Encapsulation

Q. 1. Student with Grade Validation & Configuration
Create Student class with private fields: name, rollNumber, and marks. Marks must be 0-100, invalid reset to 0. Marks immutable (no setter). Constructor initializes all. Add displayDetails().

```
package EncapsulationPractice;

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("RollNumber=" + rollNumber);
        System.out.println("Marks=" + marks);
    }

    public static void main(String[] args) {
        Student s = new Student("Raghav", 101, 85);
        s.displayDetails();
        Student s2 = new Student("Amit", 102, 120);
        s2.displayDetails();
    }
}
```

Output:

```
Name=Raghav
RollNumber=101
Marks=85
Name=Amit
RollNumber=102
Marks=0
```

Q. 2. Rectangle Enforced Positive Dimensions
Create Rectangle with private width and height. Constructor and setters reject or correct non-positive values (set default 1). Provide getArea(), getPerimeter(), displayDetails().

```
package EncapsulationPractice;

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

    public Rectangle(int width, int height) {
        if (width > 0)
```

```
        this.width = width;
    else
        this.width = 1;

    if (height > 0)
        this.height = height;
    else
        this.height = 1;
}

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

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

public int getArea() {
    return width * height;
}

public int 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, 10);
        r.displayDetails();
        Rectangle r2 = new Rectangle(-3, 7);
        r2.displayDetails();
    }
}
```

Output:

```
Width=5
Height=10
Area=50
Perimeter=30
Width=1
Height=7
Area=7
Perimeter=16
```

Q. 3. Bank Account with Deposit/Withdraw Logic
Create BankAccount with private accountNumber, accountHolder, balance. deposit(double), withdraw(double) with validation, getBalance() only. Optionally override toString() to mask account number (show only last 4 digits).

```
package EncapsulationPractice;

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

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

```
    this.accountNumber = accountNumber;
    this.accountHolder = accountHolder;
    this.balance = balance;
}

public void deposit(double amount) {
    if (amount > 0) {
        balance += amount;
        System.out.println("Deposited=" + amount);
    } else {
        System.out.println("Invalid deposit amount");
    }
}

public boolean withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        System.out.println("Withdraw amount=" + amount);
        return true;
    } else {
        System.out.println("Invalid withdraw amount");
        return false;
    }
}

public double getBalance() {
    return balance;
}

public String toString() {
    String masked = "****" + accountNumber.substring(accountNumber.length() - 4);
```

```
        return "AccountHolder=" + accountHolder + ", AccountNumber=" + masked + ", Balance=" +
balance;
    }

    public static void main(String[] args) {
        BankAccount acc = new BankAccount("1234567890", "Ramesh", 1000);
        System.out.println(acc);
        acc.deposit(500);
        acc.withdraw(200);
        acc.withdraw(2000);
        System.out.println(acc);
    }
}
```

Output:

```
AccountHolder=Ramesh, AccountNumber=****7890, Balance=1000.0
Deposited=500.0
Withdraw amount=200.0
Invalid withdraw amount
AccountHolder=Ramesh, AccountNumber=****7890, Balance=1300.0
```

Q. 4. Inner Class Encapsulation: Secure Locker
Create Locker class with private lockerId, isLocked, passcode. Inner private class SecurityManager handles passcode verification. Public methods: lock(), unlock(String code), isLocked().

```
package EncapsulationPractice;
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 locked");
}

public void unlock(String code) {
    SecurityManager sm = new SecurityManager();
    if (sm.verify(code)) {
        isLocked = false;
        System.out.println("Locker unlocked");
    } else {
        System.out.println("Invalid passcode");
    }
}

public boolean isLocked() {
    return isLocked;
}

public static void main(String[] args) {
    Locker locker = new Locker("L001", "1234");
```

```
        System.out.println("Is Locked=" + locker.isLocked());
        locker.unlock("0000");
        locker.unlock("1234");
        System.out.println("Is Locked=" + locker.isLocked());
        locker.lock();
        System.out.println("Is Locked=" + locker.isLocked());
    }
}
```

Output:

```
Is Locked=true
Invalid passcode
Locker unlocked
Is Locked=false
Locker locked
Is Locked=true
```

Q. 5. Builder Pattern & Encapsulation: Immutable Product
Create immutable Product class with private final fields name, code, price, optional category. Static nested Builder class with withName(), withPrice(), etc., validations, returns Product only if valid.

