

**Core OOP Principles - Encapsulation - Practice Problem - Any Four**

�� **PRACTICE PROBLEM 1: Access Modifiers - The Four Levels of Security**

PROGRAM:

File 1: AccessModifierDemo.java

package com.company.security;

public class AccessModifierDemo {

// Fields

private int privateField; // only inside this class

String defaultField; // package-private

protected double protectedField; // package + subclasses

public boolean publicField; // everywhere

// Constructor

public AccessModifierDemo(int p, String d, double pr, boolean pub) {

this.privateField = p;

this.defaultField = d;

this.protectedField = pr;

this.publicField = pub;

System.***out***.println("AccessModifierDemo constructor called\n");

}

// Methods

private void privateMethod() {

System.***out***.println("Private method called");

}

void defaultMethod() {

System.***out***.println("Default method called");

}

protected void protectedMethod() {

System.***out***.println("Protected method called");

}

public void publicMethod() {

System.***out***.println("Public method called");

}

// Test internal access (inside same class)

public void testInternalAccess() {

System.***out***.println("\n\nInside AccessModifierDemo for testing internal access:\n");

System.***out***.println("privateField = " + privateField);

System.***out***.println("defaultField = " + defaultField);

System.***out***.println("protectedField = " + protectedField);

System.***out***.println("publicField = " + publicField + "\n");

privateMethod();

defaultMethod();

protectedMethod();

publicMethod();

}

// Main to run the demo

public static void main(String[] args) {

AccessModifierDemo obj = new AccessModifierDemo(10, "Hello", 20.5, true);

// Accessible here (same class file)

System.***out***.println("obj.publicField = " + obj.publicField);

obj.publicMethod();

System.***out***.println("obj.defaultField = " + obj.defaultField);

obj.defaultMethod();

System.***out***.println("obj.protectedField = " + obj.protectedField);

obj.protectedMethod();

System.***out***.println("obj.privateField = " + obj.privateField);

obj.privateMethod();

// privateField / privateMethod are NOT accessible from other classes,

// but they are accessible here inside the class (see testInternalAccess)

obj.testInternalAccess();

// Call the same-package test

SamePackageTest.*testAccess*();

}

}

File 2: SamePackageTest.java

package com.company.security;

public class SamePackageTest {

public static void testAccess() {

AccessModifierDemo obj = new AccessModifierDemo(42, "World", 55.5, false);

System.***out***.println("Inside SamePackageTest:\n");

// public is accessible

System.***out***.println("publicField = " + obj.publicField);

obj.publicMethod();

// default (package-private) is accessible (same package)

System.***out***.println("defaultField = " + obj.defaultField);

obj.defaultMethod();

// protected is accessible (same package)

System.***out***.println("protectedField = " + obj.protectedField);

obj.protectedMethod();

// private NOT accessible here:

// obj.privateMethod(); -> compile error if uncommented

}

}

OUTPUT:

AccessModifierDemo constructor called

obj.publicField = true

Public method called

obj.defaultField = Hello

Default method called

obj.protectedField = 20.5

Protected method called

obj.privateField = 10

Private method called

Inside AccessModifierDemo for testing internal access:

privateField = 10

defaultField = Hello

protectedField = 20.5

publicField = true

Private method called

Default method called

Protected method called

Public method called

AccessModifierDemo constructor called

Inside SamePackageTest:

publicField = false

Public method called

defaultField = World

Default method called

protectedField = 55.5

Protected method called

�� **PRACTICE PROBLEM 2: Cross-Package Visibility Rules**

**Testing access modifiers across different packages**

File 1: PackageTestMain.java

package com.company.main;

import com.company.security.AccessModifierDemo;

public class PackageTestMain {

public static void main(String[] args) {

AccessModifierDemo obj = new AccessModifierDemo(1, "CrossPkg", 2.2, true);

System.***out***.println("=== Cross-Package Access Test ===");

// ❌ privateField not accessible

// System.out.println(obj.privateField);

// ❌ defaultField not accessible (default = package-private)

