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.

```
class Student1 {
 private String name;
 private int rollNumber;
 private int marks;

 public Student1(String name, int rollNumber, int marks) {
     this.name = name;
     this.rollNumber = rollNumber;
     if (isvalidmarks(marks)) {
        this.marks = marks;
    } else {
        this.marks = 0;
    }
}

private boolean isvalidmarks(int marks) {
    return marks >= 0 && marks <= 100;
}

public String getName() {
    return name;
}

public int getRollNumber() {
    return rollNumber;
}

public int getMarks() {
    return marks;
}

public void displayDetails() {
    System.out.println("Student Name: " + name);
```

```
     System.out.println("Roll Number : " + rollNumber);
     System.out.println("Marks       : " + marks);
 }

 public void inputMarks(int newMarks) {
     if (isvalidmarks(newMarks) && newMarks > this.marks) {
         this.marks = newMarks;
     } else {
         System.out.println("Invalid or lower marks. Update rejected.");
     }
 }
}


public class Student {

          public static void main(String[] args) {
                    // TODO Auto-generated method stub
                    Student1 s1 = new Student1("Alice", 101, 85);
        s1.displayDetails();

        System.out.println();

        Student1 s2 = new Student1("Bob", 102, 150);
        s2.displayDetails();

        System.out.println();

        s1.inputMarks(90);
        s1.displayDetails();

        System.out.println();

        s1.inputMarks(80);
        s1.displayDetails();
          }
}
```

Output:

```
Student Name: Alice
Roll Number : 101
Marks       : 85
Student Name: Bob
Roll Number : 102
Marks       : 0
Student Name: Alice
Roll Number : 101
Marks       : 90
Invalid or lower marks. Update rejected.
```

Student Name: Alice

Roll Number : 101

Marks : 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.

```
class Rectangle1 {
   private double width;
   private double height;
   public Rectangle1(double width, double height) {
      setWidth(width);
      setHeight(height);
   }
   public void setWidth(double width) {
      if (width > 0) {
         this.width = width;
      } else {
        System.out.println("Invalid width. Default value is 1.0.");
        this.width = 1.0;
    }
}
public void setHeight(double height) {
    if (height > 0) {
        this.height = height;
    } else {
        System.out.println("Invalid height. Default value is 1.0.");
        this.height = 1.0;
    }
}
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("Rectangle Details:");
        System.out.println("Width     : " + width);
        System.out.println("Height    : " + height);
        System.out.println("Area      : " + getArea());
        System.out.println("Perimeter : " + getPerimeter());
    }
}
public class Rectangle {
          public static void main(String[] args) {
                    Rectangle1 r1 = new Rectangle1(5.0, 3.0);
        r1.displayDetails();

        System.out.println();

        Rectangle1 r2 = new Rectangle1(-4.0, 0);
        r2.displayDetails();

        System.out.println();

        r2.setWidth(7.5);
        r2.setHeight(4.5);
        r2.displayDetails();
          }
}
```

Output:

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

Invalid width. Default value is 1.0.
Invalid height. Default value is 1.0.
Rectangle Details:
Width     : 1.0
Height    : 1.0
Area      : 1.0
Perimeter : 4.0

Rectangle Details:
Width     : 7.5
Height    : 4.5
Area      : 33.75
Perimeter : 24.0
```

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.

```
import java.util.ArrayList;

import java.util.List;

public class BankAccount {

    private String accountNumber;

    private String accountHolder;

    private double balance;

    private class Transaction {

        private String type; // "Deposit" or "Withdraw"

        private double amount;
        public Transaction(String type, double amount) {
            this.type = type;
            this.amount = amount;
        }
        public String toString() {
            return type + ": $" + String.format("%.2f", amount);
        }
    }

    private List<Transaction> transactionHistory;
    public BankAccount(String accountNumber, String accountHolder, double initialBalance) {
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
        this.balance = initialBalance >= 0 ? initialBalance : 0;
        this.transactionHistory = new ArrayList<>();
        if (initialBalance > 0) {
            transactionHistory.add(new Transaction("Initial Balance", initialBalance));
        }
    }
    public boolean deposit(double amount) {
        if (amount <= 0) {
            System.out.println("Deposit amount must be positive.");
            return false;
        }
        balance += amount;
        transactionHistory.add(new Transaction("Deposit",
amount));
        return true;
    }
    public boolean withdraw(double amount) {
        if (amount <= 0) {
            System.out.println("Withdraw amount must be
positive.");
            return false;
        }
        if (amount > balance) {
            System.out.println("Insufficient funds. Withdrawal
denied.");
            return false;
        }
        balance-= amount;
        transactionHistory.add(new Transaction("Withdraw",
amount));
        return true;
    }
    public double getBalance() {
        return balance;
    }



    private String getMaskedAccountNumber() {
        int len = accountNumber.length();
        if (len <= 4) {
            return accountNumber; // Not enough digits to mask
        }
        String masked = "*".repeat(len- 4) +
accountNumber.substring(len- 4);
        return masked;
    }
    public String toString() {
        return "Account Holder: " + accountHolder +
               "\nAccount Number: " + getMaskedAccountNumber() +
               "\nBalance: $" + String.format("%.2f", balance);
    }

    public String getLastTransaction() {
        if (transactionHistory.isEmpty()) return null;
        return transactionHistory.get(transactionHistory.size()-
1).toString();
    }
}
```

