**DAY 5 ANSWERS:**

1. Student with Grade Validation & Configuration

Ensure marks are always valid and immutable once set.

* Create a Student class with private fields: name, rollNumber, and marks.
* Use a constructor to initialize all values and enforce marks to be between 0 and 100; invalid values reset to 0.
* Provide getter methods, but no setter for marks (immutable after object creation).
* Add displayDetails() to print all fields.

In future versions, you might allow updating marks only via a special inputMarks(int newMarks) method that has stricter logic (e.g. cannot reduce marks). Design accordingly.

ANS:

public class Student {

private String name;

private int rollNumber;

private int marks;

public Student(String name, int rollNumber, int marks) {

this.name = name;

this.rollNumber = rollNumber;

if (marks >= 0 && marks <= 100) {

this.marks = marks;

} else {

System.out.println("Invalid marks for " + name + ". Setting to 0.");

this.marks = 0;

}

}

public String getName() {

return name;

}

public int getRollNumber() {

return rollNumber;

}

public int getMarks() {

return marks;

}

public void inputMarks(int newMarks) {

if (newMarks >= 0 && newMarks <= 100) {

if (newMarks >= this.marks) {

this.marks = newMarks;

} else {

System.out.println("Marks cannot be reduced.");

}

} else {

System.out.println("Invalid marks. Must be between 0 and 100.");

}

}

public void displayDetails() {

System.out.println("Name: " + name);

System.out.println("Roll Number: " + rollNumber);

System.out.println("Marks: " + marks);

System.out.println("------------------------");

}

public static void main(String[] args) {

Student s1 = new Student("Ravi", 101, 85);

Student s2 = new Student("Priya", 102, 120);

s1.displayDetails();

s2.displayDetails();

s1.inputMarks(80);

s1.displayDetails();

s1.inputMarks(90);

s1.displayDetails();

}

}

2. Rectangle Enforced Positive Dimensions

Encapsulate validation and provide derived calculations.

* Build a Rectangle class with private width and height.
* Constructor and setters should reject or correct non-positive values (e.g., use default or throw an exception).
* Provide getArea() and getPerimeter() methods.
* Include displayDetails() method.

ANS:

public class Rectangle {

private double width;

private double height;

private static final double DEFAULT\_VALUE = 1.0;

public Rectangle(double width, double height) {

setWidth(width);

setHeight(height);

}

public void setWidth(double width) {

if (width > 0) {

this.width = width;

} else {

System.out.println("Invalid width! Setting to default (" + DEFAULT\_VALUE + ").");

this.width = DEFAULT\_VALUE;

}

}

public void setHeight(double height) {

if (height > 0) {

this.height = height;

} else {

System.out.println("Invalid height! Setting to default (" + DEFAULT\_VALUE + ").");

this.height = DEFAULT\_VALUE;

}

}

public double getWidth() {

return width;

}

public double getHeight() {

return height;

}

public double getArea() {

return width \* height;

}

public double getPerimeter() {

return 2 \* (width + height);

}

public void displayDetails() {

System.out.println("Width: " + width);

System.out.println("Height: " + height);

System.out.println("Area: " + getArea());

System.out.println("Perimeter: " + getPerimeter());

System.out.println("------------------------");

}

public static void main(String[] args) {

Rectangle r1 = new Rectangle(5, 3);

Rectangle r2 = new Rectangle(-2, 4);

r1.displayDetails();

r2.displayDetails();

r1.setHeight(-5);

r1.displayDetails();

}

}

3. Advanced: Bank Account with Deposit/Withdraw Logic

Transaction validation and encapsulation protection.

* Create a BankAccount class with private accountNumber, accountHolder, balance.
* Provide:
  + deposit(double amount) — ignores or rejects negative.
  + withdraw(double amount) — prevents overdraft and returns a boolean success.
  + Getter for balance but no setter.
* Optionally override toString() to display masked account number and details.
* Track transaction history internally using a private list (or inner class for transaction object).
* Expose a method getLastTransaction() but do not expose the full internal list.

