Individual Project: File Processor Deliverable 1

Note: In the version of the assignment we initially released, there was a mistake in the expected results of some of the provided examples and test cases. If you downloaded the assignment and the test cases before Sunday, July 5, at 10:05pm Eastern Time, please redownload them. Sorry for the inconvenience.

Project Goals

In this project, you will be developing a simple Java application (`fileprocessor`) using an agile, test-driven process involving multiple deliverables. While you will receive one grade for the entire project, each deliverable must be completed by its own due date, and all deliverables will contribute to the overall project grade.

Specification of the `fileprocessor` Utility

Program `fileprocessor` is a simple command-line utility written in Java with the following specification:

- NAME
  `fileprocessor` - utility for analyzing a text file
- SYNOPSIS
  `fileprocessor OPT <filename>`

  where `OPT` can be **zero or more** of
  - `-s`
  - `(-r|-k) <string s>`
  - `-t [integer n]`
  - `-l`
- COMMAND-LINE ARGUMENTS AND OPTIONS

  `<filename>`: file to be analyzed

  `-s`: if specified, `fileprocessor` will sort the lines in the file alphanumerically, (1) ignoring (but keeping) non-alphanumeric characters, (2) with numbers preceding letters, and (3) with capital letters preceding lowercase letters (i.e., 0-9, A-Z, a-z). Any line containing only non-alphanumeric characters is removed.

<required>

`(-r|-k)` `<string s>`: if specified, `fileprocessor` will remove (`-r`) or keep (`-k`) only lines in the file which contain string s (case sensitive). Options `-r` and `-k` are mutually exclusive.

<optional>:b

`-t` `[integer n]`: if specified, `fileprocessor` will keep only the first n characters of each line. Value n must be a positive integer. If n is omitted, its default value is 1.

`-l`: if specified, `fileprocessor` will add the line number at the beginning of each line, followed by a space and with the first line having line number 1.

If **no** OPT flag is specified, `fileprocessor` will leave the file unchanged.

- NOTES

  While the last command-line parameter provided is always treated as the filename, OPT flags can be provided in any order and will be applied in the order in which they are listed above (i.e., -s first and -l last).

- EXAMPLES OF USAGE

Example 1:

```
fileprocessor -s -l file1.txt
```

sort
add line numbers

File content before:

```
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello
```

File content after:

```
1 --’’--911
2 #%Albert
3 Beatrice
4 Hello
5 albert
6 hello
```

Example 2:

```
fileprocessor -s -l -r Hell file1.txt
```

sort
add line numbers
remove string containing Hell

File content before:

```
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
```

hello

File content after:
1 --’’--911
2 #%Albert
3 Beatrice
4 albert
5 hello

Example 3:
```
fileprocessor -l -s -k ello file1.txt
```

add line numbers
sort
keep string containing ello

File content before:
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello

File content after:
1 Hello
2 hello

Example 4:
```
fileprocessor -t 2 -s -k l file1.txt
```

display first two char
sort
keep letters containing l

even if t comes first, #% is not omitted when sorted, Sorting is based on original string.

File content before:
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello

File content after:
#%
He
al
he

Example 5:
```
fileprocessor -t -s -k l -l file1.txt
```

display only first char
sort
keep letters containing l
add line number

File content before:
Hello
Beatrice

```
albert
@#$%
#%Albert
--’’--911
hello
```

File content after:

```
1 #
2 H
3 a
4 h
```

Deliverables Summary

This part of the document is provided to help you keep track of where you are in the individual project and will be updated in future deliverables.

<u>DELIVERABLE 1</u> (this deliverable, see below for details)

- **Provided:**
  - `fileprocessor` specification
  - Skeleton of the main class for `fileprocessor`
  - Example tests and skeleton of the test class to submit
  - JUnit libraries

- **Expected:**
  - Part I (Category Partition)
    - `catpart.txt`: TSL file you created
    - `catpart.txt.tsl`: test specifications generated by the `TSLgenerator` tool when run on your TSL file.
  - Part II (Junit Tests)
    - Junit tests derived from your category partition test frames (`MyMainTest.java`)

DELIVERABLE 2

- **provided:** TBD
- **expected:** TBD

DELIVERABLE 3

- **provided:** TBD

- **expected:** `TBD`

Deliverable 1: Instructions

Part I

Generate **between 30 and 60 test-case specifications** (i.e., generated test frames), inclusive, for the `fileprocessor` utility using the category-partition method presented in lesson P4L2. **Make sure to watch the lesson and demo before getting started.**

When defining your test specifications, your goal is to suitably cover the domain of the application under test, including **relevant erroneous inputs and input combinations**. Just to give you an example, if you were testing a calculator, you may want to cover the case of a division by zero.

