MANDATORY EXERCISES WEEK1

**DESIGN PATTERN AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

Scenario:

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

Steps:

1. Create a New Java Project:

o Create a new Java project named SingletonPatternExample.

2. Define a Singleton Class:

o Create a class named Logger that has a private static instance of itself.

o Ensure the constructor of Logger is private.

o Provide a public static method to get the instance of the Logger class.

3. Implement the Singleton Pattern:

o Write code to ensure that the Logger class follows the Singleton design pattern.

4. Test the Singleton Implementation:

o Create a test class to verify that only one instance of Logger is created and used across the application  
  
**CODE:**  
  
public class Logger {

// Step 2: Private static instance

private static Logger instance;

// Step 2: Private constructor to prevent instantiation

private Logger() {

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

}

// Step 2: Public static method to get the instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // Lazy initialization

}

return instance;

}

// Example method to log messages

public void log(String message) {

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

}

}  
  
  
  
public class SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("First message");

Logger logger2 = Logger.getInstance();

logger2.log("Second message");

// Verify both logger1 and logger2 are same instances

if (logger1 == logger2) {

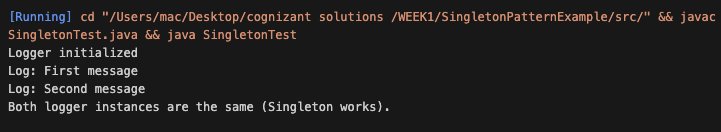
System.out.println("Both logger instances are the same (Singleton works).");

} else {

System.out.println("Different logger instances (Singleton failed).");

}

}

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

Scenario:

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

Steps:

1. Create a New Java Project:

o Create a new Java project named FactoryMethodPatternExample.

2. Define Document Classes:

o Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.

3. Create Concrete Document Classes:

o Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.

4. Implement the Factory Method:

o Create an abstract class DocumentFactory with a method createDocument().

o Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.

5. Test the Factory Method Implementation:

o Create a test class to demonstrate the creation of different document types using the factory method.  
  
**CODE:**public interface Document {

void open();

}  
  
  
  
public class WordDocument implements Document {

public void open() {

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

}

}  
  
  
  
public class PdfDocument implements Document {

public void open() {

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

}

}  
  
  
  
public class ExcelDocument implements Document {

public void open() {

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

}

}  
  
  
  
public abstract class DocumentFactory {

public abstract Document createDocument();

}

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class FactoryMethodTest {

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();

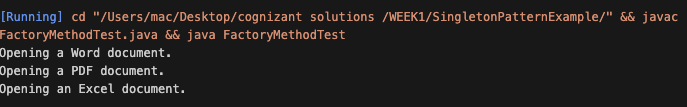
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**OUTPUT:  
**

**DATA STRUCTURES AND ALGORITHM:  
  
Exercise 2: E-commerce Platform Search Function**

Scenario:

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

Steps:

1. Understand Asymptotic Notation:

o Explain Big O notation and how it helps in analyzing algorithms.

o Describe the best, average, and worst-case scenarios for search operations.

2. Setup:

o Create a class Product with attributes for searching, such as productId, productName, and category.

3. Implementation:

o Implement linear search and binary search algorithms.

o Store products in an array for linear search and a sorted array for binary search.

4. Analysis:

o Compare the time complexity of linear and binary search algorithms.

o Discuss which algorithm is more suitable for your platform and why.  
  
  
**CODE:**

public class Product implements Comparable<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 int compareTo(Product other) {

return this.productId - other.productId; // For sorting based on productId

}

@Override

public String toString() {

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

}

}

import java.util.Arrays;

public class ProductSearch {

// Linear Search: O(n)

public static Product linearSearch(Product[] products, int id) {

for (Product product : products) {

if (product.productId == id) {

return product;

}

}

return null;

}

// Binary Search: O(log n)

public static Product binarySearch(Product[] products, int id) {

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (products[mid].productId == id) {

return products[mid];

} else if (products[mid].productId < id) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

import java.util.Arrays;

public class SearchTest {

public static void main(String[] args) {

Product[] products = {

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

new Product(205, "Shoes", "Footwear"),

new Product(150, "Watch", "Accessories"),

new Product(123, "Phone", "Electronics"),

new Product(180, "Book", "Stationery")

};

// Linear search (unsorted array)

Product result1 = ProductSearch.linearSearch(products, 123);

System.out.println("Linear Search Found: " + result1);

// Sort array for binary search

Arrays.sort(products);

// Binary search (sorted array)

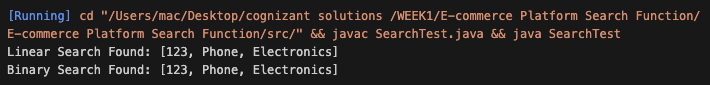
Product result2 = ProductSearch.binarySearch(products, 123);

System.out.println("Binary Search Found: " + result2);

}

}

**OUTPUT:**

****

**Exercise 7: Financial Forecasting**

Scenario:

You are developing a financial forecasting tool that predicts future values based on past data.

Steps:

1. Understand Recursive Algorithms:

o Explain the concept of recursion and how it can simplify certain problems.

2. Setup:

o Create a method to calculate the future value using a recursive approach.

3. Implementation:

o Implement a recursive algorithm to predict future values based on past growth rates.

4. Analysis:

o Discuss the time complexity of your recursive algorithm.

o Explain how to optimize the recursive solution to avoid excessive computation.

**CODE:**public class FinancialForecast {

// Recursive function to calculate future value

public static double predictFutureValue(double currentValue, double growthRate, int years) {

// Base case

if (years == 0) {

return currentValue;

}

// Recursive case: value grows by rate each year

return predictFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

}

public class ForecastTest {

public static void main(String[] args) {

double initialInvestment = 10000.0; // ₹10,000

double annualGrowthRate = 0.08; // 8%

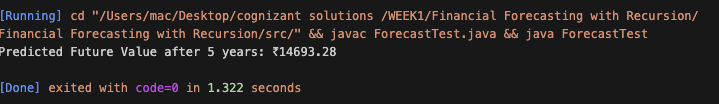
int years = 5;

double futureValue = FinancialForecast.predictFutureValue(initialInvestment, annualGrowthRate, years);

System.out.printf("Predicted Future Value after %d years: ₹%.2f%n", years, futureValue);

}

}

**OUTPUT:  
**