Encapsulation

1. Student with Grade Validation & Configuration

Ensure marks are always valid and immutable once set.

- Create a Student class with private fields: name, rollNumber, and marks.
- Use a constructor to initialize all values and enforce marks to be between 0 and 100; invalid values reset to 0.
- Provide getter methods, but no setter for marks (immutable after object creation).
- Add displayDetails() to print all fields.

In future versions, you might allow updating marks only via a special inputMarks(int newMarks) method that has stricter logic (e.g. cannot reduce marks). Design accordingly.

```
package day5_Assessment;
public class Encapsulation_Student_Class {
   private String name;
   private int rollNumber;
   private int marks;

   public Encapsulation_Student_Class(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 inputMarks(int newMarks) {
        if (newMarks >= 0 && newMarks <= 100 && newMarks > this.marks) {
            this.marks = newMarks;
        }
    }

    public void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Roll Number: " + rollNumber);
        System.out.println("Marks: " + marks);
    }

    public static void main(String[] args) {
            Encapsulation_Student_Class s1 = new Encapsulation_Student_Class("Manasa", 101, 85);
        s1.displayDetails();

        s1.inputMarks(70);
        System.out.println("After trying to reduce marks:");
        s1.displayDetails();

        s1.inputMarks(90);
        System.out.println("After increasing marks:");
        s1.displayDetails();
    }
}
```

Output:

```
Name: Manasa
Roll Number: 101
Marks: 85
After trying to reduce marks:
Name: Manasa
Roll Number: 101
Marks: 85
After increasing marks:
Name: Manasa
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.

```
/*
2. Rectangle Enforced Positive Dimensions
Encapsulate validation and provide derived calculations.
•	Build a Rectangle class with private width and height.
•	Constructor and setters should reject or correct non-positive values (e.g., use default or throw an exception).
•	Provide getArea() and getPerimeter() methods.
•	Include displayDetails() method.
*/
package day5_Assessment;
public class MoveableShapesSimulation {
    private double width;
    private double height;
    public MoveableShapesSimulation(double width, double height) {
        if (width > 0 && height > 0) {
            this.width = width;
            this.height = height;
        } else {
            this.width = 1.0;
            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());
    }
    public static void main(String[] args) {
        MoveableShapesSimulation r1 = new MoveableShapesSimulation(5.0, 3.0);
        r1.displayDetails();

        r1.setWidth(-2.0);
        r1.setHeight(4.0);
        System.out.println("After trying to update dimensions:");
        r1.displayDetails();
```

```
    }
}
```
Output:

Width: 5.0

Height: 3.0

Area: 15.0

Perimeter: 16.0

After trying to update dimensions:

Width: 5.0

Height: 4.0

Area: 20.0

Perimeter: 18.0

---

3. Advanced: Bank Account with Deposit/Withdraw Logic

Transaction validation and encapsulation protection.

- Create a BankAccount class with private accountNumber, accountHolder, balance.
- Provide:
  - deposit(double amount) — ignores or rejects negative.
  - withdraw(double amount) — prevents overdraft and returns a boolean success.
  - Getter for balance but no setter.
- Optionally override toString() to display masked account number and details.
- Track transaction history internally using a private list (or inner class for transaction object).

- Expose a method getLastTransaction() but do not expose the full internal list.

```
package day5_Assessment;

import java.util.ArrayList;

import java.util.List;

class BankAccount {

    private String accountNumber;

    private String accountHolder;

    private double balance;

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

    public BankAccount(String accountNumber, String accountHolder,
double initialDeposit) {

        this.accountNumber = accountNumber;

        this.accountHolder = accountHolder;

        this.balance = Math.max(0, initialDeposit);

        if (initialDeposit > 0) {

            transactionHistory.add(new Transaction("Initial Deposit",
initialDeposit));

        }

    }

    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.";
        }
        return transactionHistory.get(transactionHistory.size() -
1).toString();
    }
    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;
        private long timestamp;

        public Transaction(String type, double amount) {
            this.type = type;
            this.amount = amount;
            this.timestamp = System.currentTimeMillis();
        }
        public String toString() {
            return type + " of ₹" + amount + " at " + new
java.util.Date(timestamp);
        }
    }
}
public class BankDemo {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount("1234567890123456",
"Manasa", 5000);
        acc.deposit(1500);
        acc.withdraw(2000);
        acc.withdraw(6000);
```

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

Output:

Account Holder: Manasa

Account Number: ****3456

Balance: ₹4500.0

Last Transaction: Withdraw of ₹2000.0 at Sun Aug 10 10:53:29 IST 2025

Current Balance: ₹4500.0

4. Inner Class Encapsulation: Secure Locker

Encapsulate helper logic inside the class.

- Implement a class Locker with private fields such as lockerId, isLocked, and passcode.
- Use an inner private class SecurityManager to handle passcode verification logic.
- Only expose public methods: lock(), unlock(String code), isLocked().
- Password attempts should not leak verification logic externally—only success/failure.
- Ensure no direct access to passcode or the inner SecurityManager from outside.

