Certainly! Let's take the example of an **Employee Management System** that includes **Employee**, **Department**, and **Manager** as key components. This example will showcase Object-Oriented Programming (OOP) concepts in JavaScript, including objects, classes, inheritance, encapsulation, polymorphism, and abstraction.

## ****JavaScript OOP Example: Employee Management System****

### ****1. Objects in JavaScript****

An object is a collection of key-value pairs. Let’s start by creating a basic Employee object.

let employee = {

name: "John Doe",

position: "Software Developer",

department: "Engineering",

displayInfo: function() {

console.log(`Name: ${this.name}`);

console.log(`Position: ${this.position}`);

console.log(`Department: ${this.department}`);

}

};

employee.displayInfo();

// Output:

// Name: John Doe

// Position: Software Developer

// Department: Engineering

### ****2. Classes and Constructors****

Classes are blueprints for creating objects. Here we’ll define an Employee class and create employee objects using a constructor.

class Employee {

constructor(name, position, department) {

this.name = name;

this.position = position;

this.department = department;

}

displayInfo() {

console.log(`Name: ${this.name}`);

console.log(`Position: ${this.position}`);

console.log(`Department: ${this.department}`);

}

}

// Create a new employee object using the class

const employee1 = new Employee("Jane Smith", "HR Manager", "Human Resources");

employee1.displayInfo();

// Output:

// Name: Jane Smith

// Position: HR Manager

// Department: Human Resources

### ****3. Inheritance in JavaScript****

Inheritance allows one class to inherit the properties and methods of another. Here, we'll create a Manager class that extends the Employee class.

class Manager extends Employee {

constructor(name, position, department, teamSize) {

super(name, position, department); // Call the parent class constructor

this.teamSize = teamSize;

}

displayInfo() {

super.displayInfo(); // Call the parent class method

console.log(`Team Size: ${this.teamSize}`);

}

manageTeam() {

console.log(`${this.name} is managing a team of ${this.teamSize} people.`);

}

}

// Create a Manager object

const manager1 = new Manager("Sarah Lee", "Engineering Manager", "Engineering", 8);

manager1.displayInfo();

// Output:

// Name: Sarah Lee

// Position: Engineering Manager

// Department: Engineering

// Team Size: 8

manager1.manageTeam();

// Output:

// Sarah Lee is managing a team of 8 people.

### ****4. Encapsulation****

Encapsulation involves bundling data and methods that work on that data within a class, and restricting access to some of the object's properties (e.g., making them private). We can simulate private properties using a # syntax in ES2022.

class Employee {

#salary; // Private field

constructor(name, position, department, salary) {

this.name = name;

this.position = position;

this.department = department;

this.#salary = salary; // Private property

}

getSalary() {

return this.#salary;

}

setSalary(newSalary) {

if (newSalary < 0) {

console.log("Salary cannot be negative.");

} else {

this.#salary = newSalary;

}

}

displayInfo() {

console.log(`Name: ${this.name}`);

console.log(`Position: ${this.position}`);

console.log(`Department: ${this.department}`);

console.log(`Salary: $${this.#salary}`);

}

}

// Create an Employee object with private salary

const employee2 = new Employee("Alice Turner", "Developer", "Engineering", 70000);

employee2.displayInfo();

// Output:

// Name: Alice Turner

// Position: Developer

// Department: Engineering

// Salary: $70000

employee2.setSalary(80000);

console.log(employee2.getSalary()); // 80000

### ****5. Polymorphism****

Polymorphism allows different classes to implement the same method in different ways. For example, both Manager and Employee can have a displayInfo method, but the Manager class can add extra information such as the team size.

class Employee {

constructor(name, position, department) {

this.name = name;

this.position = position;

this.department = department;

}

displayInfo() {

console.log(`Name: ${this.name}`);

console.log(`Position: ${this.position}`);

console.log(`Department: ${this.department}`);

}

}

class Manager extends Employee {

constructor(name, position, department, teamSize) {

super(name, position, department);

this.teamSize = teamSize;

}

displayInfo() {

super.displayInfo(); // Call the parent class method

console.log(`Team Size: ${this.teamSize}`);

}

}

const employee3 = new Employee("Bob Brown", "Sales Executive", "Sales");

const manager2 = new Manager("Lisa Clark", "Sales Manager", "Sales", 10);

const employees = [employee3, manager2];

employees.forEach(employee => {

employee.displayInfo();

console.log("------");

});

// Output:

// Name: Bob Brown

// Position: Sales Executive

// Department: Sales

// ------

// Name: Lisa Clark

// Position: Sales Manager

// Department: Sales

// Team Size: 10

In the above code, even though both Employee and Manager have the same displayInfo method, they behave differently, demonstrating polymorphism.

### ****6. Abstraction****

Abstraction hides the complex implementation details and exposes only the necessary information to the user. We can use abstraction to create methods that perform multiple internal tasks, but users don’t need to know the details.

class Department {

constructor(name) {

this.name = name;

}

hireEmployee(name, position) {

console.log(`Hiring ${name} for the ${this.name} department as a ${position}.`);

// Other complex operations can be hidden here, like creating employee records.

}

}

class Employee {

constructor(name, position) {

this.name = name;

this.position = position;

}

joinDepartment(department) {

department.hireEmployee(this.name, this.position);

}

}

const engineering = new Department("Engineering");

const employee4 = new Employee("Mark Smith", "Software Engineer");

employee4.joinDepartment(engineering);

// Output:

// Hiring Mark Smith for the Engineering department as a Software Engineer.

In the joinDepartment method, the abstraction allows the employee to join a department without needing to know the underlying complexities of how the hiring process works.

### ****7. Getter and Setter Methods****

Getters and setters provide controlled access to the properties of an object. Let's use getter and setter methods to manage the salary of an Employee.

class Employee {

constructor(name, position, department, salary) {

this.name = name;

this.position = position;

this.department = department;

this.\_salary = salary; // \_salary is conventionally treated as a private property

}

// Getter for salary

get salary() {

return this.\_salary;

}

// Setter for salary

set salary(newSalary) {

if (newSalary < 0) {

console.log("Salary cannot be negative.");

} else {

this.\_salary = newSalary;

}

}

displayInfo() {

console.log(`Name: ${this.name}`);

console.log(`Position: ${this.position}`);

console.log(`Department: ${this.department}`);

console.log(`Salary: $${this.\_salary}`);

}

}

// Create an Employee object

const employee5 = new Employee("Alice Johnson", "Product Manager", "Product", 90000);

employee5.displayInfo();

// Output:

// Name: Alice Johnson

// Position: Product Manager

// Department: Product

// Salary: $90000

employee5.salary = 95000; // Set a new salary

console.log(employee5.salary); // 95000

employee5.salary = -5000; // Invalid salary input

// Output: "Salary cannot be negative."

### Summary of Key OOP Principles in JavaScript:

1. **Objects**: Use objects to represent entities like employees, departments, etc.
2. **Classes & Constructors**: Use classes to define blueprints and constructors to initialize objects.
3. **Inheritance**: Extend base classes to create more specific classes, like Manager inheriting from Employee.
4. **Encapsulation**: Hide sensitive data and provide controlled access through getters and setters.
5. **Polymorphism**: Methods in different classes can be overridden to provide custom behavior.
6. **Abstraction**: Hide complex internal processes and expose only necessary functionality.
7. **Getters/Setters**: Provide controlled access to object properties.

By organizing the code in this way, you can model real-world entities like employees and departments in an object-oriented manner, making the system more maintainable and scalable.