**Week 1**

**Design Patterns and Principles**

**Exercise 1**: Singleton Pattern Implementation: *SingletonPatternExample*

*Code:*

***// Logger.java***

public class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger Initialized");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // Lazy initialization

}

return instance;

}

public void log(String message) {

System.out.println("Log: " + message);

}

}

***// Main.java***

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("First log message");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message");

// Test if both logger references point to the same instance

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 are the same instance.");

} else {

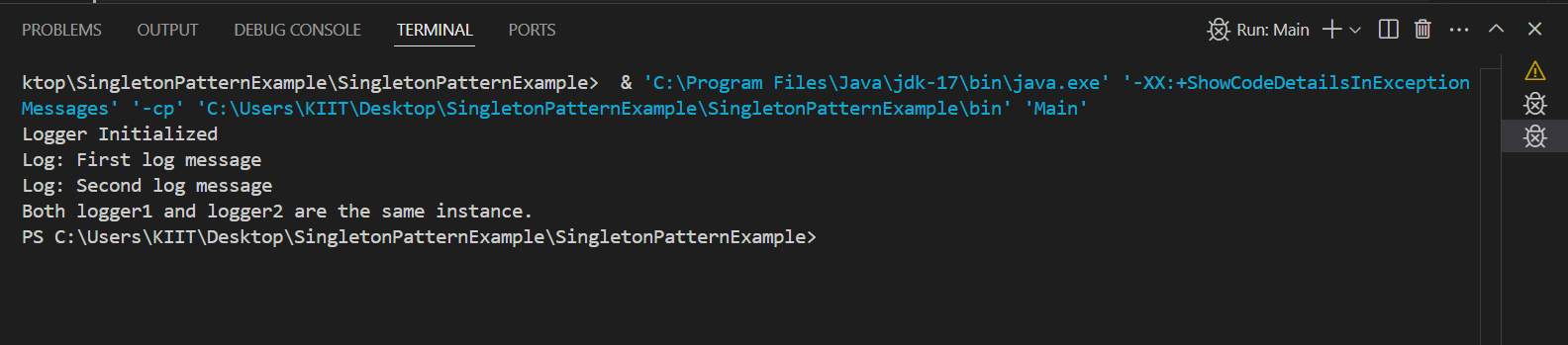
System.out.println("Different instances created!");

}

}

}

*Output:*

**

**Exercise 2**: Factory Method Pattern Implementation: *FactoryMethodPatternExample*

*Code:*

***// Document.java***

public interface Document {

void open();

}

***// WordDocument.java***

public class WordDocument implements Document {

@Override

public void open() {

System.out.println("Opening Word document...");

}

}

***// PdfDocument.java***

public class PdfDocument implements Document {

@Override

public void open() {

System.out.println("Opening PDF document...");

}

}

***// ExcelDocument.java***

public class ExcelDocument implements Document {

@Override

public void open() {

System.out.println("Opening Excel document...");

}

}

***// DocumentFactory.java***

public abstract class DocumentFactory {

public abstract Document createDocument();