```
}
```

```
public class BankAccountDemo {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount("1234567890123456", "ajay", 500);
        System.out.println(acc);

        System.out.println("\nDeposit $200:");
        acc.deposit(200);
        System.out.println(acc);

        System.out.println("\nWithdraw $100:");
        acc.withdraw(100);
        System.out.println(acc);

        System.out.println("\nAttempt to withdraw $700 (should fail):");
        acc.withdraw(700);
        System.out.println(acc);

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

Output:

```
Account Holder: ajay
Account Number: ************3456
Balance: 500.00

Deposit 200
Account Holder: ajay
Account Number: ************3456
Balance: 700.00

Withdraw 100
Account Holder: ajay
Account Number: ************3456
Balance: 600.00

Attempt to withdraw 700 (should fail):
Insufficient funds. Withdrawal denied.
Account Holder: ajay
Account Number: ************3456
Balance: 600.00

Last Transaction: Withdraw: 100.00
```

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 class SecurityManager {

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

    public Locker(String lockerId, String passcode) {
        this.lockerId = lockerId;
        this.passcode = passcode;
        this.isLocked = false;
        this.securityManager = new SecurityManager();
    }
    public void lock() {
        isLocked = true;
        System.out.println("Locker " + lockerId + " is now locked.");
    }
    public boolean unlock(String code) {
        boolean success = securityManager.verify(code);
        if (success) {
            isLocked = false;
            System.out.println("Locker " + lockerId + " unlocked successfully.");
        } else {
            System.out.println("Failed to unlock locker " + lockerId + ". Incorrect passcode.");
        }
        return success;
    }

    public boolean isLocked() {
        return isLocked;
    }
}

public class LockerDemo {
    public static void main(String[] args) {
        Locker locker = new Locker("LKR123", "secure123");

        locker.lock();

        System.out.println("Is locker locked? " + locker.isLocked());

        System.out.println("\nAttempt to unlock with wrong
code:");
        locker.unlock("wrongpass");

        System.out.println("Is locker locked? " + locker.isLocked());

        System.out.println("\nAttempt to unlock with correct
code:");
        locker.unlock("secure123");
        System.out.println("Is locker locked? " + locker.isLocked());
    }
}
```

Output:

```
Locker LKR123 is now locked.
Is locker locked? true

Attempt to unlock with wrong code:
Failed to unlock locker LKR123. Incorrect passcode.
Is locker locked? true

Attempt to unlock with correct code:
Locker LKR123 unlocked successfully.
Is locker locked? false
```

Interface

1. 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.

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

class MovablePoint implements Movable {
   int x, y;
   int xSpeed, ySpeed;

   public MovablePoint(int x, int y, int xSpeed, int ySpeed) {
      this.x = x;
      this.y = y;
      this.xSpeed = xSpeed;
      this.ySpeed = ySpeed;
```

```java
    }

    public void moveUp() {
        y += ySpeed;
    }
    public void moveDown() {
        y-= ySpeed;
    }
    public void moveLeft() {
        x-= xSpeed;
    }
    public void moveRight() {
        x += xSpeed;
    }
    public String toString() {
        return "Point(" + x + ", " + y + "), speed=(" + xSpeed + ", " + ySpeed + ")";
    }
}

class MovableCircle implements Movable {
    private int radius;
    private 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 with radius " + radius + " at center " + center;
    }
}

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;
    }
    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 [Top Left: " + topLeft + ", Bottom Right: " +
bottomRight + "]";
    }
}
public class MovableShapesDemo {
    public static void main(String[] args) {
        MovablePoint point = new MovablePoint(0, 0, 2, 3);
        MovableCircle circle = new MovableCircle(5, new
MovablePoint(10, 10, 1, 1));
        MovablePoint rectTopLeft = new MovablePoint(5, 5, 2, 2);
        MovablePoint rectBottomRight = new MovablePoint(15, 1,
2, 2);
        MovableRectangle rectangle = new
MovableRectangle(rectTopLeft, rectBottomRight);

        System.out.println("Initial positions:");
        System.out.println(point);
        System.out.println(circle);
        System.out.println(rectangle);

        System.out.println("\nMoving up:");
        point.moveUp();
        circle.moveUp();
        rectangle.moveUp();

        System.out.println(point);
        System.out.println(circle);
        System.out.println(rectangle);

        System.out.println("\nMoving right:");
        point.moveRight();
        circle.moveRight();
        rectangle.moveRight();

        System.out.println(point);
        System.out.println(circle);
        System.out.println(rectangle);
    }
}
```

