University of Stirling
Computing Science and Mathematics
ITNP001 Principles and Practice of Programming
Assignment 2 Autumn 2015
A Simple Text Editor

Submission deadline: 6pm on Monday 30th November 2015

This assignment is worth 30% of the final mark for the module.

Independent work

Note that we require that the work you submit is your own, independent work. "Plagiarism" means presenting the work of others as your own. The University takes a very serious view of plagiarism, and the penalties can be severe.
Work which is submitted for assessment must be your own work. All students should note that the University has a formal policy on plagiarism which can be found at
`http://www.stir.ac.uk/academicpolicy/handbook/assessment/#q-8`
However, this does allow discussion of general issues. Please read the 'Note on Independent Work' available via the ITNP001 home page, assignments link. Be warned: copying or borrowing code does not necessarily guarantee a quality product, and may damage your grade. Also, if you show or give your code to someone else then you may be penalized. We will be submitting all the solutions to a "similarity" checking service, and **any apparent cases of collusion will be investigated carefully.**

1 Introduction

The goal of this assignment is to complete a simple text editor. You should familiarise yourself with the lecture material on *String*s before beginning. Reading the corresponding parts of the text-book would also be a good idea. Review the *Directory* practical as it does some similar things.

The automated system for distributing and collecting assignments will be used. On 20th November 2015, a new folder called *ITNP001\Assignment2* will be created in your home folder. You will see that the folder contains *Editor.java* and *EditorGUI.java* (both finished classes) and *Formatter.java* (an incomplete class that handles text). *Editor.java* is the main class.

JavaDoc comments have been supplied. You can see these in BlueJ by choosing 'Documentation' instead of 'Source Code' in the editor. Alternatively, you can use a web browser to open *Editor.html*, *EditorGUI.html* and *Formatter.html* in the *doc* folder. This will let you get an overview of the code without having to read the detail.

The text editor resembles programs like NotePad that you are already familiar with. However, it does not open or save files – all text must be typed in afresh, and will be lost when the program closes. It is suggested that you use NotePad to create and save your own sample text file. You can then open this file, and copy-and-paste the text into your own editor while testing it. A paragraph can have multiple lines, so paragraphs are separated by a blank line.

Build a new Non BlueJ project in the folder that you have been delivered and look at *Editor.java* and *EditorGUI.java*. You should not change these as they are already complete. *EditorGUI.java* creates the application window with the main text area and the buttons for text operations. Since the *formatter* object needs to make use of the main text area, it is given this as a parameter when the *Formatter* constructor is called.

Now look at *Formatter.java*. This is fairly long, but some of the code has already been written for you. You will need to understand the methods that have already been written since you will need to use them. Some of the methods are incomplete, and comments marked '***' suggest how you should complete the code – see below.

There are several steps to completing the program, implementing the six editing functions given in the next section. The different functions contribute the given %age of the overall mark for the assignment, and the Assignment Criteria section on page 5 gives more information about the marking.

2 Approach

The aim is to provide the following text editing functions – details are given in the next section:

Clear: (20%) deletes all text from the text area.

Lines: (20%) reports a line count for the text area.

Words: (15%) reports a word count for the text area.

Find: (15%) is used to find the next instance of the text entered into the search box. *The search starts at the current position in the text* (i.e. wherever you click to set the cursor). If you call Find repeatedly, it will find the next instance of the text (or say that no further instance can be found).

Replace: (15%) is similar. If the text is found, it is replaced by the text in the replace box.

Tidy: (15%) is used to clean up the text layout.

As you complete each method, check that it works before continuing.

For debugging, remember that you can use the BlueJ debugger to pause the program, view the current values of variables, and step through.

Alternatively, there is a *debug* method that you can call with a string to be printed in the console window. You might find it helpful to debug your program by including temporary statements like:

```
debug("Text position is " + textPosition);
```

This will let you see what your program is doing as it runs. Remove (or comment out) the debug output when you are sure your program is behaving correctly.

