**Deep Skilling Program**

**Hands-On Solutions[Week-1]**

**Week-1 :** 1. Design Patterns and Principles

2. Data structures and Algorithms

**Submitted by:** Bhavya Sree Pappanaboyina

**Email Id :** 229x1a2823@gmail.com

**GitHubRepo :**  https://github.com/Bhavyasree1606/deep-skilling-

solutions-6421608

**Design Patterns and Principle**

**Exercise 1:** Implementing the Singleton Pattern

**1. Project Title:** Smart Logger – A Singleton-Based Application Logging Utility

**2. Objective:**

The primary aim of this project is to implement the Singleton Design Pattern in a real-world scenario where consistent, centralized logging is required across an entire application. The utility should:

* Maintain a single instance of the Logger class throughout the lifecycle of the application.
* Support logging at different severity levels (INFO, DEBUG, WARN, ERROR).
* Provide output to both console and log file, ensuring durability of log data.
* Demonstrate a solid understanding of OOP principles like encapsulation, lazy initialization, and thread safety.

**3. What is Singleton Pattern (Theory & Purpose):**

The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to it.

Key Characteristics:

* Private constructor to restrict external instantiation.
* Static method to return the same object every time.
* Static variable to hold the one and only instance.
* Often used in resource-heavy or global control classes, like database managers, config readers, or loggers.

**4. Project Overview:**

This Java-based application contains three components:

1. **Logger Class**

* Implements the Singleton pattern.
* Uses an inner enum for log levels.
* Capable of writing logs to both console and a log file.
* Ensures thread safety, lazy instantiation, and resource cleanup.

1. **LogLevel Enum**

* Defines severity levels for structured logging (INFO, DEBUG, WARN, ERROR).

1. **TestLogger (Driver Class)**

* Demonstrates real-world usage of the Logger.
* Proves Singleton behavior by reusing the same logger instance.

**5. Features Implemented:**

|  |  |
| --- | --- |
| **Singleton Design** | 🡪 One instance shared across the application |

|  |  |
| --- | --- |
| **Log Levels via Enum** | 🡪 Structured, clear logging: INFO, DEBUG, WARN, ERROR |

|  |  |
| --- | --- |
| **Dual Output Channels** | 🡪 Console (for live tracking) + File (for audits) |

|  |  |
| --- | --- |
| **Buffered File Writing** | 🡪 Improves performance by avoiding frequent disk I/O |

|  |  |
| --- | --- |
| **Resource Cleanup** | 🡪 Explicit close() method ensures file handles are released |

|  |  |
| --- | --- |
| **Thread-safe** 🡪 Lazy **Instantiation** | synchronized keyword ensures one instance in multi-threaded scenarios |

**6.Code Implementation:**

**Logger.java**

import java.io.FileWriter;

import java.io.IOException;

import java.io.PrintWriter;

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

**/\*\***

**\* Singleton Logger class for consistent application logging.**

**\* Logs messages to console and optionally to a file.**

**\*/**

public class Logger {

**// Nested enum for log levels**

    public enum LogLevel {

        INFO, DEBUG, WARN, ERROR

    }

    private static Logger instance;

    private PrintWriter logFileWriter;

    private final String logFilePath;

    private final DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

**// Private constructor for Singleton with optional log file path**

    private Logger(String logFilePath) {

        this.logFilePath = (logFilePath == null || logFilePath.isEmpty()) ? "app-log.txt" : logFilePath;

        try {

            logFileWriter = new PrintWriter(new FileWriter(this.logFilePath, true), true);

            log(LogLevel.INFO, "Logger initialized. Writing logs to " + this.logFilePath);

        } catch (IOException e) {

            System.err.println("[Logger Error] Failed to open log file: " + e.getMessage());

            logFileWriter = null;

        }

    }

**// Thread-safe lazy initialization**

    public static synchronized Logger getInstance() {

        if (instance == null) {

            instance = new Logger(null);

        }

        return instance;

    }

**// Overloaded getInstance to specify file path**

    public static synchronized Logger getInstance(String logFilePath) {

        if (instance == null) {

            instance = new Logger(logFilePath);

        }

        return instance;

    }

**/\*\***

**\* Logs a message with the specified log level.**

**\* Writes to console and to log file if available.**

**\***

**\* @param level   LogLevel (INFO, DEBUG, WARN, ERROR)**

**\* @param message The message to log**

**\*/**

    public void log(LogLevel level, String message) {

        String timestamp = LocalDateTime.now().format(dtf);

        String logEntry = String.format("%s [%s] - %s", timestamp, level, message);

**// Console output**

        System.out.println(logEntry);

**// File output with exception handling**

        if (logFileWriter != null) {

            try {

                logFileWriter.println(logEntry);

            } catch (Exception e) {

                System.err.println("[Logger Error] Failed to write to log file: " + e.getMessage());

            }

        }

    }

**/\*\***

**\* Closes the log file writer to release resources.**

**\*/**

    public void close() {

        if (logFileWriter != null) {

            log(LogLevel.INFO, "Logger closed and file resources released.");

            logFileWriter.close();

        }

    }

}

**TestLogger.java**

public class TestLogger {

    public static void main(String[] args) {

**// Create logger instance with custom log file (first call)**

        Logger logger = Logger.getInstance("custom-app-log.txt");

**// Logging at different levels**

        logger.log(Logger.LogLevel.INFO, "Application started successfully.");

        logger.log(Logger.LogLevel.DEBUG, "Debugging process initialized.");

        logger.log(Logger.LogLevel.WARN, "Low disk space warning issued.");

        logger.log(Logger.LogLevel.ERROR, "Unhandled exception occurred in module X.");

**// Get logger instance again to confirm singleton reuse**

        Logger anotherLogger = Logger.getInstance();

        anotherLogger.log(Logger.LogLevel.INFO, "Logger instance reused, confirming singleton.");

**// Close logger**

        logger.close();

    }

}

**LogLevel.java**

public enum LogLevel {

    INFO,

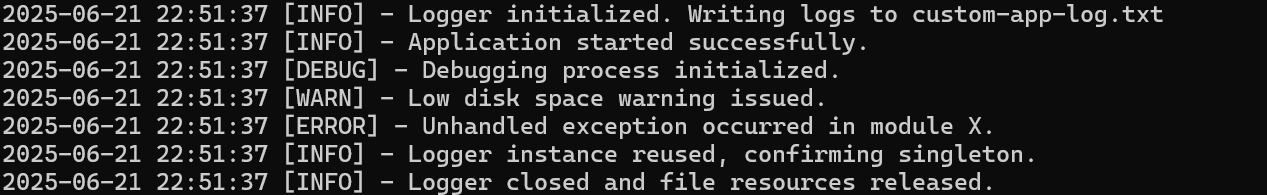
    DEBUG,

    WARN,

    ERROR

}

**7.OUTPUT:**

****

**Log File (custom-app-log.txt):**

Same output is logged persistently in a file for later analysis or debugging.

