**Student Name :** KARTHYAYENI P

**Registration No :** 22CSR089

**Course/Batch :** B.E COMPUTER SCIENCE AND ENGINEERING

**EXERCISE 1: IMPLEMENTING THE SINGLETON PATTERN**

**Introduction:**

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

**Objective:**

The goal is to implement the Singleton Pattern and verify that only one instance of the Logger class is created and shared across all parts of the application.

**Implementation Breakdown:**

1. Singleton Class – TestLogger

public class Logger {

    private static final Logger singleInstance = new Logger();

    private Logger() {

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

    }

    public static Logger getInstance() {

        return singleInstance;

    }

    public void printLog(String msg) {

        System.out.println("[Logger] " + msg);

    }

}

1. Test Class – TestLogger

public class TestLogger {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.printLog("First log message");

        Logger logger2 = Logger.getInstance();

        logger2.printLog("Second log message");

        if (logger1 == logger2) {

            System.out.println("Singleton confirmed: Same instance used.");

        } else {

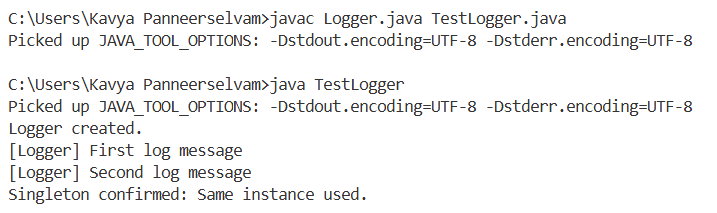
            System.out.println("Singleton failed: Different instances.");

        }

    }

}

**Output:**

****

**Conclusion:**

This assignment successfully demonstrates the Singleton Pattern. The Logger class was designed to allow only a single instance, which helps ensure consistent logging across the system.

**EXERCISE 2: IMPLEMENTING THE FACTORY METHOD PATTERN**

**Introduction:**

The Factory Method Pattern is a design pattern used to create objects without specifying their exact class. It promotes flexibility and scalability.This project uses the pattern to create different types of documents like Word, PDF, and Excel through dedicated factory classes.

**Objective:**

To implement a system that creates Word, PDF, and Excel documents using the Factory Method Pattern in a clean and extensible way.

**Implementation Breakdown:**

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Paths;

// 1. Document interface

interface Document {

    void open();

    void save();

    void close();

    String getType();

}

// 2. Concrete Document Classes

class WordDocument implements Document {

    private String fileName;

    private String content;

    public WordDocument() {

        this.fileName = "document.txt";

        this.content = "";

    }

    @Override

    public void open() {

        System.out.println("Opening Word document: " + fileName);

        content = "=== WORD DOCUMENT ===\n";

    }

    @Override

    public void save() {

        try {

            Files.write(Paths.get(fileName), content.getBytes());

            System.out.println("Word document saved: " + fileName);

        } catch (IOException e) {

            System.err.println("Error saving Word document: " + e.getMessage());

        }

    }

    @Override

    public void close() {

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

    }

    @Override

    public String getType() {

        return "Word Document";

    }

    public void addContent(String text) {

        content += text + "\n";

    }

}

class PdfDocument implements Document {

    private String fileName;

    private StringBuilder content;

    public PdfDocument() {

        this.fileName = "document.pdf";

        this.content = new StringBuilder();

    }

    @Override

    public void open() {

        System.out.println("Opening PDF document: " + fileName);

        content.append("%PDF-1.4\n");

        content.append("1 0 obj << /Type /Catalog /Pages 2 0 R >> endobj\n");

        content.append("2 0 obj << /Type /Pages /Kids [3 0 R] /Count 1 >> endobj\n");

        content.append("3 0 obj << /Type /Page /Parent 2 0 R /Contents 4 0 R >> endobj\n");

        content.append("4 0 obj << /Length 44 >> stream\nBT /F1 12 Tf 100 700 Td (Hello PDF!) Tj ET\nendstream endobj\n");

        content.append("xref\n0 5\ntrailer << /Size 5 /Root 1 0 R >>\nstartxref\n238\n%%EOF");

    }

    @Override

    public void save() {

        try {

            Files.write(Paths.get(fileName), content.toString().getBytes());

            System.out.println("PDF document saved: " + fileName);

        } catch (IOException e) {

            System.err.println("Error saving PDF document: " + e.getMessage());

        }

    }

    @Override

    public void close() {

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

    }

    @Override

    public String getType() {

        return "PDF Document";

    }

}

class ExcelDocument implements Document {

    private String fileName;

    private String content;

    public ExcelDocument() {

        this.fileName = "document.csv";

        this.content = "";

    }

    @Override

    public void open() {

        System.out.println("Opening Excel document: " + fileName);

        content = "Name,Age,Department\n";

    }

    @Override

    public void save() {

        try {

            Files.write(Paths.get(fileName), content.getBytes());

            System.out.println("Excel document saved: " + fileName);

        } catch (IOException e) {

            System.err.println("Error saving Excel document: " + e.getMessage());

        }

    }

    @Override

    public void close() {

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

    }

    @Override

    public String getType() {

        return "Excel Document";

    }

    public void addRow(String name, String age, String dept) {

        content += name + "," + age + "," + dept + "\n";

    }

}

// 3. Abstract Factory Class

abstract class DocumentFactory {

    public abstract Document createDocument();

}

// 4. Concrete Factory Classes

class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

// 5. Test Class

public class FactoryMethodPatternExample {

    public static void main(String[] args) {

        System.out.println("=== Factory Method Pattern Demo ===\n");

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        ((WordDocument) wordDoc).addContent("This is a Word document content.");

        wordDoc.save();

        wordDoc.close();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        pdfDoc.save();

        pdfDoc.close();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

        ((ExcelDocument) excelDoc).addRow("John", "25", "IT");

        ((ExcelDocument) excelDoc).addRow("Mary", "30", "HR");

        excelDoc.save();

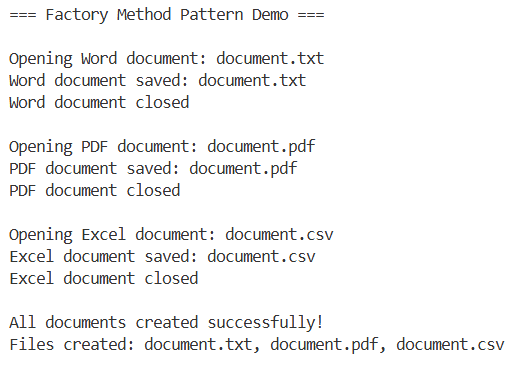
        excelDoc.close();

        System.out.println("Files created: document.txt, document.pdf, document.csv");

    }

}

**Output:**

****

**Conclusion:**

The Factory Method Pattern helps separate object creation logic from usage, making the system easy to maintain and extend for future document types.