**Student Name :** HARINEE N J

**Registration No :** 22CSR063

**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. Logger Class

public class Logger {

    private static Logger instance;

    private Logger() {

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

    }

    public static synchronized Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

        System.out.println("[LOG] " + java.time.LocalDateTime.now() + ": " + message);

    }

    public void logError(String error) {

        System.out.println("[ERROR] " + java.time.LocalDateTime.now() + ": " + error);

    }

    public void logInfo(String info) {

        System.out.println("[INFO] " + java.time.LocalDateTime.now() + ": " + info);

    }

}

1. LoggerTest Class

public class LoggerTest {

    public static void main(String[] args) {

        System.out.println("=== Testing Singleton Pattern ===");

        Logger logger1 = Logger.getInstance();

        logger1.log("First logger instance created");

        Logger logger2 = Logger.getInstance();

        logger2.logInfo("Second logger instance requested");

        System.out.println("Are both instances the same? " + (logger1 == logger2));

        System.out.println("logger1 hashCode: " + logger1.hashCode());

        System.out.println("logger2 hashCode: " + logger2.hashCode());

        logger1.log("Message from logger1");

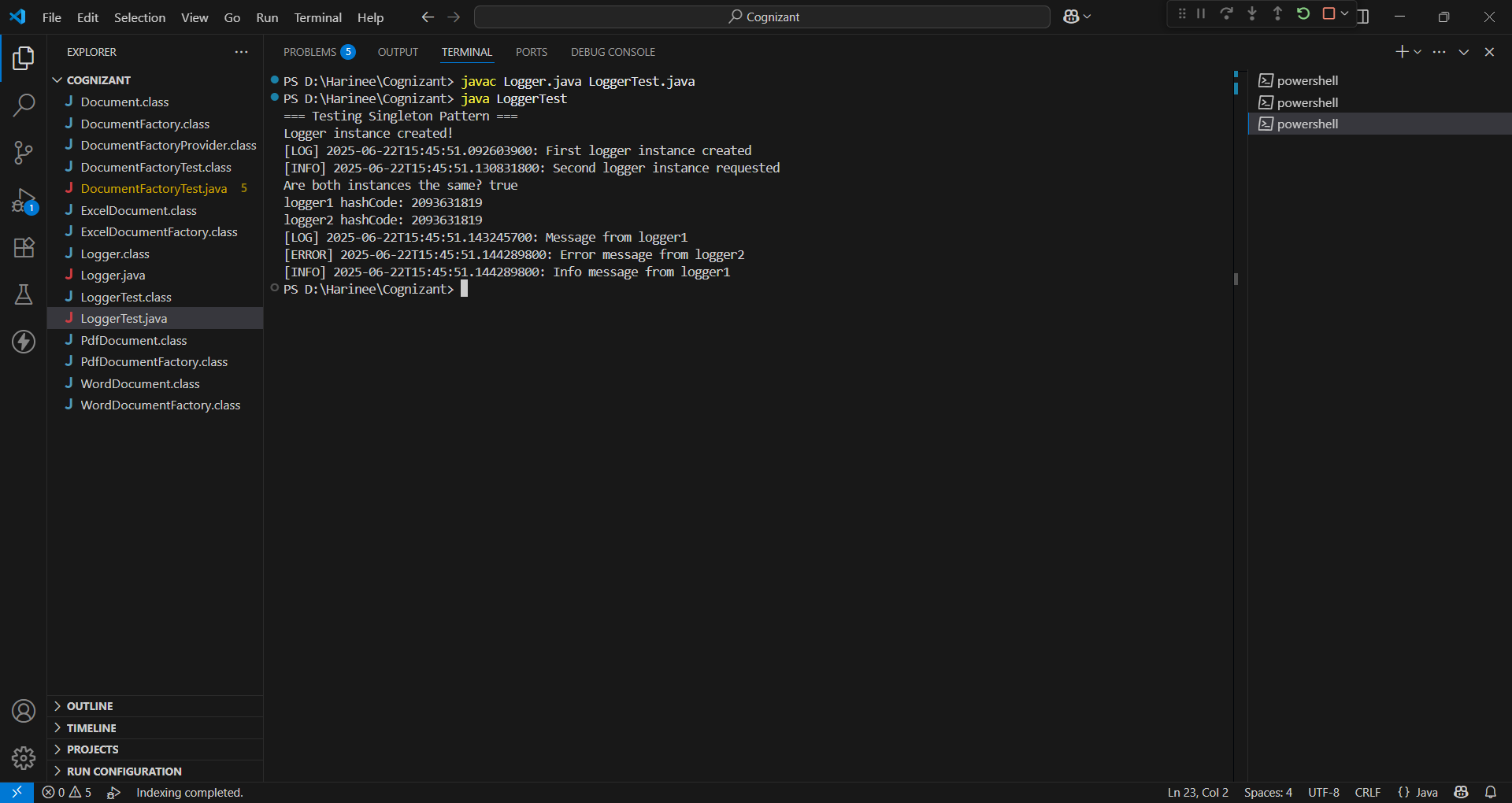
        logger2.logError("Error message from logger2");

        logger1.logInfo("Info message from logger1");

