**WEEK – 01**

**Module 01 – Design Patterns 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.

**Solution:**

**Logger.java:**  
package singleton;

public class Logger {

private static Logger *instance*;

private Logger() {

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

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void log(String message) {

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

}

}

**Main.java:**

package singleton;

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

Logger logger2 = Logger.*getInstance*();

logger1.log("This is the first log message.");

logger2.log("This is the second log message.");

if (logger1 == logger2) {

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

} else {

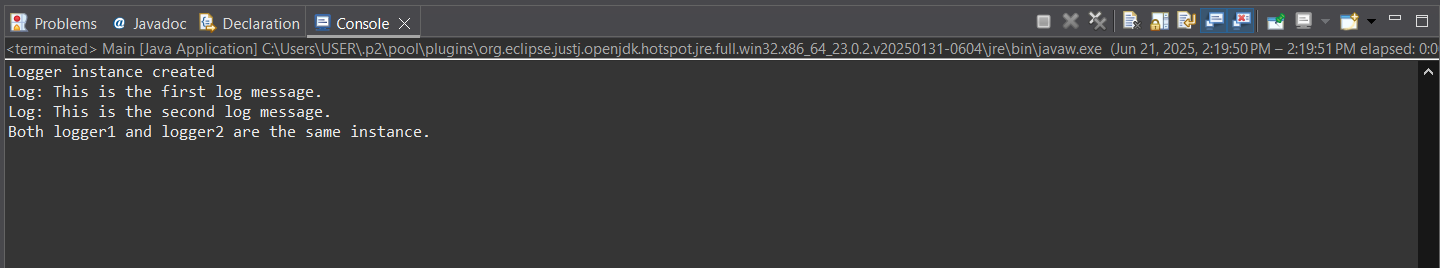
System.*out*.println("logger1 and logger2 are different instances.");

}

}

}

**Output:**



**Explanation:**

* The above code uses the Singleton Design Pattern which ensures that only one instance of the Logger class exists throughout the application.
* The class has a private static variable to hold the single instance and a private constructor to prevent direct instantiation.
* A public static method getInstance() provides access to the instance, creating it if it doesn't already exist. This ensures consistent logging across the application.
* A test class confirms that multiple calls to getInstance() return the same object, verifying the Singleton behavior and promoting controlled access to the shared logging resource.

**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.

**Solution:**

**Document.java:**

package FactoryMethodPatternExample;

public interface Document {

void open();

String getFormat();

}

**WordFile.java:**

package FactoryMethodPatternExample;

public class WordFile implements Document{

public void open() {

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

}

public String getFormat() {

return "Format: .docx";

}

}

**PdfFile.java:**

package FactoryMethodPatternExample;

public class PdfFile implements Document {

public void open() {

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

}

public String getFormat() {

return "Format: .pdf";

}

}

**ExcelFile.java:**

package FactoryMethodPatternExample;

public class ExcelFile implements Document {

public void open() {

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

}

public String getFormat() {

return "Format: .xlsx";

}

}

**DocumentCreator.java:**

package FactoryMethodPatternExample;

public abstract class DocumentCreator {

public abstract Document createDocument();

public void previewDocument() {

Document document = createDocument();

document.open();

System.***out***.println(document.getFormat());

}

}

**WordCreator.java:**

package FactoryMethodPatternExample;

public class WordCreator extends DocumentCreator {

public Document createDocument() {

return new WordFile();

}

}

**PdfCreator.java:**

package FactoryMethodPatternExample;

public class PdfCreator extends DocumentCreator {

public Document createDocument() {

return new PdfFile();

}

}

**ExcelCreator.java:**

package FactoryMethodPatternExample;

public class ExcelCreator extends DocumentCreator {

public Document createDocument() {

return new ExcelFile();

}

}

**Main.java:**

package FactoryMethodPatternExample;

public class Main {

public static void main(String[] args) {

// TODO Auto-generated method stub

DocumentCreator wordCreator = new WordCreator();

