Problem Statements

**1. Employee Management System**

* **Description**: Build an employee management system with the following requirements:
  + Use an abstract class Employee with fields like employeeId, name, and baseSalary.
  + Provide an abstract method calculateSalary() and a concrete method displayDetails().
  + Create two subclasses: FullTimeEmployee and PartTimeEmployee, implementing calculateSalary() based on work hours or fixed salary.
  + Use encapsulation to restrict direct access to fields and provide getter and setter methods.
  + Create an interface Department with methods like assignDepartment() and getDepartmentDetails().
  + Ensure polymorphism by processing a list of employees and displaying their details using the Employee reference.

Soln:

abstract class Employee {

private int employeeId;

private String name;

private double baseSalary;

public Employee(int employeeId, String name, double baseSalary) {

this.employeeId = employeeId;

this.name = name;

this.baseSalary = baseSalary;

}

public int getEmployeeId() { return employeeId; }

public String getName() { return name; }

public double getBaseSalary() { return baseSalary; }

public void setBaseSalary(double baseSalary) { this.baseSalary = baseSalary; }

public abstract double calculateSalary();

public void displayDetails() {

System.out.println("ID: " + employeeId + ", Name: " + name + ", Base Salary: " + baseSalary);

}

}

class FullTimeEmployee extends Employee {

public FullTimeEmployee(int id, String name, double salary) {

super(id, name, salary);

}

@Override

public double calculateSalary() {

return getBaseSalary(); // fixed salary

}

}

class PartTimeEmployee extends Employee {

private int hoursWorked;

private double hourlyRate;

public PartTimeEmployee(int id, String name, double base, int hours, double rate) {

super(id, name, base);

this.hoursWorked = hours;

this.hourlyRate = rate;

}

@Override

public double calculateSalary() {

return hoursWorked \* hourlyRate;

}

}

interface Department {

void assignDepartment(String dept);

String getDepartmentDetails();

}

public class Encapsulation {

public static void main(String[] args) {

Employee[] employees = {

new FullTimeEmployee(1, "Alice", 50000),

new PartTimeEmployee(2, "Bob", 0, 20, 200)

};

for (Employee e : employees) {

e.displayDetails();

System.out.println("Salary: " + e.calculateSalary());

}

}

}

**2. E-Commerce Platform**

* **Description**: Develop a simplified e-commerce platform:
  + Create an abstract class Product with fields like productId, name, and price, and an abstract method calculateDiscount().
  + Extend it into concrete classes: Electronics, Clothing, and Groceries.
  + Implement an interface Taxable with methods calculateTax() and getTaxDetails() for applicable product categories.
  + Use encapsulation to protect product details, allowing updates only through setter methods.
  + Showcase polymorphism by creating a method that calculates and prints the final price (price + tax - discount) for a list of Product.

Soln:

abstract class Product {

private int productId;

private String name;

private double price;

public Product(int productId, String name, double price) {

this.productId = productId;

this.name = name;

this.price = price;

}

public double getPrice() { return price; }

public abstract double calculateDiscount();

public int getProductId() { return productId; }

public String getName() { return name; }

}

interface Taxable {

double calculateTax();

String getTaxDetails();

}

class Electronics extends Product implements Taxable {

public Electronics(int id, String name, double price) {

super(id, name, price);

}

public double calculateDiscount() { return getPrice() \* 0.1; }

public double calculateTax() { return getPrice() \* 0.18; }

public String getTaxDetails() { return "18% GST"; }

}

class Clothing extends Product implements Taxable {

public Clothing(int id, String name, double price) {

super(id, name, price);

}

public double calculateDiscount() { return getPrice() \* 0.2; }

public double calculateTax() { return getPrice() \* 0.12; }

public String getTaxDetails() { return "12% GST"; }

}

class Groceries extends Product {

public Groceries(int id, String name, double price) {

super(id, name, price);

}

public double calculateDiscount() { return getPrice() \* 0.05; }

}

public class Encapsulation {

public static void main(String[] args) {

Product[] items = {

new Electronics(101, "Laptop", 50000),

new Clothing(102, "Jeans", 2000),

new Groceries(103, "Rice", 1000)

};

for (Product p : items) {

double discount = p.calculateDiscount();

double tax = (p instanceof Taxable) ? ((Taxable)p).calculateTax() : 0;

double finalPrice = p.getPrice() + tax - discount;

