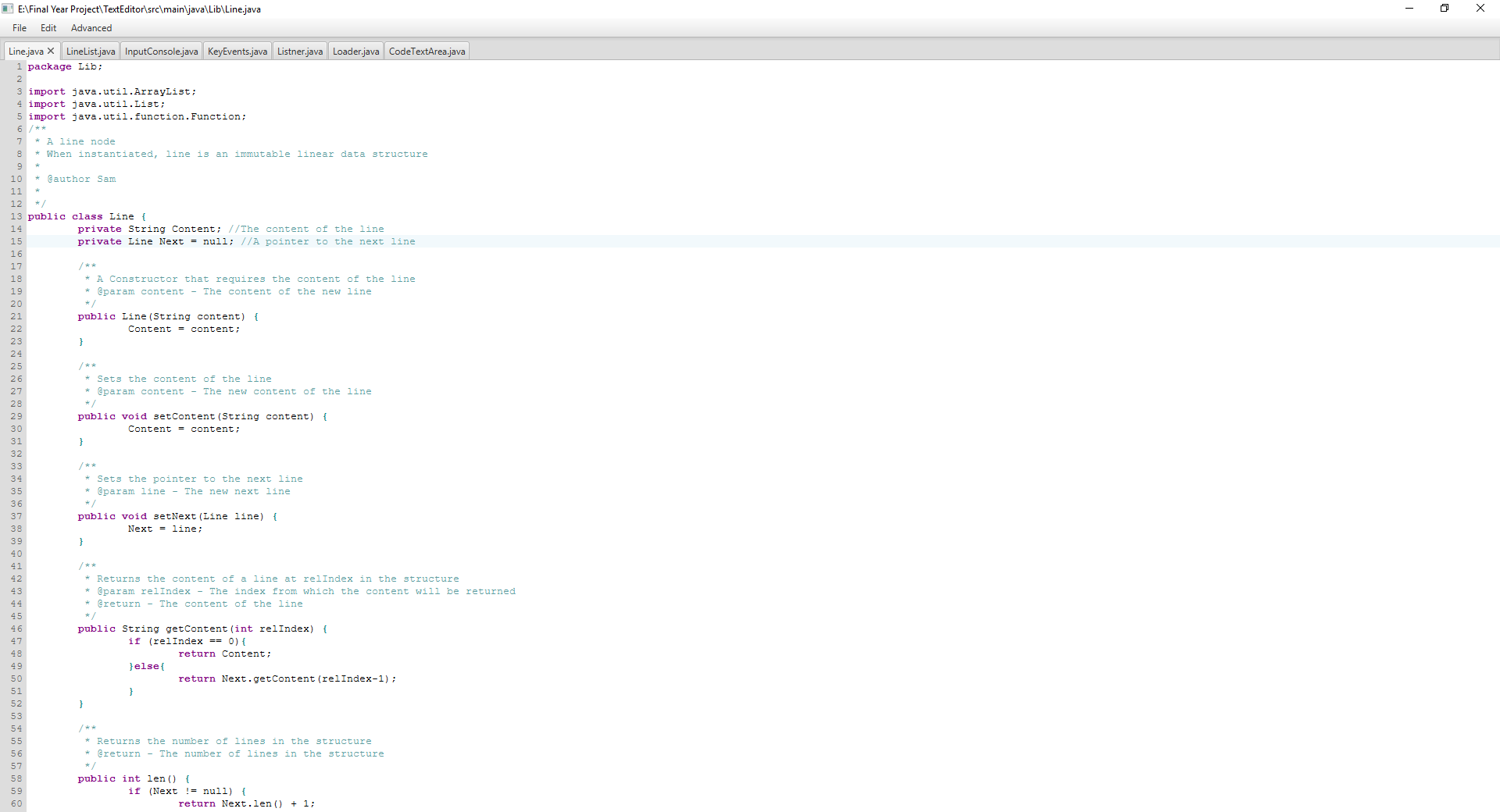
Creation of a Text Editor

P15194468

Sam Fulller

2018



# 1 Abstract

Text editors are an important piece of software, allowing users to view and edit a text file. For this project a text editor was created, and in this report the features and development of the program will be explained. The text editor uses a graphical interface to display and edit text, however also contains terminal input allowing for input of repetitive commands. A small language was created with the ability to combine commands, repeat commands and parse arguments.

Table of Contents

[1 Abstract 1](#_Toc512282857)

[2 Introduction 3](#_Toc512282858)

[3 Development 4](#_Toc512282859)

[3.1 Data Structures 4](#_Toc512282860)

[3.2 Graphical Interface 6](#_Toc512282861)

[3.3 Terminal Interface 9](#_Toc512282862)

[3.4 Events 16](#_Toc512282863)

[3.5 Resource Files 19](#_Toc512282864)

[4 Conclusion 20](#_Toc512282865)

[5 Tools Used 21](#_Toc512282866)

[6 Appendicies 22](#_Toc512282867)

# 2 Introduction

Text editors are used by programmers to open, edit and save plain text files, these text files can be in a specific computer language or simply plain text files. Although integrated development environments provide many useful features for creating and debugging code, text editors still have their place. An IDEs range of features are useless if editing a language it does not support, therefore text editors are superior when opening multiple files of different languages, or less popular languages that don’t have an IDE.

There are two main types of text editor: those with terminal interfaces and those with graphical interfaces. Text editors with graphical interfaces use the mouse for navigation and the keyboard for entering text with functions appearing in a menu bar. This style of text editor is more intuitive, the cursor can be positioned, text edited, and functions used without the need to learn specific terminal commands. Terminal interface text editors are often very powerful at performing simple repetitive tasks, for example using the Vim text editor the command 6dw will delete six words. This method of input is much faster and often provides more complex and powerful commands than those you find in a graphical interface.

The text editor created for this project combines the usability of a graphical interface and the power of terminal commands. The graphical interface of the project consists of three components, a menu bar, a tab pane, and a code text area.

The first component, the menu bar, allows access to multiple features: undo/redo, find, replace, new, open, save, and save as, these features can also be accessed through shortcuts and terminal functions. These features are the basis of a text editor and therefore need to be as accessible as possible.

The second component, the tab pane, simply enables multiple files to be opened at once in the same instance of the text editor. Using the open feature will open a new tab with the text from a specified file inside, whereas clicking new will just create a new tab. Tabs can be closed, if the last tab is closed, a new empty tab is opened.

The third component, the code text area, is a custom object that extends the RichTextFX code area. This is component that displays the text. The code text area also highlights language syntax and the find string.

The terminal interface of the project is simply the command prompt that launched the program or the eclipse console. The terminal interface has many commands that allow for repetitive actions to be performed quickly.

I initially chose to create a text editor not only because of the opportunity to extend my experience with object orientated programming, but also for the useful product created. A text editor that I can continue to develop for use with the languages I enjoy programming in, with the features I need (and ability to code in new ones) appeals to me and will be a program I will use even after handing in the project. This project has improved by ability with creating data structures, file handling and JavaFX. In addition, creating a text editor has taught me about maven and using dependencies like RichTextFX in eclipse.

# 3 Development

## 3.1 Data Structures

A text editor requires all text to be stored inside of a data structure. Line is a complicated data structure because it acts as both a node and a linked list. If this structure were to be used by other programmers, a wrapper class would be developed that stores a line node and allows access to the nodes linked list commands. However; since this data structure is only used in this project the functionality of the wrapper is not needed.

Each line node stores a pointer to the content of the line and a pointer to the next line in the collection. This means that duplicate lines can have pointers to the same string, saving memory. Figure 3.1 – 1 shows this concept and how each line stores only pointers.

Line 1

Content

Next

Line 2

Content

Next

String 1

String 2

Figure 3.1 - 1

When used as a collection, line acts as a linked list with the ability to add, remove and get lines at given indexes. These functions act recursively through the linked list.

The project also requires undo functionality, to achieve this linelist contains a list of the different line structures. In order for each change to be added to the linelist and not affect any previous line structures the line data structure is immutable, with each function returning a new line structure with the required function performed.

Line 1

Content

Next

Line 3

Content

Next

String 1

String 2

Line 1

Content

Next

Line 2

Content

Next

String 3

LineList

Figure 3.1 - 2

Figure 3.1 - 2 shows a representation of memory after a new line is added to the text. Initially only the top row of lines (line 1 and line 3) are in memory. When the new line (line 2) is added, first the structure is navigated recursively, with each line preceding the index of the new line (in this example just line 1) being duplicated but the pointers to strings in these lines remaining the same. Once the index for the new line is reached, the new line is inserted with a pointer to the same next line as the initial version. This keeps all of the following lines identical, and therefore keeps memory useage to a minimum. A similar process is used for the remove and edit functions, but with the line being removed or changed instead of a new one added.

Changes can take up multiple lines (for example copying a large portion of text should only count as a single update) therefore the prep() function exists, this indicates to linelist that all changes between now and the next prep() call should all count as the same version. This way a loop can add each line of an edit in turn without creating unnecessary undo points. This method of updating changes to memory creates many loose ends, with many heads being created and then “forgotten”. These “forgotten” heads with no pointers to them are collected by the java garbage collector, keeping the memory free.

Before implementing these data structures many other solutions were conceived and trialled. One alternative consisted of simply a list of lists, however this implementation requires each line to be stored in memory for each version. This can quickly become resource intensive as memory usage is multiplied with each version. Another implementation idea involved storing each line structure inside of a tree, this would remove the limitation of a linear structure. In the current system if changes are undone and a new change is made, all undone changes are lost. In a tree system, a change would simply make a new branch. The linear method was chosen over a tree because the tree proved difficult to navigate.

