Oops in js

OOP in JavaScript provides a **way to structure and organize code** in a more modular and **reusable** manner. By utilizing objects, classes, inheritance, encapsulation, and polymorphism

1.class

2.object

3.encapsulatiion

4.inheritance

5.polymorphism

6.abstraction

Constructor class

Construtor class is a way to define a template for creating object

* class was introduced recently in ES6
* class is a collection of properties and methods(static/intance)
* syntax :
* class Classname{
* }
*
* obj syntax :
* new Classname(arguments)
* constructor should be one

Class Definition

- **Class**: A class is a template for creating objects. It encapsulates data and functions that operate on that data.
- **Properties (state)**: Variables that belong to the class.
- **Methods (behaviour)**: Functions that belong to the class.

Ex:-

Marker Class

The **Marker** class has three properties: **color, lid**, and **type**. It also has a constructor to initialize these properties.

```
class Marker {
  color; // property to hold the color of the marker
  lid; // property to hold the type of lid of the marker
  type; // property to hold the type of the marker

  // Constructor to initialize the properties
  constructor(color,lid, type) {
    this.color = color; // initializes the color property
    this.lid = lid; // initializes the lid property
    this.type = type; // initializes the type property
  }

  // method initialisation - it is a behaviour of object
  write() {
      console.log("Writing with " + this.color + " marker");
  }
}
```

Creating Objects

Objects are instances of a class. The new keyword is used to create an object from a class.

```
var marker1 = new Marker("red", "square", "permanent");
var marker2 = new Marker("black", "circle", "temporary");
```

Logging the object

```
//logging the objects
console.log(marker1);
console.log(marker2);
```

Calling the method

```
//calling the methods
marker1.write();
```

Key Points

- **Class Definition**: Defines the state and behavior of objects.
- **Properties**: Variables within a class that hold data.
- **Methods** : function that holds the behaviour
- **Constructor**: Special method to initialize properties when an object is created.
- **this Keyword**: Refers to the current instance of the class.

- **Creating Objects**: Use new keyword followed by class name and arguments to create objects.

use the class keyword to create a JavaScript class.

```
// create a class
class Person{
   // body of class
};
```

To use the class constructor we must need to assign object to it

```
class Person{
    constructor (name,age){
        this.name=name;
        this.age =age;
    }

    hi(a,b){
        return "my name is "+ this.name+ " age is " + this.age;
    }
 }

 var b=new Person("teja",25);
console.log(b);//{name:teja; age:25}
console.log(b.age);//25

console.log(b.hi());//my name is teja age is 25
```

by using prototypes we can add properties directly

Person.prototype.fullName = “sai teja”

Javascript classes with private fields

Class Definition

- **Class**: Employee
  - **Private Fields**: #name, #age, #salary (indicated by # prefix)
  - Introduced to provide encapsulation and data protection within a class.
  - Enhances security and prevents accidental modification of sensitive data.

```
class Employee {
#name;
#age;
#salary;

constructor(name, age, salary) {
    this.#name = "John";    // Private field initialization
    this.#age = 30;         // Private field initialization
    this.#salary = 50000;   // Private field initialization
}

//private fields can only accessed inside the class
 hello(){
    console.log("Hello, my name is " + this.# )
    }
```

```
}
```

Private Fields

- **Definition**: Private fields are declared using the # symbol before the field name (#name, #age, #salary).
- **Access Control**: Private fields can only be accessed and modified from inside the class where they are defined.

Accessing object will thrown an error because we cannot able to access them in outside

```
var person1=new Employee("john", 25, 50000);
  console.log(person1.name)// error - private property cannot be accessed directly
```

method will print the output because properties can be accessed inside a class

```
person1.hello();
```

Encapsulation

Encapsulation in JavaScript refers to the **bundling of data (attributes) and methods (functions) that operate on the data into a single unit**, typically known as an object. This concept helps in hiding the internal

state of an object and only exposing the necessary functionalities to interact with that state.

Access the values using getter

Syntax:

```
// we can giving access
get methodname(){
  return this.#name;
}
```

```
// we can giving access
get accessname(){
  return this.#name;
}
```

```
class Employee {
#name;
#age;
#salary;

constructor(name, age, salary) {
    this.#name = name;    // Private field initialization
    this.#age = age;         // Private field initialization
    this.#salary = salary;   // Private field initialization
}
hello(){
    console.log("Hello, my name is " + this.#name + " and I am a " )
```

```
  }


  get accessname(){
    return this.#name;
  }

}
```

```
var person1=new Employee("john", 25, 50000);
//we can able to access now because of encapsulation
console.log(person1.accessname)// john - will not thrown an error
```

Modify the values using setter

```
  set setsalary(sal){
    this.#salary=sal;
  }


  get getsalary(){
    return this.#salary;
  }
```

```
//creating a object
 var person1=new Employee("john", 25, 50000);
```

```
//setting the values
person1.setsalary=99500;

//accessing the updated values
console.log(person1.getsalary);
```

//complete program

```
class Employee {
#name;
#age;
#salary;

constructor(name, age, salary) {
    this.#name = name;    // Private field initialization
    this.#age = age;         // Private field initialization
    this.#salary = salary;   // Private field initialization
}
hello(){
    console.log("Hello, my name is " + this.#name + " and I am a " )
}

// we can giving access
get accessname(){
  return this.#name;
}

//giving the access to change the values
set setsalary(sal){
  this.#salary=sal;
}
```

```
  get getsalary(){
    return this.#salary;
  }

}

//creating a object
 var person1=new Employee("john", 25, 50000);

 //setting the values
 person1.setsalary=99500;

 //accessing the updated values
 console.log(person1.getsalary);
```

Certainly! Let's break down the concept of inheritance in JavaScript using the provided code as an example and create detailed notes.

