Exercise – File Handling

Objective

The objective of this exercise is to look at reading and writing operations on files. By the end of this we should be comfortable with some basic file operations in Java

A lot of this exercise will be down to you to look up how to do the specific file operations, use the Java API or the oracle tutorials if you get stuck as well class notes and demos.

Overview

Part 1: Read in a file

There are a number of ways to read in files using Java. We are going to read in a file line by line and output the contents. File input and output operations can cause a lot of potential exceptions so we want to be as specific as possible when catching exceptions, don't just use Exception!

1. Create a class called MyFileReader with a main method inside it
2. Create a basic empty constructor for the class
3. Implement an readAndPrint() method
    a. We want to use both a FileReader and a BufferedFileReader object in this method. Look them both up in the API and get familiar with some of the options. FileReaders allow us to read in a file character by character, which isn't always convenient. The BufferedFileReader allows us to read in the file line by line instead.
    b. Create a FileReader and a BufferedReader in the method
        i. You will need to create a new File object with the path to the file. This is provided to the File object as a String. To define the file you can either give the relative or absolute path.

            Beware: if you use the windows slash divider for directories "\" then you need to add a second one "\\" as the slash is used in strings as an escape character. The first slash escapes the second one, and so it is processed in the string as expected.

            It is also possible to use the forward slash, "/" on windows operating systems, despite the fact that windows usually insists on using "\".
    c. Read in each line from the file and print it out using System.out.println()
        i. Refer to the slides to see an example of this, or try and work it out using the API methods!
    d. Two exceptions need to be caught for the code to work, a FileNotFoundException and the generic IOException. Exceptions are

handled in the order they are caught, so make sure the most specific one is first in the list. Add a try/catch block for the FileNotFound and IO exceptions.

It is harder to test if we can catch the IOException, but we can test the FileNotFoundException easily (and in fact, you probably will constantly). Print out which exception you are in and test your code. Try and get it both to read in your file correctly and also to throw the FileNotFoundException.

e. It is not best practice to open resources such as files without closing them when finished. To ensure that this happens add a finally block to your code which calls the close() method on the buffered writer.

   You will need to handle any exceptions in this code as well! This can lead to very confusing to follow code, which we can deal with better by throwing exceptions out of the method, we'll have a look at doing that later!

Part 2: Writing to a file

Writing to a file is another common file operation that is useful to know. To do this we are going to look at the FileWriter and BufferedWriter classes.

4. Look up the writer classes in the API and oracle tutorials. (Use google if you can't remember or don't know the addresses). Try and get an understanding about how to write to a file.
5. Create the method writeMyFile() in the MyFileReader class.
6. In the method create a new BufferedWriter that uses a new FileWriter as a parameter. The File passed to the FileWriter can be anything you like. I chose "Output.txt"
7. Create a for loop that goes from 0 to 100, on each loop we want to write something to the file, we can do this using the append() method in the BufferedWriter class. This will write the contents of the character sequence (or string) you pass it to the file.

   ```
   bw.append("This is line: " + x + "\n");
   ```

   the "\n" on the end adds the new line character to the line in the file. If you open the file using the standard windows notepad you will see no difference, but if you use a text editor that is able to deal with all types of line endings then each appended piece of text will appear on a new line.

   The windows specific line end command is "\r\n" which will ensure that even notepad can open it correctly.

8. Add the try/catch block around the code you've written, there is only one type of exception thrown here, the IOException. Put a System.out.println() statement so you can see when this is thrown.

9. Finally, we need a finally block in our code to close the file. This will flush the output ensuring that everything is saved to the file before closing it. Write the finally block in your code
10. To test this we call writeMyFile from your main method. Depending on where you told it to write the file to, you should be able to find the file on your system. If no path was given, and just the filename used then this will be in the base directory of your exercise folder in the workspace.

Part 3: BadLineException

In this part we are going to read in a file expecting a specific format:

This is line: 0

This is line: 1

This is line: 2

… etc

If a line is read in that doesn't conform to this pattern then we are going to throw a BadLineException.

11. First we need to write the BadLineException. Create a class called BadLineException in the same package as your other class.
    a. BadLineException should extend Exception
    b. We want to store the line of text that was read and considered bad, so create a field to store this text.
    c. Write a getter for the line field
    d. Write a constructor for the class, it should take in a String message and the line of text that was read when the exception was thrown. The message should be passed to the super constructor
12. In MyFileReader create a new method called readMyFile
    a. Create a BufferedReader and FileReader which open file you wrote in Part 2.
    b. Read in each line and check to see if the line starts with "This is line:", if it does then print it out, if it doesn't then throw a new BadLineException.

       (Look up the startsWith() method in the String class for help using it.)
    c. Initialise the BadLineException with a message that says what the problem was and the line that we were on.
13. Now we need to handle those exceptions. Instead of using a try/catch block in the method add "throws" to the method signature for each of the exceptions raised.

14. Just because we are throwing the exceptions up the call stack to be dealt with we still need to add a try/finally block to our code. The finally block should close the file. This will be called before the exception is thrown.
15. In the main method call your readMyFile method and catch any exceptions that could be thrown.
    a. Print out the message stored in each exception and for the BadLineException print out the line which caused the problem.
16. Initially, the file should be read in successfully. Edit your file in notepad (or eclipse) to contain a line which isn't formatted correctly to see if you can get your exception to be thrown.

Glossary of key terms

Exception

It is a runtime error

StringBuffer

It is a mutable class object, means the objects of StringBuffer can be modified after creation

BufferedReader

It is one of the character stream classes from Java API which is can read the data line-by-line

FileReader

It is one of the character stream classes from Java API which is can refer to the physical location of a file. It is used for implementing reading activities

BufferedWriter

This class is used to write the content into a character stream

FileWriter

It is one of the character stream classes from Java API which is can refer to the physical location of a file. It is used for implements writing operations

IOException

It is an exception class which is associated with I/O activities