// System.out.println(obj.defaultField);

// ❌ protectedField not accessible (only accessible via subclass, not plain object)

// System.out.println(obj.protectedField);

// ✅ publicField accessible everywhere

System.***out***.println("publicField = " + obj.publicField);

// ❌ privateMethod() not accessible

// obj.privateMethod();

// ❌ defaultMethod() not accessible

// obj.defaultMethod();

// ❌ protectedMethod() not accessible directly

// obj.protectedMethod();

// ✅ publicMethod accessible everywhere

obj.publicMethod();

}

}

# File 2: ExtendedDemo.java

package com.company.extended;

import com.company.security.AccessModifierDemo;

public class ExtendedDemo extends AccessModifierDemo {

// Constructor that calls parent constructor

public ExtendedDemo(int p, String d, double pr, boolean pub) {

super(p, d, pr, pub);

}

public void testInheritedAccess() {

System.***out***.println("\n=== Subclass (Cross-Package) Access Test ===");

// ❌ privateField not inherited

// System.out.println(privateField);

// ❌ defaultField not accessible (different package, not subclass)

// System.out.println(defaultField);

// ✅ protectedField accessible (inherited in subclass)

System.***out***.println("protectedField = " + protectedField);

// ✅ publicField always accessible

System.***out***.println("publicField = " + publicField);

// ❌ privateMethod not accessible

// privateMethod();

// ❌ defaultMethod not accessible

// defaultMethod();

// ✅ protectedMethod accessible via inheritance

protectedMethod();

// ✅ publicMethod always accessible

publicMethod();

}

// Overriding a protected method from parent

*@Override*

protected void protectedMethod() {

System.***out***.println("Overridden protected method called in ExtendedDemo");

}

public static void main(String[] args) {

ExtendedDemo child = new ExtendedDemo(5, "SubClass", 9.9, false);

child.testInheritedAccess();

// Comparing parent vs child

AccessModifierDemo parent = new AccessModifierDemo(10, "Parent", 11.1, true);

System.***out***.println("\nComparing parent vs child:");

parent.publicMethod();

child.publicMethod();

}

}

OUTPUT:

AccessModifierDemo constructor called

=== Subclass (Cross-Package) Access Test ===

protectedField = 9.9

publicField = false

Overridden protected method called in ExtendedDemo

Public method called

AccessModifierDemo constructor called

Comparing parent vs child:

Public method called

Public method called

�� **PRACTICE PROBLEM 3: Data Hiding Mastery Implementing proper encapsulation with private fields and public methods**

package com.company.encapsulation;

public class SecureBankAccount {

// ================== PRIVATE FIELDS ==================

private final String accountNumber; // read-only

private double balance; // controlled access

private int pin; // write-only

private boolean isLocked; // internal security

private int failedAttempts; // failed PIN counter

// ================== PRIVATE CONSTANTS ==================

private static final int ***MAX\_FAILED\_ATTEMPTS*** = 3;

private static final double ***MIN\_BALANCE*** = 0.0;

// ================== CONSTRUCTOR ==================

public SecureBankAccount(String accountNumber, double initialBalance) {

this.accountNumber = accountNumber;

this.balance = Math.*max*(initialBalance, ***MIN\_BALANCE***);

this.pin = 0; // must be set later

this.isLocked = false;

this.failedAttempts = 0;

System.***out***.println("Account " + accountNumber + " created with balance: " + this.balance);

}

// ================== ACCOUNT INFO METHODS ==================

public String getAccountNumber() {

return accountNumber;

}

public double getBalance() {

if (isLocked) {

System.***out***.println("Account is locked. Cannot display balance.");

return -1;

}

return balance;

}

public boolean isAccountLocked() {

return isLocked;

}

// ================== SECURITY METHODS ==================

public boolean setPin(int oldPin, int newPin) {

if (this.pin == oldPin) {

this.pin = newPin;

System.***out***.println("PIN updated successfully.");

return true;

}

System.***out***.println("Failed to update PIN. Old PIN incorrect.");

return false;

