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).

Program:

```
class Student {
        private String Studentname;
        private int Studentrollno;
        private int Studentmarks;
        public Student(String Studentname,int Studentrollno,int Studentmarks)
        {
                this.Studentname=Studentname;
                this.Studentrollno=Studentrollno;
                if(Studentmarks>=0 && Studentmarks<=100)
                {
                        this.Studentmarks = Studentmarks;
                }
                else
                {
                        this.Studentmarks=0;
```

```
            }
        }
        public String getname() {
            return Studentname;
        }
        public int getrollno() {
            return Studentrollno;
        }
        public int getmarks() {
            return Studentmarks;
        }
        public void displayDetails() {
            System.out.println("Studentname:"+ Studentname);
            System.out.println("Studentrollno:"+ Studentrollno);
            System.out.println("Studentmarks:"+ Studentmarks);
        }
//for future versions
public void inputMarks(int newMarks)
{
 if (newMarks > Studentmarks)
    {
     if (newMarks >= 0 && newMarks <= 100)
     {
        this.Studentmarks = newMarks;
     }
     else
```

```
    {
        System.out.println("Invalid marks provided. Marks must be between 0 and
100.");
    }
 }
 else {
    System.out.println("Marks cannot be reduced.");
 }
}
}
public class Student_Encapsulation {
   public static void main(String[] args) {
        Student student1 = new Student("Jayanth", 1, 60);
        Student student2 = new Student("Srihari", 2, 72);
        Student student3 = new Student("Ramesh", 3, 65);
       System.out.println("1st Student Details:");
       student1.displayDetails();
       System.out.println("\n2nd Student Details:");
       student2.displayDetails();
       System.out.println("\n3rd Student Details:");
       student3.displayDetails();
       System.out.println("\nUpdating 3st Student marks:");
       student3.inputMarks(90);
       student3.displayDetails();
       student3.inputMarks(85);
       student3.displayDetails();
```

```
    }
}
```

Output=

```
1st Student Details:
Studentname: Jayanth
Studentrollno:1
Studentmarks:60
2nd Student Details:
Studentname: Srihari
Studentrollno:2
Studentmarks:72
3rd Student Details:
Studentname: Ramesh
Studentrollno:3
Studentmarks:65
Updating 3st Student marks:
Studentname: Ramesh
Studentrollno:3
Studentmarks:90
Marks cannot be reduced.
Studentname: Ramesh
Studentrollno:3
Studentmarks:90
```

---

2. Rectangle Enforced Positive Dimensions

Encapsulate validation and provide derived calculations.

- **Build a Rectangle class with private width and height.**
- **Constructor and setters should reject or correct non-positive values (e.g., use default or throw an exception).**
- **Provide getArea() and getPerimeter() methods.**
- **Include displayDetails() method.**

Program:

```
public class Rectangle {
    private float width;
    private float height;
    public Rectangle(float width, float height) {
       if (width > 0 && height > 0) {
            this.width = width;
            this.height = height;
       } else {
            System.out.println("Invalid dimensions.");
            this.width = 1;
            this.height = 1;
       }
   }
    public void setWidth(float width) {
       if (width > 0) {
            this.width = width;
       } else {
            System.out.println("Width must be positive.");
       }
    }
```

```
public void setHeight(float height) {
    if (height > 0) {
        this.height = height;
    } else {
        System.out.println("Height must be positive.");
    }
}
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(5, 2);
    r1.displayDetails();
    System.out.println();
    Rectangle r2 = new Rectangle(-2, 1);
    r2.displayDetails();
    System.out.println();
```

```
        r2.setWidth(4);
        r2.setHeight(-3);
        r2.displayDetails();
    }
}
```

Output=

Width: 5.0

Height: 2.0

Area: 10.0

Perimeter: 14.0

Invalid dimensions.

Width: 1.0

Height: 1.0

Area: 1.0

Perimeter: 4.0

Height must be positive.