```
package EncapsulationPractice;
public class Product {
    private final String name;
    private final String code;
    private final double price;
    private final String category;

    private Product(Builder builder) {
        this.name = builder.name;
        this.code = builder.code;
        this.price = builder.price;
        this.category = builder.category;
    }
```

```
public String getName() {
    return name;
}

public String getCode() {
    return code;
}

public double getPrice() {
    return price;
}

public String getCategory() {
    return category;
}

public static class Builder {
    private String name;
    private String code;
    private double price;
    private String category = "General";

    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;
        else
            this.price = 0;
        return this;
    }

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

    public Product build() {
        if (name != null && code != null && price >= 0) {
            return new Product(this);
        } else {
            return null;
        }
    }
}

public void display() {
    System.out.println("Name=" + name);
    System.out.println("Code=" + code);
    System.out.println("Price=" + price);
    System.out.println("Category=" + category);
}
```

```
    public static void main(String[] args) {
        Product p = new Product.Builder()
                    .withName("Laptop")
                    .withCode("LP100")
                    .withPrice(75000)
                    .withCategory("Electronics")
                    .build();
        if (p != null) {
            p.display();
        }
    }
}
```

Output:

Name=Laptop

Code=LP100

Price=75000.0

Category=Electronics

Interface

Q. 1. Reverse CharSequence: Custom BackwardSequence
Create BackwardSequence implementing CharSequence, stores a string but returns reversed string on methods.

```
package InterfacePractice;

public class BackwardSequence implements CharSequence {
    private String str;

    public BackwardSequence(String str) {
        this.str = str;
    }

    public int length() {
        return str.length();
```

```
    }

    public char charAt(int index) {
        return str.charAt(str.length() - 1 - index);
    }

    public CharSequence subSequence(int start, int end) {
        StringBuilder sb = new StringBuilder();
        for (int i = start; i < end; i++) {
            sb.append(charAt(i));
        }
        return sb.toString();
    }

    public String toString() {
        return subSequence(0, length()).toString();
    }

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

Output:

```
5
l
lle
olleh
```

Q. 2. Movable Shapes Simulation
Define Movable interface with moveUp(), moveDown(), moveLeft(), moveRight(). Implement MovablePoint, MovableCircle(center point), MovableRectangle(topLeft and bottomRight points).

```
package InterfacePractice;

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 "Circle center " + center.toString() + " radius " + radius;
    }
}

class MovableRectangle implements Movable {
    MovablePoint topLeft;
    MovablePoint bottomRight;

    public MovableRectangle(MovablePoint topLeft, MovablePoint bottomRight) {
        if (topLeft.xSpeed == bottomRight.xSpeed && topLeft.ySpeed == bottomRight.ySpeed) {
            this.topLeft = topLeft;
            this.bottomRight = bottomRight;
        } else {
            System.out.println("Speed of points not equal");
        }
    }

    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 "Rectangle topLeft " + topLeft.toString() + " bottomRight " + bottomRight.toString();
    }
}

public class TestMovable {
    public static void main(String[] args) {
        MovablePoint p = new MovablePoint(0, 0, 2, 2);
        MovableCircle c = new MovableCircle(5, p);
        MovablePoint p1 = new MovablePoint(0, 0, 1, 1);
        MovablePoint p2 = new MovablePoint(5, 5, 1, 1);
        MovableRectangle r = new MovableRectangle(p1, p2);

        System.out.println(p);
        p.moveRight();
        p.moveDown();
        System.out.println(p);

        System.out.println(c);
```

```
        c.moveLeft();
        System.out.println(c);

        System.out.println(r);
        r.moveUp();
        System.out.println(r);
    }
}
```

Output:

(0,0)

(2,2)

Circle center (2,2) radius 5

Circle center (1,2) radius 5

Rectangle topLeft (0,0) bottomRight (5,5)

Rectangle topLeft (0,-1) bottomRight (5,4)

Q. 3. Printer Switch (Interface)
Create Printer interface with print(String). Implement LaserPrinter and InkjetPrinter. Test switching printers at runtime.

