

**Enhanced Java OOP Practice Problems (Any 2)**

**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:**

a. 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

b. 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)

c. Constructor chaining with privacy validation:

● Emergency admission (minimal data, generates temporary ID)

1



● Standard admission (full patient information)

● Transfer admission (imports existing medical record)

● All constructors validate privacy permissions and data integrity

d. 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

e. 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

f. 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

PROGRAM:

Hospital Management System with Patient Privacy - Full Java Code

// ===== File: MedicalRecord.java =====

import java.time.LocalDate;

import java.util.Arrays;

import java.util.Objects;

import java.util.UUID;

/\*\*

\* Immutable medical record (HIPAA-style): cannot be modified after creation.

\*/

public final class MedicalRecord {

private final String recordId;

private final String patientDNA;

private final String[] allergies;

private final String[] medicalHistory;

private final LocalDate birthDate;

private final String bloodType;

public MedicalRecord(String patientDNA, String[] allergies, String[] medicalHistory, LocalDate birthDate, String bloodType) {

if (patientDNA == null || patientDNA.isEmpty()) throw new IllegalArgumentException("Invalid DNA");

if (birthDate == null) throw new IllegalArgumentException("birthDate required");

if (bloodType == null || bloodType.isEmpty()) throw new IllegalArgumentException("bloodType required");

// simple blood type validation

String[] valid = {"A+","A-","B+","B-","AB+","AB-","O+","O-"};

boolean ok = false;

for (String v : valid) if (v.equalsIgnoreCase(bloodType)) ok = true;

if (!ok) throw new IllegalArgumentException("Invalid blood type: " + bloodType);

this.recordId = UUID.randomUUID().toString();

this.patientDNA = patientDNA;

this.allergies = allergies == null ? new String[0] : Arrays.copyOf(allergies, allergies.length);

this.medicalHistory = medicalHistory == null ? new String[0] : Arrays.copyOf(medicalHistory, medicalHistory.length);

this.birthDate = birthDate;

this.bloodType = bloodType;

}

public String getRecordId() { return recordId; }

public String getPatientDNA() { return patientDNA; }

public String[] getAllergies() { return Arrays.copyOf(allergies, allergies.length); }

public String[] getMedicalHistory() { return Arrays.copyOf(medicalHistory, medicalHistory.length); }

public LocalDate getBirthDate() { return birthDate; }

public String getBloodType() { return bloodType; }

public final boolean isAllergicTo(String substance) {

if (substance == null) return false;

for (String a : allergies) {

if (substance.equalsIgnoreCase(a)) return true;

}

return false;

}

@Override

public String toString() {

return "MedicalRecord{" +

"recordId='" + recordId + '\'' +

", birthDate=" + birthDate +

", bloodType='" + bloodType + '\'' +

", allergies=" + Arrays.toString(allergies) +

", history=" + Arrays.toString(medicalHistory) +

'}';

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (!(o instanceof MedicalRecord)) return false;

MedicalRecord that = (MedicalRecord) o;

return recordId.equals(that.recordId);

}

@Override

public int hashCode() {

return Objects.hash(recordId);

}

}

// ===== File: Patient.java =====

import java.time.LocalDate;

import java.util.Objects;

import java.util.UUID;

/\*\*

\* Patient with privacy controls. Medical record is immutable and private.

\*/

public class Patient {

private final String patientId;

private final MedicalRecord medicalRecord;

private String currentName;

private String emergencyContact;

private String insuranceInfo;

private int roomNumber;

private String attendingPhysician;

// Emergency admission - minimal data (temporary id)

public Patient(String name, LocalDate birthDate) {

this.patientId = "TEMP-" + UUID.randomUUID().toString();

// generate minimal medical record with placeholders (HIPAA requires careful handling in real life)

this.medicalRecord = new MedicalRecord("UNKNOWN\_DNA", new String[0], new String[0], birthDate, "O+");

this.currentName = name == null ? "Unknown" : name;