Do not manually generate combinations of inputs as single choices. Instead, use multiple categories and choices with necessary constraints to cause the tool to generate meaningful combinations. Using again the calculator example, you should not offer choices *"add"*, *"multiply"*, and also *"add and multiply"* in a single category.

In particular, make sure to use constraints (error and single), selector expression (if), and properties appropriately, rather than eliminating choices, to keep the number of test cases within the specified thresholds.

Note that **the domain is that of the java application under test**, so you can assume that anything the shell would reject (e.g., unmatched double quotes) will not reach the application. In other words, **you must test for invalid input arguments, but do not need to test for errors involving parsing the command-line arguments before they are sent to the java application.** To illustrate, the sample tests in Part II will demonstrate how input arguments would be sent to your application.

Please also keep in mind that you are only required to specify test inputs, but you do not have to specify the expected outcome for such inputs in Part I. It is therefore OK if you do not know how the system would behave for a specific input. Using once more the calculator example, you could test the case of a division by zero even if you did not know how exactly the calculator would behave for that input.

Tools and Useful Files

You will use the `TSLgenerator` tool to generate test frames starting from a TSL file, just like we did in the demo for lesson P4L2. Versions of the `TSLgenerator` for Linux, Mac OS X, and Windows, together with a user manual, are available at:

- TSLgenerator-manual.txt
- TSL generator for Linux
- TSL generator for Mac OS
- TSL generator for Windows 8 and newer
- TSL generator for Windows XP and earlier

We are also providing the TSL file for the example we used in the lesson, cp-example.txt, for your reference.

Important:

- **These are command-line tools**, which means that **you have to run them from the command line**, as we do in the video demo, rather than by clicking on them.
- On Linux and Mac systems, you may need to change the permissions of the files to make them executable using the `chmod` utility. To run the tool on a Mac, for instance, you should do the following, from a terminal:

  ```
  chmod +x TSLgenerator-mac
  ./TSLgenerator-mac <command line arguments>
  ```
- You can run the `TSLgenerator` as follows:

  ```
  <tool> [--manpage] [-cs] infile [-o outfile]
  ```

  Where `<tool>` is the specific tool for your architecture, and the command-line flags have the following meaning:

  ```
  --manpage    Prints the man page for the tool.

  -c           Reports the number of test frames that would
               be generated, without actually producing them.

  -s           Outputs the result to standard output.

  -o outfile   Outputs the result to file outfile, unless the
               -s option is also used.
  ```
- If you encounter issues while using the tool, please post a public question on Piazza and consider running the tool on the VM provided for the class or on a different platform (if you have the option to do so). Gradescope will execute the tool on a Linux platform.

Committing Part I

- Create a directory "`IndividualProject`" **in the personal GitHub repo we assigned to you.**
- Add to this new directory two text files:

    - `catpart.txt`: TSL file you created.
    - `catpart.txt.tsl`: test specifications generated by the `TSLgenerator` tool when it processes your TSL file.
- Commit and push your files to GitHub. (You can also do this only at the end of Part II, but it is always safer to have intermediate commits.)

Part II

In this second part of the deliverable, you will create actual test cases implementing the test specifications you created in Part I. (As discussed in the lesson on the category-partition method, each test frame is a test spec that can be instantiated as an individual concrete test case). To do so, you should perform the following steps:

