Core OOP Principles - Encapsulation - Practice Problem - Any Four (3;4;6;7 done)

# 🛠 PRACTICE PROBLEM 1: Access Modifiers - The Four Levels of Security

## Understanding private, default, protected, public modifiers

// File: AccessModifierDemo.java

package com.company.security;

public class AccessModifierDemo {

// TODO: Create four different fields with different access modifiers:

// - privateField (int) - only accessible within this class

// - defaultField (String) - accessible within same package

// - protectedField (double) - accessible in package + subclasses

// - publicField (boolean) - accessible everywhere

// TODO: Create four methods with matching access levels:

// - privateMethod() - prints "Private method called"

// - defaultMethod() - prints "Default method called"

// - protectedMethod() - prints "Protected method called"

// - publicMethod() - prints "Public method called"

// TODO: Create a constructor that initializes all fields

// TODO: Create a public method testInternalAccess() that:

// - Accesses and prints all four fields

// - Calls all four methods

// - Demonstrates that private members are accessible within same class

public static void main(String[] args) {

// TODO: Create an AccessModifierDemo object

// TODO: Try to access each field and method

// TODO: Document in comments which ones work and which cause errors

// TODO: Call testInternalAccess() to show internal accessibility

}

}

// TODO: Create a second class in the SAME package:

class SamePackageTest {

public static void testAccess() {

// TODO: Create AccessModifierDemo object

// TODO: Try accessing each field and method

// TODO: Document which access modifiers work within same package

}

}

# 🛠 PRACTICE PROBLEM 2: Cross-Package Visibility Rules

## Testing access modifiers across different packages

// File: com/company/main/PackageTestMain.java

package com.company.main;

// TODO: Import the AccessModifierDemo class from com.company.security

public class PackageTestMain {

public static void main(String[] args) {

// TODO: Create AccessModifierDemo object

// TODO: Attempt to access each field and method

// TODO: Document which access modifiers work across packages

// TODO: Explain why certain accesses fail

}

}

// TODO: Create a subclass in different package:

// File: com/company/extended/ExtendedDemo.

package com.company.extended;

// TODO: Import AccessModifierDemo

// TODO: Create class ExtendedDemo that extends AccessModifierDemo

public class ExtendedDemo extends AccessModifierDemo {

// TODO: Create constructor that calls super constructor

public void testInheritedAccess() {

// TODO: Try accessing inherited fields with different modifiers

// TODO: Try calling inherited methods with different modifiers

// TODO: Document which protected members are accessible

// TODO: Show that private members are NOT inherited

}

// TODO: Override protected method from parent class

public static void main(String[] args) {

// TODO: Test inheritance access rules

// TODO: Create both parent and child objects

// TODO: Compare what each can access

}

}

# 🛠 PRACTICE PROBLEM 3: Data Hiding Mastery

## Implementing proper encapsulation with private fields and public methods

public class SecureBankAccount {

// TODO: Create private fields that should NEVER be accessed directly:

// - accountNumber (String) - read-only after creation

// - balance (double) - only modified through controlled methods

// - pin (int) - write-only for security

// - isLocked (boolean) - internal security state

// - failedAttempts (int) - internal security counter

// TODO: Create private constants:

// - MAX\_FAILED\_ATTEMPTS (int) = 3

// - MIN\_BALANCE (double) = 0.0

// TODO: Create constructor that takes accountNumber and initial balance

// TODO: Initialize pin to 0 (must be set separately)

// TODO: Create PUBLIC methods for controlled access:

// Account Info Methods:

// - getAccountNumber() - returns account number

// - getBalance() - returns current balance (only if not locked)

// - isAccountLocked() - returns lock status

// Security Methods:

// - setPin(int oldPin, int newPin) - changes PIN if old PIN correct

// - validatePin(int enteredPin) - checks PIN, handles failed attempts

// - unlockAccount(int correctPin) - unlocks if PIN correct

// Transaction Methods:

// - deposit(double amount, int pin) - adds money if PIN valid

// - withdraw(double amount, int pin) - removes money if PIN valid and sufficient funds

// - transfer(SecureBankAccount target, double amount, int pin) - transfers between accounts

// TODO: Create private helper methods:

// - lockAccount() - sets isLocked to true

// - resetFailedAttempts() - resets counter to 0

// - incrementFailedAttempts() - increases counter, locks if needed

public static void main(String[] args) {

// TODO: Create two SecureBankAccount objects

// TODO: Try to access private fields directly (should fail)

// TODO: Demonstrate proper usage through public methods:

// - Set PINs for both accounts

// - Make deposits and withdrawals

// - Show security features (account locking)

// - Transfer money between accounts

// TODO: Attempt security breaches:

|  |  |  |
| --- | --- | --- |
|  | *//*  *//*  *//* | * *Wrong PIN multiple times* * *Withdrawing more than balance* * *Operating on locked account* |
| }  } |  |  |