3 The details

You are recommended to work through the following steps methodically. Only *Formatter.java* needs to be changed. Go through the steps little by little and you will get there! The amount of programming needed to complete this stage is less than a page. However, do not sit at the computer and try to write code blindly from the suggestions that follow. Think out what you want to write *before* you go to the lab. You can waste a lot of time at the computer trying to write code that should be thought about offline. In fact, if you get stuck it is good practice to print your code and study it in peace and quiet. Trying to fix it at the computer can be counter-productive.

In a few places you are asked to look up the Java library documentation to find out about methods you have not used before. Start by looking up the *JTextArea* class, in particular the *getText* and *setText* methods.

1. Begin with the *clearText* method (called when the Clear button is clicked). Set the contents of the text area to an empty string. Type (or paste) in some text and click Clear. Does it disappear?

2. Now turn your attention to the *countLines* method (Lines button). A line is considered to be one or more characters ending with a newline (or the end of text). As you loop through the characters, the variable *lineStart* should be set to the position of the character at the start of a line.

   In *countLines*, call the *getText* method (in *Formatter*) to get the contents of the text area into *textWords*. Loop through the text one character at a time, using *charAt* to extract the character at each position. Check if the character is a newline (i.e. '\n'). If so, check how many characters are in the current line by subtracting *lineStart* from the loop counter. If this is greater than zero, increment the line count. If this is zero (i.e. there is nothing in the line) then the line count is not incremented. Irrespective of this check, set *lineStart* to the loop counter plus 1 (i.e. the start of the new line).

   After the loop ends, check if the text ended with a newline (use method *endsWith* from the *String* class). If so, and the last line has characters, then increment the line count.

   Start with an empty text area. Does Lines report 0? Type (or paste) in a few lines of text and check the line count. Try this if there are blank lines, and check if the count is right when the last piece of text does not end with a newline.

3. The *countWords* method (Words button) is roughly similar. The program should use a simple concept for a word, i.e. any text that appears between spaces. This will treat 'trivial' as a single word, and also words with punctuation such as 'non-trivial' and 'ending;'. Other things like '1248' and '!*=?' will also be treated as words. *That is fine.* A small complication is that there might be several spaces on either side of a word. Indeed, other forms of 'white space' (such as tabs and newlines) might appear between words.

   Use the *getText* method to get the contents of the text area into *textWords*. Loop through *textWords* one character at a time. Check if the character is a white space. (Look up the Java documentation for class *Character*, method *isWhitespace*.)

   Your code should keep track of whether it is currently inside a word or not (variable *inWord*). If you are inside a word when you find a space, then you just left the word. If you are not in a word and the current character is not a space, then you just entered a word; increment the word count at this point.

   Start with an empty text area. Does Words report 0? Type (or paste) in a few lines of text and check the word count. Try adding extra spaces (and tabs) at the start and end of the text, as well as between words. Is the word count the same?

4. The *findText* method (Find button) should start by checking the length of the search text. If it is zero, there is nothing useful to search for. If it is non-zero, then call *getText* in *Formatter*. Apart from setting *textWords*, this method also notes the current position in the text (in *textPosition*). Look for the search text in *textWords* (use *indexOf* with *textPosition* as the starting point). If the search text is found, you need to highlight it by calculating where the search text starts (i.e. the result of *indexOf*) and where it ends (which depends on the length of the search text). Give these values as parameters to the *setSelection* method in *Formatter*. Then update *textPosition* to the end of the found text (to make sure that a new instance is found next time).

   Type (or paste) in some lines of text with some repeated words. Enter the word you wish to search for. Click at the start of the text to set the current position. Does Find report and highlight the word you were looking for? If you keep clicking Find, are later instances reported? Does it tell you if you search for an empty word, of if the word cannot be found?

5. The *replaceText* method (Replace button) is roughly similar. Initially you should call *findText*. If this reports the search text is found, proceed to replace it. This is a little tricky because *findText* will already have moved *textPosition* to after the word. The start of the word to be replaced is

this position less the length of the search text. Now construct the *new contents* of the text area as follows: everything from the start up to the beginning of the text you found (use *substring*); then the replacement text; then everything from the end of the text you found (use *substring*). Finally, call *setSelection* to highlight the replaced word.