- Download archive individualproject.tar.gz
- Unpack the archive in the directory "`IndividualProject`", which you created in Part I of the deliverable. Hereafter, we will refer to this directory as `<dir>`.
- After unpacking, you should see the following structure:
  - `<dir>/fileprocessor/src/edu/gatech/seclass/fileprocessor/Main.java`
    This is a skeleton of the `Main` class of the `fileprocessor` utility, which we provide so that the test cases for `fileprocessor` can be compiled. It contains an empty `main` method and a method usage, which prints on standard error a usage message and should be called when the program is invoked incorrectly. In case you wonder, this method is provided for consistency in test results.
  - `<dir>/fileprocessor/test/edu/gatech/seclass/fileprocessor/MainTest.java`
    This is a test class with a few test cases for the fileprocessor utility that you can use as an example and that correspond to the examples of usage of `fileprocessor` that we provided. In addition to providing this initial set of tests, class `MainTest` also provides some utility methods that you can leverage/adapt and that may help you implement your own test cases:
    - `File createTmpFile()`
      Creates a File object for a new temporary file in a platform-independent way.
    - `File createInputFile*()`
      Examples of how to create, leveraging method `createTmpFile`, input files with given contents as inputs for your test cases.
    - `String getFileContent(String filename)`
      Returns a `String` object with the content of the specified file, which is useful for checking the outcome of a test case.
  - `<dir>/fileprocessor/test/edu/gatech/seclass/fileprocessor/MyMainTest.java`
    This is an empty test class in which you will add your test cases, provided for your convenience.
  - `<dir>/fileprocessor/lib/junit-4.12.jar` and `<dir>/fileprocessor/lib/hamcrest-core-1.3.jar`
    JUnit and Hamcrest libraries to be used for the assignment.
- Use the test frames from Part I to generate additional JUnit test cases for the `fileprocessor` utility and put them in the test class `MyMainTest` (i.e., **do not add**

your test cases to class `MainTest`). For ease of grading, please name your test cases `fileprocessorTest1`, `fileprocessorTest2`, and so on. Each test should contain a concise comment that indicates which test frame the test case implements. Use the following format for your comments, before each test:
`// Frame #: <test case number in file catpart.txt.tsl>`

Your test frames should contain enough information to create relevant test cases. **If you cannot implement your test frames as useful JUnit tests (e.g., because the test frames do not provide enough information), you should revisit Part I.** Extending the calculator example, if your test frame specified a numerical input, and you realized that you should use both negative and positive numbers in your actual test case, you should revise your categories and choices so that this is reflected in your test frames

- **If you are uncertain what the result should be for a test, you may make a reasonable assumption on what to use for your test oracle.** While you should include a test oracle, we will not grade the accuracy of the test oracle itself.

  Feel free to reuse and adapt, when creating your test cases, some of the code we provided in the `MainTest` class. Feel also free to implement your test cases differently. Basically, class `MainTest` is provided for your convenience and to help you get started. Whether you leverage class `MainTest` or not, your test cases should assume (just like the test cases in `MainTest` do) that the `fileprocessor` utility will be executed from the command line, as follows:

  ```
  java -cp <classpath> edu.gatech.seclass.fileprocessor.Main <arguments>
  ```

  Important: For this deliverable, do not implement the fileprocessor utility, but only the test cases for it. This means that most, if not all of your test cases will fail, which is fine.

Committing Part II and Submitting the Deliverable

- As usual, commit and push your code to your individual, assigned private repository.
- Make sure that all Java files are committed and pushed, including the ones we provided.
- Make also sure to commit and push the provided libraries (lib directory). To do so, you may need to force add the jar files (i.e., "`git add -f lib/*`"), which are typically excluded by the "`.gitignore`" file.
- You can check that you committed and pushed all the files you needed by doing the following:
  - Clone a fresh copy of your personal repo in another directory
  - Go to directory `IndividualProject/fileprocessor` in this fresh clone
  - Compile your code. One way to do is to run, from a Unix-like shell:
    ```
    javac -cp lib/\* -d classes src/edu/gatech/seclass/fileprocessor/*.java
    test/edu/gatech/seclass/fileprocessor/*.java
    ```

(on some platforms, you may need to first create directory "`classes`")

- Run your tests. Again, from a Unix-like shell, you can run:

```
java -cp classes:lib/\* org.junit.runner.JUnitCore
edu.gatech.seclass.fileprocessor.MyMainTest[1]
```

(at least some of the tests should fail with an `ArithmeticException`)

- **Submit on gradescope a file, called `submission.txt` that contains, in two separate lines (1) your GT username and (2) the commit ID for your submission.** For example, the content of file `submission`.txt for George P. Burdell could look something like the following:

`submission.txt`

```
gpburdell1
81b2f59
```

- **As soon as you submit, Gradescope will verify your submission** by making sure that your files are present and in the correct location, as well as a few additional minor checks. If you pass all these checks, you will see a placeholder grade of 10 and a positive message from Gradescope. Otherwise, you will see a grade of 0 and an error message with some diagnostic information. Please note that **a positive response from Gradescope only indicates that you passed the initial checks and is meant to prevent a number of trivial errors.** Please also note that **if your submission does not pass the Gradescope checks, it will not be graded and will receive a 0**, so please make sure to pay attention to the feedback you receive when you submit and keep in mind that **you can resubmit as many times as you want before the deadline.**[2]
- We may add additional checks to Gradescope in the next few days. If so, we will notify you on Piazza and ask you to check that your code still passes the required checks (if it did before).

[1] If using a Windows-based system, you may need to replace ":" with ";" in the classpath ("-cp").

[2] Although we tested the checker, it is possible that it might not handle correctly some corner cases. If you receive feedback that seems to be incorrect, please contact us on Piazza.

Individual Project: File Processor Deliverable 2

Project Goals

In this project, you will be developing a simple Java application (`fileprocessor`) using an agile, test-driven process involving multiple deliverables. While you will receive one grade for the entire project, each deliverable must be completed by its own due date, and all deliverables will contribute to the overall project grade.

Specification of the `fileprocessor` Utility

Program `fileprocessor` is a simple command-line utility written in Java with the following specification:

- NAME
  `fileprocessor` - utility for analyzing a text file
- SYNOPSIS
  `fileprocessor OPT <filename>`

  where `OPT` can be **zero or more** of
  - `-s`
  - `(-r|-k) <string s>`
  - `-t [integer n]`
  - `-l`
- COMMAND-LINE ARGUMENTS AND OPTIONS

  `<filename>`: file to be analyzed

  `-s`: if specified, `fileprocessor` will sort the lines in the file alphanumerically, (1) ignoring (but keeping) non-alphanumeric characters, (2) with numbers preceding letters, and (3) with capital letters preceding lowercase letters (i.e., 0-9, A-Z, a-z). Any line containing only non-alphanumeric characters is removed.

  `(-r|-k) <string s>`: if specified, `fileprocessor` will remove (`-r`) or keep (`-k`) only lines in the file which contain string `s` (case sensitive). Options `-r` and `-k` are mutually exclusive.

  `-t [integer n]`: if specified, `fileprocessor` will keep only the first `n` characters of each line. Value `n` must be a positive integer. If `n` is omitted, its default value is 1.

  `-l`: if specified, `fileprocessor` will add the line number at the beginning of each line, followed by a space and with the first line having line number 1.

If **no** OPT flag is specified, `fileprocessor` will leave the file unchanged.

- NOTES

  While the last command-line parameter provided is always treated as the filename, OPT flags can be provided in any order and will be applied in the order in which they are listed above (i.e., -s first and -l last).

- EXAMPLES OF USAGE

  Example 1:
  `fileprocessor -s -l file1.txt`

  File content before:
  Hello
  Beatrice
  albert
  @#$%
  #%Albert
  --’’--911
  hello

  File content after:
  1 --’’--911
  2 #%Albert
  3 Beatrice
  4 Hello
  5 albert
  6 hello

  Example 2:
  `fileprocessor -s -l -r Hell file1.txt`

  File content before:
  Hello
  Beatrice
  albert
  @#$%
  #%Albert
  --’’--911
  hello

  File content after:
  1 --’’--911
  2 #%Albert
  3 Beatrice
  4 albert
  5 hello

Example 3:

```
fileprocessor -l -s -k ello file1.txt
```

File content before:

```
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello
```

File content after:

```
1 Hello
2 hello
```

Example 4:

```
fileprocessor -t 2 -s -k l file1.txt
```

File content before:

```
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello
```

File content after:

```
#%
He
al
he
```

Example 5:

```
fileprocessor -t -s -k l -l file1.txt
```

File content before:

```
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello
```

File content after:

```
1 #
2 H
```

3 a
4 h

Deliverables Summary

This part of the document is provided to help you keep track of where you are in the individual project and will be updated in future deliverables.