}

***// WordDocumentFactory.java***

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

***// PdfDocumentFactory.java***

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

***// ExcelDocumentFactory.java***

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

***// Main.java***

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

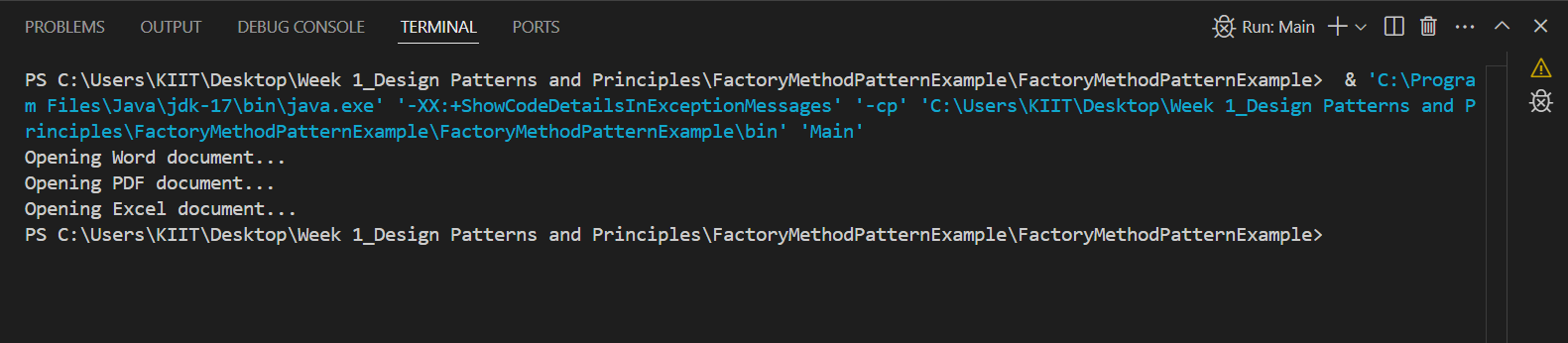
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

*Output:*



**Exercise 3**: Builder Pattern Implementation: *BuliderPatternExample*

*Code:*

***// Computer.java***

public class Computer {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

// Static nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void displayConfig() {

System.out.println("CPU: " + CPU);

System.out.println("RAM: " + RAM);

System.out.println("Storage: " + (storage != null ? storage : "Not included"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not included"));

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

}

}

***// Main.java***

public class Main {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder("Intel i5", "8GB").build();

basicComputer.displayConfig();

Computer gamingComputer = new Computer.Builder("AMD Ryzen 7", "16GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 3060")

.build();

gamingComputer.displayConfig();

Computer officeComputer = new Computer.Builder("Intel i7", "16GB")

.setStorage("512GB SSD")

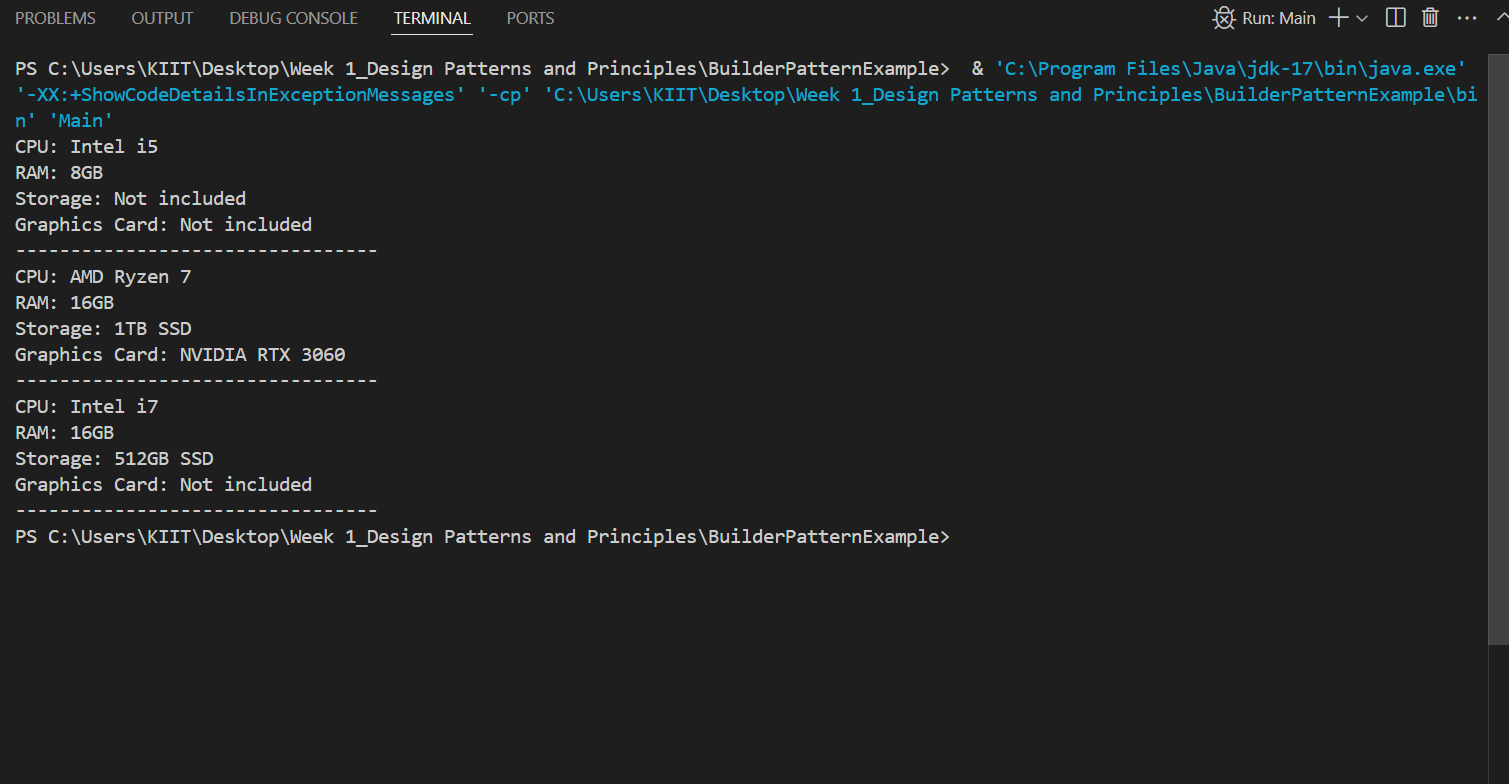
.build();