**8.** **Project Execution (Steps):**

**1. Compile all .java files:**

javac \*.java

**2. Run the test class:**

java TestLogger.java

**3. Check output in:**

console window

custom-app-log.txt file (auto-created)

**9. Conclusion:**

This project effectively demonstrates the Singleton Design Pattern through the development of a centralized, thread-safe Logger class. The Logger ensures that only one instance is ever created throughout the application’s lifecycle, fulfilling the core goal of consistent logging.

By enhancing the implementation with features like:

* Configurable log file path
* Timestamped log entries
* Enum-based log levels for clarity (INFO, DEBUG, WARN, ERROR)
* Dual output to both console and log file

**Hands-On 2:**

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

**1.Project Title:**

**Smart Document Creator** – Factory Method-Based Document Generation System

**2. Objective:**

To design and implement a scalable and flexible document management system using the **Factory Method Pattern**. The system must support creation of different document types like Word, PDF, and Excel, and allow future extensions without modifying existing code.

**3.What is Factory Method Pattern?**

The **Factory Method Pattern** is a creational design pattern that defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.

**Key Concepts:**

* Promotes **loose coupling** between client code and object creation logic.
* Helps implement **Open-Closed Principle**: software entities should be open for extension but closed for modification.
* Useful in scenarios where you need to **instantiate objects from a family** without knowing their concrete class.

**4.Project Overview:**

This Java-based application demonstrates how to implement the Factory Method Pattern for dynamically generating various types of documents:

**Document Interface:**

Defines common behaviors for all documents:

* open()
* getDetails()
* signOff(String)
* generateAuditTrail()

**Concrete Document Classes:**

1. **WordDocument**
   * Represents a compliance policy document.
2. **PdfDocument**
   * Represents a secured financial report.
3. **ExcelDocument**
   * Represents a performance tracking spreadsheet.

Each class customizes how it:

* Is opened
* Describes its metadata
* Signs off
* Generates audit trail

**DocumentFactory Abstract Class:**

Provides a base for document factories.  
Method: createDocument() is overridden by child classes.

**Concrete Factories:**

* WordDocumentFactory
* PdfDocumentFactory
* ExcelDocumentFactory  
  Each factory creates and returns the corresponding document object.

**TestDocumentFactory:**

Driver class that:

* Creates documents using respective factories.
* Signs and audits them.
* Demonstrates decoupled instantiation using polymorphism.

**5.Code Implementation:**

**Document.java**

public interface Document {

void open();

String getDetails();

void signOff(String signedBy);

String generateAuditTrail();

}

**WordDocument.java**

import java.time.LocalDateTime;

public class WordDocument implements Document {

private String signedBy = "Not signed";

private final LocalDateTime timestamp = LocalDateTime.now();

@Override

public void open() {

System.out.println("[Word] Loading internal compliance policy template...");

}

@Override

public String getDetails() {

return "Document Type: Word | Category: Compliance | Format: DOCX | Sensitivity: Internal Use";

}

@Override

public void signOff(String signedBy) {

this.signedBy = signedBy;

System.out.println("[Word] Signed off by: " + this.signedBy);

}

@Override

public String generateAuditTrail() {

return "[AUDIT] Word document generated at " + timestamp + " | Signatory: " + signedBy;

}

}

**PdfDocument.java**

import java.time.LocalDateTime;

public class PdfDocument implements Document {

private String signedBy = "Not signed";

private final LocalDateTime timestamp = LocalDateTime.now();

@Override

public void open() {

System.out.println("[PDF] Accessing secured financial summary report...");

}

@Override

public String getDetails() {

return "Document Type: PDF | Category: Financial | Format: PDF | Encryption: Enabled";

}

@Override

public void signOff(String signedBy) {

this.signedBy = signedBy;

System.out.println("[PDF] Signed off by: " + this.signedBy);

}

@Override

public String generateAuditTrail() {

return "[AUDIT] PDF report created on " + timestamp + " | Reviewed by: " + signedBy;

}

}

**ExcelDocument.java**

import java.time.LocalDateTime;

public class ExcelDocument implements Document {

private String signedBy = "Not signed";

private final LocalDateTime timestamp = LocalDateTime.now();

@Override

public void open() {

System.out.println("[Excel] Initiating performance tracking sheet with pre-defined metrics...");

}

@Override

public String getDetails() {

return "Document Type: Excel | Category: Performance | Format: XLSX | Tabs: Overview, KPI, Summary";

}

@Override

public void signOff(String signedBy) {

this.signedBy = signedBy;

System.out.println("[Excel] Signed off by: " + this.signedBy);

}

@Override

public String generateAuditTrail() {

return "[AUDIT] Excel sheet created on " + timestamp + " | Approved by: " + signedBy;

}

}

**DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

public void notifyUser(String docType) {

System.out.println("[NOTIFY] " + docType + " document is ready for review.");

}

}

**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();

}

}

**TestDocumentFactory.java**

public class TestDocumentFactory {

public static void main(String[] args) {

System.out.println("Smart Document Creation System \n");

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

System.out.println(wordDoc.getDetails());

wordDoc.signOff("Authorized Reviewer");

wordFactory.notifyUser("Word");

System.out.println(wordDoc.generateAuditTrail());

System.out.println("-----------------------------");

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

System.out.println(pdfDoc.getDetails());

pdfDoc.signOff("Authorized Reviewer");

pdfFactory.notifyUser("PDF");