~~DELIVERABLE 1~~ ~~(done)~~

- ~~**Provided:**~~
  - ~~`fileprocessor` specification~~
  - ~~Skelet.on of the main class for `fileprocessor`~~
  - ~~Example tests and skeleton of the test class to submit~~
  - ~~JUnit libraries~~

- ~~**Expected:**~~
  - ~~Part I (Category Partition)~~
    - ~~`catpart.txt`: TSL file you created~~
    - ~~`catpart.txt.tsl`: test specifications generated by the TSLgenerator tool when run on your TSL file.~~
  - ~~Part II (Junit Tests)~~
    - ~~Junit tests derived from your category partition test frames (`MyMainTest.java`)~~

DELIVERABLE 2 (this deliverable, see below for details)

- **Provided:**
  - Reference implementation of the `fileprocessor` utility (through Gradescope)
  - Set of instructor-provided test cases for `fileprocessor` (through Gradescope)

- **Expected:**
  - Possibly revised set of tests that pass on the reference implementation.
  - [OPTIONAL, for extra credit] implementation of `fileprocessor` that passes all the instructor-provided test cases.

DELIVERABLE 3

- **provided:** TBD
- **expected:** TBD

Deliverable 2: Instructions

In this deliverable, we will assume that someone else in your company implemented the `fileprocessor` utility while you were creating test cases for it. For the sake of the assignment, we will also assume that your colleague is an infallible developer and produced a flawless implementation. Consequently, **all the test cases you created for Deliverable 1 are supposed to pass on this implementation**.

To complete this deliverable, you will thus submit your test cases (i.e., class `MyMainTest`) and make sure that they pass on the reference implementation (Gradescope will run them for you and report information about passing and failing tests). Your grade will be proportional to the percentage of your test cases that pass. For example, if your test class consists of 50 test cases, and 40 of them pass on the reference implementation, your grade will be 80.

Optionally, for an extra 10 points, you can provide your own implementation of the `fileprocessor` utility. Gradescope will run a set of instructor-provided test cases and assign extra points proportionally to how many of these test cases pass on your code. For example, if your implementation of `fileprocessor` passes 50 of the 66 test cases, your extra credit will be 7.58. If you do not want to take advantage of this extra credit opportunity, you can simply submit the skeleton of the `Main` class that we provide for Deliverable 1. Please note, however, that you have to provide a compilable `Main` class, or Gradescope will reject your submission.

Committing and Submitting the Deliverable

- As usual, commit and push your code to your individual, assigned private repository.
- Make sure that all Java files and necessary libraries are committed and pushed.
- As you did for Deliverable 1, you can check that you committed and pushed all the files you needed by doing the following:
  - Clone a fresh copy of your personal repo in another directory
  - Go to directory `IndividualProject/fileprocessor` in this fresh clone
  - Compile your code. One way to do is to run, from a Unix-like shell:
    ```
    javac -cp lib/\* -d classes src/edu/gatech/seclass/fileprocessor/*.java
    test/edu/gatech/seclass/fileprocessor/*.java
    ```
    (on some platforms, you may need to first create directory "`classes`")
  - Run your tests. Again, from a Unix-like shell, you can run:
    ```
    java -cp classes:lib/\* org.junit.runner.JUnitCore
    edu.gatech.seclass.fileprocessor.MyMainTest
    ```
    [1]
- **Submit on gradescope a file, called `submission.txt` that contains, in two separate lines (1) your GT username and (2) the commit ID for your submission.** For example, the content of file `submission`.txt for George P. Burdell could look something

[1] If using a Windows-based system, you may need to replace ":" with ";" in the classpath ("-cp").

like the following:
`submission.txt`

```
gpburdell1
81b2f59
```

- **As soon as you submit, Gradescope will grade your submission** by:
    - Making sure that your files are present and in the correct location.
    - Performing the same sanity checks it performed in D1.
    - Compiling and running your tests against the reference implementation of `fileprocessor`.
    - Compiling your implementation of `fileprocessor` and running a set of instructor-provided tests against it.