}

public boolean validatePin(int enteredPin) {

if (isLocked) {

System.***out***.println("Account is locked. Access denied.");

return false;

}

if (enteredPin == this.pin) {

resetFailedAttempts();

return true;

} else {

incrementFailedAttempts();

return false;

}

}

public boolean unlockAccount(int correctPin) {

if (this.pin == correctPin) {

resetFailedAttempts();

isLocked = false;

System.***out***.println("Account unlocked successfully.");

return true;

}

System.***out***.println("Unlock failed. Wrong PIN.");

return false;

}

// ================== TRANSACTION METHODS ==================

public void deposit(double amount, int pin) {

if (validatePin(pin)) {

if (amount > 0) {

balance += amount;

System.***out***.println("Deposited: " + amount + ", New Balance: " + balance);

} else {

System.***out***.println("Invalid deposit amount.");

}

}

}

public void withdraw(double amount, int pin) {

if (validatePin(pin)) {

if (amount > 0 && amount <= balance) {

balance -= amount;

System.***out***.println("Withdrew: " + amount + ", Remaining Balance: " + balance);

} else {

System.***out***.println("Withdrawal failed: insufficient funds or invalid amount.");

}

}

}

public void transfer(SecureBankAccount target, double amount, int pin) {

if (validatePin(pin)) {

if (amount > 0 && amount <= balance) {

this.balance -= amount;

target.balance += amount;

System.***out***.println("Transferred " + amount + " to " + target.getAccountNumber());

} else {

System.***out***.println("Transfer failed: insufficient funds or invalid amount.");

}

}

}

// ================== PRIVATE HELPERS ==================

private void lockAccount() {

isLocked = true;

System.***out***.println("Account locked due to too many failed attempts!");

}

private void resetFailedAttempts() {

failedAttempts = 0;

}

private void incrementFailedAttempts() {

failedAttempts++;

System.***out***.println("Failed attempts: " + failedAttempts);

if (failedAttempts >= ***MAX\_FAILED\_ATTEMPTS***) {

lockAccount();

}

}

// ================== TEST MAIN ==================

public static void main(String[] args) {

SecureBankAccount acc1 = new SecureBankAccount("ACC123", 1000);

SecureBankAccount acc2 = new SecureBankAccount("ACC456", 500);

// ❌ Direct access not possible (compile-time errors if you try):

// acc1.balance = 9999; <-- Not allowed

// System.out.println(acc1.pin); <-- Not allowed

// ✅ Proper usage:

acc1.setPin(0, 1234); // First time set

acc2.setPin(0, 4321);

acc1.deposit(500, 1234);

acc1.withdraw(200, 1234);

// ❌ Wrong PIN attempts

acc1.withdraw(100, 9999);

acc1.withdraw(100, 9999);

acc1.withdraw(100, 9999); // This will lock the account

// ❌ Attempting transactions on locked account

acc1.deposit(100, 1234);

// ✅ Unlocking with correct PIN

acc1.unlockAccount(1234);

// ✅ Transfer money

acc1.transfer(acc2, 300, 1234);

System.***out***.println("Final Balances:");

System.***out***.println("Acc1: " + acc1.getBalance());

System.***out***.println("Acc2: " + acc2.getBalance());

}

}

OUTPUT:

Account ACC123 created with balance: 1000.0

Account ACC456 created with balance: 500.0

PIN updated successfully.

PIN updated successfully.

Deposited: 500.0, New Balance: 1500.0

Withdrew: 200.0, Remaining Balance: 1300.0

Failed attempts: 1

Failed attempts: 2

Failed attempts: 3

Account locked due to too many failed attempts!

Account is locked. Access denied.

Account unlocked successfully.

Transferred 300.0 to ACC456

Final Balances:

Acc1: 1000.0

Acc2: 800.0

�� **PRACTICE PROBLEM 4: JavaBean Standards Implementation**

**Creating professional JavaBean-compliant classes**

# **Full Solution: EmployeeBean.java**

package com.company.javabeans;

import java.io.Serializable;

import java.text.NumberFormat;

import java.time.LocalDate;

import java.time.Period;

import java.util.Date;

import java.util.Objects;

/\*\*

\* EmployeeBean class demonstrating JavaBean standards.