# public class SecureBankAccount {

# private String accountNumber;

# private double balance;

# private int pin;

# private boolean isLocked;

# private int failedAttempts;

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

# public SecureBankAccount(String accountNumber, double initialBalance) {

# this.accountNumber = accountNumber;

# this.balance = initialBalance;

# }

# public String getAccountNumber() { return accountNumber; }

# public double getBalance() { return isLocked ? -1 : balance; }

# public boolean isAccountLocked() { return isLocked; }

# public boolean setPin(int oldPin, int newPin) {

# if (this.pin == oldPin) {

# this.pin = newPin;

# return true;

# }

# return false;

# }

# public boolean validatePin(int enteredPin) {

# if (enteredPin == pin) {

# failedAttempts = 0;

# return true;

# }

# failedAttempts++;

# if (failedAttempts >= MAX\_FAILED\_ATTEMPTS) isLocked = true;

# return false;

# }

# public boolean unlockAccount(int correctPin) {

# if (correctPin == pin) {

# isLocked = false;

# failedAttempts = 0;

# return true;

# }

# return false;

# }

# public void deposit(double amount, int pin) {

# if (validatePin(pin) && !isLocked) balance += amount;

# }

# public void withdraw(double amount, int pin) {

# if (validatePin(pin) && !isLocked && balance >= amount) balance -= amount;

# }

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

# if (validatePin(pin) && !isLocked && balance >= amount) {

# balance -= amount;

# target.balance += amount;

# }

# }

# public static void main(String[] args) {

# SecureBankAccount acc1 = new SecureBankAccount("A1", 1000);

# SecureBankAccount acc2 = new SecureBankAccount("A2", 500);

# acc1.setPin(0, 1234);

# acc1.deposit(500, 1234);

# acc1.withdraw(200, 1234);

# acc1.transfer(acc2, 300, 1234);

# System.out.println(acc1.getBalance());

# System.out.println(acc2.getBalance());

# }

# }

# 

# 🛠 PRACTICE PROBLEM 4: JavaBean Standards Implementation

## Creating professional JavaBean-compliant classes

import java.io.Serializable;

public class EmployeeBean implements Serializable {

// TODO: Create private fields following JavaBean conventions:

// - employeeId (String)

// - firstName (String)

// - lastName (String)

// - salary (double)

// - department (String)

// - hireDate (java.util.Date)

// - isActive (boolean)

// TODO: Create default no-argument constructor (JavaBean requirement)

// TODO: Create parameterized constructor for convenience

// TODO: Generate standard JavaBean getter methods:

// - getEmployeeId(), getFirstName(), getLastName(), etc.

// - Follow naming convention: get + PropertyName

// - For boolean: isActive() instead of getIsActive()

// TODO: Generate standard JavaBean setter methods:

// - setEmployeeId(String id), setFirstName(String name), etc.

// - Follow naming convention: set + PropertyName

// - Include validation where appropriate

// TODO: Create computed properties (getters without corresponding fields):

// - getFullName() - returns firstName + " " + lastName

// - getYearsOfService() - calculates years since hireDate

// - getFormattedSalary() - returns salary with currency formatting

// TODO: Create derived properties with validation:

// - setFullName(String fullName) - splits into firstName/lastName

// - setSalary(double salary) - validates positive amount

// TODO: Override toString() to display all properties

// TODO: Override equals() and hashCode() based on employeeId

public static void main(String[] args) {

// TODO: Create EmployeeBean using default constructor + setters

// TODO: Create EmployeeBean using parameterized constructor

// TODO: Demonstrate all getter methods

// TODO: Test computed properties

// TODO: Test validation in setter methods

// TODO: Show JavaBean in action with collections (sorting, searching)

// TODO: Create an array of EmployeeBeans and demonstrate:

// - Sorting by salary using computed properties

// - Filtering active employees

// - Bulk operations using JavaBean conventions

}

}

// TODO: Create a JavaBean utility class:

class JavaBeanProcessor {

// TODO: Create static method printAllProperties(EmployeeBean emp)

// - Uses reflection to find all getter methods

// - Calls each getter and prints property name and value

// - Demonstrates JavaBean introspection capabilities

// TODO: Create static method copyProperties(EmployeeBean source, EmployeeBean target)

// - Uses reflection to copy all properties from source to target

// - Demonstrates JavaBean framework integration potential

}

import java.io.Serializable;

import java.util.Date;

public class EmployeeBean implements Serializable {

private String employeeId, firstName, lastName, department;

private double salary;

private Date hireDate;

private boolean isActive;

public EmployeeBean() {}

public EmployeeBean(String employeeId, String firstName, String lastName, double salary, String department, Date hireDate, boolean isActive) {

this.employeeId = employeeId;

this.firstName = firstName;

this.lastName = lastName;

this.salary = salary;

this.department = department;

this.hireDate = hireDate;

this.isActive = isActive;