officeComputer.displayConfig();

}

}

*Output:*



**Data Structures and Algorithms**

**Exercise 2**: E-commerce Platform Search Function

*Code:*

***// Product class setup***

public class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return productId + " - " + productName + " (" + category + ")";

}

}

***// search algorithm implementation***

import java.util.Arrays;

import java.util.Comparator;

public class ECommerceSearch {

// Linear Search

public static Product linearSearch(Product[] products, String targetName) {

for (Product product : products) {

if (product.productName.equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

// Binary Search

public static Product binarySearch(Product[] products, String targetName) {

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

int result = products[mid].productName.compareToIgnoreCase(targetName);

if (result == 0)

return products[mid];

else if (result < 0)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shoes", "Footwear"),

new Product(103, "Mobile", "Electronics"),

new Product(104, "T-Shirt", "Apparel"),

new Product(105, "Watch", "Accessories")

};

// Linear Search Example

System.out.println("🔍 Linear Search:");

Product found1 = linearSearch(products, "Mobile");

System.out.println(found1 != null ? "Found: " + found1 : "Product not found");

// Sort for Binary Search

Arrays.sort(products, Comparator.comparing(p -> p.productName));

// Binary Search Example

System.out.println("\n🔍 Binary Search:");

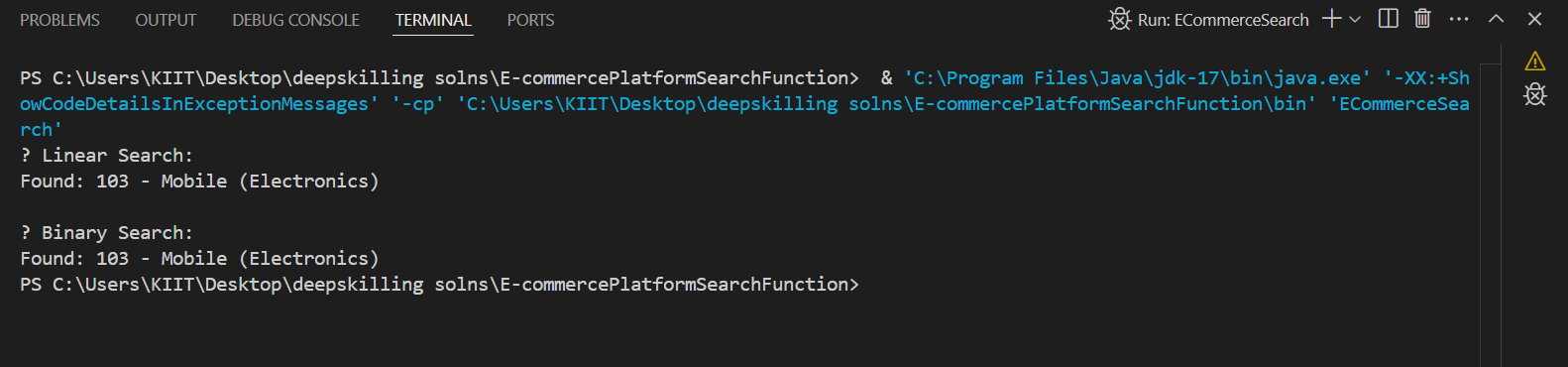
Product found2 = binarySearch(products, "Mobile");

