1st exercise SINGLETON PATTERN

public class Singleton {

private static Singleton instance = null;

private Singleton() {

if (instance != null) {

throw new RuntimeException("Use getInstance() method to get the single instance.");

}

}

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

public void setValue(String value) {

System.out.println("Setting value: " + value);

}

public static void main(String[] args) {

Singleton singleton1 = Singleton.getInstance();

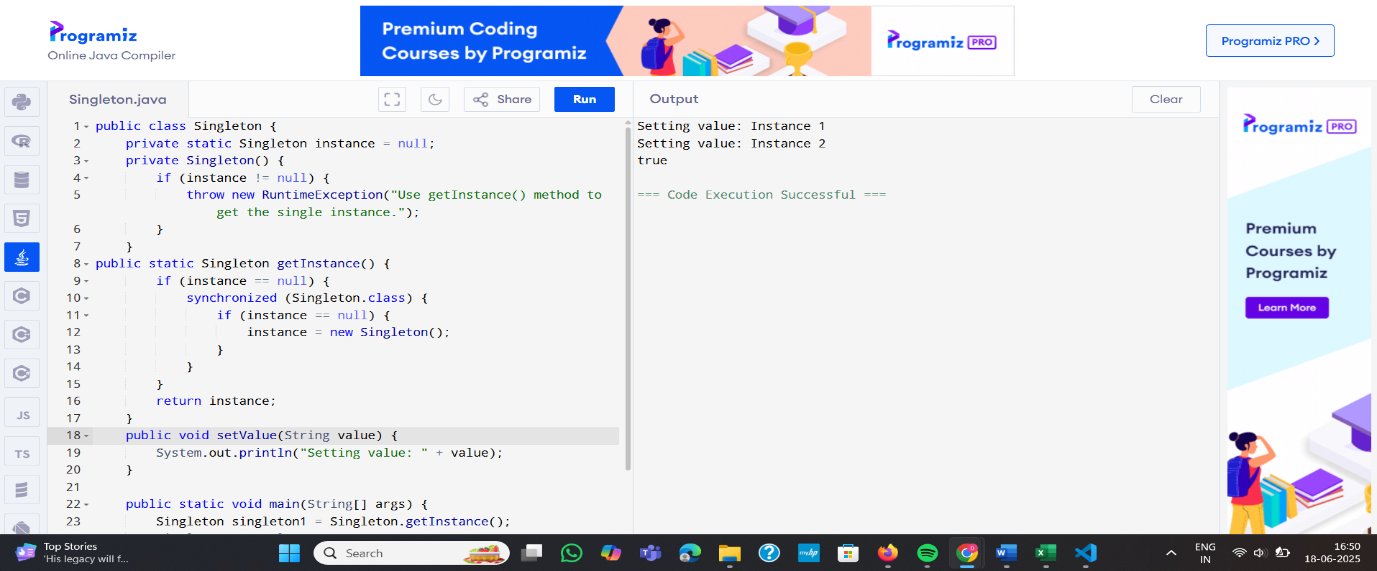
singleton1.setValue("Instance 1");

Singleton singleton2 = Singleton.getInstance();

singleton2.setValue("Instance 2");

System.out.println(singleton1 == singleton2);

}}



2nd exercise FACTORY METHOD

public class FactoryMethodDemo {

public static void main(String[] args) {

Creator creatorA = new ConcreteCreatorA();

Product productA = creatorA.factoryMethod();

productA.use();

Creator creatorB = new ConcreteCreatorB();

Product productB = creatorB.factoryMethod();

productB.use();

}

}interface Product {

void use();

}

class ConcreteProductA implements Product {

public void use() {

System.out.println("Using Product A");

}

}

class ConcreteProductB implements Product {

public void use() {

System.out.println("Using Product B");

