CS4218 Assumptions Document

Project Assumptions

The team tries to follow the Linux implementation as closely as possible. Hence, unless otherwise stated, it is safe to assume that the implementation follows the Linux implementation. However, these are some gray areas where the team has made certain assumptions to guide our implementation of the CS4218 project.

General Assumption:

1. **Ordering of Options**

- Because of the fact that different Linux system has different rules on the ordering of options (e.g., wc Application -w is valid on WSL Ubuntu but is not valid on MAC as -w is seen as a file name), the team has decided to allow different ordering rules that are functions based
- Each ordering rule is specified below
- *(Note: For flexible ordering rules where it is allowed to have <command> [FILENAME] [OPTIONS], filenames with dashes(-) are not supported as they are deemed as OPTIONS flags. Repeated [OPTIONS] can also be supplied and the command will execute according to the [OPTIONS] that has appeared, unless stated otherwise. E.g. "sort -n -n" works the same as "sort -n".)*

2. **Project Description Requirements take precedence over the Linux Implementation.**

- (e.g., ls -d is not supported for this application, although it is supposed in Linux)
- (e.g., As long as it meets the specified requirements, some of the different such as extra spaces are reasonable for the Java Application as long as it does not affect the correctness)

3. **Do not handle filename with IO Redirection symbols (< or >)**

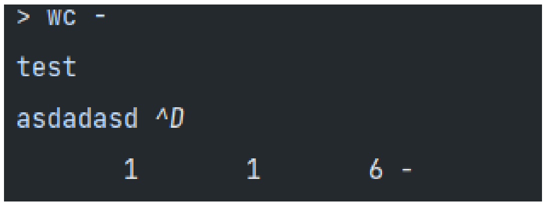
   < or > symbols are used for IO Redirection. Using any filenames with those symbols at the front may result in an undesired output due to how arguments are parsed as it is difficult to differentiate whether it is a filename with those symbols or to redirect the input and output of the provided file. E.g. ">example.txt" would be seen as redirecting the output into a file with the filename "example.txt".

4. **Pressing CTRL + D when executing any command terminates the shell**

   Since the commands and the shell share the same input stream, when we CTRL + D in a command's execution, it closes that input stream, and the shell application cannot continue executing.

5. **Limitation of StdIn with Intellij**

   When typing in standard input, the user must enter a new line for the line result to be recognized. For example, "test" is not considered in the calculation of word, line and bytes, while "test" + ENTER / Newline is considered



```
> wc -
test
asdadasd ^D
       1       1       6 -
```

6. **Accepted Language**

   The shell only accepts ASCII characters and not any other non-english languages. Applies to filenames and shell inputs.

7. **File creation**

   For commands that will result in file creation, if a filename of "*.txt" is supplied, different results will occur on different OS. On windows, it will result in an error as the creation of files with filenames "*" is not allowed. On Linux/Mac, it will run as per normal as the creation of files with filenames that include "*" is allowed. Similar rules apply to directory's creation as well.

8. **File permissions**

   For input files supplied that do not have read permissions, an error will occur. For output files supplied that do not have write permissions, an error will occur. These may behave differently depending on the OS the user on.

9. **Console / Error Message**

   Generally, the console / error messages returned from the program might not be the same as their Linux counterpart. However, the error messages should generally still make sense. *(e.g. When an invalid file is given, an invalid file message or failure message is given)*

10. **Operating System**

   Testing of the implementation is mostly done on Windows due to the limited accessibility of machines of other operating systems like Mac. Although the differences between operating systems are taken into account to the best of our abilities, it is best to assume that this shell implementation is intended to work only on Windows.

11. **Directory Path Arguments**

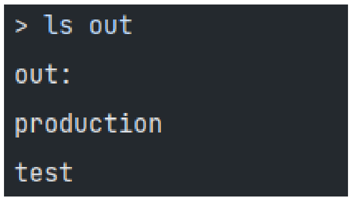
   For any applications or commands that take in the directory path as arguments, the user's OS rules take precedence. E.g., directory with special characters such as **colon :** and **asterisk *** are deemed illegal on Windows while accepted on Linux / macOS. Such behavior will follow the user's machine OS, and it is unable to be controlled by the Shell Application.

Functions Based Assumptions:

1. **ls Command**

- Hidden files check is done using the File.isHidden() method. This method according to the official JavaDocs is system-dependent and might have different results depending on the OS. https://docs.oracle.com/javase/7/docs/api/java/io/File.html#isHidden()
- Allows flexible ordering of [OPTIONS] to appear anywhere after the ls command

  When multiple files are specified, even if 1 of them produces an error, the program will still continue running and list the remaining files.
- Following ubuntu 18, the following assumptions for results display are taken:
  - Files are separated using a new line (Rather than spaces)
  - For single directory results, no file headers are displayed
  - The folder name will be appended in front of the files when running this command with folders.

