**digital nurture Deep skilling-Week1**

## **Design principles & Patterns**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

Logger.java

package singletonpattern;

public class Logger {

private static Logger *instance*;

private Logger() {

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

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void log(String message) {

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

}

}

TestLogger.java

package singletonpattern;

public class TestLogger {

public static void main(String[] args) {

// **TODO** Auto-generated method stub

Logger logger1 = Logger.*getInstance*();

logger1.log("Starting application...");

Logger logger2 = Logger.*getInstance*();

logger2.log("Continuing application...");

if (logger1 == logger2) {

System.***out***.println("Only one instance of Logger is used.");

} else {

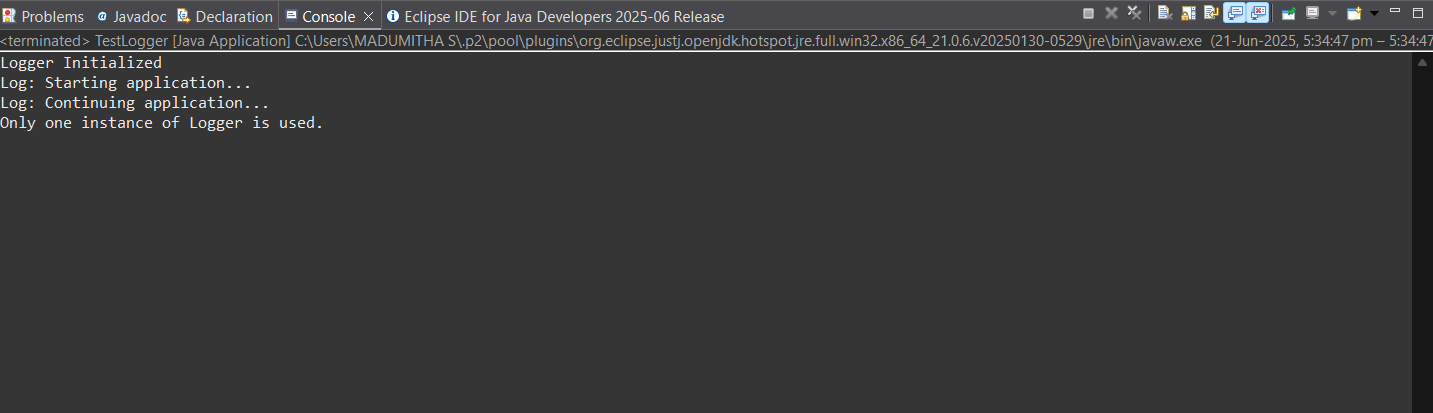
System.***out***.println("Different instances exist (Singleton failed).");

}

}

}

**Output:**



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

**Code:**

Document.java

package factorymethodpattern;

public interface Document {

void open();

}

DocumentFactory.java

package factorymethodpattern;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

ExcelDocument.java

package factorymethodpattern;

public class ExcelDocument implements Document {

*@Override*

public void open() {

// **TODO** Auto-generated method stub

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

}

}

ExcelDocumentFactory.java

package factorymethodpattern;

public class ExcelDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new ExcelDocument();

}

}

PdfDocument.java

package factorymethodpattern;

public class PdfDocument implements Document {

*@Override*

public void open() {

// **TODO** Auto-generated method stub

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

}

PdfDocumentFactory.java

package factorymethodpattern;

public class PdfDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new PdfDocument();

}

}

WordDocument.java

package factorymethodpattern;

public class WordDocument implements Document {

*@Override*

public void open() {

// **TODO** Auto-generated method stub

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

}

}

WordDocumentFactory.java

package factorymethodpattern;

public class WordDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new WordDocument();

}

}

TestFactory.java

package factorymethodpattern;

public class TestFactory {

public static void main(String[] args) {

// **TODO** Auto-generated method stub

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

// Create a PDF document

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

// Create an Excel document

DocumentFactory excelFactory = new ExcelDocumentFactory();

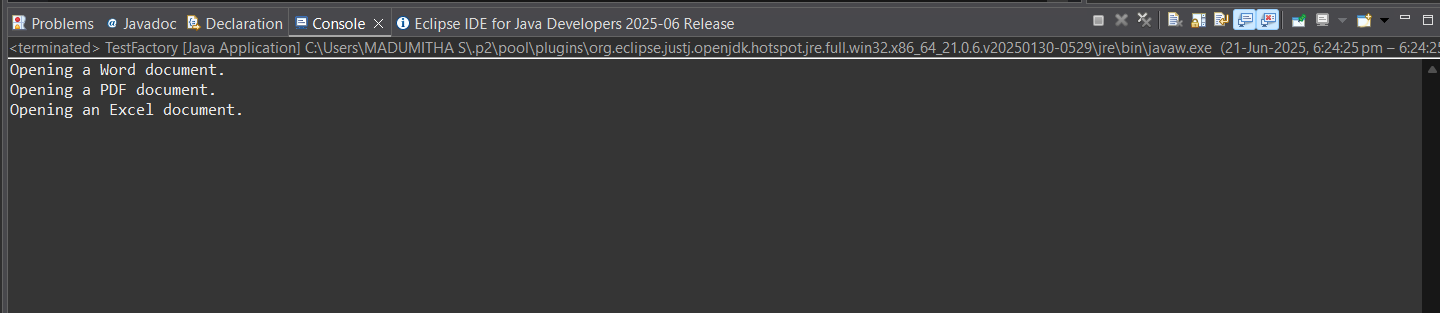
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**