ANS:

import java.util.ArrayList;

import java.util.List;

public class BankAccount {

private String accountNumber;

private String accountHolder;

private double balance;

private List<Transaction> transactions;

private class Transaction {

private String type; // "Deposit" or "Withdraw"

private double amount;

private double balanceAfter;

Transaction(String type, double amount, double balanceAfter) {

this.type = type;

this.amount = amount;

this.balanceAfter = balanceAfter;

}

@Override

public String toString() {

return type + " of ₹" + amount + " | Balance after: ₹" + balanceAfter;

}

}

public BankAccount(String accountNumber, String accountHolder, double initialBalance) {

this.accountNumber = accountNumber;

this.accountHolder = accountHolder;

this.balance = Math.max(initialBalance, 0); // No negative starting balance

this.transactions = new ArrayList<>();

transactions.add(new Transaction("Account Opened", this.balance, this.balance));

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

transactions.add(new Transaction("Deposit", amount, balance));

System.out.println("Deposited ₹" + amount + " successfully.");

} else {

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

}

}

public boolean withdraw(double amount) {

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

balance -= amount;

transactions.add(new Transaction("Withdraw", amount, balance));

System.out.println("Withdrew ₹" + amount + " successfully.");

return true;

} else {

System.out.println("Withdrawal failed. Insufficient balance or invalid amount.");

return false;

}

}

public double getBalance() {

return balance;

}

public String getLastTransaction() {

if (transactions.isEmpty()) {

return "No transactions available.";

}

return transactions.get(transactions.size() - 1).toString();

}

@Override

public String toString() {

String maskedAccNum = "\*\*\*\*" + accountNumber.substring(accountNumber.length() - 4);

return "Account Holder: " + accountHolder +

"\nAccount Number: " + maskedAccNum +

"\nBalance: ₹" + balance;

}

public static void main(String[] args) {

BankAccount account = new BankAccount("123456789012", "Ramesh", 5000);

System.out.println(account); // Masked account details

account.deposit(1500);

account.withdraw(2000);

account.withdraw(6000);

System.out.println("\nLast Transaction: " + account.getLastTransaction());

System.out.println("\nFinal Account Details:");

System.out.println(account);

}

}

4. Inner Class Encapsulation: Secure Locker

Encapsulate helper logic inside the class.

* Implement a class Locker with private fields such as lockerId, isLocked, and passcode.
* Use an inner private class SecurityManager to handle passcode verification logic.
* Only expose public methods: lock(), unlock(String code), isLocked().
* Password attempts should not leak verification logic externally—only success/failure.
* Ensure no direct access to passcode or the inner SecurityManager from outside.

ANS:

public class Locker {

private String lockerId;

private boolean isLocked;

private String passcode;

private class SecurityManager {

private boolean verify(String code) {

return passcode.equals(code);

}

}

public Locker(String lockerId, String passcode) {

this.lockerId = lockerId;

this.passcode = passcode;

this.isLocked = true; // default locked

}

public void lock() {

if (!isLocked) {

isLocked = true;

System.out.println("Locker " + lockerId + " is now locked.");

} else {

System.out.println("Locker " + lockerId + " is already locked.");

}

}

public void unlock(String code) {

SecurityManager sm = new SecurityManager();

if (sm.verify(code)) {

if (isLocked) {

isLocked = false;

System.out.println("Locker " + lockerId + " is now unlocked.");

} else {

System.out.println("Locker " + lockerId + " is already unlocked.");

}

} else {

System.out.println("Access Denied: Incorrect passcode.");

}

}

public boolean isLocked() {

return isLocked;

}

public static void main(String[] args) {

Locker myLocker = new Locker("LKR123", "secret123");

System.out.println("Initial Locked State: " + myLocker.isLocked());

myLocker.unlock("wrongcode");

myLocker.unlock("secret123");

myLocker.lock();

myLocker.lock();

}

}

5. Builder Pattern & Encapsulation: Immutable Product

Use Builder design to create immutable class with encapsulation.

* Create an immutable Product class with private final fields such as name, code, price, and optional category.
* Use a static nested Builder inside the Product class. Provide methods like withName(), withPrice(), etc., that apply validation (e.g. non-negative price).
* The outer class should have only getter methods, no setters.

The builder returns a new Product instance only when all validations succeed.

ANS:

public class Product {

private final String name;

private final String code;

private final double price;

private final String category;

private Product(Builder builder) {

this.name = builder.name;

this.code = builder.code;

this.price = builder.price;

this.category = builder.category;

}

public String getName() {

return name;

}

public String getCode() {

return code;

}

public double getPrice() {

return price;

}

public String getCategory() {

return category;

}

public String toString() {

return "Product {" +

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

", code='" + code + '\'' +

", price=₹" + price +

", category='" + (category != null ? category : "N/A") + '\'' +

'}';

}

public static class Builder {

private String name;

private String code;

private double price;

private String category;

public Builder withName(String name) {

if (name == null || name.trim().isEmpty()) {

throw new IllegalArgumentException("Product name cannot be empty.");

}

this.name = name;

return this;

