

# File I/O - Writing - Individual Exercises

---

The purpose of this exercise is to provide you with the opportunity to create command line applications that are able to create and write to files.

## Learning Objectives

After completing this exercise, students will understand:

- How to provide arguments to a command line application from the console.
- How to programmatically create and write to text files.
- How to read, interpret, and resolve errors related to file I/O.

## Evaluation Criteria & Functional Requirements

Your code will be evaluated based on the following criteria:

- The project must not have any build errors.
- The expected results are output to the files.
- Paths to files are not hard coded. (ie. We should not need to change your code to run the application.)

### FizzWriter

Create a program to write out the result of FizzBuzz (1 to 300) to a file called **FizzBuzz.txt**. The file should be written out to the same directory as this README file.

- If the number is divisible by 3 or contains a 3, print "Fizz"
- If the number is divisible by 5 or contains a 5, print "Buzz"
- If the number is divisible by 3 and 5, print "FizzBuzz"
- Otherwise print the number.

To run the application, the following command should be run at the root of this directory:

```
mvn exec:java -Dexec.mainClass="com.techelevator.FizzWriter"
```

Expected Output:

```
FizzBuzz.txt has been created.
```

Things to keep in mind:

- When you run the command **wc -l FizzBuzz.txt**, the result should be **300 300 FizzBuzz.txt**
- Use the **less** command or your favorite text editor to verify the contents of the file are what you expect them to be. **The contents of the file will be evaluated based on the requirements specified above.**
- A new file should be created each time the application is run.

- There is not any user interaction in this application. (The application should run and terminate. You should not need to press a key to stop the application.)

## File Splitter (Challenge)

Develop an application that takes a significantly large input file and splits it into smaller file chunks. These types of files use to be used quite common back in the earlier days of computing when disks such as floppies were much smaller and couldn't hold a larger program on their own.

In order to determine how many files need to be produced, ask the user for the maximum amount of lines that should appear in each output file.

To run the application, the following command should be run at the root of this directory:

```
mvn exec:java -Dexec.mainClass="com.techelevator.FileSplitter"
```

Sample Input/Output:

```
Where is the input file (please include the path to the file)? [path-to-file]/input.txt
How many lines of text (max) should there be in the split files? 3
The input file has 50 lines of text.
```

Each file that is created should have a sequential number assigned to it.

For a 50 line input file "input.txt" this will produce 17 output files.

**\*\*GENERATING OUTPUT\*\***

```
Generating input-1.txt
Generating input-2.txt
Generating input-3.txt
Generating input-4.txt
Generating input-5.txt
Generating input-6.txt
Generating input-7.txt
Generating input-8.txt
Generating input-9.txt
Generating input-10.txt
Generating input-11.txt
Generating input-12.txt
Generating input-13.txt
Generating input-14.txt
Generating input-15.txt
Generating input-16.txt
Generating input-17.txt
```

Things to keep in mind:

- When you run the command `find . -name '[your-input-file-name]*.[your-input-file-extension]' | xargs wc -l`, the result should list each of the files that match that name, as well as the line counts for each of the files. For instance, given the example above, the result of the command would be:

```
3 ./input-4.txt
3 ./input-5.txt
3 ./input-7.txt
3 ./input-6.txt
3 ./input-2.txt
3 ./input-3.txt
3 ./input-1.txt
3 ./input-12.txt
3 ./input-13.txt
3 ./input-11.txt
3 ./input-10.txt
3 ./input-14.txt
3 ./input-15.txt
2 ./input-17.txt
3 ./input-8.txt
3 ./input-9.txt
3 ./input-16.txt
50 total
```

- Use the `less` command or your favorite text editor to verify the contents of the file are what you expect them to be.
- The input file name should be the prefix (the first part of the file name) followed by a dash (-), then the number of the current file, and finally ending with the file extension of the input file. For instance, if the name of the file was `big-old-file.md`, the file names should be `big-old-file-1.md`, `big-old-file-2.md`, etc.
- Output files will be written to the directory the input file is in.
- The application should run and terminate. You should not need to press a key to stop the application.

## Getting Started

- Import the `file-io-part2-exercises` project into Eclipse.
- Open the java file for the application you are working on. The files are located in the `src/main/java/com/techelevator` directory.
- Provide enough code to get the program started.
- Verify your work on the command line and by running the `wc` command specified in the requirements.
- Repeat until the features have been implemented as outlined in the functional requirements.

## Tips and Tricks

- The `wc` command displays the number of lines, words, and bytes contained in each input file, or standard input (if no file is specified) to the standard output. A line is defined as a string of characters delimited by a newline character. Characters beyond the final newline character will not be included in the line count. The `-l` flag is used to determine the number of lines in the file, and the `-w` flag is used to determine the

number of words in the file. For more information about what `wc` provides, try typing `man wc`. (NOTE: `man` can be used on the terminal with any command to get details about the command and how it works.)

- The Java [File class](#) is a helpful resource for learning about how to interact with files and directories programmatically.
-