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.

```
package day_5_encapsulation;


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 void inputMarks(int newMarks) {
    if (newMarks > this.marks && newMarks <= 100) {
        this.marks = newMarks;
    } else {
        System.out.println("Invalid update: Marks must increase and be <= 100.");
    }
}

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                Student s1 = new Student("Raniya", 101, 95);
    Student s2 = new Student("Rahul", 102, 105);

    s1.displayDetails();
```

```
        System.out.println("-----");
        s2.displayDetails();

        s1.inputMarks(98);
        s1.inputMarks(90);
        System.out.println("-----After Updating Marks-----");
        s1.displayDetails();
            }
}
```

Output:

```
Name: Raniya
Roll Number: 101
Marks: 95
-----
Name: Rahul
Roll Number: 102
Marks: 0
Invalid update: Marks must increase and be <= 100.
-----After Updating Marks-----
Name: Raniya
Roll Number: 101
Marks: 98
```

---

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.

```
package day_5_encapsulation;
```

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

	public Rectangle(double width, double height) {
		this.width = (width > 0) ? width : 1;
		this.height = (height > 0) ? height : 1;
	}

	public void setWidth(double width) {
		if (width > 0) {
			this.width = width;
		} else {
			System.out.println("Width must be positive.");
		}
	}

	public void setHeight(double height) {
		if (height > 0) {
			this.height = height;
		} else {
			System.out.println("Height must be positive.");
		}
	}

	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());
    }

    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 5);
        Rectangle r2 = new Rectangle(-4, 0);

        System.out.println("Rectangle 1:");
        r1.displayDetails();

        System.out.println("\nRectangle 2 (after constructor validation):");
        r2.displayDetails();
        r2.setWidth(7);
        r2.setHeight(3);
        System.out.println("\nRectangle 2 (after setting valid dimensions):");
        r2.displayDetails();
```

```
        }
    }
```

Output:

```
Rectangle 1:
Width: 10.0
Height: 5.0
Area: 50.0
Perimeter: 30.0

Rectangle 2 (after constructor validation):
Width: 1.0
Height: 1.0
Area: 1.0
Perimeter: 4.0

Rectangle 2 (after setting valid dimensions):
Width: 7.0
Height: 3.0
Area: 21.0
Perimeter: 20.0
```

---

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.

```
package day_5_encapsulation;


import java.util.ArrayList;
import java.util.List;


public class BankAccount {
	private String accountNumber;
	private String accountHolder;
	private double balance;
	private List<String> transactionHistory;
	public BankAccount(String accountNumber, String accountHolder, double initialBalance) {
		this.accountNumber = accountNumber;
		this.accountHolder = accountHolder;
		this.balance = Math.max(initialBalance, 0);
		this.transactionHistory = new ArrayList<>();
	}


	public void deposit(double amount) {
		if (amount > 0) {
			balance += amount;
			transactionHistory.add("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("Withdraw: ₹" + amount);
                return true;
            } else {
                System.out.println("Withdrawal failed: Insufficient funds or
invalid amount.");
                return false;
            }
        }
        public double getBalance() {
            return balance;
        }
        public String getLastTransaction() {
            if (transactionHistory.isEmpty()) {
                return "No transactions yet.";
            }
            return transactionHistory.get(transactionHistory.size() - 1);
        }
        @Override
        public String toString() {
            String maskedAcc = "****" +
accountNumber.substring(accountNumber.length() - 4);
            return "Account Holder: " + accountHolder +
                   "\nAccount Number: " + maskedAcc +
                   "\nBalance: ₹" + balance;
        }
        public static void main(String[] args) {
            BankAccount acc = new BankAccount("1234567890", "Raniya", 5000);
            System.out.println(acc);
            acc.deposit(2000);
            acc.withdraw(1000);
            acc.withdraw(10000)
            System.out.println("\nAfter transactions:");
```

```
            System.out.println(acc);
            System.out.println("Last Transaction: " + acc.getLastTransaction());
        }
    }
```

Output:

Account Holder: Raniya

Account Number: ****7890

Balance: ₹5000.0

Withdrawal failed: Insufficient funds or invalid amount.

After transactions:

Account Holder: Raniya

Account Number: ****7890

Balance: ₹6000.0

Last Transaction: Withdraw: ₹1000.0

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.

