Individual Project: txter

Deliverable 1

## Project Goals

In this project, you will be developing a Java application, txter, using an agile, test-driven development process across multiple deliverables. You will receive one grade for the entire project, but each deliverable must be completed by its own due date and all deliverables will contribute to the overall project grade.

## Specification

txter is a command-line utility written in Java with the following specification:

### Summary

txter allows for simple text manipulation of the contents of a file.

### Syntax

txter [OPTIONS] FILE

### Description

The program txter performs basic text transformation on the lines of text from an input file. It is invoked as a command-line tool using the [syntax described above](#_tyjcwt), after compilation. The program writes the transformed text to the standard output and errors or usage messages to the standard error without modifying the input file. The FILE parameter is required and must be the last parameter as shown above. The options allowed in the program, which are optional, delimited by the left [ and right ] brackets, may be provided in any order and are described as follows:

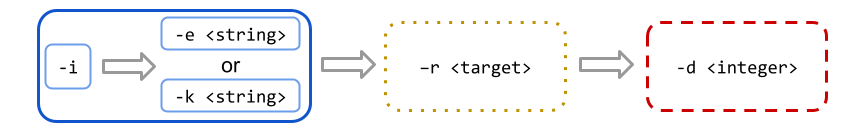
| **Option** | **Description** |
| --- | --- |
| -e <string> | **Exclude Lines**: Exclude lines containing the given substring. This option is mutually exclusive with -k. |
| -k <string> | **Keep Lines:** Keep only the lines containing the given substring. This option is mutually exclusive with -e. |
| -i | **Case Insensitive**: Applies case insensitive matching when used with "Keep Lines" and "Exclude Lines". This option takes no parameters. |
| -r <target> | **Reverse text**: A valid target here is either “words” or “text”. Reverse words or text in each line according to the parameter “words” or “text”. The "words" string parameter will reverse the order of words in each line, but leave the characters of the words in the original order. The "text" string parameter will reverse the characters within each line. Spaces should be considered as a word or characters here. |
| -d <integer> | **Duplicate Lines:** Duplicate each line to the file the number of times specified. 0 is minimum and 9 is maximum for <integer>. |

#### Order of execution

The last command-line parameter provided is always treated as the filename, as shown in the [syntax section](#_tyjcwt), while OPTIONS flags can appear in any order and parsed as they appear from left to right. This means that the following two commands are equivalent when executed on the command line:

| Example 1 | txter -d 1 -k ..? input.txt |
| --- | --- |
| Example 2 | txter -k ..? -d 1 input.txt |

In the above examples, (Example 1) parses -d first, then -k, and finally input.txt while (Example 2) parses -k first, then -d, and finally input.txt. These two will result in the same output (assuming that the same input.txt is used for both) because the **parsing of options is independent of their execution order**. The order of execution for each option is given in the diagram below (note that the colors and border lines are there for ease of viewing):



The above diagram of the execution order of options can also be described as follows:

1. If -i is **present**, then -k or -e shall keep or exclude lines containing matched strings with case **insensitive** manner.If -i is **not present**, then -k or -e shall keep or exclude lines containing exact matched strings with case **sensitive** manner.
2. If -r is present, then text in each line shall be reversed according to its parameter.
3. If -d is present, each line shall be duplicated by <integer> times.

#### Notes

* To keep this application simple, all errors shall result in the display of the standard usage message. This is shown in the [syntax section](#_tyjcwt) above as well as in the files provided to you which are linked in [later sections](#_6aksms3x82t6).
* If options are repeated, only their last occurrence is applied. The [Examples section](#_2s8eyo1) below shows a case of repeated options.
* An empty input file shall produce an empty output. The [Examples section](#_2s8eyo1) below shows a case of an empty input file.
* You shall assume that the command line parameter strings will not contain newline characters (\r, \n, \r\n) as the behavior of the program would be platform dependent and may result in error during grading. Therefore there should be no test cases using these values as option parameters. Additionally, you may assume that your application will be called with valid String[] args.