  If any of the above steps fails, you will see a grade of 0 and an error message with some diagnostic information. Please note that, as before, **if your submission does not pass the Gradescope checks, it will not be graded and will receive a 0**. Conversely, **if Gradescope can successfully compile and run your code and tests, you will immediately receive a grade that is your actual grade for this deliverable**. Note that **you can resubmit as many times as you want before the deadline.**

Notes

- Your D1 submission should be able to pass all the checks and be graded, so it may be worth doing an initial submission right away, as a sanity check. Moreover, it could be the case that all of your test cases are already passing on the reference implementation, in which case you would be done with the mandatory part of the deliverable.
- You may have to modify some of your tests for them to pass on the reference implementation. This is expected and fine. You will not be penalized for this, and you will not need to modify your category-partition file accordingly.
- Because of the way the autograder works, we are expecting test class `MyMainTest` to be self-contained. That is, `MyMainTest` should not rely on any external classes or resources.
- Because the usage message in the example tests we provided for Deliverable 1 used `filesummary` as the name of the utility, we maintained that name in our test cases and reference implementation (for consistency). Please update your code and test cases accordingly, if needed.
- For obvious reasons, you will not be able to see the reference implementation, nor the instructor-provided test cases.
- If you decide to go for the extra credit opportunity, we recommend that you first make sure that all of your test cases pass on the reference implementation of the `fileprocessor` utility (mandatory part). You can then use your test suite to check your own implementation of `fileprocessor`, and submit it when it passes all of your tests.
- While the autograder provides details on which ones of your test cases fail on the reference code, it only reports the number of instructor-provided test

cases that fail on your code. However, fixing your own test cases should help you figure out why your code is failing on some of these instructor-provided test cases.
- Although we tested the autograder, it may still handle some corner cases incorrectly. If you receive feedback that seems to be incorrect, please contact us on Piazza.

Individual Project: File Processor Deliverable 3

Project Goals

In this project, you will be developing a simple Java application (`fileprocessor`) using an agile, test-driven process involving multiple deliverables. While you will receive one grade for the entire project, each deliverable must be completed by its own due date, and all deliverables will contribute to the overall project grade.

Specification of the `fileprocessor` Utility

Program `fileprocessor` is a simple command-line utility written in Java with the following specification:

- NAME
  `fileprocessor` - utility for analyzing a text file
- SYNOPSIS
  `fileprocessor OPT <filename>`

  where `OPT` can be **zero or more** of
  - `-s`
  - `(-r|-k) <string s>`
  - `-t [integer n]`
  - `-l`
- COMMAND-LINE ARGUMENTS AND OPTIONS

  `<filename>`: file to be analyzed

  `-s`: if specified, `fileprocessor` will sort the lines in the file alphanumerically, (1) ignoring (but keeping) non-alphanumeric characters, (2) with numbers preceding letters, and (3) with capital letters preceding lowercase letters (i.e., 0-9, A-Z, a-z). Any line containing only non-alphanumeric characters is removed.

  `(-r|-k) <string s>`: if specified, `fileprocessor` will remove (`-r`) or keep (`-k`) only lines in the file which contain string `s` (case sensitive). Options `-r` and `-k` are mutually exclusive.

  `-t [integer n]`: if specified, `fileprocessor` will keep only the first `n` characters of each line. Value `n` must be a positive integer. If `n` is omitted, its default value is 1.

  `-l`: if specified, `fileprocessor` will add the line number at the beginning of each line, followed by a space and with the first line having line number 1.

If **no** OPT flag is specified, `fileprocessor` will leave the file unchanged.

- NOTES

  While the last command-line parameter provided is always treated as the filename, OPT flags can be provided in any order and will be applied in the order in which they are listed above (i.e., -s first and -l last).

- EXAMPLES OF USAGE

  Example 1:

  ```
  fileprocessor -s -l file1.txt
  ```

  File content before:

  ```
  Hello
  Beatrice
  albert
  @#$%
  #%Albert
  --’’--911
  hello
  ```

  File content after:

  ```
  1 --’’--911
  2 #%Albert
  3 Beatrice
  4 Hello
  5 albert
  6 hello
  ```

  Example 2:

  ```
  fileprocessor -s -l -r Hell file1.txt
  ```

  File content before:

  ```
  Hello
  Beatrice
  albert
  @#$%
  #%Albert
  --’’--911
  hello
  ```

  File content after:

  ```
  1 --’’--911
  2 #%Albert
  3 Beatrice
  4 albert
  5 hello
  ```

Example 3:
```
fileprocessor -l -s -k ello file1.txt
```
File content before:
```
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello
```
File content after:
```
1 Hello
2 hello
```

Example 4:
```
fileprocessor -t 2 -s -k l file1.txt
```
File content before:
```
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello
```
File content after:
```
#%
He
al
he
```

Example 5:
```
fileprocessor -t -s -k l -l file1.txt
```
File content before:
```
Hello
Beatrice
albert
@#$%
#%Albert
--’’--911
hello
```
File content after:
```
1 #
2 H
```

3 a
4 h

Deliverables Summary

This part of the document is provided to help you keep track of where you are in the individual project and will be updated in future deliverables.

