**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 Practice\_assign;

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 {

System.out.println("Invalid marks, setting marks to 0");

this.marks = 0;

}

}

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);

}

// For future use: marks update only if new marks are higher or equal

public void inputMarks(int newMarks) {

if(newMarks >= 0 && newMarks <= 100) {

if(newMarks >= this.marks) {

this.marks = newMarks;

System.out.println("Marks updated to " + newMarks);

} else {

System.out.println("New marks cannot be less than current marks");

}

} else {

System.out.println("Invalid marks entered. Update failed");

}

}

}

public class TestStudent {

public static void main(String[] args) {

Student s1 = new Student("Alice", 101, 95);

s1.displayDetails();

System.out.println();

Student s2 = new Student("Bob", 102, 105); // Invalid marks

s2.displayDetails();

System.out.println();

s1.inputMarks(97); // Increase marks

s1.displayDetails();

System.out.println();

s1.inputMarks(90); // Attempt to reduce marks (should fail)

s1.displayDetails();

}

}

Output:

Name: Alice

Roll Number: 101

Marks: 95

Invalid marks, setting marks to 0

Name: Bob

Roll Number: 102

Marks: 0

Marks updated to 97

Name: Alice

Roll Number: 101

Marks: 97

New marks cannot be less than current marks

Name: Alice

Roll Number: 101

Marks: 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 Practice\_assign;

public class Rectangle {

private double width;

private double height;

public Rectangle(double width, double height) {

if (width > 0) {

this.width = width;

} else {

System.out.println("Invalid width, setting to 1");

this.width = 1;

}

if (height > 0) {

this.height = height;

} else {

System.out.println("Invalid height, setting to 1");

this.height = 1;

}

}

public void setWidth(double width) {

if (width > 0) {

this.width = width;

} else {

System.out.println("Invalid width. Width not changed.");

}

}

public void setHeight(double height) {

if (height > 0) {

this.height = height;

} else {

System.out.println("Invalid height. Height not changed.");

}

}

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 class Assign\_17 {

public static void main(String[] args) {

// Valid rectangle

Rectangle r1 = new Rectangle(5, 10);

r1.displayDetails();

System.out.println();

// Rectangle with invalid width and height (negative)

Rectangle r2 = new Rectangle(-3, 0);

r2.displayDetails();

System.out.println();

// Trying to set invalid width and height via setters

r2.setWidth(-4);

r2.setHeight(-5);

// Update with valid values

r2.setWidth(7);

r2.setHeight(8);

r2.displayDetails();

}

}

Output:

Width: 5.0

Height: 10.0

Area: 50.0

Perimeter: 30.0

Invalid width, setting to 1

Invalid height, setting to 1

Width: 1.0

Height: 1.0

Area: 1.0

Perimeter: 4.0

Invalid width. Width not changed.

Invalid height. Height not changed.

Width: 7.0

Height: 8.0

Area: 56.0

Perimeter: 30.0

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.

Code:

package Practice\_assign;

import java.util.ArrayList;

import java.util.List;

public class BankAccount {

private String accountNumber;

private String accountHolder;

private double balance;

// Inner class to represent a transaction

private class Transaction {

String type;

double amount;

Transaction(String type, double amount) {

this.type = type;

this.amount = amount;

}

@Override

public String toString() {

return type + ": " + amount;

}

}

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

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

this.accountNumber = accountNumber;

this.accountHolder = accountHolder;

if (initialBalance >= 0)

this.balance = initialBalance;

else

this.balance = 0;

}

public void deposit(double amount) {

if (amount <= 0) {

System.out.println("Deposit failed: Amount must be positive.");

return;

}

balance += amount;

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

System.out.println("Deposited: " + amount);

}

public boolean withdraw(double amount) {

if (amount <= 0) {

System.out.println("Withdrawal failed: Amount must be positive.");

return false;

}

if (amount > balance) {

System.out.println("Withdrawal failed: Insufficient funds.");

return false;

}

balance -= amount;

transactions.add(new Transaction("Withdrawal", amount));

System.out.println("Withdrawn: " + amount);

return true;

}

public double getBalance() {

return balance;

}

// Returns masked account number and account details

@Override

public String toString() {

String masked = "\*\*\*\*" + accountNumber.substring(accountNumber.length() - 4);

return "Account Number: " + masked + ", Account Holder: " + accountHolder + ", Balance: " + balance;

}

// Returns last transaction or null if none

public String getLastTransaction() {

if (transactions.isEmpty()) {

return "No transactions yet.";

}

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

}

}

package Practice\_assign;

public class Assign\_18 {

public static void main(String[] args) {

BankAccount account = new BankAccount("1234567890", "John Doe", 1000);

System.out.println(account);

account.deposit(500);

account.withdraw(200);

account.withdraw(2000); // Should fail

account.deposit(-50); // Should fail

System.out.println("Current balance: " + account.getBalance());

System.out.println("Last transaction: " + account.getLastTransaction());

System.out.println(account);

}

}

Output:

Account Number: \*\*\*\*7890,

Account Holder: John Doe, Balance: 1000.0

Deposited: 500.0

Withdrawn: 200.0

Withdrawal failed: Insufficient funds.