\* - Implements Serializable

\* - Has private fields

\* - Provides no-arg + parameterized constructors

\* - Provides standard getters and setters

\* - Provides computed and derived properties

\*/

public class EmployeeBean implements Serializable {

private static final long ***serialVersionUID*** = 1L;

// ============ PRIVATE FIELDS ============

private String employeeId;

private String firstName;

private String lastName;

private double salary;

private String department;

private LocalDate hireDate;

private boolean isActive;

// ============ CONSTRUCTORS ============

// No-arg constructor (JavaBean requirement)

public EmployeeBean() {

}

// Parameterized constructor

public EmployeeBean(String employeeId, String firstName, String lastName,

double salary, String department, LocalDate hireDate, boolean isActive) {

this.employeeId = employeeId;

this.firstName = firstName;

this.lastName = lastName;

setSalary(salary); // use setter for validation

this.department = department;

this.hireDate = hireDate;

this.isActive = isActive;

}

// ============ STANDARD GETTERS ============

public String getEmployeeId() { return employeeId; }

public String getFirstName() { return firstName; }

public String getLastName() { return lastName; }

public double getSalary() { return salary; }

public String getDepartment() { return department; }

public LocalDate getHireDate() { return hireDate; }

public boolean isActive() { return isActive; }

// ============ STANDARD SETTERS ============

public void setEmployeeId(String employeeId) { this.employeeId = employeeId; }

public void setFirstName(String firstName) { this.firstName = firstName; }

public void setLastName(String lastName) { this.lastName = lastName; }

public void setDepartment(String department) { this.department = department; }

public void setHireDate(LocalDate hireDate) { this.hireDate = hireDate; }

public void setActive(boolean active) { isActive = active; }

// salary setter with validation

public void setSalary(double salary) {

if (salary < 0) {

throw new IllegalArgumentException("Salary must be positive.");

}

this.salary = salary;

}

// ============ COMPUTED PROPERTIES ============

public String getFullName() {

return (firstName != null ? firstName : "") + " " +

(lastName != null ? lastName : "");

}

public int getYearsOfService() {

if (hireDate == null) return 0;

return Period.*between*(hireDate, LocalDate.*now*()).getYears();

}

public String getFormattedSalary() {

return NumberFormat.*getCurrencyInstance*().format(salary);

}

// ============ DERIVED PROPERTIES ============

public void setFullName(String fullName) {

if (fullName == null || !fullName.contains(" ")) {

throw new IllegalArgumentException("Full name must include first and last name.");

}

String[] parts = fullName.split(" ", 2);

this.firstName = parts[0];

this.lastName = parts[1];

}

// ============ OVERRIDES ============

*@Override*

public String toString() {

return "EmployeeBean{" +

"employeeId='" + employeeId + '\'' +

", fullName='" + getFullName() + '\'' +

", salary=" + getFormattedSalary() +

", department='" + department + '\'' +

", hireDate=" + hireDate +

", yearsOfService=" + getYearsOfService() +

", isActive=" + isActive +

'}';

}

*@Override*

public boolean equals(Object o) {

if (this == o) return true;

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

EmployeeBean that = (EmployeeBean) o;

return Objects.*equals*(employeeId, that.employeeId);

}

*@Override*

public int hashCode() {

return Objects.*hash*(employeeId);

}

// ============ DEMO MAIN ============

public static void main(String[] args) {

// Using default constructor + setters

EmployeeBean emp1 = new EmployeeBean();

emp1.setEmployeeId("E001");

emp1.setFullName("Alice Johnson");

emp1.setSalary(60000);

emp1.setDepartment("HR");

emp1.setHireDate(LocalDate.*of*(2018, 5, 10));

emp1.setActive(true);

