Q.1 Explain Hoisting in JavaScript

Ans:

Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their respective scopes during the compilation phase, before the actual execution of the code. This means that regardless of where variables and functions are declared in the code, they are treated as if they are declared at the top of their scope.

Hoisting can be divided into two types: hoisting of variable declarations and hoisting of function declarations.

1. Variable Hoisting:

When a variable is declared using the `var` keyword, the declaration is hoisted to the top of its scope while the assignment of a value to the variable remains in its original place.

For example:

```
console.log(x); // undefined

var x = 5;
```

The above code is interpreted by JavaScript as:

```
var x; // Variable declaration is hoisted

console.log(x); // undefined

x = 5; // Variable assignment remains in its original place
```

As you can see, the variable `x` is hoisted to the top, which is why it does not throw an error when we try to access it before its actual declaration. However, the assignment of `x = 5` is not hoisted, so it retains its original position and is executed in the normal order.

2. Function Hoisting:

Function declarations in JavaScript are also hoisted to the top of their scope. This means that you can call a function before its actual declaration in the code. For example:

```
sayHello(); // "Hello"
```

```
function sayHello() {

  console.log("Hello");

}
```

In the above code, the function `sayHello` is hoisted to the top, allowing us to call it before its actual declaration.

Q.2 Explain Temporal Dead Zone?

Ans:

1. The TDZ(**Temporal Dead Zone)** occurs when accessing a variable before its declaration with `let` or `const` keywords.

2. Variables declared with `let` and `const` are hoisted to the top of their scope, but they remain uninitialized within the TDZ.

3. Trying to access a variable within its TDZ results in a "ReferenceError" at runtime.

4. The TDZ promotes cleaner code by preventing the use of variables before they are properly declared and initialized.

5. The TDZ applies only to variables declared with `let` and `const`, not to variables declared with `var`.

An example that illustrates the Temporal Dead Zone:

```
console.log(x); // ReferenceError: Cannot access 'x' before initialization

let x = 5;
```

In the above code, we try to access the variable `x` before its declaration. However, since `x` is declared with `let`, it is in the Temporal Dead Zone until the line `let x = 5;` is reached. Therefore, trying to access `x` before its declaration throws a `ReferenceError`.

The TDZ ensures that variables are accessed only after they have been properly declared and initialized. It helps prevent potential issues caused by accessing variables in an undefined state, promoting more reliable and predictable code.

Q.3 Difference between var & let?

Ans:
The differences between `var` and `let` in JavaScript are:

1. Scope:

   - `var`: Variables declared with `var` have function scope or global scope. This means they are accessible throughout the entire function or globally if declared outside any function.

   - `let`: Variables declared with `let` have block scope. They are only accessible within the block where they are declared, which is typically a set of curly braces `{}`.

2. Hoisting:

   - `var`: Variables declared with `var` are hoisted to the top of their scope during the compilation phase. This means that the variable declarations are moved to the top, while the assignments remain in their original positions. The variables are initialized with the value `undefined` during the hoisting process.

   - `let`: Variables declared with `let` are also hoisted to the top of their scope, but they remain uninitialized in the Temporal Dead Zone (TDZ). Trying to access a variable within its TDZ results in a runtime error. The variables are initialized only at the point of their actual declaration.

3. Redeclaration:

   - `var`: Variables declared with `var` can be redeclared within the same scope without any issues. In other words, you can use the same variable name again within the same function or global scope.

   - `let`: Variables declared with `let` cannot be redeclared within the same block scope. If you attempt to redeclare a variable with the same name within the same block, it will result in a syntax error.

4. Access in closures:

- `var`: Variables declared with `var` have function scope, which means they are accessible within closures created within the same function. Closures are functions that remember the environment in which they were created, including the variables.

- `let`: Variables declared with `let` have block scope, and they behave more predictably when used within closures. They are accessible within closures created within the same block scope, ensuring that each closure has its own independent copy of the variable.

