INHERITANCE IN JS

Prototype-based Inheritance:

In prototype-based inheritance, each object has an internal link to another object called its prototype. When you try to access a property or method of an object, JavaScript looks for it first in the object itself, and if not found, it looks into its prototype, and so on up the prototype chain.

Here's an example of prototype-based inheritance:

```
// Parent constructor function
function Animal(name) {
  this.name = name;
}

// Adding a method to the prototype of Animal
Animal.prototype.sayName = function() {
  console.log('My name is ' + this.name);
};

// Child constructor function inheriting from Animal
function Dog(name, breed) {
  Animal.call(this, name); // Call parent constructor
  this.breed = breed;
}

// Setting up prototype chain
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
```

```
// Adding a method to the Dog prototype
Dog.prototype.bark = function() {
  console.log('Woof! I am a ' + this.breed);
};

// Create instances
const animal = new Animal('Tommy');
const dog = new Dog('Buddy', 'Labrador');

animal.sayName(); // Output: My name is Tommy
dog.sayName();    // Output: My name is Buddy
dog.bark();       // Output: Woof! I am a Labrador
```

ES6 Classes:

With the introduction of ES6 (ECMAScript 2015), JavaScript supports class-based syntax, which provides a more familiar and structured way to work with inheritance.

Here's how the above example can be rewritten using ES6 classes:

```
// Parent class
class Animal {
  constructor(name) {
    this.name = name;
  }
  sayName() {
    console.log('My name is ' + this.name);
  }
}
// Child class inheriting from Animal
```

```
class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Call parent constructor
    this.breed = breed;
  }
  bark() {
    console.log('Woof! I am a ' + this.breed);
  }
}
// Create instances
const animal = new Animal('Tommy');
const dog = new Dog('Buddy', 'Labrador');
animal.sayName(); // Output: My name is Tommy
dog.sayName();    // Output: My name is Buddy
dog.bark();       // Output: Woof! I am a Labrador
```

Advantages:

1. **Code Reusability:** Inheritance allows classes to inherit fields and methods from other classes. This promotes code reuse since common functionalities can be defined in a parent class and inherited by multiple child classes.
2. **Polymorphism:** Inheritance enables polymorphic behavior, where objects of child classes can be treated as objects of their parent class. This allows for more flexible and generalized coding, as methods can be defined to accept objects of the parent class but can work with objects of child classes as well.

Disadvantages:

1. **Tight Coupling:** Inheritance can lead to tight coupling between classes, where changes in the parent class may affect the behavior of child classes. This can make the codebase more fragile and difficult to maintain, especially when modifications to the parent class are required.
2. **Inheritance Hierarchy Complexity:** As the inheritance hierarchy grows deeper, it can become complex and difficult to understand. This can lead to maintenance issues and can make it challenging to extend or modify the hierarchy without unintended consequences.

> Additionally, excessive inheritance can hinder code readability and increase the learning curve for developers working with the codebase.

HOISTING IN JS

Hoisting is a JavaScript behavior where variable and function declarations are moved to the top of their containing scope during the compile phase, before the code execution. This means you can use variables and functions before they are declared in your code. This behavior can sometimes lead to unexpected results if not understood properly.

Hoisting of Variable Declarations:

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

var x = 5;

console.log(x); // Output: 5

In the above code, `x` is declared using `var` after the first `console.log`. However, it doesn't throw an error. Instead, it outputs `undefined`. This is because during the compilation phase, JavaScript hoists the variable declaration `var x;` to the top of its scope, making it accessible throughout the scope.

Hoisting of Function Declarations:

```
sayHello(); // Output: Hello!

function sayHello() {
  console.log('Hello!');
}
```

In this example, the **`sayHello`** function is called before its declaration. Despite the call appearing before the function definition, it still executes without an error. This is because the function declaration **`function sayHello() {...}`** is hoisted to the top of the scope.

Hoisting in ES6 (let and const):

Variables declared with **`let`** and **`const`** are hoisted differently compared to **`var`**. While the declaration is hoisted to the top of the block scope, they're not initialized until the actual line of code is executed.