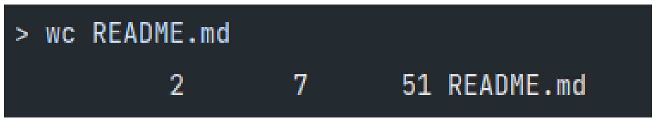


```
> ls out
out:
production
test
```

(Sample Ls Command Output)

2. **wc Command**

- WC follows the Linux implementation for counting the number of lines. (It detects \n for a new line)
    - https://unix.stackexchange.com/questions/83408/why-does-wc-l-tell-me-that-this-non-empty-file-has-0-lines
- The command must be in the exact format of wc [OPTIONS] [FILENAME]
    - E.g. For wc Application.java -l, -l is considered as a file name because of the format
- There are some inconsistencies with the wc command across operating systems that is replicated in the implementation as well
    - https://stackoverflow.com/questions/12616039/wc-command-of-mac-showing-one-less-result
    - https://stackoverflow.com/questions/15290203/why-is-a-newline-2-bytes-in-windows
- The team follows the requirement stated in the FAQ and used tabs between the strings in the output. Following the skeleton code, numbers are printed in the '%7d' format on top of being separated by tabs ('\t').
    - Sample output is as follows:



```
> wc README.md
          2        7       51 README.md
```

3. **cp Command**

- cp follows the Linux implementation whereby
    - A directory cannot be copied into itself

- If the destination does not exist as a file or folder, it will create a file with the destination as its name

  Eg. cp a.txt DIRECTORY where DIRECTORY does not exist in current directory

  → A file named DIRECTORY will be created and the content of a.txt will be copied into it

- Allows flexible ordering of [OPTIONS] to appear anywhere after the cp command
- Overwriting of files that requires permission is not supported
- When multiple files/folders are given as arguments, even if 1 of them produce an error, the remaining files/folders will still be copied over

4. tee Command

- Allows flexible ordering of [OPTIONS] to appear anywhere after the tee command
- Running commands like `tee *.txt` with no ".txt" files in the directory will behave differently based on the OS. Refer to General Assumptions #7 for the specifics

5. Quoting

- When the same types of quotes nested within itself (eg. "hello "world""), the evaluation of quoting should match each quote to the nearest one

  Eg. `echo "Hello "World""` → `"Hello "` + `World` + `""` → `Hello World` displayed

6. IO Redirection

- Only allow a maximum of one input and output redirection for each call command. Generally means no `< file1.txt < file2.txt`, no `> output1.txt > output2.txt`. However, may allow `< *.txt` or `> *.txt` on Linux/Mac since the OS allows creation of files with "*" as filenames, not referring to all files, unlike on Windows.
- For output redirection, even if there is no output, a file will be generated if that file does not exist. However, does not create a new directory and a file if the directory does not exist.

- For output redirection, does not open the output stream if the file does not have write permissions.

7. Sort Command

- Allows flexible ordering of [OPTIONS] to appear anywhere after sort command.
- Will sort according to conditions stated in the project document, different from Linux (sort purely by ASCII values). For this sort, special characters (anything not a-z, A-Z, 0-9) comes first, followed by numbers, followed by capital letters, followed by small letters, and are sorted by ASCII values within their classes as stated in the project document.
- Can sort according to inputs from stdin together with inputs from files by providing “-” as a filename. Similar to Linux. E.g. `sort file1.txt - file2.txt` will sort all inputs from file1.txt, file2.txt and supplied inputs from stdin.
- All files that are supplied must be valid in order for sort to work. E.g. `sort file1.txt nonexistfile.txt file2.txt` will throw an exception.
- Negative numbers like “-1” are treated as special characters due to “-”.
- -n flag does not precede over -f flag, unlike in Linux.

8. Cut Command

- [OPTION] and [LIST] must be supplied.
- [OPTION], [LIST], [FILES…] must be supplied in the following order: `cut OPTION LIST FILES…`
- Does not allow repeated [OPTION]. E.g. `cut -c -c` and `cut -cc` is not allowed.
- Numbers provided in [LIST] must be smaller than Integer.MAX_VALUE.
- Cut can still work when invalid files are provided, even with a mix of valid and invalid files. Valid cut outputs will be displayed for valid files with error messages for invalid files in the order they are supplied. `System.out.println` method is used to output error messages instead of throwing exceptions to replicate output behaviour in Linux.
- Empty files will not produce any outputs. Files will still be created if there is output redirection.
- For the -c option, cutting by character position. Lines will be cut by characters and not bytes unlike in Linux where -c and -b work the same. E.g. `cut -c 1` with input line ñ will output ñ, not a symbol for non-printing characters.