## **Data structures and Algorithms**

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

1. **Understand Asymptotic Notation:**

**Big O notation** describes how an algorithm's **runtime or space requirements grow** relative to the input size (n), especially as n becomes large.

* O (log n)-Binary Search
* O(n)-Linear Search

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best** | O(1) (first element) | O(1) (middle element) |
| **Average** | O(n/2) → O(n) | O(log n) |
| **Worst** | O(n) (last or not found) | O(log n) |

1. **Code:**

**Product.java**

package ecommerce;

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 + "]";

}

}

**SearchDemo.java**

package ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

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

for (Product p : products) {

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

return p;

}

}

return null;

}

// Binary Search by productName (Array must be sorted first)

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

// **TODO** Auto-generated method stub

Product[] products = {

new Product(101, "Shoes", "Footwear"),

new Product(102, "Phone", "Electronics"),

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

new Product(104, "Watch", "Accessories"),

new Product(105, "Shirt", "Apparel")

};

// For Binary Search, sort the array

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

// Test Linear Search

Product result1 = *linearSearch*(products, "Laptop");

System.***out***.println("Linear Search: " + (result1 != null ? result1 : "Not Found"));

// Test Binary Search

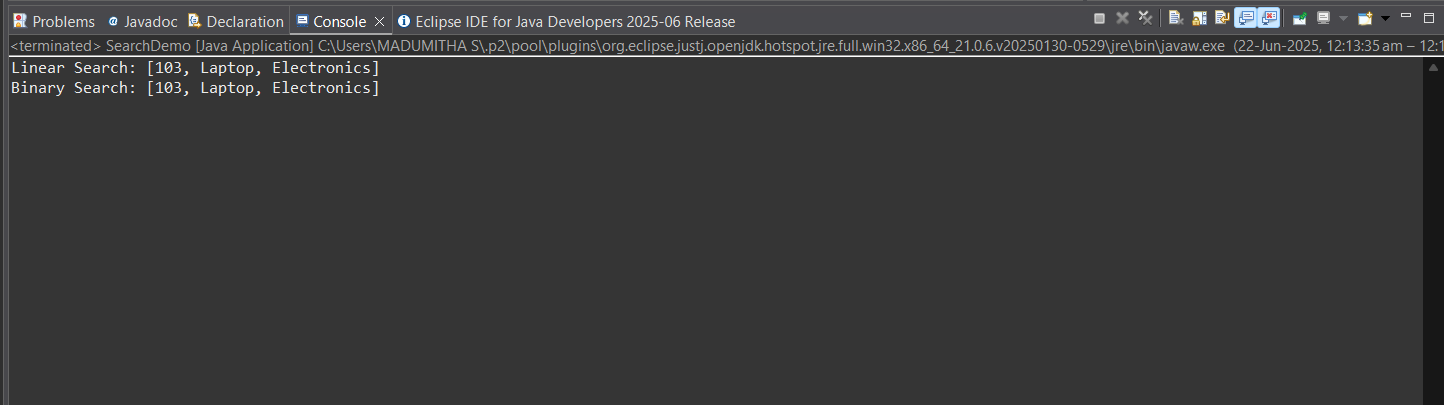
Product result2 = *binarySearch*(products, "Laptop");

System.***out***.println("Binary Search: " + (result2 != null ? result2 : "Not Found"));

}

}

1. **Output:**



1. **Analysis:**

| **Algorithm** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| **Linear Search** | O(n) | Works on unsorted arrays |
| **Binary Search** | O(log n) | Requires **sorted** array |

Linear Search could be used if:

* The array is small
* Data is unsorted

But for an e-commerce platform the product data is huge and also search must be fast and optimized. Hence Binary Search (with sorted data or index structures like BSTs or hash maps) is more suitable.

**Exercise 7: Financial Forecasting**

1. **Understand Recursive Algorithms:**

Recursion is when a method calls itself to solve smaller instances of the same problem.

For Example in real life, to compute future value for year 5, compute year 4 → which needs year 3 → and so on until year 0 (base case).