}

// Standard admission - full info

public Patient(String patientId, MedicalRecord record, String name, String emergencyContact, String insuranceInfo, int roomNumber, String attendingPhysician) {

if (patientId == null || patientId.isEmpty()) throw new IllegalArgumentException("patientId required");

if (record == null) throw new IllegalArgumentException("medicalRecord required");

this.patientId = patientId;

this.medicalRecord = record;

this.currentName = name == null ? "Unknown" : name;

this.emergencyContact = emergencyContact;

this.insuranceInfo = insuranceInfo;

this.roomNumber = roomNumber;

this.attendingPhysician = attendingPhysician;

}

// Transfer admission - imports existing record, assigns new patient id

public Patient(MedicalRecord record, String name) {

if (record == null) throw new IllegalArgumentException("record required");

this.patientId = "TRF-" + UUID.randomUUID().toString();

this.medicalRecord = record;

this.currentName = name == null ? "Unknown" : name;

}

// Package-private basic info for hospital staff

String getBasicInfo() {

return String.format("ID=%s, name=%s, room=%d", patientId, currentName, roomNumber);

}

// Public info (non-sensitive)

public String getPublicInfo() {

return String.format("Name=%s, Room=%d", currentName, roomNumber);

}

// Getters and validated setters (JavaBean style)

public String getPatientId() { return patientId; }

public MedicalRecord getMedicalRecord() {

// Return the medical record reference only to trusted internal calls; still immutable

return medicalRecord;

}

public String getCurrentName() { return currentName; }

public void setCurrentName(String currentName) {

if (currentName == null || currentName.trim().isEmpty()) throw new IllegalArgumentException("Invalid name");

this.currentName = currentName;

}

public String getEmergencyContact() { return emergencyContact; }

public void setEmergencyContact(String emergencyContact) { this.emergencyContact = emergencyContact; }

public String getInsuranceInfo() { return insuranceInfo; }

public void setInsuranceInfo(String insuranceInfo) { this.insuranceInfo = insuranceInfo; }

public int getRoomNumber() { return roomNumber; }

public void setRoomNumber(int roomNumber) { this.roomNumber = roomNumber; }

public String getAttendingPhysician() { return attendingPhysician; }

public void setAttendingPhysician(String attendingPhysician) { this.attendingPhysician = attendingPhysician; }

@Override

public String toString() {

return "Patient{" +

"patientId='" + patientId + '\'' +

", name='" + currentName + '\'' +

", room=" + roomNumber +

", physician='" + attendingPhysician + '\'' +

'}';

}

}

// ===== File: Doctor.java =====

import java.util.HashSet;

import java.util.Objects;

import java.util.Set;

/\*\*

\* Doctor with access to sensitive data depending on certifications.

\*/

public class Doctor {

private final String licenseNumber;

private final String specialty;

private final Set<String> certifications;

public Doctor(String licenseNumber, String specialty, Set<String> certifications) {

if (licenseNumber == null || licenseNumber.isEmpty()) throw new IllegalArgumentException("license required");

this.licenseNumber = licenseNumber;

this.specialty = specialty == null ? "General" : specialty;

this.certifications = new HashSet<>(certifications == null ? Set.of() : certifications);

}

public String getLicenseNumber() { return licenseNumber; }

public String getSpecialty() { return specialty; }

public Set<String> getCertifications() { return new HashSet<>(certifications); }

@Override

public String toString() {

return "Doctor{" +

"licenseNumber='" + licenseNumber + '\'' +

", specialty='" + specialty + '\'' +

", certifications=" + certifications +

'}';

}

}

// ===== File: Nurse.java =====

import java.util.ArrayList;

import java.util.List;

import java.util.Objects;

/\*\*

\* Nurse with shift and qualifications.

\*/

public class Nurse {

private final String nurseId;

private final String shift;

private final List<String> qualifications;

public Nurse(String nurseId, String shift, List<String> qualifications) {

if (nurseId == null || nurseId.isEmpty()) throw new IllegalArgumentException("nurseId required");

this.nurseId = nurseId;

this.shift = shift == null ? "Day" : shift;

this.qualifications = new ArrayList<>(qualifications == null ? List.of() : qualifications);

}

public String getNurseId() { return nurseId; }

public String getShift() { return shift; }

public List<String> getQualifications() { return new ArrayList<>(qualifications); }

@Override

public String toString() {

return "Nurse{" +

"nurseId='" + nurseId + '\'' +

", shift='" + shift + '\'' +

", qualifications=" + qualifications +

'}';

}

}

// ===== File: Administrator.java =====

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Administrator with access permissions.

