**Design Pattern and Principals**

**Exercise 1: Implementing the Singleton Pattern**

Code:

class Logger{

    private static Logger instance;

    private Logger(){

        System.out.println("Logger initialized.");

    }

    public static synchronized Logger getInstance(){

        if(instance==null){

            instance = new Logger();

        }

        return instance;

    }

    public void log(String msg){

        System.out.println("Log: "+msg);

    }

}

public class SingletonPatternExample

{

    public static void main(String[] args){

        Logger logger1= Logger.getInstance();

        Logger logger2= Logger.getInstance();

        logger1.log("App started");

        logger1.log("App running");

        if(logger1==logger2){

            System.out.println("Both logger instance are same(Singleton).");

        }else{

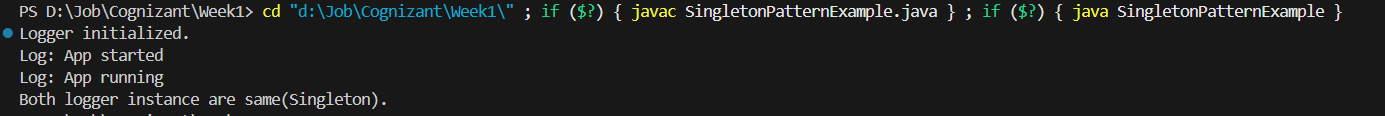
            System.out.println("Different logger instance (Not Singleton).");

        }

    }

}

Output



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

Code

interface Document{

    void open();

}

class WordDocument implements Document{

    public void open(){

        System.out.println("This is a Word Document");

    }

}

class PdfDocument implements Document{

    public void open(){

        System.out.println("This is a PDF Document");

    }

}

class ExcelDocument implements Document{

    public void open(){

        System.out.println("This is a Excel Document");

    }

}

abstract class DocumentFactory{

    public abstract Document createDocument();

}

class WordDocFactory extends DocumentFactory{

    public Document createDocument(){

        return new WordDocument();

    }

}

class PdfDocFactory extends DocumentFactory{

    public Document createDocument(){

        return new PdfDocument();

    }

}

class ExcelDocFactory extends DocumentFactory{

    public Document createDocument(){

        return new ExcelDocument();

    }

}

public class FactoryMethodPatternExample {

    public static void main(String[] args){

        //Word

        DocumentFactory wordFac = new WordDocFactory();

        Document wordFile = wordFac.createDocument();

        wordFile.open();

        //PDF

        DocumentFactory PdfFac = new PdfDocFactory();

        Document PdfFile = PdfFac.createDocument();

        PdfFile.open();

        //Excel

        DocumentFactory ExcelFac = new ExcelDocFactory();

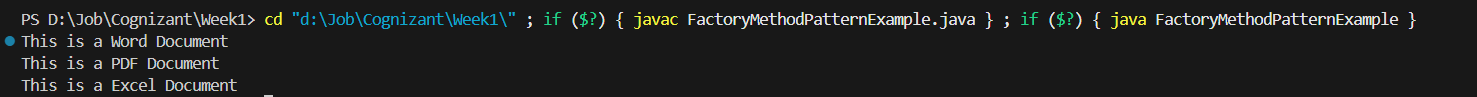
        Document ExcelFile = ExcelFac.createDocument();

        ExcelFile.open();

    }

}

Output



**Data Structures and Algorithm**

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

**Step1: Understand Asymptotic Notation**

1. Explain Big O notation and how it helps in analyzing algorithms.

Ans: Big O notation describes the time complexity (or space complexity) of an algorithm in terms of input size n. It provides an upper bound on running time — useful for understanding scalability.

Q> Describe the best, average, and worst-case scenarios for search operations

Ans:

|  |  |  |
| --- | --- | --- |
| **Case** | **Linear Search** | **Binary Search** |
| Best | O(1) if target is found at starting position of the array | O(1) if middle element of the sorted array is target |
| Average | O(n) | O(log n) |
| Worst | O(n) | O(log n) |

Code:

import java.util.\*;

class Product{

    private int productId;

    private String productName;

    private String category;

    public Product(int id,String pName,String cat){

        this.productId=id;

        this.productName=pName;

        this.category=cat;

    }

    int getProdId(){

        return this.productId;

