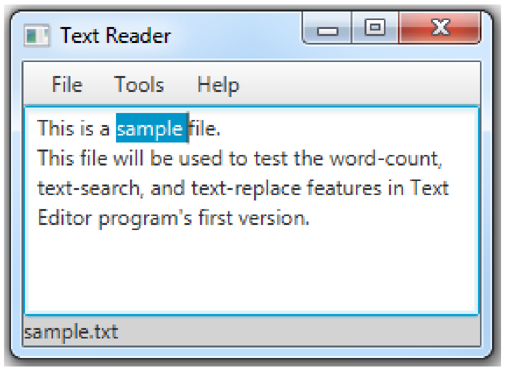
# 95-807 HW3

In HW3 you will deliver the final version of Text Editor. There will be three new functionalities added to the app in this version. But before going there, you need to further tighten up the existing functionality.

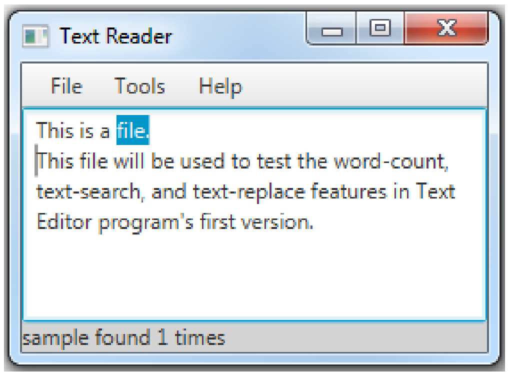
You may have noticed in HW2 version that the text displayed in the fileTextArea is not in synch with what is used for searching and replacing in your program. That is because after loading text from the file into fileContent and displaying it into fileTextArea, they are never synched up again. So as shown in Fig. 1 and 2, although I deleted the word 'sample' from fileTextArea, when I use the Search tool to find 'sample', it says 'sample' is found at the position where it initially was. That is because the search method is still looking at old fileContent while the fileTextArea has changed. Obviously, that is not right!



This is a sample file.
This file will be used to test the word-count, text-search, and text-replace features in Text Editor program's first version.

sample.txt

*Figure 1: TextEditor v2 text*



This is a file.
This file will be used to test the word-count, text-search, and text-replace features in Text Editor program's first version.

sample found 1 times

*Figure 2: TextEditor v2 fileTextArea not synched with fileContent*

There are two ways to deal with this problem. A painful way is to programmatically update fileContent from fileTextArea before every search / replace / word count etc. An easier way is to use something called as '**property-binding**'. But you will need to understand what that is, and for that I have posted a video along with HW3.

To apply this idea to HW3, I have created a new member variable in TextEditor.java.

```
StringProperty fileContentProperty = new SimpleStringProperty();
```

I have then bound this to the fileTextArea in setScreen() method so that change in one changes the other:

```
fileTextArea.textProperty().bindBidirectional(fileContentProperty);
```

So after reading data from the txt file into a StringBuilder fileContent, you will put it in fileContentProperty as

```
fileContentProperty.setValue(fileContent.toString());
```

Similarly, whenever you need to read the content from fileContentProperty(), you will use `fileContentProperty.getValue().`

You will need to think through this a little more to understand what else needs to change in your program. For example, your FileUtilities methods for search / replace / countWords / countUniqueWords take StringBuilder as an argument. So you will need to create a StringBuilder object from fileContentProperty everytime to pass to any of these methods.