DocumentCreator pdfCreator = new PdfCreator();

DocumentCreator excelCreator = new ExcelCreator();

System.*out*.println("Previewing Word File:");

wordCreator.previewDocument();

System.*out*.println("\nPreviewing PDF File:");

pdfCreator.previewDocument();

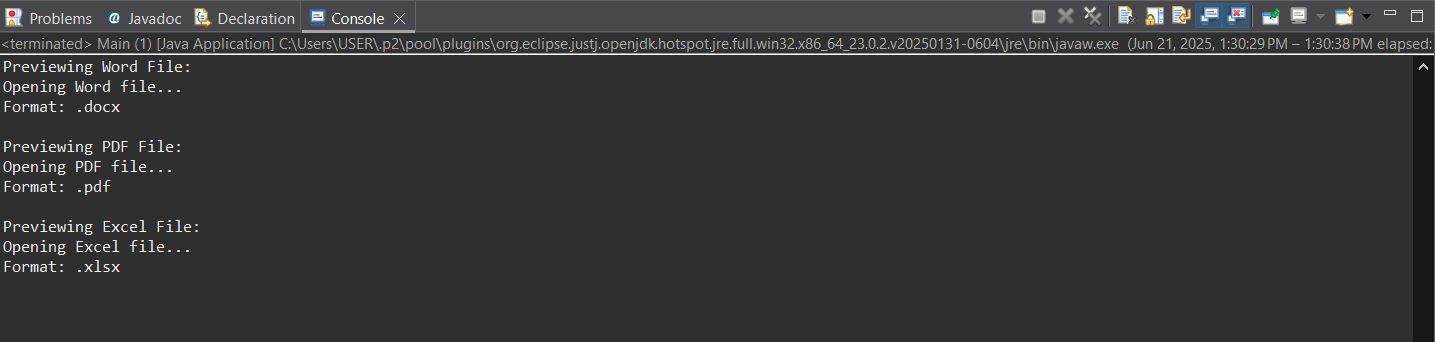
System.*out*.println("\nPreviewing Excel File:");

excelCreator.previewDocument();

}

}

**Output:**



**Explanation:**

* This implementation uses the Factory Method Pattern to create different types of documents like Word, PDF, and Excel.
* An interface Document defines common methods (open() and getFormat()), which are implemented by concrete classes.
* The abstract class DocumentCreator defines the factory method createDocument() and a shared previewDocument() method.
* Concrete factories like WordCreator, PdfCreator, and ExcelCreator create specific document types.
* The Main class demonstrates how different document types can be created and used without knowing their exact classes, ensures the flexibility and easy scalability.

**Module 02 – Algorithms and Data Structures**

**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.

**Solution:**

**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;

}

}

**SearchUtils.java:**

package ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtils {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

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

if (comparison < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

}

**Main.java:**

package ecommerce;

public class Main {

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, "Watch", "Accessories")

};

Product result1 = SearchUtils.linearSearch(products, "Phone");

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

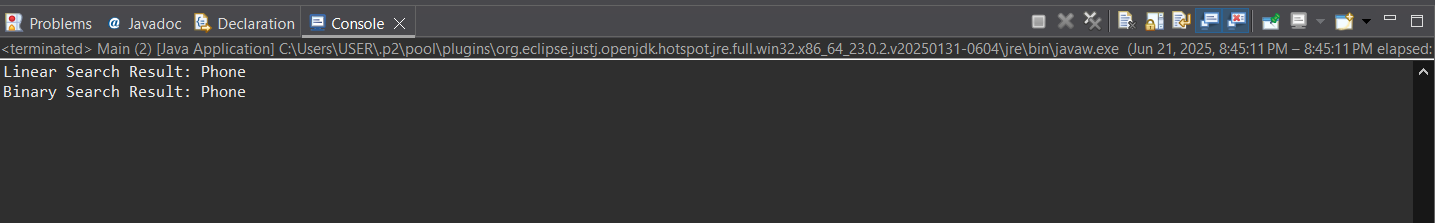
Product result2 = SearchUtils.binarySearch(products, "Phone");