- For the -b option, cutting by byte position. Lines will be cut by bytes as usual, similar in Linux. `cut -b 1` with input line ñ will output a symbol for a non printing character as "ñ" consists of two bytes.
- May not show the exact same error outputs as in Linux but error messages would still make sense. Just for more clarity of the error.
- When doing unit tests on methods other than the `run` method, `setStdout` method is used to set a non-null OutputStream that the results would be written into. This is due to a global OutputStream variable used to replicate immediate output results while entering inputs from stdin.

9. **Echo Command**

- Works similarly to the Linux implementation.
- Does not accept any [OPTION].

10. **Grep Command**

- [OPTIONS] PATTERN [FILES] can be supplied as arguments. With no FILES, or when FILES is -, grep will read standard input. The sequence of arguments must be strictly followed and any missing arguments may cause unintentional unexpected behavior. E.g., `grep A.txt`, although A.txt is semantically a file name, it is interpreted as a pattern in such use-cases according to the sequence of arguments and will request inputs via standard input.
- The filename prefix flag (-H) will prefix all inputs from the input stream with “(standard input)” as the filename, following the Linux implementation.
- Unable to accept empty pattern input while Linux shell takes in an empty pattern and returns the whole file.
- The count lines flag (-c) have precedence over the other flags, and the shell will output line count.
- The count lines flag (-c) are applied to each individual input source instead of a summation of the count of all input sources.
- With the filename prefix flag (-H), if more than one matching line is found within the same text file, all the matching lines will be prefixed with the filename.
- Grep (normal or count lines) with multiple files or file with input stream will automatically prefix the outputs with the file names regardless of the -H flag.
- To replicate the output behavior in Linux, `System.out.println` method is used to output error messages (related to insufficient permission, is a directory or file not found) instead of throwing exceptions or appending to the result string (the original behavior from the skeleton codes). This is also used to allow Shell Program to print outputs on the fly and the results are also appended to the result string.

11. cat Command

- Contrary to actual Linux implementation, when line number flag (-n) is included:
    - there would not be any spacing between the start of line and the line number
    - a SPACE is used instead of TAB in between the line number and the content
- Both of the above are in compliance with the example given in the project description document.
- While some Linux implementations disallow the line number flag (-n) to be anywhere in the command, our implementation allows for it. Therefore, commands such as cat file1.txt -n will be considered valid.
- In compliance with Section 10 of the project description, if a command such as `cat - text1.txt - text2.txt` is supplied, then the expected behavior will be:

```
> abc
> 123
(Ctrl+D)
Abc
123
<content of text1.txt>
<content of text2.txt>
```

- Based on the above point, this implementation will then deviate from the Linux implementation where standard inputs are returned on demand.
- Since standard outputs are not given to the separate catFiles, catStdin and catFileAndStdin methods, if a cat command is run with invalid files, there will not be execution for any file, but instead an error will be thrown. For example, `cat exist.txt invalid.txt` will throw a “file does not exist” error for invalid.txt and there will not be any output for exist.txt.

12. rm Command

- Similar to the Linux implementation, rm does not allow removal of files/directories for “.” and “..”
- Error for invalid files are deferred. For example, `rm exist1.txt invalid.txt exist2.txt` will remove both exist1.txt and exist2.txt and throw an error for invalid.txt for being an invalid file.

13. mv Command

- The mv command moves files / folder to its target, similar to its Linux counterpart.

Even when there are invalid source files specified, the remaining valid source files will still be moved as long as the target given is valid.

14. Call Command

- While the initial instructions for Milestone 1 are to create unit tests for the “call” command, the team felt that we will not be able to truly isolate “CallCommand” as a single unit, hence, only integration tests were carried out for it.
- If we take the CallCommand.evaluate() method as an example, it depends on 3 main classes: IORedirectionHandler, ApplicationRunner, and ArgumentResolver. Therefore, to truly isolate the class, we’ll have to mock each of the three classes. However, creating mocks does not serve its purpose as the behavior will be predictable according to the mock behavior given.
- Additionally, the three dependent classes have their own unit tests, and we assume that these tests are done in a sound manner. Therefore, we proceeded with an integration test from the bottom-up, with these three main classes.

