Q.1 Explain Hoisting in JavaScript?

Ans:- Hoisting in JavaScript is a mechanism by which variable and function declarations are moved to the top of their containing scope during the compilation phase. This means that regardless of where variables and functions are declared within a scope, they are interpreted as if they were declared at the beginning of that scope. However, only the declarations are hoisted, not the initializations or assignments.

Q.2 Explain Temporal Dead Zone?

Ans:- The Temporal Dead Zone (TDZ) is a behavior in JavaScript that occurs when variables declared with let and const are not accessible before they are declared in their containing block. During the TDZ, accessing such variables will result in a ReferenceError. The TDZ ends when the variable declaration is reached in the code. It is a period between the start of the scope and the actual declaration where the variable exists but cannot be accessed.

Q.3 Difference between var & let?

Ans:- The main difference between var and let in JavaScript is their scoping behavior.

var has function-level scope, meaning a variable declared with var is accessible throughout the entire function in which it is defined, regardless of block scope. It is also hoisted to the top of its containing scope during compilation.

let, on the other hand, has block-level scope. A variable declared with let is limited in scope to the block (enclosed by curly braces) in which it is defined. It is not hoisted and remains inaccessible until its declaration is reached in the code.

In summary, var has function-level scope and is hoisted, while let has block-level scope and is not hoisted.

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

Ans:- ECMAScript 6 (ES6), also known as ECMAScript 2015, introduced several major features to JavaScript. Some of the key features are:

Arrow Functions: A concise syntax for defining functions, with automatic binding of this.

Classes: A new syntax for defining classes, along with support for constructors, inheritance, and static methods.

Block-Scoped Variables: The let and const keywords allow the declaration of block-scoped variables, providing better control over variable scope.

Template Literals: Enhanced string literals that support multiline strings and string interpolation using backticks (`).

Destructuring Assignment: Simplifies the extraction of values from arrays and objects using a concise syntax.

Enhanced Object Literals: Adds new syntax for defining object properties and methods, including shorthand property names and computed property names.

Default Parameters: Allows default values to be specified for function parameters.

Rest and Spread Operators: The rest operator (...) gathers function arguments into an array, while the spread operator (...) spreads arrays into separate elements.

Promises: Provides a built-in mechanism for handling asynchronous operations, making it easier to write asynchronous code.

Modules: Introduces a standardized module system for organizing and importing/exporting code between files.

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

Ans:- The main difference between `let` and `const` in JavaScript is the mutability of the variables they declare.

`let` allows variable values to be reassigned. It is useful for variables that need to change their value over time.

`const`, on the other hand, declares variables that are constants, meaning their values cannot be reassigned once they are initialized. It is suitable for variables that are meant to remain unchanged throughout the program.

In summary, `let` variables are mutable and can be reassigned, while `const` variables are constants and their values cannot be changed after initialization.

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

Ans:- Template literals, introduced in ES6 (ECMAScript 2015), are an enhanced syntax for defining strings in JavaScript. They allow for easy embedding of expressions and multiline strings.

Template literals are denoted by backticks ( ) instead of single or double quotes. Within template literals, expressions can be embedded using the **${}** syntax.

Q.7 What's difference between map & forEach?

Ans:-
The main difference between **map** and **forEach** in JavaScript is their return value and the purpose they serve.

forEach is an array method that iterates over each element in an array and performs a specified action or callback function on each element. It does not return a new array; instead, it modifies the existing array in place.

map, on the other hand, also iterates over each element in an array and executes a callback function. However, it returns a new array with the results of applying the callback function to each element. It creates a transformed version of the original array, leaving the original array unchanged.

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

Ans:- In ES6 (ECMAScript 2015), you can use destructuring assignment to extract values from objects and arrays in a concise manner.

To destructure objects, you can use the following syntax:

const { property1, property2 } = object

This assigns the values of `property1` and `property2` from the `object` to the variables with the same names.

To destructure arrays, you can use a similar syntax:

const [element1, element2] = array

This assigns the values of `element1` and `element2` from the `array` to the variables with the same names.

You can also destructure objects and arrays while declaring variables:

const { property1, property2 } = { property1: 'value1', property2: 'value2' }

const [element1, element2] = ['value1', 'value2']

In addition to simple variable assignment, you can also assign default values and use nested destructuring for more complex structures.

Destructuring makes it easier to extract specific values from objects and arrays without accessing them directly through dot notation or indexing.

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

Ans:- In ES6, you can define default parameter values for function parameters using a concise syntax.

To define a default parameter value, you can assign a value directly to the function parameter in the function declaration:

function myFunction(param = defaultValue)

In the example above, if the **param** is not provided when invoking **myFunction()**, it will default to **defaultValue**.

Here's an example demonstrating the usage of default parameter values:

function greet(name = 'Anonymous') { console.log(`Hello, ${name}!`); } greet();

In the **greet** function, if the **name** parameter is not provided, it defaults to **'Anonymous'**. This allows you to provide a default value that will be used if the parameter is omitted or passed as **undefined**.

Default parameter values provide a convenient way to handle cases where a parameter is not provided or lacks a value.

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

Ans:-
The spread operator (**...**) in ES6 has two primary purposes: to spread

elements of an iterable (like an array or string) into individual elements, and to copy properties from one object to another.

1. Spread Operator for Arrays and Iterables: When used with an array or any iterable, the spread operator expands the elements into individual values. It allows you to pass multiple elements as arguments to a function, concatenate arrays, or create a copy of an array.
2. Spread Operator for Objects: The spread operator can also be used to copy properties from one object to another or merge multiple objects into a new one.

   In short, the spread operator (**...**) in ES6 allows for the expansion of elements in arrays/iterables or the merging/copying of properties from one object to another. It provides a concise and versatile way to work with arrays, objects, and other iterable data structures.