Day – 5

**Encapsulation**

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.

Code:

public class Student {

private String name;

private int rollNumber;

private final int marks; // immutable after construction

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

this.name = name;

this.rollNumber = rollNumber;

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

this.marks = marks;

} else {

this.marks = 0; // reset invalid marks

}

}

public String getName() { return name; }

public int getRollNumber() { return rollNumber; }

public int getMarks() { return marks; }

// In future: special method for updating marks

/\*

public void inputMarks(int newMarks) {

// e.g., only allow increasing marks, not reducing:

if (newMarks > this.marks && newMarks <= 100) {

// update logic here, but currently not allowed

}

}

\*/

public void displayDetails() {

System.out.printf("Name: %s, Roll Number: %d, Marks: %d%n", name, rollNumber, marks);

}

}

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.

Code:

public class Rectangle {

private double width;

private double height;

public Rectangle(double width, double height) {

this.width = (width > 0) ? width : 1.0; // default to 1.0

this.height = (height > 0) ? height : 1.0;

}

public void setWidth(double width) {

if (width > 0) this.width = width;

// Optionally throw exception or assign default if invalid

}

public void setHeight(double height) {

if (height > 0) this.height = height;

}

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.printf("Width: %.2f, Height: %.2f, Area: %.2f, Perimeter: %.2f%n",

width, height, getArea(), getPerimeter());

}

}

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.

Code:

import java.util.ArrayList;   
import java.util.List;  
public class BankAccount {   
private String accountNumber;   
private String accountHolder;   
private double balance;  
private final List transactions = new ArrayList<>();  
private static class Transaction {  
 enum Type { DEPOSIT, WITHDRAW }  
 private final Type type;  
 private final double amount;  
 private final long time;  
 Transaction(Type type, double amount) {  
 this.type = type;  
 this.amount = amount;  
 this.time = System.currentTimeMillis();  
 }  
 public String toString() {  
 return String.format("%s %.2f @%d", type, amount, time);  
 }  
}  
public BankAccount(String accountNumber, String accountHolder, double initialBalance) {  
 this.accountNumber = accountNumber;  
 this.accountHolder = accountHolder;  
 this.balance = Math.max(0, initialBalance); // no negative opening balance  
}  
public boolean deposit(double amount) {  
 if (amount > 0) {  
 balance += amount;  
 transactions.add(new Transaction(Transaction.Type.DEPOSIT, amount));  
 return true;  
 }  
 return false;  
}  
  
public boolean withdraw(double amount) {  
 if (amount > 0 && balance >= amount) {  
 balance -= amount;  
 transactions.add(new Transaction(Transaction.Type.WITHDRAW, amount));  
 return true;  
 }  
 return false;  
}  
  
public double getBalance() { return balance; }  
  
public Transaction getLastTransaction() {  
 if (transactions.isEmpty())  
 return null;  
 return transactions.get(transactions.size() - 1);  
}  
  
@Override  
public String toString() {  
 String masked = "\*\*\*\*" + accountNumber.substring(Math.max(0, accountNumber.length() - 4));  
 return String.format("Account[%s] Holder: %s, Balance: %.2f", masked, accountHolder, balance);  
}

}

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.

Code:

public class Locker {

private final String lockerId;

private boolean isLocked;

private final String passcode;

private final SecurityManager security;

// Inner helper class (not accessible externally)

private class SecurityManager {

boolean verifyPasscode(String code) {

return passcode.equals(code);

}

}

public Locker(String lockerId, String passcode) {

this.lockerId = lockerId;

this.passcode = passcode;

this.isLocked = true; // Default locked

this.security = new SecurityManager();

}

public void lock() {

isLocked = true;

}

public boolean unlock(String code) {

if (security.verifyPasscode(code)) {

isLocked = false;

return true;

}

return false;

}

public boolean isLocked() {

return isLocked;

}

}

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.

Code:

public final 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 static class Builder {

private String name;

private String code;

private double price;

private String category;

public Builder withName(String name) {

this.name = name;

return this;

}

public Builder withCode(String code) {

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 || price < 0)

throw new IllegalStateException("Missing required fields or invalid price");

return new Product(this);

}

}

}

**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.

Code:

public class BackwardSequence implements CharSequence {

private final String reversed;

public BackwardSequence(String original) {

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

}

@Override

public int length() {

return reversed.length();

}

@Override

public char charAt(int index) {

return reversed.charAt(index);

}

@Override

public CharSequence subSequence(int start, int end) {

return reversed.subSequence(start, end);

}

@Override

public String toString() {

return reversed;

}

public static void main(String[] args) {

BackwardSequence seq = new BackwardSequence("hello");

System.out.println("Sequence: " + seq);

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

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

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

}

}

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.

