Enhanced Java OOP Practice Problems (Any 2) (3;4 done)

# Assignment Problem 1: 🏥 Hospital Management System with Patient Privacy

## Topics Covered: Access Modifiers, Encapsulation, Immutable Medical Records, JavaBean Standards

**Requirements:** Design a hospital management system that demonstrates strict data privacy through access control, immutable medical history, and professional healthcare data handling.

## Core Tasks:

1. Immutable MedicalRecord class:
   * final class with private final String recordId, patientDNA, String[] allergies, medicalHistory[]
   * private final LocalDate birthDate, String bloodType (Permanent medical facts)
   * Constructor with HIPAA compliance validation
   * Only getters with defensive copying, medical data cannot be modified after creation
   * public final boolean isAllergicTo(String substance) - cannot be overridden for safety
2. Patient class with privacy levels:
   * private final String patientId, MedicalRecord medicalRecord

(Protected health information)

* + private String currentName, emergencyContact, insuranceInfo

(Modifiable personal data)

* + private int roomNumber, String attendingPhysician (Current treatment info)
  + Package-private String getBasicInfo() for hospital staff access
  + public String getPublicInfo() - only non-sensitive data (name, room number)

1. Constructor chaining with privacy validation:
   * Emergency admission (minimal data, generates temporary ID)
   * Standard admission (full patient information)
   * Transfer admission (imports existing medical record)
   * All constructors validate privacy permissions and data integrity
2. Separate medical staff classes:
   * Doctor class with private final String licenseNumber, specialty, Set<String> certifications
   * Nurse class with private final String nurseId, String shift, List<String> qualifications
   * Administrator class with private final String adminId, List<String> accessPermissions
   * Each class has different access levels to patient data based on role
3. HospitalSystem class with access control:
   * private final Map<String, Object> patientRegistry (stores different patient types)
   * public boolean admitPatient(Object patient, Object staff) - use instanceof for role validation
   * private boolean validateStaffAccess(Object staff, Object patient)

- privacy protection

* + Package-private methods for internal hospital operations
  + Static final constants for hospital policies and privacy rules

1. JavaBean compliance:
   * All classes follow healthcare data standards
   * Immutable medical data with only getters
   * Validated setters for modifiable information
   * Audit trail methods in toString() implementations

# Assignment Problem 2: 🏪 E-Commerce Order Processing with Immutable Products

## Topics Covered: final Keyword, Immutable Objects, Access Modifiers, Constructor Overloading

**Requirements:** Create an e-commerce system where product information is immutable, order processing follows strict business rules, and customer data has controlled access.

## Core Tasks:

1. Immutable Product class:
   * final class with private final String productId, name, category, manufacturer
   * private final double basePrice, double weight, String[] features
   * private final Map<String, String> specifications - technical details
   * Factory methods: createElectronics(), createClothing(), createBooks()
   * Only getters with defensive copying for collections
   * public final double calculateTax(String region) - cannot be overridden for business consistency
2. Customer class with privacy tiers:
   * private final String customerId, String email (Permanent account info)
   * private String name, phoneNumber, String preferredLanguage

(Modifiable personal data)

* + private final String accountCreationDate (Immutable account history)
  + Package-private getCreditRating() for internal business operations
  + public getPublicProfile() - safe customer information for reviews/ratings

1. ShoppingCart class with access control:
   * private final String cartId, String customerId (Cart ownership)
   * private List<Object> items (stores different product types)
   * private double totalAmount, int itemCount (Calculated values)
   * public boolean addItem(Object product, int quantity) - use instanceof for product validation
   * private double calculateDiscount() - internal pricing logic
   * Package-private getCartSummary() for checkout process
2. Constructor chaining for different order types:
   * Guest checkout (minimal customer info)
   * Registered customer (full account access)
   * Premium member (special pricing and features)
   * Corporate account (bulk ordering with company validation)
3. Separate order processing classes:
   * Order class with private final String orderId, LocalDateTime orderTime
   * PaymentProcessor class with private final String processorId, securityKey
   * ShippingCalculator class with private final Map<String, Double> shippingRates
   * Each class handles specific business logic with appropriate access control
4. ECommerceSystem final class:
   * Cannot be extended, maintains business rule integrity
   * private static final Map<String, Object> productCatalog
   * public static boolean processOrder(Object order, Object customer)
   * Static methods for inventory management and order fulfillment