// Using parameterized constructor

EmployeeBean emp2 = new EmployeeBean("E002", "Bob", "Smith",

75000, "Finance", LocalDate.*of*(2020, 3, 15), true);

// Demonstrating getters

System.***out***.println("Employee 1 Name: " + emp1.getFullName());

System.***out***.println("Employee 2 Salary: " + emp2.getFormattedSalary());

// Testing computed properties

System.***out***.println("Years of Service (emp1): " + emp1.getYearsOfService());

// Testing validation

try {

emp1.setSalary(-5000);

} catch (IllegalArgumentException e) {

System.***out***.println("Validation caught: " + e.getMessage());

}

// Show employees

System.***out***.println(emp1);

System.***out***.println(emp2);

// Collections demo

java.util.List<EmployeeBean> employees = java.util.Arrays.*asList*(emp1, emp2);

employees.stream()

.filter(EmployeeBean::isActive)

.sorted((a, b) -> Double.*compare*(b.getSalary(), a.getSalary()))

.forEach(System.***out***::println);

}

}

**Utility Class: JavaBeanProcessor.java**

package com.company.javabeans;

import java.lang.reflect.Method;

/\*\*

\* Utility class to demonstrate JavaBean introspection via reflection.

\*/

public class JavaBeanProcessor {

// Print all properties by calling getters

public static void printAllProperties(EmployeeBean emp) {

try {

Method[] methods = EmployeeBean.class.getMethods();

for (Method m : methods) {

if ((m.getName().startsWith("get") || m.getName().startsWith("is"))

&& m.getParameterCount() == 0) {

Object value = m.invoke(emp);

String property = m.getName().replaceFirst("get|is", "");

System.***out***.println(property + ": " + value);

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

// Copy all properties from one EmployeeBean to another

public static void copyProperties(EmployeeBean source, EmployeeBean target) {

try {

Method[] methods = EmployeeBean.class.getMethods();

for (Method m : methods) {

if (m.getName().startsWith("get") && m.getParameterCount() == 0) {

Object value = m.invoke(source);

String setterName = "set" + m.getName().substring(3);

try {

Method setter = EmployeeBean.class.getMethod(setterName, m.getReturnType());

setter.invoke(target, value);

} catch (NoSuchMethodException ignored) {

// skip if setter doesn't exist (e.g., read-only)

}

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

// Test main

public static void main(String[] args) {

EmployeeBean emp1 = new EmployeeBean("E003", "Charlie", "Brown", 50000,

"IT", java.time.LocalDate.*of*(2019, 8, 20), true);

EmployeeBean emp2 = new EmployeeBean();

System.***out***.println("Properties of emp1:");

*printAllProperties*(emp1);

System.***out***.println("\nCopying properties from emp1 to emp2...");

*copyProperties*(emp1, emp2);

System.***out***.println("emp2 after copy: " + emp2);

}

}

OUTPUT:

Employee 1 Name: Alice Johnson

Employee 2 Salary: ₹75,000.00

Years of Service (emp1): 7

Validation caught: Salary must be positive.

EmployeeBean{employeeId='E001', fullName='Alice Johnson', salary=₹60,000.00, department='HR', hireDate=2018-05-10, yearsOfService=7, isActive=true}

EmployeeBean{employeeId='E002', fullName='Bob Smith', salary=₹75,000.00, department='Finance', hireDate=2020-03-15, yearsOfService=5, isActive=true}

EmployeeBean{employeeId='E002', fullName='Bob Smith', salary=₹75,000.00, department='Finance', hireDate=2020-03-15, yearsOfService=5, isActive=true}

EmployeeBean{employeeId='E001', fullName='Alice Johnson', salary=₹60,000.00, department='HR', hireDate=2018-05-10, yearsOfService=7, isActive=true}

