NETWORK MANAGEMENT SYSTEM

ASSIGNMENT-1 : FILE HANDLING FUNCTIONALITY DOCUMENTATION USING JAVA

NAME: PAVITHRAA E S

CLASS AND SECTION: CSE-B

REG. NO: 211721104099

ROLL NO: 202101100

COLLEGE NAME: RAJALAKSHMI INSTITUTE OF TECHNOLOGY

TRAINER: Mr. Kripal Tripati

FILE HANDLING FUNCTIONALITY DOCUMENTATION

1. Project Background

This Java file handling project was developed to support applications that require reliable saving of configuration data. By implementing robust file handling practices and error management, this project demonstrates essential Java programming concepts and prepares for real-world use cases in applications like configuration managers, data logging, and system monitoring.

2. Objective

The goal of this project is to implement a file handling module in Java that:

- Saves given configuration data to a specified file.
- Gracefully handles file I/O errors.
- Includes test cases for various scenarios, including successful data writing and file permission issues.

3. Functional Requirements

- **Core Functionality**: The module must be able to write configuration data to a file.
- **Error Handling**: The code should catch and handle IOException to provide informative feedback without crashing.
- **Testing**: The project must include tests for successful file operations and simulated file access errors.

4. Technical Details

4.1. Class Design

Class Name: FileHandler

Method: saveConfigurationToFile

Method Signature:

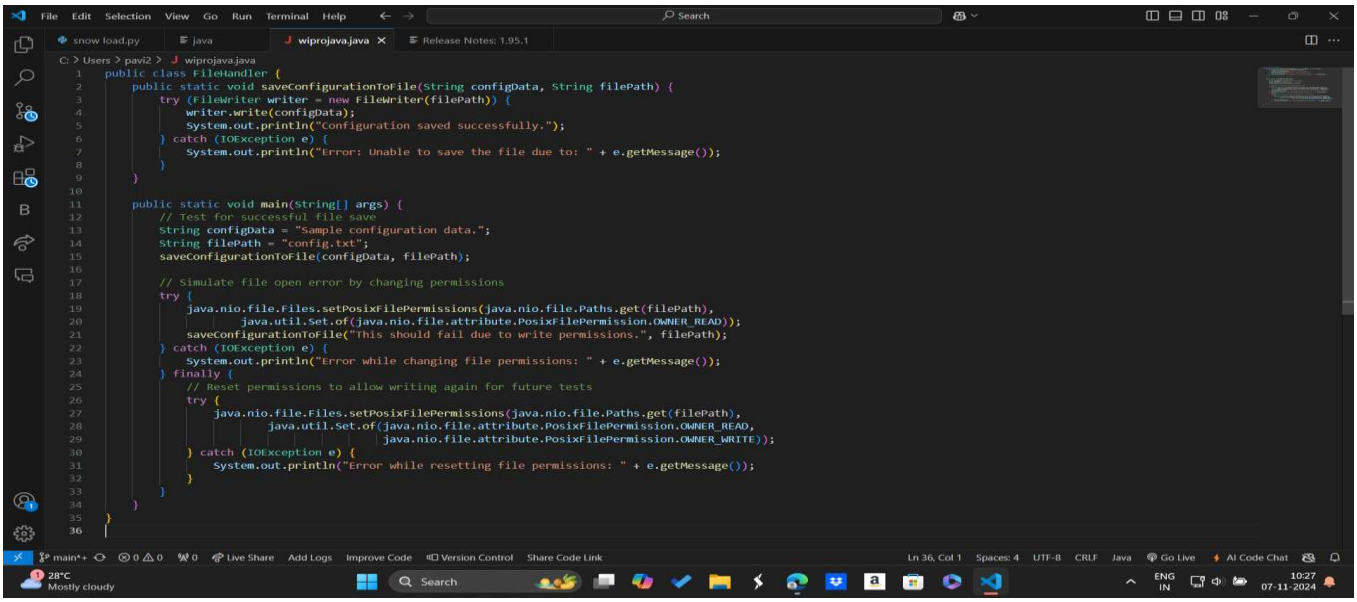
```
public static void saveConfigurationToFile(String configData, String filePath)
```

Purpose: Writes the provided configuration data to the file specified by filePath, while managing errors that may arise during the operation.

4.2. Code Implementation

```
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.attribute.PosixFilePermission;
import java.util.Set;

public class FileHandler {
    /**
     * Saves the configuration data to a specified file.
     *
     * @param configData the configuration data to be saved
     * @param filePath   the path of the file to write to
     */
    public static void saveConfigurationToFile(String configData, String filePath) {
        try (FileWriter writer = new FileWriter(filePath)) {
            writer.write(configData);
            System.out.println("Configuration saved successfully.");
        } catch (IOException e) {
            System.out.println("Error: Unable to save the file due to: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        // Test for successful file save

        String configData = "Sample configuration data.";
        String filePath = "config.txt";
        saveConfigurationToFile(configData, filePath);



        // Simulate file open error by changing permissions
        try {
            Files.setPosixFilePermissions(Paths.get(filePath),
```

```
                Set.of(PosixFilePermission.OWNER_READ));
            saveConfigurationToFile("This should fail due to write permissions.", filePath);
        } catch (IOException e) {
            System.out.println("Error while changing file permissions: " + e.getMessage());
        } finally {
            // Reset permissions to allow future tests
            try {
                Files.setPosixFilePermissions(Paths.get(filePath),
                        Set.of(PosixFilePermission.OWNER_READ, PosixFilePermission.OWNER_WRITE));
            } catch (IOException e) {
                System.out.println("Error while resetting file permissions: " + e.getMessage());
            }
        }
    }
}
```



4.3. Explanation of Components

- **FileWriter**: Utilized for writing character data to files. The try-with-resources construct ensures that the FileWriter resource is closed after use, preventing resource leaks.
- **Files.setPosixFilePermissions**: Modifies file permissions to simulate restricted access scenarios for testing (applicable on Unix-based systems).