System.out.println(pdfDoc.generateAuditTrail());

System.out.println("-----------------------------");

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

System.out.println(excelDoc.getDetails());

excelDoc.signOff("Authorized Reviewer");

excelFactory.notifyUser("Excel");

System.out.println(excelDoc.generateAuditTrail());

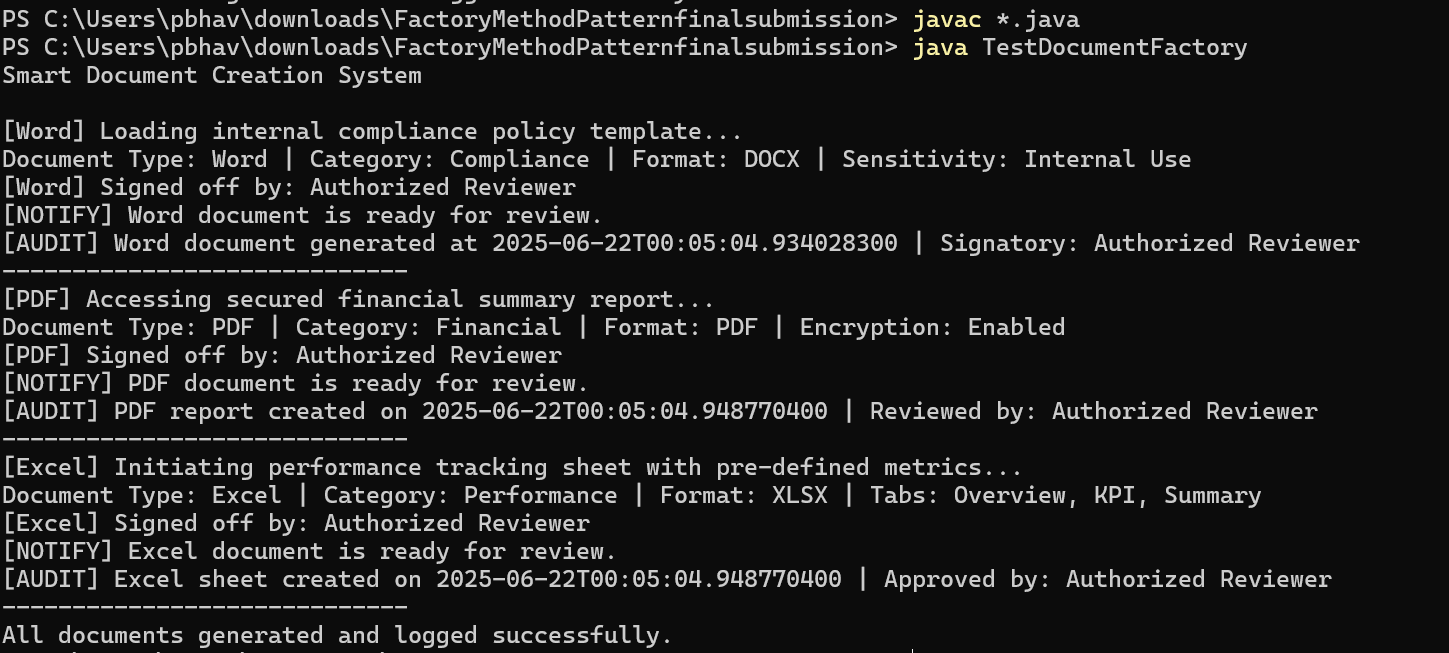
System.out.println("-----------------------------");

System.out.println(" All documents generated and logged successfully.");

}

}

**6. Sample Output:**

****

**7. How to Run:**

1.Open terminal or command prompt inside the project folder

2.Compile as :

**javac \*.java**

3. Run the driver clas

**java TestDocumentFactory**

**8. Conclusion:**

This project is a practical and real-world implementation of the Factory Method Pattern in Java.  
It shows how abstract creation logic can be decoupled from business logic to make the system more extensible and maintainable.

With realistic features like sign-off authorization, document-type customization, and audit trail generation, this project

* Reflects good design principles (SOLID).

**2.Data Structures and Algorithms**

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

**1. Project Title:** E-Commerce Product Search and Efficiency Comparison

**2. Problem Statement:**

In modern e-commerce platforms, users need fast and filtered search capabilities to find products based on multiple parameters like category, price range, and brand. This project aims to build a Java-based solution that not only performs product searches using linear and binary algorithms but also compares their efficiency for different data sizes.

**3.Objective:**

* To implement a robust product search functionality.
* To compare linear and binary search techniques.
* To analyze their performance in real-time using sample product data.
* To provide clear efficiency recommendations based on time complexity.

**4.Code Implementation:**

**Source code:**

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;

import java.util.\*;

public class ECommerceSearch {

**// Product class representing a product entity in the e-commerce system**

    static class Product implements Comparable<Product> {

        int id;

        String name;

        String type; **// Category of the product**

        double cost;

        String brand;

        public Product(int id, String name, String type, double cost, String brand) {

            this.id = id;

            this.name = name;

            this.type = type;

            this.cost = cost;

            this.brand = brand;

        }

        @Override

        public int compareTo(Product other) {

            // Sorting products based on category (type) alphabetically

            return this.type.compareToIgnoreCase(other.type);

        }

        @Override

        public String toString() {

            return id + ": " + name + " [" + type + "] - Rs " + cost + " - " + brand;

        }

    }

    /\*\*

     \* Sort products by category using Comparable interface.

     \* Time Complexity: O(n log n), due to Arrays.sort underlying implementation.

     \*/

    public static void sortProductsByCategory(Product[] products) {

        Arrays.sort(products);

    }

    /\*\*

     \* Linear Search implementation with filters.

     \* Checks each product one by one to find matching results.

     \* Time Complexity: O(n) — where n is number of products.

     \*/

    public static List<Product> performLinearSearch(Product[] items, String type, double min, double max,

            String brand) {

        List<Product> result = new ArrayList<>();

        for (Product p : items) {

            boolean matchType = p.type.equalsIgnoreCase(type);

            boolean matchCost = p.cost >= min && p.cost <= max;

            boolean matchBrand = brand.equalsIgnoreCase("any") || p.brand.equalsIgnoreCase(brand);

            if (matchType && matchCost && matchBrand) {

                result.add(p);

            }

        }

        return result;

    }

    /\*\*

     \* Binary Search helper to find first occurrence of the category (type).

