**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.

Program:

class Student {

// Private fields for encapsulation

private String name;

private int rollNumber;

private int marks;

// Constructor with marks validation

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 entered! Setting marks to 0.");

this.marks = 0;

}

}

// Getter methods

public String getName() {

return name;

}

public int getRollNumber() {

return rollNumber;

}

public int getMarks() {

return marks;

}

// Display method

public void displayDetails() {

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

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

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

}

// Future feature: method to input marks with stricter logic

public void inputMarks(int newMarks) {

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

if (newMarks > this.marks) {

this.marks = newMarks;

System.out.println("Marks updated successfully!");

} else {

System.out.println("New marks must be greater than current marks!");

}

} else {

System.out.println("Invalid marks! Please enter between 0 and 100.");

}

}

}

public class Main {

public static void main(String[] args) {

// Creating student object with valid marks

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

s1.displayDetails();

System.out.println("\nTrying to update marks (future feature):");

s1.inputMarks(90); // Allowed

s1.displayDetails();

s1.inputMarks(80); // Not allowed

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.

Program:

class Rectangle {

// Private fields (Encapsulation)

private double width;

private double height;

// Constructor with validation

public Rectangle(double width, double height) {

setWidth(width); // Use setter for validation

setHeight(height);

}

// Setter with validation for width

public void setWidth(double width) {

if (width > 0) {

this.width = width;

} else {

System.out.println("Invalid width. Setting default value to 1.");

this.width = 1;

}

}

// Setter with validation for height

public void setHeight(double height) {

if (height > 0) {

this.height = height;

} else {

System.out.println("Invalid height. Setting default value to 1.");

this.height = 1;

}

}

// Getters

public double getWidth() {

return width;

}

public double getHeight() {

return height;

}

// Method to calculate area

public double getArea() {

return width \* height;

}

// Method to calculate perimeter

public double getPerimeter() {

return 2 \* (width + height);

}

// Display method

public void displayDetails() {

System.out.println("Rectangle Details:");

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

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

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

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

}

}

// Main class to test Rectangle

public class Main {

public static void main(String[] args) {

Rectangle rect1 = new Rectangle(10, 5);

rect1.displayDetails();

System.out.println();

Rectangle rect2 = new Rectangle(-4, 8); // Invalid width

rect2.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.

Program:

import java.util.ArrayList;

import java.util.List;

class BankAccount {

private String accountNumber;

private String accountHolder;

private double balance;

private List<Transaction> transactionHistory;

// Inner class for transactions

private class Transaction {

String type;

double amount;

Transaction(String type, double amount) {

this.type = type;

this.amount = amount;

}

@Override

public String toString() {

return type + ": " + amount;

}

}

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

this.accountNumber = accountNumber;

this.accountHolder = accountHolder;

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

this.transactionHistory = new ArrayList<>();

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

transactionHistory.add(new Transaction("Deposit", amount));

} else {

System.out.println("Deposit amount must be positive.");

}

}

public boolean withdraw(double amount) {

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

balance -= amount;

transactionHistory.add(new Transaction("Withdraw", amount));

return true;

} else {

System.out.println("Invalid withdrawal. Either negative or insufficient funds.");

return false;

}

}

public double getBalance() {

return balance;

}

public String getLastTransaction() {

if (transactionHistory.isEmpty()) {

return "No transactions yet.";

}

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

}

@Override

public String toString() {

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.

Program:

class Locker {

private String lockerId;

private boolean isLocked;

private String passcode;

// Constructor

public Locker(String lockerId, String passcode) {

this.lockerId = lockerId;

this.passcode = passcode;

this.isLocked = true; // Default state is locked

}

// Private inner class for security

private class SecurityManager {

private boolean verifyPasscode(String code) {

return passcode.equals(code);

}

}

// Public method to lock

public void lock() {

if (!isLocked) {

isLocked = true;

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

} else {

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

}

}

// Public method to unlock with passcode

public boolean unlock(String code) {

SecurityManager sm = new SecurityManager();

if (sm.verifyPasscode(code)) {

if (isLocked) {

isLocked = false;

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

return true;

} else {

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

return true;

}

} else {

System.out.println("Incorrect passcode! Access denied.");

return false;

}

}

// Check if locker is locked

public boolean isLocked() {

return isLocked;

}

// Display details (without exposing passcode)

public void displayDetails() {

System.out.println("Locker ID: " + lockerId);

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

}

}

// Test class

public class Main {

public static void main(String[] args) {

Locker locker = new Locker("L001", "1234");

locker.displayDetails();

// Try wrong code

locker.unlock("0000");

// Try correct code

locker.unlock("1234");

// Lock again

locker.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.

