WEEK 5

Name:Ramesh Harisabapathi Chettiar

Date of Submission: 17/09/25

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

}

}

Ans.

package com.company.security;

public class AccessModifierDemo {

private int privateField;

String defaultField;

protected double protectedField;

public boolean publicField;

public AccessModifierDemo(int privateField, String defaultField,

double protectedField, boolean publicField) {

this.privateField = privateField;

this.defaultField = defaultField;

this.protectedField = protectedField;

this.publicField = publicField;

}

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");

}

public void testInternalAccess() {

System.out.println("Private field: " + privateField);

System.out.println("Default field: " + defaultField);

System.out.println("Protected field: " + protectedField);

System.out.println("Public field: " + publicField);

privateMethod();

defaultMethod();

protectedMethod();

publicMethod();

}

public static void main(String[] args) {

AccessModifierDemo demo = new AccessModifierDemo(1, "default", 3.14, true);

System.out.println(demo.publicField);

demo.publicMethod();

demo.testInternalAccess();

}

}

class SamePackageTest {

public static void testAccess() {

AccessModifierDemo demo = new AccessModifierDemo(1, "test", 2.5, false);

System.out.println(demo.defaultField);

System.out.println(demo.protectedField);

System.out.println(demo.publicField);

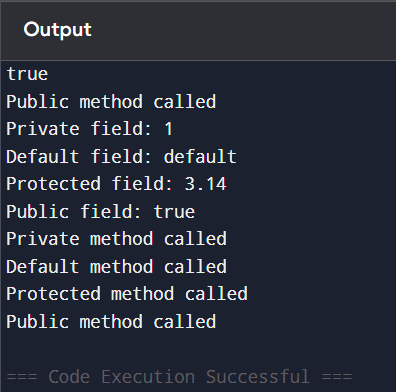
demo.defaultMethod();

demo.protectedMethod();

demo.publicMethod();

}

}



Q2. 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

}

}

Ans. public class SecureBankAccount {

private final String accountNumber;

private double balance;

private int pin;

private boolean isLocked;

private int failedAttempts;

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

private static final double MIN\_BALANCE = 0.0;

public SecureBankAccount(String accountNumber, double initialBalance) {

this.accountNumber = accountNumber;

this.balance = initialBalance;

this.pin = 0;

this.isLocked = false;

this.failedAttempts = 0;

}

public String getAccountNumber() {

return accountNumber;

}

public double getBalance() {

if (isLocked) throw new IllegalStateException("Account locked");

return balance;

}

public boolean isAccountLocked() {

return isLocked;

}

public void setPin(int oldPin, int newPin) {

if (validatePin(oldPin)) {

this.pin = newPin;

resetFailedAttempts();

}

}

public boolean validatePin(int enteredPin) {

if (isLocked) return false;

if (enteredPin == pin) {

resetFailedAttempts();

return true;

} else {

incrementFailedAttempts();

return false;

}

}

public void unlockAccount(int correctPin) {

if (correctPin == pin) {

isLocked = false;

resetFailedAttempts();

}

}

public void deposit(double amount, int pin) {

if (!validatePin(pin)) throw new SecurityException("Invalid PIN");

if (amount <= 0) throw new IllegalArgumentException("Invalid amount");

balance += amount;

}

public void withdraw(double amount, int pin) {

if (!validatePin(pin)) throw new SecurityException("Invalid PIN");

if (amount <= 0) throw new IllegalArgumentException("Invalid amount");

if (balance - amount < MIN\_BALANCE) throw new IllegalArgumentException("Insufficient funds");

balance -= amount;

}

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

withdraw(amount, pin);

target.deposit(amount, target.pin);

}

private void lockAccount() {

isLocked = true;

}

private void resetFailedAttempts() {

failedAttempts = 0;

}

private void incrementFailedAttempts() {

failedAttempts++;

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

lockAccount();

}

}

public static void main(String[] args) {

SecureBankAccount acc1 = new SecureBankAccount("123", 1000);

SecureBankAccount acc2 = new SecureBankAccount("456", 500);

acc1.setPin(0, 1234);

acc2.setPin(0, 5678);

acc1.deposit(200, 1234);

acc1.withdraw(100, 1234);

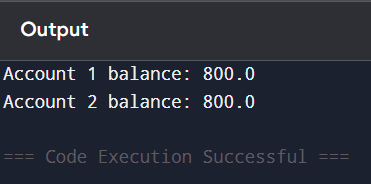
acc1.transfer(acc2, 300, 1234);

System.out.println("Account 1 balance: " + acc1.getBalance());