�� **PRACTICE PROBLEM 5: Read-Only and Write-Only Properties**

**Implementing controlled property access patterns**

import java.time.LocalDateTime;

import java.util.UUID;

public class SmartDevice {

*// TODO: Create fields for different property types:*

*// Read-only properties:*

*// - deviceId (String) - set once during construction*

*// - manufacturingDate (LocalDateTime) - set once during construction*

*// - serialNumber (String) - generated automatically*

*// Write-only properties:*

*// - encryptionKey (String) - can be set but never retrieved*

*// - adminPassword (String) - can be changed but never read*

*// Read-write properties:*

6



*// - deviceName (String) - normal getter/setter*

*// - isEnabled (boolean) - normal getter/setter*

*// Computed read-only properties:*

*// - uptime (long) - calculated from startup time*

*// - deviceAge (int) - calculated from manufacturing date*

*// TODO: Private fields for internal state:*

*// - startupTime (LocalDateTime)*

*// - hashedEncryptionKey (int) - stores hash, not actual key*

*// - hashedAdminPassword (int) - stores hash, not actual password*

*// TODO: Create constructor that:*

*// - Sets read-only properties (deviceId, manufacturingDate, serialNumber) // - Records startup time*

*// - Requires initial deviceName*

*// TODO: Implement Read-Only Property Methods:*

*// - getDeviceId() - returns device ID*

*// - getManufacturingDate() - returns manufacturing date*

*// - getSerialNumber() - returns serial number*

*// - getUptime() - calculates time since startup*

*// - getDeviceAge() - calculates age from manufacturing date*

*// - NO setter methods for these properties*

*// TODO: Implement Write-Only Property Methods:*

*// - setEncryptionKey(String key) - stores hash, validates strength*

*// - setAdminPassword(String password) - stores hash, validates complexity // - NO getter methods for these properties*

*// - validateEncryptionKey(String key) - returns boolean without exposing key // - validateAdminPassword(String password) - returns boolean without exposing password*

*// TODO: Implement Read-Write Property Methods:*

*// - getDeviceName() / setDeviceName(String name)*

*// - isEnabled() / setEnabled(boolean enabled)*

*// TODO: Create utility methods:*

*// - getPropertyInfo() - returns map of property types and access levels // - resetDevice() - clears write-only properties, keeps read-only intact*

public static void main(String[] args) {

*// TODO: Create SmartDevice object*

7



*// TODO: Demonstrate read-only properties:*

*// - Show that values are set during construction*

*// - Attempt to find setter methods (should not exist)*

*// - Display computed read-only properties*

*// TODO: Demonstrate write-only properties:*

*// - Set encryption key and admin password*

*// - Attempt to retrieve them directly (should not be possible)*

*// - Use validation methods to verify they're set correctly*

*// TODO: Demonstrate read-write properties:*

*// - Normal getter/setter operations*

*// - Show they can be both read and modified*

*// TODO: Create multiple devices and show property independence // TODO: Test property access patterns with different scenarios*

}

}

�� **PRACTICE PROBLEM 6: Immutable Objects - The Unbreakable Design**

**Creating completely immutable objects with defensive programming**

import java.util.\*;

import java.time.LocalDate;

*// TODO: Make this class immutable by following all immutability rules* public final class ImmutableStudent {

*// TODO: Declare ALL fields as private and final:*

*// - studentId (String)*

*// - name (String)*

*// - birthDate (LocalDate)*

*// - courses (List<String>) - mutable collection that needs defensive copying // - grades (Map<String, Double>) - mutable collection that needs defensive copying // - graduationDate (LocalDate) - can be null initially*

*// TODO: Create constructor that:*

*// - Takes all parameters including collections*

8



*// - Makes defensive copies of all mutable parameters*

*// - Validates all inputs (non-null, non-empty where appropriate)*

*// - Initializes all final fields*

*// TODO: Create getter methods that:*

*// - Return primitive/immutable values directly*

*// - Return defensive copies of mutable objects*

*// - NEVER expose internal mutable state*

*// - getStudentId() - returns String directly*

*// - getName() - returns String directly*

*// - getBirthDate() - returns LocalDate directly (immutable)*

*// - getCourses() - returns new ArrayList copy*

*// - getGrades() - returns new HashMap copy*

*// - getGraduationDate() - returns LocalDate (can be null)*

*// TODO: Create computed property methods:*

*// - getAge() - calculates from birth date*

*// - getGPA() - calculates from grades map*

*// - getTotalCourses() - returns course count*

*// - isGraduated() - returns true if graduation date is set*

*// TODO: Create "modification" methods that return NEW instances:*

*// - withGraduationDate(LocalDate date) - returns new ImmutableStudent with graduation date set*

