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.

A.

```
package encapsulation_example;
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("Student Details:");
        System.out.println("Name       : " + name);
        System.out.println("Roll Number: " + rollNumber);
        System.out.println("Marks      : " + marks);
    }
    public void inputMarks(int newMarks) {
        System.out.println("Marks are immutable. Updates are not allowed in the
current version.");
    }
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 101, 85);
        s1.displayDetails();
        s1.inputMarks(90);
        s1.displayDetails();
    }
}
```

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.

```
A. package day5;
public class Rectangle {
    private double width;
    private double height;
    public Rectangle() {
        this.width = 1.0;
        this.height = 1.0;
    }
    public Rectangle(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. Setting to default value 1.0");
            this.width = 1.0;
        }
    }
    public void setHeight(double height) {
        if (height > 0) this.height = height;
        else {
            System.out.println("Invalid height. Setting to default value 1.0");
            this.height = 1.0;
```

```java
        }
    }
    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 static void main(String[] args) {
        Rectangle rect1 = new Rectangle(5, 3);
        rect1.displayDetails();
        System.out.println();
        Rectangle rect2 = new Rectangle(-2, 0);
        rect2.displayDetails();
    }
}
```

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.

A.

```java
package day5;

import java.util.ArrayList;

import java.util.List;

public class BankAccount {

    private String accountNumber;

    private String accountHolder;

    private double balance;

    private List<Transaction> transactionHistory;

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

        this.accountNumber = accountNumber;

        this.accountHolder = accountHolder;

        this.balance = initialBalance > 0 ? initialBalance : 0.0;

        this.transactionHistory = new ArrayList<>();
    }
    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 Transaction getLastTransaction() {
        if (transactionHistory.isEmpty()) return null;
        return transactionHistory.get(transactionHistory.size() - 1);
    }
    @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;
        }
        public String toString() {
            return type + ": " + amount;
        }
    }
    public static void main(String[] args) {
        BankAccount acc = new BankAccount("1234567890", "Alice", 500.0);
        acc.deposit(150.0);
        acc.withdraw(100.0);
        acc.withdraw(1000.0); // Should fail
        System.out.println(acc.toString());
        System.out.println("Last Transaction: " + acc.getLastTransaction());
    }
}
```

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.

A.

```
package day5;
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;
        System.out.println("Locker is now locked.");
    }
    public boolean unlock(String code) {
        if (securityManager.verify(code)) {
            isLocked = false;
            System.out.println("Locker unlocked successfully.");
            return true;
        } else {
            System.out.println("Incorrect passcode. Access denied.");
            return false;
        }
    }
    public boolean isLocked() {
        return isLocked;
    }
    private class SecurityManager {
        private boolean verify(String inputCode) {
            return inputCode.equals(passcode);
        }
    }
    public static void main(String[] args) {
        Locker locker = new Locker("L001", "1234");
        System.out.println("Is locked? " + locker.isLocked());
        locker.unlock("0000"); // Wrong
        locker.unlock("1234"); // Correct
        System.out.println("Is locked? " + locker.isLocked());
        locker.lock();
        System.out.println("Is locked? " + locker.isLocked());
    }
}
```

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.

A. **package** day5;

```
public final 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);
        }
    }
    @Override
    public String toString() {
        return "Product{name='" + name + "', code='" + code + "', price=" + price + ", category='" + category + "'}";
    }
    public static void main(String[] args) {
        Product product = new Product.Builder()
                .withName("Laptop")
                .withCode("LP1001")
                .withPrice(1200.00)
                .withCategory("Electronics")
                .build();

        System.out.println(product);
    }
}
```

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.

```
A. package day5;
public class BackwardSequence implements CharSequence {
    private String original;
    public BackwardSequence(String original) {
        this.original = original != null ? original : "";
    }
    @Override
    public int length() {
        return original.length();
    }
    @Override
    public char charAt(int index) {
        if(index < 0 || index >= length()) throw new IndexOutOfBoundsException();
        return original.charAt(length() - 1 - index);
    }
    @Override
    public CharSequence subSequence(int start, int end) {
        if(start < 0 || end > length() || start > end) throw new IndexOutOfBoundsException();
        StringBuilder sb = new StringBuilder();
        for(int i = start; i < end; i++) sb.append(charAt(i));
        return sb.toString();
    }
    @Override
    public String toString() {
        return new StringBuilder(original).reverse().toString();
    }
    public static void main(String[] args) {
        BackwardSequence seq = new BackwardSequence("hello");
        System.out.println("Original: " + seq.original);
        System.out.println("Length: " + seq.length());
        System.out.println("charAt(0): " + seq.charAt(0));
        System.out.println("charAt(4): " + seq.charAt(4));
        System.out.println("subSequence(1,4): " + seq.subSequence(1,4));
        System.out.println("toString(): " + seq.toString());
    }
}
```

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.