```
package day_5_encapsulation;

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

```
    private final SecurityManager securityManager = new SecurityManager();

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

    public void lock() {
        isLocked = true;
        System.out.println("Locker " + lockerId + " is now locked.");
    }

    public boolean unlock(String code) {
        if (securityManager.verify(code)) {
            isLocked = false;
            System.out.println("Locker " + lockerId + " unlocked
successfully.");
            return true;
        } else {
            System.out.println("Incorrect passcode.");
            return false;
        }
    }

    public boolean isLocked() {
        return isLocked;
    }

    private class SecurityManager {
        private boolean verify(String inputCode) {
            return passcode.equals(inputCode);
```

```
        }
    }

    public static void main(String[] args) {
        Locker locker = new Locker("LOCK001", "1234");

        System.out.println("Is Locked? " + locker.isLocked());
        locker.unlock("1111");
        locker.unlock("1234");

        locker.lock();
    }
}
```

Output

```
Is Locked? true
Incorrect passcode.
Locker LOCK001 unlocked successfully.
Locker LOCK001 is now locked.
```

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.

```
package day_5_interface;

public class BackwardSequence implements CharSequence {
```

```
private final String original;
private final String reversed;

public BackwardSequence(String original) {
   this.original = original;
   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 start or end index.");
   }
   return reversed.substring(start, end);
}

public String toString() {
   return reversed;
}

public static void main(String[] args) {
```

```
        BackwardSequence bs = new BackwardSequence("hello");


        System.out.println("Original: hello");
        System.out.println("Reversed: " +bs);
        System.out.println("Length: " +bs.length());
        System.out.println("Char at index 1: " +bs.charAt(1));
        System.out.println("Subsequence (1, 4): " +bs.subSequence(1, 4));
    }
}
```

Output:

```
Original: hello
Reversed: olleh
Length: 5
Char at index 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.

```
package day_5_interface;


public class MovableShapeSimulation {


        interface Movable{
```

```
        void moveUp();
        void moveDown();
        void moveLeft();
        void moveRight();
    }

    static 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 "Point(" + x + ", " + y + ")";
}









    }

    static class MovableCircle implements Movable {
private int radius;
private 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(radius=" + radius + ", center=" + center + ")";
    }
  }

  // MovableRectangle class
  static class MovableRectangle implements Movable {
    private MovablePoint topLeft;
    private MovablePoint bottomRight;

    public MovableRectangle(MovablePoint topLeft, MovablePoint bottomRight) {
      if (topLeft.xSpeed != bottomRight.xSpeed || topLeft.ySpeed !=
bottomRight.ySpeed) {
        throw new IllegalArgumentException("Speeds of topLeft and bottomRight
must be the same.");
      }
```

```
    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(TopLeft=" + topLeft + ", BottomRight=" + bottomRight + ")";
```

```
    }
  }

      public static void main(String[] args) {
            // TODO Auto-generated method stub
            MovablePoint p1 = new MovablePoint(0, 0, 2, 2);
    System.out.println("Initial Point: " + p1);
    p1.moveRight();
    p1.moveUp();
    System.out.println("After Move: " + p1);
    // MovableCircle
    MovableCircle circle = new MovableCircle(5, new MovablePoint(10, 10, 1, 1));
    System.out.println("\nInitial Circle: " + circle);
    circle.moveDown();
    circle.moveLeft();
    System.out.println("After Move: " + circle);
    // MovableRectangle
    MovablePoint topLeft = new MovablePoint(5, 5, 2, 2);
    MovablePoint bottomRight = new MovablePoint(15, 1, 2, 2);
    MovableRectangle rectangle = new MovableRectangle(topLeft, bottomRight);
    System.out.println("\nInitial Rectangle: " + rectangle);
    rectangle.moveUp();
    rectangle.moveRight();
    System.out.println("After Move: " + rectangle);
      }
}
```

Output:

```
Initial Point: Point(0, 0)
After Move: Point(2, -2)

Initial Circle: Circle(radius=5, center=Point(10, 10))
```

After Move: Circle(radius=5, center=Point(9, 11))

Initial Rectangle: Rectangle(TopLeft=Point(5, 5), BottomRight=Point(15, 1))

After Move: Rectangle(TopLeft=Point(7, 3), BottomRight=Point(17, -1))

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.

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.

