**WEEK - 1**

**Design principles & Patterns**

1) Exercise 1: Implementing the Singleton Pattern

CODE:

public class Logger {

private static Logger *instance*;

private Logger() {

System.***out***.println("Logger instance created.");

}

public static Logger getInstance() {

if (*instance* == null) {

synchronized (Logger.class) {

if (*instance* == null) {

*instance* = new Logger();

}

}

}

return *instance*;

}

public void log(String message) {

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

}

}

public class SingletonTest {

public static void main(String[] args) {

System.***out***.println("--- Testing Singleton Logger ---");

Logger logger1 = Logger.*getInstance*();

logger1.log("First log message from logger1.");

Logger logger2 = Logger.*getInstance*();

logger2.log("Second log message from logger2.");

Logger logger3 = Logger.*getInstance*();

logger3.log("Third log message from logger3.");

System.***out***.println("\nVerifying instances:");

System.***out***.println("logger1 == logger2: " + (logger1 == logger2));

System.***out***.println("logger1 == logger3: " + (logger1 == logger3));

System.***out***.println("logger2 == logger3: " + (logger2 == logger3));

}

}

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

2) Exercise 2: Implementing the Factory Method Pattern

CODE:

public interface Document {

void open();

void save();

void close();

}

public abstract class DocumentFactory {

public abstract Document createDocument();

public void processDocument() {

Document doc = createDocument(); // Call the factory method

doc.open();

doc.save();

doc.close();

System.***out***.println("Document processed successfully.\n");

}

}

public class WordDocument implements Document {

*@Override*

public void open() {

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

}

*@Override*

public void save() {

System.***out***.println("Saving Word document.");

}

*@Override*

public void close() {

System.***out***.println("Closing Word document.");

}

}

public class WordDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

System.***out***.println("Creating Word document...");

return new WordDocument();

}

}

public class ExcelDocument implements Document {

*@Override*

public void open() {

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

}

*@Override*

public void save() {

System.***out***.println("Saving Excel document.");

}

*@Override*

public void close() {

System.***out***.println("Closing Excel document.");

}

}

public class ExcelDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

System.***out***.println("Creating Excel document...");

return new ExcelDocument();

}

}

public class PdfDocument implements Document {

*@Override*

public void open() {

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

}

*@Override*

public void save() {

System.***out***.println("Saving PDF document.");

}

*@Override*

public void close() {

System.***out***.println("Closing PDF document.");

}

}

public class PdfDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

System.***out***.println("Creating PDF document...");

return new PdfDocument();

}

}

public class FactoryMethodTest {

public static void main(String[] args) {

System.***out***.println("--- Demonstrating Factory Method Pattern ---");

DocumentFactory wordFactory = new WordDocumentFactory();

System.***out***.println("Using Word Document Factory:");

wordFactory.processDocument();

DocumentFactory pdfFactory = new PdfDocumentFactory();

System.***out***.println("Using PDF Document Factory:");

pdfFactory.processDocument();

DocumentFactory excelFactory = new ExcelDocumentFactory();

System.***out***.println("Using Excel Document Factory:");

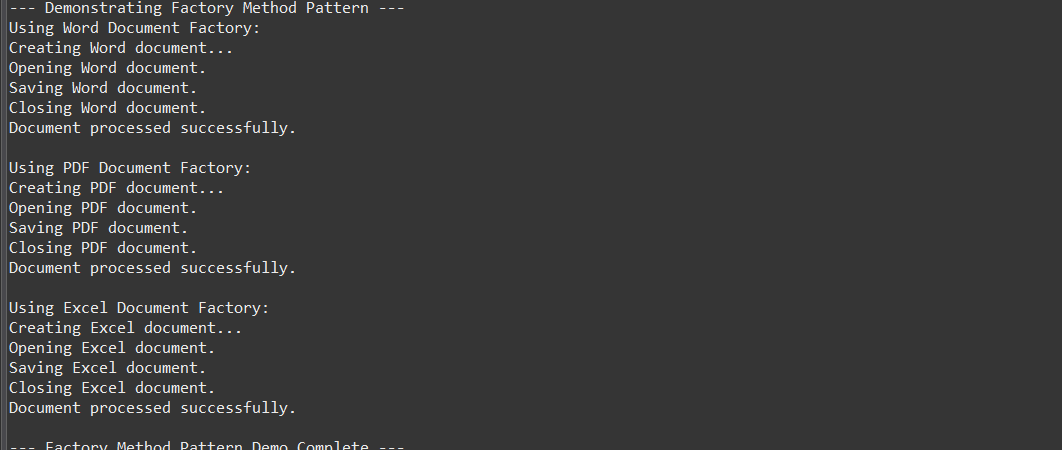
excelFactory.processDocument();