Outptut:

```
Initial positions:
Point(0, 0), speed=(2, 3)
Circle with radius 5 at center Point(10, 10), speed=(1, 1)
```

Rectangle [Top Left: Point(5, 5), speed=(2, 2), Bottom Right: Point(15, 1), speed=(2, 2)]

Moving up:

Point(0, 3), speed=(2, 3)

Circle with radius 5 at center Point(10, 11), speed=(1, 1)

Rectangle [Top Left: Point(5, 7), speed=(2, 2), Bottom Right: Point(15, 3), speed=(2, 2)]

Moving right:

Point(2, 3), speed=(2, 3)

Circle with radius 5 at center Point(11, 11), speed=(1, 1)

Rectangle [Top Left: Point(7, 7), speed=(2, 2), Bottom Right: Point(17, 3), speed=(2, 2)]

2. 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.

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

class LaserPrinter implements Printer {
    public void print(String document) {
        System.out.println("LaserPrinter is printing the document: " + document);
        System.out.println("Output is crisp and fast.");
    }
}
class InkjetPrinter implements Printer {
    public void print(String document) {
        System.out.println("InkjetPrinter is printing the document: " + document);
        System.out.println("Output has rich colors but takes longer.");
    }
}
public class PrinterDemo {
    public static void main(String[] args) {
        Printer p;
        p = new LaserPrinter();
        p.print("Annual Report.pdf");
        System.out.println();
        p = new InkjetPrinter();
        p.print("Family Photo.jpg");
    }
}
```

Output:

```
LaserPrinter is printing the document: Annual Report.pdf
Output is crisp and fast.

InkjetPrinter is printing the document: Family Photo.jpg
Output has rich colors but takes longer.
```

3. 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 BaseVehicle {
    void start();
}

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

class Car implements AdvancedVehicle {
    private int fuelLevel;

    public Car(int initialFuel) {
        this.fuelLevel = initialFuel;
    }
    public void start() {
        if (fuelLevel > 0) {
            System.out.println("Car started with fuel level: " +
fuelLevel);
        } else {
            System.out.println("Cannot start. Fuel tank is empty.");
        }
    }
    public void stop() {
        System.out.println("Car stopped.");
    }
    public boolean refuel(int amount) {
        if (amount <= 0) {
            System.out.println("Refuel amount must be positive.");
            return false;
        }
        fuelLevel += amount;
        System.out.println("Car refueled by " + amount + " units.
Current fuel: " + fuelLevel);
        return true;
    }
    public int getFuelLevel() {
        return fuelLevel;
    }
}
public class VehicleDemo {
    public static void main(String[] args) {
        Car myCar = new Car(10);


        BaseVehicle baseVehicleRef = myCar;
        baseVehicleRef.start();
        System.out.println();
        AdvancedVehicle advancedVehicleRef = myCar;
        advancedVehicleRef.start();
        advancedVehicleRef.stop();
        advancedVehicleRef.refuel(20);

        System.out.println("Fuel after refuel: " +
myCar.getFuelLevel());
    }
}
```

Output:

```
Car started with fuel level: 10


Car started with fuel level: 10
Car stopped.
Car refueled by 20 units. Current fuel: 30
Fuel after refuel: 30
```

4. 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().

```
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.";
    }
}
class Rectangle implements Polygon {
    private double length;
    private double width;

    public 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;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }
    public double getArea() {
        return 0.5 * base * height;
    }
}
public class PolygonDemo {
    public static void main(String[] args) {
        Polygon rect = new Rectangle(10, 5);
        Polygon tri = new Triangle(6, 4);

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

        System.out.println("Rectangle area: " + rect.getArea());
        System.out.println("Rectangle perimeter: " +
rect.getPerimeter(10, 5, 10, 5));

        System.out.println();

        System.out.println("Triangle area: " + tri.getArea());
        System.out.println("Triangle perimeter: " +
tri.getPerimeter(6, 7, 8));
    }
}
```

Output:

```
Polygons are 2D shapes with straight sides.

Rectangle area: 50.0
Rectangle perimeter: 30.0

Triangle area: 12.0
Triangle perimeter: 21.0
```

