A Hospital Management System maintains information about Doctor and Patient.
Create a class Doctor with the following attributes:
doctorId (int)
doctorName (String)
specialization (String)
Create a class Patient with the following attributes:
patientId (int)
patientName (String)
assignedDoctor (Doctor)
(Here, Patient has an association relationship with Doctor)
In the Patient class:
Write a method assignDoctor(Doctor d) that accepts a Doctor object as a method parameter and assigns it to the patient.
Write a method generateDoctorInfo() that returns a Doctor object.
Create a class Hospital that contains a method:
admitPatient(Patient p) which accepts a Patient object as a parameter and prints the patient details along with the assigned doctor's information.
In the main() method:
Create at least one Doctor object.
Create a Patient object and assign a doctor using assignDoctor().
Pass the Patient object to the admitPatient() method.

এই প্রশ্নটি নিচের OOP টপিক থেকে করা হয়েছে:

📌 Main Topics:

- **Class & Object**
- **Association Relationship** (Patient has-a Doctor)
- **Constructor**
- **Method**
- **Method Parameter Passing (Object as Parameter)**
- **Encapsulation**
- **Object Returning from Method**

Output

=== Patient Details ===
Patient ID: 201
Patient Name: Karim
=== Assigned Doctor Details ===
Doctor ID: 101
Doctor Name: Dr. Rahman
Specialization: Cardiologist

Solution 👍

```
// Doctor Class
class Doctor {
    int doctorId;
    String doctorName;
    String specialization;

    // Constructor
    public Doctor(int doctorId, String doctorName, String specialization) {
        this.doctorId = doctorId;
        this.doctorName = doctorName;
        this.specialization = specialization;
    }

    // Method to display doctor info
    public void displayDoctorInfo() {
        System.out.println("Doctor ID: " + doctorId);
        System.out.println("Doctor Name: " + doctorName);
        System.out.println("Specialization: " + specialization);
    }
}

// Patient Class
class Patient {
    int patientId;
    String patientName;
    Doctor assignedDoctor;  // Association

    // Constructor
    public Patient(int patientId, String patientName) {
        this.patientId = patientId;
        this.patientName = patientName;
    }

    // Assign Doctor Method
    public void assignDoctor(Doctor d) {
        this.assignedDoctor = d;
    }
    // Return Doctor Object
    public Doctor generateDoctorInfo() {
        return assignedDoctor;
    }

    // Display Patient Info
    public void displayPatientInfo() {
        System.out.println("Patient ID: " + patientId);
        System.out.println("Patient Name: " + patientName);
    }
}
// Hospital Class
class Hospital {
    public void admitPatient(Patient p) {
        System.out.println("=== Patient Details ===");
        p.displayPatientInfo();
        System.out.println("=== Assigned Doctor Details ===");
        Doctor d = p.generateDoctorInfo();
        if (d != null) {
            d.displayDoctorInfo();
        } else {
            System.out.println("No Doctor Assigned.");
        }
    }
}
// Main Class
public class Main {
    public static void main(String[] args) {

        // Create Doctor Object
        Doctor doc1 = new Doctor(101, "Dr. Rahman", "Cardiologist");

        // Create Patient Object
        Patient pat1 = new Patient(201, "Karim");

        // Assign Doctor to Patient
        pat1.assignDoctor(doc1);

        // Create Hospital Object
        Hospital hospital = new Hospital();
        // Admit Patient
        hospital.admitPatient(pat1);
    }
}
```

A Payment Processing System supports different types of payments.