}

public Builder withCode(String code) {

if (code == null || code.trim().isEmpty()) {

throw new IllegalArgumentException("Product code cannot be empty.");

}

this.code = code;

return this;

}

public Builder withPrice(double price) {

if (price < 0) {

throw new IllegalArgumentException("Price cannot be negative.");

}

this.price = price;

return this;

}

public Builder withCategory(String category) {

this.category = category;

return this;

}

public Product build() {

if (name == null || code == null) {

throw new IllegalStateException("Name and Code are required fields.");

}

return new Product(this);

}

}

public static void main(String[] args) {

Product p1 = new Product.Builder()

.withName("Laptop")

.withCode("LP1001")

.withPrice(75000)

.withCategory("Electronics")

.build();

Product p2 = new Product.Builder()

.withName("Book")

.withCode("BK2025")

.withPrice(499)

.build();

System.out.println(p1);

System.out.println(p2);

}

}

**Interface**

1. Reverse CharSequence: Custom BackwardSequence

* Create a class BackwardSequence that implements java.lang.CharSequence.
* Internally store a String and implement all required methods: length(), charAt(), subSequence(), and toString().
* The sequence should be the reverse of the stored string (e.g., new BackwardSequence("hello") yields "olleh").
* Write a main() method to test each method.

ANS:

public class BackwardSequence implements CharSequence {

private final String reversed;

public BackwardSequence(String original) {

if (original == null) {

throw new IllegalArgumentException("Input string cannot be null");

}

this.reversed = new StringBuilder(original).reverse().toString();

}

public int length() {

return reversed.length();

}

public char charAt(int index) {

if (index < 0 || index >= reversed.length()) {

throw new IndexOutOfBoundsException("Index: " + index);

}

return reversed.charAt(index);

}

public CharSequence subSequence(int start, int end) {

if (start < 0 || end > reversed.length() || start > end) {

throw new IndexOutOfBoundsException("Invalid subsequence range");

}

return reversed.substring(start, end);

}

public String toString() {

return reversed;

}

public static void main(String[] args) {

BackwardSequence seq = new BackwardSequence("hello");

System.out.println("Reversed String: " + seq);

System.out.println("Length: " + seq.length());

System.out.println("Character at index 0: " + seq.charAt(0));

System.out.println("Character at index 4: " + seq.charAt(4));

System.out.println("SubSequence(1, 4): " + seq.subSequence(1, 4));

try {

seq.charAt(10);

} catch (IndexOutOfBoundsException e) {

System.out.println("Error: " + e.getMessage());

}

}

}

2. Moveable Shapes Simulation

* Define an interface Movable with methods: moveUp(), moveDown(), moveLeft(), moveRight().
* Implement classes:
  + MovablePoint(x, y, xSpeed, ySpeed) implements Movable
  + MovableCircle(radius, center: MovablePoint)
  + MovableRectangle(topLeft: MovablePoint, bottomRight: MovablePoint) (ensuring both points have same speed)
* Provide toString() to display positions.
* In main(), create a few objects and call move methods to simulate motion.

ANS:

interface Movable {

void moveUp();

void moveDown();

void moveLeft();

void moveRight();

}

// MovablePoint Class

class MovablePoint implements Movable {

int x, y, xSpeed, ySpeed;

public MovablePoint(int x, int y, int xSpeed, int ySpeed) {

this.x = x;

this.y = y;

this.xSpeed = xSpeed;

this.ySpeed = ySpeed;

}

@Override

public void moveUp() {

y += ySpeed;

}

@Override

public void moveDown() {

y -= ySpeed;

}

@Override

public void moveLeft() {

x -= xSpeed;

}

@Override

public void moveRight() {

x += xSpeed;

}

@Override

public String toString() {

return "(" + x + ", " + y + ") Speed(" + xSpeed + ", " + ySpeed + ")";

}

}

class MovableCircle implements Movable {

int radius;

MovablePoint center;

public MovableCircle(int radius, MovablePoint center) {

this.radius = radius;

this.center = center;

}

@Override

public void moveUp() {

center.moveUp();

}

@Override

public void moveDown() {

center.moveDown();

}

@Override

public void moveLeft() {

center.moveLeft();

}

@Override

public void moveRight() {

center.moveRight();

}

@Override

public String toString() {

return "Circle [Center: " + center + ", Radius: " + radius + "]";

}

}

class MovableRectangle implements Movable {

MovablePoint topLeft;

MovablePoint bottomRight;

public MovableRectangle(MovablePoint topLeft, MovablePoint bottomRight) {

if (topLeft.xSpeed != bottomRight.xSpeed || topLeft.ySpeed != bottomRight.ySpeed) {

throw new IllegalArgumentException("Points must have the same speed.");