Type (or paste) in some lines of text with some repeated words. Enter the word you wish to search for, and what you wish to replace it with. Click at the start of the text to set the current position. Does Replace find the word you were looking for and replace it? If you keep clicking Replace, are later instances replaced? Does it tell you if you search for an empty word, of if the word cannot be found?

6. The last task in this stage is to make *tidyText* work (Tidy button). As you edit text, it might end up a bit messy (e.g. extra spaces and newlines). The goal of tidying up the text is to fix these problems.

   Although you don't have to use them, Java provides some convenient methods that you are recommended to use. Look up class *String*, methods *replaceAll* and *replaceFirst*. You will see that these work with regular expressions – patterns that define what to search for. You can replace a pattern by something else (an empty replacement string means it should be deleted). Regular expression patterns can be very complex, so you need to know only some special characters. A caret ('^') means the start of a string, a dollar ('$') means the end of a string, and '\n' means a newline. For one or more instances of something, append '+' (e.g. '\n+' to mean one or more newlines, '\n\n+' to mean two or more).

   Start by getting the text in the main area into *textWords*. Then apply a number of replacements: change one or more consecutive spaces into a single space, remove an initial or final space, remove a space at the start or end of a line. Three or more newlines should be turned into two (so that exactly one blank line appears between paragraphs). If the text does not end with a newline, append one. Finally, set the text in the main area to the updated value of *textWords*. *[You should add these steps one at a time, testing carefully as you go.]*

   Type (or paste) in several paragraphs, each with several lines of text. Make sure your text needs various kinds of tidying (e.g. it might have several spaces between words, and several blank lines between paragraphs). Check systematically that all the aspects to be tidied are handled properly.

4 Submission

This assignment is mandatory and counts for 30% of the final marks. You are required to leave the files which you wish to submit in the *ITNP001\Assignment2* folder, with *exactly the same names* as they had originally. You may work with these files all the time, or may copy your work to these files just before the deadline. It is *your* responsibility to make sure your current work is available on **30th November 2015 at 6pm**. At this point, the folder will be copied for marking. Any other files in the folder will also be copied, but ignored if not relevant. Please make sure that what you submit *compiles* properly, as non-compiling code cannot be tested. You can comment out any code that you have tried but which does not compile.

It follows that you do not need to submit any paper at all, or to take any action other than making sure that on the submission day the files contain what they ought to contain, and are in your `ITNP001\Assignment2` folder with the correct names.

The automatic collection system is designed to contain safeguards against mistakes. You should receive email messages

1. When the files are set up in your file space.
2. On each of the last few days leading up to the submission date.

3. After the files have been “collected” on the submission date.

It is up to you to check these mail messages. If they do not arrive at all, or they contain warnings or messages which you do not understand, then *contact Simon Jones (email* `sbj@cs.stir.ac.uk`*) at once.*

5 Assessment criteria

We shall be assessing your work with respect to various criteria, including:

- correctness of operation,
- appropriate use of Java constructs,
- suitably informative choices of variable and method names,
- consistency, legibility and tidiness of program layout,
- effective use of comment text.

6 Keeping backups of your work

I **very strongly recommend** that you make backup copies of your work from time to time — perhaps after each successfully completed development step, or more frequently such as every day, or even more frequently. It is easy to do this: On the desktop, open your `home\ITNP001` folder, click *once* on the `Assignment2` folder icon to highlight it, select **Copy** from the **Edit** menu, and then select **Paste** from the **Edit** menu. You will see a new folder appear called `Copy of Assignment2` (the next will be `Copy (2)...`, etc for subsequent copies). If you ever need to backtrack, you can just copy your `.java` file from the most recent `Copy...` folder to the main `Assignment2` folder. I suggest that you treat the `Copy ...` folders as your *safe archive — leave them alone and continue your work in the main* `Assignment2` *folder*.
Keeping regular print-outs of your work can also be a life-saver.
In past years a number of students have found themselves in extremely distressing circumstances as a result of ignoring the above advice.

Simon Jones November 2015