    }

}

**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:**

interface Document {

void create();

void open();

void save();

void close();

void displayInfo();

}

class WordDocument implements Document {

private String name;

public WordDocument(String name) {

this.name = name;

}

@Override

public void create() {

System.out.println("WordDocument: Creating " + name);

}

@Override

public void open() {

System.out.println("WordDocument: Opening " + name);

}

@Override

public void save() {

System.out.println("WordDocument: Saving " + name);

}

@Override

public void close() {

System.out.println("WordDocument: Closing " + name);

}

@Override

public void displayInfo() {

System.out.println("Document Type: Word Document, Name: " + name);

}

}

class PdfDocument implements Document {

private String name;

public PdfDocument(String name) {

this.name = name;

}

@Override

public void create() {

System.out.println("PdfDocument: Creating " + name);

}

@Override

public void open() {

System.out.println("PdfDocument: Opening " + name);

}

@Override

public void save() {

System.out.println("PdfDocument: Saving " + name);

}

@Override

public void close() {

System.out.println("PdfDocument: Closing " + name);

}

@Override

public void displayInfo() {

System.out.println("Document Type: PDF Document, Name: " + name);

}

}

class ExcelDocument implements Document {

private String name;

public ExcelDocument(String name) {

this.name = name;

}

@Override

public void create() {

System.out.println("ExcelDocument: Creating " + name);

}

@Override

public void open() {

System.out.println("ExcelDocument: Opening " + name);

}

@Override

public void save() {

System.out.println("ExcelDocument: Saving " + name);

}

@Override

public void close() {

System.out.println("ExcelDocument: Closing " + name);

}

@Override

public void displayInfo() {

System.out.println("Document Type: Excel Document, Name: " + name);

}

}

abstract class DocumentFactory {

public abstract Document createDocument(String name);

public Document processDocument(String name) {

Document document = createDocument(name);

document.create();

document.open();

return document;

}

}

class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument(String name) {

System.out.println("WordDocumentFactory: Creating Word document");

return new WordDocument(name);

}

}

class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument(String name) {

System.out.println("PdfDocumentFactory: Creating PDF document");

return new PdfDocument(name);

}

}

class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument(String name) {

System.out.println("ExcelDocumentFactory: Creating Excel document");

return new ExcelDocument(name);

}

}

class DocumentFactoryProvider {

public static DocumentFactory getFactory(String documentType) {

switch (documentType.toLowerCase()) {

case "word":

case "doc":

case "docx":

return new WordDocumentFactory();

case "pdf":

return new PdfDocumentFactory();

case "excel":

case "xls":

case "xlsx":

return new ExcelDocumentFactory();

default:

throw new IllegalArgumentException("Unknown document type: " + documentType);

}

}

}

public class DocumentFactoryTest {

public static void main(String[] args) {

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

testSpecificFactories();

System.out.println("\n" + "=".repeat(50) + "\n");

testFactoryProvider();

System.out.println("\n" + "=".repeat(50) + "\n");

simulateDocumentManagement();

}

private static void testSpecificFactories() {

System.out.println("=== Test 1: Using Specific Factories ===\n");

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.processDocument("Report.docx");

wordDoc.displayInfo();

wordDoc.save();

wordDoc.close();

System.out.println();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.processDocument("Manual.pdf");

pdfDoc.displayInfo();

pdfDoc.save();

pdfDoc.close();

System.out.println();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.processDocument("Budget.xlsx");

excelDoc.displayInfo();

excelDoc.save();

excelDoc.close();

}

private static void testFactoryProvider() {

System.out.println("=== Test 2: Using Factory Provider ===\n");

String[] documentTypes = {"word", "pdf", "excel"};

String[] documentNames = {"Proposal.docx", "Invoice.pdf", "Sales.xlsx"};

for (int i = 0; i < documentTypes.length; i++) {

try {

DocumentFactory factory = DocumentFactoryProvider.getFactory(documentTypes[i]);

Document document = factory.createDocument(documentNames[i]);

document.displayInfo();

document.create();

document.open();

document.save();

document.close();

System.out.println();

} catch (IllegalArgumentException e) {

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

}

}

}

private static void simulateDocumentManagement() {

System.out.println("=== Test 3: Document Management Simulation ===\n");

String[][] documents = {

{"word", "Meeting\_Notes.docx"},

{"pdf", "Contract.pdf"},

{"excel", "Financial\_Report.xlsx"},

{"word", "Project\_Plan.docx"},

{"pdf", "User\_Guide.pdf"}

};