System.***out***.println("--- Factory Method Pattern Demo Complete ---");

}

}

OUTPUT:



3) Exercise 2: E-commerce Platform Search Function

CODE:

import java.util.Objects; // For Objects.hash and Objects.equals

public class Product implements Comparable<Product> {

private String productId;

private String productName;

private String category;

private double price; // Adding price for more realistic data

public Product(String productId, String productName, String category, double price) {

this.productId = productId;

this.productName = productName;

this.category = category;

this.price = price;

}

public String getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

public double getPrice() {

return price;

}

*@Override*

public String toString() {

return "Product [ID=" + productId + ", Name=" + productName + ", Category=" + category + ", Price=" + price + "]";

}

*@Override*

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Product product = (Product) o;

return Objects.*equals*(productId, product.productId);

}

*@Override*

public int hashCode() {

return Objects.*hash*(productId);

}

*@Override*

public int compareTo(Product other) {

return this.productId.compareTo(other.productId);

}

}

import java.util.Arrays;

public class SearchAlgorithms {

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

for (int i = 0; i < products.length; i++) {

if (products[i].getProductId().equals(productIdToFind)) {

return products[i];

}

}

return null;

}

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

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

int comparison = productIdToFind.compareTo(products[mid].getProductId());

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return null;

}

public static Product[] generateProducts(int count) {

Product[] products = new Product[count];

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

products[i] = new Product("P" + String.*format*("%05d", i), "Product " + i, "Category " + (i % 5), 10.0 + i);

}

return products;

}

}

import java.util.Arrays;

import java.util.Random;

public class SearchAnalysis {

public static void main(String[] args) {

System.***out***.println("--- Search Algorithm Analysis ---");

int numberOfProducts = 100000;

Product[] allProducts = SearchAlgorithms.*generateProducts*(numberOfProducts);

System.***out***.println("\n--- Linear Search ---");

String productIdToFindLinear = "P05000";

long startTimeLinear = System.*nanoTime*();

Product foundProductLinear = SearchAlgorithms.*linearSearch*(allProducts, productIdToFindLinear);

long endTimeLinear = System.*nanoTime*();

long durationLinear = (endTimeLinear - startTimeLinear) / 1\_000\_000;

if (foundProductLinear != null) {

System.***out***.println("Linear Search: Found product: " + foundProductLinear);

} else {

System.***out***.println("Linear Search: Product " + productIdToFindLinear + " not found.");

}

System.***out***.println("Linear Search Time: " + durationLinear + " ms");

System.***out***.println("\n--- Binary Search ---");

Product[] sortedProducts = Arrays.*copyOf*(allProducts, allProducts.length);

Arrays.*sort*(sortedProducts); // Sorts based on Product's compareTo (productId)

System.***out***.println("Array sorted for Binary Search.");

String productIdToFindBinary = "P05000"; // A product in the middle

long startTimeBinary = System.*nanoTime*();

Product foundProductBinary = SearchAlgorithms.*binarySearch*(sortedProducts, productIdToFindBinary);

long endTimeBinary = System.*nanoTime*();

long durationBinary = (endTimeBinary - startTimeBinary) / 1\_000\_000;

if (foundProductBinary != null) {

System.***out***.println("Binary Search: Found product: " + foundProductBinary);

