**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.

Code; **package** Encapsulation\_practice;

**import** java.util.\*;

**public** **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** (marks >= 0 && marks <= 100) {

**this**.marks = marks;

} **else** {

**this**.marks = 0; // Reset to 0 if invalid

}

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getRollNumber() {

**return** rollNumber;

}

**public** **void** setRollNumber(**int** rollNumber) {

**this**.rollNumber = rollNumber;

}

**void** displayDetails() {

System.***out***.println("Name is :"+name);

System.***out***.println("RollNumber is :"+rollNumber);

System.***out***.println("Marks is :"+marks);

}

**public** **void** inputMarks(**int** newMarks) {

**if** (newMarks > marks && newMarks <= 100) {

marks = newMarks;

System.***out***.println("The new marks added are: " + newMarks);

} **else** **if** (newMarks <= marks) {

System.***out***.println("It is not possible to reduce marks.");

} **else** {

System.***out***.println("Invalid marks. Marks should be between 0 and 100.");

}

}

**public** **static** **void** main(String[] args) {

Student1 s = **new** Student1("raju", 21, 95);

s.displayDetails();

s.inputMarks(97);

s.displayDetails();

s.inputMarks(90);

s.displayDetails();

}

}

output; Name is :raju

RollNumber is :21

Marks is :95

The new marks added are: 97

Name is :raju

RollNumber is :21

Marks is :97

It is not possible to reduce marks.

Name is :raju

RollNumber is :21

Marks is :97

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.

Code; **package** Encapsulation\_practice;

**import** java.util.\*;

**public** **class** Rectangle {

**private** **int** width;

**private** **int** height;

**private** **static** **final** **int** ***DEFAULT\_WIDTH*** = 1;

**private** **static** **final** **int** ***DEFAULT\_HEIGHT*** = 1;

**public** Rectangle(**int** width,**int** height){

setWidth(width);

setHeight(height);

}

**public** **void** setWidth(**int** width) {

**if** (width > 0) {

**this**.width = width;

} **else** {

System.***out***.println("Invalid width provided. Using default: " + ***DEFAULT\_WIDTH***);

**this**.width = ***DEFAULT\_WIDTH***;

}

}

**public** **void** setHeight(**int** height) {

**if** (height > 0) {

**this**.height = height;

} **else** {

System.***out***.println("Invalid height provided. Using default: " + ***DEFAULT\_HEIGHT***);

**this**.height = ***DEFAULT\_HEIGHT***;

}

}

**public** **int** getWidth() {

**return** width;

}

**public** **int** getHeight() {

**return** height;

}

**public** **int** getArea() {

**return** width\*height;

}

**public** **int** getPerimeter() {

**return** 2\*(width+height);

}

**public** String toString() {

**return** width+" rectangle of "+height;

}

**void** displayDetails() {

System.***out***.println("Rectangle Details:");

System.***out***.println("Width: " + width);

System.***out***.println("Height: " + height);

System.***out***.println("Area of rectangle is: " + getArea());

System.***out***.println("Perimeter of rectangle is: " + getPerimeter());

System.***out***.println("---------------------------");

}

**public** **static** **void** main(String[] args) {

Rectangle r1 = **new** Rectangle(3, 5);

r1.displayDetails();

Rectangle r2 = **new** Rectangle(-6, 8);

r2.displayDetails();

Rectangle r3 = **new** Rectangle(4, -2);

r3.displayDetails();

Rectangle r4 = **new** Rectangle(-3, -7);

r4.displayDetails();

}

}

Output; Rectangle Details:

Width: 3

Height: 5

Area of rectangle is: 15

Perimeter of rectangle is: 16

---------------------------

Invalid width provided. Using default: 1

Rectangle Details:

Width: 1

Height: 8

Area of rectangle is: 8

Perimeter of rectangle is: 18

---------------------------

Invalid height provided. Using default: 1

Rectangle Details:

Width: 4

Height: 1

Area of rectangle is: 4

Perimeter of rectangle is: 10

---------------------------

Invalid width provided. Using default: 1

Invalid height provided. Using default: 1

Rectangle Details:

Width: 1

Height: 1

Area of rectangle is: 1

Perimeter of rectangle is: 4

---------------------------

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.

