Q.1 Hoisting in JavaScript:

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

For example, consider the following code snippet:

```javascript
console.log(x); // Output: undefined
var x = 5;
```

💡 Q.2 Temporal Dead Zone:

The Temporal Dead Zone (TDZ) is a behavior introduced in ECMAScript 6 (ES6) to address issues related to hoisting and variable visibility. It refers to the period in the code before a variable is declared, but still exists within its scope. During the TDZ, accessing the variable results in a `ReferenceError`.

Consider the following example:

```javascript
console.log(x); // Throws a ReferenceError
let x = 5;
```

Q.3 Difference between `var` and `let`:

The main differences between `var` and `let` in JavaScript are as follows:

1. **Scope**: Variables declared with `var` are function-scoped, meaning they are accessible within the entire function in which they are declared, including any nested blocks. Variables declared with `let` are block-scoped, meaning they are only accessible within the block in which they are declared.

2. **Hoisting**: Variable declarations with `var` are hoisted to the top of their scope, allowing them to be accessed before they are declared. Variable declarations with `let` are not hoisted, and accessing them before they are declared results in a `ReferenceError`.

3. **Re-declaration**: Variables declared with `var` can be re-declared within the same scope without any issues. On the other hand, re-declaring a variable with the same name using `let` within the same scope will result in a `SyntaxError`.

4. **Global scope**: Variables declared with `var` outside of any function become properties of the global object (`window` in browser environments, `global` in Node.js). Variables declared with `let` outside of any block have block scope and are not added as properties of the global object.

Overall, using `let` provides better control over variable scoping and helps prevent certain types of bugs that can occur with the use of `var`.

Q.4 Major features introduced in ECMAScript 6:

1. **let** and **const**: `let` and `const` were introduced as block-scoped alternatives to `var`. `let` allows the declaration of block-scoped variables, and `const` allows the declaration of block-scoped variables that cannot be reassigned.

2. **Arrow Functions**: Arrow functions provide a concise syntax for writing function expressions. They have a shorter syntax compared to traditional function expressions and inherit the `this` value from the surrounding context.

3. **Classes**: ES6 introduced a more structured syntax for creating classes in JavaScript, following the principles of object-oriented programming. Classes provide a way to define blueprints for creating objects and support inheritance and encapsulation.

4. **Modules**: ES6 introduced native support for modules, allowing JavaScript code to be organized into separate files. Modules enable better code organization, encapsulation, and reusability.

5. **Template Literals**: Template literals provide an improved way to work with strings in JavaScript. They allow embedding expressions and variables directly within the string using template placeholders.

Q.5 Difference between `let` and `const`:

The main difference between `let` and `const` in JavaScript is as follows:

1. **Reassignment**: Variables declared with `let` can be reassigned to new values, while variables declared with `const` cannot be reassigned after they are initialized. However, it's important to note that `const` does not make the variable itself immutable; it only prevents reassignment of the variable.

Example with `let`:
```javascript
let x = 5;
x = 10; // Reassignment is allowed
```

Example with `const`:
```javascript
const y = 5;
y = 10; // Throws a TypeError: Assignment to constant variable.
```

2. **Block scope**: Both `let` and `const` are block-scoped, meaning they are only accessible within the block in which they are declared. They have similar scoping behavior.

3. **Hoisting**: Variable declarations with `let` and `const` are hoisted to the top of their block scope, but they are not initialized. This means that accessing a variable before its declaration in the block results in a `ReferenceError`.

Q.6 Template literals in ES6:

Template literals, introduced in ECMAScript 6 (ES6), provide an enhanced way to work with strings in JavaScript. They allow embedding expressions and variables directly within the string using template placeholders, denoted by backticks

(\`). Template literals offer the following features:

1. **String interpolation**: Template literals allow you to embed expressions within the string by using the `${expression}` syntax. The expression inside `${}` is evaluated and the result is inserted into the string.

   Example:
   ```javascript
   const name = 'John';
   console.log(`Hello, ${name}!`); // Output: Hello, John!
   ```

2. **Multiline strings**: Template literals can span multiple lines without the need for escape characters or concatenation. They preserve newlines and whitespace.

   Example:
   ```javascript
   const message = `
   ```

   This is a
   multiline string.
   `;
   console.log(message);
   // Output:
   //
   // This is a
   // multiline string.