Deposit failed: Amount must be positive.

Current balance: 1300.0

Last transaction: Withdrawal: 200.0

Account Number: \*\*\*\*7890, Account Holder: John Doe, Balance: 1300.0

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 Practice\_assign;

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 = true; // starts locked

}

// Inner private class to handle passcode verification

private class SecurityManager {

boolean verifyPasscode(String inputCode) {

return passcode.equals(inputCode);

}

}

private SecurityManager securityManager = new SecurityManager();

public void lock() {

isLocked = true;

System.out.println("Locker " + lockerId + " locked.");

}

public boolean unlock(String code) {

if(securityManager.verifyPasscode(code)) {

isLocked = false;

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

return true;

} else {

System.out.println("Incorrect passcode. Unlock failed.");

return false;

}

}

public boolean isLocked() {

return isLocked;

}

}

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 empty");

}

this.name = name;

return this;

}

public Builder withCode(String code) {

if(code == null || code.isEmpty()) {

throw new IllegalArgumentException("Code cannot be 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; // optional, can be null or empty

return this;

}

public Product build() {

if(name == null || code == null) {

throw new IllegalStateException("Name and Code are required");

}

return new Product(this);

}

}

}

public class Assign\_19 {

public static void main(String[] args) {

// Testing Locker

Locker locker = new Locker("L123", "pass123");

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

locker.unlock("wrongpass"); // fail

locker.unlock("pass123"); // success

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

locker.lock();

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

System.out.println();

// Testing Product Builder

try {

Product p = new Product.Builder()

.withName("Laptop")

.withCode("LP100")

.withPrice(1200.50)

.withCategory("Electronics")

.build();

System.out.println("Product created:");

System.out.println("Name: " + p.getName());

System.out.println("Code: " + p.getCode());

System.out.println("Price: " + p.getPrice());

System.out.println("Category: " + p.getCategory());

} catch (Exception e) {

System.out.println("Failed to create product: " + e.getMessage());

}

}

}

Locker 101 is now locked.

Unlock failed! Incorrect passcode.

Locker 101 is now unlocked.

Product Details:

Name: Laptop

Code: P123

Price: 59999.99

Category: Electronics

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.

**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.

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.

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.

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.

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.

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:

import java.time.LocalDateTime;

import java.util.ArrayList;

import java.util.List;

// 1. Builder Pattern & Encapsulation: Immutable Product

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; }

@Override

public String toString() {

return "Product: " + name + " (Code: " + code + ", Price: " + price +

", Category: " + (category != null ? category : "N/A") + ")";

}

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.isBlank())

throw new IllegalArgumentException("Name required");

this.name = name;

return this;

}

public Builder withCode(String code) {

if (code == null || code.isBlank())

throw new IllegalArgumentException("Code required");

this.code = code;

return this;

}

public Builder withPrice(double price) {

if (price < 0)

throw new IllegalArgumentException("Price must be non-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 & Code are mandatory");

return new Product(this);

}

}

}

// 1.2 Reverse CharSequence

class BackwardSequence implements CharSequence {

private final String reversed;

public BackwardSequence(String input) {

if (input == null) throw new IllegalArgumentException("Null string");

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

}

@Override

public int length() { return reversed.length(); }

@Override

public char charAt(int index) { return reversed.charAt(index); }

@Override

public CharSequence subSequence(int start, int end) {

return reversed.subSequence(start, end);

}

@Override

public String toString() { return reversed; }

}

// 2. Moveable Shapes Simulation

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;

}

@Override public void moveUp() { y -= ySpeed; }

@Override public void moveDown() { y += ySpeed; }

@Override public void moveLeft() { x -= xSpeed; }

@Override public void moveRight() { x += xSpeed; }

@Override public String toString() { return "(" + x + "," + y + ")"; }

}

class MovableCircle implements Movable {

int radius;

MovablePoint center;

public MovableCircle(int radius, MovablePoint center) {

this.radius = radius; this.center = center;

}

@Override public void moveUp() { center.moveUp(); }

@Override public void moveDown() { center.moveDown(); }

@Override public void moveLeft() { center.moveLeft(); }

@Override public void moveRight() { center.moveRight(); }

@Override public String toString() {

return "Circle center=" + center + " radius=" + radius;