Because there is no way of visually testing these classes both data structures are tested using Junit test cases, these can be found at appendix 3.1 – 1 and appendix 3.1 - 2. Line test contains simple tests ensuring that the line data structures functions perform correctly. Line list test contains tests with massive amounts of lines to test how many lines java can handle.

Due to the recursive nature of the line structure, the project cannot store more than around 9000 lines. This is a completely reasonable limitation, very few files are longer than this. However; the program could be updated so that instead of accessing lines recursively, it is accessed through a loop updating a pointer each iteration.

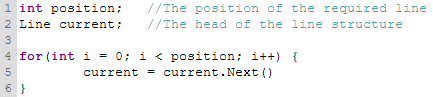


Figure 3.1 - 3

Figure 3.1 – 3 shows an example of this type of loop. Position would be set to the position of the line required, and current would be initially set to the head of the line structure. With each iteration of the loop current is updated to be the next line in the structure without the need for recursion. After the loop is finished, current is equal to the line at the index of position. Removing the recursive factor would keep the prevent the stack from overflowing, and therefore allow the program to store significantly more lines.

The nature of object orientated programming would make this change relatively easy, with only the methods in LineList needing to be updated. This is an example of how the program will continue to be developed in the future.

## 3.2 Graphical Interface

Loader contains the main function of the program, it sets up the listener, input console and the graphical interface. The loader combines three main components. The first component is the menu bar, this provides access to the features like find and undo. These features can also be accessed by shortcuts and terminal functions, as explained later in the report. The second components is the tab pane, which simply provides a method of opening more than one file at once. New tabs can be made using the menu bar and old ones can be removed. Each tab contains its own code text area, the third component.

The code text area contains its own line list and the file path of the file it relates to, new tabs don’t relate to a file and so instead are left empty. Tabs take the name of the file that their code text area relate to, or “unsaved” if they don’t relate to a file, and the title of the stage changes to the absolute file path of the current tab.

The code text area calls two events 500ms after the last change to the displayed text, if a change happens in those 500ms, the events are not called. The first is the change calculation function, the other handles syntax highlighting.

When the program detects a change in the text, it cannot simply enter all displayed text into the line list, that would create a new set of lines each time. There needs to be a change calculation function so that only the changes in the displayed text are applied to the line list.

Line 1

Line 2

Line 3

Line 4

Line 5

Line 1

Line A

Line B

Line 5

Memory

Display

Line 2

Line 3

Line 4

Line A

Line B

Memory

Display

Figure 3.2 - 1

The changes are calculated by comparing the two lists and removing all similarities from either end. Figure 3.2 – 1 shows two states of two lists, before and after the changes have been calculated. The algorithm first loops from index 1 towards the middle; after comparing the two line 1 appearances, line 1 is removed, because it is duplicated no changes need to be made to the line list regarding this line. The algorithm then compares line 2 and line A, because they are different the loop breaks, and the lists are then compared from the other end. Once both ends have been compared and duplicates removed, the lists contain only the lines that are different. The lines in the memory list appeared in line list but are have been deleted by the user and are therefore removed from line list, lines that appear in the display list have been added by the user and are then added to the line list.

The code text area also provides syntax highlighting, this is done using java regular expressions and cascading style sheets. When the program is launched, an initial default style sheet is loaded and applied, this contains the style for the find string and all text. If the syntax highlighter is set to a language, that languages style sheet and regex file are loaded to every code text area, alternatively if the highlighter is set to auto, the file type is used as the language. The CSS styles are simply applied to the code text area; however, the java regular expressions must be combined into a pattern. The pattern is then applied to the text through a matcher, for each group the matcher finds the CSS tag is added to the style span of the code text area.

Initially the code text area was a Swing text area, however this did not format the text correctly when getting text from the area. The text area was updated to the JavaFX text area, although when attempting to complete the syntax highlighting it was realised that the JavaFX text area did not accommodate for colouring only parts of the text. After researching alternatives RichTextFX (Martinez, n.d.) was selected for its ability to highlight parts of the text and a function that provides line numbers. RichTextFX is installed to the project through maven.

Both classes were fully tested to ensure all their components work. In addition, the project was partially coded using the graphical interface, this not only tests each function works as intended, but that the project is usable.

The tab pane and menu bar were tested by simply clicking each button and ensuring they work, the function of each button is tested in events. The syntax highlighting and find function required more detailed testing to ensure the features work properly.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tested feature | Test | Input | Output | Result |
| Menu bar | A visual check to make sure all buttons appear | N/A | Appendix 3.2 – 1  Appendix 3.2 – 2  Appendix 3.2 – 3 | Each button appears, and is in the correct position |
| Menu bar | Each button is clicked to ensure they all work | Each button is clicked |  | When clicked each button performs its function |
| Tab pane | A new tab is created | New tab button is clicked | Appendix 3.2 – 4 | A new empty tab appears and is selected |
| Tab pane | All tabs are closed | The x on each tab is pressed |  | A new empty tab appears |
| Find | A file is opened and a string that appears multiple times is found | Line.java is opened  “public” is found | Appendix 3.2 – 5 | The red from find takes priority over the purple from keyword |
| Find | A continuation on the previous test, the string is now removed. | Same as the last test  The find string is removed | Appendix 3.2 – 6 | All instances of public change back to purple |
| Replace | A file is opened and a string is replaced with another string | Java.css is opened  “bold” is replaced with “test” | Appendix 3.2 – 7 | All instances of bold are replaced with test |
| Replace | A file is opened and a string is replaced with nothing | Java.css is opened  “bold” is replaced with “” | Appendix 3.2 – 8 | All instances of bold are removed |
| Replace | A file is opened and nothing is entered | Java.css is opened  “” is replaced with “” | Appendix 3.2 - 9 | The replace button cannot be pressed |

Whilst coding the project inside of itself, it was found that having to switch between files by saving, closing and then opening the new file was an inelegant solution. Initially there were no tabs in the program and the menu bar and text area were simply in the loader, once tabs were added these were split up into two classes. The loader now contains the features common across all tabs, for example the menu bar and input console. Code text area now contains all tab specific content like the file and text. This separation means multiple code text areas can now be instantiated and each placed in a separate tab.

Using the program also revealed that when syntax highlighting is determined automatically find would not be highlighted. This is because the find CSS tag used to be placed with the language CSS tags. The fix for this was to separate the find tag into another default CSS file along with font size.

## 3.3 Terminal Interface

The terminal interface allows for simple repetitive commands to be completed quickly, Appendix 3.3 – 1 shows a list of all commands, the arguments they accept and return, and the function of each. This interface can perform most tasks the graphical interface can, however provides additional functions for repetitive tasks.

Commands are entered in sequence the last character splitting up each argument, for example “add:5:”foo”:” will call the add command with the arguments 5 and “foo”, the colon can be replaced with any single character. In a command where a colon is needed as part of the argument a semicolon might be more appropriate: “add;5;”foo:”;”. This may not be the most elegant solution, but it allows for the input of all characters, as appose to forcing the joining character to be a colon where a colon entered in a string would not be detected. The joining character allows for the input to be split into separate arguments and placed in a list.

Commands can also be strung together, with commands that return arguments feeding them in to other commands. A good example of this is the echo command where “echo:getactivetab:” will return the index of the active tab. Once the list of arguments is generated, they are each executed from right to left. Getactivetab will be run first and return the index of the active tab to the list in its place, echo then prints the next argument in the list to the console, which in this case is the result of the getactivetab argument.

Any argument that is not a recognised command is turned into a function and evaluated. “echo; 5 + 2;” will be print 7, 5 + 2 is turned into a function and calculated before echo prints it to the console. The function is applied to the last argument, this allows the user to call x, for example “echo; x + 5; 2;” will still print out 7. This may not appear useful, however the same principle can be applied to other functions. “activetab; x+1; getactivetab;” will get the current active tabs index, then add one to the index, and finally set the active tab as the new index, effectively moving the focus to the next tab along.

The program calculates integer and string arguments separately, if x is an integer the function is applied as an integer function, otherwise it is applied as a string function. If the system did not differentiate between the two and x is always treated as a string, “echo; x + 5; 2;” will return 25, when applied to strings + concatenates the two arguments.

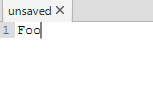
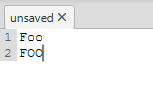
The repeat function allows a single command to be repeated any number of times. “repeat:10:add:5:”foo”:” will add the line “foo” at the line number 5 a total of 10 times. The repeat function allows repetitive tasks to be completed. This function works by keeping a repeat counter of the number of times the code has already been run, if the counter is less than the argument passed to repeat, the entire command line is re-evaluated. In future versions, the repeat argument could be streamlined to only re-evaluate every command with side effects.

The remove function accepts any number of line numbers, and removes the line at each. A list is generated with each of the given line numbers, it is then ordered from largest to smallest. If this step is not completed and for example, line 5 is removed before line 10, then after the deletion of line 5, line 10 is moved to line 9. Ordering from largest to smallest ensures that the correct lines are removed.

The ! function removes itself and the next argument from the list. This allows for later arguments to be inputted into a command. “echo:!:echo:4:”bar”:” will print 4 and then bar, in this example the ! removes the 4 so that the preceding echo instead prints bar.