Part A: Base Class – Payment
Create a class Payment with the following:
Attribute:
amount (double)
Constructor to initialize the amount.
A method void processPayment()
that prints: "Processing general payment of amount: <amount>"
Part B: Derived Class – OnlinePayment
Create a class OnlinePayment that extends Payment with:
Additional attribute:
platform (String)
Constructor to initialize both amount and platform.
Override the processPayment() method to print:
"Processing online payment via <platform> of amount: <amount>"
Part C: Further Derived Class – MobileBankingPayment
Create a class MobileBankingPayment that extends OnlinePayment with:
Additional attribute:
mobileNumber (String)
Constructor to initialize all attributes.
Override the processPayment() method to print:
"Processing mobile banking payment from <mobileNumber> via <platform> of amount: <amount>"
Part D: Runtime Polymorphism Demonstration
Create a separate class PaymentService with a method:
void executePayment(Payment p)
This method should:
Accept a Payment reference
Call processPayment() on the received object
Part E: Main Method
In the main() method:
Create an object of MobileBankingPayment
Store it in a Payment reference variable
Pass the reference to executePayment()

কোন টপিক থেকে প্রশ্নটি করা হয়েছে?

Class & Object
Inheritance (Single & Multilevel)

- Payment → OnlinePayment → MobileBankingPayment

Method Overriding
Runtime Polymorphism (Dynamic Method Dispatch)
Upcasting (Child object → Parent reference)
Constructor Chaining (super keyword)

Solution 👍

```
// Base Class
class Payment {
    double amount;

    // Constructor
    public Payment(double amount) {
        this.amount = amount;
    }

    // Method
    public void processPayment() {
        System.out.println("Processing general payment of amount: " + amount);
    }
}

// Derived Class
class OnlinePayment extends Payment {
    String platform;

    // Constructor
    public OnlinePayment(double amount, String platform) {
        super(amount); // Call parent constructor
        this.platform = platform;
    }

    // Overriding method
    @Override
    public void processPayment() {
        System.out.println("Processing online payment via "
                + platform + " of amount: " + amount);
    }
}

// Further Derived Class
class MobileBankingPayment extends OnlinePayment {
    String mobileNumber;

    // Constructor
    public MobileBankingPayment(double amount, String platform, String mobileNumber) {
        super(amount, platform); // Call parent constructor
        this.mobileNumber = mobileNumber;
    }

    // Overriding method
    @Override
    public void processPayment() {
        System.out.println("Processing mobile banking payment from "
                + mobileNumber + " via " + platform
                + " of amount: " + amount);
    }
}

// Runtime Polymorphism Class
class PaymentService {

    public void executePayment(Payment p) {
        p.processPayment();  // Dynamic Method Dispatch
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {

        // Upcasting (Runtime Polymorphism)
        Payment p = new MobileBankingPayment(
                5000.0,
                "bKash",
                "017XXXXXXXX"
        );

        PaymentService service = new PaymentService();

        service.executePayment(p);
    }
}
```

A Smart Home Automation System controls different types of devices.

Part A: Base Class – Device

Create a class Device with:

Attribute:

deviceId (int)

Constructor to initialize deviceId

Method:

void operate()

that prints:

"Operating generic device"

Part B: Intermediate Class – Appliance

Create a class Appliance that extends Device with:

Additional attribute:

powerConsumption (double)

Constructor to initialize both deviceId and powerConsumption

Override the operate() method to print:

"Operating appliance with power consumption: <powerConsumption>W"

Part C: Derived Class – SmartAppliance

Create a class SmartAppliance that extends Appliance with:

Additional attribute:

wifiEnabled (boolean)

Constructor to initialize all attributes

Override the operate() method to:

Call the parent class version using super.operate()

Then print:

"Smart features enabled: <wifiEnabled>"

Part D: Runtime Polymorphism

Create a class HomeController with a method:

void controlDevice(Device d)

This method should:

Accept a Device reference

Call operate() on the received object

Part E: Main Method

In the main() method:

Create an object of SmartAppliance

Assign it to a Device reference variable

Pass the reference to controlDevice()

কোন টপিক থেকে প্রশ্নটি করা হয়েছে?