}

this.topLeft = topLeft;

this.bottomRight = bottomRight;

}

@Override

public void moveUp() {

topLeft.moveUp();

bottomRight.moveUp();

}

@Override

public void moveDown() {

topLeft.moveDown();

bottomRight.moveDown();

}

@Override

public void moveLeft() {

topLeft.moveLeft();

bottomRight.moveLeft();

}

public void moveRight() {

topLeft.moveRight();

bottomRight.moveRight();

}

public String toString() {

return "Rectangle [TopLeft: " + topLeft + ", BottomRight: " + bottomRight + "]";

}

}

public class MovableShapesTest {

public static void main(String[] args) {

MovablePoint p1 = new MovablePoint(0, 0, 2, 3);

System.out.println("Point: " + p1);

p1.moveUp();

p1.moveRight();

System.out.println("Point after moving: " + p1);

MovableCircle c1 = new MovableCircle(5, new MovablePoint(10, 10, 1, 1));

System.out.println("\n" + c1);

c1.moveDown();

c1.moveLeft();

System.out.println("Circle after moving: " + c1);

MovableRectangle r1 = new MovableRectangle(

new MovablePoint(0, 10, 2, 2),

new MovablePoint(5, 0, 2, 2)

);

System.out.println("\n" + r1);

r1.moveRight();

r1.moveDown();

System.out.println("Rectangle after moving: " + r1);

}

}

3. Contract Programming: Printer Switch

* Declare an interface Printer with method void print(String document).
* Implement two classes: LaserPrinter and InkjetPrinter, each providing unique behavior.
* In the client code, declare Printer p;, switch implementations at runtime, and test printing.

ANS:

interface Printer {

void print(String document);

}

class LaserPrinter implements Printer {

@Override

public void print(String document) {

System.out.println("[Laser Printer] Printing with high speed and sharp text: " + document);

}

}

class InkjetPrinter implements Printer {

@Override

public void print(String document) {

System.out.println("[Inkjet Printer] Printing with rich colors and high quality: " + document);

}

}

public class PrinterSwitchTest {

public static void main(String[] args) {

Printer p;

p = new LaserPrinter();

p.print("Monthly Report - Laser Mode");

p = new InkjetPrinter();

p.print("Photo Album - Inkjet Mode");

}

}

4. Extended Interface Hierarchy

* Define interface BaseVehicle with method void start().
* Define interface AdvancedVehicle that extends BaseVehicle, adding method void stop() and boolean refuel(int amount).
* Implement Car to satisfy both interfaces; include a constructor initializing fuel level.
* In Main, manipulate the object via both interface types.

ANS:

interface BaseVehicle {

void start();

}

interface AdvancedVehicle extends BaseVehicle {

void stop();

boolean refuel(int amount);

}

class Car implements AdvancedVehicle {

private int fuelLevel;

public Car(int initialFuel) {

if (initialFuel < 0) {

throw new IllegalArgumentException("Fuel level cannot be negative.");

}

this.fuelLevel = initialFuel;

}

@Override

public void start() {

if (fuelLevel > 0) {

System.out.println("Car started. Vroom!");

} else {

System.out.println("Cannot start. No fuel.");

}

}

@Override

public void stop() {

System.out.println("Car stopped.");

}

@Override

public boolean refuel(int amount) {

if (amount > 0) {

fuelLevel += amount;

System.out.println("Refueled " + amount + " liters. Current fuel: " + fuelLevel + " liters.");

return true;

} else {

System.out.println("Invalid fuel amount. Must be positive.");

return false;

}

}

@Override

public String toString() {

return "Car {Fuel Level: " + fuelLevel + " liters}";

}

}

public class VehicleTest {

public static void main(String[] args) {

BaseVehicle bv = new Car(5);

bv.start();

AdvancedVehicle av = (AdvancedVehicle) bv;

av.stop();

av.refuel(10);

av.start();

av.refuel(-5);

System.out.println(av);

}

}

5. Nested Interface for Callback Handling

* Create a class TimeServer which declares a public static nested interface named Client with void updateTime(LocalDateTime now).
* The server class should have method registerClient(Client client) and notifyClients() to pass current time.
* Implement at least two classes implementing Client, registering them, and simulate notifications.