Code; **package** Encapsulation\_practice;

**import** java.util.List;

**import** java.util.ArrayList;

**public** **class** BankAccount1 {

**private** **long** accountNumber;

**private** String accountholder;

**private** **double** balance;

**private** List<Transaction> transactions = **new** ArrayList<>();

**public** BankAccount1(**long** accountNumber,String accountholder,**double** balance) {

**this**.accountNumber=accountNumber;

**this**.accountholder=accountholder;

**this**.balance=balance;

}

**public** **void** deposit(**double** amount) {

**if** (amount <= 0)

{

System.***out***.println("Deposit amount must be positive.");

**return**;

}

balance += amount;

transactions.add(**new** Transaction("Deposit", amount));

}

**public** **boolean** withdraw(**double** amount) {

**if** (amount <= 0) {

System.***out***.println("Withdrawal amount must be positive.");

**return** **false**;

}

**if** (amount > balance) {

System.***out***.println("Insufficient balance.");

**return** **false**;

}

balance -= amount;

transactions.add(**new** Transaction("Withdraw", amount));

**return** **true**;

}

**public** **double** getbalance() {

**return** balance;

}

**public** **long** getMaskedAccountNumber() {

String accStr = String.*valueOf*(accountNumber);

**if** (accStr.length() > 4) {

**return** Long.*parseLong*("\*\*\*\*" + accStr.substring(accStr.length() - 4));

}

**return** accountNumber;

}

**public** String getLastTransaction() {

**if** (transactions.isEmpty()) {

**return** "No transactions yet.";

}

**return** transactions.get(transactions.size() - 1).toString();

}

**public** String toString() {

**return** "Account Holder: " + accountholder +

", Account Number: \*\*\*\*" + String.*valueOf*(accountNumber).substring(String.*valueOf*(accountNumber).length() - 4) +

", Balance: ₹" + balance;

}

**private** **class** Transaction {

**private** String type;

**private** **double** amount;

**public** Transaction(String type, **double** amount) {

**this**.type = type;

**this**.amount = amount;

}

@Override

**public** String toString() {

**return** type + " of ₹" + amount;

}

}

**public** **static** **void** main(String[] args) {

BankAccount1 acc = **new** BankAccount1(123456789012L, "Jagadheeswar", 5000.0);

acc.deposit(1500);

acc.withdraw(2000);

acc.withdraw(6000);

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

System.***out***.println("Last Transaction: " + acc.getLastTransaction());

}

}

Output; Insufficient balance.

Account Holder: Jagadheeswar, Account Number: \*\*\*\*9012, Balance: ₹4500.0

Last Transaction: Withdraw of ₹2000.0

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.

Code; **package** Encapsulation\_practice;

**import** java.util.\*;

**public** **class** Locker {

**private** **int** lockerid;

**private** **boolean** isLocked;

**private** **long** passcode;

**public** Locker(**int** lockerid,**long** passcode) {

**this**.lockerid=lockerid;

**this**.isLocked=**true**;

**this**.passcode=passcode;

}

**public** **int** getlockerid() {

**return** lockerid;

}

**public** **boolean** getisLocked() {

**return** isLocked;

}

**public** **long** getpasscode() {

**return** passcode;

}

**public** **void** lock() {

isLocked=**true**;

System.***out***.println("Locker with" +lockerid+" is now locked.");

}

**public** **boolean** unlock(String code) {

SecurityManager sm = **new** SecurityManager();

**boolean** success = sm.verify(code);

**if** (success) {

isLocked = **false**;

System.***out***.println("Locker " + lockerid + " unlocked successfully.");

} **else** {

System.***out***.println("Incorrect passcode. Locker remains locked.");

}

**return** success;

}

**public** **boolean** isLocked() {

**return** isLocked;

}

**private** **class** SecurityManager {

**private** **boolean** verify(String inputCode) {

**try** {

**long** code = Long.*parseLong*(inputCode);

**return** code == passcode;

} **catch** (NumberFormatException e) {

**return** **false**;

}

}

}

**public** String toString() {

**return** "Locker ID: " + lockerid + ", Locked: " + isLocked;

}

**public** **static** **void** main(String[] args) {

Locker locker = **new** Locker(101, 123456);

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

locker.unlock("123456");

System.***out***.println(locker.isLocked());

locker.lock();

locker.unlock("000000");

}

}

Output; Locker ID: 101, Locked: true

Locker 101 unlocked successfully.

false

Locker with101 is now locked.