}

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 Date getHireDate() { return hireDate; }

public boolean isActive() { return isActive; }

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 setSalary(double salary) { this.salary = salary; }

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

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

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

public String getFullName() { return firstName + " " + lastName; }

public String toString() {

return employeeId + " " + getFullName() + " " + department + " " + salary;

}

public static void main(String[] args) {

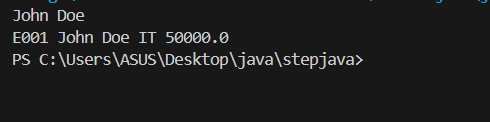
EmployeeBean emp = new EmployeeBean("E001", "John", "Doe", 50000, "IT", new Date(), true);

System.out.println(emp.getFullName());

System.out.println(emp.toString());

}

}



# 🛠 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:

// - 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

// 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

// - 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

}

// 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

// - Reliable hashing

// - No defensive copying needed when sharing

// - Simplified reasoning about state

}

}

import java.time.LocalDate;

import java.util.\*;

public final class ImmutableStudent {

private final String studentId;

private final String name;

private final LocalDate birthDate;

private final List<String> courses;

private final Map<String, Double> grades;

private final LocalDate graduationDate;

public ImmutableStudent(String studentId, String name, LocalDate birthDate, List<String> courses, Map<String, Double> grades, LocalDate graduationDate) {

this.studentId = studentId;

this.name = name;

this.birthDate = birthDate;

this.courses = new ArrayList<>(courses);

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

this.graduationDate = graduationDate;

}

public String getStudentId() { return studentId; }

public String getName() { return name; }

public LocalDate getBirthDate() { return birthDate; }

public List<String> getCourses() { return new ArrayList<>(courses); }

public Map<String, Double> getGrades() { return new HashMap<>(grades); }

public LocalDate getGraduationDate() { return graduationDate; }

public int getAge() { return LocalDate.now().getYear() - birthDate.getYear(); }

public double getGPA() { return grades.values().stream().mapToDouble(Double::doubleValue).average().orElse(0); }

public int getTotalCourses() { return courses.size(); }

public boolean isGraduated() { return graduationDate != null; }

public ImmutableStudent withAdditionalCourse(String course) {

List<String> newCourses = new ArrayList<>(courses);

newCourses.add(course);

return new ImmutableStudent(studentId, name, birthDate, newCourses, grades, graduationDate);

}

public static void main(String[] args) {

List<String> courses = Arrays.asList("Math", "Science");

Map<String, Double> grades = new HashMap<>();

grades.put("Math", 90.0);

grades.put("Science", 85.0);

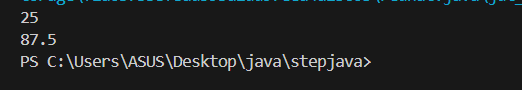
ImmutableStudent student = new ImmutableStudent("S1", "Alice", LocalDate.of(2000, 1, 1), courses, grades, null);

System.out.println(student.getAge());

System.out.println(student.getGPA());

}

}



# 🛠 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)

// - 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:

*// 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

}

import java.time.LocalDateTime;

import java.util.\*;

public class SecureConfiguration {

public static final String APPLICATION\_NAME = "SecureApp";

public static final String VERSION = "1.0";

public static final int MAX\_CONNECTIONS = 10;

public static final long DEFAULT\_TIMEOUT = 3000L;

private final String configId;

private final LocalDateTime creationTime;

private final Set<String> allowedOperations;

private final Map<String, String> serverSettings;

public SecureConfiguration(String configId, Set<String> allowedOperations, Map<String, String> serverSettings) {

this.configId = configId;

this.creationTime = LocalDateTime.now();

this.allowedOperations = new HashSet<>(allowedOperations);

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

}

public String getConfigId() { return configId; }

public LocalDateTime getCreationTime() { return creationTime; }

public Set<String> getAllowedOperations() { return new HashSet<>(allowedOperations); }

public Map<String, String> getServerSettings() { return new HashMap<>(serverSettings); }

public static void main(String[] args) {

Set<String> ops = new HashSet<>(Arrays.asList("read", "write"));

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

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

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

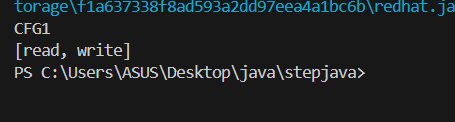
SecureConfiguration config = new SecureConfiguration("CFG1", ops, settings);

System.out.println(config.getConfigId());

System.out.println(config.getAllowedOperations());

}

}



## 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.