📌 **Main Topics:**

1. **Class & Object**
2. **Multilevel Inheritance**
   - Device → Appliance → SmartAppliance
3. **Method Overriding**
4. **super keyword (Method Calling)**
5. **Runtime Polymorphism**
6. **Upcasting (Child object → Parent reference)**

Solution 👍

```
// Base Class
class Device {
    int deviceId;
    // Constructor
    public Device(int deviceId) {
        this.deviceId = deviceId;
    }
    // Method
    public void operate() {
        System.out.println("Operating generic device");
    }
}
// Intermediate Class
class Appliance extends Device {
    double powerConsumption;
    // Constructor
    public Appliance(int deviceId, double powerConsumption) {
        super(deviceId); // Call parent constructor
        this.powerConsumption = powerConsumption;
    }
    // Overriding method
    @Override
    public void operate() {
        System.out.println("Operating appliance with power consumption: "
                + powerConsumption + "W");
    }
}
// Derived Class
class SmartAppliance extends Appliance {
    boolean wifiEnabled;
    // Constructor
    public SmartAppliance(int deviceId, double powerConsumption, boolean wifiEnabled) {
        super(deviceId, powerConsumption); // Call parent constructor
        this.wifiEnabled = wifiEnabled;
    }
    // Overriding method
    @Override
    public void operate() {
        super.operate();  // Call parent version
        System.out.println("Smart features enabled: " + wifiEnabled);
    }
}
// Runtime Polymorphism Class
class HomeController {
    public void controlDevice(Device d) {
        d.operate();
// Dynamic Method Dispatch
    }
}
// Main Class
public class Main {
    public static void main(String[] args) {
        // Upcasting
        Device d = new SmartAppliance(101, 1500.0, true);
HomeController controller = new HomeController();
        controller.controlDevice(d);
    }
}
```

Problem 4:An Employee Management System manages different levels of employees in an organization.

Part A: Base Class – Employee

Create a class Employee with:

Attributes:

empId (int)

basicSalary (double)

Constructor to initialize all attributes

Method:

double calculateSalary()

that returns the basicSalary

Part B: Intermediate Class – Manager

Create a class Manager that extends Employee with:

Additional attribute:

managementAllowance (double)

Constructor to initialize all attributes

Override the calculateSalary() method to return:

basicSalary + managementAllowance

Part C: Derived Class – SeniorManager

Create a class SeniorManager that extends Manager with:

Additional attribute:

performanceBonus (double)

Constructor to initialize all attributes

Override the calculateSalary() method to return:

basicSalary + managementAllowance + performanceBonus

Part D: Runtime Polymorphism Demonstration

Create a class PayrollService with a method:

void generatePayroll(Employee e)

This method should:

Accept an Employee reference

Call calculateSalary()

Print the total salary

Part E: Main Method

In the main() method:

Create an object of SeniorManager

Store it in an Employee reference variable

Pass the reference to generatePayroll()

কোন টপিক থেকে প্রশ্নটি করা হয়েছে?

📌 **Main Concepts:**

1. **Class & Object**
2. **Multilevel Inheritance**
    - Employee → Manager → SeniorManager
3. **Method Overriding**
4. **Runtime Polymorphism (Dynamic Method Dispatch)**
5. **Upcasting**
6. **Constructor Chaining (super keyword)**

Solution 👍

```
// Base Class
class Employee {
    int empId;
    double basicSalary;
    // Constructor
    public Employee(int empId, double basicSalary) {
        this.empId = empId;
        this.basicSalary = basicSalary;
    }
    // Method
    public double calculateSalary() {
        return basicSalary;
    }
}
// Intermediate Class
class Manager extends Employee {
    double managementAllowance;
    // Constructor
    public Manager(int empId, double basicSalary, double managementAllowance) {
        super(empId, basicSalary);  // Call parent constructor
        this.managementAllowance = managementAllowance;
    }
    // Overriding method
    @Override
    public double calculateSalary() {
        return basicSalary + managementAllowance;
    }
}
// Derived Class
class SeniorManager extends Manager {
    double performanceBonus;

    // Constructor
    public SeniorManager(int empId, double basicSalary,
                         double managementAllowance, double performanceBonus) {
        super(empId, basicSalary, managementAllowance); // Call parent constructor
        this.performanceBonus = performanceBonus;
    }
    // Overriding method
    @Override
    public double calculateSalary() {
        return basicSalary + managementAllowance + performanceBonus;
    }
}
// Runtime Polymorphism Class
class PayrollService {
    public void generatePayroll(Employee e) {
        double totalSalary = e.calculateSalary();
        System.out.println("Total Salary: " + totalSalary);
    }
}
// Main Class
public class Main {
    public static void main(String[] args) {
        // Upcasting
        Employee e = new SeniorManager(101, 50000, 10000, 15000);
        PayrollService service = new PayrollService();
        service.generatePayroll(e);
    }
}
```

A Company Resource Management System keeps track of employees and the total number of staff members using a static class variable.

