MANDATORY HANDS-ON EXERCISES (WEEK 1)

**Exercise 1: Implementing the Singleton Pattern**

**Solution:**

**1.Definfing a singleton class**

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: " + message);  
 }  
}

**2. Test the Singleton Implementation**

public class LoggerTest {  
 public static void main(String[] args) {  
 Logger logger1 = Logger.getInstance();  
 logger1.log("System startup complete.");  
  
 Logger logger2 = Logger.getInstance();  
 logger2.log("User login successful.");  
  
 if (logger1 == logger2) {  
 System.out.println("Logger is a singleton: Only one instance exists.");  
 } else {  
 System.out.println("Singleton pattern failed: multiple instances found.");  
 }  
 }  
}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

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

**Solution:**

**1.Interface for documents**

**Document.java**

public interface Document {

void open();

}

**WordDocument.java**

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

**PdfDocument.java**

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

**2. Implementation of Factory Method**

**DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**3.Testing the factory method implementation**

public class DocumentFactoryTest {  
 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:**A black background with lights

AI-generated content may be incorrect.

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

**Solution:**

**1.BIG O Notation:**

Big O notation describes the upper bound of an algorithm’s running time or space requirement as the input size grows. It helps us compare algorithms independent of hardware or specific inputs.

**Examples:**

* O(1) → Constant time: time doesn’t depend on input size (e.g., accessing an array by index)
* O(n) → Linear time: time grows linearly with input size (e.g., linear search)
* O(log n) → Logarithmic time: time grows logarithmically (e.g., binary search)
* O(n²) → Quadratic time: time grows quadratically (e.g., bubble sort)

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

**2. Setup**

**Product class**

public class Product {

private int productId;

private String productName;

private String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

}

**3.Implementation**

* **Linear Search**

public class LinearSearch {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

}

* **Binary Search**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

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

int cmp = products[mid].getProductName().compareToIgnoreCase(targetName);

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void sortProductsByName(Product[] products) {

Arrays.sort(products, Comparator.comparing(Product::getProductName, String.CASE\_INSENSITIVE\_ORDER));

}

}

**Example:**

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Fashion"),

new Product(3, "Mobile", "Electronics"),

new Product(4, "Bag", "Fashion")

};

Product found1 = LinearSearch.linearSearch(products, "Mobile");

System.out.println("Linear Search Result: " + (found1 != null ? found1.getProductName() : "Not Found"));

BinarySearch.sortProductsByName(products);

Product found2 = BinarySearch.binarySearch(products, "Mobile");

System.out.println("Binary Search Result: " + (found2 != null ? found2.getProductName() : "Not Found"));

}

}

**Output:**

**A blurry image of a computer screen

AI-generated content may be incorrect.**

**Exercise 7: Financial Forecasting**

**Solution:**

**Recursion:**

* Recursion is when a method calls itself to solve a smaller instance of the same problem.
* It’s like solving a big problem by breaking it down repeatedly into smaller, similar subproblems.
* Classic examples: factorial, Fibonacci sequence, tree traversal.

**2.Setup(Formula)**

Future Value = Present Value × (1 + growthRate) ^ years

**3.Implementation**

public class FinancialForecast {

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

if (years == 0) {

return presentValue;

} else {

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

}

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05;

int years = 5;

double futureValue = predictFutureValue(presentValue, growthRate, years);

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

}

}

**Output:**

**A computer screen shot of a computer screen

AI-generated content may be incorrect.**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |