**Student Name :** Jayaanth S R

**Registration No :** 22CSR081

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

**EXERCISE 1: IMPLEMENTING THE SINGLETON PATTERN**

**Introduction:**

The Singleton Pattern ensures that a class has only one instance and provides a global access point to it. It is used for managing shared resources like loggers, config files, DB connections, etc.

**Objective:**

To create a singleton Logger class that guarantees only one instance is used to handle all logging activities across the application.

**Implementation Breakdown:**

1. Logger.java

public class Logger {

    private static final Logger loggerInstance = new Logger();

    private Logger() {

        System.out.println("[Logger System] Instance initialized.");

    }

    public static Logger getLogger() {

        return loggerInstance;

    }

    public void record(String logMsg) {

        System.out.println(">> LOG: " + logMsg);

    }

}

1. LoggerTest

public class LoggerTest {

    public static void main(String[] args) {

        Logger logOne = Logger.getLogger();

        logOne.record("First event triggered.");

        Logger logTwo = Logger.getLogger();

        logTwo.record("Second event executed.");

        if (logOne == logTwo) {

            System.out.println("Singleton Logger is working correctly.");

        } else {

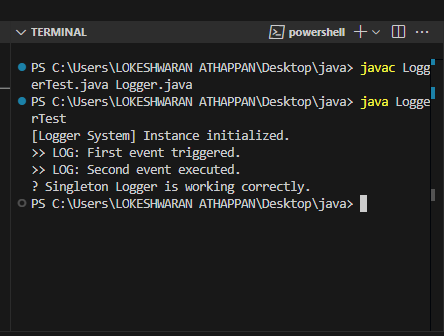
            System.out.println("Logger is not Singleton.");

        }

    }

}

**Output:**

****

**Conclusion:**

The Singleton Pattern was successfully implemented using a Logger class. The test verified that both references pointed to the same instance, confirming singleton behaviour. This design pattern helps manage global shared resources with consistent control.

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

**Introduction:**

The Factory Method Pattern is used to create objects without specifying the exact class. It improves flexibility and allows easy object creation based on type.

**Objective:**

To build a document system that can generate Word, PDF, and Excel files using factories, making the code cleaner and easier to extend.

**Implementation Breakdown:**

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Paths;

// 1. Document interface

interface Document {

    void create();

    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.docx";

        this.content = "";

    }

    @Override

    public void create() {

        System.out.println("Initializing Word file: " + fileName);

        content = "=== MICROSOFT WORD FILE ===\n";

        content += "Word processor document generated via Factory Pattern implementation.\n";

        content += "Text formatting and styling capabilities available.\n";

    }

    @Override

    public void open() {

        System.out.println("Launching Word file: " + fileName);

    }

    @Override

    public void save() {

        try {

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

            System.out.println("Word file successfully stored: " + fileName);

        } catch (IOException e) {

            System.err.println("Failed to store Word file: " + e.getMessage());

        }

    }

    @Override

    public void close() {

        System.out.println("Terminating Word file: " + fileName);

    }

    @Override

    public String getType() {

        return "Word Document";

    }

    public void insertText(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 create() {

        System.out.println("Generating PDF file: " + fileName);

        // Basic PDF structure

        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 85 >> stream\n");

        content.append("BT /F1 12 Tf 100 700 Td (PDF File Generated Successfully!) Tj ");

        content.append("0 -15 Td (Factory Pattern Implementation) Tj ET\n");

        content.append("endstream endobj\n");

        content.append("xref\n0 5\ntrailer << /Size 5 /Root 1 0 R >>\nstartxref\n280\n%%EOF");

    }

    @Override

    public void open() {

        System.out.println("Accessing PDF file: " + fileName);

    }

    @Override

    public void save() {

        try {

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

            System.out.println("PDF file written to disk: " + fileName);

        } catch (IOException e) {

            System.err.println("Unable to write PDF file: " + e.getMessage());