Part A: Base Class – Staff

Create a class Staff with:

Attributes:

staffId (int)

basicPay (double)

Static attribute:

totalStaff (int)

(This variable counts the total number of staff objects created.)

Constructor to initialize instance variables and increment totalStaff.

Method:

double calculatePay()

that returns basicPay.

Part B: Intermediate Class – Supervisor

Create a class Supervisor that extends Staff with:

Additional attribute:

supervisionAllowance (double)

Constructor to initialize all attributes.

Override calculatePay() to return:

basicPay + supervisionAllowance

Part C: Derived Class – Manager

Create a class Manager that extends Supervisor with:

Additional attribute:

managementBonus (double)

Constructor to initialize all attributes.

Override calculatePay() to return:

basicPay + supervisionAllowance + managementBonus

Part D: Runtime Polymorphism

Create a class PayrollSystem with a method:

void generatePaySlip(Staff s)

This method should:

Accept a Staff reference

Call calculatePay() and print the total pay

Part E: Main Method

In the main() method:

Create at least:

one Supervisor object

one Manager object

Store both objects in Staff reference variables.

Pass them to generatePaySlip().

Print the value of Staff.totalStaff.

কোন টপিক থেকে প্রশ্নটি করা হয়েছে?

📌 **Main Concepts Used:**

1. **Abstract Class**
2. **Abstract Method**
3. **Concrete Method**
4. **Interface**
5. **Multiple Inheritance using Interface**
6. **Method Overriding**
7. **Runtime Polymorphism**
8. **Upcasting**
9. **Encapsulation (Protected attributes)**

Solution 👍

```
// Base Class
class Employee {
    int empId;
    double basicSalary;
    // Constructor
    public Employee(int empId, double basicSalary) {
        this.empId = empId;
        this.basicSalary = basicSalary;
    }
    // Method
    public double calculateSalary() {
        return basicSalary;
    }
}
// Intermediate Class
class Manager extends Employee {
    double managementAllowance;
    // Constructor
    public Manager(int empId, double basicSalary, double managementAllowance) {
        super(empId, basicSalary);  // Call parent constructor
        this.managementAllowance = managementAllowance;
    }
    // Overriding method
    @Override
    public double calculateSalary() {
        return basicSalary + managementAllowance;
    }
}
// Derived Class
class SeniorManager extends Manager {
    double performanceBonus;
    // Constructor
    public SeniorManager(int empId, double basicSalary,
                         double managementAllowance, double performanceBonus) {
        super(empId, basicSalary, managementAllowance); // Call parent constructor
        this.performanceBonus = performanceBonus;
    }
    // Overriding method
    @Override
    public double calculateSalary() {
        return basicSalary + managementAllowance + performanceBonus;
    }
}
// Runtime Polymorphism Class
class PayrollService {
    public void generatePayroll(Employee e) {
        double totalSalary = e.calculateSalary();
        System.out.println("Total Salary: " + totalSalary);
    }
}
// Main Class
public class Main {
    public static void main(String[] args) {
        // Upcasting
        Employee e = new SeniorManager(101, 50000, 10000, 15000);
        PayrollService service = new PayrollService();
        service.generatePayroll(e);
    }
}
```

Output 👍

Total Salary: 75000.0

A University Course Management System manages different types of courses and their evaluation methods.

Part A: Abstract Class – Course

Create an abstract class Course with:

Protected attributes:

courseCode (String)

credit (double)

Constructor to initialize the attributes.

An abstract method:

abstract double calculateFinalMarks();

A concrete method:

void displayCourseInfo()

that prints course code and credit.

