DSA in JavaScript

OOPS: -

Classes and Objects: -

Classes are blueprints for creating objects.

Use the keyword class to create a class.

Always add a method named constructor ().

Objects are instances of classes.

```
class Car {
  constructor (brand,model) {
    this.brand = brand;
    this.model= model;
  }
displayInfo() {
 console.log('Car :'+ this.brand +" "+ this.model);
 }
}
const myCar = new Car("Toyota","Corolla");
myCar.displayInfo(); //Car : Toyota Corolla
```

The example above creates a class named "Car".

The class has two initial properties: "brand" and "model".

The example above uses the Car class to create a car object called myCar.

Constructor: -

The constructor method is called automatically when a new object is created.

It has to have the exact name "constructor"

It is executed automatically when a new object is created

It is used to initialize object properties

If you do not define a constructor method, JavaScript will add an empty constructor method.

JavaScript is a prototype-based language, which means it doesn't have traditional classes (until ES6). Instead, it uses objects and prototypes to achieve inheritance and encapsulation.

```
// Object literal
const car = {
   brand: 'Maruti Suzuki',
   model: 'Alto',
   start: function() {
      console.log(`${this.brand} ${this.model} is starting.`);
   }
};
car.start(); // Output: Maruti Suzuki Alto is starting.
```

Prototypes: -

Every object in JavaScript has a hidden property that links it to another object.

```
// Constructor function
function Car(brand, model) {
   this.brand = brand;
   this.model = model;
}
// Adding a method to the prototype
Car.prototype.start = function() {
   console.log(`${this.brand} ${this.model} is starting...`);
};
const myCar = new Car('Toyota', 'Innova');
myCar.start(); // Output: Toyota Innova is starting...
// Object.create()
const coder = {
        isStudying: false,
        printIntroduction: function () {
                console.log(`My name is ${this.name}. Am I
```

```
                studying?: ${this.isStudying}.`)
        }
}
// Object.create() method
const me = Object.create(coder);
// "name" is a property set on "me", but not on "coder"
me.name = 'Mukul';
// Inherited properties can be overwritten
me.isStudying = true;
me.printIntroduction();
```

Prototype Chain

When you access a property or method on an object, JavaScript looks for it in the object itself. If it doesn't find it, it looks up the prototype chain.

```
console.log(myCar.toString()); // Output: [object Object]
```

Here, toString is not defined in myCar or Car.prototype, but it's available in the Object.prototype

Encapsulation: -

Tt refers to the bundling of data (variables) and methods (functions) that operate on that data into a single unit.

```
class Person{
   #ssn;
   constructor (name,ssn){
   this.name = name;
   this.#ssn = ssn;
   }
}
const person = new Person("john","NU123");
console.log(person.name); //john
console.log(person.#ssn); //error
```

Inheritance: -

It allows a child class to inherit properties and methods from parent class.

```
class Vehicle {
    constructor(type) {
        this.type = type;
    }

    drive() {
        console.log(`Driving a ${this.type}`);
    }
}
class ElectricCar extends Vehicle {
    constructor(brand, model) {
        super('Electric Car'); // Call the parent constructor
        this.brand = brand;
        this.model = model;
    }

    charge() {
        console.log(`${this.brand} ${this.model} is charging...`);
    }
}
const myElectricCar = new ElectricCar('MG', 'EcoDrive');
myElectricCar.drive(); // Output: Driving a Electric Car
myElectricCar.charge(); // Output: MG EcoDrive is charging...
```

The super () method refers to the parent class.

By calling the super () method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

The name of the getter/setter method cannot be the same as the name of the property.

Polymorphism: -

It allows a subclass to modify or override methods from a parent class.

```
class Shape{
   draw(){
      console.log("Drawing a shape: ");
   }
}
class Circle extends Shape{
   draw(){
      console.log("drawing a circle");
   }
}
const shape1 = new Shape();
shape1.draw(); // drawing a shape
const shape2 = new Circle();
shape2.draw(); // drawing a circle.
```

Method overriding and method overloading are the ways through which polymorphism can be achieved.