}

}

class MovableRectangle implements Movable {

MovablePoint topLeft, 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;

}

@Override public void moveUp() { topLeft.moveUp(); bottomRight.moveUp(); }

@Override public void moveDown() { topLeft.moveDown(); bottomRight.moveDown(); }

@Override public void moveLeft() { topLeft.moveLeft(); bottomRight.moveLeft(); }

@Override public void moveRight() { topLeft.moveRight(); bottomRight.moveRight(); }

@Override public String toString() {

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

}

}

// 3. Contract Programming: Printer Switch

interface Printer {

void print(String document);

}

class LaserPrinter implements Printer {

@Override public void print(String document) {

System.out.println("[LaserPrinter] Printing: " + document);

}

}

class InkjetPrinter implements Printer {

@Override public void print(String document) {

System.out.println("[InkjetPrinter] Printing: " + document);

}

}

// 4. Extended Interface Hierarchy

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; }

@Override public void start() {

if (fuel > 0) System.out.println("Car started");

else System.out.println("No fuel to start");

}

@Override public void stop() {

System.out.println("Car stopped");

}

@Override public boolean refuel(int amount) {

if (amount > 0) {

fuel += amount;

System.out.println("Refueled " + amount + " liters");

return true;

}

return false;

}

}

// 5. Nested Interface for Callback Handling

class TimeServer {

public static interface Client {

void updateTime(LocalDateTime now);

}

private final List<Client> clients = new ArrayList<>();

public void registerClient(Client c) { clients.add(c); }

public void notifyClients() {

LocalDateTime now = LocalDateTime.now();

for (Client c : clients) {

c.updateTime(now);

}

}

}

class ClockDisplay implements TimeServer.Client {

@Override public void updateTime(LocalDateTime now) {

System.out.println("[ClockDisplay] Time: " + now);

}

}

class LoggerDisplay implements TimeServer.Client {

@Override public void updateTime(LocalDateTime now) {

System.out.println("[Logger] Logged time: " + now);

}

}

// 6. Default and Static Methods in Interfaces

interface Polygon {

double getArea();

default double getPerimeter(int... sides) {

double sum = 0;

for (int s : sides) sum += s;

return sum;

}

static String shapeInfo() {

return "Polygon: a flat shape with straight sides.";

}

}

class RectanglePoly implements Polygon {

private int length, width;

public RectanglePoly(int l, int w) { this.length = l; this.width = w; }

@Override public double getArea() { return length \* width; }

}

class TrianglePoly implements Polygon {

private int base, height;

public TrianglePoly(int b, int h) { this.base = b; this.height = h; }

@Override public double getArea() { return 0.5 \* base \* height; }

}

// Main class to test all parts

public class Main {

public static void main(String[] args) {

// 1. Product Builder

Product product = new Product.Builder()

.withName("Laptop")

.withCode("X123")

.withPrice(75000)

.withCategory("Electronics")

.build();

System.out.println(product);

// 1.2 BackwardSequence

BackwardSequence bs = new BackwardSequence("hello");

System.out.println("Backward: " + bs);

System.out.println("Char at 1: " + bs.charAt(1));

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

// 2. Movable shapes

MovableCircle circle = new MovableCircle(5, new MovablePoint(0,0,2,2));

circle.moveRight(); circle.moveUp();

System.out.println(circle);

MovableRectangle rect = new MovableRectangle(

new MovablePoint(0,0,1,1),

new MovablePoint(4,4,1,1));

rect.moveDown(); rect.moveRight();

System.out.println(rect);

// 3. Printer switch

Printer p = new LaserPrinter();

p.print("Test Page 1");

p = new InkjetPrinter();

p.print("Test Page 2");

// 4. Interface hierarchy

AdvancedVehicle car = new Car(5);

car.start();

car.stop();

car.refuel(10);

// 5. TimeServer callbacks

TimeServer server = new TimeServer();

server.registerClient(new ClockDisplay());

server.registerClient(new LoggerDisplay());

server.notifyClients();

// 6. Polygon default & static methods

Polygon rectPoly = new RectanglePoly(4,5);

System.out.println("Rect Area: " + rectPoly.getArea());

System.out.println("Rect Perimeter: " + rectPoly.getPerimeter(4,5,4,5));

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

Polygon triPoly = new TrianglePoly(6,4);

System.out.println("Triangle Area: " + triPoly.getArea());

}

}

Output:

Product: Laptop (Code: X123, Price: 75000.0, Category: Electronics)

Backward: olleh

Char at 1: l

SubSeq(1,4): lle

Circle center=(2,-2) radius=5

Rect TL=(1,1) BR=(5,5)

[LaserPrinter] Printing: Test Page 1

[InkjetPrinter] Printing: Test Page 2

Car started

Car stopped

Refueled 10 liters

[ClockDisplay] Time: 2025-08-10T12:34:56.789

[Logger] Logged time: 2025-08-10T12:34:56.789

Rect Area: 20.0

Rect Perimeter: 18.0

Polygon: a flat shape with straight sides.