Program

// Immutable Product Class with Builder Pattern

public final class Product {

// Private final fields - Immutable

private final String name;

private final String code;

private final double price;

private final String category;

// Private constructor - Only Builder can create instances

private Product(Builder builder) {

this.name = builder.name;

this.code = builder.code;

this.price = builder.price;

this.category = builder.category;

}

// Getters - No setters to ensure immutability

public String getName() {

return name;

}

public String getCode() {

return code;

}

public double getPrice() {

return price;

}

public String getCategory() {

return category;

}

// toString() for display

@Override

public String toString() {

return "Product{" +

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

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

", price=" + price +

", category='" + category + '\'' +

'}';

}

// Static Nested Builder Class

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.isBlank()) {

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

}

this.name = name;

return this;

}

public Builder withCode(String code) {

if (code == null || code.isBlank()) {

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; // category is optional

return this;

}

// Final build method - validates all required fields before creating the Product

public Product build() {

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

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

}

return new Product(this);

}

}

// Main method for testing

public static void main(String[] args) {

// Creating an immutable Product using Builder

Product p1 = new Product.Builder()

.withName("Laptop")

.withCode("LP1001")

.withPrice(75000)

.withCategory("Electronics")

.build();

System.out.println(p1);

}

}

**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.

Program:

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 bs = new BackwardSequence("hello");

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

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

System.out.println("CharAt(2): " + bs.charAt(2));

System.out.println("SubSequence(1,4): " + bs.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.

Program:

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;

}

@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 + ")";

}

}

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, bottomRight;

public MovableRectangle(MovablePoint topLeft, MovablePoint bottomRight) {

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

throw new IllegalArgumentException("Points must have 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(); }

@Override

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

@Override

public String toString() {

return "Rectangle TL=" + topLeft + ", BR=" + bottomRight;

}

}

class MovableTest {

public static void main(String[] args) {

MovableCircle c = new MovableCircle(5, new MovablePoint(0, 0, 2, 2));

System.out.println(c);

c.moveRight();

c.moveDown();

System.out.println("After Move: " + c);

MovableRectangle r = new MovableRectangle(

new MovablePoint(0,0,1,1),

new MovablePoint(3,3,1,1)

);

System.out.println(r);

r.moveUp();

r.moveLeft();

System.out.println("After Move: " + r);

}

}

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.

Program:

interface Printer {

void print(String document);

}

class LaserPrinter implements Printer {

public void print(String document) {

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

}

}

class InkjetPrinter implements Printer {

public void print(String document) {

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

}

}

class PrinterTest {

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.

Program:

interface BaseVehicle {

void start();

}

interface AdvancedVehicle extends BaseVehicle {

void stop();

boolean refuel(int amount);

}

class Car implements AdvancedVehicle {

private int fuel;

public Car(int fuel) {

this.fuel = fuel;

}

@Override

public void start() {

if (fuel > 0) System.out.println("Car started");

else System.out.println("No fuel to start");

}

@Override

public void stop() {

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

}

@Override

public boolean refuel(int amount) {

if (amount > 0) {

fuel += amount;

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

return true;

}

return false;

}

}

class VehicleTest {

public static void main(String[] args) {

BaseVehicle bv = new Car(0);

bv.start();

AdvancedVehicle av = (AdvancedVehicle) bv;

av.refuel(10);

bv.start();

av.stop();

}

}

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.

