = [1, 2, 3];

numbers.push(4); // Valid - Modifying array elements

```

In the example above, the `person` object and the `numbers` array are declared as `const`, so you cannot reassign them to new objects or arrays. However, you can still modify their properties or elements.

To summarize, the difference between `let` and `const` can be understood as:

- `let` variables are mutable and can be reassigned new values.

- `const` variables are constants and cannot be reassigned once assigned, but their contents can still be mutable for objects and arrays.

**Q.6** What is template literals in ES6 and how do you use them?

Sol-> Template literals, introduced in ECMAScript 6 (ES6), provide an improved way to work with strings in JavaScript. They allow for easy interpolation of variables and expressions within strings, support multi-line strings, and offer additional string formatting capabilities.

Here's how you can use template literals in ES6:

1. Basic Syntax:

- Template literals are denoted by backticks (` `) instead of single or double quotes used for regular strings.

- Within a template literal, you can directly embed variables or expressions using the `${}` syntax.

2. Variable Interpolation:

- To include a variable value within a template literal, enclose it within `${}`.

- The variable's value will be evaluated and inserted into the string at that position.

```

const name = 'John';

const age = 30;

const greeting = `Hello, my name is ${name} and I am ${age} years old.`;

console.log(greeting);

// Output: Hello, my name is John and I am 30 years old.

```

In the example above, the `name` and `age` variables are interpolated within the template literal using `${}`. The resulting string `greeting` combines the variable values with the surrounding text.

3. Expression Evaluation:

- Template literals can also contain JavaScript expressions within `${}`.

- These expressions can be any valid JavaScript code, including arithmetic operations, function calls, and conditional statements.

```

const a = 5;

const b = 10;

const result = `The sum of ${a} and ${b} is ${a + b}.`;

console.log(result);

// Output: The sum of 5 and 10 is 15.

```

In the above example, the expression `${a + b}` within the template literal evaluates the sum of `a` and `b` and includes the result in the resulting string.

4. Multi-line Strings:

- Template literals also support multi-line strings without the need for escape characters or concatenation.

- Line breaks and indentation within the template literal are preserved in the resulting string.

```

const multiLineString = `

This is a multi-line string

using template literals.

Line breaks and indentation are preserved.

`;

console.log(multiLineString);

/\*

Output:

This is a multi-line string

using template literals.

Line breaks and indentation are preserved.

\*/

```

In the above example, the template literal spans multiple lines, and the resulting string preserves the line breaks and indentation within it.

Template literals provide a more concise and expressive way to work with strings, making code more readable and reducing the need for string concatenation or escape characters. They are widely used in modern JavaScript development for tasks such as generating dynamic strings, constructing URLs, and rendering HTML templates.

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

Sol-> Both `map()` and `forEach()` are array methods in JavaScript that allow you to iterate over an array and perform operations on its elements. However, there are a few key differences between them:q

1. Return Value:

- `map()`: The `map()` method returns a new array containing the results of applying a provided function to each element of the original array. It creates a new array by transforming each element based on the callback function's return value.

- `forEach()`: The `forEach()` method does not return anything. It simply iterates over the array and executes a provided callback function on each element.

2. Usage:

- `map()`: `map()` is often used when you want to create a new array by performing some operation or transformation on each element of the original array. It is commonly used when you need to derive a new array based on the existing array.

- `forEach()`: `forEach()` is useful when you want to perform a specific operation or action on each element of the array, such as logging values, modifying the original array in place, or executing a function for its side effects. It is commonly used for iteration purposes.

3. Modifying the Original Array:

- `map()`: The original array remains unchanged when using `map()`. It creates a new array based on the transformation applied to each element, leaving the original array intact.

- `forEach()`: `forEach()` does not create a new array, but it allows modifying the original array in place. You can modify the elements of the array within the callback function. However, care should be taken not to change the array structure, as `forEach()` is primarily intended for iteration and side effects rather than modifying the array structure.

Here's an example to illustrate the difference:

```

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

const squaredNumbers = numbers.map(num => num \*\* 2);

console.log(squaredNumbers);

// Output: [1, 4, 9, 16, 25]

numbers.forEach(num => console.log(num));

// Output: 1, 2, 3, 4, 5

```

In the example above, `map()` is used to transform each element of the `numbers` array by squaring them, resulting in a new array `squaredNumbers`. On the other hand, `forEach()` is used to iterate over the `numbers` array and log each element to the console.

In summary, the key differences between `map()` and `forEach()` are the return value (creating a new array vs. no return) and the primary use case (transformation vs. iteration with side effects). Understanding these differences will help you choose the appropriate method based on your specific requirements.

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

Sol-> In ES6 (ECMAScript 2015), you can use object and array destructuring to extract values from objects and arrays into individual variables. It provides a concise and convenient way to unpack values from data structures.

1. Object Destructuring:

To destructure an object, you can use curly braces `{}` and assign variable names corresponding to the keys of the object. Here's an example:

```

// Example object

const person = {

firstName: 'John',

lastName: 'Doe',

age: 30,

};

// Destructuring the object

const { firstName, lastName, age } = person;

console.log(firstName); // Output: John

console.log(lastName); // Output: Doe

console.log(age); // Output: 30

```

2. Array Destructuring:

To destructure an array, you can use square brackets `[]` and assign variables based on the order of elements in the array. Here's an example:

```

// Example array

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

// Destructuring the array

const [first, second, ...rest] = numbers;

console.log(first); // Output: 1

console.log(second); // Output: 2

console.log(rest); // Output: [3, 4, 5]

```

3. Default Values:

You can also provide default values in case the property or index doesn't exist in the object or array, respectively:

```

const person = {

firstName: 'John',

lastName: 'Doe',

};

const { firstName, lastName, age = 25 } = person;

console.log(firstName); // Output: John

console.log(lastName); // Output: Doe

console.log(age); // Output: 25 (default value since 'age' is not present in the object)

```

```

const numbers = [1, 2];

const [first, second, third = 3] = numbers;

console.log(first); // Output: 1

console.log(second); // Output: 2

console.log(third); // Output: 3 (default value since the third element is not present in the array)

```

Object and array destructuring can significantly simplify your code and make it more readable when working with complex data structures.

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

Sol-> In ES6 (ECMAScript 2015) and beyond, you can define default parameter values for function parameters directly within the function's parameter list. This feature allows you to set default values for parameters when they are not provided by the caller. To define default parameter values in ES6 functions, you can use the following syntax:

```

function functionName(param1 = defaultValue1, param2 = defaultValue2, ...) {

// function body

}

```

Here's a brief explanation:

1. `functionName`: This is the name of your function.

2. `param1`, `param2`, ...: These are the parameters that your function accepts.

3. `defaultValue1`, `defaultValue2`, ...: These are the default values assigned to the parameters if no value is provided during the function call.

Here's a simple example of a function with default parameter values:

```

function greet(name = "Guest") {

console.log(`Hello, ${name}!`);

}

// When no argument is provided, the default value "Guest" will be used.

greet(); // Output: Hello, Guest!

// When an argument is provided, it will override the default value.

greet("John"); // Output: Hello, John!

```

In this example, the `greet` function accepts a single parameter `name`, and if no argument is passed during the function call, it will default to "Guest". If an argument is provided, it will use that argument as the value of the `name` parameter.

By using default parameter values, you can make your functions more flexible and concise, reducing the need for additional checks for undefined or null values.

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

Sol-> The spread operator (`...`) in ES6 (ECMAScript 2015) is a powerful feature that serves multiple purposes. It allows you to manipulate arrays and objects in a concise and convenient way. The primary purposes of the spread operator are as follows:

1. Array Spread:

The spread operator can be used to expand an array into individual elements. This is helpful in scenarios where you want to pass the elements of an array as individual arguments to a function or create a new array by combining the elements of multiple arrays.

Example - Passing array elements as function arguments:

```

const numbers = [1, 2, 3];

console.log(...numbers); // Output: 1 2 3

```

Example - Creating a new array by combining two arrays:

```

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

const combinedArray = [...arr1, ...arr2];

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

```

2. Object Spread:

The spread operator can also be used to create new objects by merging properties from existing objects or adding new properties to an object.

Example - Creating a new object by merging properties from two objects:

```

const obj1 = { name: 'John', age: 30 };

const obj2 = { occupation: 'Engineer', city: 'New York' };

const mergedObject = { ...obj1, ...obj2 };

console.log(mergedObject);

// Output: { name: 'John', age: 30, occupation: 'Engineer', city: 'New York' }

```

Example - Adding new properties to an object:

```

const person = { name: 'Alice', age: 25 };

const extendedPerson = { ...person, country: 'USA', occupation: 'Designer' };

console.log(extendedPerson);

// Output: { name: 'Alice', age: 25, country: 'USA', occupation: 'Designer' }

```

3. Function Arguments:

The spread operator can also be used as a parameter to accept a variable number of arguments in a function.

Example - Accepting a variable number of arguments in a function:

```

function sum(...numbers) {

return numbers.reduce((acc, num) => acc + num, 0);

}

console.log(sum(1, 2, 3, 4)); // Output: 10

```

In summary, the spread operator in ES6 allows you to easily manipulate arrays and objects, create new arrays and objects by combining existing ones, and handle functions with a variable number of arguments. It is a versatile tool that enhances code readability and simplifies common programming tasks.