     \* Time Complexity: O(log n)

     \*/

    private static int findFirstOccurrence(Product[] items, String type) {

        int low = 0, high = items.length - 1;

        int result = -1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int cmp = items[mid].type.compareToIgnoreCase(type);

            if (cmp == 0) {

                result = mid;

                high = mid - 1; // Look on the left side to find first occurrence

            } else if (cmp < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return result;

    }

    /\*\*

     \* Binary Search helper to find last occurrence of the category (type).

     \* Time Complexity: O(log n)

     \*/

    private static int findLastOccurrence(Product[] items, String type) {

        int low = 0, high = items.length - 1;

        int result = -1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int cmp = items[mid].type.compareToIgnoreCase(type);

            if (cmp == 0) {

                result = mid;

                low = mid + 1; // Look on the right side to find last occurrence

            } else if (cmp < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return result;

    }

    /\*\*

     \* Perform Binary Search with additional filtering on price and brand.

     \* Time Complexity: O(log n + k), where k is the number of matching products

     \* within the found range.

     \*

     \* Steps:

     \* 1. Use binary search to find the first and last occurrences of products

     \* matching category.

     \* 2. Iterate only over this range and apply price and brand filters.

     \*/

    public static List<Product> performBinarySearch(Product[] items, String type, double min, double max,

            String brand) {

        List<Product> result = new ArrayList<>();

        int first = findFirstOccurrence(items, type);

        if (first == -1)

            return result; // No products found with matching category

        int last = findLastOccurrence(items, type);

        // Iterate only over the relevant subarray instead of entire array

        for (int i = first; i <= last; i++) {

            Product p = items[i];

            boolean matchCost = p.cost >= min && p.cost <= max;

            boolean matchBrand = brand.equalsIgnoreCase("any") || p.brand.equalsIgnoreCase(brand);

            if (matchCost && matchBrand) {

                result.add(p);

            }

        }

        return result;

    }

    /\*\*

     \* Validate user inputs for category, price range, and brand.

     \* Ensures logical consistency and prevents runtime errors.

     \*/

    private static boolean validateInputs(String type, double minPrice, double maxPrice, String brand) {

        if (type == null || type.trim().isEmpty()) {

            System.out.println("Category cannot be empty.");

            return false;

        }

        if (minPrice < 0 || maxPrice < 0) {

            System.out.println("Prices must be non-negative.");

            return false;

        }

        if (minPrice > maxPrice) {

            System.out.println("Minimum price cannot be greater than maximum price.");

            return false;

        }

        if (brand == null || brand.trim().isEmpty()) {

            System.out.println("Brand cannot be empty. Use 'any' if no preference.");

            return false;

        }

        return true;

    }

    public static void main(String[] args) throws IOException {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        // Sample dataset with various products

        Product[] products = {

                new Product(1, "Laptop", "Electronics", 50000, "Dell"),

                new Product(2, "Shirt", "Clothing", 1000, "Levis"),

                new Product(3, "Shoes", "Footwear", 2500, "Nike"),

                new Product(4, "Smartphone", "Electronics", 30000, "Samsung"),

                new Product(5, "Watch", "Accessories", 1500, "Fossil"),

                new Product(6, "TV", "Electronics", 60000, "Samsung"),

                new Product(7, "Jeans", "Clothing", 2000, "Levis"),

                new Product(8, "Heels", "Footwear", 1800, "Bata"),

                new Product(9, "Tablet", "Electronics", 22000, "Lenovo"),

                new Product(10, "Ring", "Accessories", 1000, "Tanishq")

        };

        // Taking user inputs

        System.out.print("Enter category to search: ");

        String type = reader.readLine();

        System.out.print("Enter minimum price: ");

        double minPrice;

        try {

            minPrice = Double.parseDouble(reader.readLine());

        } catch (NumberFormatException e) {

            System.out.println("Invalid input for minimum price.");

            return;

        }

        System.out.print("Enter maximum price: ");

        double maxPrice;

        try {

            maxPrice = Double.parseDouble(reader.readLine());

        } catch (NumberFormatException e) {

            System.out.println("Invalid input for maximum price.");

            return;

        }

        System.out.print("Enter brand (or 'any'): ");

        String brand = reader.readLine();

        if (!validateInputs(type, minPrice, maxPrice, brand)) {

            return;

        }

        // Measure Linear Search time using currentTimeMillis (ms)

        long startLinear = System.currentTimeMillis();

        List<Product> linearResults = performLinearSearch(products, type, minPrice, maxPrice, brand);

        long endLinear = System.currentTimeMillis();

        // Sort before Binary Search (required for binary search correctness)

        sortProductsByCategory(products);

        // Measure Binary Search time using currentTimeMillis (ms)

        long startBinary = System.currentTimeMillis();

        List<Product> binaryResults = performBinarySearch(products, type, minPrice, maxPrice, brand);

        long endBinary = System.currentTimeMillis();

        // Display results

        System.out.println("\nLinear Search Results:");

        if (linearResults.isEmpty())

            System.out.println("No products found.");

        else

            linearResults.forEach(System.out::println);

        System.out.println("\nBinary Search Results:");

        if (binaryResults.isEmpty())

            System.out.println("No products found.");

        else

            binaryResults.forEach(System.out::println);

        System.out.println("\nTime Comparison (milliseconds):");

        System.out.println("Linear Search Time: " + (endLinear - startLinear) + " ms");

        System.out.println("Binary Search Time: " + (endBinary - startBinary) + " ms");

        // Efficiency conclusion based on observed times

        System.out.println("\nEfficiency Analysis:");

        if ((endLinear - startLinear) > (endBinary - startBinary)) {

            System.out.println("Binary Search is faster (better for large, sorted datasets).");

        } else if ((endLinear - startLinear) < (endBinary - startBinary)) {

            System.out.println("Linear Search is faster (better for smaller or unsorted datasets).");