For method overloading, there is no direct way. We need to check arguments length.

Abstraction: -

It hides complex logic. It does not have built in support for abstract classes but can be done using base classes.

```
class Vehicle{
   constructor() {
      if(this.constructor == Vehicle) {
         throw new Error("Abstract class cannot be instantiated");
      }
   }
   move (){
      throw new Error("Abstract method must be implemented");
```

```
    }
}
class Car extends Vehicle{
    move(){
        console.log("car is moving");
    }
}
const car = new Car();
car.move();
//const vehicle = new Vehicle(); // error
```

Composition over Inheritance: -

Composition is a design principle where you build complex objects by combining simpler objects, rather than inheriting from a base class. This makes your code more flexible and reusable.

```
// Smaller, reusable components
class Engine {
    start() {
        console.log("Engine started...");
    }
}
class Battery {
    charge() {
        console.log("Battery charging...");
    }
}
// Composing a car using smaller components
class ElectricCar {
    constructor() {
        this.engine = new Engine();
        this.battery = new Battery();
```

```
    }
    start() {
        this.engine.start();
        this.battery.charge();
        console.log("Electric car is ready to drive!");
    }
}
const myElectricCar = new ElectricCar();
myElectricCar.start();
// Output:
// Engine started...
// Battery charging...
// Electric car is ready to drive!
```

Design Patterns: -

Design patterns are reusable solutions to common problems in software design. Let's look at two popular patterns: Factory Pattern and Singleton Pattern.

Factory Pattern

The Factory Pattern is used to create objects without specifying the exact class of the object.

```
class Car {
    constructor(brand, model) {
        this.brand = brand;
        this.model = model;
    }
    drive() {
        console.log(`${this.brand} ${this.model} is driving...`);
    }
}
class ElectricCar extends Car {
    charge() {
```

```
        console.log(`${this.brand} ${this.model} is charging...`);
    }
}
class CarFactory {
    static createCar(type, brand, model) {
        switch (type) {
            case 'electric':
                return new ElectricCar(brand, model);
            default:
                return new Car(brand, model);
        }
    }
}
const myCar = CarFactory.createCar('electric', 'MG', 'EcoDrive');
myCar.drive(); // Output: MG EcoDrive is driving...
myCar.charge(); // Output: MG EcoDrive is charging...
```

Singleton Pattern: -

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it.

```
class ConnectedCarSystem {
    constructor() {
        if (ConnectedCarSystem.instance) {
            return ConnectedCarSystem.instance;
        }
        this.cars = [];
        ConnectedCarSystem.instance = this;
    }
    addCar(car) {
        this.cars.push(car);
```

```
    }
    listCars() {
        console.log("Connected Cars:", this.cars);
    }
}
const system1 = new ConnectedCarSystem();
const system2 = new ConnectedCarSystem();
system1.addCar({ brand: 'MG', model: 'ConnectedCarV1' });
system2.listCars(); // Output: Connected Cars: [ { brand: 'MG', model: 'ConnectedCarV1' } ]
console.log(system1 === system2); // Output: true (both are the same instance)
```

Static Methods: -

Static class methods are defined on the class itself. It is called through class and not by class object.