When joined together these functions become increasingly powerful for example a function that creates a duplicate line in uppercase:

“add;2;x.toUpperCase();getline;1”

Future updates of this text editor will have the ability to bind commands to keys. This can be done by introducing a bind command that accepts a key and a series of commands, the key and command is added to a list, whenever the listener is called the list is checked for the combination. If the combination has been pressed the command is executed. This feature will allow commands such as “activetab; x+1; getactivetab;” to be bound to a key, this particular command would select the next tab along.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tested feature | Test | Input | Output | Result |
| echo | The echo command is tested with a simple string | echo;”hello world”; |  | hello world is echoed into the console |
| echo | Echo is entered with no following string | echo; |  | Error is returned and more commands can be entered |
| New tab | New tab command with a positive integer argument | newtab;3; | Appendix 3.3 – 1 | 3 new tabs are created |
| New tab | New tab command with 0 argument | newtab;0; |  | Error is returned and more commands can be entered |
| New tab | New tab command with a string argument | newtab;”foo”; |  | Error is returned and more commands can be entered |
| New tab | New tab command with no argument | newtab; | Appendix 3.3 – 2 | A single new tab is created |
| Active tab | Active tab command with positive argument | 4 additional tabs are created and the first selected  activetab;3; | Appendix 3.3 – 3 | The fourth tab is selected (assuming the first is 0) this is correct |
| Active tab | Active tab command with 0 argument | Same setup as previous test  activetab;0; | Appendix 3.3 – 4 | The first tab is selected |
| Active tab | Active tab command with negative argument | activetab;-10; |  | Error is returned and more commands can be entered |
| Active tab | Active tab command with no argument | echo;activetab; |  | Prints the index of the current tab |
| Get active tab | Echo the result of the get active tab command | echo;getactivetab; |  | Prints the index of the current tab |
| open | Open with a correct file path | open; “E:/Final Year Project/TextEditor  /src/main/java/Lib/Line.java”; | Appendix 3.3 – 5  Appendix 3.3 – 6 | The file is opened in a new tab |
| open | Open with a file path to a non existant file | open; “E:/NotAFile.txt”; |  | Error is returned and more commands can be entered |
| Open  FAILED – code updated | Open with no arguments | open; |  | Throws an exeption, code needs to be modified |
| open | Repeat of the test above, but with modified code | open; |  | Error is returned and more commands can be entered |
| save | Save command on an opened file | Line.java is opened and changes are made  save; |  | File has been saved and the edits have been checked in Microsoft’s notepad |
| Save | Save command on an unsaved file | save; |  | Error is returned and more commands can be entered |
| undo | An undo with no arguments | undo; |  | Undoes the last change |
| undo | Undo with positive argument | undo;4 |  | Undoes the last 4 changes |
| undo | Undo with 0 argument | undo;0; |  | Error is returned and more commands can be entered |
| redo | Redo with no argument | redo; |  | The last undo is redone |
| redo | redo with positive argument | redo;4; |  | The last 4 undoes are redone |
| redo | redo with 0 argument | redo;0; |  | Error is returned and more commands can be entered |
| find | Find with a string argument | find;”foo”; |  | All instances of foo are highlighted |
| find | Find with no arguments | find; |  | The previous find is removed |
| replace | Replace with both arguments | replace;”foo”;”bar”; |  | All instances of foo are replaced with bar |
| replace | Replace with a single argument | replace;”foo”; |  | Removes all instances of foo |
| replace | Replace with no arguments | replace; |  | Error is returned and more commands can be entered |
| map | Map with a function argument | map; x+”foo”; |  | All lines have foo appended |
| map | Map with no argument | map; |  | Error is returned and more commands can be entered |
| add | Add with both arguments | add;3;”foo”; | Appendix 3.3 – 7 | The line foo is added at line number 3, empty lines are added before |
| add | Add with a single argument | add;2; | Appendix 3.3 - 8 | An empty line is added at line number 2 |
| add | Add with two arguments, first argument is 0 | add;0;”foo”; |  | Error is returned and more commands can be entered |
| Line gen | Line gen with all arguments | linegen;1000;1;”foo”; | Appendix 3.3 – 9 | 1000 lines of foo are entered |
| Line gen | First argument is 0 | linegen;0;1;”foo”; |  | Error is returned and more commands can be entered |
| Line gen | Second argument is 0 | linegen;1000;0;”foo”; |  | Error is returned and more commands can be entered |
| Line gen | Line gen with only 2 arguments | linegen;1000;1; | Appendix 3.3 – 10 | 1000 empty lines are entered |
| repeat | Repeat is with correct number and an echo command | repeat;5;echo;”foo”; |  | Foo is printed exactly 5 times |
| repeat | Repeat with an integer and no command | repeat;10; |  | The code repeats nothing 10 times, future improvements may make checks for this |
| Repeat | Repeat with 0 | repeat;0;echo;”foo”; |  | Echo is called once, when repeat is called an error is returned and more commands can be entered |
| ! | ! command with an echo before and two arguments after | echo;!;”foo”;”bar” |  | Foo is removed and bar Is printed |
| ! | ! command with no argument | !; |  | Error is returned and more commands can be entered |
| Argument evaluation | Evaluation of a single string argument | echo;”hello “ + “world”; |  | Hello world is printed |
| Argument evaluation | Evaluation of two string arguments, using x to join them | echo;”hello “ + x; “world”; |  | Hello world is printed |
| Argument evaluation | Evaluation of a single integer argument | echo;10 \* 5; |  | 50 is printed |
| Argument evaluation | Evaluation of two integer arguments, using x to join them | echo;10 \* x; 5; |  | 50 is printed |

## 3.4 Events

The events can be accessed in three ways the menu bar, shortcuts, and terminal commands. The shortcuts are handled by an event that calls the listener class with the key code of the key that has been pressed. Shortcuts consist of multiple keys, and a key press refers to that of only one key, therefore a Boolean must be stored containing if the key is pressed or not. When the key pressed is called the Boolean for the key that has been pressed is set to true, when key released is called it is set to false. When both keys of a shortcut are true the matching key event is called. In addition, the listener calls the code text areas change calculator whenever the arrow keys, enter, or mouse button is pressed, this ensures undo points are created at relevant times.

Key events contain the event of each function. The open event first creates a new tab for the file to be loaded into, the user is then prompted to give a file path using a file chooser. If the syntax highlighter is set to auto the file extension is used to load the relevant highlighter files if found. All lines are then read into the line list of the new tab.

Save places the line list text into the file, if the code text area does not have a file then save calls the save as function which requests a file path before saving. Save as requests a file path from the user by creating a file chooser, the given file path is then added to the code text area, which allows for syntax highlighting files to be selected and the tab/stage title to be renamed. Finally, the text is then saved to file.

Undo and redo call the undo and redo functions of line list. The find function sets the find string in the code text area to the string inputted to an input dialog, the function then calls the syntax highlighter so that the text can be re-formatted.



Figure 3.4 – 1

Figure 3.4 – 1 shows the code for the map function. The map function takes a input string and changes it to a java function, this is then be applied to each line of text. The javascript nashorn engine is needed to perform this because java is statically typed and therefore cannot create runnable code dynamically. The function returned by the engine is a java lambda function and can therefore use java commands, for example “x.toUpperCase()” will change all lines to upper case.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tested feature | Test | Input | Output | Result |
| Open | Opening a file | Line.java | Appendix 3.4 – 1 | The contents of the file are opened in a new windo |
| Open | Cancelling the file chooser | Cancel is clicked |  | Nothing happens |
| Save | A change is made and then the file is saved, the file is then opened in another text editor to ensure the change is made. | Line.java | Appendix 3.4 – 2 | A popup informs the file has been saved, opening in notepad shows the changes have been made |
| Undo | A change is made and then undo pressed | A new file  Several changes are entered  Undo is then pressed |  | When undo is pressed, the last change is removed |
| Undo | A large amount of text is pasted in and then undo is pressed | The text from line.java is pasted in | Appendix 3.4 – 3  Appendix 3.4 – 4 | All text pasted in is removed |
| Undo | No change is made and undo is pressed | No input |  | Nothing happens |
| Redo | A change is made, it is undone, it is then redone | A new file  Several changes are entered  Several changes are undone  Redo is then pressed |  | The change to the text reappears |
| Redo | A large amount of text is pasted in it is undone, it is then redone | The text from line.java is pasted in, undone and then redo is pressed | Appendix 3.4 – 5 | All text pasted reappears |
| Redo | No undo is made and redo is pressed | Undo is pressed |  | Nothing happens |
| Find and replace | A file is opened, and a common string is replaced with another | Line.java  Public is replaced with foo | Appendix 3.4 – 6 | All instances of public are replaced with foo |
| Find and replace | A file is opened, and a common string is replaced with nothing. | Line.java  Public is replaced with nothing | Appendix 3.5 – 7 | All instances of public are removed |
| Map | A string that is not a function is entered | foo |  | An error is returned |
| Map | A function is entered | List.java  x + “foo” | Appendix 3.6 – 8 | Foo is appended to every line |
| Map | Nothing is entered |  |  | An error is returned |

## 3.5 Resource Files

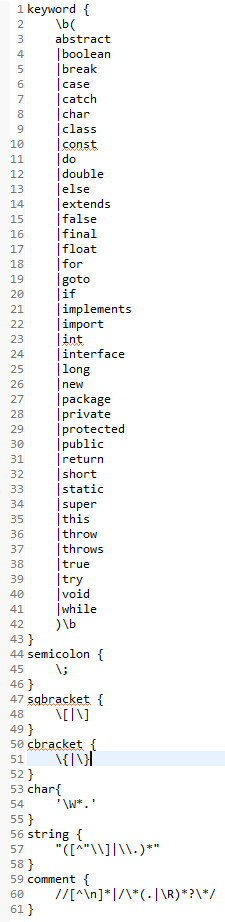


Figure 3.5 – 1

The config.txt and default.css contain all of the customisable elements of the program. Config.txt stores the height and with of the window, these are measured in pixels and when changed the size of all components scales to match. The syntax highlighter refers to the current files being used to highlight the text java and CSS syntaxes have been included, however these files can easily be created or modified to work with other languages. The syntax highlighter option can also be set to auto, using the file extension to open the highlighter files.