Code:

interface Movable {

void moveUp();

void moveDown();

void moveLeft();

void moveRight();

}

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;

}

public void moveUp() { y += ySpeed; }

public void moveDown() { y -= ySpeed; }

public void moveLeft() { x -= xSpeed; }

public void moveRight() { x += xSpeed; }

public String toString() {

return String.format("Point(%d, %d)", x, y);

}

}

class MovableCircle implements Movable {

int radius;

MovablePoint center;

public MovableCircle(int radius, MovablePoint center) {

this.radius = radius;

this.center = center;

}

public void moveUp() { center.moveUp(); }

public void moveDown() { center.moveDown(); }

public void moveLeft() { center.moveLeft(); }

public void moveRight() { center.moveRight(); }

public String toString() {

return String.format("Circle[radius=%d, center=%s]", radius, center);

}

}

class MovableRectangle implements Movable {

MovablePoint topLeft, bottomRight;

public MovableRectangle(MovablePoint topLeft, MovablePoint bottomRight) {

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

throw new IllegalArgumentException("Both points must have same speed");

}

this.topLeft = topLeft;

this.bottomRight = bottomRight;

}

public void moveUp() { topLeft.moveUp(); bottomRight.moveUp(); }

public void moveDown() { topLeft.moveDown(); bottomRight.moveDown(); }

public void moveLeft() { topLeft.moveLeft(); bottomRight.moveLeft(); }

public void moveRight() { topLeft.moveRight(); bottomRight.moveRight(); }

public String toString() {

return String.format("Rectangle[TopLeft=%s, BottomRight=%s]", topLeft, bottomRight);

}

}

class MovableTest {

public static void main(String[] args) {

MovablePoint p1 = new MovablePoint(0, 0, 1, 1);

MovableCircle c = new MovableCircle(5, p1);

c.moveRight();

c.moveUp();

System.out.println(c);

MovableRectangle rect = new MovableRectangle(

new MovablePoint(0, 5, 2, 2),

new MovablePoint(10, 0, 2, 2)

);

rect.moveDown();

rect.moveLeft();

System.out.println(rect);

}

}

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.

Code:

interface Printer {

void print(String document);

}

class LaserPrinter implements Printer {

@Override

public void print(String document) {

System.out.println("Laser printing: " + document);

}

}

class InkjetPrinter implements Printer {

@Override

public void print(String document) {

System.out.println("Inkjet printing: " + document);

}

}

public class PrinterSwitch {

public static void main(String[] args) {

Printer p = new LaserPrinter();

p.print("Test Document 1");

p = new InkjetPrinter();

p.print("Test Document 2");

}

}

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.

Code:

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) {

this.fuelLevel = initialFuel >= 0 ? initialFuel : 0;

}

@Override

public void start() {

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

}

@Override

public void stop() {

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

}

@Override

public boolean refuel(int amount) {

if (amount <= 0) return false;

fuelLevel += amount;

return true;

}

public int getFuelLevel() {

return fuelLevel;

}

}

public class VehicleHierarchy {

public static void main(String[] args) {

BaseVehicle base = new Car(50);

AdvancedVehicle advanced = (Car)base;

base.start();

advanced.refuel(20);

advanced.stop();

System.out.println("Fuel level: " + ((Car)advanced).getFuelLevel());

}

}

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.

Code:

import java.time.LocalDateTime;

import java.util.\*;

class TimeServer {

public static interface Client {

void updateTime(LocalDateTime now);

}

private final List<Client> clients;

public TimeServer() {

this.clients = new ArrayList<>();

}

public void registerClient(Client client) {

if (client != null) {

clients.add(client);

}

}

public void notifyClients() {

LocalDateTime now = LocalDateTime.now();

for (Client client : clients) {

client.updateTime(now);

}

}

}

class DigitalClock implements TimeServer.Client {

@Override

public void updateTime(LocalDateTime now) {

System.out.println("Digital Clock updated: " + now);

}

}

class AnalogClock implements TimeServer.Client {

@Override

public void updateTime(LocalDateTime now) {

System.out.println("Analog Clock updated: " + now);

}

}

public class TimeServerDemo {

public static void main(String[] args) {

TimeServer server = new TimeServer();

server.registerClient(new DigitalClock());

server.registerClient(new AnalogClock());

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().

Code:

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 closed shape with straight sides";

}

}

class Rect implements Polygon {

private final double width;

private final double height;

public Rect(double width, double height) {

this.width = width > 0 ? width : 1.0;

this.height = height > 0 ? height : 1.0;

}

@Override

public double getArea() {

return width \* height;

}

}

class Triangle implements Polygon {

private final double base;

private final double height;

public Triangle(double base, double height) {

this.base = base > 0 ? base : 1.0;

this.height = height > 0 ? height : 1.0;

}

@Override

public double getArea() {

return 0.5 \* base \* height;

}

}

public class PolygonDemo {

public static void main(String[] args) {

Polygon rect = new Rect(4, 5);

Polygon tri = new Triangle(3, 6);

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

System.out.println("Rectangle Perimeter: " + rect.getPerimeter(4, 5, 4, 5));