    }

    void showDetails(){

        System.out.println("ProductId: "+this.productId+" ProductName: "+this.productName+" Category: "+this.category);

    }

}

class pSearch{

    public void linearSearch(Product[] prods,int targetId){

        for(Product i : prods){

            if(i.getProdId()==targetId){

                System.out.println("Product Found");

                i.showDetails();

                return;

            }

        }

        System.out.println("Product Not Found");

    }

    public void binarySearch(Product[] prods,int targetId){

        int left = 0;

        int right = prods.length-1;

        int mid;

        while(left<=right){

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

            if(prods[mid].getProdId()==targetId){

                System.out.println("Product Found");

                prods[mid].showDetails();

                return;

            }else if(targetId>prods[mid].getProdId()){

                left=mid+1;

            }else{

                right=mid-1;

            }

        }

        System.out.println("Product Not Found");

    }

    void sortByProdId(Product[] prods){

        for(int i=0;i<prods.length-1;i++){

            int f=0;

            for(int j=0;j<prods.length-i-1;j++){

                if(prods[j].getProdId()>prods[j+1].getProdId()){

                    Product temp=prods[j];

                    prods[j]=prods[j+1];

                    prods[j+1]=temp;

                    f=1;

                }

            }

            if(f==0){

                return;

            }

        }

    }

}

public class ProductSearch {

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        Product prod[];

        pSearch search = new pSearch();

        int n,i;

        System.out.println("Enter number of Items: ");

        n=sc.nextInt();

        prod = new Product[n];

        for(i=0;i<n;i++){

            System.out.println("Enter productId, ProductName, Category: ");

            int id=sc.nextInt();

            String pName=sc.next();

            String cat=sc.next();

            prod[i] = new Product(id,pName,cat);

        }

        for(i=0;i<n;i++){

            prod[i].showDetails();;

        }

        System.out.println("Enter the target Value: ");

        int target = sc.nextInt();

        //Linear Search

        System.out.println("Search by Linear Search: ");

        search.linearSearch(prod, target);

        //Binary Search

        System.out.println("Search by Binary Search: ");

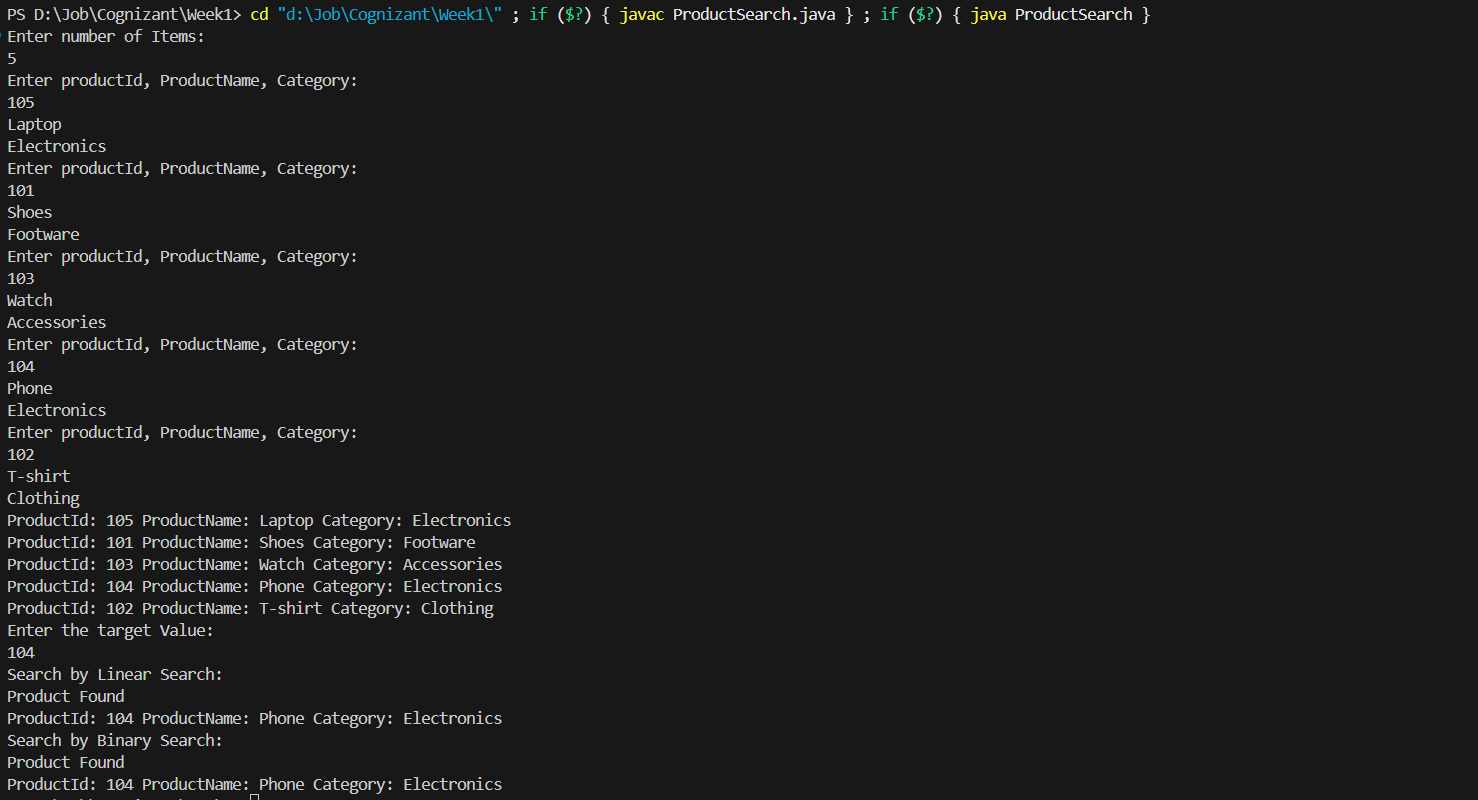
        search.sortByProdId(prod);