}

}

abstract class Creator {

abstract Product factoryMethod();}

class ConcreteCreatorA extends Creator {

Product factoryMethod() {

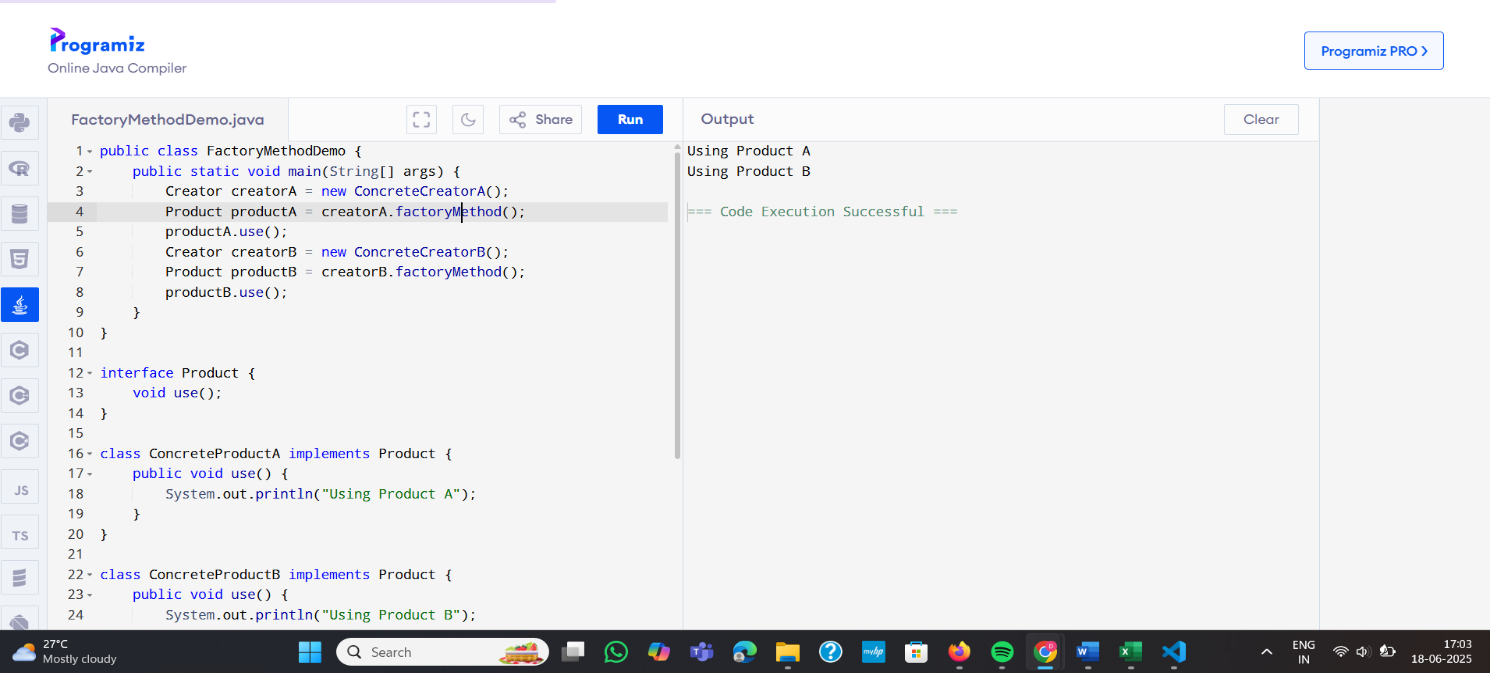
return new ConcreteProductA();

}

}

class ConcreteCreatorB extends Creator {

Product factoryMethod() {

return new ConcreteProductB();}

3RD exercise DSA

import java.util.\*;

import java.util.stream.Collectors;

public class EcommerceSearchDemo {

public static void main(String[] args) {

SearchService searchService = new SearchService();

System.out.println("Search by name 'phone':");

List<Product> nameResults = searchService.searchByName("phone");

nameResults.forEach(System.out::println);

System.out.println("\nSearch by category 'Clothing':");

List<Product> categoryResults = searchService.searchByCategory("Clothing");

categoryResults.forEach(System.out::println);

}

}

class Product {

private String name;

private String category;

private double price;

public Product(String name, String category, double price) {

this.name = name;

this.category = category;

this.price = price;

}

public String getName() {

return name;

}

public String getCategory() {

return category;

}

public double getPrice() {

return price;

}

@Override

public String toString() {

return "Product{name='" + name + "', category='" + category + "', price=" + price + "}";

}

}

class SearchService {

private List<Product> products;

private Map<String, List<Product>> categoryIndex;

public SearchService() {

products = new ArrayList<>();

products.add(new Product("Laptop", "Electronics", 999.99));

products.add(new Product("Smartphone", "Electronics", 499.99));

products.add(new Product("T-Shirt", "Clothing", 19.99));

products.add(new Product("Jeans", "Clothing", 39.99));

buildCategoryIndex();

}private void buildCategoryIndex() {

categoryIndex = new HashMap<>();

for (Product product : products) {

String category = product.getCategory().toLowerCase();

categoryIndex.computeIfAbsent(category, k -> new ArrayList<>()).add(product)}

} public List<Product> searchByName(String query) {

String lowerQuery = query.toLowerCase();

return products.stream()

.filter(p -> p.getName().toLowerCase().contains(lowerQuery))

.collect(Collectors.toList());

