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.

```
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 >= 0 && newMarks <= 100 && newMarks > this.marks) {
            this.marks = newMarks;
        }
    }
}
```

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.**

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

    public Rectangle(double width, double height) {
        if (width > 0) {
            this.width = width;
        } else {
            this.width = 1.0;
        }
        if (height > 0) {
            this.height = height;
        } else {
            this.height = 1.0;
        }
    }

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

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

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }

    public double getArea() {
        return width * height;
    }
```

```
    public double getPerimeter() {
        return 2 * (width + height);
    }

    public void displayDetails() {
        System.out.println("Width: " + width);
        System.out.println("Height: " + height);
        System.out.println("Area: " + getArea());
        System.out.println("Perimeter: " + getPerimeter());
    }
}
```

3. Advanced: Bank Account with Deposit/Withdraw Logic

Transaction validation and encapsulation protection.

- **Create a BankAccount class with private accountNumber, accountHolder, balance.**
- **Provide:**
    - **deposit(double amount) — ignores or rejects negative.**
    - **withdraw(double amount) — prevents overdraft and returns a boolean success.**
    - **Getter for balance but no setter.**
- **Optionally override toString() to display masked account number and details.**
- **Track transaction history internally using a private list (or inner class for transaction object).**
- **Expose a method getLastTransaction() but do not expose the full internal list.**

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

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

    private List<Transaction> transactionHistory = new ArrayList<>();
```

```
public BankAccount(String accountNumber, String accountHolder, double initialBalance) {
    this.accountNumber = accountNumber;
    this.accountHolder = accountHolder;
    this.balance = initialBalance >= 0 ? initialBalance : 0;
    if (initialBalance > 0) {
        transactionHistory.add(new Transaction("Deposit", initialBalance));
    }
}

public boolean deposit(double amount) {
    if (amount <= 0) {
        return false;
    }
    balance += amount;
    transactionHistory.add(new Transaction("Deposit", amount));
    return true;
}

public boolean withdraw(double amount) {
    if (amount <= 0 || amount > balance) {
        return false;
    }
    balance -= amount;
    transactionHistory.add(new Transaction("Withdraw", amount));
    return true;
}

public double getBalance() {
    return balance;
}

public String getLastTransaction() {
    if (transactionHistory.isEmpty()) {
        return "No transactions yet.";
        }
        Transaction last = transactionHistory.get(transactionHistory.size() - 1);
        return last.toString();
    }

    @Override
    public String toString() {
        String maskedAccount = "****" + accountNumber.substring(accountNumber.length() - 4);
        return "Account Holder: " + accountHolder + "\nAccount Number: " + maskedAccount + "\nBalance: "
+ balance;
    }

    private class Transaction {
        private String type;
        private double amount;

        public Transaction(String type, double amount) {
            this.type = type;
            this.amount = amount;
        }

        @Override
        public String toString() {
            return type + ": " + amount;
        }
    }
}
```

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.**

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

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

    public void lock() {
        isLocked = true;
    }

    public boolean unlock(String code) {
        if (securityManager.verify(code)) {
            isLocked = false;
            return true;
        }
        return false;
    }

    public boolean isLocked() {
        return isLocked;
    }

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

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.**

```
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;

        public Builder withName(String name) {
            if (name != null && !name.trim().isEmpty()) {
                this.name = name;
            }
            return this;
        }

        public Builder withCode(String code) {
            if (code != null && !code.trim().isEmpty()) {
                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() {
            if (name == null || code == null || price < 0) {
                throw new IllegalStateException("Invalid product data");
            }
            return new Product(this);
        }
    }
}
```