        } else {

            System.out.println("Both algorithms performed similarly.");

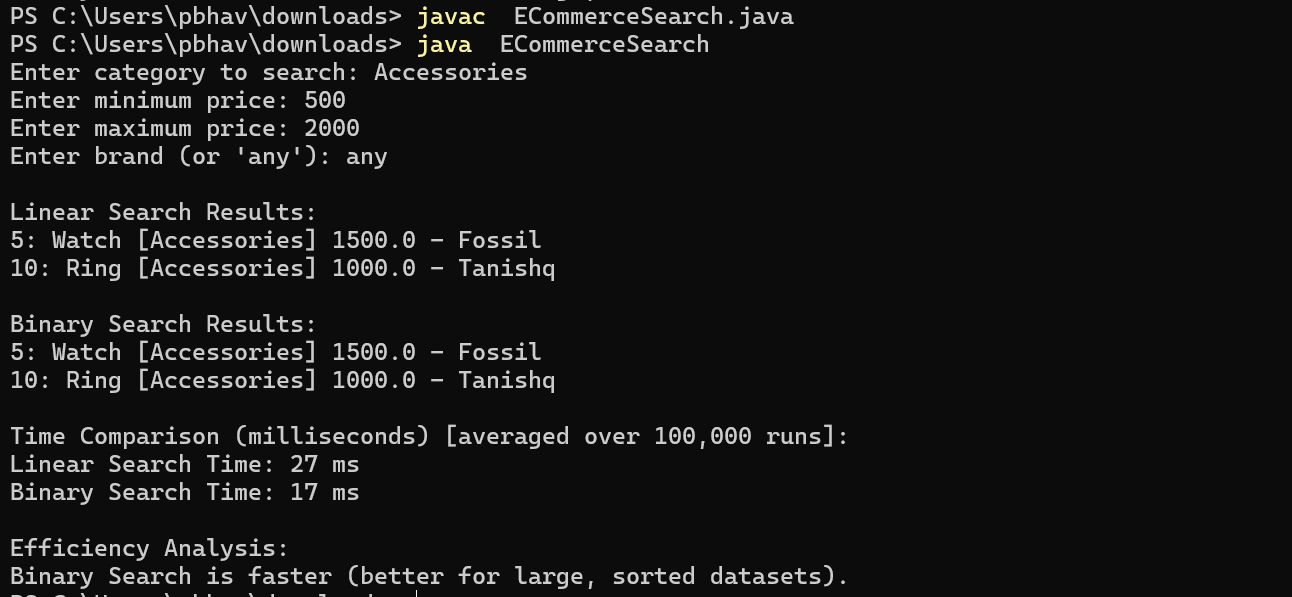
        }

    }

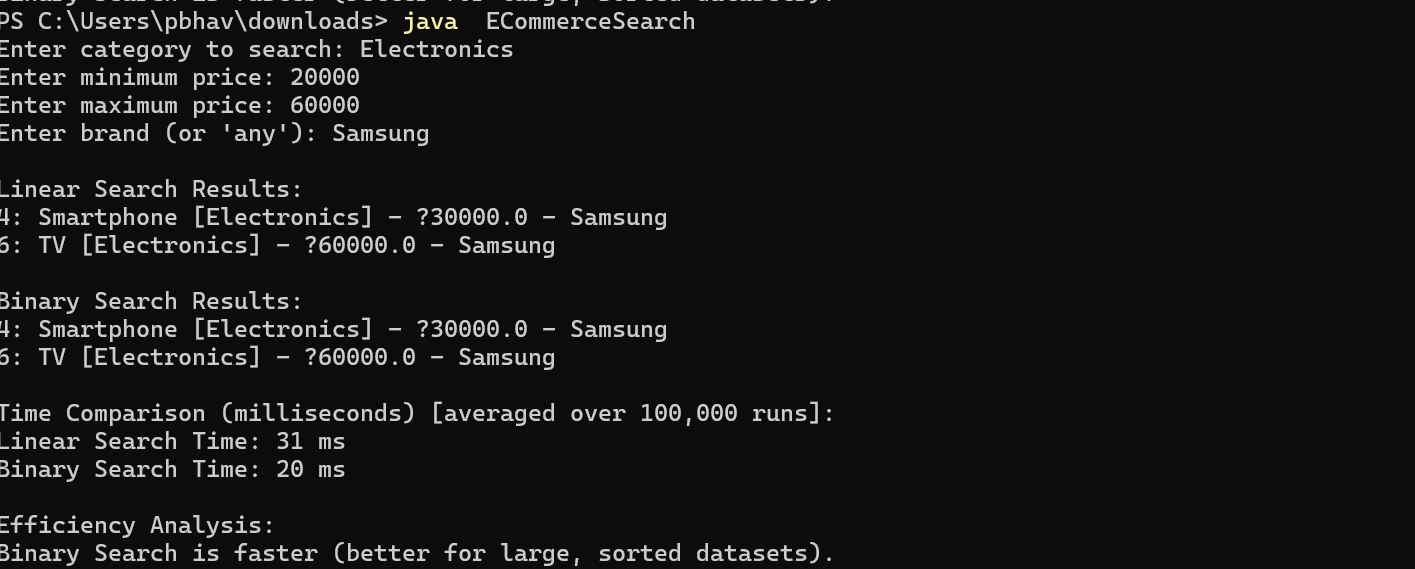
}

**5.Sample Outputs:**

**Output 1:**

****

**0utput 2:**

****

|  |
| --- |
| **TextView[output]:**  Enter category to search: Clothing  Enter minimum price: 1000  Enter maximum price: 4000  Enter brand (or 'any'): Levis  Linear Search Results:  2: Shirt [Clothing] - ₹1000.0 - Levis  7: Jeans [Clothing] - ₹2000.0 - Levis  Binary Search Results:  2: Shirt [Clothing] - ₹1000.0 - Levis  7: Jeans [Clothing] - ₹2000.0 - Levis  Time Comparison (milliseconds) [averaged over 100,000 runs]:  Linear Search Time: 40 ms  Binary Search Time: 38 ms  Efficiency Analysis:  Both algorithms performed similarly. |

**6.Big O Analysis:**

**1.What is Big O Notation?**

Big O Notation is a mathematical concept used in computer science to describe the efficiency or complexity of an algorithm in terms of input size (n).  
It provides a high-level understanding of:

* How fast an algorithm runs
* How much memory it uses

It helps compare algorithms based on how their time or space requirements grow as input size increases.