Lambda expressions

1. Sum of Two Integers

```
interface Sum {
    int add(int a, int b);
}

public class LambdaSumDemo {
    public static void main(String[] args) {
        Sum sum = (a, b)-> a + b;

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

Output:

```
Sum: 30
```

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

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

public class SumCalculatorDemo {
    public static void main(String[] args) {
        SumCalculator calculator = (a, b)-> a + b;

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

Output:

```
Sum: 40
```

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

        String test1 = "";
        String test2 = "Hello";

        System.out.println("Is test1 empty? " + isEmpty.test(test1));
        System.out.println("Is test2 empty? " + isEmpty.test(test2));
    }
}
```

Output:

```
Is test1 empty? true
Is test2 empty? False
```

4. Filter Even or Odd Numbers

```
import java.util.*;
import java.util.function.Predicate;
import java.util.stream.Collectors;
public class EvenOddFilter {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 15, 20, 25, 30, 35,
40);
        List<Integer> evenNumbers = numbers.stream()
            .filter(n-> n % 2 == 0)
            .collect(Collectors.toList());
        List<Integer> oddNumbers = numbers.stream()
            .filter(n-> n % 2 != 0)
            .collect(Collectors.toList());

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

Output:

```
Even Numbers: [10, 20, 30, 40]
Odd Numbers: [15, 25, 35]
```

5. Convert Strings to Uppercase/Lowercase

```
import java.util.function.Function;
public class StringCaseConverter {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("hello", "World", "Java",
"Lambda");

        Function<String, String> toUpperCase = s-> s.toUpperCase();
   Function<String, String> toLowerCase = s-> s.toLowerCase();

        List<String> uppercased = words.stream()
            .map(toUpperCase)
            .collect(Collectors.toList());

        List<String> lowercased = words.stream()
            .map(toLowerCase)
            .collect(Collectors.toList());

        System.out.println("Uppercase: " + uppercased);
        System.out.println("Lowercase: " + lowercased);
    }
}
```

```
Output:
Uppercase: [HELLO, WORLD, JAVA, LAMBDA]
Lowercase: [hello, world, java, lambda]
```

6. Sort Strings by Length or Alphabetically

```
import java.util.*;
public class SortStrings {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList("apple", "banana", "cherry", 
"date", "fig", "grape");
        List<String> sortedByLength = new ArrayList<>(strings);
        sortedByLength.sort((s1, s2)-> Integer.compare(s1.length(), 
s2.length()));
        System.out.println("Sorted by length: " + sortedByLength);

        List<String> sortedAlphabetically = new ArrayList<>(strings);
        sortedAlphabetically.sort((s1, s2)-> s1.compareTo(s2));
        System.out.println("Sorted alphabetically: " + 
sortedAlphabetically);
    }
}
```

```
Output:
Sorted by length: [fig, date, apple, grape, banana, cherry]
Sorted alphabetically: [apple, banana, cherry, date, fig, grape]
```

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

```
import java.util.Arrays;
public class AggregateOperations {
    public static void main(String[] args) {
        double[] numbers = {3.5, 2.1, 7.4, 1.6, 9.0};
        double sum = Arrays.stream(numbers)
                           .sum();
        System.out.println("Sum: " + sum);



        double max = Arrays.stream(numbers)
                           .max()
                           .orElse(Double.NaN);  // Handle empty array case
        System.out.println("Max: " + max);

        double avg = Arrays.stream(numbers)
                           .average()
                           .orElse(Double.NaN);  // Handle empty array case
        System.out.println("Average: " + avg);
    }
}
```

Output:

```
Sum: 23.6
```

```
Max: 9.0
Average: 4.72
```

8. Create similar lambdas for max/min.

```
import java.util.Arrays;
public class MaxMinWithLambdas {
    public static void main(String[] args) {
        double[] numbers = {3.5, 2.1, 7.4, 1.6, 9.0};

        double max = Arrays.stream(numbers)
                        .reduce((a, b)-> a > b ? a : b)
                        .orElse(Double.NaN);
        System.out.println("Max (reduce): " + max);

        double min = Arrays.stream(numbers)
                        .reduce((a, b)-> a < b ? a : b)
                        .orElse(Double.NaN);
        System.out.println("Min (reduce): " + min);
    }
}
```

Output:

```
Max (reduce): 9.0
Min (reduce): 1.6
```

9. Calculate Factorial

```
import java.util.stream.IntStream;
public class FactorialLambda {
    public static void main(String[] args) {
        int n = 5;

        long factorial = IntStream.rangeClosed(1, n)
                                  .reduce(1, (a, b)-> a * b);

        System.out.println(n + "! = " + factorial);
    }
}
```

Output:

```
5! = 120
```