Default.css contains two tags that will occur in every file. The text tag refers to all text and is an easy way to change the font size of the application. The find tag refers to the string currently being found, as default it is set to red and bold, however these can be changed to the users ideal settings.

Commands.txt contains details of all commands that can be used in the terminal interface. The file is printed out when the help command is called, this could all be stored inside of a print statement as a long string, however this would clutter up the code and make adding information about new functions difficult. This file is not meant to be edited by the user.

Each languages syntax highlighter has two sets of files: pattern files and CSS files. Pattern files contain labels relating to the CSS tags and java regular expressions. Each appearance of the regex receives its relevant CSS tag. To read each tag and regex in properly, the program reads in each line, any line ending in a { becomes the tag for the following regex, all code between the open brace and the close brace becomes the regex. Because all lines are trimmed and concatenated multiple lines can be used for the same regex, this can help make the file easier to read. Figure 3.5 – 1 shows java.txt each regex is separated into its CSS tag, the keyword regex takes up multiple lines for ease of understanding.

CSS files simply contain the same tags with formatting CSS, both JavaFX and RichTextFX allow for CSS files to be applied. Java.css has been created using a similar highlighting style to eclipse.

Two pairs of syntax files have been created, java.css and java.txt show how the highlighting feature work, and provide full highlighting of the java language. css.css and css.txt have been made to show switching between languages using the auto feature, they are therefore not as comprehensive and although they do provide reasonable syntax highlighting for CSS future versions will improve upon these files.

# 4 Conclusion

The product of this project was a useful text editor with functions that help both editing text and code. I am unsure of any other text editor that allows for both use of a graphical interface and for terminal commands to be entered. I will undoubtedly use the text editor in future, particularly for the map function which allows for repetitive changes to be applied to a large amount of text automatically. The mapping feature was used to help change the text from commands.txt into a string by using the command “” \”” + x + “\”+”;”, this string was removed and placed in a file to be read to make the code easier to understand.

Bracket matching and multiline selection were both initially planned to be included in the project. Despite being useful features, it is still every easy to edit code without them. These features may make it in to future versions, however other more exciting features will be added first. Because I believe the terminal interface is what makes this text editor superior to other text editors future additions will most likely be expanding the terminal commands. I have mentioned changes in the report that I would like to add. I am particularly excited to code in macro commands, allowing commands to be assigned to keys.

No software development method was chosen to develop the project, instead code was separated into classes. This was particularly useful when changing the view from initially a swing text area to a javafx text area finally to a richtextfx code area. In each of these changes a new class was written and swapped into the java code.

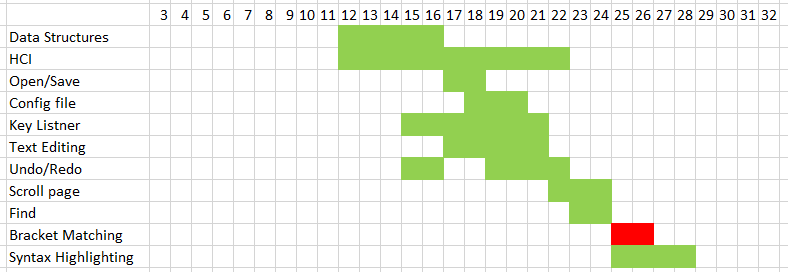


Figure 4 – 1

In the project contract a gantt chart is given, around week 20 it was clear that this gantt chart was unachievable and left too long for the report and very little time for the development of each feature, for this reason a new gantt chart was created. Figure 4 – 1 shows the new updated gantt chart, and the timings each section was started and completed in weeks at the top.

If the project were to be redone, and a consideration for future projects, I would research more into text area alternatives sooner, a large amount of time was spent changing from text area to text area. If one solution was properly researched and found early on in the project this could have saved me a lot of time, and perhaps the timings of the initial gantt chart could have been kept to.

# 5 Tools Used

Eclipse Oxygen

Eclipse oxygen is the Java IDE that was used for this project.

Availiable at: <https://www.eclipse.org/downloads/download.php?file=/oomph/epp/oxygen/R2/eclipse-inst-win64.exe>

RichTextFX

RichtextFX is a powerful tool for allowing rich text to be displayed using javafx

<https://github.com/FXMisc/RichTextFX>

# 6 Appendicies

Appendix 3.1 – 1

**package** Test;

**import** **static** org.junit.Assert.\*;

**import** java.util.Optional;

**import** java.util.stream.Stream;

**import** org.junit.Test;

**import** Lib.Line;

**public** **class** LineTest {

Line head1 = **new** Line("this is the first line");

Line head2 = head1.add(1, **new** Line("this is a second line"));

Line head3 = head2.remove(0);

Line head4 = head2.edit(1, **new** Line("this is a different second line"));

@Test

**public** **void** lenTest() {

*assertEquals*("Length is not 1 when instanciated", 1, head1.len());

}

@Test

**public** **void** addTest() {

*assertEquals*(2, head2.len());

}

@Test

**public** **void** removeTest() {

*assertEquals*(1, head3.len());

*assertEquals*("this is a second line", head3.getContent(0));

}

@Test

**public** **void** editTest() {

*assertEquals*(2, head4.len());

*assertEquals*("this is a different second line", head4.getContent(1));

}

@Test

**public** **void** bigTest() {

Optional<Line> head5 = Stream.*iterate*(0, x -> x + 1)

.limit(100)

.map(x -> "index number " + x)

.map(x -> **new** Line(x))

.reduce((x,y) -> x.add(x.len(), y));

//test its all set up correctly

*assertEquals*("index number 1", head5.get().getContent(1));

*assertEquals*("index number 50", head5.get().getContent(50));

*assertEquals*("index number 99", head5.get().getContent(99));

*assertEquals*(100, head5.get().len());

//test removing 3 lines

Line head6 = head5.get().remove(30).remove(60).remove(90);

*assertEquals*("index number 1", head6.getContent(1));

*assertEquals*("index number 50", head6.getContent(49));

*assertEquals*("index number 99", head6.getContent(96));

*assertEquals*(97, head6.len());

//test adding a line

Line head7 = head5.get().add(50, **new** Line("Hello"));

*assertEquals*("index number 1", head7.getContent(1));

*assertEquals*("Hello", head7.getContent(50));

*assertEquals*("index number 99", head7.getContent(100));

*assertEquals*(101, head7.len());

//check head 5 is not changed

*assertEquals*("index number 1", head5.get().getContent(1));

*assertEquals*("index number 50", head5.get().getContent(50));

*assertEquals*("index number 99", head5.get().getContent(99));

*assertEquals*(100, head5.get().len());

}

}

Appendix 3.1 – 2

**package** Test;

**import** **static** org.junit.Assert.\*;

**import** java.util.stream.Stream;

**import** org.junit.Test;

**import** Lib.LineList;

**public** **class** LineListTest {

@Test

**public** **void** loadTest() {

LineList test1 = **new** LineList();

test1.reset();

Stream.*iterate*(0, x -> x + 1)

.limit(500)

.map(x -> "index number " + x)

.forEachOrdered(x -> test1.load(x));

*assertEquals*(500, test1.len());

*assertEquals*("index number 0", test1.getLine(000));

*assertEquals*("index number 300", test1.getLine(300));

*assertEquals*("index number 499", test1.getLine(499));

}

@Test

**public** **void** addTest() {

LineList test2 = **new** LineList();

test2.reset();

Stream.*iterate*(0, x -> x + 1)

.limit(500)

.map(x -> "index number " + x)

.forEachOrdered(x -> test2.load(x));

test2.add(0, "This is an additional line");

test2.add(250, "This is another additional line");

test2.add(400, "This is a third additional line");

*assertEquals*(503, test2.len());

*assertEquals*("This is an additional line", test2.getLine(0));

*assertEquals*("index number 100", test2.getLine(101));

*assertEquals*("index number 300", test2.getLine(302));

*assertEquals*("This is another additional line", test2.getLine(250));

*assertEquals*("index number 499", test2.getLine(502));

*assertEquals*("This is a third additional line", test2.getLine(400));

}

@Test

**public** **void** removeTest() {

LineList test3 = **new** LineList();

test3.reset();

Stream.*iterate*(0, x -> x + 1)

.limit(500)

.map(x -> "index number " + x)

.forEachOrdered(x -> test3.load(x));

test3.remove(0);

test3.remove(200);

test3.remove(400);

*assertEquals*(497, test3.len());

*assertEquals*("index number 1", test3.getLine(0));

*assertEquals*("index number 300", test3.getLine(298));

*assertEquals*("index number 499", test3.getLine(496));

}

@Test

**public** **void** undoTest() {

LineList test5 = **new** LineList();

test5.reset();

Stream.*iterate*(0, x -> x + 1)

.limit(500)

.map(x -> "index number " + x)

