**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 Assignment5;
* public class Grade\_validation {
* private String name;
* private int rollNumber;
* private int marks;
* public Grade\_validation(String name, int rollNumber, int marks) {
* this.name = name;
* this.rollNumber = rollNumber;
* if (marks >= 0 && marks <= 100) {
* this.marks = marks;
* } else {
* this.marks = 0;
* }
* }
* // Getter methods
* public String getName() {
* return name;
* }
* public int getRollNumber() {
* return rollNumber;
* }
* public int getMarks() {
* return marks;
* }
* public void displayDetails() {
* System.***out***.println("Name: " + name);
* System.***out***.println("Roll Number: " + rollNumber);
* System.***out***.println("Marks: " + marks);
* }
* public void inputMarks(int newMarks) {
* if (newMarks > this.marks && newMarks <= 100) {
* this.marks = newMarks;
* } else {
* System.***out***.println("Invalid mark update. Either less than existing or out of range.");
* }
* }
* public static void main(String[] args) {
* Grade\_validation s1 = new Grade\_validation("Ravi", 101, 85);
* s1.displayDetails();
* s1.inputMarks(90);
* s1.inputMarks(50);
* System.***out***.println("After attempted updates:");
* 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.
* package Assignment5;
* public class Rectangle {
* private double width;
* private double height;
* public Rectangle(double width, double height) {
* if (width > 0) {
* this.width = width;
* } else {
* this.width = 1;
* System.***out***.println("Invalid width. Setting default 1.");
* }
* if (height > 0) {
* this.height = height;
* } else {
* this.height = 1;
* System.***out***.println("Invalid height. Setting default 1.");
* }
* }
* public void setWidth(double width) {
* if (width > 0) {
* this.width = width;
* } else {
* System.***out***.println("Width must be positive.");
* }
* }
* public void setHeight(double height) {
* if (height > 0) {
* this.height = height;
* } else {
* System.***out***.println("Height must be positive.");
* }
* }
* 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) {
* Rectangle rect1 = new Rectangle(5, -3);
* rect1.displayDetails();
* System.***out***.println();
* rect1.setHeight(4);
* rect1.setWidth(-2);
* rect1.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.
* package Assignment5;
* import java.util.ArrayList;
* import java.util.List;
* class Transaction {
* String type;
* double amount;
* Transaction(String type, double amount) {
* this.type = type;
* this.amount = amount;
* }
* public String toString() {
* return type + ": ₹" + amount;
* }
* }
* public class Bank\_account {
* private String accountNumber;
* private String accountHolder;
* private double balance;
* private List<Transaction> transactionHistory = new ArrayList<>();
* public Bank\_account(String accountNumber, String accountHolder, double initialBalance) {
* this.accountNumber = accountNumber;
* this.accountHolder = accountHolder;
* this.balance = initialBalance;
* }
* public void deposit(double amount) {
* if (amount > 0) {
* balance += amount;
* transactionHistory.add(new Transaction("Deposit", amount));
* System.***out***.println("Deposit successful.");
* } else {
* System.***out***.println("Invalid deposit amount.");
* }
* }
* public boolean withdraw(double amount) {
* if (amount > 0 && amount <= balance) {
* balance -= amount;
* transactionHistory.add(new Transaction("Withdraw", amount));
* System.***out***.println("Withdrawal successful.");
* return true;
* } else {
* System.***out***.println("Withdrawal failed: Insufficient funds or invalid amount.");
* return false;
* }
* }
* 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 masked = "\*\*\*\*" + accountNumber.substring(accountNumber.length() - 4);
* return "Account Holder: " + accountHolder +
* "\nAccount Number: " + masked +
* "\nBalance: ₹" + balance;
* }
* public static void main(String[] args) {
* Bank\_account account = new Bank\_account("1234567890", "Anish", 5000.0);
* System.***out***.println(account);
* account.deposit(1000);
* account.withdraw(2000);
* account.withdraw(5000);
* System.***out***.println("\nCurrent Balance: ₹" + account.getBalance());
* System.***out***.println("Last Transaction: " + account.getLastTransaction());
* System.***out***.println("\n" + account);
* }
* }

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 Assignment5;
* public class Locker {
* private String lockerId;
* private boolean isLocked;
* private String passcode;
* public Locker(String lockerId, String passcode) {
* this.lockerId = lockerId;
* this.passcode = passcode;
* this.isLocked = false;
* }
* private class SecurityManager {
* private boolean verify(String inputCode) {
* return passcode.equals(inputCode);
* }
* }
* public void lock() {
* isLocked = true;
* System.***out***.println("Locker is now locked.");
* }
* public void unlock(String code) {
* SecurityManager sm = new SecurityManager();
* if (sm.verify(code)) {
* isLocked = false;
* System.***out***.println("Locker unlocked successfully.");
* } else {
* System.***out***.println("Incorrect passcode. Access denied.");
* }
* }
* public boolean isLocked() {
* return isLocked;
* }
* public static void main(String[] args) {
* Locker myLocker = new Locker("L001", "1234");
* System.***out***.println("Is locker locked? " + myLocker.isLocked());
* myLocker.lock();
* System.***out***.println("Is locker locked? " + myLocker.isLocked());
* myLocker.unlock("0000");
* System.***out***.println("Is locker locked? " + myLocker.isLocked());
* myLocker.unlock("1234");
* System.***out***.println("Is locker locked? " + myLocker.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.
* package Assignment5;
* 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) {
* this.name = name;
* return this;
* }
* public Builder withCode(String code) {
* 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 String toString() {
* return "Product Details:\n" +
* "Name: " + name + "\n" +
* "Code: " + code + "\n" +
* "Price: ₹" + price + "\n" +
* "Category: " + (category != null ? category : "N/A");
* }
* public static void main(String[] args) {
* Product product = new Product.Builder()
* .withName("Laptop")
* .withCode("LP1001")
* .withPrice(55000)
* .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.**