#### Program errors

* The last line of a **non-empty** input file must be terminated by a newline. If the non-empty input file does not terminate in a newline, the program shall generate an error.
* All parameters of program options are required and shall result in an error if omitted. This means that parsing an option that should include parameters but doesn't should result in an error. The [Examples section](#_2s8eyo1) below shows a case of missing option parameters.
* Specifying option parameters with the wrong type shall result in an error.
* Specifying option -i without -e or -k shall result in an error.
* Specifying option -i with parameters shall result in an error.
* Specifying options -e and -k simultaneously shall result in an error.
* Specifying option -r with a parameter outside of the allowed values ("words", "text") shall result in an error.
* Specifying option -d with an integer outside of the inclusive range 0 to 9 as the duplication parameter shall result in an error.

### 

### Examples of Usage

The examples described here can also be seen in JUnit 5 form on the MainTest.java file [provided to you in the below sections](#_3whwml4). In the following, "↵" represents a newline character.

| Example 1 | |
| --- | --- |
| txter -e test sample.txt | |
| input sample.txt |  |
| stdout | **nothing sent to stdout** |
| stderr | **nothing sent to stderr** |

| Example 2 | |
| --- | --- |
| txter -i sample.txt | |
| input sample.txt | Hello, world!↵  ↵  How are you?↵ |
| stdout | **nothing sent to stdout** |
| stderr | Usage: txter [ -i | -k substring | -e substring | -r target | -d num ] FILE↵ |

| Example 3 | |
| --- | --- |
| txter -e Bird sample.txt | |
| input sample.txt | Dog goes "woof".↵  Cat goes "meow".↵  Bird goes "tweet".↵ |
| stdout | Dog goes "woof".↵  Cat goes "meow".↵ |
| stderr | **nothing sent to stderr** |

| Example 4 | |
| --- | --- |
| txter -k Mouse sample.txt | |
| input sample.txt | Dog goes "woof".↵  Cat goes "meow".↵  Bird goes "tweet".↵ |
| stdout | **nothing sent to stdout** |
| stderr | **nothing sent to stderr** |

| Example 5 | |
| --- | --- |
| txter -r words sample.txt | |
| input sample.txt | Baby Shark , doo - doo, doo-doo, doo-doo ↵ |
| stdout | doo-doo doo-doo, doo, - doo , Shark Baby↵ |
| stderr | **nothing sent to stderr** |

| Example 6 | |
| --- | --- |
| txter -r text sample.txt | |
| input sample.txt | !syas siht tahw wonk reven lliw uoY↵  ...tiaw ho↵ |
| stdout | You will never know what this says!↵  oh wait...↵ |
| stderr | **nothing sent to stderr** |

| Example 7 | |
| --- | --- |
| txter -d 8 -d 2 sample.txt | |
| input sample.txt | Baby Shark, doo-doo, doo-doo, doo-doo↵  Mommy Shark, doo-doo, doo-doo, doo-doo↵ |
| stdout | Baby Shark, doo-doo, doo-doo, doo-doo↵  Baby Shark, doo-doo, doo-doo, doo-doo↵  Baby Shark, doo-doo, doo-doo, doo-doo↵  Mommy Shark, doo-doo, doo-doo, doo-doo↵  Mommy Shark, doo-doo, doo-doo, doo-doo↵  Mommy Shark, doo-doo, doo-doo, doo-doo↵ |
| stderr | **nothing sent to stderr** |

| Example 8 | |
| --- | --- |
| txter -r words -d 1 -k cat -i sample.txti | |
| input sample.txt | Dog goes "woof".↵  Cat goes "meow".↵  Bird goes "tweet".↵ |
| stdout | "meow". goes Cat↵  "meow". goes Cat↵ |
| stderr | **nothing sent to stderr** |

## 

## Deliverables Summary

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

### Deliverable 1

**Provided**

* txter specification ([linked in this document above](#_3znysh7))
* Skeleton of the main class for txter ([linked below](#_3whwml4))
* JUnit 5 example tests and skeleton of test class to submit ([linked below](#_3whwml4))
* JUnit 5 library ([linked below](#_3whwml4))

**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)
  + MyMainTest.java: JUnit tests derived from your category partition test frames

### 

### Deliverable 2

#### Provided: TBD

#### Expected: TBD

### 

### Deliverable 3

#### Provided: TBD

#### Expected: TBD

### 

### Deliverable 4

#### Provided: TBD

#### Expected: TBD

## Instructions

Deliverable 1 is split up in two parts: Part I and Part II. Follow the instructions for each of the parts as described below.

