**Week 1**

**Mandatory Hands-on**

**Design Patterns And Principles**

Exercise 1

**Code**

public class Logger {  
 private static Logger *instance* = new Logger();  
 int x=10;  
 private Logger(){  
  
 }  
  
 public static Logger getInstance(){  
 return *instance*;  
 }  
}  
  
class Testing{  
 public static void main(String[]Args){  
 Logger num1=Logger.*getInstance*();  
 Logger num2=Logger.*getInstance*();  
  
 num1.x=num1.x-5;  
 System.*out*.println("Here is the first instance called "+num1.x);  
 System.*out*.println("Here is the second instance called "+num2.x);  
 }  
}

**Output**

A black screen with a black border

AI-generated content may be incorrect.

This proves that only one instance of the Logger class was created with the help of a private constructor, thus following the Singleton Design Pattern. Only num1 was changed, but since num1 and num2 refer to the same instance, both got changed.

Exercise 2

Code

**DocumentFactory.java**

abstract class DocumentFactory {  
 abstract String createDocument();  
}  
  
class WordPrint extends DocumentFactory{  
 String createDocument(){  
 String type ="word";  
 return type;  
 }  
}  
  
class ExcelPrint extends DocumentFactory{  
 String createDocument(){  
 String type ="excel";  
 return type;  
 }  
}  
  
class PdfPrint extends DocumentFactory{  
 String createDocument(){  
 String type ="pdf";  
 return type;  
 }  
}

**WordDocument.java**

public interface WordDocument {  
  
}  
  
class Word implements WordDocument{  
 void open(){  
 //Opening document  
 }  
  
}

**ExcelDocument.java**

public interface ExcelDocument{  
   
}  
class Excel implements ExcelDocument{  
 void open(){  
 //Opening document  
 }  
}

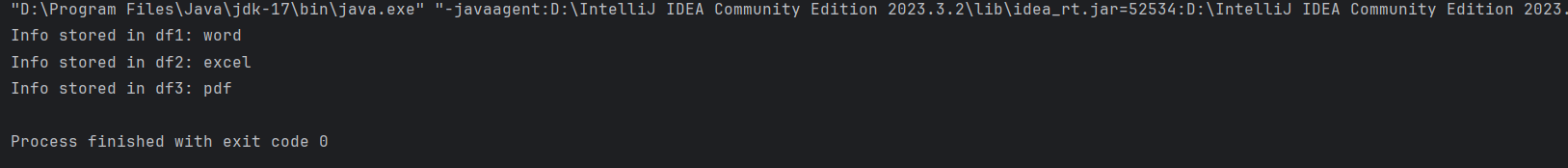
**PdfDocument.java**

public interface PdfDocument{  
   
}  
class Pdf implements PdfDocument{  
 void open(){  
 //Opening document  
 }  
}

**Testing.java**

public class Testing {  
 public static void main(String[]Args){  
  
 DocumentFactory df1= new WordPrint();  
 DocumentFactory df2= new ExcelPrint();  
 DocumentFactory df3= new PdfPrint();  
  
 System.*out*.println("Info stored in df1: "+df1.createDocument());  
 System.*out*.println("Info stored in df2: "+df2.createDocument());  
 System.*out*.println("Info stored in df3: "+df3.createDocument());  
 }  
}

**Output**



**Algorithms\_DataStructures**

Exercise 2

Big O Notation

The worst-case time or space complexity of an algorithm as a function of input size n is expressed using Big O notation. It aids in assessing how well an algorithm works and scales with increasing input. It disregards constant and less important operations in favour of the dominant term. This enables us to evaluate algorithms based on their efficiency.

Best, average, and worst-case scenarios for search operations

Best Case: The element is located at the first position (for example, O(1) in linear search).

Average Case: The element is in the middle; the linear search takes O(n/2) time on average, which simplifies to O(n).

In the worst scenario, the element is either missing or at the end of the list; therefore, it takes O(n) for linear search and O(log n) for binary search in a sorted array because the entire list must be scanned.

Code

**Product.java**

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;  
 }  
  
 @Override  
 public String toString() {  
 return "Product[ID=" + productId + ", Name=" + productName + ", Category=" + category + "]";  
 }  
}

**Searching.Java**

import java.util.Arrays;  
import java.util.Comparator;  
  
public class Searching {  
  
 public static Product linearSearch(Product[] products, int targetId) {  
 for (Product p : products) {  
 if (p.getProductId() == targetId) {  
 return p;  
 }  
 }  
 return null;  
 }  
  
 public static Product binarySearch(Product[] products, int targetId) {  
 int left = 0;  
 int right = products.length - 1;  
  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 int midId = products[mid].getProductId();  
  
 if (midId == targetId) {  
 return products[mid];  
 } else if (midId < targetId) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
 return null;  
 }  
  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(103, "Mouse", "Electronics"),  
 new Product(101, "Keyboard", "Electronics"),  
 new Product(102, "Monitor", "Electronics"),  
 new Product(104, "Printer", "Electronics")  
 };  
  
 System.*out*.println("Linear Search Result:");  
 Product foundLinear = *linearSearch*(products, 102);  
 System.*out*.println(foundLinear != null ? foundLinear : "Product not found");  
  
 Arrays.sort(products, Comparator.comparingInt(Product::getProductId));  
  
 System.*out*.println("Binary Search Result:");  
 Product foundBinary = *binarySearch*(products, 102);  
 System.*out*.println(foundBinary != null ? foundBinary : "Product not found");  
 }  
}

**Output**

**A screen shot of a computer code

AI-generated content may be incorrect.**

In this implementation, linear search and binary search are used to locate a product by its productId from an array of Product objects. The time complexity of linear search is O(n) in the worst case, as it may need to scan each element sequentially. In contrast, binary search offers a better time complexity of O(log n) but requires the array to be sorted beforehand. For this platform, if the product list is large and primarily used for search operations without frequent insertions, binary search is more suitable due to its faster performance. However, linear search remains more flexible when working with small or unsorted datasets.

Exercise 7

Recursion is a programming technique where a method calls itself to solve smaller instances of a problem. It simplifies complex tasks like tree traversal, factorials, or computations based on repeated patterns, by breaking them into smaller subproblems that follow the same logic.

Code

public class FutureValueCalculator {

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.1; // 10%

int years = 3;

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

System.out.println("Future Value after " + years + " years: " + futureValue);

}

}

Output

A screen shot of a computer

AI-generated content may be incorrect.

The time complexity of the recursive future value algorithm is O(n), where n is the number of years. This is because the function makes one recursive call for each year, reducing the years value by 1 at each step until it reaches 0.

Java does not optimize tail-recursive calls by default, so iteration is usually better.