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

```
let x = 5;
```

Hoisting of Function Expressions:

Function expressions are not hoisted in the same way as function declarations. Only the variable declaration is hoisted, not the function initialization.

```
sayHello(); // Output: TypeError: sayHello is not a function

var sayHello = function() {

  console.log('Hello!');

};
```

Hoisting of Function Declarations vs. Function Expressions: combine b4 and 1st fn

Hoisting in Nested Scopes:

Hoisting also occurs within nested scopes. Variables declared within a block or function are hoisted to the top of their containing scope, but they are not accessible outside of that scope. For example:

```
function foo() {

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

  var x = 10;

}

foo();

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

In this example, **x** is hoisted to the top of the **foo** function scope, so it's accessible anywhere within the function, but it's not accessible outside of the function.

Advantages of Hoisting:

1. **Flexibility:** Hoisting allows you to write code in a more flexible way. You can declare variables or functions anywhere in your code, and JavaScript will still understand them.
2. **Function Availability:** Hoisting ensures that functions can be called before they are declared in the code, which can be handy for organizing your code logically.

Disadvantages of Hoisting:

1. **Debugging Challenges:** Hoisting might make debugging a bit trickier, as variables and functions can appear to be in different places in the code than where they were actually declared.
2. **Readability Concerns:** Sometimes hoisting can make code harder to understand, especially for beginners. It might not be clear at first glance where a variable or function is actually declared.

ARRAYS IN JS

Modifying Elements:

fruits[1] = 'grape'; // Change 'banana' to 'grape'

console.log(fruits); // Output: ['apple', 'grape', 'orange']

Array Methods:

JavaScript provides many built-in methods to manipulate arrays, such as `concat()`, `slice()`, `join()`, `indexOf()`, `includes()`, `sort()`, `reverse()`, etc

let reversedArray = fruits.reverse(); // Reverse the array

console.log(reversedArray); // Output: ['orange', 'grape', 'apple']

Multidimensional Arrays:

let matrix = [[1, 2], [3, 4], [5, 6]];

console.log(matrix[1][0]); // Output: 3

Spread Operator:

const fruits = ['apple', 'banana', 'orange'];

const moreFruits = ['kiwi', 'pineapple'];

const allFruits = [...fruits, ...moreFruits];

console.log(allFruits); // Output: ['apple', 'banana', 'orange', 'kiwi', 'pineapple']

Sparse Arrays:

JavaScript arrays can be sparse, meaning they can have holes (missing indices) in them.

```
let sparseArray = [1, , , 4];

console.log(sparseArray.length); // Output: 4
```

Accessing and Modifying DOM IN JS

Accessing DOM Elements:

ACCESSING

1.getElementById(): You can get an element by its unique ID using `document.getElementById('id')`.

Eg: const myElement = document.getElementById('myId');

2.getElementsByClassName(): You can get elements by their class names using `document.getElementsByClassName('className')`. It returns a collection of elements.

Eg: const myElements = document.getElementsByClassName('myClass');

3. getElementsByTagName(): You can get elements by their tag names using `document.getElementsByTagName('tagName')`. It also returns a collection.

4 querySelector(): You can use CSS selectors to get the first matching element using `document.querySelector('selector')`

```
Eg: const myElement = document.querySelector('#myId .myClass');
```

MODIFYING

```
<script>

  // Accessing elements by ID

  const myParagraph = document.getElementById('myParagraph');

  const myButton = document.getElementById('myButton');


  // Modifying text content

  myParagraph.textContent = 'This text is changed!';
```

```
  // Adding event listener to the button
  myButton.addEventListener('click', function() {
    myParagraph.textContent = 'Text changed by clicking the button!';
  });

  // Creating a new paragraph element
  const newParagraph = document.createElement('p');
  newParagraph.textContent = 'This is a new paragraph added dynamically!';
  document.body.appendChild(newParagraph);
</script>
```