.forEachOrdered(x -> test5.load(x));

test5.prep();

test5.add(100, "This is an additional line");

test5.prep();

test5.add(250, "This is another additional line");

*assertEquals*(502, test5.len());

*assertEquals*("This is an additional line", test5.getLine(100));

*assertEquals*("This is another additional line", test5.getLine(250));

test5.undo();

*assertEquals*(501, test5.len());

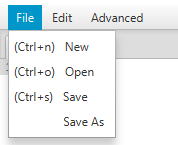
*assertEquals*("This is an additional line", test5.getLine(100));

*assertEquals*("index number 250", test5.getLine(251));

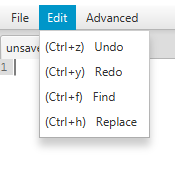
}

}

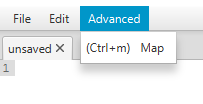
Appendix 3.2 – 1



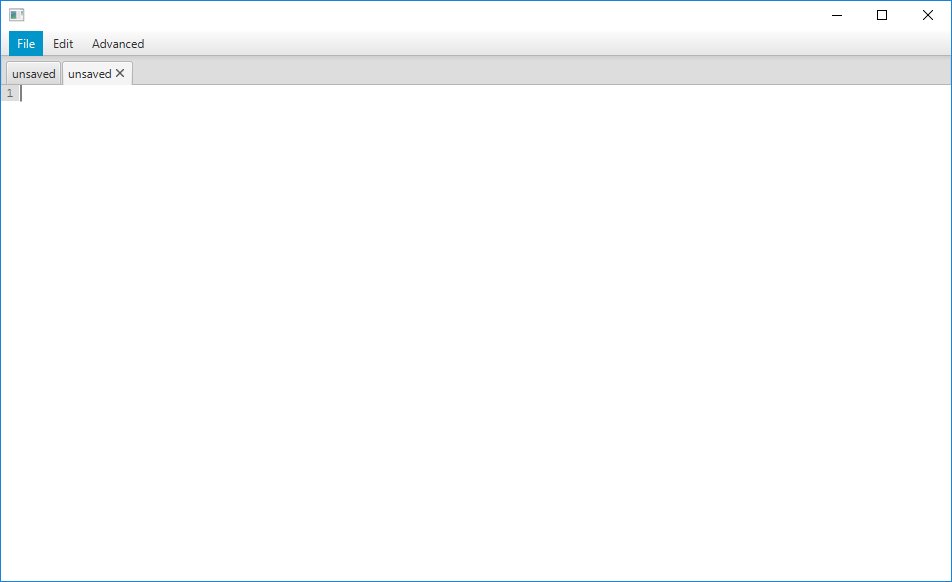
Appendix 3.2 – 2

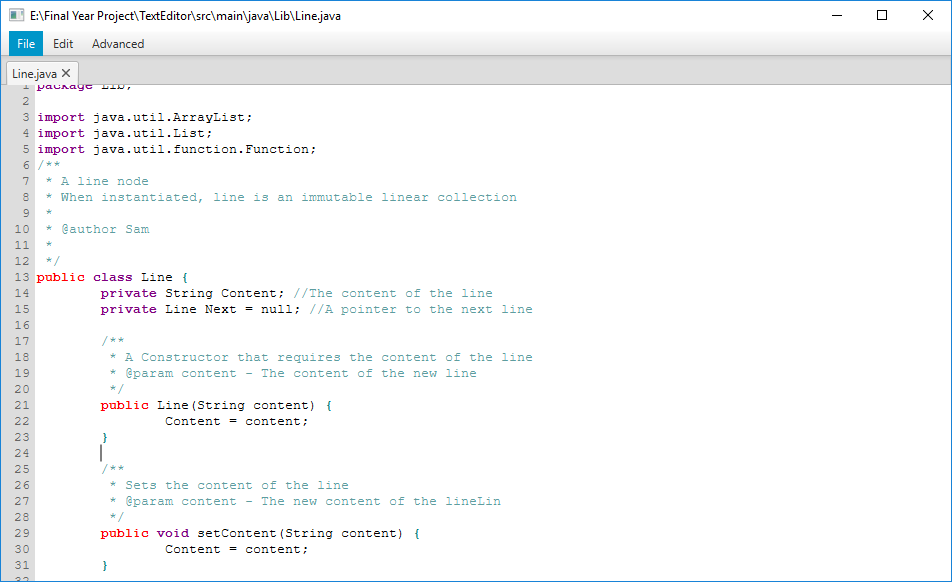


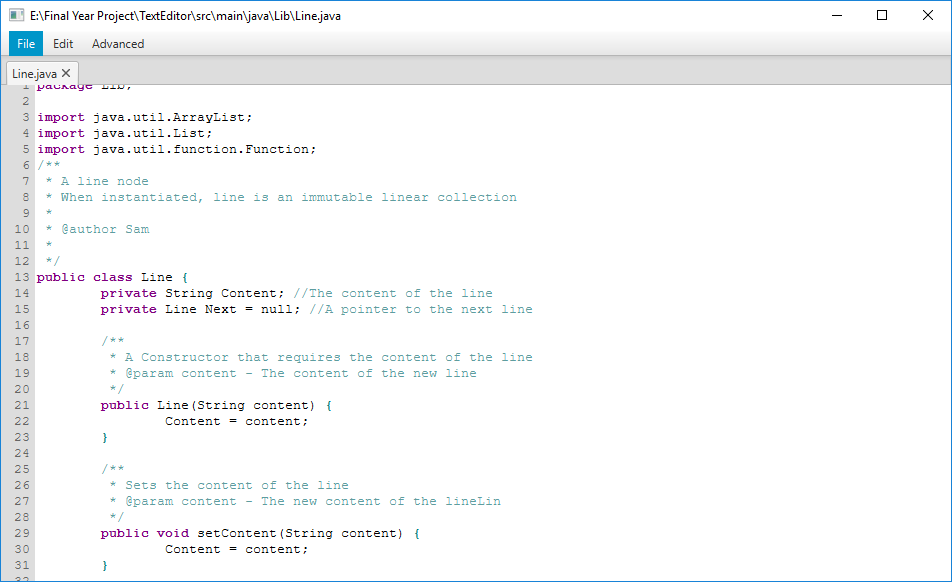
Appendix 3.2 – 3



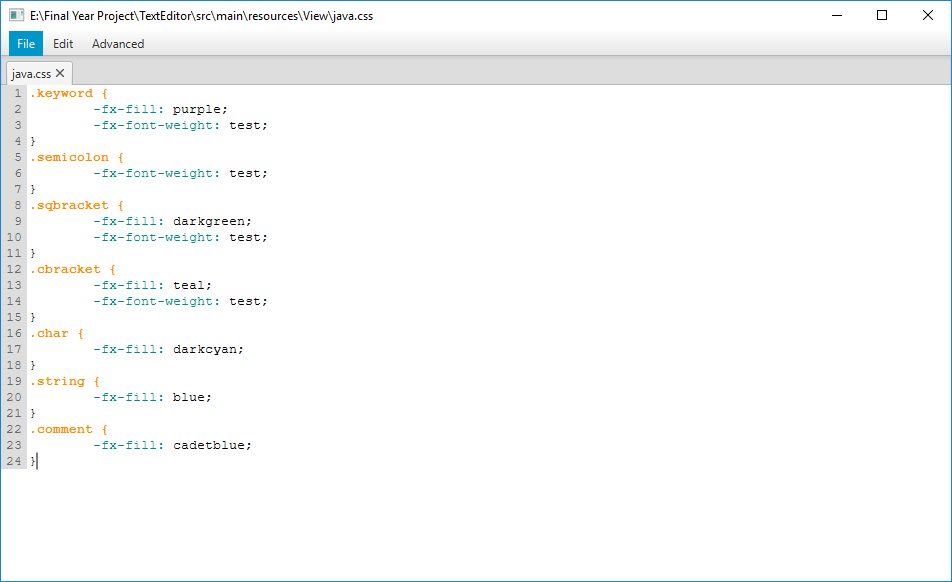
Appendix 3.2 – 4



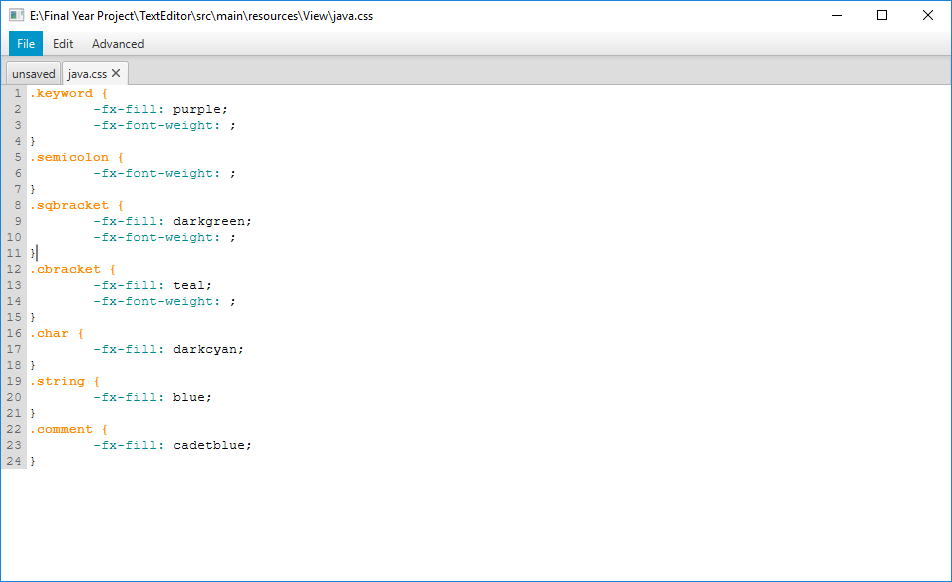
Appendix 3.2 – 5 

Appendix 3.2 – 6

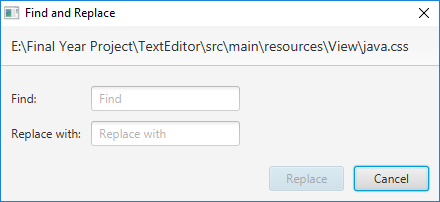
Appendix 3.2 – 7



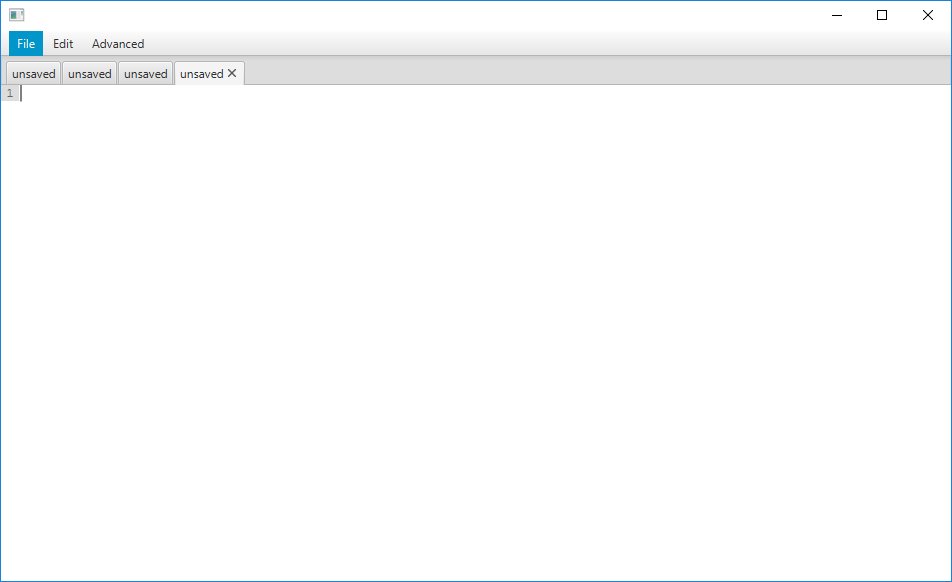