package Assignment5;

public class BackwardSequence implements CharSequence {

private String reversed;

public BackwardSequence(String input) {

StringBuilder sb = new StringBuilder(input);

this.reversed = sb.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.substring(start, end);

}

public String toString() {

return reversed;

}

public static void main(String[] args) {

BackwardSequence bs = new BackwardSequence("hello");

System.***out***.println("Reversed string: " + bs);

System.***out***.println("Length: " + bs.length());

System.***out***.println("Character at index 1: " + bs.charAt(1));

System.***out***.println("SubSequence (1, 4): " + bs.subSequence(1, 4));

}

}

2. Moveable Shapes Simulation

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

package Assignment5;

interface Movable {

void moveUp();

void moveDown();

void moveLeft();

void moveRight();

}

class MovablePoint implements Movable {

int x, y, xSpeed, ySpeed;

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 {

int radius; MovablePoint center;

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 center=" + center + " radius=" + radius;

}

}

class MovableRectangle implements Movable {

MovablePoint topLeft, bottomRight;

MovableRectangle(MovablePoint t, MovablePoint b) {

if (t.xSpeed != b.xSpeed || t.ySpeed != b.ySpeed)

throw new IllegalArgumentException("Speeds must match");

this.topLeft = t;

this.bottomRight = b;

}

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 "Rect TL=" + topLeft + " BR=" + bottomRight;

}

}

public class Main {

public static void main(String[] args) {

MovablePoint p = new MovablePoint(0,0,1,1);

MovableCircle c = new MovableCircle(5, new MovablePoint(2,2,1,1));

MovableRectangle r = new MovableRectangle(

new MovablePoint(0,5,2,2),

new MovablePoint(4,1,2,2)

);

System.***out***.println(p);

System.***out***.println(c);

System.***out***.println(r);

p.moveRight();

c.moveUp();

r.moveLeft();

System.***out***.println(p);

System.***out***.println(c);

System.***out***.println(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.
* package Assignment5;
* interface Printer {
* void print(String document);
* }
* class LaserPrinter implements Printer {
* public void print(String document) {
* System.***out***.println("Laser Printer printing: " + document);
* }
* }
* class InkjetPrinter implements Printer {
* public void print(String document) {
* System.***out***.println("Inkjet Printer printing: " + document);
* }
* }
* public class PrinterSwitch {
* public static void main(String[] args) {
* Printer p;
* p = new LaserPrinter();
* p.print("Invoice.pdf");
* p = new InkjetPrinter();
* p.print("Photo.jpg");
* }
* }

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 Assignment5;

interface BaseVehicle {

void start();

}

interface AdvancedVehicle extends BaseVehicle {

void stop();

boolean refuel(int amount);

}

class Car implements AdvancedVehicle {

int fuel;

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

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) {

fuel += amount;

System.***out***.println("Refueled " + amount + " liters. Fuel now: " + fuel);

return true;

}

}

public class Hierarchy {

public static void main(String[] args) {

BaseVehicle bv = new Car(5);

bv.start();

AdvancedVehicle av = (AdvancedVehicle) bv;

av.stop();

av.refuel(10);

av.start();

}

}

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 Assignment5;

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 c : clients) {

c.updateTime(now);

}

}

}

class DigitalClock implements TimeServer.Client {

public void updateTime(LocalDateTime now) {

System.***out***.println("DigitalClock: Current time is " + now);

}

}

class AnalogClock implements TimeServer.Client {

public void updateTime(LocalDateTime now) {

System.***out***.println("AnalogClock: Current time is " + now);

}

}

public class CallbackHandling {

public static void main(String[] args) {

TimeServer server = new TimeServer();

server.registerClient(new DigitalClock());

server.registerClient(new AnalogClock());