*// - withAdditionalCourse(String course) - returns new ImmutableStudent with course added // - withGrade(String course, double grade) - returns new ImmutableStudent with grade added/updated*

*// - withName(String newName) - returns new ImmutableStudent with updated name*

*// TODO: Override Object methods properly:*

*// - equals(Object obj) - based on all fields including collections*

*// - hashCode() - consistent with equals, stable across calls*

*// - toString() - includes all relevant information*

*// TODO: Create builder pattern for complex construction:*

public static class Builder {

*// TODO: Create private mutable fields for building*

*// TODO: Create fluent setter methods that return Builder*

*// TODO: Create build() method that returns ImmutableStudent*

*// TODO: Include validation in build() method*

}

9



*// TODO: Create factory methods:*

*// - createBasicStudent(String id, String name, LocalDate birthDate)*

*// - createGraduatedStudent(String id, String name, LocalDate birthDate, LocalDate graduationDate)*

public static void main(String[] args) {

*// TODO: Test immutability extensively:*

*// 1. Create ImmutableStudent with collections*

List<String> courses = new ArrayList<>(Arrays.asList("Math", "Science")); Map<String, Double> grades = new HashMap<>();

grades.put("Math", 95.0);

grades.put("Science", 87.0);

*// TODO: Create student and verify original collections can be modified without affecting student*

*// 2. Test that returned collections are defensive copies:*

*// TODO: Get courses/grades from student and modify them*

*// TODO: Verify original student is unchanged*

*// 3. Test "modification" methods:*

*// TODO: Use withXXX methods to create new instances*

*// TODO: Verify original student is unchanged*

*// TODO: Verify new instances have expected changes*

*// 4. Test Builder pattern:*

*// TODO: Create complex student using builder*

*// TODO: Show fluent interface in action*

*// 5. Test in collections:*

*// TODO: Use ImmutableStudent as HashMap key*

*// TODO: Add to HashSet and verify no duplicates*

*// TODO: Sort collection of students*

*// 6. Test thread safety:*

*// TODO: Access same ImmutableStudent from multiple threads*

*// TODO: Show no synchronization needed*

*// TODO: Compare with mutable equivalent and show benefits:*

*// - Thread safety*

10

*// - Reliable hashing*

**

*// - No defensive copying needed when sharing*

*// - Simplified reasoning about state*

}

}

�� **PRACTICE PROBLEM 7: Final Fields and Advanced Encapsulation**

**Mastering final keyword and creating bulletproof encapsulation**

import java.util.\*;

import java.time.LocalDateTime;

public class SecureConfiguration {

*// TODO: Create different types of final fields:*

*// Compile-time constants (static final):*

*// - APPLICATION\_NAME (String)*

*// - VERSION (String)*

*// - MAX\_CONNECTIONS (int)*

*// - DEFAULT\_TIMEOUT (long)*

*// Instance constants (final, set in constructor):*

*// - configId (String) - unique identifier*

*// - creationTime (LocalDateTime) - when config was created*

*// - allowedOperations (Set<String>) - operations this config permits*

*// Final references to mutable objects (deep encapsulation needed):*

*// - serverSettings (Map<String, String>) - final reference, mutable contents // - userPermissions (List<String>) - final reference, mutable contents // - securityRules (Properties) - final reference, mutable contents*

*// TODO: Create private final fields that require complex initialization: // - encryptedData (byte[]) - set through encryption process*