Appendix 3.2 – 8



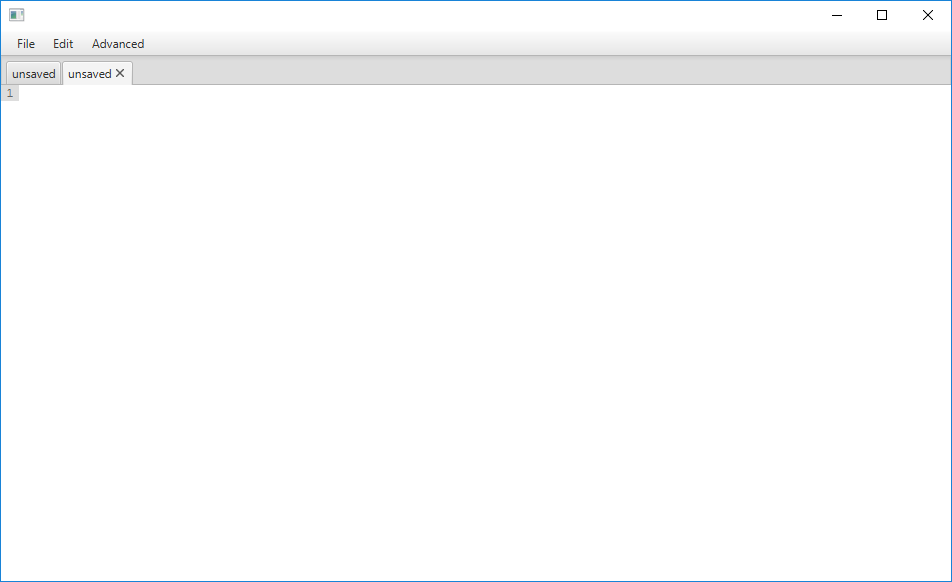
Appendix 3.2 – 9



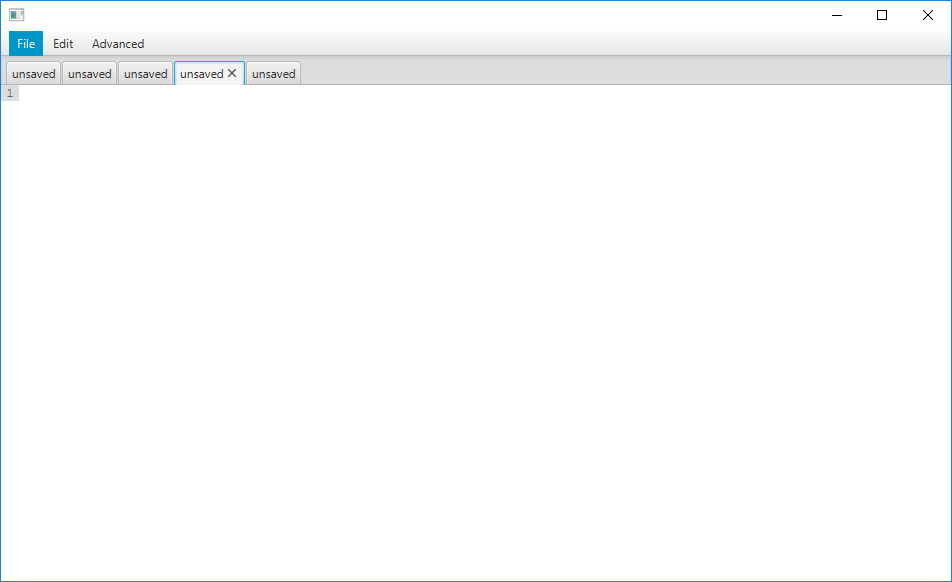
Appendix 3.3 – 1



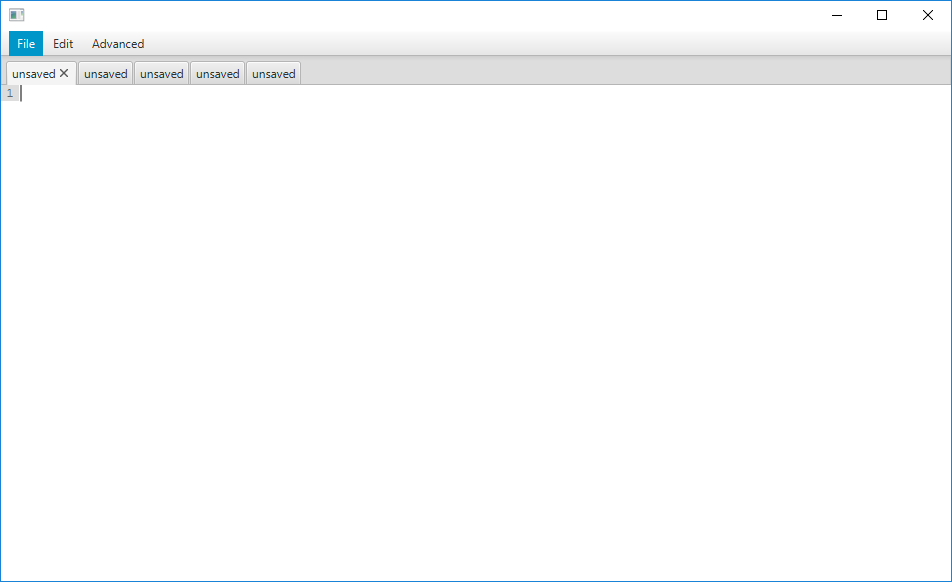
Appendix 3.3 – 2



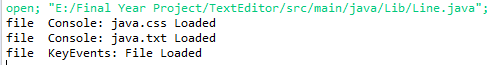
Appendix 3.3 – 3



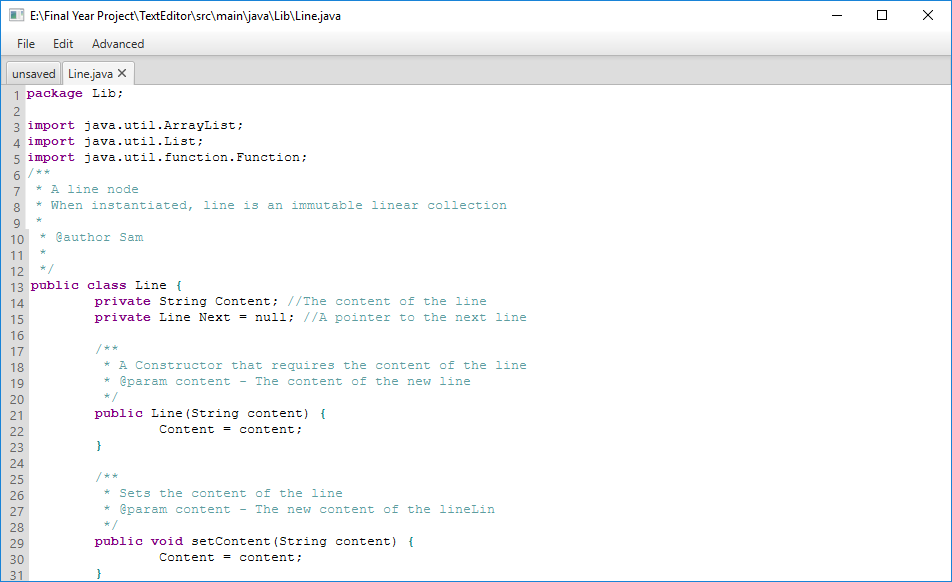
Appendix 3.3 – 4



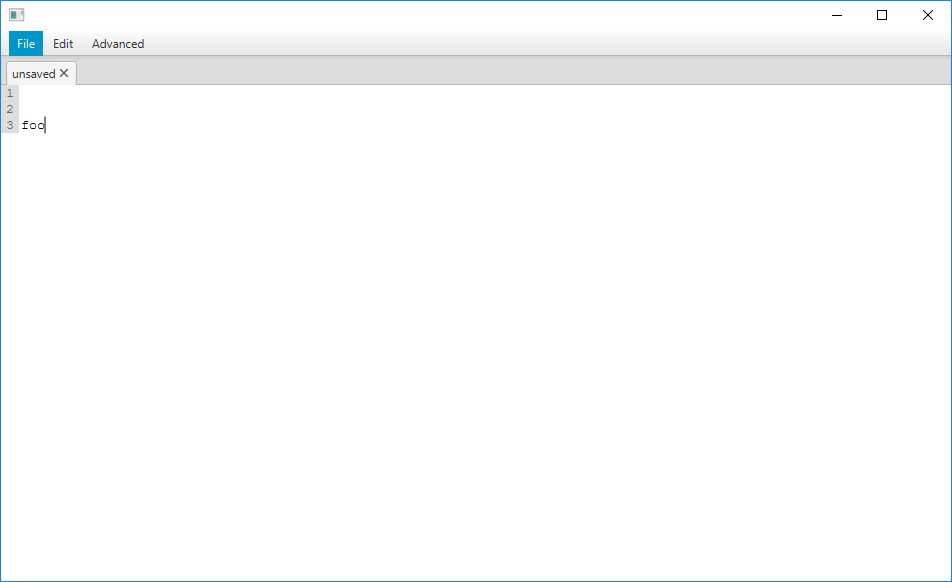
Appendix 3.3 – 5

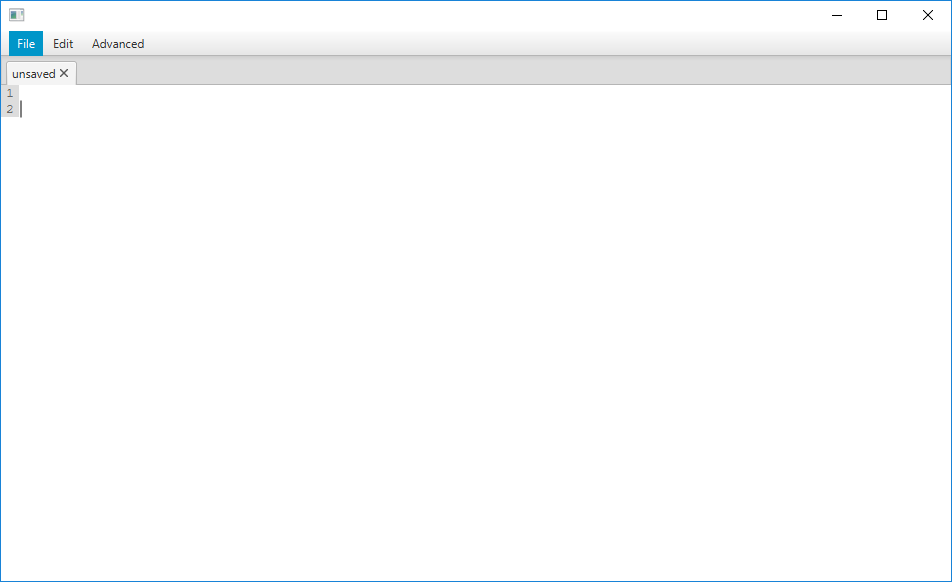


Appendix 3.3 – 6

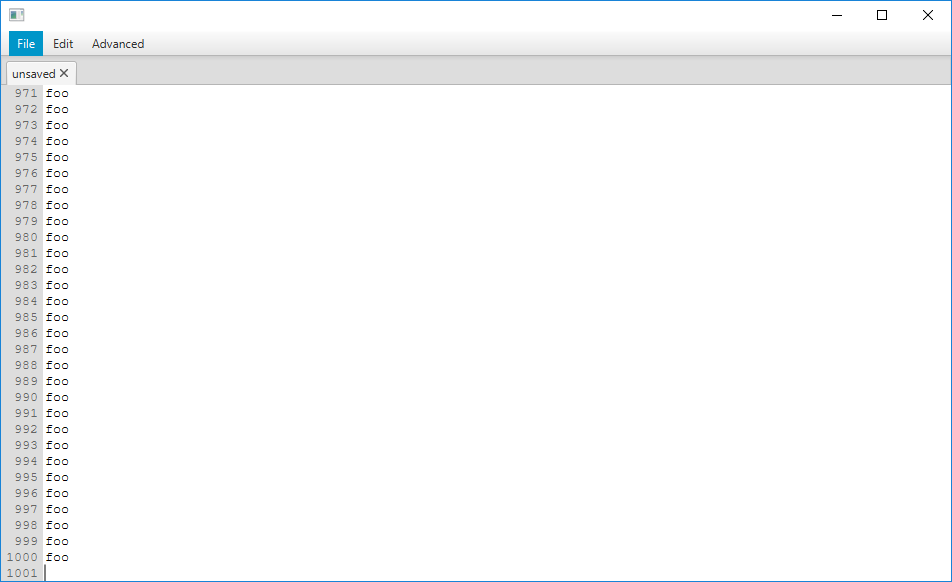


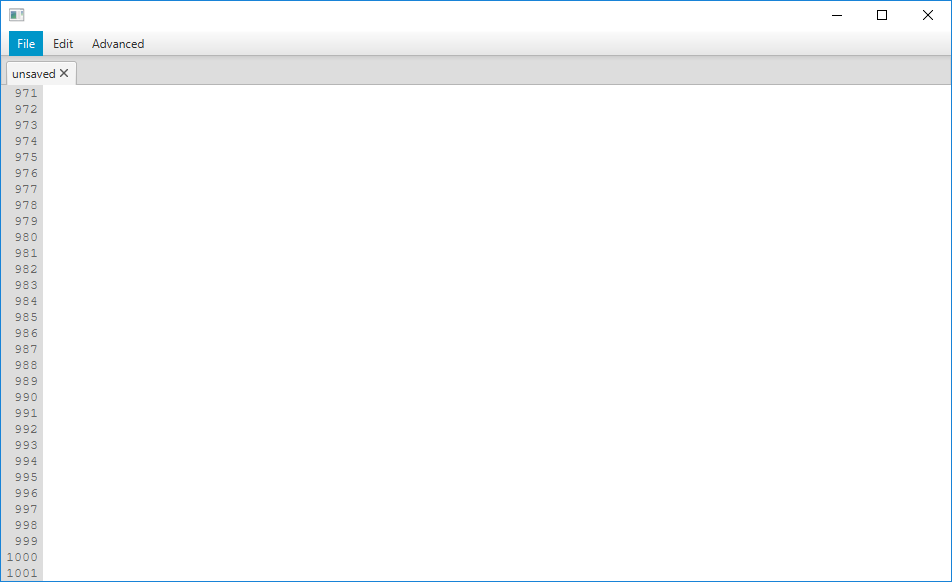