# Assignment Problem 3: 🎓 University Course Registration with Academic Records

## Topics Covered: Access Modifiers, Encapsulation, Immutable Academic History, Constructor Chaining

**Requirements:** Build a university system where academic records are permanent, course enrollment has prerequisites, and student information has appropriate access levels.

## Core Tasks:

1. Immutable AcademicRecord class:
   * final class with private final String studentId, String major, LocalDate enrollmentDate
   * private final Map<String, String> completedCourses - course grades (permanent)
   * private final double cumulativeGPA - calculated from completed courses
   * private final String[] academicHonors - awards and recognitions
   * Only getters, transcript data cannot be modified after recording
   * public final boolean meetsPrerequisites(String courseCode) - academic validation
2. Student class with educational privacy:
   * private final String studentId, AcademicRecord academicRecord

(Protected educational records)

* + private String currentName, email, String phoneNumber (Modifiable contact info)
  + private String currentAddress, emergencyContact (Personal information)
  + Package-private getAcademicStanding() for faculty access
  + public getContactInfo() - safe information for class communications

1. Course-related classes:
   * Course class with private final String courseCode, title, int creditHours, String[] prerequisites
   * Professor class with private final String facultyId, String department, List<String> qualifications
   * Classroom class with private final String roomNumber, int capacity, String[] equipment
   * Each maintains immutable core properties with controlled modification of schedules
2. Constructor chaining for different student types:
   * New freshman (creates fresh academic record)
   * Transfer student (imports existing credits)
   * Graduate student (includes undergraduate transcript)
   * International student (includes visa and language proficiency data)
3. RegistrationSystem class with access control:
   * private final Map<String, Object> enrolledStudents (different student types)
   * public boolean enrollStudent(Object student, Object course) - use instanceof for validation
   * private boolean validatePrerequisites(Object student, Object course)
   * Package-private methods for academic administration
   * Static final academic policies and enrollment rules
4. JavaBean compliance with academic standards:
   * Proper getter/setter naming for modifiable data
   * Immutable academic records with only getters
   * Standard toString() methods for transcripts and reports
   * equals() and hashCode() based on permanent student identifiers

import java.time.LocalDate;

import java.util.\*;

final class AcademicRecord {

private final String studentId;

private final String major;

private final LocalDate enrollmentDate;

private final Map<String, String> completedCourses;

private final double cumulativeGPA;

private final String[] academicHonors;

public AcademicRecord(String studentId, String major, LocalDate enrollmentDate, Map<String, String> completedCourses, double cumulativeGPA, String[] academicHonors) {

this.studentId = studentId;

this.major = major;

this.enrollmentDate = enrollmentDate;

this.completedCourses = new HashMap<>(completedCourses);

this.cumulativeGPA = cumulativeGPA;

this.academicHonors = academicHonors.clone();

}

public String getStudentId() { return studentId; }

public String getMajor() { return major; }

public LocalDate getEnrollmentDate() { return enrollmentDate; }

public Map<String, String> getCompletedCourses() {

return new HashMap<>(completedCourses);

}

public double getCumulativeGPA() { return cumulativeGPA; }

public String[] getAcademicHonors() {

return academicHonors.clone();

}

public final boolean meetsPrerequisites(String courseCode) {

return completedCourses.containsKey(courseCode);

}

public static void main(String[] args) {

Map<String, String> completedCourses = new HashMap<>();

completedCourses.put("CS101", "A");

completedCourses.put("MATH101", "B+");