```
package day5_Assessment;
public class Locker {
```

```
    private final String lockerId;
    private boolean isLocked;
    private final String passcode;

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

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

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

    public boolean isLocked() {
        return isLocked;
    }

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

    public static void main(String[] args) {
        Locker locker = new Locker("L123", "secure123");

        System.out.println("Is Locked? " + locker.isLocked());
        locker.unlock("wrong123");
        System.out.println("Is Locked? " + locker.isLocked());
        locker.unlock("secure123");
        System.out.println("Is Locked? " + locker.isLocked());
        locker.lock();
        System.out.println("Is Locked? " + locker.isLocked());
    }
}
```

Output:

```
Is Locked? true
Incorrect passcode. Locker remains locked.
Is Locked? true
Locker L123 unlocked successfully.
Is Locked? false
Locker L123 is now locked.
Is Locked? true
```

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.

```
package day5_Assessment;
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("Name cannot be null or empty");
            }
            this.name = name;
            return this;
        }
```

```
        public Builder withCode(String code) {
            if (code == null || code.isEmpty()) {
                throw new IllegalArgumentException("Code cannot be null or empty");
            }
            this.code = code;
            return this;
        }
        public Builder withPrice(double price) {
            if (price < 0) {
                throw new IllegalArgumentException("Price cannot be negative");
            }
            this.price = price;
            return this;
        }
        public Builder withCategory(String category) {
            this.category = category;
            return this;
        }
        public Product build() {
            if (name == null || code == null) {
                throw new IllegalStateException("Name and code are required");
            }
            return new Product(this);
        }
    }
    public static void main(String[] args) {
        Product product = new Product.Builder()
                .withName("Laptop")
                .withCode("LP123")
                .withPrice(55000.0)
                .withCategory("Electronics")
                .build();
        System.out.println("Product Details:");
        System.out.println("Name: " + product.getName());
        System.out.println("Code: " + product.getCode());
        System.out.println("Price: " + product.getPrice());
        System.out.println("Category: " + product.getCategory());
    }
}
```

Output:

```
Product Details:
Name: Laptop
Code: LP123
Price: 55000.0
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.

```
package day5_Assessment;
class BackwardSequence implements CharSequence {
    private final String original;
    private final String reversed;
    public BackwardSequence(String original) {
        this.original = original;
        this.reversed = new StringBuilder(original).reverse().toString();
    }
    public int length() {
        return reversed.length();
    }
    public char charAt(int index) {
        return reversed.charAt(index);
    }
    public CharSequence subSequence(int start, int end) {
        return reversed.subSequence(start, end);
    }
    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(1): " + bs.charAt(1));
        System.out.println("subSequence(1, 4): " + bs.subSequence(1, 4));
    }
}
```

Output:

```
toString(): olleh
length(): 5
charAt(1): l
subSequence(1, 4): lle
```

2. Moveable Shapes Simulation

- Define an interface Movable with methods: moveUp(), moveDown(), moveLeft(), moveRight().

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

```
package day5_Assessment;
public class MoveableShapesSimulation {
    private double width;
    private double height;
    public MoveableShapesSimulation(double width, double height) {
        if (width > 0 && height > 0) {
            this.width = width;
            this.height = height;
        } else {
            this.width = 1.0;
            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());
    }
    public static void main(String[] args) {
```

```
        MoveableShapesSimulation r1 = new MoveableShapesSimulation(5.0, 3.0);
        r1.displayDetails();

        r1.setWidth(-2.0);
        r1.setHeight(4.0);
        System.out.println("After trying to update dimensions:");
        r1.displayDetails();
    }
}
```

Output:

```
Width: 5.0
Height: 3.0
Area: 15.0
Perimeter: 16.0
After trying to update dimensions:
Width: 5.0
Height: 4.0
Area: 20.0
Perimeter: 18.0
```

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.

```
package day5_Assessment;
interface Printer {
    void print(String document);
}
class LaserPrinter implements Printer {
    public void print(String document) {
        System.out.println("LaserPrinter printing: " + document.toUpperCase());
    }
}
class InkjetPrinter implements Printer {
    public void print(String document) {
        System.out.println("InkjetPrinter printing: " + document.toLowerCase());
    }
}
public class PrinterSwitch {
    public static void main(String[] args) {
        Printer p;
        p = new LaserPrinter();
        p.print("Contract Programming Document");
        p = new InkjetPrinter();
        p.print("Contract Programming Document");
```

```
    }
}
```

Output:

```
LaserPrinter printing: CONTRACT PROGRAMMING DOCUMENT
InkjetPrinter printing: contract programming 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.