Part B: Interface – TheoryEvaluation

Create an interface TheoryEvaluation with:

double theoryMarks();

Part C: Interface – LabEvaluation

Create another interface LabEvaluation with:

double labMarks();

Part D: Multiple Inheritance Using Interfaces

Create a class HybridCourse that:

extends Course

implements both TheoryEvaluation and LabEvaluation

In this class:

Implement all interface methods.

Override calculateFinalMarks() to return:

theoryMarks() + labMarks()

Part E: Runtime Polymorphism

Create a class CourseService with a method:

void printFinalResult(Course c)

This method should:

Accept a Course reference

Call calculateFinalMarks()

Display the final marks

Part F: Main Method

In the main() method:

Create an object of HybridCourse

Store it in a Course reference variable

Pass the reference to printFinalResult()

কোন টপিক থেকে প্রশ্নটি করা হয়েছে?

📌 Main Concepts Used:

1. Abstract Class
2. Abstract Method
3. Concrete Method
4. Interface
5. Multiple Inheritance using Interface
6. Method Overriding
7. Runtime Polymorphism
8. Upcasting

Solution 👍

```
// Abstract Class
abstract class Course {
    protected String courseCode;
    protected double credit;
    // Constructor
    public Course(String courseCode, double credit) {
        this.courseCode = courseCode;
        this.credit = credit;
    }
    // Abstract Method
    abstract double calculateFinalMarks();
    // Concrete Method
    public void displayCourseInfo() {
        System.out.println("Course Code: " + courseCode);
        System.out.println("Credit: " + credit);
    }
}
// Interface 1
interface TheoryEvaluation {
    double theoryMarks();
}
// Interface 2
interface LabEvaluation {
    double labMarks();
}
// HybridCourse Class
class HybridCourse extends Course implements TheoryEvaluation, LabEvaluation {
    double theory;
    double lab;
    public HybridCourse(String courseCode, double credit, double theory, double lab) {
        super(courseCode, credit);
        this.theory = theory;
        this.lab = lab;
    }
    // Implement interface methods
    public double theoryMarks() {
        return theory;
    }
    public double labMarks() {
        return lab;
    }
    // Override abstract method
    @Override
    double calculateFinalMarks() {
        return theoryMarks() + labMarks();
    }
}
// Service Class
class CourseService {
    public void printFinalResult(Course c) {
        c.displayCourseInfo();
        double finalMarks = c.calculateFinalMarks();
        System.out.println("Final Marks: " + finalMarks);
    }
}
// Main Class
public class Main {
    public static void main(String[] args) {
        // Upcasting
        Course c = new HybridCourse("CSE101", 3.0, 70, 25);
        CourseService service = new CourseService();
        service.printFinalResult(c);
    }
}
```

Output 👍

Course Code: CSE101

Credit: 3.0

Final Marks: 95.0

Create a class named StudyTracker that monitors a student's daily study progress.

Class: StudyTracker

The class should contain the following instance variables:

studentId (int) – unique ID of the student

hoursStudied (int) – total hours studied by the student

dailyGoal (int) – target study hours for a day

The class should also contain the following static variable:

totalTrackers (int) – keeps track of the total number of students using the StudyTracker app (Suggestion: count the total number of StudyTracker objects created)

Tasks

1. Constructor

Initialize all instance variables using a constructor.

Increase totalTrackers whenever a new StudyTracker object is created.

2. Methods

Implement the following methods:

(a) addStudyHours(int h)

Increases hoursStudied by h

Returns the current StudyTracker object

(b) mergeTracker(StudyTracker st)

Adds the hoursStudied of the parameter student to the current student

Returns the current StudyTracker object

(c) remainingHours()

Calculates how many more hours are needed to reach dailyGoal

Displays the remaining hours

Returns the current StudyTracker object

(d) compareProgress(StudyTracker st)

Compares the hoursStudied of the current student with another student

Displays:

"Student <studentId> is ahead" if the current student studied more hours

"Student <studentId> is behind" if the current student studied fewer hours

"Both students studied the same amount" if both studied equally

Returns the current StudyTracker object

(e) resetIfGoalReached()

Checks whether hoursStudied is greater than or equal to dailyGoal

If the goal is reached:

Displays a congratulatory message

Resets hoursStudied to 0

Otherwise:

Displays "Daily goal not reached yet"

Returns the current StudyTracker object

(f) getTotalTrackers() (static method)

Displays the total number of StudyTracker objects created

Main Class

In the main method:

Create an array of StudyTracker objects (size ≥ 2).