~~DELIVERABLE 1 (done)~~

- ~~**Provided:**~~
  - ~~`fileprocessor` specification~~
  - ~~Skeleton of the main class for `fileprocessor`~~
  - ~~Example tests and skeleton of the test class to submit~~
  - ~~JUnit libraries~~
- ~~**Expected:**~~
  - ~~Part I (Category Partition)~~
    - ~~`catpart.txt`: TSL file you created~~
    - ~~`catpart.txt.tsl`: test specifications generated by the `TSLgenerator` tool when run on your TSL file.~~
  - ~~Part II (Junit Tests)~~
    - ~~Junit tests derived from your category partition test frames (`MyMainTest.java`)~~

~~DELIVERABLE 2 (done)~~

- ~~**Provided:**~~
  - ~~A reference implementation of the `fileprocessor` utility (through Gradescope)~~
  - ~~Set of instructor-provided test cases for `fileprocessor` (through Gradescope)~~
- ~~**Expected:**~~
  - ~~Possibly revised set of tests that pass on the reference implementation.~~
  - ~~[OPTIONAL, for extra credit] implementation of `fileprocessor` that passes all the instructor-provided test cases.~~

DELIVERABLE 3 (this deliverable, see below for details)

- **Provided:**
  - Code and interfaces to use as a starting point for this deliverable
  - Instructor-provided test cases for the `FileProcessor` class (through Gradescope)

    - A reference implementation of the `FileProcessor` class (through Gradescope)

- **Expected:**
    - Implementation of the `FileProcessor` class that passes all instructor-provided tests
    - **[OPTIONAL, for extra credit]** test cases for the `FileProcessor` class that pass on the provided reference implementation.

Deliverable 3: Instructions

Your project manager has received positive feedback for the command-line version of the `fileprocessor` utility. They have therefore decided to provide its functionality through a library and an API, so that additional applications can be built on top of it. Part of your team has already created a Java interface for the API (`FileProcessorInterface`), which you will be able to download, and a set of tests for it, which you will be able to run on your code through Gradescope. **Your job is to develop a class, named `FileProcessor`, that implements the `FileProcessorInterface` interface and passes the provided tests.**

To complete this deliverable, you will submit your code (i.e., class `FileProcessor` and related files) and make sure that it passes all the instructor-provided tests (Gradescope will run them for you and report information about passing and failing tests). **Your grade will be proportional to the percentage of instructor test cases that pass on your code.**

Optionally, **for an extra 10 points**, you can create and submit, **after you achieve a perfect score on the mandatory part of the deliverable**, 30 (or more) different and non trivial JUnit test cases for the `FileProcessor` class, in a class called `MyLibTest`. Gradescope will run your tests on a reference implementation of the class, report the results, and assign extra points to your submission proportionally to how many of your test cases pass. If you do not want to take advantage of this extra credit opportunity, you can simply submit the skeleton of the `MyLibTest` provided as part of this deliverable (you have to provide a `MyLibTest` class anyway, or Gradescope will reject your submission).

Detailed Steps and Submission Information

- Download archive [individualproject-d3.tar.gz]().
- Unpack the archive in the directory "`IndividualProject`", which you created in Part I of the deliverable. Hereafter, we will refer to this directory as `<dir>`.
- After unpacking, you should see the following files:
    - `<dir>/fileprocessor/src/edu/gatech/seclass/fileprocessor/FileProcessorInterface.java`
      This is the interface that you have to implement in class `FileProcessor`. Make sure to read the description of the methods and their parameters in the

comments before each method, including the details about exceptions. Please ask on Piazza if something is not clear. Given that the core functionality did not change, the methods should be self-explanatory.