System.out.println(found2 != null ? "Found: " + found2 : "Product not found");

}

}

*Output:*



*Time complexity analysis:*

For linear search, the best case is O(1), average case is O(n/2) and the worst case is O(n).

For binary search, the best case is O(1), average case is O(log n) and the worst case is O(log n).

Hence, the time complexity of binary search is faster.

**Exercise 7**: Financial Forecasting

*Code:*

public class FinancialForecast {

*// Recursive function to calculate future value*

public static double futureValueRecursive(double initialAmount, double growthRate, int years) {

if (years == 0) {

return initialAmount;

}

return futureValueRecursive(initialAmount, growthRate, years - 1) \* (1 + growthRate);

}

*// Optimized version using Memoization*

public static double futureValueMemoized(double initialAmount, double growthRate, int years, Double[] memo) {

if (years == 0) {

return initialAmount;

}

if (memo[years] != null) {

return memo[years];

}

memo[years] = futureValueMemoized(initialAmount, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

public static void main(String[] args) {

double initialInvestment = 10000; // Initial amount

double annualGrowthRate = 0.10; // 10% growth

int years = 5; // Forecasting for 5 years

*// Recursive approach*

double resultRecursive = futureValueRecursive(initialInvestment, annualGrowthRate, years);

System.out.printf("Future Value (Recursive): %.2f%n", resultRecursive);

*// Optimized with memoization*

Double[] memo = new Double[years + 1];

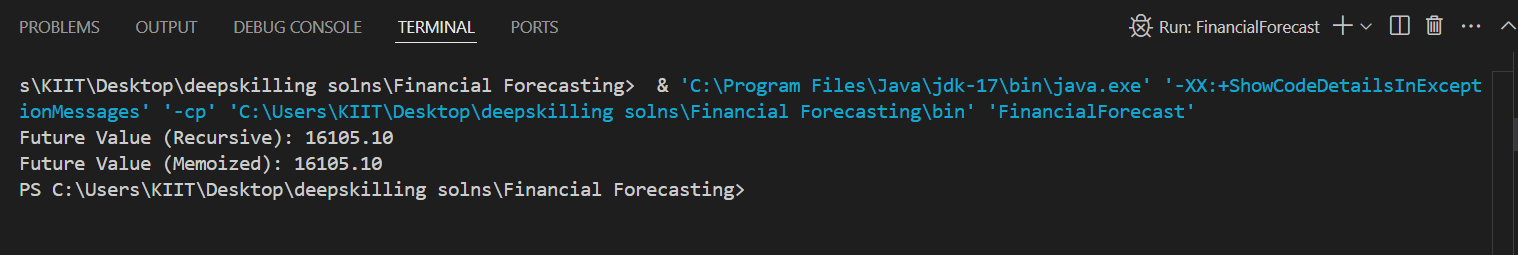
double resultMemoized = futureValueMemoized(initialInvestment, annualGrowthRate, years, memo);

System.out.printf("Future Value (Memoized): %.2f%n", resultMemoized);

}

}

*Output:*

**

*Time Complexity Analysis:*

Naive Recursion: O(n) — Each year calculation requires calling the previous one.

Memoized Version: Still O(n) — but with no repeated calculations, faster in practice.