AcademicRecord record = new AcademicRecord(

"S123",

"Computer Science",

LocalDate.of(2022, 8, 15),

completedCourses,

3.75,

new String[]{"Dean's List"}

);

System.out.println(record.getStudentId());

System.out.println(record.getMajor());

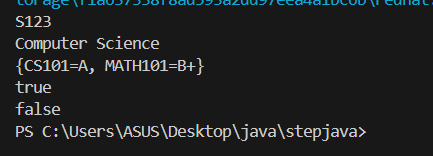
System.out.println(record.getCompletedCourses());

System.out.println(record.meetsPrerequisites("CS101")); // true

System.out.println(record.meetsPrerequisites("PHY101")); // false

}

}



# Assignment Problem 4: 🏦 Banking System with Account Security and Transaction History

## Topics Covered: Complete Integration, final Security Methods, Immutable Transaction Records, Access Control

**Requirements:** Design a comprehensive banking system demonstrating enterprise-level security, immutable financial records, and strict access control for sensitive financial data.

## Core Tasks:

1. Immutable Transaction class:
   * final class with private final String transactionId, LocalDateTime timestamp
   * private final double amount, String transactionType, String description
   * private final String fromAccount, toAccount (Account references)
   * private final Map<String, String> metadata - additional transaction details
   * Only getters with defensive copying
   * public final boolean isValid() - cannot be overridden for financial integrity
2. BankAccount class with financial security:
   * private final String accountNumber, String accountType, LocalDate openDate (Immutable account identity)
   * private double balance, String accountStatus (Controlled financial state)
   * private final String ownerId (Permanent account ownership)
   * private List<Object> transactionHistory (stores different transaction types)
   * Package-private getInternalBalance() for bank operations
   * public getPublicAccountInfo() - safe account details (masked account number, type)
3. Banking staff classes with role-based access:
   * BankTeller class with private final String tellerId, String branch, List<String> permissions
   * AccountManager class with private final String managerId, Set<String> clientAccounts
   * Auditor class with private final String auditorId, String[] accessRights
   * Each role has different access levels to account information and transaction capabilities
4. Constructor chaining for different account types:
   * Basic savings account (minimal features)
   * Checking account with overdraft protection
   * Premium account with investment options
   * Business account with multiple signatories and higher limits
5. BankingSystem final class with security:
   * Cannot be extended to prevent security breaches
   * private static final Map<String, Object> accountRegistry
   * public static boolean processTransaction(Object transaction, Object staff)
   * private static boolean validateStaffPermissions(Object staff, String operation)
   * Final methods for critical operations: authenticateUser(), validateTransaction()
6. Security and validation features:
   * SecurityValidator class with private final String[] securityRules
   * TransactionLimits immutable class with daily/monthly limits
   * Use instanceof for different transaction types (deposit, withdrawal, transfer)
   * Access control matrix determining which staff roles can perform which operations
7. JavaBean compliance with banking standards:
   * All financial data follows banking regulations
   * Immutable transaction records with comprehensive audit trails
   * Validated setters for account status changes
   * Secure toString() methods that don't expose sensitive information
8. Advanced security scenarios:
   * Fraud detection using transaction patterns
   * Account freezing capabilities with proper authorization
   * Multi-factor authentication for high-value transactions
   * Regulatory compliance reporting with appropriate data access controls

import java.time.LocalDateTime;

import java.util.\*;

final class Transaction {

private final String transactionId;

private final LocalDateTime timestamp;

private final double amount;

private final String type;

private final String fromAccount, toAccount;

private final Map<String, String> metadata;

public Transaction(String transactionId, LocalDateTime timestamp, double amount, String type, String fromAccount, String toAccount, Map<String, String> metadata) {

this.transactionId = transactionId;

this.timestamp = timestamp;

this.amount = amount;

this.type = type;

this.fromAccount = fromAccount;

this.toAccount = toAccount;

this.metadata = new HashMap<>(metadata);

}

public final boolean isValid() {

return amount > 0;

}

}