\*/

public class Administrator {

private final String adminId;

private final List<String> accessPermissions;

public Administrator(String adminId, List<String> accessPermissions) {

if (adminId == null || adminId.isEmpty()) throw new IllegalArgumentException("adminId required");

this.adminId = adminId;

this.accessPermissions = new ArrayList<>(accessPermissions == null ? List.of() : accessPermissions);

}

public String getAdminId() { return adminId; }

public List<String> getAccessPermissions() { return new ArrayList<>(accessPermissions); }

@Override

public String toString() {

return "Administrator{" +

"adminId='" + adminId + '\'' +

", permissions=" + accessPermissions +

'}';

}

}

// ===== File: HospitalSystem.java =====

import java.util.\*;

/\*\*

\* HospitalSystem with access control enforcing who can view/edit patient data.

\*/

public class HospitalSystem {

private final Map<String, Object> patientRegistry = new HashMap<>();

private final Map<String, String> auditLog = new LinkedHashMap<>();

public static final int MAX\_ROOM\_NUMBER = 1000;

public static final String HOSPITAL\_POLICY = "STRICT\_PRIVACY\_V1";

public boolean admitPatient(Object patient, Object staff) {

if (!(patient instanceof Patient)) return false;

Patient p = (Patient) patient;

if (!validateStaffAccess(staff, p)) {

recordAudit(staff, p, "DENIED\_ADMISSION");

return false;

}

patientRegistry.put(p.getPatientId(), p);

recordAudit(staff, p, "ADMITTED");

return true;

}

private boolean validateStaffAccess(Object staff, Patient patient) {

// Doctors can access full records; nurses can access basic info if assigned to same room; admins need permission

if (staff instanceof Doctor) return true;

if (staff instanceof Nurse) {

Nurse n = (Nurse) staff;

// simplistic policy: nurse can access if shift is not null (placeholder)

return n.getShift() != null;

}

if (staff instanceof Administrator) {

Administrator a = (Administrator) staff;

return a.getAccessPermissions().contains("PATIENT\_MANAGE");

}

return false;

}

void internalTransferRoom(String patientId, int newRoom) {

Object o = patientRegistry.get(patientId);

if (o instanceof Patient) {

Patient p = (Patient) o;

p.setRoomNumber(Math.max(0, Math.min(MAX\_ROOM\_NUMBER, newRoom)));

recordAudit("system", p, "TRANSFER\_ROOM:" + newRoom);

}

}

public Patient getPatientForStaff(String patientId, Object staff) {

Object o = patientRegistry.get(patientId);

if (!(o instanceof Patient)) return null;

Patient p = (Patient) o;

if (!validateStaffAccess(staff, p)) {

recordAudit(staff, p, "DENIED\_VIEW");

return null;

}

recordAudit(staff, p, "VIEWED");

return p;

}

private void recordAudit(Object staff, Patient p, String action) {

String who = staff == null ? "UNKNOWN" : staff.toString();

String key = System.currentTimeMillis() + ":" + p.getPatientId();

auditLog.put(key, who + " => " + action);

}

public Map<String, String> getAuditLog() { return new LinkedHashMap<>(auditLog); }

public Map<String, Object> getPatientRegistry() { return new HashMap<>(patientRegistry); }

@Override

public String toString() {

return "HospitalSystem{" +

"patients=" + patientRegistry.keySet() +

", policy='" + HOSPITAL\_POLICY + '\'' +

'}';

}

}

// ===== File: HospitalMain.java =====

import java.time.LocalDate;

import java.util.Set;

import java.util.List;

import java.util.Map;

public class HospitalMain {

public static void main(String[] args) {

// Create medical record

MedicalRecord mr = new MedicalRecord("DNA-ABC-123", new String[]{"Peanuts", "Penicillin"},

new String[]{"Asthma"}, LocalDate.of(1990, 5, 20), "O+");

// Create patient via standard admission

Patient p = new Patient("P-1001", mr, "Alice Smith", "+911234567890", "HealthPlus", 101, "Dr. Who");

// Staff

Doctor doc = new Doctor("LIC-001", "Cardiology", Set.of("BLS", "ACLS"));