System.out.println("Processing multiple documents in document management system:\n");

for (String[] docInfo : documents) {

String type = docInfo[0];

String name = docInfo[1];

try {

System.out.println("Processing: " + name);

DocumentFactory factory = DocumentFactoryProvider.getFactory(type);

Document document = factory.processDocument(name);

document.displayInfo();

document.save();

document.close();

System.out.println("✓ " + name + " processed successfully\n");

} catch (Exception e) {

System.out.println("✗ Failed to process " + name + ": " + e.getMessage() + "\n");

}

}

System.out.println("Testing with invalid document type:");

try {

DocumentFactory factory = DocumentFactoryProvider.getFactory("powerpoint");

Document document = factory.createDocument("Presentation.pptx");

} catch (IllegalArgumentException e) {

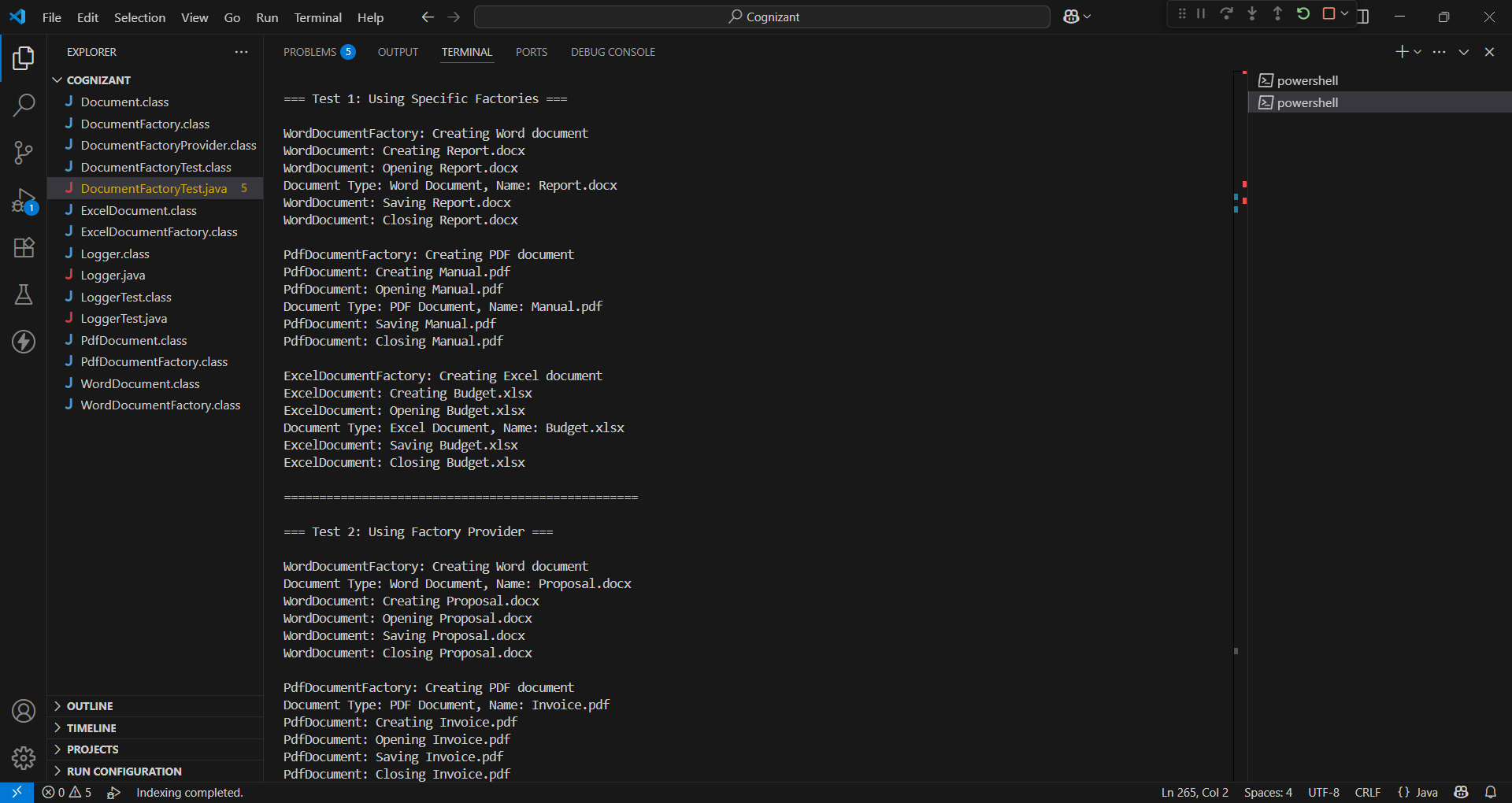
System.out.println("✓ Error handling works: " + e.getMessage());

}

}

}

**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.