15. Pipe Command

- While the initial instructions for Milestone 1 is to create unit tests for the “pipe” operator, the team felt that we will not be able to truly isolate “PipeCommand” as a single unit, hence, only integration tests were carried out for it.
- The pipe command/operator strongly depends on the behavior of the list of “CallCommand” being provided to it. However, to mock the “CallCommand” class would involve mocking its evaluate() method and manipulation of InputStream/OutputStream which is difficult to achieve.
- Therefore, based on the fact that “CallCommand” has been thoroughly tested, we would rely on the actual mechanisms on it and carry out an integration test between CallCommand and PipeCommand.

16. Sequence Command

- Works similarly to the Linux implementation.
- Sequence command is used when the semi-colon (;) operator is used.
- Sequence command will continue to operate even if the previous command in the sequence returns an error message.

17. Exit Command

- Upon calling exit of the shell program, the ExitApplication will throw an ExitException which will be caught in the main Shell program to execute `System.exit(0)` instead of exiting directly from the ExitApplication. This allows better testability for writing test cases.

- As such, the in execution behavior will differ slightly when Exit is used with commands such as **Sequence**, **Command Substitution**, **Pipe**, etc. It will not end the program immediately. E.g., `exit; echo “hello”` will still execute echo, then exit the application (complying to the project’s requirement for sequence).

18. Command Substitution

- As described in this assumption document, the respective command's behavior takes hold as the command substitution does not modify any of the behaviors.
- In the scenario of the exit command executed as a command substitution, the program will exit instead of displaying an empty line or standard input which slightly differs from the Linux shell implementation.
- Nested command substitution (nested back quotes) is not supported. It will be treated as a continuous command following Linux's behavior, and the last intended command will be treated as a string.
- e.g., ``echo `echo `echo CS4218`` ``, output: `echo CS4218`
- Incomplete command substitution syntax is not supported, and in the events where single or double quotes wrap back quotes, the quoting rules take precedence. e.g., ``echo 'a`a'`` will disable the backquote as it is treated as a special character, and, e.g., ``echo "a`a"``, since there is a matching pair of double quotes, it will take precedence and output anything before the back quotes. Anything after the back quote is invalid and discarded.

19. Uniq Command

- Allows flexible ordering of [OPTIONS] to appear anywhere after the uniq command.
- Reads all inputs first before outputting the result unlike in Linux where there is immediate output of results while providing input. Example:

```
> uniq
Test
Test
(Ctrl + D)
Test
```

- Input of “test” is treated the same as “test\n”, unlike in Linux as there is a difference of “\n”. This is due to the limitations of `readLine` method of reader classes. Example, `uniq` without options with input of “test\ntest” is same as input of “test\ntest\n” where the output is “test”.
- For -c option, uses “\t” for the first spacing to replicate output in Linux.

- Would throw the same error if -c and -D options are present together. Similar in Linux.
- When doing unit test on methods other than the `run` method, `setStdout` method is used to set a non-null OutputStream that the results would be written into. This is due to a global OutputStream variable used to satisfy the description of the methods provided by the interface.

20. Paste command

- The serial flag (‘-s’) can appear anywhere before or after the file arguments.
- Behavior of paste without any argument (ie. “paste”) is based on Linux’s implementation where the application will keep asking for user input and echo any given line:

```
> paste
a
a
b
b
c
c
```

- The output printed by calling Paste with files that have very different line length will be based on the behavior of Linux’s implementation and not WSL’s implementation
- In compliance with Section 10 of the project description, if a command such as `paste - a.txt -` is executed, then the expected behavior will be:

```
> paste - a.txt -
1
2
(CTRL + D)
1   a   2
    b
```

21. Cd Application

- Adopting Linux’s behavior, cd must take in one argument. If there is more than one argument, it is deemed as an error.
- As mentioned in the CdApplication.java comments in the skeleton code, cd without arguments is not supported and is deemed as an error.

22. Globbing

- The arguments are case-sensitive. If the file or folder names are capitalized e.g. README, `ls read`* will not return any results.
- As Windows does not accept invalid characters such as * in the directory name, globbing with an asterisk may throw InvalidPathException error such as Illegal char <*>. This can be witnessed when attempting to use globbing on files or folders that do not exist. e.g., `ls hello*` where there are no files that start with the prefix "hello", the Shell Application will treat `hello*` as a string argument which will cause erroneous behavior on Windows as described.
- Globbing with an asterisk at the front as pattern will only supports dot and alphanumeric characters and not the usage of other symbols e.g., */ or */*. Due to the OS interpreting the directory path and folders differently, it's difficult to capture such semantics via the Shell Application.
- Due to OS differences in interpreting "/" as names e.g., a/b/c.txt is interpreted as a:b:c.txt in macOS, files or folders containing slash "/" may or may not be output through the use of globbing.