### Part I

Generate 50 to 90 (inclusive) test-case specifications (i.e., generated test frames) for the txter utility using the category-partition methods present 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 input and input combinations. Just to give you an example, if you were testing a calculator, you may want to cover the case of division by zero.

**Do not manually generate combinations of inputs as single choices**. Instead, use multiple categories and choices with necessary constraints for the tool to generate meaningful combinations. Using the calculator example, you should not offer choices "add", "multiple", and also "add and multiply" in a single category - an example of what **not to do** can be found in [calculator-manual-combinations.txt](https://drive.google.com/file/d/1_sjrsrktXHCdS3GXX95F9E_zxdM6MqW8/view?usp=sharing). In particular, make sure to use constraints (error and single), selector expressions (if), and properties appropriately, rather than elimination 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, null) will not reach the application. In other words, you must test for invalid input arguments, but you do not need to test for errors involving parsing the command-line arguments before they are sent to the Java application. In other words, you may assume that the Main.main() will be called with a valid String[] of args. Although this application will only use the simplified command-line parsing given in the specification above, you can find more details about comprehensive command-line argument parsing at [this link](https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html) (GNU "Program Argument Syntax Conventions"). To illustrate the sample tests in Part II will demonstrate how input arguments would be sent to your application.

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 the calculator example, you could test the case of a division by zero even if you do 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, and Windows, together with a user manual, are available at:

* [TSLgenerator-manual.txt](https://drive.google.com/file/d/0B6urvdQprh5Nc3VCc25ITzBZSVE/view?usp=sharing&resourcekey=0-83Er3T6ZDJNN82NCz1tkGw)
* [TSL generator for Linux](https://drive.google.com/file/d/1-HIXRIYxKxl84EkTfBmGg8zl5Wku3JIr/view?usp=sharing)
* [TSL generator for Mac OS](https://drive.google.com/file/d/1wtyKIm1cXFCWvqRUZ6tnYQoLWZjJ-75t/view?usp=sharing)
* [TSL generator for Windows 8 and newer](https://drive.google.com/file/d/1QnuTHzJX4w8zgw3HVhl3L_DFu2Rl2F78/view?usp=sharing)
* [TSL generator for Windows XP and earlier](https://drive.google.com/file/d/1PqeeFKKS8jceg_TuiR-JmyQyB6MobnJV/view?usp=sharing)

We are also providing the TSL file for the example used in the lesson, [cp-example.txt](https://drive.google.com/file/d/0B6urvdQprh5NUkNJVjVLdzNaUWc/view?usp=sharing&resourcekey=0--Qmzq5CTzAJUdQTPecoKag), for reference, as well as an example for explaining <n/a> values, [tsl-na-example.md](https://drive.google.com/file/d/1cqPdFRFil-YH-XnP8lqSOqKeKLlwBpbs/view?usp=sharing).

Please note that:

* These are **command-line tools**, which means they have to be run 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 Ed Discussion and consider running the tool on a different platform (if you have the option to do so). Gradescope will execute the tool on a Linux platform.

#### Committing Part I

1. Create a directory "IndividualProject" **in the personal GitHub repo we assigned to you**.
2. Add the following two text files to this new directory:
   1. catpart.txt: TSL file you created.
   2. catpart.txt.tsl: test specifications generated by the TSLgenerator tool when it processes your TSL file.
3. Commit and push your files to GitHub.

### 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 specification that can be instantiated as an individual concrete test case. To do so, you should perform the following tests:

1. Download archive [individualproject-d1.tar.gz](https://drive.google.com/file/d/1uCyKurgY0TqFm2Mcp-y9Kogt0BbSOr-u/view?usp=sharing)
2. 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>.
3. After unpacking, you should see the following structure:
   1. <dir>/txter/src/edu/gatech/seclass/txter/  
      Main.java

This is a skeleton of the Main class of the txter utility, which we provide so that the test cases for txter can be compiled. It contains an empty main method and a method usage, which prints, on standard error, a usage message that should be called when the program is invoked incorrectly. In case you wonder, this method is provided for consistency in test results.