Appendix 3.3 – 7



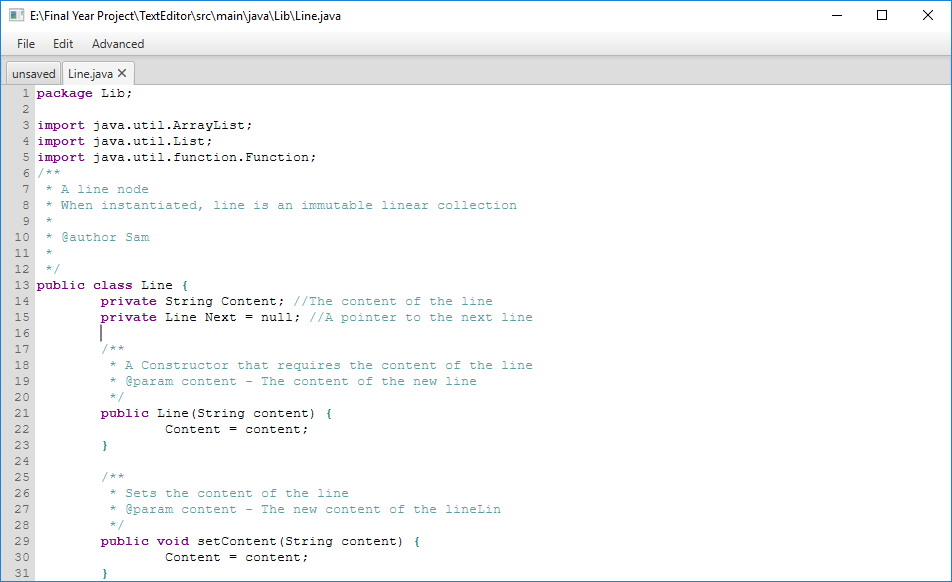
Appendix 3.3 – 8

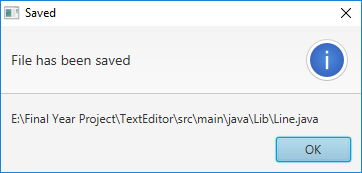
Appendix 3.3 – 9



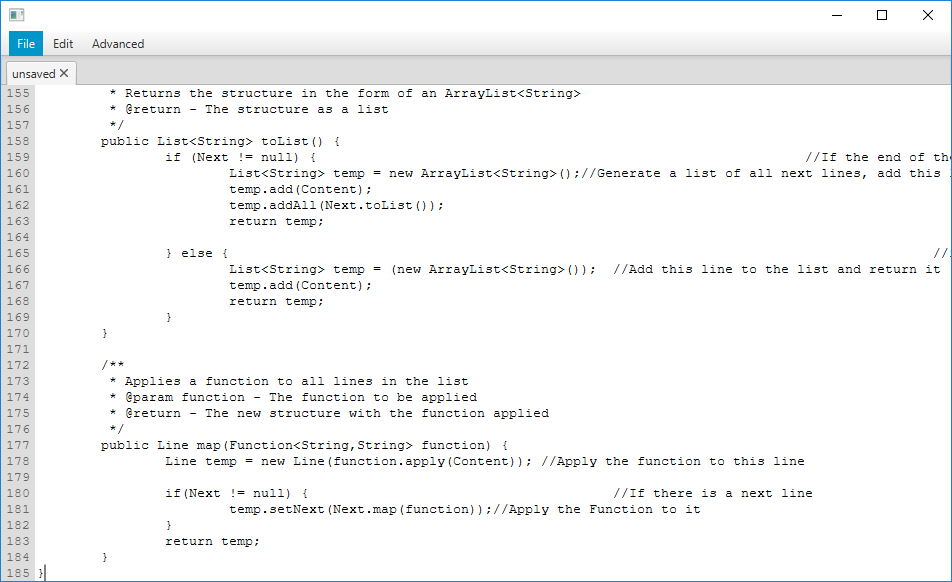
Appendix 3.3 – 10 

Appendix 3.3 – 1

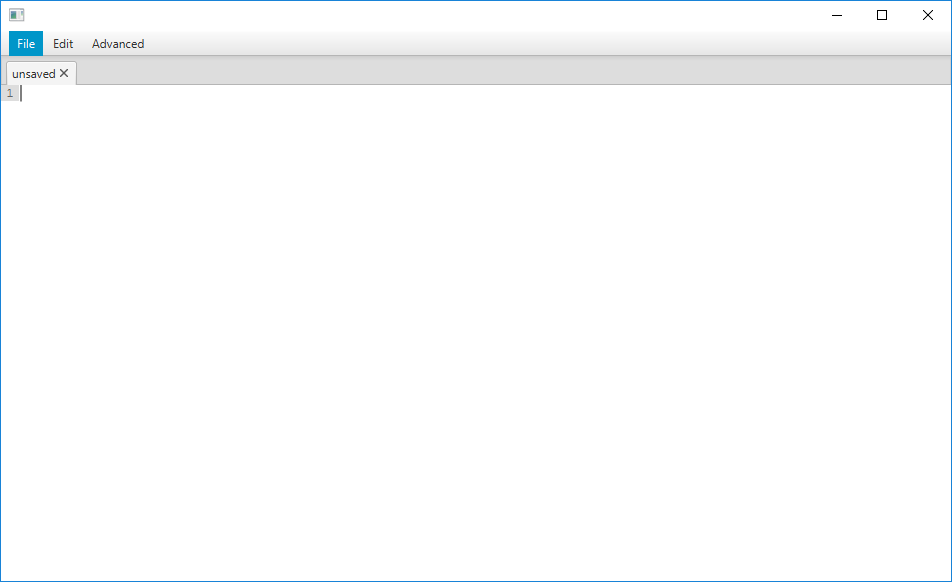


Appendix 3.4 – 2

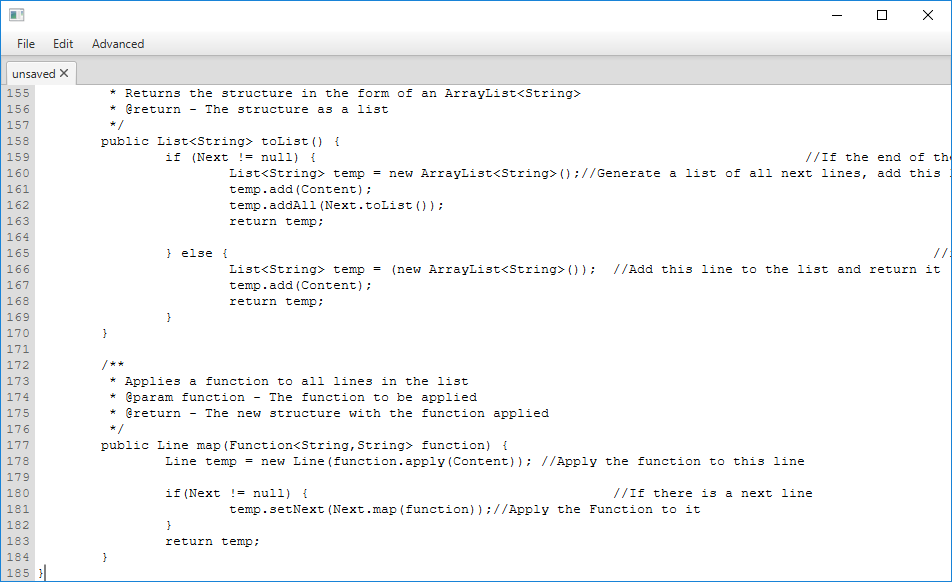
Appendix 3.4 – 3



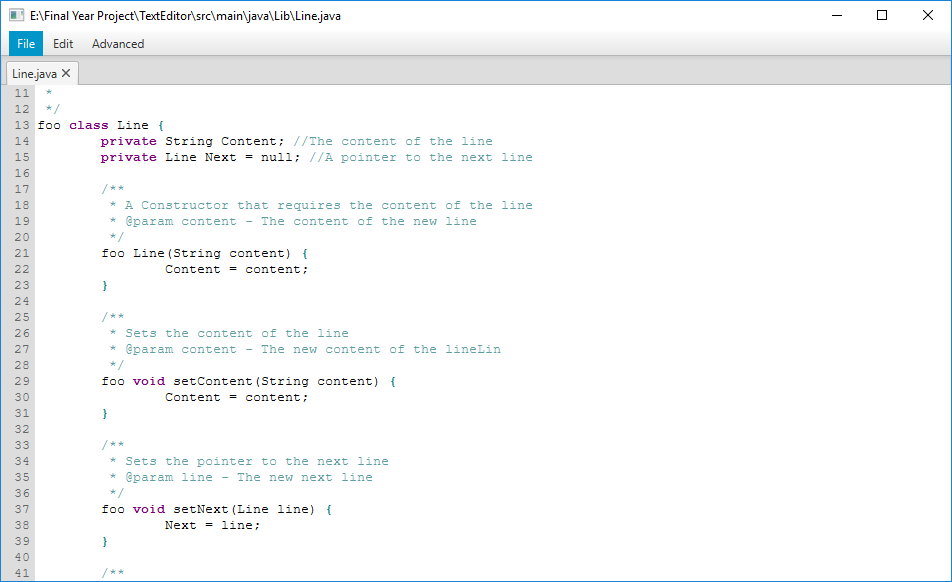
Appendix 3.4 – 4



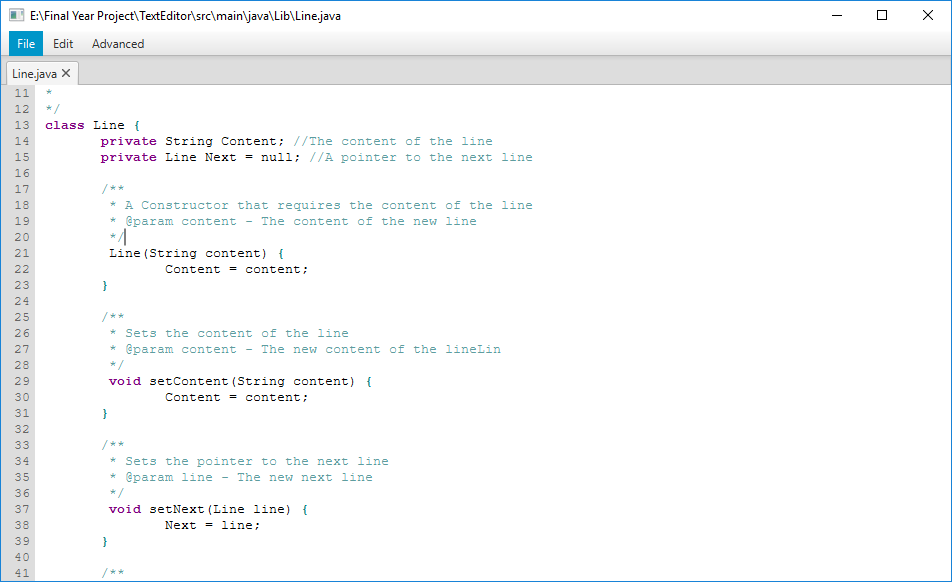
Appendix 3.4 – 5



Appendix 3.4 – 6



Appendix 3.5 – 7



Appendix 3.6 – 8