Width: 4.0

Height: 1.0

Area: 4.0

Perimeter: 10.0

---

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.**

Program:

```
import java.util.ArrayList;
import java.util.List;
public class BankAccount_encapsulation {
    private String accountNumber;
    private String accountHolder;
    private double balance;
    private List<String> transactionHistory = new ArrayList<>();
    public BankAccount_encapsulation(String accountNumber, String accountHolder, double initialBalance) {
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
        this.balance = initialBalance;
    }
    public void deposit(double amount) {
        if (amount > 0) {
            balance =balance + amount;
            transactionHistory.add("Deposited: " + amount);
        } else {
```

```
            System.out.println("Deposit amount must be positive.");
        }
    }
    public boolean withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance = balance + amount;
            transactionHistory.add("Withdraw: " + amount);
            return true;
        } else {
            transactionHistory.add("Failed withdrawal: " + amount);
            return false;
        }
    }
    public double getBalance() {
        return balance;
    }
    public String getLastTransaction() {
        if (transactionHistory.isEmpty()) {
            return "No transactions yet.";
        }
        return transactionHistory.get(transactionHistory.size() - 1);
    }
    public String toString() {
        String maskedAccount = "****" +
accountNumber.substring(accountNumber.length() - 4);
        return "Account Holder: " + accountHolder + ", Account Number: " +
maskedAccount + ", Balance:" + balance;
```

```
    }
        public static void main(String[] args) {
            BankAccount_encapsulation account = new
BankAccount_encapsulation("1234567890", "Nikhitha", 5000);
            account.deposit(1500);
            account.withdraw(2000);
            account.withdraw(7000);
            System.out.println(account);
            System.out.println("Last Transaction: " + account.getLastTransaction());
            System.out.println("Balance: ₹" + account.getBalance());
        }
    }
```

Output

Account Holder: Nikhitha, Account Number: ****7890, Balance:15500.0

Last Transaction: Withdraw: 7000.0

Balance: ₹15500.0

---

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.**

Program:

```
public class Locker {
    private String lockerId;
    private boolean locked;
    private String passcode;
    private class SecurityManager {
        private boolean verify(String code) {
            return passcode.equals(code);
        }
    }
    public Locker(String lockerId, String passcode) {
        this.lockerId = lockerId;
        this.passcode = passcode;
        this.locked = true;
    }
    public void lock() {
        locked = true;
        System.out.println("Locker locked.");
    }
    public void unlock(String code) {
        SecurityManager sm = new SecurityManager();
        if (sm.verify(code)) {
            locked = false;
            System.out.println("Locker unlocked successfully.");
        } else {
            System.out.println("Incorrect passcode. Access denied.");
```

```
        }
    }
    public boolean isLocked() {
        return locked;
    }
    public void displayStatus() {
        System.out.println("Locker ID: " + lockerId);
        System.out.println("Status: " + (locked? "Locked" : "Unlocked"));
    }
    public static void main(String[] args) {
        Locker myLocker = new Locker("L123", "g134");
        myLocker.displayStatus();
        myLocker.unlock("0000");
        myLocker.unlock("g134");
        myLocker.displayStatus();
        myLocker.lock();
        myLocker.displayStatus();
    }
}
```

Output=

Locker ID: L123

Status: Locked

Incorrect passcode. Access denied.

Locker unlocked successfully.

Locker ID: L123

Status: Unlocked

Locker locked.

Locker ID: L123

Status: 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.**

Program:

```
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.isEmpty()) {
                throw new IllegalArgumentException("Product name cannot be
empty.");
            }
            this.name = name;
            return this;
        }
        public Builder withCode(String code) {
            if (code == null || code.isEmpty()) {
```

```
            throw new IllegalArgumentException("Product code cannot be
empty.");
        }
        this.code = code;
        return this;
    }
    public Builder withPrice(double price) {
        if (price < 0) {
            throw new IllegalArgumentException("Price must be non-
negative.");
        }
        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("Missing required fields or
invalid values.");
        }
        return new Product(this);
    }
  }
  public void displayDetails() {
```

```
        System.out.println("Product Name: " + name);
        System.out.println("Code: " + code);
        System.out.println("Price: ₹" + price);
        System.out.println("Category: " + (category != null ? category :
"N/A"));
    }
    public static void main(String[] args) {
        Product p = new Product.Builder()
                .withName("tablet")
                .withCode("3763d")
                .withPrice(596478.99)
                .withCategory("Electronics")
                .build();
        p.displayDetails();
    }
}
```

Output=

Product Name: tablet

Code: 3763d

Price: ₹596478.99

Category: Electronics

Interface

1. Reverse CharSequence: Custom BackwardSequence

- **Create a class BackwardSequence that implements java.lang.CharSequence.**
- **Internally store a String and implement all required methods: length(), charAt(), subSequence(), and toString().**

- **The sequence should be the reverse of the stored string (e.g., new BackwardSequence("hello") yields "olleh").**
- **Write a main() method to test each method.**

Program:

```
public class BackwardSequence_interface implements CharSequence {
        private String reversed;
        public BackwardSequence_interface(String input) {
                this.reversed=new StringBuilder(input).reverse().toString();
        }
        public int length() {
                return 0;
        }
        public char charAt(int index) {
                return 0;
        }
        public CharSequence subSequence(int start, int end) {
                return null;
        }
    public String toString() {
       return reversed;
    }
public static void main(String[] args) {
        BackwardSequence_interface seq = new
BackwardSequence_interface("hello");
                System.out.println("Full reversed string: " + seq);
                System.out.println("Length: " + seq.length());
                System.out.println("Character at index 1: " + seq.charAt(1));
                System.out.println("Subsequence (1, 4): " + seq.subSequence(1, 4));
```

```
        }
    }
```