System.out.println(p.getName() + ": Final Price = " + finalPrice);

}

}

}

**3. Vehicle Rental System**

* **Description**: Design a system to manage vehicle rentals:
  + Define an abstract class Vehicle with fields like vehicleNumber, type, and rentalRate.
  + Add an abstract method calculateRentalCost(int days).
  + Create subclasses Car, Bike, and Truck with specific implementations of calculateRentalCost().
  + Use an interface Insurable with methods calculateInsurance() and getInsuranceDetails().
  + Apply encapsulation to restrict access to sensitive details like insurance policy numbers.
  + Demonstrate polymorphism by iterating over a list of vehicles and calculating rental and insurance costs for each.

Soln:

abstract class Vehicle {

private String vehicleNumber, type;

private double rentalRate;

public Vehicle(String number, String type, double rate) {

this.vehicleNumber = number;

this.type = type;

this.rentalRate = rate;

}

public String getVehicleNumber() { return vehicleNumber; }

public double getRentalRate() { return rentalRate; }

public abstract double calculateRentalCost(int days);

}

interface Insurable {

double calculateInsurance();

String getInsuranceDetails();

}

class Car extends Vehicle implements Insurable {

public Car(String number, double rate) {

super(number, "Car", rate);

}

public double calculateRentalCost(int days) {

return getRentalRate() \* days;

}

public double calculateInsurance() { return 1000; }

public String getInsuranceDetails() { return "Standard Car Insurance"; }

}

class Bike extends Vehicle {

public Bike(String number, double rate) {

super(number, "Bike", rate);

}

public double calculateRentalCost(int days) {

return getRentalRate() \* days;

}

}

class Truck extends Vehicle implements Insurable {

public Truck(String number, double rate) {

super(number, "Truck", rate);

}

public double calculateRentalCost(int days) {

return getRentalRate() \* days + 500; // additional charge

}

public double calculateInsurance() { return 2000; }

public String getInsuranceDetails() { return "Heavy Vehicle Insurance"; }

}

public class Encapsulation {

public static void main(String[] args) {

Vehicle[] fleet = {

new Car("KA1234", 1000),

new Bike("KA5678", 300),

new Truck("KA9012", 2000)

};

for (Vehicle v : fleet) {

System.out.println("Vehicle: " + v.getVehicleNumber() + ", Cost: " + v.calculateRentalCost(5));

if (v instanceof Insurable) {

Insurable i = (Insurable)v;

System.out.println("Insurance: " + i.calculateInsurance());

}

}

}

}

**4. Banking System**

* **Description**: Create a banking system with different account types:
  + Define an abstract class BankAccount with fields like accountNumber, holderName, and balance.
  + Add methods like deposit(double amount) and withdraw(double amount) (concrete) and calculateInterest() (abstract).
  + Implement subclasses SavingsAccount and CurrentAccount with unique interest calculations.
  + Create an interface Loanable with methods applyForLoan() and calculateLoanEligibility().
  + Use encapsulation to secure account details and restrict unauthorized access.
  + Demonstrate polymorphism by processing different account types and calculating interest dynamically.

Soln:

abstract class BankAccount {

private String accountNumber, holderName;

private double balance;

public BankAccount(String accountNumber, String holderName, double balance) {

this.accountNumber = accountNumber;

this.holderName = holderName;

this.balance = balance;

}

public void deposit(double amount) { balance += amount; }

public void withdraw(double amount) { if (balance >= amount) balance -= amount; }

public double getBalance() { return balance; }

public abstract double calculateInterest();

}

interface Loanable {

void applyForLoan(double amount);

boolean calculateLoanEligibility();

}

class SavingsAccount extends BankAccount implements Loanable {

public SavingsAccount(String acc, String name, double balance) {

super(acc, name, balance);

}

public double calculateInterest() {

return getBalance() \* 0.04;

}

public void applyForLoan(double amount) {

System.out.println("Savings Loan Applied: " + amount);

}

public boolean calculateLoanEligibility() {

return getBalance() > 10000;

}

}

class CurrentAccount extends BankAccount {

public CurrentAccount(String acc, String name, double balance) {

super(acc, name, balance);

}

public double calculateInterest() {

return 0; // no interest

}

}

public class Encapsulation {

public static void main(String[] args) {

BankAccount[] accounts = {

new SavingsAccount("A1", "John", 15000),

new CurrentAccount("A2", "Jane", 20000)