3. **Expression evaluation**: Template literals allow executing JavaScript expressions within `${}`. This can be useful for performing calculations or calling functions.

   Example:
   ```javascript
   const x = 5;
   const y = 10;
   console.log(`The sum of ${x} and ${y} is ${x + y}.`); // Output: The sum of 5 and 10 is 15.
   ```

Template literals provide a more concise and readable way to work with strings in JavaScript, especially when dynamic values or multiline strings are involved.

Q.7 Difference between `map` and `forEach`:

`map` and `forEach` are both methods available for iterating over arrays in JavaScript, but they have some key differences:

1. **Return value**: The `map` method returns a new array with 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 return value of the provided function. The `forEach` method, on the other hand, does not return anything. It simply executes a provided function once for each element in the array.

2. **Mutability**: The `map` method does not modify the original array. Instead, it creates a new array with the transformed values. The original array remains unchanged. Conversely, the `forEach` method does not create a new array but can modify the elements of the original array within the provided function if desired.

3. **Usage**: The `map` method is typically used when you want to transform each element of an array and collect the results in a new array. It is useful when you need to perform a computation or manipulation on each element. The `forEach` method is generally used for executing a function on each element of an array without any need for collecting or transforming the values.

Example using `map`:
```javascript
const numbers = [1, 2, 3];
const doubled = numbers.map((num) => num * 2);
console.log(doubled); // Output: [2, 4, 6]
```

Q.8 Destructuring objects and arrays in ES6:

Destructuring is a feature introduced in ECMAScript 6 (ES6) that allows you to extract values from objects and arrays into distinct variables using a concise syntax. It provides an easy way to unpack values and assign them to variables.

1. **Destructuring Objects**:
   When destructuring objects, you can use the object's structure to extract specific values into variables with the same names as the object's properties.

   Example:
   ```javascript
   const person = { name: 'John', age: 30 };
   const { name, age } = person;
   console.log(name); // Output: John
   console.log(age); // Output: 30
   ```

2. **Destructuring Arrays**:
   When destructuring arrays, you can extract individual elements into separate variables based on their position in the array.

   Example:
   ```javascript
   const numbers = [1, 2, 3];
   const [first, second, third] = numbers;
   console.log(first); // Output: 1
   console.log(second); // Output: 2
   console.log(third); // Output: 3
   ```

Destructuring objects and arrays provides a more concise and readable way to extract values and assign them to variables, reducing the need for explicit property or index access.

Q.9 Default parameter values in ES6 functions:

In ECMAScript 6 (ES6), you can define default parameter values for function parameters. Default values allow you to specify a fallback value that will be used if the corresponding argument is not provided when the function is called.

Example:
```javascript
function greet(name = 'Anonymous') {
  console.log(`Hello, ${name}!`);
}
```
Default parameter values are especially useful when you want to handle cases where an argument is missing or undefined. They allow you to provide sensible defaults without requiring explicit checks or conditional logic within the function.

Q.10 Purpose of the spread operator (`...`) in ES6:

The spread operator (`...`) is a powerful feature introduced in ECMAScript 6 (ES6) that allows the expansion of elements from an iterable (e.g., an array or string) into individual elements. It is used in various contexts and provides several useful functionalities:

1. **Array/Object Spread**: The spread operator can be used to create a shallow copy of an array or merge multiple arrays into a new array.

  Example - Array Spread:
  ```javascript
  const arr1 = [1, 2, 3];
  const arr2 = [...arr1, 4, 5, 6];
  console.log(arr2); // Output: [1, 2, 3, 4, 5, 6]
  ```

  Example

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

2. **Function Arguments**: The spread operator can be used to pass an array of arguments to a function as separate arguments.

  Example:
  ```javascript
  function sum(a, b, c) {
  ```

```
     return a + b + c;
   }

   const numbers = [1, 2, 3];
   const result = sum(...numbers);
   console.log(result); // Output: 6
   ```

3. **Destructuring**: The spread operator can be used in array destructuring to assign the remaining elements to a new array.

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

4. **String Conversion**: The spread operator can be used to convert a string into an array of characters.

   Example:
   ```javascript
   const str = 'Hello';
   const chars = [...str];
   console.log(chars); // Output: ['H', 'e', 'l', 'l', 'o']
   ```

The spread operator provides a concise and versatile way to work with arrays, objects, and other iterables in JavaScript, allowing you to easily manipulate and combine values in various contexts.