```
package day5_Assessment;
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("Car started.");
        } else {
            System.out.println("Cannot start. No fuel.");
        }
    }
    public void stop() {
        System.out.println("Car stopped.");
    }
    public boolean refuel(int amount) {
        if (amount > 0) {
            fuel += amount;
            System.out.println("Refueled. Current fuel: " + fuel);
            return true;
        } else {
            System.out.println("Invalid fuel amount.");
            return false;
```

```
        }
    }
}
public class InterfaceHierarchy {
    public static void main(String[] args) {
        BaseVehicle base = new Car(0);
        base.start();
        AdvancedVehicle adv = new Car(10);
        adv.start();
        adv.refuel(20);
        adv.stop();
    }
}
```

Output:

```
Cannot start. No fuel.
Car started.
Refueled. Current fuel: 30
Car stopped.
```

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.

```
package day5_Assessment;

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 currentTime = LocalDateTime.now();
        for (Client client : clients) {
            client.updateTime(currentTime);
        }
    }
}
class ClockDisplay implements TimeServer.Client {
    private String name;

    public ClockDisplay(String name) {
        this.name = name;
    }
    public void updateTime(LocalDateTime now) {
        System.out.println(name + " Clock updated: " + now);
    }
}
class LoggerService implements TimeServer.Client {
    public void updateTime(LocalDateTime now) {
        System.out.println("Log: Time updated to " + now);
    }
```

```
}
public class CallbackHandling {
    public static void main(String[] args) {
        TimeServer server = new TimeServer();
        ClockDisplay clock1 = new ClockDisplay("Digital");
        ClockDisplay clock2 = new ClockDisplay("Analog");
        LoggerService logger = new LoggerService();
        server.registerClient(clock1);
        server.registerClient(clock2);
        server.registerClient(logger);
        server.notifyClients();
    }

}
```

Output:

```
Digital Clock updated: 2025-08-10T11:08:20.863304400
Analog Clock updated: 2025-08-10T11:08:20.863304400
Log: Time updated to 2025-08-10T11:08:20.863304400
```

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().

```
package day5_Assessment;
interface Polygon {
    double getArea();
    default double getPerimeter(int... sides) {
        double perimeter = 0;
        for (int side : sides) {
            perimeter += side;
        }
        return perimeter;
    }
    static String shapeInfo() {
        return "Polygon interface provides area and perimeter functionalities.";
    }
}
class Rectangle2 implements Polygon {
    private double length;
    private double width;
    public Rectangle2(double length, double width) {
        this.length = length;
        this.width = width;
    }
    public double getArea() {
        return length * width;
    }
}
class Triangle1 implements Polygon {
    private double base;
    private double height;
    public Triangle1(double base, double height) {
        this.base = base;
        this.height = height;
    }
    public double getArea() {
        return 0.5 * base * height;
    }
}
public class StaticMethodInInterface {
    public static void main(String[] args) {
        Rectangle2 rect = new Rectangle2(10, 5);
        Triangle1 tri = new Triangle1(8, 6);
        System.out.println("Info: " + Polygon.shapeInfo());
        System.out.println("Rectangle Area: " + rect.getArea());
        System.out.println("Rectangle Perimeter: " + rect.getPerimeter(10, 5, 10, 5));
        System.out.println("Triangle Area: " + tri.getArea());
        System.out.println("Triangle Perimeter: " + tri.getPerimeter(8, 7, 9));
    }
```

Output:
Info: Polygon interface provides area and perimeter functionalities.
Rectangle Area: 50.0
Rectangle Perimeter: 30.0
Triangle Area: 24.0
Triangle Perimeter: 24.0

Lambda expressions

1. Sum of Two Integers

```
package day5_Assessment;
interface Sum {
   int add(int a, int b);
}

public class SumCalculatorLambda {
   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.

```
package day5_Assessment;
@FunctionalInterface
interface SumCalculator {
 int sum(int a, int b);
}
public class LambdaSumDemo {
 public static void main(String[] args) {
    SumCalculator calculator = (a, b) -> a + b;
    int result = calculator.sum(10, 20);
    System.out.println("Sum: " + result);
 }
}
```