} else {

System.***out***.println("Binary Search: Product " + productIdToFindBinary + " not found.");

}

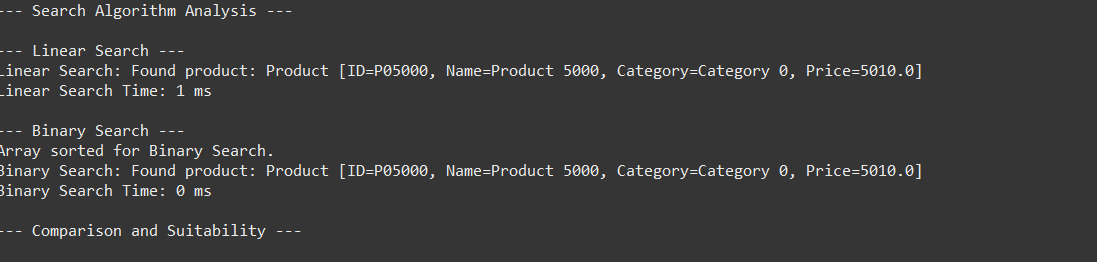
System.***out***.println("Binary Search Time: " + durationBinary + " ms");

System.***out***.println("\n--- Comparison and Suitability ---");

}

}

OUTPUT:



4) Exercise 7: Financial Forecasting

CODE:

public class FinancialForecaster {

public double calculateFutureValueRecursive(double initialValue, double annualGrowthRate, int years) {

if (years == 0) {

return initialValue;

}

double valuePreviousYear = calculateFutureValueRecursive(initialValue, annualGrowthRate, years - 1);

return valuePreviousYear \* (1 + annualGrowthRate);

}

public double calculateFutureValueIterative(double initialValue, double annualGrowthRate, int years) {

double futureValue = initialValue;

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

futureValue \*= (1 + annualGrowthRate);

}

return futureValue;

}

}

public class ForecastingAnalysis {

public static void main(String[] args) {

System.***out***.println("--- Financial Forecasting Analysis ---");

FinancialForecaster forecaster = new FinancialForecaster();

double initialAmount = 1000.0;

double growthRate = 0.05;

int forecastYears = 10;

System.***out***.println("Initial Amount: $" + initialAmount);

System.***out***.println("Annual Growth Rate: " + (growthRate \* 100) + "%");

System.***out***.println("Forecast Years: " + forecastYears);

long startTimeRecursive = System.*nanoTime*();

double futureValueRecursive = forecaster.calculateFutureValueRecursive(initialAmount, growthRate, forecastYears);

long endTimeRecursive = System.*nanoTime*();

long durationRecursive = (endTimeRecursive - startTimeRecursive) / 1\_000\_000;

System.***out***.printf("\nFuture Value (Recursive): $%.2f%n", futureValueRecursive);

System.***out***.println("Recursive Calculation Time: " + durationRecursive + " ms");

long startTimeIterative = System.*nanoTime*();

double futureValueIterative = forecaster.calculateFutureValueIterative(initialAmount, growthRate, forecastYears);

long endTimeIterative = System.*nanoTime*();

long durationIterative = (endTimeIterative - startTimeIterative) / 1\_000\_000;

System.***out***.printf("Future Value (Iterative): $%.2f%n", futureValueIterative);

System.***out***.println("Iterative Calculation Time: " + durationIterative + " ms");

int largeForecastYears = 1000;

System.***out***.println("\n--- Testing with a larger number of years (" + largeForecastYears + ") ---");

startTimeIterative = System.*nanoTime*();

futureValueIterative = forecaster.calculateFutureValueIterative(initialAmount, growthRate, largeForecastYears);

endTimeIterative = System.*nanoTime*();

durationIterative = (endTimeIterative - startTimeIterative) / 1\_000\_000;

System.***out***.printf("Future Value (Iterative, %d years): $%.2f%n", largeForecastYears, futureValueIterative);

System.***out***.println("Iterative Calculation Time: " + durationIterative + " ms");

}

}

OUTPUT:

