Object-Oriented Programming (OOP) in JavaScript

What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, promoting reusability, modularity, and scalability. JavaScript supports OOP principles using **prototypes** and **ES6 classes**.

Core OOP Principles

1. **Encapsulation**: Bundling data and methods that operate on the data within objects.
2. **Abstraction**: Hiding implementation details and exposing only the necessary functionalities.
3. **Inheritance**: Allowing one class (child) to derive properties and methods from another class (parent).
4. **Polymorphism**: Allowing different classes to be treated as instances of the same class through method overriding.

Creating Objects in JavaScript

1. Object Literals

```
const person = {
    name: "John",
    age: 30,
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};
person.greet();
```

2. Constructor Functions

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        console.log(`Hello, my name is ${this.name}`);
    };
}
const person1 = new Person("Alice", 25);
person1.greet();
```

3. Prototypes

```
function Animal(name) {
    this.name = name;
}
Animal.prototype.speak = function() {
    console.log(`${this.name} makes a noise.`);
};
const dog = new Animal("Dog");
dog.speak();
```

4. ES6 Classes

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log(`Hello, my name is ${this.name}`);
    }
}
const person2 = new Person("Bob", 35);
person2.greet();
```

OOP Features in JavaScript

1. Encapsulation

```
class Car {
    constructor(brand) {
        let _brand = brand; // Private variable
        this.getBrand = function() {
            return _brand;
        };
    }
}
const car = new Car("Toyota");
console.log(car.getBrand());
```

2. Abstraction

Abstraction allows us to hide implementation details and expose only necessary methods.

```
class Car {
    startCar() {
        this.#igniteEngine(); // ✅ Allowed inside the class
        console.log("Car is ready to drive!");
    }

    #igniteEngine() {  // Private method (ES6+)
        console.log("Engine started...");
    }
}

const myCar = new Car();
myCar.startCar(); // ✅ Works fine

myCar.#igniteEngine(); // ❌ Error: Private field '#igniteEngine' must be declared in a
```

The igniteEngine() method is hidden from outside users.

The user just calls startCar() without needing to know how the engine works

3. Inheritance

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    speak() {
        console.log(`${this.name} makes a sound.`);
    }
}
class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}
const myDog = new Dog("Buddy");
myDog.speak();
```

4. Polymorphism

```
class Employee {
    constructor(name) {
        this.name = name;
    }
    getDetails() {
        return `Employee: ${this.name}`;
    }
}

class Manager extends Employee {
    constructor(name, department) {
        super(name);  // ✅ Calls Employee's constructor first
        this.department = department;
    }
    getDetails() {
        return `Manager: ${this.name}, Department: ${this.department}`;
    }
}

const emp = new Employee("John");
console.log(emp.getDetails()); // Output: Employee: John

const mgr = new Manager("Alice", "HR");
console.log(mgr.getDetails()); // Output: Manager: Alice, Department: HR
```

Where is Polymorphism Happening?

Method Overriding:

- `getDetails()` is defined in the **parent class** **(`Employee` **********).
- `Manager` **overrides** `getDetails()` with a new version specific to `Manager` .

Same Method, Different Behavior:

- When calling `getDetails()` on an **Employee**, we get:
  "Employee: John"

- When calling `getDetails()` on a **Manager**, we get:
  "Manager: Alice, Department: HR"
- **Even though we call the same method, the output differs based on the object type → Polymorphism!**

Conclusion

JavaScript's OOP features allow developers to create structured and reusable code. With ES6 classes and prototypes, OOP in JavaScript is powerful, enabling better code organization and maintainability.