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

Ans:
The Major features introduced in ECMAScript 6 (ES6):

1. `let` and `const` Declarations: Introduced block-scoped variable declarations with `let` and constant declarations with `const`, providing more predictable scoping rules compared to `var`.

2. Arrow Functions: Introduced a concise syntax for writing functions using the arrow (`=>`) notation, allowing for shorter and more expressive function definitions, and automatically capturing the surrounding `this` value.

3. Classes: Introduced class syntax for creating objects and implementing object-oriented programming (OOP) concepts, including constructors, inheritance, and static methods, making it easier to work with prototypal inheritance.

4. Template Literals: Introduced a new syntax for creating strings using backticks (` `), allowing for string interpolation, multi-line strings, and embedding expressions directly within strings, improving code readability.

5. Destructuring Assignment: Provided a concise syntax for extracting values from arrays or objects and assigning them to variables in a single expression, making it easier to work with complex data structures.

6. Modules: Introduced a standardized module system for JavaScript, allowing for better organization, reuse, and encapsulation of code by defining dependencies and exporting/importing functionality between modules.

7. Promises: Introduced a built-in mechanism for handling asynchronous operations using promises, which are objects representing the eventual completion (or failure) of an asynchronous operation. Promises simplify asynchronous programming and improve readability by allowing a more linear and declarative coding style.

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

Ans:

The main difference between `let` and `const` in JavaScript lies in their assignment and reassignment capabilities:

1. Assignment and Reassignment:

- `let`: Variables declared with `let` can be assigned a value upon declaration and can also be reassigned later in the code.

- `const`: Variables declared with `const` must be assigned a value upon declaration, and they cannot be reassigned later. The value assigned to a `const` variable is constant and cannot be changed.

```
let x = 5;
x = 10; // Reassignment allowed
console.log(x); // Output: 10

const y = 5;
y = 10; // Error: Cannot reassign a constant variable
console.log(y);
```

2. Mutable vs. Immutable:

- `let`: Variables declared with `let` are mutable. Their values can be changed or mutated during the execution of the program.

- `const`: Variables declared with `const` are immutable. Once assigned a value, that value cannot be changed or mutated throughout the program.

```
let array = [1, 2, 3];
array.push(4); // Mutation allowed
console.log(array); // Output: [1, 2, 3, 4]

const pi = 3.14159;
pi = 3; // Error: Cannot assign a new value to a constant variable
console.log(pi);
```

3. Block Scope:

   - Both `let` and `const` variables are block-scoped. They are accessible only within the block where they are declared, such as within loops, conditionals, or functions.

```
{
  let blockScoped = "Hello";
  const constantValue = "World";
}

console.log(blockScoped); // ReferenceError: blockScoped is not defined
console.log(constantValue); // ReferenceError: constantValue is not defined
```

In the example, the `blockScoped` variable and `constantValue` constant are declared within a block, and attempting to access them outside the block results in a reference error.

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

Ans:

1. Template literals provide a convenient syntax for creating strings in JavaScript.

2. They are defined using backticks (` `) instead of single or double quotes.

3. Placeholders `${}` can be included within the template literal for variable interpolation and expression evaluation.

4. Variables, expressions, or function calls can be included within the placeholders, and their values are evaluated and replaced at runtime.

5. Template literals support easy interpolation of variables and expressions within the string.

6. They eliminate the need for manual string concatenation using the `+` operator.

7. Template literals support multi-line strings without needing explicit line breaks or concatenation.

8. New lines and formatting within the template literal are preserved in the resulting string.

9. Template literals improve code readability and maintainability by allowing a more expressive and concise syntax for creating strings.

Example usage:

```
const name = "John";
const age = 30;

// Basic usage
const greeting = `Hello, ${name}!`;

// Expressions within template literals
const info = `Name: ${name}, Age: ${age}, Next year's age: ${age + 1}`;

// Multi-line strings
const message = `This is
a multi-line
string.`;
```

Q.7 What’s difference between map & forEach

Ans:
The main differences between `map` and `forEach` in JavaScript are:

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.

