**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

package week1;

public class Logger {

private static Logger *inst*;

private Logger() {

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

}

public static Logger getInst() {

if (*inst* == null) {

*inst* = new Logger();

}

return *inst*;

}

public void log(String msg) {

System.*out*.println("Log: " + msg);

}

public static void main(String[] args) {

Logger l1 = Logger.*getInst*();

Logger l2 = Logger.*getInst*();

l1.log("First message.");

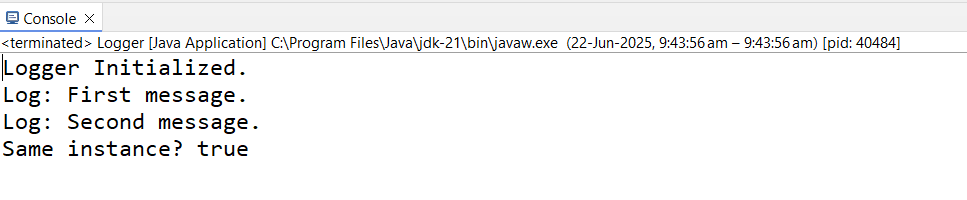
l2.log("Second message.");

System.*out*.println("Same instance? " + (l1 == l2));

}

}

* **OUTPUT:**



**Exercise 2: Implementing the Factory Method Pattern**

interface Document {

void open();

}

class WordDoc implements Document {

public void open() {

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

}

}

class PdfDoc implements Document {

public void open() {

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

}

}

class ExcelDoc implements Document {

public void open() {

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

}

}

abstract class DocFactory {

public abstract Document create();

}

class WordFactory extends DocFactory {

public Document create() {

return new WordDoc();

}

}

class PdfFactory extends DocFactory {

public Document create() {

return new PdfDoc();

}

}

class ExcelFactory extends DocFactory {

public Document create() {

return new ExcelDoc();

}

}

public class FactoryMethodDemo {

public static void main(String[] args) {

DocFactory f1 = new WordFactory();

Document d1 = f1.create();

d1.open();

DocFactory f2 = new PdfFactory();

Document d2 = f2.create();

d2.open();

DocFactory f3 = new ExcelFactory();

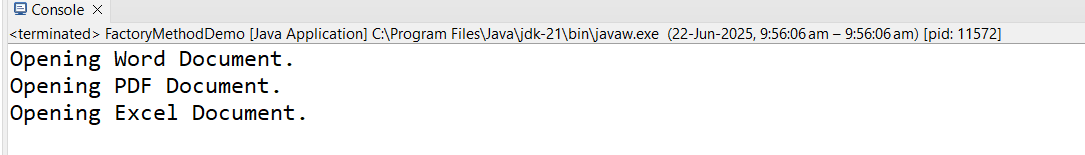
Document d3 = f3.create();

d3.open();

}

}

* **OUTPUT:**



**Algorithms\_Data Structures**

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

package week1;

import java.util.\*;

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;

}

public String toString() {

return "[" + productId + ", " + productName + ", " + category + "]";

}

}

public class ECommerceSearch {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

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

while (low <= high) {

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

int cmp = products[mid].productName.compareToIgnoreCase(name);

if (cmp == 0) return products[mid];

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void sortByName(Product[] products) {

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

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Fashion"),

new Product(3, "Phone", "Electronics"),

new Product(4, "T-shirt", "Fashion")

};

System.*out*.println("Linear Search: " + *linearSearch*(products, "Phone"));

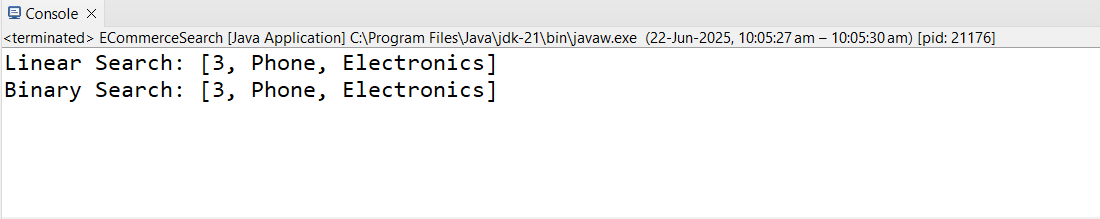
*sortByName*(products);

System.*out*.println("Binary Search: " + *binarySearch*(products, "Phone"));

}

}

* **OUTPUT:**



**Exercise 7: Financial Forecasting**

package week1;

public class FinancialForecasting {

public static void main(String[] args) {

double currentValue = 1000.0;

double[] growthRates = {0.05, 0.03, 0.04, 0.02};

double futureValue = *calculateFutureValue*(currentValue, growthRates);

System.*out*.printf("Future value after %d periods: $%.2f%n",

growthRates.length, futureValue);

}

public static double calculateFutureValue(double currentValue, double[] growthRates) {

return *calculateFutureValueRecursive*(currentValue, growthRates, 0);

}

private static double calculateFutureValueRecursive(double currentValue, double[] growthRate int index) {

if (index >= growthRates.length) {

return currentValue;

}

double nextValue = currentValue \* (1 + growthRates[index]);

return *calculateFutureValueRecursive*(nextValue, growthRates, index + 1);

}

}

* **OUTPUT:**