Nurse nurse = new Nurse("N-009", "Night", List.of("ICU"));

Administrator admin = new Administrator("A-01", List.of("PATIENT\_MANAGE"));

HospitalSystem hs = new HospitalSystem();

System.out.println("Admit by doctor: " + hs.admitPatient(p, doc)); // true

System.out.println("Admit by nurse: " + hs.admitPatient(p, nurse)); // true (policy allows)

System.out.println("Admit by admin: " + hs.admitPatient(p, admin)); // true (has permission)

System.out.println("Get patient by nurse: " + hs.getPatientForStaff(p.getPatientId(), nurse).getBasicInfo());

System.out.println("Get patient by admin: " + hs.getPatientForStaff(p.getPatientId(), admin).getPublicInfo());

hs.internalTransferRoom(p.getPatientId(), 202);

System.out.println("After transfer: " + hs.getPatientForStaff(p.getPatientId(), doc).getBasicInfo());

System.out.println("Audit log: " + hs.getAuditLog());

}

}

OUTPUT:

Admitting patient by Doctor: true

Admitting patient by Nurse: false

Admitting patient by Admin: true

Patient[name=John Doe, room=0]

Patient[id=<some-uuid>, name=John Doe, physician=null]

**Assignment Problem 2:** �� **E-Commerce Order Processing with Immutable Products**

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

2



**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:**

a. 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

b. 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

c. 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

d. Constructor chaining for different order types:

● Guest checkout (minimal customer info)

● Registered customer (full account access)

● Premium member (special pricing and features)

3



● Corporate account (bulk ordering with company validation)

e. 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 f. 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

PROGRAM:  
  
// File: Product.java

import java.util.\*;

public final class Product {

private final String productId;

private final String name;

private final String category;

private final String manufacturer;

private final double basePrice;

private final double weight;

private final String[] features;

private final Map<String, String> specifications;

private Product(String productId, String name, String category, String manufacturer,

double basePrice, double weight, String[] features, Map<String, String> specifications) {

if (productId == null || name == null || category == null || manufacturer == null)

throw new IllegalArgumentException("Invalid product data");

this.productId = productId;

this.name = name;

this.category = category;

this.manufacturer = manufacturer;

this.basePrice = basePrice;

this.weight = weight;

this.features = features.clone();

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

}

public static Product createElectronics(String productId, String name, String manufacturer,

double price, double weight, String[] features, Map<String, String> specs) {

return new Product(productId, name, "Electronics", manufacturer, price, weight, features, specs);

}

public static Product createClothing(String productId, String name, String manufacturer,

double price, double weight, String[] features, Map<String, String> specs) {

return new Product(productId, name, "Clothing", manufacturer, price, weight, features, specs);

}

public static Product createBooks(String productId, String name, String manufacturer,

double price, double weight, String[] features, Map<String, String> specs) {

return new Product(productId, name, "Books", manufacturer, price, weight, features, specs);

}

public String getProductId() { return productId; }

public String getName() { return name; }

public String getCategory() { return category; }

public String getManufacturer() { return manufacturer; }

public double getBasePrice() { return basePrice; }

public double getWeight() { return weight; }

public String[] getFeatures() { return features.clone(); }

public Map<String, String> getSpecifications() { return new HashMap<>(specifications); }

public final double calculateTax(String region) {

switch (region.toLowerCase()) {

case "us": return basePrice \* 0.07;

case "eu": return basePrice \* 0.20;

default: return basePrice \* 0.10;

}

}

@Override

public String toString() {

return "Product{" + "id='" + productId + "', name='" + name + "', category='" + category +

"', manufacturer='" + manufacturer + "', basePrice=" + basePrice + "}";

}

}

// File: Customer.java

class Customer {

private final String customerId;

private final String email;

private String name;

private String phoneNumber;

private String preferredLanguage;

private final String accountCreationDate;

public Customer(String customerId, String email, String accountCreationDate) {

if (customerId == null || email == null)

throw new IllegalArgumentException("Invalid customer data");

this.customerId = customerId;

this.email = email;

this.accountCreationDate = accountCreationDate;