If you want to use the object inside the static method, you can send it as a parameter.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello() {
    return "Hello!!";
  }
}
const myCar = new Car("Ford");
Car.hello(); //correct
myCar.hello(); //wrong
```

Note: -

Unlike functions, and other JavaScript declarations, class declarations are not hoisted. That means that you must declare a class before you can use it.

JavaScript Strings: -

Strings are for storing text.

It is immutable, meaning their values cannot be changed after creation.

let str1 = 'hello’; //using single quote

let str2 = "world"; //using double quotes

let str3 = `hello ${str2}`;//template literal (back ticks)

You can use quotes inside a string, if they don't match the quotes surrounding the string.

Templates allow single and double quotes inside a string.

String length: -

let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

let length = text.length; //26

backslash escape character: -

it helps in inserting double quotes or apostrophe in string.

let text = "We are the so-called \"Vikings\" from the north.";

\\ - inserts backlash character.

\b - Backspace (removes previous character)

\n - New Line

\t - Horizontal Tabulator

Strings can also be defined as objects with the keyword new.

let y = new String("John");

Do not create String objects. The new keyword complicates the code and slows down execution speed.

String objects can produce unexpected results like it shows equal for == and not equal to for ===

Accessing Characters: -

let str = "Hello";

console.log(str.charAt(1)); // Output: e returns character

console.log(str.charCodeAt(1)); // Output: 101 (Unicode for 'e')

console.log(str.at(1)) // output : e (modern)

console.log(str[1]); // like in array

The at() method is a new addition to JavaScript.

It allows the use of negative indexes while charAt() do not.

Now you can use myString.at(-2) instead of charAt(myString.length-2).

Extracting String Parts: -

slice(start, end): Extracts a part of a string.

substring(start, end): Similar to slice, but does not accept negative indices.

substr(start, length): Extracts a specific number of characters.

```
let text = "JavaScript";
console.log (text.slice(0, 4));  // Output: Java
console.log (text.substring(4, 10));  // Output: Script
console.log (text.substr(4, 6));  // Output: Script
```

If you omit the second parameter, the method will slice out the rest of the string.

If a parameter is negative, the position is counted from the end of the string.

Changing Case: -

```
console.log("hello".toUpperCase()); // Output: HELLO
console.log("WORLD".toLowerCase()); // Output: world
```

String Concatenation: -

a) Using + Operator

```
let firstName = "John";
let lastName = "Doe";
console.log(firstName + " " + lastName); // Output: John Doe
```

b) Using concat() Method

```
console.log(firstName.concat(" ", lastName)); // Output: John Doe
```

c) Using Template Literals

```
console.log(`${firstName} ${lastName}`); // Output: John Doe
```

Removing Whitespaces: -

trim(): Removes whitespace from both sides.

trimStart(), trimEnd() meaning is self – explanatory

```
let spaced = "   Hello World!   ";
console.log(spaced.trim()); // Output: "Hello World!"
console.log(spaced.trimStart()); // Output: "Hello World!   "
```

console.log(spaced.trimEnd()); // Output: "   Hello World!"

JavaScript String split(): -

A string can be converted to an array with the split () method:

text.split(",") // Split on commas

text.split(" ") // Split on spaces

If the separator is omitted, the returned array will contain the whole string in index [0].

If the separator is "", the returned array will be an array of single characters.

Replacing String Content: -

replace(oldValue, newValue): Replaces a substring (only first occurrence).

replaceAll(oldValue, newValue): Replaces all occurrences.

let msg = "I love JavaScript!";

console.log(msg.replace("JavaScript", "Python")); // Output: I love Python!

JavaScript String repeat(): -

The repeat() method returns a string with a number of copies of a string.

let text = "Hello world!";

let result = text.repeat(2); // Hello world!Hello world!

JavaScript String Padding: -

padStart() and padEnd() to support padding at the beginning and at the end of a string.

Pad a string with "0" until it reaches the length 4:

let text = "5";

let padded = text.padStart(4,"0"); // 0005

similarly, padEnd() will pad at end.

String Search Methods: -

indexOf(substring): Returns the position of the first match.

lastIndexOf(substring): Returns the last occurrence position.

Both methods can accept a second parameter as the starting position for the search.

includes(substring): Returns true if found, else false.

startsWith(substring), endsWith(substring)

the above two can take another parameter.

```
let sentence = "JavaScript is fun";

console.log(sentence.indexOf("Script")); // Output: 4

console.log(sentence.includes("fun")); // Output: true

console.log(sentence.startsWith("Java")); // Output: true

console.log(sentence.endsWith("fun")); // Output: true
```

The search() method searches a string for a string or regular expressions and returns the position of the match.