System.out.println("Account 2 balance: " + acc2.getBalance());

}

}



Q3. 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

}

Ans. import java.io.Serializable;

import java.util.Date;

import java.text.NumberFormat;

public class EmployeeBean implements Serializable {

private String employeeId;

private String firstName;

private String lastName;

private double salary;

private String department;

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) {

if (salary < 0) throw new IllegalArgumentException("Salary cannot be negative");

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 int getYearsOfService() {

if (hireDate == null) return 0;

long diff = new Date().getTime() - hireDate.getTime();

return (int) (diff / (1000L \* 60 \* 60 \* 24 \* 365));

}

public String getFormattedSalary() {

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

}

public void setFullName(String fullName) {

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

this.firstName = parts[0];

this.lastName = parts.length > 1 ? parts[1] : "";

}

@Override

public String toString() {

return "EmployeeBean{" +

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

", firstName='" + firstName + '\'' +

", lastName='" + lastName + '\'' +

", salary=" + salary +

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

", hireDate=" + hireDate +

", isActive=" + isActive +

'}';

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

EmployeeBean that = (EmployeeBean) o;

return employeeId.equals(that.employeeId);

}

@Override

public int hashCode() {

return employeeId.hashCode();

}

public static void main(String[] args) {

EmployeeBean emp1 = new EmployeeBean();

emp1.setEmployeeId("E001");

emp1.setFullName("John Doe");

emp1.setSalary(50000);

emp1.setActive(true);

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

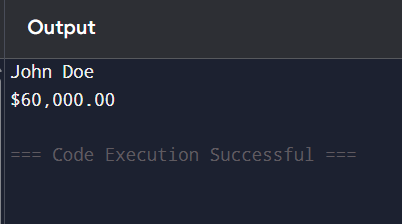
60000, "IT", new Date(), true);

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

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

}

}



Q4. 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.util.\*;

import java.time.LocalDate;

import java.time.Period;

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 = Objects.requireNonNull(studentId);

this.name = Objects.requireNonNull(name);

this.birthDate = Objects.requireNonNull(birthDate);

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

this.grades = new HashMap<>(Objects.requireNonNull(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 Period.between(birthDate, LocalDate.now()).getYears();

}

public double getGPA() {

if (grades.isEmpty()) return 0.0;

double sum = grades.values().stream().mapToDouble(Double::doubleValue).sum();

return sum / grades.size();

}

public int getTotalCourses() {

return courses.size();

}

public boolean isGraduated() {

return graduationDate != null;

}

public ImmutableStudent withGraduationDate(LocalDate date) {

return new ImmutableStudent(studentId, name, birthDate, courses, grades, date);

}

public ImmutableStudent withAdditionalCourse(String course) {

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

newCourses.add(course);

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

}

public ImmutableStudent withGrade(String course, double grade) {

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

newGrades.put(course, grade);

return new ImmutableStudent(studentId, name, birthDate, courses, newGrades, graduationDate);

}

public ImmutableStudent withName(String newName) {

return new ImmutableStudent(studentId, newName, birthDate, courses, grades, graduationDate);

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

ImmutableStudent that = (ImmutableStudent) o;

return studentId.equals(that.studentId);

}

@Override

public int hashCode() {

return studentId.hashCode();

}

@Override

public String toString() {

return "ImmutableStudent{" +

"studentId='" + studentId + '\'' +

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

", birthDate=" + birthDate +

", courses=" + courses +

", grades=" + grades +

", graduationDate=" + graduationDate +

'}';

}

public static class Builder {

private String studentId;

private String name;

private LocalDate birthDate;

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

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

private LocalDate graduationDate;

public Builder studentId(String studentId) {

this.studentId = studentId;

return this;

}

public Builder name(String name) {

this.name = name;

return this;

}

public Builder birthDate(LocalDate birthDate) {

this.birthDate = birthDate;

return this;

}

public Builder courses(List<String> courses) {

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

return this;

}

public Builder grades(Map<String, Double> grades) {

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

return this;

}

public Builder graduationDate(LocalDate graduationDate) {

this.graduationDate = graduationDate;

return this;

}

public ImmutableStudent build() {

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

}

}

public static void main(String[] args) {

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

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

grades.put("Math", 95.0);

grades.put("Science", 87.0);

ImmutableStudent student = new ImmutableStudent("S001", "John",

LocalDate.of(2000, 1, 1),

courses, grades, null);

ImmutableStudent graduatedStudent = student.withGraduationDate(LocalDate.now());

System.out.println("Original: " + student.getGPA());

System.out.println("Graduated: " + graduatedStudent.getGPA());

}

}