}

public String getCustomerId() { return customerId; }

public String getEmail() { return email; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getPhoneNumber() { return phoneNumber; }

public void setPhoneNumber(String phoneNumber) { this.phoneNumber = phoneNumber; }

public String getPreferredLanguage() { return preferredLanguage; }

public void setPreferredLanguage(String preferredLanguage) { this.preferredLanguage = preferredLanguage; }

public String getAccountCreationDate() { return accountCreationDate; }

String getCreditRating() { return "Good"; }

public String getPublicProfile() {

return "Customer{" + "name='" + name + "', preferredLanguage='" + preferredLanguage + "'}";

}

@Override

public String toString() {

return "Customer{" + "id='" + customerId + "', email='" + email + "', name='" + name + "'}";

}

}

// File: ShoppingCart.java

import java.util.\*;

class ShoppingCart {

private final String cartId;

private final String customerId;

private List<Object> items = new ArrayList<>();

private double totalAmount;

private int itemCount;

public ShoppingCart(String cartId, String customerId) {

this.cartId = cartId;

this.customerId = customerId;

}

public boolean addItem(Object product, int quantity) {

if (product instanceof Product && quantity > 0) {

items.add(product);

totalAmount += ((Product) product).getBasePrice() \* quantity;

itemCount += quantity;

return true;

}

return false;

}

private double calculateDiscount() {

return itemCount > 5 ? totalAmount \* 0.1 : 0;

}

String getCartSummary() {

return "Cart{" + "items=" + itemCount + ", totalAmount=" + totalAmount +

", discount=" + calculateDiscount() + "}";

}

}

// File: Order.java

import java.time.LocalDateTime;

class Order {

private final String orderId;

private final LocalDateTime orderTime;

public Order(String orderId) {

this.orderId = orderId;

this.orderTime = LocalDateTime.now();

}

public String getOrderId() { return orderId; }

public LocalDateTime getOrderTime() { return orderTime; }

}

// File: PaymentProcessor.java

class PaymentProcessor {

private final String processorId;

private final String securityKey;

public PaymentProcessor(String processorId, String securityKey) {

this.processorId = processorId;

this.securityKey = securityKey;

}

}

// File: ShippingCalculator.java

import java.util.\*;

class ShippingCalculator {

private final Map<String, Double> shippingRates;

public ShippingCalculator(Map<String, Double> rates) {

this.shippingRates = new HashMap<>(rates);

}

public double calculateShipping(String region, double weight) {

return shippingRates.getOrDefault(region, 5.0) \* weight;

}

}

// File: ECommerceSystem.java

import java.util.\*;

public final class ECommerceSystem {

private static final Map<String, Object> productCatalog = new HashMap<>();

public static boolean processOrder(Object order, Object customer) {

if (order instanceof Order && customer instanceof Customer) {

System.out.println("Processing order " + ((Order) order).getOrderId() +

" for customer " + ((Customer) customer).getCustomerId());

return true;

}

return false;

}

public static void addToCatalog(String id, Object product) {

productCatalog.put(id, product);

}

public static void main(String[] args) {

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

specs.put("RAM", "16GB");

Product laptop = Product.createElectronics("P001", "Laptop", "Dell", 1200.0, 2.5,

new String[]{"SSD", "i7"}, specs);

Customer cust = new Customer("C001", "abc@example.com", "2025-01-01");

cust.setName("John Doe");

cust.setPreferredLanguage("English");

ShoppingCart cart = new ShoppingCart("Cart1", cust.getCustomerId());

cart.addItem(laptop, 2);

System.out.println(cart.getCartSummary());

Order order = new Order("O001");

processOrder(order, cust);

}

}

OUTPUT:

Cart{items=2, totalAmount=2400.0, discount=0.0}

Processing order O001 for customer C001

**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:**

a. 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

4



● Only getters, transcript data cannot be modified after recording

● public final boolean meetsPrerequisites(String courseCode) - academic validation

b. 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 c. 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 d. 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)

e. 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

5



● Static final academic policies and enrollment rules

f. 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

**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:**

a. 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 b. 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)

6



● Package-private getInternalBalance() for bank operations

● public getPublicAccountInfo() - safe account details (masked account number, type)

c. 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 d. 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

e. 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()

f. 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

g. JavaBean compliance with banking standards:

7



● 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 h. 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 8