Interface

1. Reverse CharSequence: Custom BackwardSequence

- **Create a class BackwardSequence that implements java.lang.CharSequence.**
- **Internally store a String and implement all required methods: length(), charAt(), subSequence(), and toString().**
- **The sequence should be the reverse of the stored string (e.g., new BackwardSequence("hello") yields "olleh").**
- **Write a main() method to test each method.**

```
public class BackwardSequence implements CharSequence {
    private final String original;
    private final String reversed;

    public BackwardSequence(String input) {
        this.original = input;
        this.reversed = new StringBuilder(input).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("toString(): " + bs.toString());
        System.out.println("length(): " + bs.length());
        System.out.println("charAt(0): " + bs.charAt(0));
        System.out.println("charAt(4): " + bs.charAt(4));
        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.**

// 1. Movable Interface:

```
public interface Movable {
   void moveUp();
   void moveDown();
   void moveLeft();
   void moveRight();
}
```

// 2. MovablePoint Class:

```
public 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 + ")";
    }
}
```

// 3. MovableCircle Class:

```
public 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(center=" + center + ", radius=" + radius + ")";
    }
}
```

// 4. MovableRectangle Class:

```
public 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("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(topLeft=" + topLeft + ", bottomRight=" + bottomRight + ")";
    }
}
```

// 5. Main Method to Simulate Motion:

```
public class Main {
    public static void main(String[] args) {
        MovablePoint p = new MovablePoint(0, 0, 2, 3);
        System.out.println("Initial Point: " + p);
        p.moveUp();
        p.moveRight();
        System.out.println("Moved Point: " + p);

        MovableCircle circle = new MovableCircle(5, new MovablePoint(10, 10, 1, 1));
        System.out.println("Initial Circle: " + circle);
        circle.moveDown();
        circle.moveLeft();
        System.out.println("Moved Circle: " + circle);

        MovablePoint rectTopLeft = new MovablePoint(0, 0, 2, 2);
        MovablePoint rectBottomRight = new MovablePoint(4, 4, 2, 2);
        MovableRectangle rect = new MovableRectangle(rectTopLeft, rectBottomRight);
        System.out.println("Initial Rectangle: " + rect);
        rect.moveRight();
        rect.moveDown();
        System.out.println("Moved Rectangle: " + rect);
    }
}
```

3. Contract Programming: Printer Switch

- **Declare an interface Printer with method void print(String document).**
- **Implement two classes: LaserPrinter and InkjetPrinter, each providing unique behavior.**
- **In the client code, declare Printer p;, switch implementations at runtime, and test printing.**

// 1.Printer Interface:

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

// 2. LaserPrinter Class:

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

// 3. InkjetPrinter Class:

```
public class InkjetPrinter implements Printer {
    @Override
    public void print(String document) {
        System.out.println("InkjetPrinter is printing: " + document.toLowerCase());
    }
}
```

// 4. Main Class to Test Switching:

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

        p = new LaserPrinter();
        p.print("Important Document");

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

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.**

// 1. BaseVehicle Interface:

```
public interface BaseVehicle {
    void start();
}
```

// 2. AdvancedVehicle Interface (extends BaseVehicle):

```
public interface AdvancedVehicle extends BaseVehicle {
    void stop();
    boolean refuel(int amount);
}
```

// 3. Car Class Implements AdvancedVehicle:

```
public class Car implements AdvancedVehicle {
    private int fuel;

    public Car(int initialFuel) {
        this.fuel = initialFuel;
    }

    @Override
    public void start() {
        if (fuel > 0) {
            System.out.println("Car started.");
            fuel--;
        } else {
            System.out.println("Cannot start. Fuel is empty.");
        }
    }

    @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 + " units.");
            return true;
        }
        System.out.println("Refuel failed. Amount must be positive.");
        return false;
    }

    public int getFuelLevel() {
        return fuel;
    }
}
```

// 4. Main Method to Demonstrate Interface Polymorphism:

