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.

ANS:

```
package Day_5;

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 {
            this.marks = 0;
        }
    }

    public String getName() {
        return name;
    }

    public int getRollNumber() {
        return rollNumber;
    }

    public int getMarks() {
        return marks;
    }

    public void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Roll Number: " + rollNumber);
        System.out.println("Marks: " + marks);
    }

    public static void main(String[] args) {
        Student s = new Student("Sulkshana Patil", 101, 85);
        s.displayDetails();
    }
}
```

Output:

```
Name: Sulkshana Patil
Roll Number: 101
Marks: 85
```

---

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:

```
package Day_5;

public class Rectangle {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        if (width > 0 && height > 0) {
            this.width = width;
            this.height = height;
        } else {
            this.width = 1;
            this.height = 1;
        }
    }

    public void setWidth(double width) {
        if (width > 0) {
            this.width = width;
        }
    }

    public void setHeight(double height) {
        if (height > 0) {
            this.height = 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());
    }

    public static void main(String[] args) {
        Rectangle r = new Rectangle(5, 3);
        r.displayDetails();
    }
}
```

Output:

```
Width: 5.0
Height: 3.0
Area: 15.0
Perimeter: 16.0
```

---

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:

```
package Day_4;

public class BankAccount {
    private String accountNumber;
    private String accountHolder;
    private double balance;
    private String lastTransaction;

    public BankAccount(String accountNumber, String accountHolder, double balance) {
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
        if (balance >= 0) {
            this.balance = balance;
        }
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            lastTransaction = "Deposited: " + amount;
        }
    }

    public boolean withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            lastTransaction = "Withdrawn: " + amount;
            return true;
        } else {
            return false;
        }
    }

    public double getBalance() {
        return balance;
    }

    public String getLastTransaction() {
        return lastTransaction;
    }

    public void displayDetails() {
        System.out.println("Account Holder: " + accountHolder);
        System.out.println("Account Number: ****" + accountNumber.substring(accountNumber.length()
- 4));
        System.out.println("Balance: " + balance);
    }

    public static void main(String[] args) {
        BankAccount acc = new BankAccount("1234567890", "Sulkshana", 1000);
        acc.deposit(500);
        acc.withdraw(200);
        acc.displayDetails();
        System.out.println("Last Transaction: " + acc.getLastTransaction());
    }
}
```

Output:

```
Account Holder: Sulkshana
Account Number: ****7890
Balance: 1300.0
Last Transaction: Withdrawn: 200.0
```

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:

```
package Day_5;
public class Locker {
    private String lockerId;
    private boolean isLocked;
    private String passcode;

    public Locker(String lockerId, String passcode) {
        this.lockerId = lockerId;
        this.passcode = passcode;
        this.isLocked = true;
    }

    private class SecurityManager {
        private boolean verify(String code) {
            return passcode.equals(code);
        }
    }

    public void lock() {
        isLocked = true;
        System.out.println("Locker " + lockerId + " is locked.");
    }
    public void unlock(String code) {
        SecurityManager sm = new SecurityManager();
        if (sm.verify(code)) {
            isLocked = false;
            System.out.println("Locker " + lockerId + " unlocked successfully.");
        } else {
            System.out.println("Incorrect passcode. Locker remains locked.");
        }
    }

    public boolean isLocked() {
        return isLocked;
    }

    public static void main(String[] args) {
        Locker l = new Locker("L123", "pass@123");
        l.unlock("wrong");   // fail
        l.unlock("pass@123"); // success
        l.lock();
    }
}
```

Output:

```
Incorrect passcode. Locker remains locked.
Locker L123 unlocked successfully.
Locker L123 is locked.
```

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:

```
package Encapsulation;

public class Product {
    private final String name;
    private final String code;
    private final double price;
    private final String category;
    private Product(Builder b) {
        this.name = b.name;
        this.code = b.code;
        this.price = b.price;
        this.category = b.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) {
                this.price = price;
            }
            return this;
        }
```

```
        public Builder withCategory(String category) {
            this.category = category;
            return this;
        }

        public Product build() {
            return new Product(this);
        }
    }

    public static void main(String[] args) {
        Product p = new Product.Builder()
                .withName("Laptop")
                .withCode("P123")
                .withPrice(55000)
                .withCategory("Electronics")
                .build();

        System.out.println("Product: " + p.getName() + ", Price: " + p.getPrice());
    }
}
```