        search.binarySearch(prod, target);

        sc.close();

    }

}

Output

Analysis:

Q> Compare the time complexity of linear and binary search algorithms

Ans: The Time Complexity for Linear Search and Binary Search in our problem

Is O(n) and O(log n) respectively.

Q> Discuss which algorithm is more suitable for your platform and why.

Ans: As the algorithm is designed for a E-commerce website so our Product list will be large. So we know in large dataset for searching we should use binary search as its the time complexity is less than linear search.

**Exercise 7: Financial Forecasting**

Step 1:

Q> Explain the concept of recursion and how it can simplify certain problems.

Ans: Recursion is a programming technique where a function calls itself in order to solve a problem by breaking it down into smaller subproblems.

Every recursive function has:

1. Base Case – the condition where recursion stops.
2. Recursive Case – the part where the function calls itself with a smaller input.

Recursion simplifies complex problems that have:

* Repetitive structures (e.g., trees, graphs)
* Divide and conquer patterns (e.g., merge sort, binary search)
* Problems that can be reduced to smaller versions of the same problem

Code:

import java.util.\*;

public class financialForecast {

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

        if (years == 0) {

            return principal;

        }

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

    }

    public static void main(String[] args) {

        double initialInvestment;

        double annualGrowthRate;

        int years;

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the Initial Investment, AnnualGrowth, Years: ");

        initialInvestment=sc.nextDouble();

        annualGrowthRate=sc.nextDouble();

        years=sc.nextInt();

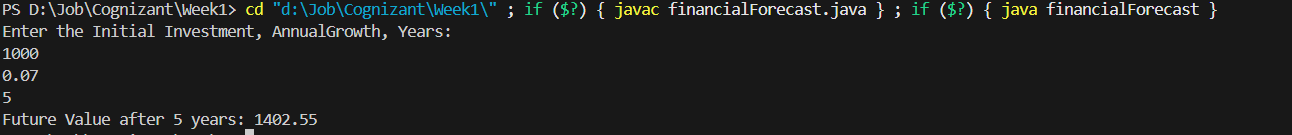
        double futureValue = calculateFutureValue(initialInvestment, annualGrowthRate, years);

        System.out.println("Future Value after "+ years+ " years: "+String.format("%.2f", futureValue));

    }

}

Output



Analysis

Q> Discuss the time complexity of your recursive algorithm.

Ans: Recursive time complexity: O(n) . Because it makes n recursive calls for n years.

Q> Explain how to optimize the recursive solution to avoid excessive computation.

Ans: We can optimize the recursive solution to avoid excessive computation by using a Iterative Approach(i.e using a for or while loop)**.** This would also reduce the space complexity.