*// - checksum (long) - calculated from all other fields*

*// TODO: Create constructor that:*

*// - Initializes ALL final fields*

*// - Performs complex initialization (encryption, checksum calculation)* 11



*// - Takes mutable collections and makes defensive copies*

*// - Validates all inputs thoroughly*

*// TODO: Create initialization helper methods (private):*

*// - initializeServerSettings(Map<String, String> settings) - validates and copies // - calculateChecksum() - computes checksum from all data*

*// - encryptSensitiveData(String data) - encrypts and returns byte array // TODO: Create getter methods with different encapsulation strategies:*

*// Simple getters for immutable final fields:*

*// - getConfigId(), getCreationTime(), getApplicationName(), etc.*

*// Defensive copying getters for mutable final references:*

*// - getServerSettings() - returns new HashMap copy*

*// - getUserPermissions() - returns new ArrayList copy*

*// - getSecurityRules() - returns new Properties copy*

*// - getAllowedOperations() - returns new HashSet copy*

*// Computed getters:*

*// - isValid() - verifies checksum matches current state*

*// - getConfigAge() - calculates time since creation*

*// - hasPermission(String permission) - checks user permissions*

*// - getSettingValue(String key, String defaultValue) - safe settings access*

*// TODO: Create controlled modification methods:*

*// - addUserPermission(String permission) - adds if not exists, recalculates checksum // - removeUserPermission(String permission) - removes if exists, recalculates checksum // - updateServerSetting(String key, String value) - updates setting, recalculates checksum // - addSecurityRule(String rule, String value) - adds rule, recalculates checksum*

*// TODO: Create validation methods:*

*// - validateIntegrity() - checks if object state is consistent*

*// - validatePermission(String operation) - checks if operation is allowed // - validateChecksum() - verifies data hasn't been corrupted*

*// TODO: Create snapshot methods:*

*// - createSnapshot() - returns new SecureConfiguration with same values // - exportSettings() - returns read-only map of all settings*

public static void main(String[] args) {

*// TODO: Test final field initialization:*

12



*// 1. Create SecureConfiguration with various data*

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

settings.put("host", "localhost");

settings.put("port", "8080");

List<String> permissions = Arrays.asList("READ", "WRITE", "DELETE"); Set<String> operations = new HashSet<>(Arrays.asList("backup", "restore"));

*// TODO: Create configuration and verify final fields are set*

*// 2. Test immutability of final references:*

*// TODO: Modify original collections and verify config is unchanged*

*// 3. Test defensive copying:*

*// TODO: Get collections from config, modify them, verify config unchanged*

*// 4. Test controlled modifications:*

*// TODO: Use modification methods and verify checksum updates // TODO: Verify integrity after each modification*

*// 5. Test validation:*

*// TODO: Attempt invalid operations and verify proper handling*

*// TODO: Test checksum validation with corrupted data*

*// 6. Test final field advantages:*

*// TODO: Show compile-time errors when trying to reassign final fields // TODO: Demonstrate thread safety of final fields*

*// TODO: Use configuration as key in HashMap (stable hash code)*

*// TODO: Create a ConfigurationManager class that:*

*// - Stores multiple SecureConfiguration objects*

*// - Shows how final fields enable safe sharing*

*// - Demonstrates that final references prevent reassignment but allow mutation* }

*// TODO: Create static factory methods with different initialization strategies: // - createDefaultConfig() - uses default values for all final fields // - createFromProperties(Properties props) - initializes from properties file // - createSecureConfig(String[] sensitiveData) - includes encryption*

}

13



**Key Learning Objectives for Each Problem:**

1. **Problem 1-2**: Understanding access modifier behavior within and across packages 2. **Problem 3**: Implementing proper data hiding with private fields and controlled public access

3. **Problem 4**: Creating JavaBean-compliant classes with standard getter/setter patterns 4. **Problem 5**: Implementing read-only and write-only properties for enhanced security 5. **Problem 6**: Building completely immutable objects with defensive copying 6. **Problem 7**: Using final fields effectively for compile-time and runtime immutability

Each problem builds upon the previous concepts while introducing more sophisticated encapsulation techniques and design patterns.

14