Now the new functionality:

1. Provide a **unique-words count** (under Tools menu): A good strategy is to use a HashSet and add all words to it. Since HashSet doesn't allow duplicates, you will get the number of unique words by taking HashSet's size. But you need to think about what is a word? If we used our old regex \\s+ to split words, then 'sentence' is different from 'sentence,', 'text' is different from 'Text'. That isn't right! So we need to change the way we counted words:
    a. The first problem is about word-separator other than white spaces. It can be solved by simply changing the regex from \\s+ to **"[^a-zA-Z0-9']+"** which translates to anything which is <u>not</u> an alphabet, number, or a single quote. So any of those characters become word-separators. This will also change the total word count. For example, the text in Fig.1 has 24 words if we use the old regex "\\s+" . But if we use the new regex, it has **27 words** because hyphen is now a word-separator.

   b. HashSet<String> will use String's equals() method to determine equality, and that is case-sensitive. So 'text' is different from 'Text'. Use toLowerCase() or toUpperCase() before adding a word to the HashSet. If you add all the words to HashSet<String> in countUniqueWords() method and get its size, you should get **23 unique words** in sample.txt.

2. **Save file** after making changes (under File menu):
   a. In SaveFileHandler, use the same FileChooser class as you did in File – Open option, but this time as fileChooser.showSaveDialog(stage)) so that it opens Save dialog box. This will work as Save As… functionality similar to what we have seen in other applications. Once you get the file name, invoke writeFile() method from FileUtilities class.
   b. FileUtilities has a new method writeFile() that takes fileName and fileContent strings. The return value of this method is a String. It returns "File saved" from within try-catch block of BufferedWriter when the file is successfully written. If it enters catch-block, it should return 'Could not save file' from the catch block. Whatever is returned, the handler updates statusLabel with it.
3. **Bind menus** with file-status as open or close. If you run the jar file, you will see that Tools menu, File-Save, and File-Close menu items start off as disabled. They are enabled when a file is opened, and disabled again when the file the closed. For this, I have used a BooleanProperty data-type – something which also uses the idea of property-binding.

```
BooleanProperty isFileClosed = new SimpleBooleanProperty(false);
```

This variable is 'bound' to three menu options in setScreen() method as:

```
toolsMenu.disableProperty().bind(isFileClosed);
saveFileMenuItem.disableProperty().bind(isFileClosed);
closeFileMenuItem.disableProperty().bind(isFileClosed);
```

As the program starts off with isFileClose as true, these menu options are disabled. **You need to set this flag to false in your OpenFileHandler by saying isFileClose.set(false);** The closeFileMenuItem handler sets this flag to true.

**Instructions**:

1. Download **TextEditor.java**, **TestTextEditor.java**, **image.png**, and **TextEd3.jar**. Run TextEd3.jar to make sure you understand how your app should work.
2. Import the image and java files into a package named **hw3**. Copy your FileUtilities.java from hw2 to hw3 package. The text file sample.txt should already be in your HW folder from HW1/2. If not, download and copy it into the project folder as it is used by Junit test-cases.
3. Fill in TextEditor.java as specified. You will need to paste the handlers from your HW2 TextEditor.java to this one. Change FileUtilities.java as needed. Test FileUtilities.java using TestTextEditor.java. Your GUI handlers need to be tested using scenario-testing approach mentioned above.

Note: When you copy some code from one class to another in Eclipse, it often inadvertently puts some import statements at the top. This may or may not show up as error on your machine, but it surely will on TA's computer! So make sure you check the import statements at the top and remove the unnecessary ones.

**Rubric**:

1. Scenario testing: 50%;
2. Unit-testing: 25%
3. Documentation (5%): Your code should be well-commented. Write your name and Andrew id at the top in each class. Indent your code properly (In Eclipse, press Ctrl-A to select all your code and then Ctrl-I to indent)
4. Code quality (15%): Use of loops, if-else, should be wisely made. Remove all unused variables and libraries.
5. Submission instructions (5%): Zip TextEditor.java and FileUtilities.java files as **AndrewID-HW3.zip**.

NO LATE SUBMISSIONS PLEASE! If you are unable to submit by due date and time, you lose all the points. So please avoid last minute submissions because sometimes Canvas may decide to quit on you! Learn to trust technology only to the extent you should! I will have no way to know what happened so do not take that risk. Submit well before time! And if you still face some issues, send your submission to me via email before due date/time.

Good luck!!