};

for (BankAccount a : accounts) {

System.out.println("Interest: " + a.calculateInterest());

}

}

}

**5. Library Management System**

* **Description**: Develop a library management system:
  + Use an abstract class LibraryItem with fields like itemId, title, and author.
  + Add an abstract method getLoanDuration() and a concrete method getItemDetails().
  + Create subclasses Book, Magazine, and DVD, overriding getLoanDuration() with specific logic.
  + Implement an interface Reservable with methods reserveItem() and checkAvailability().
  + Apply encapsulation to secure details like the borrower’s personal data.
  + Use polymorphism to allow a general LibraryItem reference to manage all items, regardless of type.

Soln:

abstract class LibraryItem {

private int itemId;

private String title, author;

public LibraryItem(int itemId, String title, String author) {

this.itemId = itemId;

this.title = title;

this.author = author;

}

public int getItemId() { return itemId; }

public String getTitle() { return title; }

public String getAuthor() { return author; }

public abstract int getLoanDuration();

public void getItemDetails() {

System.out.println("ID: " + itemId + ", Title: " + title + ", Author: " + author);

}

}

interface Reservable {

void reserveItem();

boolean checkAvailability();

}

class Book extends LibraryItem implements Reservable {

public Book(int id, String title, String author) {

super(id, title, author);

}

public int getLoanDuration() { return 14; }

public void reserveItem() {

System.out.println("Book reserved.");

}

public boolean checkAvailability() {

return true;

}

}

class Magazine extends LibraryItem {

public Magazine(int id, String title, String author) {

super(id, title, author);

}

public int getLoanDuration() { return 7; }

}

class DVD extends LibraryItem {

public DVD(int id, String title, String author) {

super(id, title, author);

}

public int getLoanDuration() { return 3; }

}

public class Encapsulation {

public static void main(String[] args) {

LibraryItem[] items = {

new Book(1, "Java Fundamentals", "Oracle"),

new Magazine(2, "Tech Monthly", "TechHouse"),

new DVD(3, "The Matrix", "Wachowski")

};

for (LibraryItem item : items) {

item.getItemDetails();

System.out.println("Loan Duration: " + item.getLoanDuration() + " days");

}

}

}

**6. Online Food Delivery System**

* **Description**: Create an online food delivery system:
  + Define an abstract class FoodItem with fields like itemName, price, and quantity.
  + Add abstract methods calculateTotalPrice() and concrete methods like getItemDetails().
  + Extend it into classes VegItem and NonVegItem, overriding calculateTotalPrice() to include additional charges (e.g., for non-veg items).
  + Use an interface Discountable with methods applyDiscount() and getDiscountDetails().
  + Demonstrate encapsulation to restrict modifications to order details and use polymorphism to handle different types of food items in a single order-processing method.

Soln:

abstract class FoodItem {

private String itemName;

private double price;

private int quantity;

public FoodItem(String itemName, double price, int quantity) {

this.itemName = itemName;

this.price = price;

this.quantity = quantity;

}

public double getPrice() { return price; }

public int getQuantity() { return quantity; }

public String getItemName() { return itemName; }

public abstract double calculateTotalPrice();

public void getItemDetails() {

System.out.println("Item: " + itemName + ", Price: " + price + ", Quantity: " + quantity);

}

}

interface Discountable {

double applyDiscount();

String getDiscountDetails();

}

class VegItem extends FoodItem implements Discountable {

public VegItem(String name, double price, int qty) {

super(name, price, qty);

}

public double calculateTotalPrice() {

return getPrice() \* getQuantity();

}

public double applyDiscount() {

return calculateTotalPrice() \* 0.1;

}

public String getDiscountDetails() {

return "10% discount on Veg items";

}

}

class NonVegItem extends FoodItem {

public NonVegItem(String name, double price, int qty) {

super(name, price, qty);

}

public double calculateTotalPrice() {

return getPrice() \* getQuantity() + 30; // extra charge

}

}

public class Encapsulation {

public static void main(String[] args) {

FoodItem[] order = {

new VegItem("Paneer Curry", 120, 2),

new NonVegItem("Chicken Biryani", 180, 1)

};

for (FoodItem item : order) {

item.getItemDetails();

double total = item.calculateTotalPrice();

if (item instanceof Discountable) {

total -= ((Discountable)item).applyDiscount();