ANS:

import java.time.LocalDateTime;

import java.util.ArrayList;

import java.util.List;

class TimeServer {

public static interface Client {

void updateTime(LocalDateTime now);

}

private List<Client> clients = new ArrayList<>();

public void registerClient(Client client) {

if (client != null) {

clients.add(client);

System.out.println("Client registered: " + client.getClass().getSimpleName());

}

}

public void notifyClients() {

LocalDateTime now = LocalDateTime.now();

System.out.println("\n[Server] Sending current time to all clients...");

for (Client client : clients) {

client.updateTime(now);

}

}

}

class DigitalClock implements TimeServer.Client {

@Override

public void updateTime(LocalDateTime now) {

System.out.println("[DigitalClock] Current Time: " + now);

}

}

class TimeLogger implements TimeServer.Client {

@Override

public void updateTime(LocalDateTime now) {

System.out.println("[TimeLogger] Time logged: " + now);

}

}

public class TimeNotificationDemo {

public static void main(String[] args) {

TimeServer server = new TimeServer();

TimeServer.Client clock = new DigitalClock();

TimeServer.Client logger = new TimeLogger();

server.registerClient(clock);

server.registerClient(logger);

server.notifyClients();

try { Thread.sleep(2000); } catch (InterruptedException e) { }

server.notifyClients();

}

}

6. Default and Static Methods in Interfaces

* Declare interface Polygon with:
  + double getArea()
  + default method default double getPerimeter(int... sides) that computes sum of sides
  + a static helper static String shapeInfo() returning a description string
* Implement classes Rectangle and Triangle, providing appropriate getArea().
* In Main, call getPerimeter(...) and Polygon.shapeInfo().

ANS:

interface Polygon {

double getArea();

default double getPerimeter(int... sides) {

double sum = 0;

for (int side : sides) {

sum += side;

}

return sum;

}

static String shapeInfo() {

return "A polygon is a 2D shape with straight sides.";

}

}

class Rectangle implements Polygon {

private double length, width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

@Override

public double getArea() {

return length \* width;

}

}

class Triangle implements Polygon {

private double base, height;

public Triangle(double base, double height) {

this.base = base;

this.height = height;

}

@Override

public double getArea() {

return 0.5 \* base \* height;

}

}

public class PolygonDemo {

public static void main(String[] args) {

Polygon rect = new Rectangle(5, 3);

Polygon tri = new Triangle(4, 6);

System.out.println("Rectangle Area: " + rect.getArea());

System.out.println("Triangle Area: " + tri.getArea());

System.out.println("Rectangle Perimeter: " + rect.getPerimeter(5, 3, 5, 3));

System.out.println("Triangle Perimeter: " + tri.getPerimeter(3, 4, 5));

System.out.println("Shape Info: " + Polygon.shapeInfo());

}

}

**Lambda expressions**

1. Sum of Two Integers

ANS:

@FunctionalInterface

interface Sum {

int add(int a, int b);

}

public class LambdaSumDemo {

public static void main(String[] args) {

Sum sum = (a, b) -> a + b;

int result = sum.add(10, 20);

System.out.println("Sum: " + result);

}

}

1. Define a functional interface SumCalculator { int sum(int a, int b); } and a lambda expression to sum two integers.

ANS:

@FunctionalInterface

interface SumCalculator {

int sum(int a, int b);

}

public class LambdaSumCalculatorDemo {

public static void main(String[] args) {

SumCalculator calculator = (a, b) -> a + b;

int result = calculator.sum(15, 25);

System.out.println("Sum: " + result);

}

}

1. Check If a String Is Empty

Create a lambda (via a functional interface like Predicate<String>) that returns true if a given string is empty.  
Predicate<String> isEmpty = s -> s.isEmpty();

ANS:

import java.util.function.Predicate;

public class PredicateIsEmptyDemo {

public static void main(String[] args) {

Predicate<String> isEmpty = s -> s != null && s.isEmpty();

System.out.println(isEmpty.test("")); // true

System.out.println(isEmpty.test("Hello")); // false

System.out.println(isEmpty.test(null)); // false (null-safe check)

}

}

1. Filter Even or Odd Numbers

ANS:

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

import java.util.stream.Collectors;

public class FilterEvenOddDemo {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(10, 15, 22, 33, 40, 51, 60);

Predicate<Integer> isEven = n -> n % 2 == 0;

Predicate<Integer> isOdd = n -> n % 2 != 0;

List<Integer> evenNumbers = numbers.stream()

.filter(isEven)

.collect(Collectors.toList());

List<Integer> oddNumbers = numbers.stream()

.filter(isOdd)

.collect(Collectors.toList());

System.out.println("Even Numbers: " + evenNumbers);

System.out.println("Odd Numbers: " + oddNumbers);

}

}