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

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

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

}

}

**Lambda expressions**

1. Sum of Two Integers

Code:

import java.util.function.BinaryOperator;

public class SumLambda {

public static void main(String[] args) {

// Lambda for Sum of Two Integers

BinaryOperator<Integer> sum = (a, b) -> a + b;

// Example usage

int num1 = 10, num2 = 20;

System.out.println("Sum: " + sum.apply(num1, num2)); // Output: Sum: 30

}

}

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

Code:

interface SumCalculator {

int sum(int a, int b);

}

public class SumCalculatorDemo {

public static void main(String[] args) {

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

System.out.println("Sum of 5 and 3: " + adder.sum(5, 3));

System.out.println("Sum of -2 and 7: " + adder.sum(-2, 7));

System.out.println("Sum of 0 and 0: " + adder.sum(0, 0));

}

}

3. 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();

Code:

import java.util.function.Predicate;

public class StringEmptyCheck {

public static void main(String[] args) {

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

System.out.println("Is '' empty? " + isEmpty.test(""));

System.out.println("Is 'hello' empty? " + isEmpty.test("hello"));

System.out.println("Is ' ' empty? " + isEmpty.test(" "));

}

}

4. Filter Even or Odd Numbers

Code:

import java.util.\*;

import java.util.function.Predicate;

import java.util.stream.Collectors;

public class NumberFilter {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

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("Original numbers: " + numbers);

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

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

}

}

5.Convert Strings to Uppercase/Lowercase

Code:

import java.util.function.Function;

public class StringCaseConverter {

public static void main(String[] args) {

Function<String, String> toUpper = s -> s.toUpperCase();

Function<String, String> toLower = s -> s.toLowerCase();

String test = "HeLLo WoRLd";

System.out.println("Original: " + test);

System.out.println("To Uppercase: " + toUpper.apply(test));

System.out.println("To Lowercase: " + toLower.apply(test));

}

}

6.Sort Strings by Length or Alphabetically

Code:

import java.util.\*;

import java.util.stream.Collectors;

public class StringSort {

public static void main(String[] args) {

List<String> strings = Arrays.asList("apple", "b", "banana", "cherry", "date");

Comparator<String> byLength = (s1, s2) -> Integer.compare(s1.length(), s2.length());

Comparator<String> byAlphabet = String::compareTo;

List<String> sortedByLength = strings.stream()

.sorted(byLength)

.collect(Collectors.toList());

List<String> sortedByAlphabet = strings.stream()

.sorted(byAlphabet)

.collect(Collectors.toList());

System.out.println("Original strings: " + strings);

System.out.println("Sorted by length: " + sortedByLength);

System.out.println("Sorted alphabetically: " + sortedByAlphabet);

}

}

7. Aggregate Operations (Sum, Max, Average) on Double Arrays

Code:

import java.util.Arrays;

import java.util.function.ToDoubleFunction;

public class DoubleArray {

public static void main(String[] args) {

double[] numbers = {1.5, 2.7, 3.2, 4.9, 5.1};

ToDoubleFunction<double[]> sum = arr -> Arrays.stream(arr).sum();

ToDoubleFunction<double[]> max = arr -> Arrays.stream(arr).max().orElse(Double.NaN);

ToDoubleFunction<double[]> min = arr -> Arrays.stream(arr).min().orElse(Double.NaN);

ToDoubleFunction<double[]> average = arr -> Arrays.stream(arr).average().orElse(Double.NaN);

System.out.println("Array: " + Arrays.toString(numbers));

System.out.println("Sum: " + sum.applyAsDouble(numbers));

System.out.println("Max: " + max.applyAsDouble(numbers));

System.out.println("Min: " + min.applyAsDouble(numbers));

System.out.println("Average: " + average.applyAsDouble(numbers));

}

}

8.Create similar lambdas for max/min.

Code:

import java.util.function.BinaryOperator;

public class MaxMinLambda {

public static void main(String[] args) {

// Lambda max of two integers

BinaryOperator<Integer> max = (a, b) -> a > b ? a : b;

// Lambda Min of two integers

BinaryOperator<Integer> min = (a, b) -> a < b ? a : b;

int num1 = 10, num2 = 20;

System.out.println("Max: " + max.apply(num1, num2)); // Output: Max: 20

System.out.println("Min: " + min.apply(num1, num2)); // Output: Min: 10

}

}

9. Calculate Factorial

Code:

import java.util.function.LongUnaryOperator;

public class calculateFactorial {

public static void main(String[] args) {

LongUnaryOperator factorial = n -> {

if (n < 0) return 0;

if (n == 0 || n == 1) return 1;

long result = 1;

for (long i = 2; i <= n; i++) {

result \*= i;

}

return result;

};

System.out.println("Factorial of 0: " + factorial.applyAsLong(0));

System.out.println("Factorial of 5: " + factorial.applyAsLong(5));

System.out.println("Factorial of 10: " + factorial.applyAsLong(10));

System.out.println("Factorial of -1: " + factorial.applyAsLong(-1));

}

}