Initialize the array with at least two StudyTracker objects.

Add study hours to different students using addStudyHours().

Merge one student's study record into another using mergeTracker().

Compare the study progress of two students using compareProgress().

Call remainingHours() for at least one student.

Reset a student's study hours if the daily goal is reached using resetIfGoalReached().
Display the total number of tracker objects using getTotalTrackers().

কোন টপিক থেকে প্রশ্নটি করা হয়েছে?

📌 Main Concepts Used:

1. Class & Object
2. Static Variable
3. Static Method
4. Constructor
5. Method Chaining (return this)
6. Object as Parameter
7. Array of Objects
8. Conditional Logic (if-else)

Solution 👍

```
class StudyTracker {   int studentId;    int
hoursStudied;  int dailyGoal;
  static int totalTrackers = 0;
   // Constructor
    public StudyTracker(int studentId, int
hoursStudied, int dailyGoal) {
        this.studentId = studentId;
        this.hoursStudied = hoursStudied;
        this.dailyGoal = dailyGoal;
        totalTrackers++; }
   // (a) Add Study Hours
    public StudyTracker addStudyHours(int h) {
        this.hoursStudied += h;  return this;  }
    // (b) Merge Tracker
    public StudyTracker mergeTracker(StudyTracker
st) {
        this.hoursStudied += st.hoursStudied;
        return this;  }
    // (c) Remaining Hours
    public StudyTracker remainingHours() {
        int remaining = dailyGoal - hoursStudied;
        if (remaining > 0) {
System.out.println("Student " + studentId +
" needs " + remaining + " more hours to reach daily
goal.");
} else { System.out.println("Student " + studentId + "
has reached or exceeded the daily goal."); } return
this; }
    // (d) Compare Progress
    public StudyTracker
compareProgress(StudyTracker st) {
        if (this.hoursStudied > st.hoursStudied) {
System.out.println("Student " + studentId + " is
ahead");} else if (this.hoursStudied <
st.hoursStudied) {System.out.println("Student " +
studentId + " is behind");
        } else {   System.out.println("Both students
studied the same amount");   }
        return this;   }
```

```
  // (e) Reset If Goal Reached
 public StudyTracker resetIfGoalReached() { if
(hoursStudied >= dailyGoal) {
 System.out.println("Congratulations Student " +
studentId +"! Daily goal achieved.");hoursStudied =
0;} else {  System.out.println("Daily goal not
reached yet for Student " + studentId);} return this; }
    // (f) Static Method
    public static void getTotalTrackers() {
        System.out.println("Total StudyTracker objects
created: " + totalTrackers);
    }
}
// Main Class
public class Main {
    public static void main(String[] args) {
        // Array of objects
        StudyTracker[] trackers = new StudyTracker[2];
        trackers[0] = new StudyTracker(101, 2, 5);
        trackers[1] = new StudyTracker(102, 3, 6);
        // Add study hours
        trackers[0].addStudyHours(2);
        trackers[1].addStudyHours(4);
        // Merge study hours
trackers[0].mergeTracker(trackers[1]);
        // Compare progress
    trackers[0].compareProgress(trackers[1]);
        // Remaining hours
        trackers[0].remainingHours();
        // Reset if goal reached
        trackers[0].resetIfGoalReached();
        // Display total trackers
        StudyTracker.getTotalTrackers();
    }
}
```

Output 👍

Student 101 is ahead

Student 101 has reached or exceeded the daily goal.

Congratulations Student 101! Daily goal achieved.

Total StudyTracker objects created: 2