Output=

Full reversed string: olleh

Length: 0

Character at index 1:

Subsequence (1, 4): null

---

2. Moveable Shapes Simulation

- **Define an interface Movable with methods: moveUp(), moveDown(), moveLeft(), moveRight().**
- **Implement classes:**
  - **MovablePoint(x, y, xSpeed, ySpeed) implements Movable**
  - **MovableCircle(radius, center: MovablePoint)**
  - **MovableRectangle(topLeft: MovablePoint, bottomRight: MovablePoint) (ensuring both points have same speed)**
- **Provide toString() to display positions.**
- **In main(), create a few objects and call move methods to simulate motion.**

Program:

```
interface 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 PrinterSwitchDemo {
 public static void main(String[] args) {
    Printer p;
    p = new LaserPrinter();
    p.print("Java Interface Documentation");
    p = new InkjetPrinter();
    p.print("Java Patterns Notes");
 }
}
```

Output:

LaserPrinter is printing: Java Interface Documentation

InkjetPrinter is printing: Java Patterns Notes

---

4. Extended Interface Hierarchy

- **Define interface BaseVehicle with method void start().**
- **Define interface AdvancedVehicle that extends BaseVehicle, adding method void stop() and boolean refuel(int amount).**
- **Implement Car to satisfy both interfaces; include a constructor initializing fuel level.**
- **In Main, manipulate the object via both interface types**

Program:

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

```
}
interface AdvancedVehicle extends BaseVehicle {
 void stop();
 boolean refuel(int amount);
}
class Car implements AdvancedVehicle {
 private int fuel;
 public Car(int initialFuel) {
    this.fuel = initialFuel;
 }
 public void start() {
    if (fuel > 0) {
       System.out.println(" Fuel level: " + fuel + "L");
    } else {
       System.out.println("No fuel.");
    }
 }
 public void stop() {
    System.out.println("Car stopped.");
 }
 public boolean refuel(int amount) {
    if (amount > 0) {
       fuel =fuel + amount;
       System.out.println("Refueled " + amount + "L. Total fuel: " + fuel + "L");
       return true;
    } else {
       System.out.println("Invalid fuel amount.");
```

```
        return false;
    }
 }
}
public class VehicleTest {
 public static void main(String[] args) {
    BaseVehicle bv = new Car(5);
    bv.start();

    AdvancedVehicle av = (AdvancedVehicle) bv;
    av.stop();
    av.refuel(10);
    av.start();
 }
}
```

Output=

Fuel level: 5L

Car stopped.

Refueled 10L. Total fuel: 15L

Fuel level: 15L

---

5. Nested Interface for Callback Handling

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

Program:

```
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
class TimeServer {
    public static interface Client {
        void updateTime(LocalDateTime now);
    }
    private List<Client> clients = new ArrayList<>();
    public void registerClient(Client client) {
        clients.add(client);
    }
    public void notifyClients() {
        LocalDateTime now = LocalDateTime.now();
        for (Client client : clients) {
            client.updateTime(now);
        }
    }
}
class DigitalClock implements TimeServer.Client {
    private String name;
    public DigitalClock(String name) {
        this.name = name;
    }
    public void updateTime(LocalDateTime now) {
        System.out.println(name + " shows time: " + now);
    }
```

```
}
class Logger implements TimeServer.Client {
    public void updateTime(LocalDateTime now) {
        System.out.println("Logger recorded time: " + now);
    }
}
public class Timeserver_demo {
    public static void main(String[] args) {
        TimeServer server = new TimeServer();
        DigitalClock clock1 = new DigitalClock("Office Clock");
        Logger logger = new Logger();
        server.registerClient(clock1);
        server.registerClient(logger);
        server.notifyClients();
    }
}
```

Output:

Office Clock shows time: 2025-08-09T20:49:01.496452600

Logger recorded time: 2025-08-09T20:49:01.496452600

---

6. Default and Static Methods in Interfaces

- **Declare interface Polygon with:**
    - **double getArea()**
    - **default method default double getPerimeter(int... sides) that computes sum of sides**
    - **a static helper static String shapeInfo() returning a description string**

- **Implement classes Rectangle and Triangle, providing appropriate getArea().**
- **In Main, call getPerimeter(...) and Polygon.shapeInfo().**