```
public class Main {
    public static void main(String[] args) {
        BaseVehicle base = new Car(2);
        base.start();
        AdvancedVehicle advanced = (AdvancedVehicle) base;
        advanced.stop();
        advanced.refuel(5);
        advanced.start();
        Car car = (Car) advanced;
        System.out.println("Current fuel: " + car.getFuelLevel());
    }
}
```

5. Nested Interface for Callback Handling

- **Create a class TimeServer which declares a public static nested interface named Client with void updateTime(LocalDateTime now).**
- **The server class should have method registerClient(Client client) and notifyClients() to pass current time.**
- **Implement at least two classes implementing Client, registering them, and simulate notifications.**

// 1. TimeServer Class with Nested Interface:

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

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

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

    public void registerClient(Client client) {
        if (client != null) {
            clients.add(client);
        }
    }

    public void notifyClients() {
        LocalDateTime currentTime = LocalDateTime.now();
        for (Client client : clients) {
            client.updateTime(currentTime);
        }
    }
}
// 2. Implementing Classes for Client:
// mobileclient
public class MobileClient implements TimeServer.Client {
    private String name;

    public MobileClient(String name) {
        this.name = name;
    }

    @Override
    public void updateTime(LocalDateTime now) {
        System.out.println("MobileClient [" + name + "] received time: " + now);
    }
}
//DestopClient
```

```
public class DesktopClient implements TimeServer.Client {
    private String id;

    public DesktopClient(String id) {
        this.id = id;
    }

    @Override
    public void updateTime(LocalDateTime now) {
        System.out.println("DesktopClient [" + id + "] updated to: " + now);
    }
}
// 3. Main Method to Register and Notify Clients:
public class Main {
    public static void main(String[] args) {
        TimeServer server = new TimeServer();

        TimeServer.Client client1 = new MobileClient("Alice");
        TimeServer.Client client2 = new DesktopClient("PC-42");

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

        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().**

// 1. Polygon Interface with Default and Static Methods:

```
public interface Polygon {
    double getArea();

    default double getPerimeter(int... sides) {
        double sum = 0;
        for (int side : sides) {
            sum += side;
        }
        return sum;
    }

    static String shapeInfo() {
        return "Polygons are 2D shapes with straight sides.";
    }
}
```

// 2. Rectangle Class Implements Polygon:

```
public class Rectangle implements Polygon {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public double getArea() {
        return length * width;
    }
}
```

// 3. Triangle Class Implements Polygon:

```
public class Triangle implements Polygon {
    private double base;
    private double height;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

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

// 4. Main Class to Test Perimeter and Static Method:

```
public class Main {
    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("Triangle perimeter: " + tri.getPerimeter(3, 4, 5));

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

Lambda expressions

1. **Sum of Two Integers**

```
import java.util.function.BiFunction;


public class Main {
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;


        int result = sum.apply(10, 20);
        System.out.println("Sum: " + result);
    }
}
```

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

```
// 1. Define the Functional Interface
@FunctionalInterface
public interface SumCalculator {
    int sum(int a, int b);
}


// 2. Use a Lambda Expression in Main
public class Main {
    public static void main(String[] args) {
        SumCalculator calculator = (a, b) -> a + b;


        int result = calculator.sum(15, 25);
        System.out.println("Sum: " + result);
    }
}
```

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();

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

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

        System.out.println("Check 1 (\"\"): " + isEmpty.test(""));
        System.out.println("Check 2 (\"hello\"): " + isEmpty.test("hello"));
    }
}
```

4. Filter Even or Odd Numbers

```
import java.util.*;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 15, 20, 25, 30, 35);

        Predicate<Integer> isEven = n -> n % 2 == 0;
        Predicate<Integer> isOdd = n -> n % 2 != 0;

        List<Integer> evenNumbers = numbers.stream()
                                           .filter(isEven)
                                           .collect(Collectors.toList());

        List<Integer> oddNumbers = numbers.stream()
```

```
                                          .filter(isOdd)
                                          .collect(Collectors.toList());

        System.out.println("Even Numbers: " + evenNumbers);
        System.out.println("Odd Numbers: " + oddNumbers);
    }
}
```