```
Sentence.search("is");
```

The search() method cannot take a second start position argument. The indexOf() method cannot take powerful search values (regular expressions).

Bitwise problems: -

1. Check if a Number is Even or Odd

```
function isEvenOrOdd(num) {
  if ( num & 1 === 1){
     return "Odd";
  }
  else{
     return "Even";
  }
}
console.log(isEvenOrOdd(4)); // Even
console.log(isEvenOrOdd(7)); // Odd
```

7 & 1

0111 & 0001 = 1 odd

6 & 1

0110 & 0001 = 0 even

2. Swap Two Numbers Without a Temporary Variable

```
function swapNumbers(a, b) {
  a = a ^ b;
```

```
    b = a ^ b;
    a = a ^ b;
    return [a, b];
  }
  console.log(swapNumbers(5, 10)); // [10, 5]
```

a= 5, b=7

a= 5 ^ 7 = 0101 ^ 0111 = 0010 - 2

b= 2 ^ 7 = 0010 ^ 0111 = 0101 - 5

a= 2 ^ 5 = 0010 ^ 0101 = 0111 - 7

now a =7,b = 5

3. Count the Number of Set Bits (1s) in a Number

```
  function countSetBits(num) {
    let count = 0;
    while (num > 0) {
      count += num & 1;
      num = num >>> 1;
    }
    return count;
  }
  console.log(countSetBits(29)); // 29 is 11101 in binary, so output is 4
```

num = 6

count = 0 + (6 &1) = 0

6 >>> 1 = 3

num = 3

count = 0 + (3 & 1) = 1

3 >>> 1 = 1

num = 1

count = 1 + (1 & 1) = 2

1 >>> 1 = 0

count = 2 + (0 & 1) = 2

0 >>> 1 = 0

num = 0

then count = 2

6 is 110. it has two 1's

4. Check if a Number is a Power of Two

```
function isPowerOfTwo(num) {
  return num > 0 && (num & (num - 1)) === 0;
}
console.log(isPowerOfTwo(16)); // true
console.log(isPowerOfTwo(15)); // false
```

num = 16

16 > 0 && (16 & (16-1)) === 0

true && 10000 & 1111 = true && 0(false) = false(0) === 0

so return true

num = 12

11 > 0 && (11 & (11-1)) === 0

true && 1011 & 1010 = true && 1010 (10)(true) = true(1) === 0

so return false

5. Find the Missing Number in a Range

```
function findMissingNumber(arr, n) {
  let xor = 0;
  for (let i = 0; i < arr.length; i++) {
    xor ^= arr[i];
  }
  for (let i = 0; i <= n; i++) {
```

```
    xor ^= i;
  }
  return xor;
}
console.log(findMissingNumber([0, 1, 3], 3)); // 2
```

6. Toggle the nth Bit of a Number

```
function toggleBit(num, n) {
  return num ^ (1 << n);
}
console.log(toggleBit(5, 1)); // 5 is 101, toggling 2nd bit gives 111 (7)
```

7. Check if the nth Bit is Set

```
function isBitSet(num, n) {
  return (num & (1 << n)) !== 0;
}
console.log(isBitSet(5, 2)); // 5 is 101, 2nd bit is 1 (true)
```

8. Set the nth Bit of a Number

```
function setBit(num, n) {
  return num | (1 << n);
}
console.log(setBit(5, 1)); // 5 is 101, setting 2nd bit gives 111 (7)
```

9. Clear the nth Bit of a Number

```
function clearBit(num, n) {
  return num & ~(1 << n);
}
console.log(clearBit(7, 1)); // 7 is 111, clearing 2nd bit gives 101 (5)
```

10. Reverse Bits of a Number

```
function reverseBits(num) {
  let reversed = 0;
  let bits = 32; // Assuming 32-bit integers
  while (bits--) {
    reversed = (reversed << 1) | (num & 1);
    num >>>= 1;
  }
  return reversed;
}
console.log(reverseBits(43261596)); // 964176192
```

The operator precedence for bitwise operators from high to low is not, shifts, and, xor and or.