Output:

```
Product: Laptop, Price: 55000.0
```

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:

```
package Day_5;

public class BackwardSequence implements CharSequence {
    private String reversed;

    public BackwardSequence(String original) {
        StringBuilder sb = new StringBuilder(original);
        this.reversed = sb.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.substring(start, end);
    }

    @Override
    public String toString() {
        return reversed;
    }

    public static void main(String[] args) {
        BackwardSequence bs = new BackwardSequence("hello");
        System.out.println("Reversed: " + bs.toString());
        System.out.println("Length: " + bs.length());
        System.out.println("Char at 1: " + bs.charAt(1));
        System.out.println("SubSequence(1,4): " + bs.subSequence(1, 4));
    }
}
```

Output:

```
Reversed: olleh
Length: 5
Char at 1: l
SubSequence(1,4): lle
```

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:

```
package Day_5;

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 "(" + 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 "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;
    }

    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 "TopLeft=" + topLeft + ", BottomRight=" + bottomRight;
    }
}

public class MovableSimulation {
    public static void main(String[] args) {
        MovablePoint p1 = new MovablePoint(0, 0, 2, 2);
        MovableCircle c = new MovableCircle(5, p1);
        System.out.println(c);
        c.moveRight();
        System.out.println(c);

        MovablePoint tl = new MovablePoint(0, 0, 1, 1);
        MovablePoint br = new MovablePoint(3, 3, 1, 1);
        MovableRectangle r = new MovableRectangle(tl, br);
        System.out.println(r);
        r.moveDown();
        System.out.println(r);
    }
}
```

Output:

```
Center=(0, 0), radius=5
Center=(2, 0), radius=5
TopLeft=(0, 0), BottomRight=(3, 3)
```

TopLeft=(0, 1), BottomRight=(3, 4)

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:

```
package Day_5;

interface Printer {
    void print(String document);
}

class LaserPrinter implements Printer {
    public void print(String document) {
        System.out.println("Laser Printer printing: " + document);
    }
}

class InkjetPrinter implements Printer {
    public void print(String document) {
        System.out.println("Inkjet Printer printing: " + document);
    }
}

public class PrinterSwitch {
    public static void main(String[] args) {
        Printer p;

        p = new LaserPrinter();
        p.print("Java Assignment");

        p = new InkjetPrinter();
        p.print("Project Report");
    }
}
```

Output:

```
Laser Printer printing: Java Assignment
Inkjet Printer printing: Project Report
```

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:

```
package Day_5;

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;
    }

    public void start() {
        if (fuel > 0) {
            System.out.println("Car started");
        } else {
            System.out.println("No fuel to start");
        }
    }

    public void stop() {
        System.out.println("Car stopped");
    }

    public boolean refuel(int amount) {
        if (amount > 0) {
            fuel += amount;
            System.out.println("Refueled: " + amount);
            return true;
        }
        return false;
    }
}
```

```
public class AdvancedVehicleDemo {
    public static void main(String[] args) {
        AdvancedVehicle myCar = new Car(0);
        myCar.start();
        myCar.refuel(20);
        myCar.start();
        myCar.stop();
    }
}
```

Output:

```
No fuel to start
Refueled: 20
Car started
Car stopped
```

5. 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:

```
package Day_5;

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 is a 2D shape with straight sides.";
    }
}

class Rect implements Polygon {
    private int length, breadth;
    public Rect(int length, int breadth) {
        this.length = length;
        this.breadth = breadth;
    }

    public double getArea() {
        return length * breadth;
    }
}

class Tri implements Polygon {
    private int base, height;

    public Tri(int base, int height) {
        this.base = base;
        this.height = height;
    }

    public double getArea() {
        return 0.5 * base * height;
    }
}

public class Rectangle1 {
    public static void main(String[] args) {
        Rect r = new Rect(5, 3);
        System.out.println("Rectangle Area: " + r.getArea());
        System.out.println("Rectangle Perimeter: " + r.getPerimeter(5, 3, 5, 3));

        Tri t = new Tri(4, 6);
        System.out.println("Triangle Area: " + t.getArea());

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

Output:

```
Rectangle Area: 15.0
Rectangle Perimeter: 16.0
Triangle Area: 12.0
Polygon is a 2D shape with straight sides.
```

Lambda expressions

1. Sum of Two Integers

ANS:

```
package Day_5;

import java.util.Arrays;
import java.util.List;

public class SortStrings {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("java", "python", "c", "html");

        System.out.println("By Length:");
        words.stream().sorted((a, b) -> a.length() - b.length())
            .forEach(System.out::println);

        System.out.println("Alphabetically:");
        words.stream().sorted()
            .forEach(System.out::println);
    }
}
```

Output:

```
python
Alphabetically:
c
html
java
python
```

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

```
package Day_5;

import java.util.Arrays;

import java.util.List;

public class AggregateOps {
    public static void main(String[] args) {
        List<Double> nums = Arrays.asList(5.0, 2.5, 8.5, 3.0);

        double sum = nums.stream().mapToDouble(Double::doubleValue).sum();
        double max = nums.stream().mapToDouble(Double::doubleValue).max().orElse(0);
        double avg = nums.stream().mapToDouble(Double::doubleValue).average().orElse(0);

        System.out.println("Sum: " + sum);
        System.out.println("Max: " + max);
        System.out.println("Average: " + avg);
    }
}
```

```
Output:
Sum: 19.0
Max: 8.5
Average: 4.75
```

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();

```
package Day_5;

import java.util.function.Predicate;

public class CheckEmptyString {
    public static void main(String[] args) {
        Predicate<String> isEmpty = s-> s.isEmpty();

        String str1 = "";
        String str2 = "Java";

        System.out.println("Is str1 empty? " + isEmpty.test(str1));
        System.out.println("Is str2 empty? " + isEmpty.test(str2));
    }
}
```

```
Output:
Is str1 empty? true
Is str2 empty? false
```

4. Filter Even or Odd Numbers

```
package Day_5;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class FilterEvenOdd {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> evens = numbers.stream()
                              .filter(n-> n % 2 == 0)
                              .collect(Collectors.toList());

        List<Integer> odds = numbers.stream()
                             .filter(n-> n % 2 != 0)
                             .collect(Collectors.toList());

        System.out.println("Even numbers: " + evens);
        System.out.println("Odd numbers: " + odds);
    }
}
```

Output:

```
    Even numbers: [2, 4, 6, 8, 10]
    Odd numbers: [1, 3, 5, 7, 9]
```

5. Convert Strings to Uppercase/Lowercase

```
package Day_5;

public class StringCaseConversionEasy {
    public static void main(String[] args) {
        String word1 = "java";
        String word2 = "Lambda";
        String word3 = "EXPRESSION";

        System.out.println("Original: " + word1 + ", " + word2 + ", " + word3);
        System.out.println("Uppercase: " + word1.toUpperCase() + ", " +
word2.toUpperCase() + ", " + word3.toUpperCase());
        System.out.println("Lowercase: " + word1.toLowerCase() + ", " +
word2.toLowerCase() + ", " + word3.toLowerCase());
    }
}
```

Output:

```
Original: java, Lambda, EXPRESSION
Uppercase: JAVA, LAMBDA, EXPRESSION
Lowercase: java, lambda, expression
```

6. Sort Strings by Length or Alphabetically

```
package Day_5;

import java.util.Arrays;

public class StringSortEasy {
    public static void main(String[] args) {
        String[] words = {"banana", "apple", "kiwi", "grape"};


        Arrays.sort(words);
        System.out.println("Sorted alphabetically:");
        for (String w : words) {
            System.out.println(w);
        }
        for (int i = 0; i < words.length; i++) {
            for (int j = i + 1; j < words.length; j++) {
                if (words[i].length() > words[j].length()) {
                    String temp = words[i];
                    words[i] = words[j];
                    words[j] = temp;
                }
            }
        }

        System.out.println("\nSorted by length:");
        for (String w : words) {
            System.out.println(w);
        }
    }
}
```

Output:

```
 kiwi

Sorted by length:
kiwi
grape
apple
banana
```

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

```
package Day_5;

public class AggregateOps {
  public static void main(String[] args) {

    double[] nums = {5.0, 2.5, 8.5, 3.0};

    double sum = 0;

    double max = nums[0];

    for (double num : nums) {

      sum += num;

      if (num > max) {

        max = num;

      }

    }

    double avg = sum / nums.length;
```

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

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

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

    }
}
```

Output:

```
Sum: 19.0
Max: 8.5
Average: 4.75
```

8. Create similar lambdas for max/min.

```
package Day_5;

public class MaxMinEasy {
    public static void main(String[] args) {
        int[] nums = {4, 7, 1, 9, 3};

        int max = nums[0];
        int min = nums[0];

        for (int n : nums) {
            if (n > max) {
                max = n;
            }
            if (n < min) {
                min = n;
            }
        }

        System.out.println("Max: " + max);
        System.out.println("Min: " + min);
    }
}
```

Output:

```
Max: 9
Min: 1
```

9. Calculate Factorial

```
package Day_5;
```

```
import java.util.Scanner;

public class FactorialEasy {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number: ");

        int n = sc.nextInt();

        int fact = 1;

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

            fact *= i;

        }

       System.out.println("Factorial: " + fact);

    }

}
```