* 1. <dir>/txter/test/edu/gatech/seclass/txter/  
     MainTest.java

This is a test class with a few test cases for the txter utility that you can use as an example; it corresponds to the examples of usage of txter that we provided. In addition to providing this initial set of tests, class MainTest also provides some utility extensions and methods that you can leverage/adapt and that may help you implement your own test cases. **We encourage you to use the methods or input files to ease your design process**. This file contains:

* Path createFile(String contents, String fileName)  
  Creates a Path object with the specified fileName for a new temporary file in a platform-independent way with the contents provided. An additional overloaded method creates an "input.txt" file with the contents provided.
* OutputCapture capture  
  Class field that captures standard output and standard error by using the OutputCapture extension below (d). Note that it must be registered with "@RegisterExtension" as shown in MainTest.java to capture the output from Main.
  1. <dir>/txter/test/edu/gatech/seclass/txter/  
     MyMainTest.java

This is an empty test class in which you will add your test cases, provided for your convenience.

* 1. <dir>/txter/test/edu/gatech/seclass/txter/  
     OutputCapture.java

This is a JUnit 5 extension class to facilitate capturing the standard output and standard error, which are needed to test the txter application. It is used on the MainTest.java file for reference and provides two methods to capture output from Main:

* public String stdout()  
  Returns a String from the current standard output stream, resetting for every test executed.
* public String stderr()  
  Returns a String from the current standard error stream, resetting for every test executed.
  1. <dir>/txter/lib/junit-platform-console-standalone-1.9.1  
     .jar

JUnit 5 library to be used for the assignment.

1. Use the test frames from Part I to generate additional JUnit test cases for the txter utility, one per frame, and put them in the test class MyMainTest. **Do not add your test cases to the MainTest class**. For ease of grading, please name your test cases txterTest1, txterTest2, 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.

1. **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 the MainTest class or not, your test cases should assume (just like the test cases in MainTest do) that the txter utility will be executed from the command line, as follows:

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

Please note:

* Make sure not to make calls to System.exit() within your tests, as that creates problems for JUnit.
* For this deliverable, do not implement the txter 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 sure to also 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/txter in this fresh clone
  + Compile your code. One way to do this is to run, from a Unix-like shell (on some platforms, you may need to first create directory "classes"):  
    javac -cp lib/\\* -d classes src/edu/gatech/seclass/txter/\*.java test/edu/gatech/seclass/txter/\*.java
  + Run your tests. Again, from a Unix-like shell, you can run:  
    java -cp classes:lib/\\* org.junit.platform.console.ConsoleLauncher --select-class edu.gatech.seclass.txter.MyMainTest[[1]](#footnote-0)
* **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**;
  + **if your submission does not pass the Gradescope checks, it 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]](#footnote-1)

## Gradescope Queries

If you need clarification or have questions regarding Gradescope output, please post privately on Ed Discussion (we will make it public if appropriate) and make sure to add, when it applies:

* a link to the Gradescope results, and
* any information that may be relevant.

The bottom line is that, to make the interaction efficient, you should make your posts as self-contained and easy-to-check as possible. The faster we can respond to the posts, the more students we can help.

## FAQ

1. What's next in Deliverable 2? What is the workload like?

In fairness to everyone, we cannot discuss future deliverables. You will have to wait to find out the details of deliverable 2 when it’s released.

1. Can I create additional tests not generated by TSL or avoid implementing some of the generated test frames?

No, for part 2 you need to stick to the test frames that were generated in part 1.

1. Aside from the program options and parameters, should the command line and file contents be considered as separate category partitions?

Yes, both command lines and file content are also inputs to the program, so they should be considered when testing.

1. Can I re-use the example test cases (input strings, arguments, etc.) or test methods for my own test cases?

Yes, you may use the example test cases to devise your own, in addition to using the structure and test methods provided.

1. Why am I getting an error stating that a property isn't defined even though it is?

Since properties are case- and space-sensitive, make sure that defined properties don't contain blank spaces between them and have the same case.

1. If using a Windows-based system, you may need to run  
   java -cp "classes;lib/\*" org.junit.platform.console.ConsoleLauncher  
   --select-class edu.gatech.seclass.txter.MyMainTest instead. [↑](#footnote-ref-0)
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 Ed Discussion. [↑](#footnote-ref-1)