**DEEP NURTURE 4.0----WEEK-1**

**Design Patterns and principles:**

**Excercise1:**

**Implementing the Singleton Pattern**

**Program:**

**package** SingletonLogger;

**public** **class** SingletonLogger {

**private** **static** SingletonLogger *instance*;

**private** SingletonLogger() {

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

}

**public** **static** SingletonLogger getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** SingletonLogger();

}

**return** *instance*;

}

**public** **void** log(String message) {

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

}

**public** **static** **void** main(String[] args) {

SingletonLogger logger1 = SingletonLogger.*getInstance*();

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

SingletonLogger logger2 = SingletonLogger.*getInstance*();

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

**if** (logger1 == logger2) {

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

} **else** {

System.***out***.println("Different instances exist - Singleton not working.");

}

}

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Excercise2:**

**Implementing the Factory Method Pattern**

**Program:**

package Pattern;

public class FactoryMethodPattern {

interface Document {

void open();

}

static class WordDocument implements Document {

public void open() {

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

}

}

static class PdfDocument implements Document {

public void open() {

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

}

}

static class ExcelDocument implements Document {

public void open() {

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

}

}

static abstract class DocumentFactory {

public abstract Document createDocument();

}

static class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

static class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

static class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excel = excelFactory.createDocument();

excel.open();

}

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Algorithm Data Structures:**

**Excercise1:**

**E-commerce Platform Search Function**

**Big O Notation:**

* Big O notation describes the worst-case time complexity of an algorithm, showing how the runtime scales with the size of the input.

**Search Scenarios:**

* **Best Case:** Found at first attempt (start of list).
* **Average Case:** Found after scanning half the list**.**
* **Worst Case:** Element not found or found at the end**.**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**Program:**

package Ecommerce;

import java.util.\*;

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.productName.compareTo(other.productName);

}

@Override

public String toString() {

return "Product{" +

"Id=" + productId +

", Name='" + productName + '\'' +

", category='" + category + '\'' +

'}';

}

}

public class SearchAlgorithms {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

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

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = new Product[] {

new Product(1, "Apple Watch", "Electronics"),

new Product(2, "Samsung TV", "Electronics"),

new Product(3, "Nike Shoes", "Fashion"),

new Product(4, "Sony Headphones", "Electronics"),

new Product(5, "Adidas Jacket", "Fashion")

};

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

Arrays.*sort*(sortedProducts);

Product linearSearchResult = *linearSearch*(products, "Sony Headphones");

System.*out*.println("Linear Search Result: " + linearSearchResult);

Product binarySearchResult = *binarySearch*(sortedProducts, "Sony Headphones");

System.*out*.println("Binary Search Result: " + binarySearchResult);

}

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Excercise2:**

**Financial Forecasting**:

* Recursion is a technique where a method calls itself to solve a smaller instance of the same problem.

**Example:**  
To compute compound future value:

* FutureValue(n)=FutureValue(n−1)×(1+r)
* Base case:FutureValue(0)=initialValue

Benefits:

* Elegant and simplifies code.
* Especially useful in problems that follow a repeated pattern (e.g., financial growth over years).

**program:**

package Financial;

public class FinancialForecaster {

public static double calculateFutureValue(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue;

} else {

return *calculateFutureValue*(presentValue, growthRate, years - 1) \* (1 + growthRate);

}

}

public static double calculateFutureValueMemoized(double presentValue, double growthRate, int years, double[] memo) {

if (years == 0) {

return presentValue;

} else if (memo[years] != 0) {

return memo[years];

} else {

double futureValue = *calculateFutureValueMemoized*(presentValue, growthRate, years - 1, memo) \* (1 + growthRate);

memo[years] = futureValue;

return futureValue;

}

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05;

int years = 5;

double futureValueRecursive = *calculateFutureValue*(presentValue, growthRate, years);

System.*out*.println("Future Value (Recursive): " + futureValueRecursive);

double[] memo = new double[years + 1];

double futureValueMemoized = *calculateFutureValueMemoized*(presentValue, growthRate, years, memo);

System.*out*.println("Future Value (Memoized): " + futureValueMemoized);

}

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**