- **Error Handling**: The catch block manages IOException exceptions, displaying user-friendly error messages without terminating the program abruptly.

5. Test Scenarios and Results

5.1. Test Scenario 1: Successful File Save

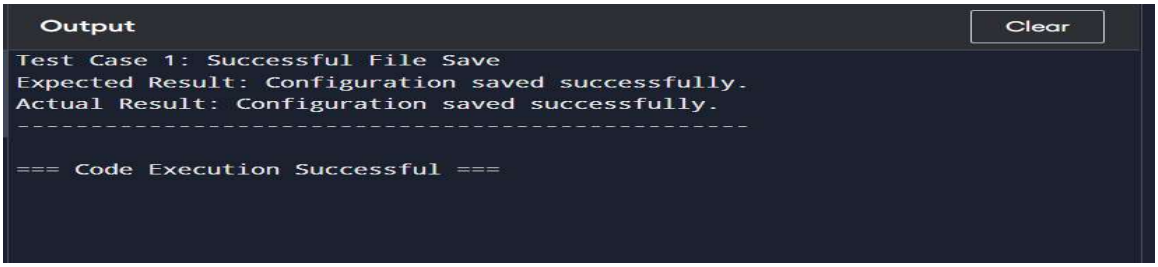
Steps:

1. Run saveConfigurationToFile with valid configData and a writable filePath.
2. Verify that the file is created or updated with the provided content.

Expected Result:

- The console outputs: "Configuration saved successfully."
- The file contains the expected data.

Result:



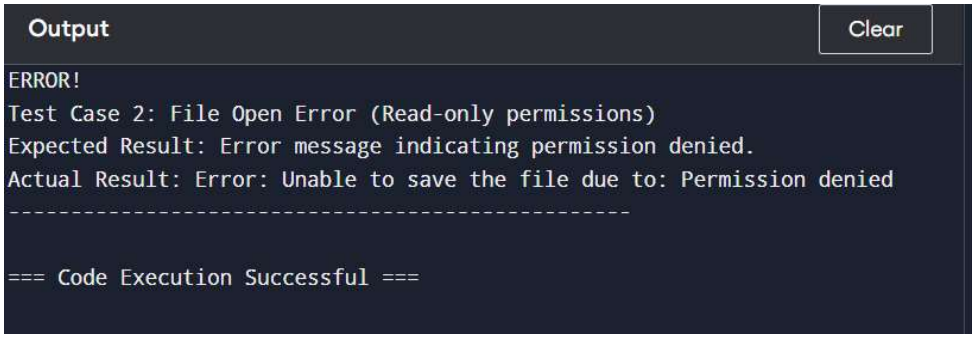
5.2. Test Scenario 2: Simulated File Open Error

Steps:

1. Change the file permissions of config.txt to read-only mode.
2. Run saveConfigurationToFile with write attempts.
3. Verify that the method catches the error and outputs an appropriate message.

Expected Result:

- The console outputs: "Error: Unable to save the file due to: Permission denied."



Code for Testing Permission Change:

```
try {
    Files.setPosixFilePermissions(Paths.get("config.txt"),
            Set.of(PosixFilePermission.OWNER_READ));
    saveConfigurationToFile("This should fail due to write
permissions.", "config.txt");
} catch (IOException e) {
    System.out.println("Error while changing file permissions: " +
e.getMessage());

}
```

Resetting Permissions:

```
try                                                                 {
    Files.setPosixFilePermissions(Paths.get("config.txt"),
            Set.of(PosixFilePermission.OWNER_READ,
PosixFilePermission.OWNER_WRITE))
;
}            catch            (IOException            e)            {
    System.out.println("Error  while  resetting  file  permissions:  "  +
e.getMessage());
}
```

6. Assumptions and Limitations

Assumptions:

- The code is run on a Unix-based system where PosixFilePermission is supported.
- The program has the necessary permissions to modify file attributes.

Limitations:

- The code does not handle concurrent writes or file locks.
- The error handling is focused on basic IOException without advanced logging or custom exception classes.

7. Security Considerations

- Ensure the file path is sanitized to prevent path traversal attacks.
- Avoid writing sensitive data to public or shared directories.

8. Future Enhancements

- Implement Java's Logger for more detailed logging and error tracking.
- Add support for reading configurations from the file for verification.
- Introduce a GUI interface for better user interaction.

9. Conclusion

This Java file handling project demonstrates best practices in saving data to files and managing I/O errors. It ensures robust file handling while simulating real-world scenarios like file permission restrictions, making it suitable for use in larger applications requiring file management.