Program:

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) {

clients.add(client);

}

public void notifyClients() {

LocalDateTime now = LocalDateTime.now();

for (Client c : clients) {

c.updateTime(now);

}

}

}

class DesktopClient implements TimeServer.Client {

public void updateTime(LocalDateTime now) {

System.out.println("Desktop shows time: " + now);

}

}

class MobileClient implements TimeServer.Client {

public void updateTime(LocalDateTime now) {

System.out.println("Mobile shows time: " + now);

}

}

class TimeTest {

public static void main(String[] args) {

TimeServer server = new TimeServer();

server.registerClient(new DesktopClient());

server.registerClient(new MobileClient());

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().

Program:

interface Polygon {

double getArea();

default double getPerimeter(int... sides) {

double sum = 0;

for (int s : sides) sum += s;

return sum;

}

static String shapeInfo() {

return "Polygon: closed shape with straight sides";

}

}

class Rectangle implements Polygon {

private int length, breadth;

public Rectangle(int length, int breadth) {

this.length = length;

this.breadth = breadth;

}

public double getArea() {

return length \* breadth;

}

}

class Triangle implements Polygon {

private int base, height;

public Triangle(int base, int height) {

this.base = base;

this.height = height;

}

public double getArea() {

return 0.5 \* base \* height;

}

}

class PolygonTest {

public static void main(String[] args) {

Rectangle rect = new Rectangle(4,5);

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

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

Triangle tri = new Triangle(3,4);

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

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

System.out.println(Polygon.shapeInfo());

}

}

**Lambda expressions**

1. Sum of Two Integers

Program:

@FunctionalInterface

interface SumCalculator {

int sum(int a, int b);

}

public class LambdaExamples {

public static void main(String[] args) {

// 1 & 2: Sum of two integers

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

System.out.println("Sum: " + adder.sum(5, 7));

}

}

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

Program:

@FunctionalInterface

interface SumCalculator {

int sum(int a, int b);

}

public class LambdaExamples {

public static void main(String[] args) {

// 1 & 2: Sum of two integers

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

System.out.println("Sum: " + adder.sum(5, 7));

}

}

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();

Program:

import java.util.function.Predicate;

public class LambdaExamples {

public static void main(String[] args) {

// Check if string is empty

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

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

System.out.println("Is empty? " + isEmpty.test("Hello")); // false

}

}

1. Filter Even or Odd Numbers

Program:

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class LambdaExamples {

public static void main(String[] args) {

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

// Filter Even numbers

List<Integer> evenNumbers = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

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

// Filter Odd numbers

List<Integer> oddNumbers = numbers.stream()

.filter(n -> n % 2 != 0)

.collect(Collectors.toList());

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

}

}

1. Convert Strings to Uppercase/Lowercase

Program:

import java.util.function.Function;

public class UpperLowerCaseLambda {

public static void main(String[] args) {

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

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

String text = "Hello World";

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

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

}

}

1. Sort Strings by Length or Alphabetically

Program:

import java.util.\*;

public class SortStringsLambda {

public static void main(String[] args) {

List<String> list = Arrays.asList("banana", "apple", "kiwi", "mango");

// Sort by length

list.sort((s1, s2) -> Integer.compare(s1.length(), s2.length()));

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

// Sort alphabetically

list.sort(String::compareTo);

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

}

}

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

Program:

import java.util.Arrays;

public class AggregateOperationsLambda {

public static void main(String[] args) {

double[] numbers = {5.5, 2.2, 8.8, 1.1};

double sum = Arrays.stream(numbers).sum();

double max = Arrays.stream(numbers).max().orElse(Double.NaN);

double avg = Arrays.stream(numbers).average().orElse(Double.NaN);

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

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

System.out.println("Average: " + avg);

}

}

1. Create similar lambdas for max/min.

Program:

import java.util.function.BiFunction;

public class MaxMinLambda {

public static void main(String[] args) {

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

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

System.out.println("Max of 10 and 20: " + max.apply(10, 20));

System.out.println("Min of 10 and 20: " + min.apply(10, 20));

}

}

1. Calculate Factorial

Program:

import java.util.function.IntFunction;

public class FactorialLambda {

public static void main(String[] args) {

IntFunction<Long> factorial = n -> {

long fact = 1;

for (int i = 1; i <= n; i++) {

fact \*= i;

}

return fact;

};

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

}

}