  - `<dir>/fileprocessor/src/edu/gatech/seclass/fileprocessor/ProcessingException.java`
    The exception to be thrown by your `FileProcessor` class in case of errors, with a suitable message. You can use the provided test cases to figure out which error messages to use. Please note that they are similar to those used for the command-line version of the utility, with some additional messages and small changes.
  - `<dir>/fileprocessor/test/edu/gatech/seclass/fileprocessor/MyLibTest.java`
    This is a test class with a single test case for the `FileProcessor` class, to give you an idea of how a test would look in this context. Your job, if you decide to go for the extra credit, will be to add test cases to the class to get to a total of 30 different test cases,[1] including the provided one. To avoid issues with Gradescope, **make sure to (1) name your test cases `fileprocessorTest*`** (where "*" can be any string, not only a number), and **(2) annotate every test with the `@Test` annotation, on the line right before the test**. To create these test cases, you can use, and suitably modify, a subset of the tests that you created for the command-line version of the `fileprocessor` utility. Please keep in mind that **checking for error conditions will involve detecting that the right exception has been thrown, and that the exception contains the correct message**, as printing a usage message would not make sense in this context.
  - `<dir>/fileprocessor/lib/junit-4.12.jar and`
    `<dir>/fileprocessor/lib/hamcrest-core-1.3.jar`
    `<dir>/fileprocessor/lib/lang-2.1.0.jar`
    Libraries to be used for the assignment.
- As usual, commit and push your code to your individual, assigned private repository.
- Make sure that all Java files and necessary libraries are committed and pushed. Gradescope will let you know if some essential file is missing.
- As you did for Deliverables 1 and 2, you can check that you committed and pushed all the files you needed by doing the following:
  - Clone a fresh copy of your personal repo in another directory
  - Go to directory `IndividualProject/fileprocessor` in this fresh clone
  - Compile your code. One way to do is to run, from a Unix-like shell:
    ```
    javac -cp lib/\* -d classes src/edu/gatech/seclass/fileprocessor/*.java
    test/edu/gatech/seclass/fileprocessor/*.java
    ```
    (on some platforms, you may need to first create directory "`classes`")
  - Run your tests. Again, from a Unix-like shell, you can run:
    ```
    java -cp classes:lib/\* org.junit.runner.JUnitCore
    ```

[1] It is fine if some tests are similar, but you should avoid having multiple copies of the same test or the same test repeated multiple times with minimal variations (e.g., the parameter for the trimming ("-t") option going from 1 to 10 or similar). I am sure you got the idea.

`edu.gatech.seclass.fileprocessor.MyLibTest`[2]

- **Submit on gradescope a file, called `submission.txt` that contains, in two separate lines (1) your GT username and (2) the commit ID for your submission.** For example, the content of file `submission.txt` for George P. Burdell could look something like the following:
  `submission.txt`
  ```
  gpburdell1
  81b2f59
  ```
- **As soon as you submit, Gradescope will grade your submission** by:
  - Making sure that your files are present and in the correct location.
  - Performing a set of sanity checks.
  - Compiling your implementation of the `FileProcessor` class and running a set of instructor-provided tests against it.
  - [If all the instructor tests pass on your code] Compiling and running your tests (against the reference implementation of the `FileProcessor` class.

  If any of the above steps fails (except for the last, optional one), you will see a grade of 0 and an error message with some diagnostic information. Please note that, as before, **if your submission does not pass the Gradescope checks, it will not be graded and will receive a 0**. Conversely, **if Gradescope can successfully compile and run your code and tests, you will immediately receive a grade that is your actual grade for this deliverable**. Note that **you can resubmit as many times as you want before the deadline.**

Notes

- If you did the optional part of Deliverable 2, you should be able to reuse and modify the code you developed to create the `FileProcessorInterface` class.
- Because of the way the autograder works, we are expecting test class `MyLibTest` to be self-contained. That is, `MyLibTest` should not rely on any external classes or resources. Except for libraries that are included in your submission.
- You will not be able to see the reference implementation, nor the instructor-provided test cases, but you will be able to see the test output.
- Usual disclaimer: Although we tested the autograder, there may still be issues with it when it runs on some of your submissions. If you receive feedback that seems to be incorrect, please contact us on Piazza, and we will do our best to react promptly.

[2] If using a Windows-based system, you may need to replace ":" with ";" in the classpath ("-cp").