A. pacakage day5;

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

```
class MovablePoint implements Movable {
    int x, y, xSpeed, ySpeed;
    public MovablePoint(int x, int y, int xSpeed, int ySpeed) {
        this.x = x; this.y = y; this.xSpeed = xSpeed; this.ySpeed = ySpeed;
    }
    public void moveUp() { y -= ySpeed; }
    public void moveDown() { y += ySpeed; }
    public void moveLeft() { x -= xSpeed; }
    public void moveRight() { x += xSpeed; }
    public String toString() {
        return "(" + x + "," + y + ")";
    }
}
class MovableCircle implements Movable {
    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(radius=" + radius + ", 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(topLeft=" + topLeft + ", bottomRight=" + bottomRight +
")";
```

```
    }
}
public class MovableTest {
    public static void main(String[] args) {
        MovablePoint p = new MovablePoint(0, 0, 2, 3);
        MovableCircle c = new MovableCircle(5, new MovablePoint(10, 10, 1,
1));
        MovableRectangle r = new MovableRectangle(new MovablePoint(0, 0, 1,
1), new MovablePoint(5, 5, 1, 1));
        System.out.println(p);
        p.moveRight();
        p.moveUp();
        System.out.println("After moving point: " + p);
        System.out.println(c);
        c.moveLeft();
        c.moveDown();
        System.out.println("After moving circle: " + c);
        System.out.println(r);
        r.moveRight();
        r.moveDown();
        System.out.println("After moving rectangle: " + r);
    }
}
```

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.

```
A. package day5;
     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 PrinterTest {
  public static void main(String[] args) {
    Printer p;
    p = new LaserPrinter();
    p.print("Hello World");
    p = new InkjetPrinter();
    p.print("Hello World");
  }
}
```

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.

```
A. package day5;
interface BaseVehicle {
    void start();
}
interface AdvancedVehicle extends BaseVehicle {
    void stop();
    boolean refuel(int amount);
}
class Car implements AdvancedVehicle {
    private int fuel;

    public Car(int fuel) {
        this.fuel = fuel > 0 ? fuel : 0;
    }
    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 " + amount + " units. Total fuel: " + fuel);
            return true;
        }
        System.out.println("Invalid refuel amount.");
        return false;
    }
    public int getFuel() {
        return fuel;
    }
}
public class VehicleTest {
    public static void main(String[] args) {
        BaseVehicle baseVehicle = new Car(0);
        baseVehicle.start();
        AdvancedVehicle advancedVehicle = (AdvancedVehicle) baseVehicle;
        advancedVehicle.refuel(10);
        advancedVehicle.start();
        advancedVehicle.stop();
    }
}
```

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.

```
A. package day5;
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) {
      clients.add(client);
   }
   public void notifyClients() {
      LocalDateTime now = LocalDateTime.now();
      for (Client client : clients) {
         client.updateTime(now);
      }
   }
   public static void main(String[] args) {
      TimeServer server = new TimeServer();

      Client client1 = new Client() {
         @Override
         public void updateTime(LocalDateTime now) {
            System.out.println("Client1 received time: " + now);
         }
      };
      Client client2 = now -> System.out.println("Client2 received time: " +
now);

      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().

```
A. interface Polygon{
double getArea();
default double getPerimeter(int... sides){
double sum=0;
for(int side:sides)sum+=side;
return sum;
}
static String shapeInfo(){
return "Polygon shapes have area and perimeter.";
}
}
class Rectangle implements Polygon{
private double width,height;
public Rectangle(double width,double height){this.width=width;this.height=height;}
public double getArea(){return width*height;}
}
class Triangle implements Polygon{
private double base,height;
public Triangle(double base,double height){this.base=base;this.height=height;}
public double getArea(){return 0.5*base*height;}
}
public class Main{
public static void main(String[] args){
Polygon rectangle=new Rectangle(5,10);
Polygon triangle=new Triangle(4,7);
System.out.println("Rectangle area: "+rectangle.getArea());
System.out.println("Triangle area: "+triangle.getArea());
System.out.println("Rectangle perimeter: "+rectangle.getPerimeter(5,10,5,10));
System.out.println("Triangle perimeter: "+triangle.getPerimeter(3,4,5));
System.out.println(Polygon.shapeInfo());
}
}
```

Lambda expressions

1. Sum of Two Integers
2. Define a functional interface SumCalculator { int sum(int a, int b); } and a lambda expression to sum two integers.
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();
4. Filter Even or Odd Numbers

5. Convert Strings to Uppercase/Lowercase
6. Sort Strings by Length or Alphabetically
7. Aggregate Operations (Sum, Max, Average) on Double Arrays
8. Create similar lambdas for max/min.
9. Calculate Factorial

SOLUTION:

```
package day5;
public class Lambda_expressions {
   // QUE 1 & 2: Sum of two integers
   interface SumCalculator {
      int sum(int a, int b);
   }
   // QUE 3: Check if string is empty
   interface StringChecker {
      boolean check(String s);
   }
   // QUE 4: Filter even or odd numbers
   interface NumberFilter {
      boolean test(int n);
   }
   // QUE 5: Convert string case
   interface StringConverter {
      String convert(String s);
   }
```

```
// QUE 6: Compare strings
interface StringComparator {
    int compare(String a, String b);
}
// QUE 7 & 8: Array processing for sum, max, min, avg
interface ArrayProcessor {
    double process(double[] arr);
}
// QUE 9: Factorial calculation
interface FactorialCalculator {
    long factorial(int n);
}
public static void main(String[] args) {
    // QUE 1 & 2
    SumCalculator sumCalc = (a, b) -> a + b;
    System.out.println("Sum: " + sumCalc.sum(10, 20));
    // QUE 3
    StringChecker isEmpty = s -> s.isEmpty();
    System.out.println("Is empty (\"\"): " + isEmpty.check(""));
    System.out.println("Is empty (\"abc\"): " + isEmpty.check("abc"));
    // QUE 4
    int[] numbers = {1, 2, 3, 4, 5, 6};
    NumberFilter isEven = n -> n % 2 == 0;
    NumberFilter isOdd = n -> n % 2 != 0;
    System.out.print("Even: ");
    for (int n : numbers) {
        if (isEven.test(n)) System.out.print(n + " ");
    }
    System.out.print("\nOdd: ");
```

```
for (int n : numbers) {
    if (isOdd.test(n)) System.out.print(n + " ");
}
System.out.println();
// QUE 5
StringConverter toUpper = s -> s.toUpperCase();
StringConverter toLower = s -> s.toLowerCase();
System.out.println("Uppercase: " + toUpper.convert("hello"));
System.out.println("Lowercase: " + toLower.convert("WORLD"));
// QUE 6
String[] words = {"banana", "apple", "pear"};
StringComparator byLength = (a, b) -> a.length() - b.length();
StringComparator alphabetical = (a, b) -> a.compareTo(b);
// Sort by length
for (int i = 0; i < words.length - 1; i++) {
    for (int j = i + 1; j < words.length; j++) {
        if (byLength.compare(words[i], words[j]) > 0) {
            String temp = words[i];
            words[i] = words[j];
            words[j] = temp;
        }
    }
}
System.out.print("Sorted by length: ");
for (String w : words) System.out.print(w + " ");
System.out.println();
// Sort alphabetically
for (int i = 0; i < words.length - 1; i++) {
    for (int j = i + 1; j < words.length; j++) {
            if (alphabetical.compare(words[i], words[j]) > 0) {
                String temp = words[i];
                words[i] = words[j];
                words[j] = temp;
            }
        }
    }
    System.out.print("Sorted alphabetically: ");
    for (String w : words) System.out.print(w + " ");
    System.out.println();
    // QUE 7 & 8
    double[] nums = {1.0, 2.5, 3.5};
    ArrayProcessor sum = arr -> {
        double s = 0;
        for (double d : arr) s += d;
        return s;
    };
    ArrayProcessor max = arr -> {
        double m = arr[0];
        for (double d : arr) if (d > m) m = d;
        return m;
    };
    ArrayProcessor min = arr -> {
        double m = arr[0];
        for (double d : arr) if (d < m) m = d;
        return m;
    };
    ArrayProcessor avg = arr -> sum.process(arr) / arr.length;
```

```
        System.out.println("Array sum: " + sum.process(nums));
        System.out.println("Array max: " + max.process(nums));
        System.out.println("Array min: " + min.process(nums));
        System.out.println("Array avg: " + avg.process(nums));
        // QUE 9
        FactorialCalculator factorial = new FactorialCalculator() {
            public long factorial(int n) {
                return (n <= 1) ? 1 : n * factorial(n - 1);
            }
        };

        System.out.println("Factorial of 5: " + factorial.factorial(5));
    }
}
```