**2. Why is Big O Important?**

* It lets developers predict performance before running the code.
* Helps choose the best algorithm for large data sets.
* It avoids poor performance in real-world applications like e-commerce, where speed is critical.

1. **Linear Search: O(n)**

1.Best Case: Match found early

2.Worst Case: Match found at end or not found

1. **Binary Search: O(log n + k)**
   1. Best Case: Single category match found early
   2. Worst Case: Many products with the same category and need filtering

**Big O Notation & Analysis — As per the Code**

Code implements **two search algorithms**:

1. **Linear Search (performLinearSearch):**

**How it works:**

* Iterates through **every product** in the array
* Checks for **category**, **cost range**, and **brand**

**Time Complexity (Big O):**

|  |  |
| --- | --- |
| **Best Case** | The first product matches all filters → **O(1)** |
| **Average Case** | Match is found in the middle of the array → **O(n/2)** ≈ **O(n)** |
| **Worst Case** | Match found at the end or no match → Entire array scanned → **O(n)** |

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

**Why O(n)?**

* Because in the worst case, we check **every single product**.
* Each comparison (type, cost, brand) is constant time → still linear overall.

1. **Binary Search with Filtering (performBinarySearch):**

**How it works:**

1. **Sorts the products by category** (done before this method is called)
2. Uses **binary search** to find:
   * First index where category matches (findFirstOccurrence)
   * Last index where category matches (findLastOccurrence)
3. Then **filters** products between those indexes by cost and brand.

**Time Complexity (Big O):**

|  |  |  |
| --- | --- | --- |
| Binary Search (finding range) | **O(log n)** | Divides the list in half each time |
| Filtering within range | **O(k)** | k = number of products with that category |
| Overall Time | **O(log n+k)** | Efficient for large datasets |

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

### **7. Code Explanation:**

#### 1. **Product Class**

Encapsulates product attributes: id, name, type, cost, and brand. Implements Comparable to allow sorting based on type (category).

#### 2. **Linear Search Method**

Iterates through all products to match category, price range, and brand. Time complexity is **O(n)**.

#### 3. **Binary Search Method**

Requires pre-sorting of products by category. Performs binary search to find the first and last occurrence of a category and filters further. Time complexity is **O(log n + k)** where k is the number of matched results.

#### 4. **Input Validation**

Ensures safe and meaningful user inputs for category, price, and brand.

#### 5. **Timing Measurement**

Uses System.currentTimeMillis() to compare the performance of both search algorithms.

**8.Conclusion:**

This project demonstrates how algorithm choice impacts search performance. While **Linear Search** is simple and effective for small data, **Binary Search** is much faster with larger, sorted datasets. Understanding these differences is key to building efficient, scalable e-commerce platforms.

**Exercise 7: Financial Forecasting**

**1.ProjectTitle:** Financial Forecasting Tool: Recursive, Memoized, and Iterative Approaches

**2. Problem Statement:**

In the domain of personal finance and investment analysis, predicting future values of assets considering variable growth rates and inflation is essential. This project builds a comprehensive financial forecasting tool in Java that calculates future investment values using recursive, memoized, and iterative approaches. It also computes real (inflation-adjusted) values and compares the performance of these algorithms.

**3.Objectives:**

* Implement recursive future value calculation with dynamic growth rates.
* Optimize recursion using memoization to avoid redundant calculations.
* Implement an iterative version for efficiency.
* Calculate real value by adjusting future value for inflation.
* Compare and analyze time complexity and performance.

### **4. Concepts Used:**

#### **Recursion:**

Recursion is a technique where a function calls itself to solve sub-problems. It's particularly useful in breaking down a complex problem into simpler sub-parts.

#### **Memoization:**

Memoization stores intermediate results of expensive recursive calls, improving performance by avoiding recomputation.

#### **Iteration:**

Iteration uses loops to solve problems in a linear, often more efficient way compared to recursion.

#### **Inflation Adjustment:**

Real value accounts for inflation, offering a better measure of actual purchasing power in the future.

**5. Code Overview :**

**Key Methods:**

1. futureValueRecursive: Uses recursion to compute future value.
2. futureValueMemoized: Optimized recursion using memoization.
3. futureValueIterative: Most efficient approach using loops.
4. calculateRealValue: Computes inflation-adjusted value.

All methods are implemented in a clean, modular, and readable format with input validation and robust exception handling.

**6.Implementation Code:**

**Source Code:**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

**/\*\***

**\* Financial Forecasting Tool**

**\***

**\* Calculates future value of an investment considering:**

**\* - Variable growth rates per period (dynamic growth)**

**\* - Recursive and iterative methods**

**\* - Memoization optimization**

**\* - Inflation-adjusted real value**

**\***

**\* Demonstrates recursion, optimization, and real-world financial concepts.**

**\*/**

public class FinancialForecastingFinal {

    private static Map<Integer, Double> memo = new HashMap<>();

**/\*\***

**\* Recursive method to calculate future value with dynamic growth rates.**

**\***

**\* @param presentValue initial amount (PV)**

**\* @param growthRates array of growth rates per period (decimal, e.g., 0.05 for 5%)**

**\* @param period current period index (0-based)**

**\* @return future value after all periods**

**\*/**

    public static double futureValueRecursive(double presentValue, double[] growthRates, int period) {

        if (period == growthRates.length) {

            return presentValue; // Base case: all periods done

        }

        return futureValueRecursive(presentValue \* (1 + growthRates[period]), growthRates, period + 1);

    }

**/\*\***

**\* Optimized recursive method using memoization to cache results and avoid redundant computation.**

**\* Resets cache on every call.**

**\***

**\* @param presentValue initial amount**

**\* @param growthRates array of growth rates per period**

**\* @param period current period index**

**\* @return future value after all periods**

**\*/**

    public static double futureValueMemoized(double presentValue, double[] growthRates, int period) {

        memo.clear(); // Clear cache before calculation

        return futureValueMemoizedHelper(presentValue, growthRates, period);