Incorrect passcode. Locker remains locked.

5. Builder Pattern & Encapsulation: Immutable Product

Use Builder design to create immutable class with encapsulation.

* Create an immutable Product class with private final fields such as name, code, price, and optional category.
* Use a static nested Builder inside the Product class. Provide methods like withName(), withPrice(), etc., that apply validation (e.g. non-negative price).
* The outer class should have only getter methods, no setters.
* The builder returns a new Product instance only when all validations succeed.

Code; **package** Encapsulation\_practice;

**public** **final** **class** Product {

**private** **final** String name;

**private** **final** **long** 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** **long** getCode() {

**return** code;

}

**public** **double** getPrice() {

**return** price;

}

**public** String getCategory() {

**return** category;

}

@Override

**public** String toString() {

**return** "Product{name='" + name + "', code=" + code +

", price=" + price + ", category='" + category + "'}";

}

**public** **static** **class** Builder {

**private** String name;

**private** **long** code;

**private** **double** price;

**private** String category;

**public** Builder withName(String name) {

**if** (name == **null** || name.trim().isEmpty()) {

**throw** **new** IllegalArgumentException("Name cannot be empty");

}

**this**.name = name;

**return** **this**;

}

**public** Builder withCode(**long** code) {

**if** (code <= 0) {

**throw** **new** IllegalArgumentException("Code must be positive");

}

**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 == 0 || price < 0) {

**throw** **new** IllegalStateException("Missing or invalid fields");

}

**return** **new** Product(**this**);

}

}

**public** **static** **void** main(String[] args) {

Product product = **new** Product.Builder()

.withName("Dairymilk")

.withCode(98654)

.withPrice(86.0)

.withCategory("Chocolates")

.build();

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

}

}

Output; Product{name='Dairymilk', code=98654, price=86.0, category='Chocolates'}

**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.

Code; **package** Interface\_practice;

**import** java.lang.CharSequence;

**public** **class** BackwardSequence **implements** CharSequence{

**private** **final** String original;

**private** **final** String reversed;

**public** BackwardSequence(String input) {

**this**.original = input;

**this**.reversed = **new** StringBuilder(input).reverse().toString();

}

**public** **int** length() {

**return** reversed.length();

}

**public** **char** charAt(**int** index) {

**if** (index < 0 || index >= reversed.length()) {

**throw** **new** IndexOutOfBoundsException("Index: " + index);

}

**return** reversed.charAt(index);

}

**public** CharSequence subSequence(**int** start, **int** end) {

**if** (start < 0 || end > reversed.length() || start > end) {

**throw** **new** IndexOutOfBoundsException("Invalid subsequence range");

}

**return** reversed.substring(start, end);

}

**public** String toString() {

**return** reversed;

}

**public** **static** **void** main(String[] args) {

BackwardSequence bs = **new** BackwardSequence("hello");

System.***out***.println("Original: hello");

System.***out***.println("Reversed (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; Original: hello

Reversed (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.

Code; **package** Interface\_practice;

**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;

}

**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 at (" + 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 with center " + center + " and radius " + radius;

}

}

**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 the 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);

System.***out***.println("Initial Point: " + p);

p.moveUp();

p.moveRight();

System.***out***.println("After moving Point: " + p);

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

System.***out***.println("\nInitial Circle: " + c);

c.moveDown();

c.moveLeft();

System.***out***.println("After moving Circle: " + c);

MovableRectangle r = **new** MovableRectangle(

**new** MovablePoint(0, 10, 1, 1),

**new** MovablePoint(10, 0, 1, 1)

);

System.***out***.println("\nInitial Rectangle: " + r);

r.moveUp();

r.moveRight();

System.***out***.println("After moving Rectangle: " + r);

}

}

Output; Initial Point: Point at (0, 0)

After moving Point: Point at (2, 3)

Initial Circle: Circle with center Point at (1, 1) and radius 5

After moving Circle: Circle with center Point at (0, 0) and radius 5

Initial Rectangle: Rectangle [TopLeft: Point at (0, 10), BottomRight: Point at (10, 0)]

After moving Rectangle: Rectangle [TopLeft: Point at (1, 11), BottomRight: Point at (11, 1)]

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.