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().
* package Assignment5;
* interface Polygon {
* double getArea();
* default double getPerimeter(int... sides) {
* double sum = 0;
* for (int s : sides)
* sum += s;
* return sum;
* }
* static String shapeInfo() {
* return "A polygon is a closed shape with straight sides.";
* }
* }
* class Rect implements Polygon {
* double length, width;
* Rect(double l, double w) {
* length = l;
* width = w;
* }
* public double getArea() {
* return length \* width;
* }
* }
* class Triangle implements Polygon {
* double base, height;
* Triangle(double b, double h) {
* base = b;
* height = h;
* }
* public double getArea() {
* return 0.5 \* base \* height;
* }
* }
* public class Mainpolygon {
* public static void main(String[] args) {
* Rect rect = new Rect(4, 5);
* Triangle tri = new Triangle(3, 6);
* System.***out***.println("Rectangle area: " + rect.getArea());
* System.***out***.println("Rectangle perimeter: " + rect.getPerimeter(4, 5, 4, 5));
* System.***out***.println("Triangle area: " + tri.getArea());
* System.***out***.println("Triangle perimeter: " + tri.getPerimeter(3, 4, 5));
* System.***out***.println(Polygon.*shapeInfo*());
* }
* }

**Lambda expressions**

1. Sum of Two Integers

package Assignment5;

interface Addable {

int add(int a, int b);

}

public class sum\_lambda {

public static void main(String[] args) {

Addable sum = (a, b) -> a + b;

int result = sum.add(10, 20);

System.***out***.println("Sum: " + result);

}

}

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

package Assignment5;

interface SumCalculator {

int sum(int a, int b);

}

public class Calc\_lambda {

public static void main(String[] args) {

SumCalculator calculator = (a, b) -> a + b;

int result = calculator.sum(15, 25);

System.***out***.println("Sum: " + result);

}

}

1. 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 Assignment5;

import java.util.function.Predicate;

public class Emptystring {

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));

}

}

1. Filter Even or Odd Numbers

package Assignment5;

1. import java.util.Scanner;
2. interface CheckNumber {
3. void test(int n);
4. }
5. public class evenodd\_lambda {
6. public static void main(String[] args) {
7. Scanner scanner = new Scanner(System.***in***);
8. System.***out***.print("Enter a number: ");
9. int number = scanner.nextInt();
10. CheckNumber check = (n) -> {
11. if (n % 2 == 0)
12. System.***out***.println(n + " is Even.");
13. else
14. System.***out***.println(n + " is Odd.");
15. };
16. check.test(number);
17. scanner.close();
18. }
19. }
20. Convert Strings to Uppercase/Lowercase

package Assignment5;

import java.util.Scanner;

interface StringConverter {

String convert(String input);

}

public class Stringcaseconverter {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.***in***);

System.***out***.print("Enter a string: ");

String input = scanner.nextLine();

StringConverter toUpper = s -> s.toUpperCase();

StringConverter toLower = s -> s.toLowerCase();

System.***out***.println("Uppercase: " + toUpper.convert(input));

System.***out***.println("Lowercase: " + toLower.convert(input));

scanner.close();

}

}

6. Sort Strings by Length or Alphabetically

package Assignment5;

import java.util.\*;

public class sortstring\_lambda {

public static void main(String[] args) {

List<String> names = Arrays.*asList*("Apple", "Banana", "Kiwi", "Mango", "Orange");

List<String> byLength = new ArrayList<>(names);

byLength.sort((s1, s2) -> Integer.*compare*(s1.length(), s2.length()));

System.***out***.println("Sorted by Length: " + byLength);

List<String> alphabetically = new ArrayList<>(names);

alphabetically.sort((s1, s2) -> s1.compareTo(s2));

System.***out***.println("Sorted Alphabetically: " + alphabetically);

}

}

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

package Assignment5;

import java.util.Scanner;

public class doublearray {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.***in***);

System.***out***.print("Enter how many numbers: ");

int size = scanner.nextInt();

double[] numbers = new double[size];

for (int i = 0; i < size; i++) {

System.***out***.print("Enter number " + (i + 1) + ": ");

numbers[i] = scanner.nextDouble();

}

double sum = 0;

double max = numbers[0];

double average;

for (double num : numbers) {

sum += num;

if (num > max) {

max = num;

}

}

average = sum / size;

System.***out***.println("Sum: " + sum);

System.***out***.println("Maximum: " + max);

System.***out***.println("Average: " + average);

scanner.close();

}

}

1. Create similar lambdas for max/min.

package Assignment5;

interface NumberCompare {

int compare(int a, int b);

}

public class maxmin\_lambda {

public static void main(String[] args) {

NumberCompare max = (a, b) -> (a > b) ? a : b;

NumberCompare min = (a, b) -> (a < b) ? a : b;

int x = 12, y = 25;

System.***out***.println("Max of " + x + " and " + y + ": " + max.compare(x, y));

System.***out***.println("Min of " + x + " and " + y + ": " + min.compare(x, y));

}

}

1. Calculate Factorial

package Assignment5;

import java.util.Scanner;

interface Factorial {

int calculate(int n);

}

public class Factorial\_lambda {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.***in***);

System.***out***.print("Enter a number: ");

int num = scanner.nextInt();

Factorial fact = (n) -> {

int result = 1;

for (int i = 1; i <= n; i++) {

result \*= i;

}

return result;

};

int output = fact.calculate(num);

System.***out***.println("Factorial of " + num + " is: " + output);

scanner.close();

}

}