**Benefits of Recursion:**

* Makes complex problems easier to express
* Naturally fits problems that reduce to smaller subproblems (like forecasting over years)

1. **Formula:**

futureValue(n) = futureValue(n - 1) \* (1 + growthRate)

1. **Code:**

package financialforecast;

public class FinancialForecast {

// Recursive method

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

if (years == 0) {

return initialValue; // base case

}

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

}

public static void main(String[] args) {

double initialValue = 10000; // ₹10,000

double growthRate = 0.10; // 10%

int years = 5;

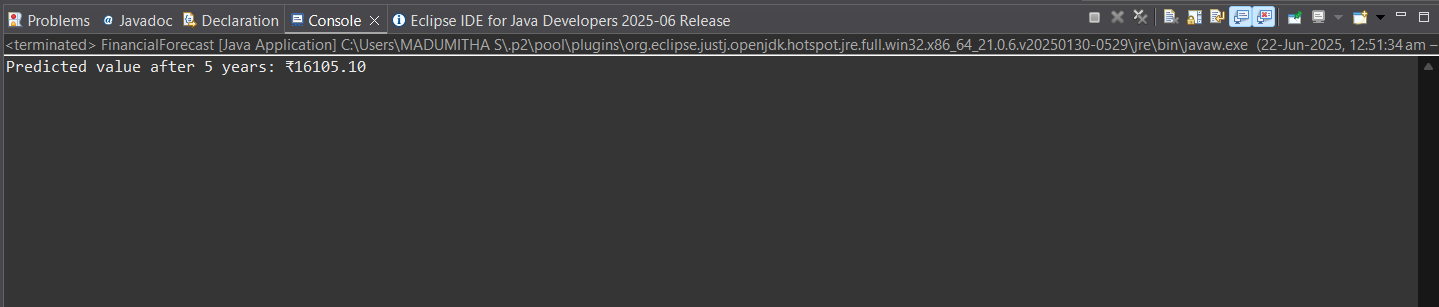
double futureValue = *predictFutureValue*(initialValue, growthRate, years);

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

}

}

**Output:**

****

1. **Analysis of the Recursive Algorithm:**

The function calls itself n times (years times).

Time Complexity = O(n)

For large n, it might cause a stack overflow or be inefficient.

1. **Optimization Techniques:**

* **Memoization (caching results)**

Store values already calculated to avoid repeated work.

* **Convert to Iterative**

Simple loop is faster and avoids recursion limits.

**Optimized Code:**

package financialforecast;

public class iterative {

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

double value = initialValue;

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

value \*= (1 + growthRate);

}

return value;

}

public static void main(String[] args) {

double initialValue = 10000; // ₹10,000

double growthRate = 0.10; // 10%

int years = 5;

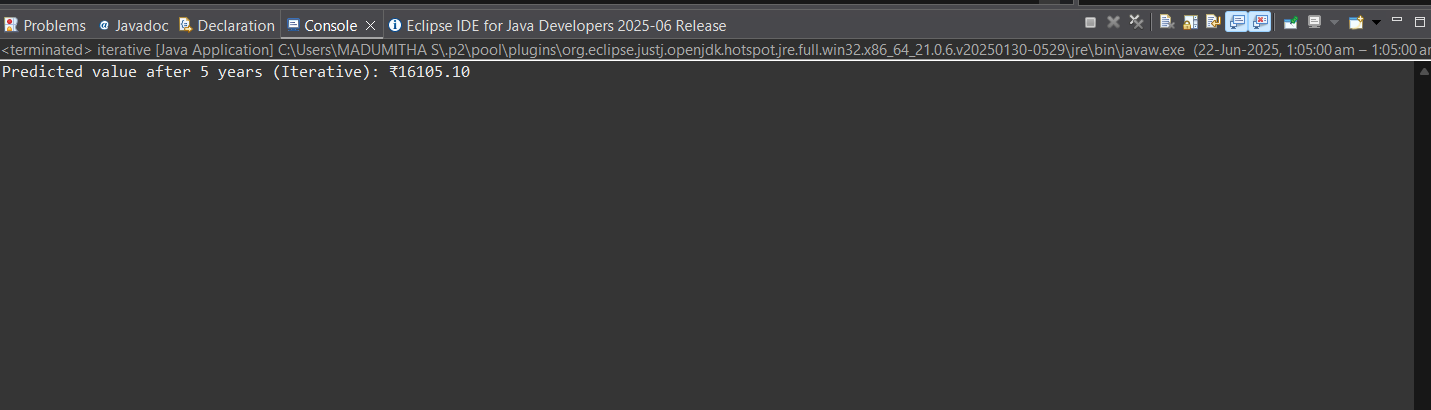
double futureValue = *predictFutureValue*(initialValue, growthRate, years);

System.***out***.printf("Predicted value after %d years (Iterative): ₹%.2f\n", years, futureValue);

}

}

**Output:**



1. **Time complexity**

| **Feature** | **Recursive Version** | **Iterative Version** |
| --- | --- | --- |
| Time Complexity | O(n) | O(n) |
| Space Complexity | O(n) (due to call stack) | O (1) |
| Risk of Stack Overflow | Yes (large n) | No |