```
interface Printer {
    void print(String document);
}

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

class InkjetPrinter implements Printer {
    public void print(String document) {
```

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

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

            p = new LaserPrinter();
            p.print("Java Programming Notes");

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

OUTPUT

LaserPrinter is printing: Java Programming Notes

InkjetPrinter is printing: Design Patterns Report

Lambda expressions

1. Sum of Two Integers

```
package day_5_interface;
interface SumOperation {
   int sum(int a, int b);
}
public class LambdaSumExample {
   public static void main(String[] args)
     SumOperation add = (a, b) -> a + b;
```

```
        int num1 = 10;
        int num2 = 20;
        int result = add.sum(num1, num2);

        System.out.println("Sum of " + num1 + " and " + num2 + " is: " + result);
    }
}
```

Output:

```
Sum of 10 and 20 is: 30
```

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

```
package day_5_interface;

interface SumCalculator {
    int sum(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        SumCalculator add = (a, b) -> a + b;
        System.out.println(add.sum(5, 7));
    }
}
```

Output:

```
12
```

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_interface;

import java.util.function.Predicate;
```

```
public class StringIsEmplty {
   public static void main(String[] args) {
      Predicate<String> isEmpty = s -> s.isEmpty();
      System.out.println(isEmpty.test(""));
      System.out.println(isEmpty.test("Hello"));
   }
}
```

Output:

true

false

4. Filter Even or Odd Numbers

```
package day_5_interface;

import java.util.*;
import java.util.stream.*;

public class OddEven {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
      List<Integer> even = numbers.stream().filter(n -> n % 2 == 0).toList();
      List<Integer> odd = numbers.stream().filter(n -> n % 2 != 0).toList();
      System.out.println(even);
      System.out.println(odd);
   }
}
```

Output:

[2, 4, 6]

[1, 3, 5]

5. Convert Strings to Uppercase/Lowercase

```
package day_5_interface;

import java.util.*;

import java.util.stream.*;


public class UPPERCASE {
```

```
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "Banana", "Cherry");
        List<String> upper = words.stream().map(s -> s.toUpperCase()).toList();
        List<String> lower = words.stream().map(s -> s.toLowerCase()).toList();
        System.out.println(upper);
        System.out.println(lower);
    }
}
```

Output:

```
[APPLE, BANANA, CHERRY]
[apple, banana, cherry]
```

6. Sort Strings by Length or Alphabetically

```
package day_5_interface;
import java.util.*;

public class SORTSTRINGS {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "kiwi", "cherry");
        words.sort((a, b) -> Integer.compare(a.length(), b.length()));
        System.out.println(words);
        words.sort((a, b) -> a.compareTo(b));
        System.out.println(words);
    }
}
```

Output:

```
[kiwi, apple, banana, cherry]
[apple, banana, cherry, kiwi]
```

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

```
package day_5_interface;

import java.util.*;
import java.util.stream.*;

public class AGGREGATE {
   public static void main(String[] args) {
      double[] nums = {2.5, 7.3, 1.8, 4.0};
      double sum = Arrays.stream(nums).sum();
      double max = Arrays.stream(nums).max().orElse(0);
      double avg = Arrays.stream(nums).average().orElse(0);
      System.out.println(sum);
      System.out.println(max);
      System.out.println(avg);
   }
}
```

Output:

15.6

7.3

3.9

8. Create similar lambdas for max/min.

```
package day_5_interface;

import java.util.*;

public class MAXMIN {
   public static void main(String[] args) {
      List<Integer> nums = Arrays.asList(5, 2, 9, 1, 7);
      int max = nums.stream().max((a, b) -> a - b).orElse(0);
```

```
            int min = nums.stream().min((a, b) -> a - b).orElse(0);
            System.out.println(max);
            System.out.println(min);
        }
    }
```

Output:

```
9
1
```

9. Calculate Factorial

```
package day_5_interface;

public class FACTORIAL {
   interface Factorial {
      int calc(int n);
   }
   public static void main(String[] args) {
      Factorial f = n -> {
         int fact = 1;
         for (int i = 1; i <= n; i++) fact *= i;
         return fact;
      };
      System.out.println(f.calc(5));
   }
}
```

Output:

```
120
```