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

}

}

**Output:**

****

**Understand Asymptotic Notation**

* Big O notation is used to describe how an algorithm’s time or space complexity grows as the input size increases.
* It helps developers evaluate the efficiency of different algorithms, especially when working with large data sets.
* Linear search has O(1) best case and O(n) worst case.
* Binary search has O(1) best case and O(log n) worst case.

**Analysis**

* Linear search has O(n) time complexity, slower for large datasets.
* Binary search has O(log n) time complexity, much faster but needs sorted data.
* For an e-commerce platform handling thousands of products, binary search is more suitable as it provides better performance with O(log n) complexity.
* Linear search can still be useful for small or unsorted collections where sorting is unnecessary.

**Code Explanation:**

* The Product class defines the structure of a product with productId, productName, and category.
* The SearchUtils class contains two methods:
* “linearSearch()” checks each product one by one to find a match by name.
* “binarySearch()” first sorts the products by name, then uses the divide-and-conquer method to search faster.
* The Main class creates an array of products and tests both search methods by trying to find a product named "Phone".
* The output confirms whether each search method successfully finds the product or not.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Solution:**

**Forecast.java:**

package forecast;

public class Forecast {

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

if (years == 0) {

return initialValue;

}

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

}

}

**Main.java:**

package forecast;

public class Main {

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 0.05;

int years = 5;

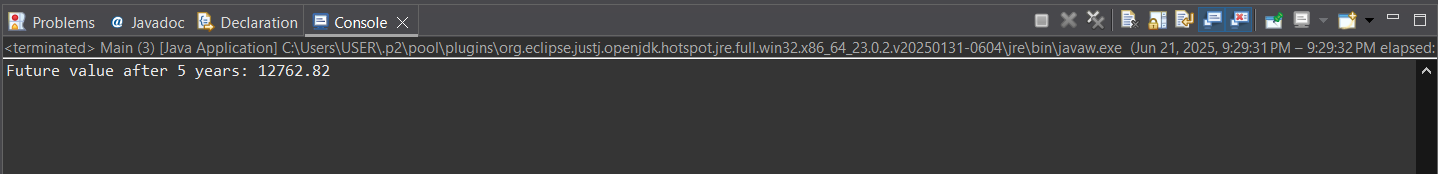
double futureValue = Forecast.predictFutureValue(initialValue, growthRate, years);

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

}

}

**Output:**



**Understand Recursive Algorithms:**

* Recursion is a programming technique where a function calls itself to solve a smaller instance of the same problem.
* It simplifies problems by breaking them into base cases and smaller subproblems, making the code easier to understand and implement in scenarios like mathematical modeling or repeated calculations (e.g., forecasting future values based on past growth).

**Analysis:**

* The time complexity of the recursive algorithm is O(n), where n is the number of years.
* This is because the method is called once for each year, with no repeated subproblems.

The Recursive approach can be optimized in two main ways:

* Use memoization to store previously computed results and avoid redundant calculations (useful in more complex variations).
* Convert the recursive method to an iterative approach, which uses a loop and avoids the overhead of recursive calls, especially for large input sizes where stack overflow could occur.

**Code Explanation:**

* The method predictFutureValue() calculates the future value of money using recursion based on initial amount, growth rate, and number of years.
* The base case checks if years == 0, and returns the initial value because no growth has occurred yet.
* A HashMap called cache is used for memoization to store previously computed results and avoid repeating the same calculations.
* If the value is not in the cache, the method recursively calculates the value for years - 1, multiplies it by (1 + growthRate), and stores the result in the cache.
* This approach improves performance by avoiding redundant recursive calls and is ideal for optimizing recursive financial forecasts.