}

System.out.println("Final Price: " + total);

}

}

}

**7. Hospital Patient Management**

* **Description**: Design a system to manage patients in a hospital:
  + Create an abstract class Patient with fields like patientId, name, and age.
  + Add an abstract method calculateBill() and a concrete method getPatientDetails().
  + Extend it into subclasses InPatient and OutPatient, implementing calculateBill() with different billing logic.
  + Implement an interface MedicalRecord with methods addRecord() and viewRecords().
  + Use encapsulation to protect sensitive patient data like diagnosis and medical history.
  + Use polymorphism to handle different patient types and display their billing details dynamically.

Soln:

abstract class Patient {

private int patientId;

private String name;

private int age;

public Patient(int id, String name, int age) {

this.patientId = id;

this.name = name;

this.age = age;

}

public int getPatientId() { return patientId; }

public String getName() { return name; }

public void getPatientDetails() {

System.out.println("ID: " + patientId + ", Name: " + name + ", Age: " + age);

}

public abstract double calculateBill();

}

interface MedicalRecord {

void addRecord(String record);

String viewRecords();

}

class InPatient extends Patient implements MedicalRecord {

private int daysAdmitted;

private String records = "";

public InPatient(int id, String name, int age, int days) {

super(id, name, age);

this.daysAdmitted = days;

}

public double calculateBill() {

return daysAdmitted \* 2000;

}

public void addRecord(String record) { records += record + "\n"; }

public String viewRecords() { return records; }

}

class OutPatient extends Patient {

public OutPatient(int id, String name, int age) {

super(id, name, age);

}

public double calculateBill() {

return 500; // fixed consultation

}

}

public class Encapsulation {

public static void main(String[] args) {

Patient[] patients = {

new InPatient(1, "Ravi", 40, 5),

new OutPatient(2, "Neha", 30)

};

for (Patient p : patients) {

p.getPatientDetails();

System.out.println("Bill: " + p.calculateBill());

}

}

}

**8. Ride-Hailing Application**

* **Description**: Develop a ride-hailing application:
  + Define an abstract class Vehicle with fields like vehicleId, driverName, and ratePerKm.
  + Add abstract methods calculateFare(double distance) and a concrete method getVehicleDetails().
  + Create subclasses Car, Bike, and Auto, overriding calculateFare() based on type-specific rates.
  + Use an interface GPS with methods getCurrentLocation() and updateLocation().
  + Secure driver and vehicle details using encapsulation.
  + Demonstrate polymorphism by creating a method to calculate fares for different vehicle types dynamically.

Soln:

abstract class Vehicle {

private String vehicleId, driverName;

private double ratePerKm;

public Vehicle(String vehicleId, String driverName, double rate) {

this.vehicleId = vehicleId;

this.driverName = driverName;

this.ratePerKm = rate;

}

public String getVehicleId() { return vehicleId; }

public String getDriverName() { return driverName; }

public double getRatePerKm() { return ratePerKm; }

public void getVehicleDetails() {

System.out.println("Vehicle: " + vehicleId + ", Driver: " + driverName);

}

public abstract double calculateFare(double distance);

}

interface GPS {

String getCurrentLocation();

void updateLocation(String location);

}

class Car extends Vehicle implements GPS {

private String currentLocation;

public Car(String id, String driver, double rate) {

super(id, driver, rate);

}

public double calculateFare(double distance) {

return distance \* getRatePerKm();

}

public String getCurrentLocation() { return currentLocation; }

public void updateLocation(String loc) { currentLocation = loc; }

}

class Bike extends Vehicle {

public Bike(String id, String driver, double rate) {

super(id, driver, rate);

}

public double calculateFare(double distance) {

return distance \* getRatePerKm() - 10; // discount

}

}

class Auto extends Vehicle {

public Auto(String id, String driver, double rate) {

super(id, driver, rate);

}

public double calculateFare(double distance) {

return distance \* getRatePerKm() + 15; // base fare

}

}

public class Encapsulation {

public static void main(String[] args) {

Vehicle[] rides = {

new Car("V1001", "Raj", 15),

new Bike("V1002", "Sam", 10),

new Auto("V1003", "Ali", 8)

};

for (Vehicle v : rides) {

v.getVehicleDetails();

System.out.println("Fare for 10km: " + v.calculateFare(10));

}

}

}