File I/O (Writing)

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

Learning objectives

After completing this exercise, you'll understand:

- How to programmatically create and write to text files.
- How to read, interpret, and resolve errors related to file I/O.

Evaluation criteria and functional requirements

Your code will be evaluated based on the following criteria:

- The project must not have any build errors.
- The expected results are sent to a file.
- Paths to files aren't hard-coded—that is, the code shouldn't need to be changed to run the application for a different file.
- The unit tests pass as expected.
    - Note: Tests are provided for the FindAndReplace and FizzWriter exercises only.

Part One: Create a find and replace program

In this exercise, you'll write a program that finds all occurrences of a user-specified word in a text file and replaces it with another user-specified word. You'll write the text with the replaced word to a different text file.

The program must prompt the user for the following values:

- The search word
- The word to replace the search word with
- The source file
    - *This must be an existing file. If the user enters an invalid source file, the program indicates this to the user and exits.*
- The destination file
    - *The program creates a copy of the source file with the requested replacements at this location. If the file already exists, it must be overwritten. If the user enters an invalid destination file, the program indicates this to the user and exits.*

> Note: This is a case-sensitive search. If your search word is `Bacon`, then `bacon` shouldn't be replaced.

Here's an example of what your application could look like:

```
What is the search word?
bacon
What is the replacement word?
ham
```

```
What is the source file?
[path-to-source-file]
What is the destination file?
[path-to-destination-file]
```

Examples

The examples below demonstrate the requirements of this exercise. The tests for this exercise test each scenario below.

The following examples use a hypothetical text file with the following contents:

```
apple Bacon coconut bacon
bread bacon Apple cherry
```

Multiple occurrences

For the search word `bacon` and the replacement word `ham`, the contents of the destination file would be:

```
apple Bacon coconut ham
bread ham Apple cherry
```

Remember that the search is case-sensitive, which is why the capitalized `Bacon` isn't replaced.

Single occurrence

For the search word `Apple` and the replacement word `carrot`, the contents of the destination file would be:

```
apple Bacon coconut bacon
bread bacon carrot cherry
```

Remember that the search is case-sensitive, which is why only the capitalized `Apple` is replaced.

No occurrences

For the search word `honey` and the replacement word `ketchup`, the contents of the destination file would be the same as the source file:

```
apple Bacon coconut bacon
bread bacon Apple cherry
```

Tests

The tests for this exercise are in the file `src/test/java/com/techelevator/FindAndReplaceTests.java`. All tests must pass to complete this exercise.

Part Two: Create a FizzWriter program

Create a program that writes out the result of FizzBuzz (1 to 300) to a file.

> Note: this version of FizzBuzz has additional requirements.

- 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.

The program must prompt the user for the following values:

- The destination file
  - *If the file already exists, it must be overwritten. If the user enters an invalid destination file, the program indicates this and exits.*

The tests for this exercise are in the file `src/test/java/com/techelevator/FizzWriterTests.java`. All tests must pass to complete this exercise.

File splitter (Challenge)

Create an application that takes a significantly large input file and splits it into smaller file chunks. These types of files were common back when floppy disks were popular and couldn't hold a larger program on their own.

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

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 must have a sequential number assigned to it.

For a 50 line input file "input.txt", this produces 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
```

Here are a few 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 lists each file that matches that name and the line counts for each of the files. For instance, given the previous example, 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 filename must be the prefix—the first part of the filename—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 is `big-old-file.md`, the filenames must be `big-old-file-1.md`, `big-old-file-2.md`, etc.
- Output files are written to the directory the input file is in.
- The application must run and exit. You shouldn't need to press a key to stop the application.

Tips and tricks

Use the `wc` command to verify your work

Verify your work on the command line by running the `wc` command specified in the requirements. 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 aren't included in the line count. The `-l` flag determines the number of lines in the file, and the `-w` flag determines the number of words in the file.

For more information about what `wc` provides, try typing `man wc`. You can use `man` in the terminal with any command to learn more about the command and how it works.

Learn more about the File class

The Java File class is a helpful resource for learning about how to interact with files and directories programmatically.