Code; **package** Interface\_practice;

**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("Contract Agreement");

p = **new** InkjetPrinter();

p.print("Contract Agreement");

}

}

Output; LaserPrinter printing: CONTRACT AGREEMENT

InkjetPrinter printing: contract agreement

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.

Code; **package** Interface\_practice;

**interface** BaseVehicle{

**void** start();

}

**interface** AdvancedVehicle **extends** BaseVehicle{

**void** stop();

**boolean** refuel(**int** amount);

}

**class** Car1 **implements** AdvancedVehicle{

**private** **int** fuelLevel;

**public** Car1(**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 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("Refueled " + amount + " units. Current fuel: " + fuelLevel);

**return** **true**;

}

**public** **int** getFuelLevel() {

**return** fuelLevel;

}

**public** String toString() {

**return** "Car with fuel level:"+fuelLevel;

}

}

**public** **class** VehicleTest {

**public** **static** **void** main(String[] args) {

AdvancedVehicle myCar = **new** Car1(10);

myCar.start();

myCar.refuel(5);

myCar.stop();

BaseVehicle baseRef = myCar;

baseRef.start();

**if** (baseRef **instanceof** AdvancedVehicle) {

AdvancedVehicle advancedRef = (AdvancedVehicle) baseRef;

advancedRef.refuel(10);

advancedRef.stop();

}

}

}

Output; Car started with fuel level: 10

Refueled 5 units. Current fuel: 15

Car stopped.

Car started with fuel level: 15

Refueled 10 units. Current fuel: 25

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.

Code; **package** Interface\_practice;

**import** java.time.LocalDateTime;

**import** java.util.\*;

**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 digitalClock = **new** DigitalClock();

Client logger = **new** TimeLogger();

server.registerClient(digitalClock);

server.registerClient(logger);

server.notifyClients();

}

}

**class** DigitalClock **implements** TimeServer.Client {

**public** **void** updateTime(LocalDateTime now) {

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

}

}

**class** TimeLogger **implements** TimeServer.Client {

**public** **void** updateTime(LocalDateTime now) {

System.***out***.println("TimeLogger: Logging time - " + now);

}

}

Output;

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().

Code; **package** Interface\_practice;

**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 is a closed figure with straight sides.";

}

}

**class** Rectangle **implements** Polygon{

**private** **int** length;

**private** **int** breadth;

**public** Rectangle(**int** length, **int** breadth) {

**this**.length = length;

**this**.breadth = breadth;

}

**public** **double** getArea() {

**return** length \* breadth;

}

}

**class** Triangle **implements** Polygon{

**private** **int** base;

**private** **int** height;

**public** Triangle(**int** base, **int** height) {

**this**.base = base;

**this**.height = height;

}

**public** **double** getArea() {

**return** 0.5 \* base \* height;

}

}

**public** **class** PolygonTest {

**public** **static** **void** main(String[] args) {

Polygon rect = **new** Rectangle(10, 5);

Polygon tri = **new** Triangle(6, 4);

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(5, 6, 7));

System.***out***.println("Shape Info: " + Polygon.*shapeInfo*());

}

}

Output; Rectangle Area: 50.0

Rectangle Perimeter: 30.0

Triangle Area: 12.0

Triangle Perimeter: 18.0

Shape Info: Polygon is a closed figure with straight sides.

**Lambda expressions**

1. Sum of Two Integers

Code; **package** Lambda\_functions;

**import** java.util.Scanner;

**interface** SumFunction{

**int** sum(**int** a,**int** b);

}

**public** **class** Sumoftwo {

**public** **static** **void** main(String[] args) {

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

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

**int** a=scan.nextInt();

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

**int** b=scan.nextInt();

SumFunction add = (x, y) -> x + y;

**int** result = add.sum(a, b);

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

}

}

Output; Enter first number: 4

Enter second number: 5

Sum is: 9

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

Code; **package** Lambda\_functions;

**import** java.util.Scanner;

**interface** SumCalculator{

**int** sum(**int** a,**int** b);

}

**public** **class** Sumoftwo {

**public** **static** **void** main(String[] args) {

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

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

**int** a=scan.nextInt();

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

**int** b=scan.nextInt();

SumCalculator add = (x, y) -> x + y;