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

2. Mutation of Original Array:

   - `map`: The `map` method does not mutate the original array. It creates and returns a new array with the transformed elements.

   - `forEach`: The `forEach` method does not create a new array. It iterates over the elements of the original array but does not modify it.

3. Usage with Return Value:

   - `map`: The return value of the `map` method can be assigned to a new variable or used directly in further computations.

   - `forEach`: Since `forEach` does not return a value, it is primarily used for executing a provided function for its side effects (such as logging, updating external state, or performing other actions).

4. Usage with Break Statements:

   - `map`: The `map` method does not provide a straightforward way to break or prematurely terminate the iteration. It applies the provided function to every element of the array.

   - `forEach`: The `forEach` method also does not provide a built-in mechanism to break or stop the iteration. It continues iterating over all elements of the array.

Example to illustrate the differences:

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

```
// Using map

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

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



// Using forEach

const doubledForEach = [];

numbers.forEach((num) => {

  doubledForEach.push(num * 2);

});

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

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

Ans:

In ECMAScript 6 (ES6), object and array destructuring provides a convenient way to extract values from objects and arrays and assign them to variables.

1. Object Destructuring:

Object destructuring allows you to extract values from an object and assign them to variables with corresponding names. You can destructure an object by providing a pattern that matches the structure of the object.

Example:

```
const person = {
  name: "John",
  age: 30,
```

```
  country: "USA"
};

// Destructuring
const { name, age, country } = person;

console.log(name); // Output: John
console.log(age); // Output: 30
console.log(country); // Output: USA
```

2. Array Destructuring:

Array destructuring allows you to extract values from an array and assign them to variables based on their position. You can destructure an array by providing a pattern that matches the structure of the array.

Example:

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

// Destructuring
const [first, second, third] = numbers;

console.log(first); // Output: 1
console.log(second); // Output: 2
console.log(third); // Output: 3
```

3. Default Values:

You can also provide default values for variables during destructuring. If the corresponding value is `undefined`, the default value will be used.

Example:

```javascript
const person = {
  name: "John",
  age: 30
};

// Destructuring with default values
const { name, age, country = "Unknown" } = person;

console.log(name); // Output: John
```

```
  console.log(age); // Output: 30
  console.log(country); // Output: Unknown
```

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

Ans:

Defining Default Parameter Values in ES6 Functions:

1. Default parameter values allow assigning a default value to a function parameter if no argument or an undefined argument is provided when the function is called.

2. Default parameter values are assigned during function parameter declaration.

3. They provide a convenient way to handle missing or undefined values without explicitly checking for them within the function body.

4. Default parameter values are used only when the corresponding arguments are undefined or not provided.

5. If an argument is passed explicitly (even as `null` or `undefined`), it overrides the default value.

6. Default parameter values can be expressions or function calls. They are evaluated each time the function is called.

7. Default parameter values can depend on other parameters in the same function declaration.

Example:

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

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

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

Ans:

The spread operator (`...`) in ECMAScript 6 (ES6) serves multiple purposes and provides a concise syntax for working with arrays, objects, and function arguments.

The spread operator can be used in the following ways:

1. Array Spreading:
   The spread operator can be used to expand an array into individual elements. It allows you to create a new array by combining the elements of an existing array with other elements.

   Example:

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

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

2. Object Spreading:

   The spread operator can also be used to spread the properties of an object into a new object. It allows you to create a shallow copy of an object or merge properties from multiple objects into a new object.

   Example:

```
const obj1 = { x: 1, y: 2 };
const obj2 = { z: 3 };

const mergedObject = { ...obj1, ...obj2 };
console.log(mergedObject); // Output: { x: 1, y: 2, z: 3 }
```

3. Function Argument Spreading:
   The spread operator can be used to pass individual elements of an array as separate arguments to a function.

   Example:

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

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

const result = sum(...numbers);
console.log(result); // Output: 6
```