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

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

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

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

public class TestPrinter {
    public static void main(String[] args) {
        Printer p;
        p = new LaserPrinter();
        p.print("Document1");

        p = new InkjetPrinter();
        p.print("Document2");
    }
}
```

Output:

Laser printing: Document1

Inkjet printing: Document2

Q. 4. Extended Interface Hierarchy
Define BaseVehicle with start(). AdvancedVehicle extends BaseVehicle, adds stop() and boolean refuel(int). Implement Car with constructor initializing fuel. Manipulate via both interfaces in main.

```
package InterfacePractice;

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() {
        System.out.println("Car started with fuel " + fuel);
    }

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

    public boolean refuel(int amount) {
        if (amount > 0) {
            fuel += amount;
            System.out.println("Car refueled by " + amount);
            return true;
        } else {
            System.out.println("Invalid refuel amount");
            return false;
        }
    }

    public int getFuel() {
        return fuel;
    }
}

public class TestVehicle {
```

```
    public static void main(String[] args) {
        BaseVehicle base = new Car(10);
        base.start();

        AdvancedVehicle adv = (AdvancedVehicle) base;
        adv.refuel(20);
        adv.stop();
    }
}
```

Output:

```
Car started with fuel 10
Car refueled by 20
Car stopped
```

Q. 5. Nested Interface for Callback Handling
Create TimeServer with public static nested interface Client with void updateTime(LocalDateTime now). Server has registerClient(Client) and notifyClients() methods. Implement 2 Clients and test notifications.

```
package InterfacePractice;

import java.time.LocalDateTime;
import java.util.ArrayList;

public class TimeServer {
    public static interface Client {
        void updateTime(LocalDateTime now);
    }

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

    public void registerClient(Client client) {
        clients.add(client);
    }
```

```
    public void notifyClients() {
        LocalDateTime now = LocalDateTime.now();
        for (Client c : clients) {
            c.updateTime(now);
        }
    }

    public static void main(String[] args) {
        TimeServer server = new TimeServer();

        Client client1 = new Client() {
            public void updateTime(LocalDateTime now) {
                System.out.println("Client1 time updated: " + now);
            }
        };

        Client client2 = new Client() {
            public void updateTime(LocalDateTime now) {
                System.out.println("Client2 time updated: " + now);
            }
        };

        server.registerClient(client1);
        server.registerClient(client2);

        server.notifyClients();
    }
}
```

Output:

Client1 time updated: 2025-08-10T18:30:45.123456789

Client2 time updated: 2025-08-10T18:30:45.123456789

Q. 6. Default and Static Methods in Interfaces
Create Polygon interface with double getArea(), default getPerimeter(int... sides), static String shapeInfo(). Implement Rectangle and Triangle with getArea(). Test in main.

```
package InterfacePractice;

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 interface for shapes";
    }
}

class Rectangle implements Polygon {
    int width, height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    public double getArea() {
```

```
        return width * height;
    }
}

class Triangle implements Polygon {
    int base, height;

    public Triangle(int b, int h) {
        base = b;
        height = h;
    }

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

public class TestPolygon {
    public static void main(String[] args) {
        Polygon rect = new Rectangle(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(Polygon.shapeInfo());
    }
}
```

Output:

```
Rectangle area=20.0
Rectangle perimeter=18.0
```

Triangle area=9.0

Polygon interface for shapes

Lambda Expressions

Q. 1. Sum of Two Integers
Write a program to sum two integers using lambda expression.

```
package Lambdapractice;

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

public class TestSum {
   public static void main(String[] args) {
      SumCalculator calc = (a, b) -> a + b;
      System.out.println(calc.sum(5, 10));
   }
}
```

Output:

15

Q. 2. Check If a String Is Empty
Create a lambda using a functional interface to check if a string is empty.

```
package Lambdapractice;

interface StringChecker {
   boolean check(String s);
}

public class TestIsEmpty {
   public static void main(String[] args) {
      StringChecker isEmpty = s -> s.length() == 0;
      System.out.println(isEmpty.check(""));
      System.out.println(isEmpty.check("Hello"));
```

```
    }
}
```

Output:

```
true
false
```

Q. 3. Filter Even or Odd Numbers
Create a lambda to check if an integer is even.