**int** result = add.sum(a, b);

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

}

}

Output; Enter first number: 5

Enter second number: 3

Sum is: 8

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();

Code; **package** Lambda\_functions;

**import** java.util.function.Predicate;

**import** java.util.Scanner;

**public** **class** String\_Empty {

**public** **static** **void** main(String[] args) {

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

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

String input=scan.nextLine();

Predicate<String> isEmpty=s->s.isEmpty();

**boolean** result = isEmpty.test(input);

System.***out***.println("String is empty: " + result);

scan.close();

}

}

Output; Enter a string:

jagadeesh

String is empty: false

1. Filter Even or Odd Numbers

Code; **package** Lambda\_functions;

**import** java.util.Scanner;

**interface** Even{

**boolean** iseven(**int** num);

}

**public** **class** LambdaExample {

**public** **static** **void** main(String[] args) {

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

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

**int** n=scan.nextInt();

Even check=(num)->num%2==0;

**if**(check.iseven(n)) {

System.***out***.println("Entered number is even number");

}**else** {

System.***out***.println("Entered number is odd number.");

}

scan.close();

}

}

Output; Enter a number:

7

Entered number is odd number.

1. Convert Strings to Uppercase/Lowercase

Code; **package** Lambda\_functions;

**import** java.util.Scanner;

**import** java.util.function.Function;

**public** **class** String\_conversion {

**public** **static** **void** main(String[] args) {

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

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

String s=scan.nextLine();

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

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

System.***out***.println("Uppercase: " + toUpperCase.apply(s));

System.***out***.println("Lowercase: " + toLowerCase.apply(s));

scan.close();

}

}

Output; Enter a string:

jagadeesh

Uppercase: JAGADEESH

Lowercase: jagadeesh

1. Sort Strings by Length or Alphabetically

Code; **package** Lambda\_functions;

**import** java.util.\*;

**public** **class** StringSort {

**public** **static** **void** main(String[] args) {

List<String> strings = Arrays.*asList*("banana", "apple", "kiwi", "mango", "blueberry");

List<String> byLength = **new** ArrayList<>(strings);

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

System.***out***.println("Sorted by length:");

byLength.forEach(System.***out***::println);

List<String> alphabetically = **new** ArrayList<>(strings);

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

System.***out***.println("\nSorted alphabetically:");

alphabetically.forEach(System.***out***::println);

}

}

Output; Sorted by length:

kiwi

apple

mango

banana

blueberry

Sorted alphabetically:

apple

banana

blueberry

kiwi

mango

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

Code; **package** Lambda\_functions;

**import** java.util.Arrays;

**public** **class** AggregateOperations {

**public** **static** **void** main(String[] args) {

**double**[] numbers = {12.5, 7.8, 19.3, 4.6, 10.0};

**double** sum = Arrays.*stream*(numbers).sum();

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

**double** max = Arrays.*stream*(numbers).max().orElse(Double.***NaN***);

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

**double** average = Arrays.*stream*(numbers).average().orElse(Double.***NaN***);

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

}

}

Output; Sum: 54.2

Max: 19.3

Average: 10.84

1. Create similar lambdas for max/min.

Code; **package** Lambda\_functions;

**import** java.util.Arrays;

**public** **class** MaxMinLambda {

**public** **static** **void** main(String[] args) {

**double**[] numbers = {12.5, 7.8, 19.3, 4.6, 10.0};

**double** maxValue = Arrays.*stream*(numbers).max().orElse(Double.***NaN***);

**double** minValue = Arrays.*stream*(numbers).min().orElse(Double.***NaN***);

System.***out***.println("Max (stream): " + maxValue);

System.***out***.println("Min (stream): " + minValue);

}

}

Output; Max (stream): 19.3

Min (stream): 4.6

1. Calculate Factorial

Code; **package** Lambda\_functions;

**import** java.util.stream.IntStream;

**import** java.util.Scanner;

**public** **class** FactorialLambda {

**public** **static** **void** main(String[] args) {

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

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

**int** num = scanner.nextInt();

**int** factorial = IntStream.*rangeClosed*(1, num)

.reduce(1, (a, b) -> a \* b);

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

}

}

Output; Enter a number to calculate factorial: 5

Factorial of 5 is: 120