        }

    }

    @Override

    public void close() {

        System.out.println("PDF file session ended: " + fileName);

    }

    @Override

    public String getType() {

        return "PDF Document";

    }

}

class ExcelDocument implements Document {

    private String fileName;

    private String content;

    public ExcelDocument() {

        this.fileName = "spreadsheet.xlsx";

        this.content = "";

    }

    @Override

    public void create() {

        System.out.println("Building Excel spreadsheet: " + fileName);

        content = "Employee\_Name,Years,Team,Income\n";

        content += "Robert Chen,25,Development,68000\n";

        content += "Sarah Wilson,31,Sales,72000\n";

        content += "David Kumar,28,Operations,65000\n";

    }

    @Override

    public void open() {

        System.out.println("Loading Excel spreadsheet: " + fileName);

    }

    @Override

    public void save() {

        try {

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

            System.out.println("Excel spreadsheet exported: " + fileName);

        } catch (IOException e) {

            System.err.println("Excel export failed: " + e.getMessage());

        }

    }

    @Override

    public void close() {

        System.out.println("Excel application terminated: " + fileName);

    }

    @Override

    public String getType() {

        return "Excel Document";

    }

    public void insertRecord(String empName, int years, String team, double income) {

        content += empName + "," + years + "," + team + "," + income + "\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("=== Document Factory Pattern Implementation Demo ===\n");

        DocumentFactory wordCreator = new WordDocumentFactory();

        Document wordFile = wordCreator.createDocument();

        wordFile.create();

        wordFile.open();

        ((WordDocument) wordFile).insertText("Extra paragraph inserted into Word file.");

        ((WordDocument) wordFile).insertText("Factory Method design pattern successfully implemented!");

        wordFile.save();

        wordFile.close();

        System.out.println("File Type: " + wordFile.getType());

        System.out.println();

        DocumentFactory pdfCreator = new PdfDocumentFactory();

        Document pdfFile = pdfCreator.createDocument();

        pdfFile.create();

        pdfFile.open();

        pdfFile.save();

        pdfFile.close();

        System.out.println("File Type: " + pdfFile.getType());

        System.out.println();

        DocumentFactory excelCreator = new ExcelDocumentFactory();

        Document excelFile = excelCreator.createDocument();

        excelFile.create();

        excelFile.open();

        ((ExcelDocument) excelFile).insertRecord("Emma Thompson", 27, "Marketing", 69000);

        ((ExcelDocument) excelFile).insertRecord("James Rodriguez", 33, "Technology", 82000);

        excelFile.save();

        System.out.println("File Type: " + excelFile.getType());

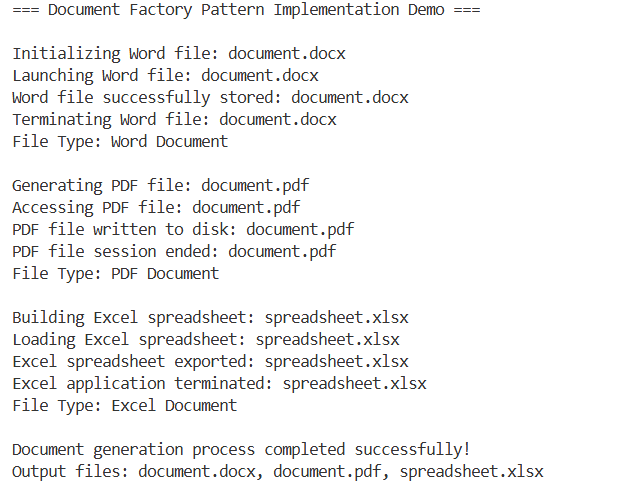
        System.out.println("Document generation process completed successfully!");

        System.out.println("Output files: document.docx, document.pdf, spreadsheet.xlsx");

    }

}

**Output:**

****

**Conclusion:**

The pattern was successfully applied to create different document types through separate factory classes. It made the system more organized, scalable, and easy to manage.