Program:

```
interface Polygon {
    double getArea();
    default double getPerimeter(int... sides) {
        double sum = 0;
        for (int side : sides) {
            sum += side;
        }
        return sum;
    }
    static String shapeInformation() {
        return "Polygons have area and perimeter.";
    }
}
class Rectangle implements Polygon {
    private double length;
    private double width;
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    public double getArea() {
        return length * width;
    }
}
```

```
class Triangle implements Polygon {
    private double base;
    private double height;
    Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }
    public double getArea() {
        return 0.5 * base * height;
    }
}
public class Area_perimeter {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(5, 4);
        Triangle tri = new Triangle(3, 6);
        System.out.println("Rectangle Area: " + rect.getArea());
        System.out.println("Rectangle Perimeter: " + rect.getPerimeter(5, 4, 5,
4));
        System.out.println("Triangle Area: " + tri.getArea());
        System.out.println("Triangle Perimeter: " + tri.getPerimeter(3, 4, 5));
        System.out.println(Polygon.shapeInfo());
    }
}
```

Output:

Rectangle Area: 20.0

Rectangle Perimeter: 18.0

Triangle Area: 9.0

Triangle Perimeter: 12.0

Polygons have area and perimeter

Lambda expressions

1. Sum of Two Integers

Program:

```
interface SumCalculator {
    int sum(int a, int b);
}
public class Sum {
    public static void main(String[] args) {
        SumCalculator add = (a, b) -> a + b;
        int result = add.sum(10, 20);
        System.out.println("Sum: " + result);
    }
}
```

Output: 30

---

2. Check If a String Is Empty (Predicate Lambda)

Program:

```
import java.util.function.Predicate;
public class EmptyString {
    public static void main(String[] args) {
        Predicate<String> isEmpty = s -> s.isEmpty();
        String s1 = "";
        String s2 = "Hello";
        System.out.println("" + s1 + " is empty? " + isEmpty.test(s1));
        System.out.println("" + s2 + " is empty? " + isEmpty.test(s2));
```

```
    }
}
```

Output:

is empty? true

Hello is empty? false

3.Filter Even or Odd Numbers

Program:

```
import java.util.List;
public class FilterEvenOddLambda {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(7,4,6,3,8,9,1);
        System.out.println("Even numbers:");
        numbers.forEach(n -> { if (n % 2 == 0) System.out.println(n); });
        System.out.println("Odd numbers:");
        numbers.forEach(n -> { if (n % 2 != 0) System.out.println(n); });
    }
}
```

Output:

```
Even numbers:
2
4
6
Odd numbers:
1
3
5
```

5.Convert Strings to Uppercase/Lowercase

Program

```
import java.util.function.Function;
public class StringCaseLambda {
    public static void main(String[] args) {
        String text = "Hello";
       //uppercase
        Function<String, String> toUpper = s -> s.toUpperCase();
        System.out.println("Uppercase: " + toUpper.apply(text));
      //lowercase
        Function<String, String> toLower = s -> s.toLowerCase();
        System.out.println("Lowercase: " + toLower.apply(text));
    }
}
```

Output:

```
Uppercase: HELLO
Lowercase: hello
```

---

6. Sort Strings by Length or Alphabetically

Program:

```
import java.util.Arrays;
import java.util.List;
public class length_alphabetical_Lambda {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("dog", "horse", "elephant", "camel");
        // Sort by length
        words.sort((a, b) -> Integer.compare(a.length(), b.length()));
```

```
        System.out.println("Sorted by length: " + words);
        // Sort alphabetically
        words.sort((a, b) -> a.compareTo(b));
        System.out.println("Sorted alphabetically: " + words);
    }
}
```

Output:

Sorted by length: [dog, horse, camel, elephant]

Sorted alphabetically: [camel, dog, elephant, horse]

---

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

Program:

```
import java.util.Arrays;
public class Aggregate_operations {
    public static void main(String[] args) {
        double[] numbers = {6.8,3.2,1.5,3.4};
        double sum = Arrays.stream(numbers).sum();
        double max = Arrays.stream(numbers).max().getAsDouble();
        double average = Arrays.stream(numbers).average().getAsDouble();
        System.out.println("Sum: " + sum);
        System.out.println("Max: " + max);
        System.out.println("Average: " + average);
    }
}
```

Output:

Sum: 14.9

Max: 6.8

Average: 3.725

---

7. Max and Min Using Lambda

Program:

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
public class Max_Min_Lambda {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(4,8,5,2,9);
        int max = numbers.stream().max((a, b) -> a - b).get();
        int min = numbers.stream().min((a, b) -> a - b).get();
        System.out.println("Max: " + max);
        System.out.println("Min: " + min);
    }
}
```

Output:

Max=9

Min=2

---

2. Calculate Factorial

Program

```
interface FactorialCalculator {
    int factorial(int n);
}
public class FactorialLambda {
    public static void main(String[] args) {
```

```
        // Lambda expression to calculate factorial
        FactorialCalculator fact = (n) -> {
            int result = 1;
            for (int i = 1; i <= n; i++) {
                result *= i;
            }
            return result;
        };
        int num = 5;
        System.out.println("Factorial of " + num + " is: " + fact.factorial(num));
    }
}
```

Output:

Factorial of 5 is: 120