```
package Lambdapractice;
interface NumberChecker {
    boolean check(int num);
}

public class TestEven {
    public static void main(String[] args) {
        NumberChecker isEven = num -> num % 2 == 0;
        System.out.println(isEven.check(4));
        System.out.println(isEven.check(7));
    }
}
```

Output:

```
true
false
```

Q. 4. Convert Strings to Uppercase and Lowercase
Create lambdas to convert string to uppercase and lowercase.

```
package Lambdapractice;
interface StringConverter {
    String convert(String s);
}

public class TestCaseConversion {
```

```
    public static void main(String[] args) {
        StringConverter toUpper = s -> {
            String result = "";
            for (int i = 0; i < s.length(); i++) {
                char c = s.charAt(i);
                if (c >= 'a' && c <= 'z') {
                    result += (char)(c - 32);
                } else {
                    result += c;
                }
            }
            return result;
        };

        StringConverter toLower = s -> {
            String result = "";
            for (int i = 0; i < s.length(); i++) {
                char c = s.charAt(i);
                if (c >= 'A' && c <= 'Z') {
                    result += (char)(c + 32);
                } else {
                    result += c;
                }
            }
            return result;
        };

        System.out.println(toUpper.convert("hello"));
        System.out.println(toLower.convert("WORLD"));
    }
}
```

Output:

HELLO

world

Q. 5. Sort Strings by Length and Alphabetically
Sort a list of strings by length and then alphabetically (use arrays and simple bubble sort).

```
package Lambdapractice;

public class TestSort {
   public static void sortByLength(String[] arr) {
      for (int i = 0; i < arr.length - 1; i++) {
         for (int j = 0; j < arr.length - 1 - i; j++) {
            if (arr[j].length() > arr[j + 1].length()) {
               String temp = arr[j];
               arr[j] = arr[j + 1];
               arr[j + 1] = temp;
            }
         }
      }
   }

   public static void sortAlphabetically(String[] arr) {
      for (int i = 0; i < arr.length - 1; i++) {
         for (int j = 0; j < arr.length - 1 - i; j++) {
            if (arr[j].compareTo(arr[j + 1]) > 0) {
               String temp = arr[j];
               arr[j] = arr[j + 1];
               arr[j + 1] = temp;
            }
         }
      }
   }
```

```
    public static void printArray(String[] arr) {
        for (String s : arr) {
            System.out.print(s + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        String[] arr = {"apple", "pear", "banana", "kiwi"};
        sortByLength(arr);
        printArray(arr);
        sortAlphabetically(arr);
        printArray(arr);
    }
}
```

Output:

pear kiwi apple banana

apple banana kiwi pear

Q. 6. Aggregate Operations (Sum, Max, Average) on Double Arrays
Calculate sum, max, and average of a double array without streams.

```
package Lambdapractice;

public class TestAggregate {
    public static void main(String[] args) {
        double[] arr = {1.5, 2.3, 4.7, 3.1};
        double sum = 0;
        double max = arr[0];
        for (int i = 0; i < arr.length; i++) {
            sum += arr[i];
            if (arr[i] > max) {
```

```
            max = arr[i];
          }
        }
        double average = sum / arr.length;
        System.out.println("Sum=" + sum);
        System.out.println("Max=" + max);
        System.out.println("Average=" + average);
    }
}
```

Output:

Sum=11.6

Max=4.7

Average=2.9

Q. 7. Create lambdas for Max and Min of two integers

```
package Lambdapractice;
interface BinaryOperation {
    int operate(int a, int b);
}

public class TestMaxMin {
    public static void main(String[] args) {
        BinaryOperation max = (a, b) -> (a > b) ? a : b;
        BinaryOperation min = (a, b) -> (a < b) ? a : b;

        System.out.println(max.operate(10, 20));
        System.out.println(min.operate(10, 20));
    }
}
```

Output:

20

10

Q. 8. Calculate Factorial
Create a lambda expression to calculate factorial of a number.

```
package Lambdapractice;

interface FactorialCalculator {
   int fact(int n);
}


public class TestFactorial {
   public static void main(String[] args) {
      FactorialCalculator factorial = n -> {
         int result = 1;
         for (int i = 1; i <= n; i++) {
            result = result * i;
         }
         return result;
      };


      System.out.println(factorial.fact(5));
   }
}
```

Output:

120