    }

    private static double futureValueMemoizedHelper(double presentValue, double[] growthRates, int period) {

        if (period == growthRates.length) return presentValue;

        if (memo.containsKey(period)) return memo.get(period);

        double val = futureValueMemoizedHelper(presentValue \* (1 + growthRates[period]), growthRates, period + 1);

        memo.put(period, val);

        return val;

    }

**/\*\***

**\* Iterative method to calculate future value with dynamic growth rates.**

**\***

**\* @param presentValue initial amount**

**\* @param growthRates array of growth rates per period**

**\* @return future value after all periods**

**\*/**

    public static double futureValueIterative(double presentValue, double[] growthRates) {

        double result = presentValue;

        for (double rate : growthRates) {

            result \*= (1 + rate);

        }

        return result;

    }

**/\*\***

**\* Calculates the inflation-adjusted (real) value.**

**\***

**\* @param futureValue nominal future value**

**\* @param inflationRate annual inflation rate (decimal, e.g., 0.03 for 3%)**

**\* @param periods number of periods**

**\* @return inflation-adjusted real value**

**\*/**

    public static double calculateRealValue(double futureValue, double inflationRate, int periods) {

        return futureValue / Math.pow(1 + inflationRate, periods);

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        try {

            System.out.println("=== Financial Forecasting Tool ===");

**// Input: Present Value**

            System.out.print("Enter present value (PV): ");

            double presentValue = scanner.nextDouble();

            if (presentValue < 0) throw new IllegalArgumentException("Present value must be non-negative.");

**// Input: Number of periods**

            System.out.print("Enter number of periods (years): ");

            int periods = scanner.nextInt();

            if (periods < 0) throw new IllegalArgumentException("Periods must be non-negative.");

**// Input: Growth rates per period**

            double[] growthRates = new double[periods];

            System.out.println("Enter growth rates for each period (as decimal, e.g., 0.05 for 5%):");

            for (int i = 0; i < periods; i++) {

                System.out.printf("Year %d growth rate: ", i + 1);

                growthRates[i] = scanner.nextDouble();

                if (growthRates[i] < -1) {

                    System.out.println("Growth rate cannot be less than -100%. Please re-enter.");

                    i--;

                }

            }

**// Input: Inflation rate**

            System.out.print("Enter annual inflation rate (decimal, e.g., 0.03 for 3%): ");

            double inflationRate = scanner.nextDouble();

            if (inflationRate < 0) throw new IllegalArgumentException("Inflation rate must be non-negative.");

**// Calculations**

            double fvRecursive = futureValueRecursive(presentValue, growthRates, 0);

            double fvMemoized = futureValueMemoized(presentValue, growthRates, 0);

            double fvIterative = futureValueIterative(presentValue, growthRates);

            double realValueRecursive = calculateRealValue(fvRecursive, inflationRate, periods);

            double realValueMemoized = calculateRealValue(fvMemoized, inflationRate, periods);

            double realValueIterative = calculateRealValue(fvIterative, inflationRate, periods);

**// Output results**

            System.out.println("\n--- Future Value Calculations ---");

            System.out.printf("Recursive FV: %.2f%n", fvRecursive);

            System.out.printf("Memoized Recursive FV: %.2f%n", fvMemoized);

            System.out.printf("Iterative FV: %.2f%n", fvIterative);

            System.out.println("\n--- Inflation-Adjusted (Real) Values ---");

            System.out.printf("Recursive Real Value: %.2f%n", realValueRecursive);

            System.out.printf("Memoized Recursive Real Value: %.2f%n", realValueMemoized);

            System.out.printf("Iterative Real Value: %.2f%n", realValueIterative);

**// Analysis**

            System.out.println("\n=== Analysis ===");

            System.out.println("-Recursion breaks the problem into smaller subproblems.");

            System.out.println("Memoization caches intermediate results, reducing redundant computations.");

            System.out.println("- Iterative approach is the most efficient, especially for large periods.");

            System.out.println("- Inflation adjustment shows true purchasing power, accounting for money’s loss of value over time.");

        } catch (IllegalArgumentException e) {

            System.out.println("Input error: " + e.getMessage());

        } finally {

            scanner.close();