JavaScript Inheritance

What is Inheritance?

- **Inheritance**: A feature in object-oriented programming that allows one class (subclass or derived class) to inherit properties and methods from another class (superclass or base class).
- **Purpose**: Promotes code reusability and establishes a natural hierarchy between classes.

Base Class: Animal

The Animal class serves as the base class with common properties and methods that can be inherited by other classes.

```
class Animal{

  weight;
  height;
  voice;

  constructor(weight, height, voice){
    this.weight = weight;
    this.height = height;
    this.voice = voice;
  }

  eating(){
    console.log("Eating...");
  }

}
```

Subclass: Dog

The Dog class extends the Animal class, inheriting its properties and methods while adding its own specific properties.

```
class Dog extends Animal {
breed;
food;

constructor(weight, height, voice, breed, food) {
    super(weight, height, voice); // Calls the constructor of the Animal class
    this.breed = breed;
    this.food = food;
}
```

```
}
```

Properties:

- Inherited: weight, height, voice
- Specific: breed, food

Constructor:

- Calls super(weight, height, voice) to initialize inherited properties.
- Initializes breed and food properties.

Creating Objects and Using Methods

```
        var dog1 = new Dog("15kgs", "2feet", "bow bow", "indie",
"nonveg");
    console.log(dog1); // Outputs: Dog { weight: '15kgs', height: '2feet',
voice: 'bow bow', breed: 'indie', food: 'nonveg' }

    dog1.eating(); // Outputs: Eating...
```

Notes

1. **Inheritance in JavaScript**:
    - **Base Class**: Defines common properties and methods.
    - **Subclass**: Inherits from the base class and can add or override properties and methods.
2. **super Keyword**:
    - Used in the constructor of the subclass to call the constructor of the base class.

    - Ensures inherited properties are properly initialized before accessing subclass-specific properties.
3. **Constructor Rules**:
    - In a subclass constructor, super must be called before accessing this.
    - Ensures proper initialization of the inherited fields.
4. **Method Inheritance**:
    - Subclasses inherit all methods from the base class.
    - Methods can be overridden in the subclass if needed.
5. **Code Reusability**:
    - Inheritance promotes code reusability by allowing subclasses to use and extend the functionality of base classes.
    - Reduces redundancy and simplifies maintenance.
6. **Example Scenario**:
    - **Base Class**: Animal with common properties (weight, height, voice) and methods (eating).
    - **Subclasses**: Dog and Cat with additional specific properties (breed, food for Dog, and color for Cat).

What is Polymorphism?

- **Polymorphism**: A core concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables methods to perform different tasks based on the object they are called on.
- **Purpose**: Enhances flexibility, reusability, and maintainability of code by allowing a single interface to represent different underlying forms (data types).

Example: Polymorphism in Action

Using the provided code as an example, we can see how polymorphism works in JavaScript

Base Class: Animal

The Animal class serves as the base class with common properties and methods that can be inherited by other classes.

```
class Animal {
  weight;
  height;
  voice;

  constructor(weight, height, voice) {
    this.weight = weight;
    this.height = height;
    this.voice = voice;
  }

  eating() {
    console.log("Eating...");
  }
}
```

Properties: weight, height, voice

Methods:

- eating(): Prints "Eating..." to the console.

Subclass: Dog

The Dog class extends the Animal class, inheriting its properties and methods but also overriding the eating method.

```
class Dog extends Animal {
  breed;
  food;
  constructor(weight, height, voice, breed, food) {
    // we cannot write anything above the super
    super(weight, height, voice);
    this.breed = breed;
    this.food = food;
  }

  eating() {
    console.log("Dog is eating " + this.food);
  }
}
```

Properties:

- Inherited: weight, height, voice
- Specific: breed, food

Methods:

- Overridden: eating(): Prints "Dog is eating " followed by the type of food the dog eats.

Object Creation and Method Overriding

```
var dog1 = new Dog("15kgs", "2feet", "bow bow", "indie", "nonveg");
console.log(dog1);
```

```
    // Outputs: Dog { weight: '15kgs', height: '2feet', voice: 'bow bow', breed: 'indie', food: 'nonveg' }

    dog1.eating();
    // Outputs: Dog is eating nonveg
```

Object Creation: dog1 is created as an instance of the Dog class.

Method Overriding: The eating method in the Dog class overrides the eating method in the Animal class.

Polymorphism

Polymorphism, in the context of object-oriented programming, refers to the ability of **different objects to respond to the same message or method invocation** in different ways. It allows objects of different classes to be treated as objects of a common superclass, providing a unified interface for different classes.

Abstraction

Abstraction in JavaScript refers to the concept of hiding complex implementation details and showing only the necessary features of an object or function. It allows developers to work with high-level representations without needing to understand all the underlying complexities.