**Exercise 1: Basic File Reading and Writing**

**Objectives:**

* Learn how to read from and write to files using Java.
* Understand basic file operations.

**Business Scenario:**

You are developing a simple note-taking application where users can save their notes to a file and read them back when needed.

**Tasks:**

1. **Setup:**
   * Create a new Java project named **NoteTakingApp**.
2. **Writing to a File:**
   * Create a class named **FileWriterDemo**.
   * Write a method **saveNoteToFile(String fileName, String note)** that:
     + Takes a file name and a note as parameters.
     + Uses **FileWriter** and **BufferedWriter** to write the note to the specified file.
3. **Reading from a File:**
   * Create a class named **FileReaderDemo**.
   * Write a method **readNoteFromFile(String fileName)** that:
     + Takes a file name as a parameter.
     + Uses **FileReader** and **BufferedReader** to read the content of the file and print it to the console.
4. **User Interaction:**
   * Create a class named **NoteApp**.
   * Write a main method that:
     + Prompts the user to enter a note.
     + Saves the note to a file using **FileWriterDemo**.
     + Reads the note back from the file using **FileReaderDemo** and displays it.

**Exercise 2: Advanced File Handling with Error Handling**

**Objectives:**

* Handle errors and exceptions during file operations.
* Implement robust file handling in Java.

**Business Scenario:**

You are developing a log management system for an application. The system should be able to write log messages to a file and read log messages from the file, with proper error handling for file operations.

**Tasks:**

1. **Setup:**
   * Create a new Java project named **LogManagementSystem**.
2. **Writing Logs to a File:**
   * Create a class named **LogWriter**.
   * Write a method **writeLog(String fileName, String logMessage)** that:
     + Takes a file name and a log message as parameters.
     + Uses **FileWriter** and **BufferedWriter** to append the log message to the specified file.
     + Implements error handling using try-catch blocks to handle **IOException**.
3. **Reading Logs from a File:**
   * Create a class named **LogReader**.
   * Write a method **readLogs(String fileName)** that:
     + Takes a file name as a parameter.
     + Uses **FileReader** and **BufferedReader** to read the content of the file line by line.
     + Implements error handling using **try-catch** blocks to handle **FileNotFoundException** and **IOException**.
4. **User Interaction:**
   * Create a class named **LogApp**.
   * Write a main method that:
     + Writes multiple log messages to a file using **LogWriter**.
     + Reads the log messages from the file using **LogReader** and displays them.
     + Demonstrates the error handling by attempting to read from a non-existent file.

**Exercise 3: File Handling with Serialization**

**Objectives:**

* Learn how to serialize and deserialize objects to and from files.
* Understand the use of **ObjectInputStream** and **ObjectOutputStream**.

**Business Scenario:**

You are developing a contact management system where contact details can be saved to a file and read back using serialization.

**Tasks:**

1. **Setup:**
   * Create a new Java project named **ContactManagementSystem**.
2. **Defining the Contact Class:**
   * Create a class named **Contact** that implements **Serializable**.
   * Define attributes for contact details like **name**, **phone**, and **email**.
   * Provide a constructor and appropriate getter and setter methods.
3. **Serializing Contacts:**
   * Create a class named ContactWriter.
   * Write a method **saveContact(String fileName, Contact contact)** that:
     + Takes a file name and a **Contact** object as parameters.
     + Uses **ObjectOutputStream** to write the **Contact** object to the specified file.
     + Implements error handling using **try-catch** blocks to handle **IOException**.
4. **Deserializing Contacts:**
   * Create a class named **ContactReader**.
   * Write a method **readContact(String fileName)** that:
     + Takes a file name as a parameter.
     + Uses **ObjectInputStream** to read the **Contact** object from the file.
     + Implements error handling using **try-catch** blocks to handle **FileNotFoundException**, **IOException**, and **ClassNotFoundException**.
5. **User Interaction:**
   * Create a class named **ContactApp**.
   * Write a main method that:
     + Creates a **Contact** object and saves it to a file using **ContactWriter**.
     + Reads the Contact object back from the file using **ContactReader** and displays the contact details.
     + Demonstrates the error handling by attempting to read from a non-existent file and handling class casting issues.

**Exercise 4: Processing Large Files**

**Objectives:**

* Handle large files efficiently using Java.
* Use buffered streams for reading and writing large files.

**Business Scenario:**

You are developing a system to process large data files containing sales records. The system should be able to read large files efficiently, process the data, and write the results to a new file.

**Tasks:**

1. **Setup:**
   * Create a new Java project named **LargeFileProcessor**.
2. **Reading Large Files:**
   * Create a class named **LargeFileReader**.
   * Write a method **readLargeFile(String inputFileName)** that:
     + Takes an input file name as a parameter.
     + Uses **BufferedReader** to read the file line by line.
     + Processes each line (e.g., parse and print sales records).
     + Implements error handling using **try-catch** blocks to handle **IOException**.
3. **Writing Processed Data:**
   * Create a class named **LargeFileWriter**.
   * Write a method **writeProcessedData(String outputFileName, List<String> processedData)** that:
     + Takes an output file name and a list of processed data as parameters.
     + Uses **BufferedWriter** to write the processed data to the specified file.
     + Implements error handling using try-catch blocks to handle **IOException**.
4. **User Interaction:**
   * Create a class named **FileProcessorApp**.
   * Write a main method that:
     + Reads a large sales data file using LargeFileReader.
     + Processes the data (e.g., filtering records, calculating totals).
     + Writes the processed data to a new file using LargeFileWriter.
     + Demonstrates the error handling by attempting to read and write to non-existent files or restricted directories.

**Exercise 5: File Handling with NIO Package**

**Objectives:**

* Learn how to use Java NIO package for file operations.
* Understand the differences between java.io and java.nio.

**Business Scenario:**

You are tasked with creating a backup system for important files. The system should copy files from a source directory to a backup directory using Java NIO for efficient file operations.

**Tasks:**

1. **Setup:**
   * Create a new Java project named **BackupSystem**.
2. **Copying Files:**
   * Create a class named **FileCopy**.
   * Write a method **copyFile(Path source, Path target)** that:
     + Takes source and target paths as parameters.
     + Uses **Files.copy** from the **NIO** package to copy the file from source to target.
     + Implements error handling using **try-catch** blocks to handle **IOException**.
3. **Copying Directories:**
   * Create a class named **DirectoryCopy**.
   * Write a method **copyDirectory(Path sourceDir, Path targetDir)** that:
     + Takes source and target directory paths as parameters.
     + Recursively copies all files and subdirectories from the source to the target directory using **Files.walk** and **Files.copy**.
     + Implements error handling using **try-catch** blocks to handle IOException.
4. **User Interaction:**
   * Create a class named **BackupApp**.
   * Write a main method that:
     + Prompts the user for the source and target directories.
     + Uses **DirectoryCopy** to backup all files and directories from the source to the target directory.
     + Demonstrates the error handling by attempting to copy to a read-only directory or a directory with insufficient space.