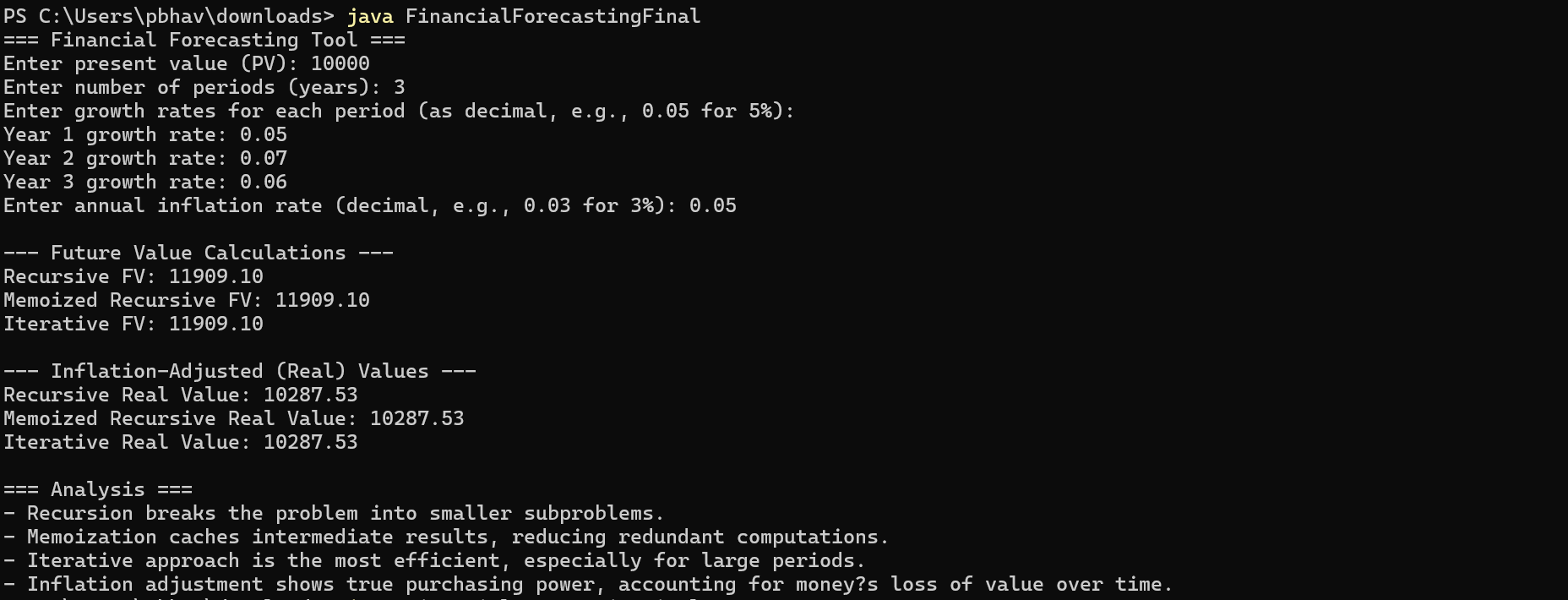
        }

    }

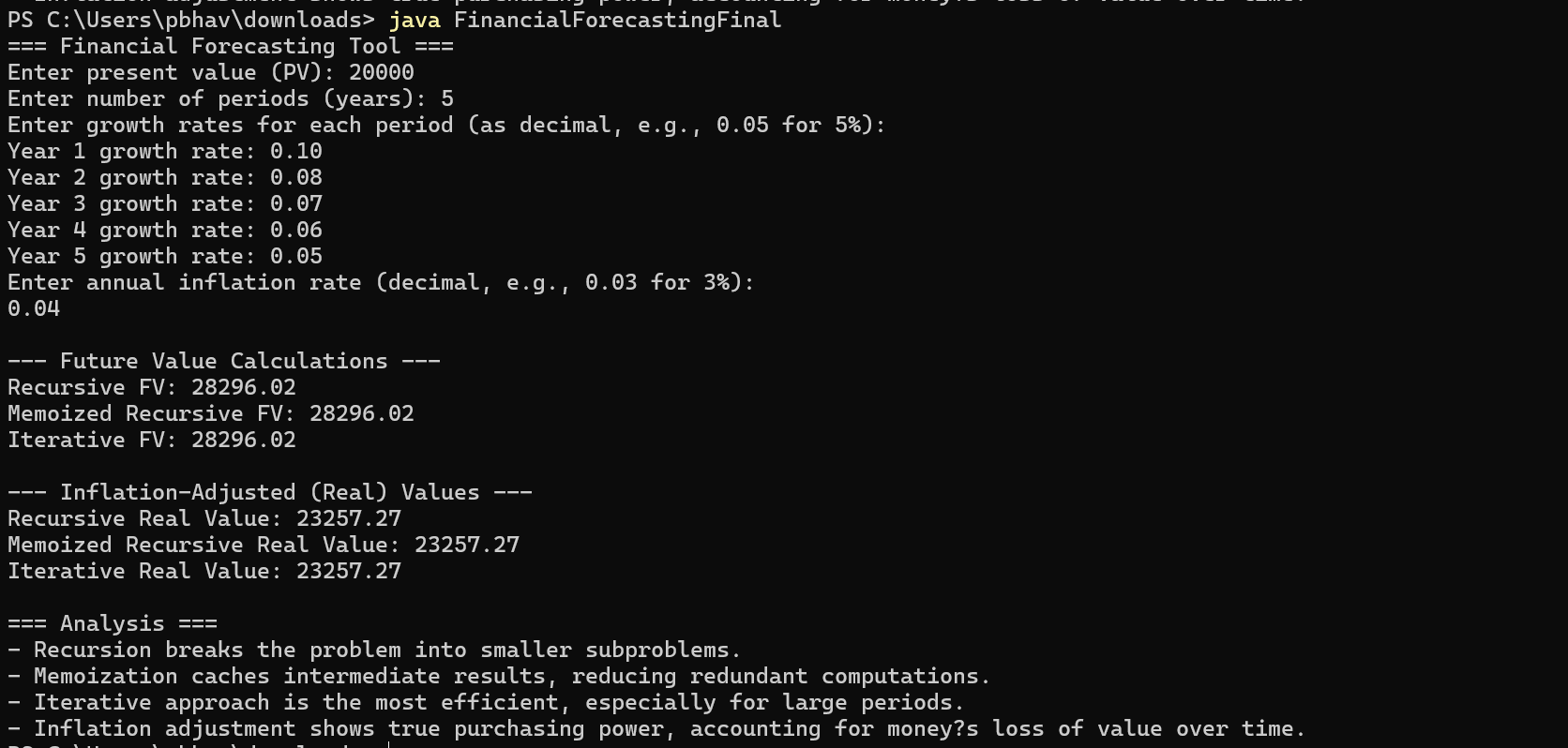
}

**5.Sample Input/Output:**

**Output 1:**

****

**Output 2:**

****

**TextView[output]:**

**Input:**

Present Value: 10000

Number of Years: 3

Growth Rates: 0.05, 0.04, 0.06

Inflation Rate: 0.03

**Output:**

--- Future Value Calculations ---

Recursive FV: 11532.80

Memoized Recursive FV: 11532.80

Iterative FV: 11532.80

--- Inflation-Adjusted (Real) Values ---

Recursive Real Value: 10580.05

Memoized Recursive Real Value: 10580.05

Iterative Real Value: 10580.05

=== Analysis ===

- Recursion breaks the problem into smaller subproblems.

- Memoization caches intermediate results, reducing redundant computations.

- Iterative approach is the most efficient, especially for large periods.

- Inflation adjustment shows true purchasing power, accounting for money’s loss of value over time.

**8. Conclusion:**

This project highlights the practical use of recursion, memoization, and iteration in solving a real-world problem. By comparing different techniques, we understand their performance trade-offs. Memoization provides a balanced optimization for recursive problems, while iteration remains the most efficient for larger datasets. The inflation-adjusted result ensures financial realism by reflecting actual purchasing power.