Triangle Area: 12.0

**Lambda expressions**

1. Sum of Two Integers

Code:

@FunctionalInterface

interface SumCalculator {

int sum(int a, int b);

}

public class LambdaSumExample {

public static void main(String[] args) {

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

int num1 = 5;

int num2 = 7;

int result = sumCalc.sum(num1, num2);

System.out.println("Sum of " + num1 + " and " + num2 + " is: " + result);

}

}

Output:

Sum of 5 and 7 is: 12

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

Code:

@FunctionalInterface

interface SumCalculator {

int sum(int a, int b);

}

public class LambdaSumCalculator {

public static void main(String[] args) {

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

int num1 = 10;

int num2 = 20;

int result = sumCalc.sum(num1, num2);

System.out.println("Sum of " + num1 + " and " + num2 + " is: " + result);

}

}

Output:

Sum of 10 and 20 is: 30

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:

import java.util.function.Predicate;

public class LambdaCheckEmptyString {

public static void main(String[] args) {

// Lambda expression to check if a string is empty

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

String str1 = "";

String str2 = "Hello";

System.out.println("Is str1 empty? " + isEmpty.test(str1)); // true

System.out.println("Is str2 empty? " + isEmpty.test(str2)); // false

}

}

Output:

Is str1 empty? true

Is str2 empty? False

1. Filter Even or Odd Numbers

Code:

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class LambdaFilterEvenOdd {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Filter even numbers

List<Integer> evens = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

// Filter odd numbers

List<Integer> odds = numbers.stream()

.filter(n -> n % 2 != 0)

.collect(Collectors.toList());

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

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

}

}

Output:

Even Numbers: [2, 4, 6, 8, 10]

Odd Numbers: [1, 3, 5, 7, 9]

1. Convert Strings to Uppercase/Lowercase

Code:

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class LambdaStringCase {

public static void main(String[] args) {

List<String> words = Arrays.asList("hello", "world", "java", "lambda");

List<String> upperCaseWords = words.stream()

.map(s -> s.toUpperCase())

.collect(Collectors.toList());

// Convert to Lowercase

List<String> lowerCaseWords = words.stream()

.map(s -> s.toLowerCase())

.collect(Collectors.toList());

System.out.println("Uppercase: " + upperCaseWords);

System.out.println("Lowercase: " + lowerCaseWords);

}

}

Output:

Uppercase: [HELLO, WORLD, JAVA, LAMBDA]

Lowercase: [hello, world, java, lambda]

1. Sort Strings by Length or Alphabetically

Code:

import java.util.Arrays;

import java.util.List;

public class LambdaSortStrings {

public static void main(String[] args) {

List<String> words = Arrays.asList("banana", "apple", "kiwi", "orange", "grape");

// Sort by length

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

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

// Sort alphabetically

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

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

}

}

Output:

Sorted by length: [kiwi, apple, grape, banana, orange]

Sorted alphabetically: [apple, banana, grape, kiwi, orange]

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

Code:

import java.util.Arrays;

public class LambdaAggregateOperations {

public static void main(String[] args) {

double[] numbers = {4.5, 2.3, 9.7, 1.4, 5.9};

// Sum

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

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

// Max

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

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

// Average

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

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

}

}

Output:

Sum: 23.8

Max: 9.7

Average: 4.76

1. Create similar lambdas for max/min.

Code:

import java.util.function.BiFunction;

public class LambdaMaxMin {

public static void main(String[] args) {

// Lambda to find maximum of two numbers

BiFunction<Integer, Integer, Integer> max = (a, b) -> (a > b) ? a : b;

// Lambda to find minimum of two numbers

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

int num1 = 15;

int num2 = 25;

System.out.println("Max: " + max.apply(num1, num2));

System.out.println("Min: " + min.apply(num1, num2));

}

}

Output:

Max: 25

Min: 15

1. Calculate Factorial

Code:

@FunctionalInterface

interface FactorialCalculator {

long factorial(int n);

}

public class LambdaFactorial {

public static void main(String[] args) {

// Lambda to calculate factorial

FactorialCalculator factorial = (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: " + factorial.factorial(number));

}

}

Output:

Factorial of 5 is: 120