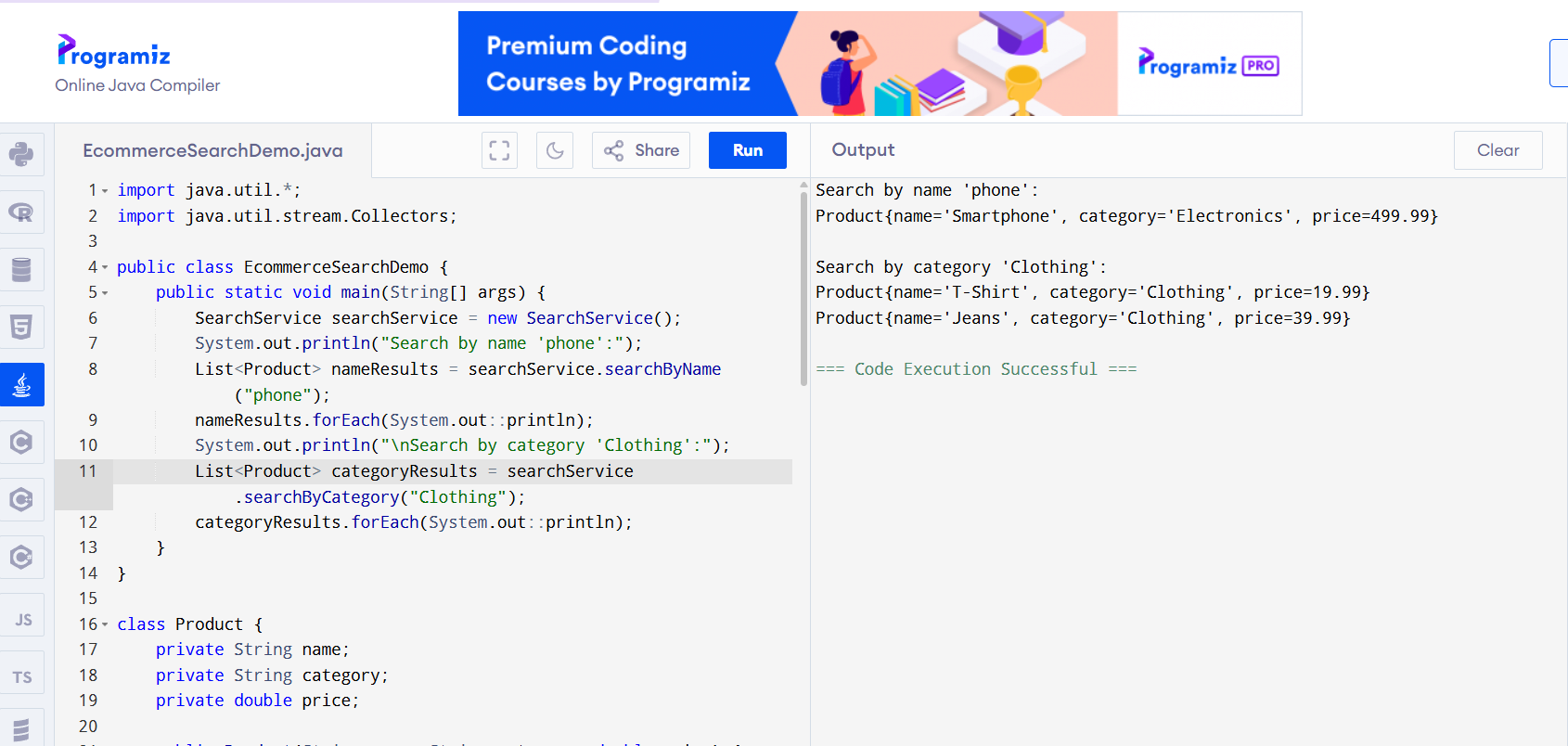
}

public List<Product> searchByCategory(String category) {

return categoryIndex.getOrDefault(category.toLowerCase(), new ArrayList<>());

}

}



4th exercise

FinancialForecast

public class FinancialForecastDemo {

public static void main(String[] args) {

FinancialForecaster forecaster = new FinancialForecaster();

List<Double> historicalData = Arrays.asList(100.0, 105.0, 110.0, 115.0, 120.0);

int forecastPeriods = 3;

try {

List<Double> forecast = forecaster.forecast(historicalData, forecastPeriods);

System.out.println("Historical data: " + historicalData);

System.out.println("Forecast for next " + forecastPeriods + " periods: " + forecast);

List<Double> sortedData = new ArrayList<>(historicalData);

Collections.sort(sortedData);

System.out.println("Sorted historical data: " + sortedData);

double key = 110.0;

int index = Collections.binarySearch(sortedData, key);

System.out.println("Binary search for " + key + ": " + (index >= 0 ? "Found at index " + index : "Not found"));

} catch (IllegalArgumentException e) {

System.out.println("Error: " + e.getMessage());

}

}

}

class FinancialForecaster {

public List<Double> forecast(List<Double> historicalData, int periods) {

if (historicalData == null || historicalData.size() < 2 || periods <= 0) {

throw new IllegalArgumentException("Invalid input: At least 2 data points required and periods must be positive.");

} int n = historicalData.size();

double sumX = 0, sumY = 0, sumXY = 0, sumXX = 0;

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

double y = historicalData.get(i);

sumX += i;

sumY += y;

sumXY += i \* y;

sumXX += i \* i;

}double meanX = sumX / n;

double meanY = sumY / n;

double m = (sumXY - sumX \* meanY) / (sumXX - sumX \* meanX);

double b = meanY - m \* meanX;

List<Double> forecast = new ArrayList<>();

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

forecast.add(m \* (n + i) + b);

}

return forecast;

}

}