Output:

Sum:30

3. Check If a String Is Empty

   Create a lambda (via a functional interface like Predicate<String>) that returns true if a given string is empty.
   Predicate<String> isEmpty = s -> s.isEmpty();

```
package day5_Assessment;
import java.util.function.Predicate;
          public class StringCheck {
             public static void main(String[] args) {
                Predicate<String> isEmpty = s -> s.isEmpty();
```

```
        String str1 = "";
        String str2 = "Hello";
        System.out.println("Is str1 empty? " + isEmpty.test(str1));
        System.out.println("Is str2 empty? " + isEmpty.test(str2));
    }
}
```

Output:

```
Is str1 empty? true
Is str2 empty? false
```

4. Filter Even or Odd Numbers

```
package day5_Assessment;

import java.util.Arrays;

import java.util.List;

public class FilterEvenOddLambda {

    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(10, 15, 20, 25, 30);

        System.out.println("Even Numbers:");

        numbers.stream().filter(n -> n % 2 ==
0).forEach(System.out::println);

        System.out.println("Odd Numbers:");

        numbers.stream().filter(n -> n % 2 !=
0).forEach(System.out::println);

    }

}
```

Output:

```
Even Numbers:
10
20
30
Odd Numbers:
15
25
```

5. Convert Strings to Uppercase/Lowercase

```
package day5_Assessment;
public class StringConversion {
   public static void main(String[] args) {
      String text = "Hello Java";
      String upper = text.toUpperCase();
      System.out.println("Uppercase: " + upper);
      String lower = text.toLowerCase();
      System.out.println("Lowercase: " + lower);
   }
}
```

Output:
Uppercase: HELLO JAVA
Lowercase: hello java

6. Sort Strings by Length or Alphabetically

```
package day5_Assessment;
import java.util.*;
public class StringSorting {
   public static void main(String[] args) {
      List<String> words = Arrays.asList("Banana", "Apple", "Mango", "Grapes", "Orange");
      List<String> alphaList = new ArrayList<>(words);
      Collections.sort(alphaList);
      System.out.println("Alphabetical Order: " + alphaList);
      List<String> lengthList = new ArrayList<>(words);
      lengthList.sort((a, b) -> Integer.compare(a.length(), b.length()));
      System.out.println("Sorted by Length: " + lengthList);
   }
}
```

Output:

Alphabetical Order: [Apple, Banana, Grapes, Mango, Orange]
Sorted by Length: [Apple, Mango, Banana, Grapes, Orange]

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

```
package day5_Assessment;
import java.util.Arrays;
public class DoubleArrayAggregates {
   public static void main(String[] args) {
      double[] numbers = { 5.5, 2.3, 8.7, 4.4, 6.6 };
      double sum = Arrays.stream(numbers).sum();
      double max = Arrays.stream(numbers).max().orElse(Double.NaN);
      double average = Arrays.stream(numbers).average().orElse(Double.NaN);
      System.out.println("Numbers: " + Arrays.toString(numbers));
      System.out.println("Sum: " + sum);
      System.out.println("Max: " + max);
      System.out.println("Average: " + average);
   }
```

```
}
```

Output:
Numbers: [5.5, 2.3, 8.7, 4.4, 6.6]
Sum: 27.5
Max: 8.7
Average: 5.5

8. Create similar lambdas for max/min.

```
package day5_Assessment;
@FunctionalInterface
interface TwoNumberOperation {
   double operate(double a, double b);
}
public class LambdaMaxMin {
   public static void main(String[] args) {
         TwoNumberOperation maxOperation = (a, b) -> (a > b) ? a : b;
      TwoNumberOperation minOperation = (a, b) -> (a < b) ? a : b;
      double x = 15.7, y = 22.3;
      System.out.println("Numbers: " + x + " , " + y);
      System.out.println("Max: " + maxOperation.operate(x, y));
      System.out.println("Min: " + minOperation.operate(x, y));
   }
}
```

Output:

Numbers: 15.7 , 22.3
Max: 22.3
Min: 15.7

9. Calculate Factorial

```
package day5_Assessment;
@FunctionalInterface
interface FactorialCalculator {
   long factorial(int n);
}
public class LambdaFactorial {
   public static void main(String[] args) {
      FactorialCalculator fact = (n) -> {
         long result = 1;
         for (int i = 1; i <= n; i++) {
            result *= i;
         }
         return result;
      };
      int number = 5;
      System.out.println("Factorial of " + number + " is: " + fact.factorial(number));
   }
}
```

Output:

Factorial of 5 is: 120