ICT-4361 Homework 6a

Purpose

This exercise will familiarize you with file processing, and provide additional experience in text processing in Java.

In Java Programming Exercise 5 you created the mechanism to understand a template, and to cause substitution to occur in one.

In this exercise, we allow the template to be stored in a file, and for the Property substitutions to do the same.

What to Hand In

Please hand in a listing for each program requested, formatted in an easy-to-read style.

Ensure your name, and the name of the file is available in a comment at the top of the file.

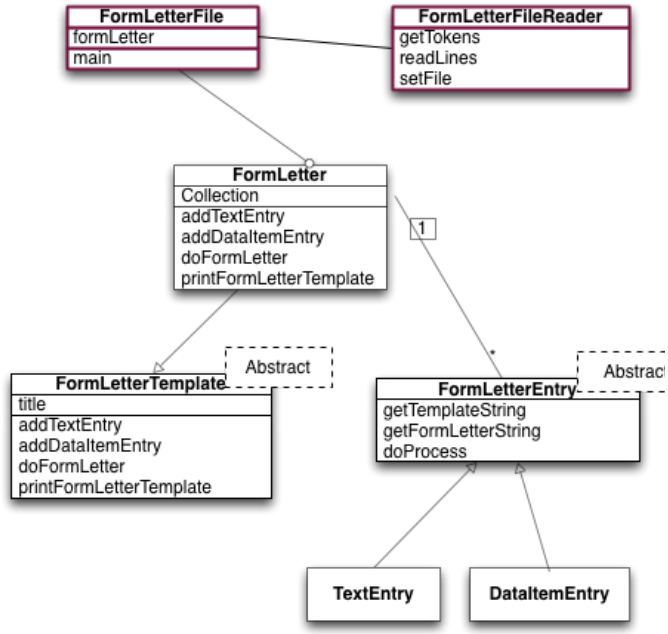
You do not need to submit files from the homework starter files that are unchanged.

Also, ensure that you have a sample of the output from the program.

If your program fails to compile, hand in your error listing as your output.

For electronic submission, "zip" your submission together into a single file, to ensure nothing is missing; for hardcopy submission in a face-to-face class, please ensure your output is neatly formatted and legible.

For each question asked, provide one or two sentences summarizing your answer. Please be both complete and succinct.



Class diagram

Problems

I. Use file processing to have your form letter and data read from the file system.

1. Begin with the classes you developed last week (or adopt last week's instructor sample solution).
2. Create a class called `FormLetterFileReader` which has the following methods:
   - A no-parameter constructor, which simply creates a `FormLetterFileReader`
   - A constructor which takes a file name
   - A `setFile` method which takes a file name, representing the file to read the `FormLetter` contents from
   - A `readLine` method which returns one line read from the opened (and buffered) file
   - A `getTokens` method which returns an array of tokens found on the line. A token is either a buffer of text, or a replacement data item. These data items are recognized by starting with a { and ending with a } .
     - Note that the `FormLetterFileReader` may store the array differently, internally; but it needs to return the result as an array of Strings. The `Strings` will be tokenized and constructed into a `FormLetter` by the `FormLetterFile`.
   - A way to test the class to ensure it works properly (e.g., read a file, and output the resulting tokens). This can be a `main` method, or can be JUnit tests.
3. Create a class called `FormLetterFile` which encapsulates a simple main method (not very different than `FormLetterHello` in many ways):
   - Gets two filenames from the command line or by prompting the user (implement one of the choices)
     - One filename is for the `FormLetter`, and one for the `Properties`.
   - Creates a new `FormLetterFileReader` using this filename as a parameter.
   - Creates a `FormLetter` instance with the filename as the title
     - While it can read a line from the FormLetterFileReader:
     - Break the line into tokens
     - For each token, if it is a simple string (i.e., doesn't begin with a {), add it as a text entry to the FormLetter.
     - Otherwise, add it as a data item entry to the FormLetter.
     - Note that various text methods, such as trimming and substrings will be needed to make this go smoothly.
   - Load a `Properties` with the contents of the associated file name.
   - Invoke the `doFormLetter` method on the FormLetter.
4. Run the `FormLetterFile` `main` method and capture the result for your submission.
5. Create your own `FormLetter` template file and associated `Properties` file, and test your program by running `FormLetterFile` with them.

Notes

- The client method is expected to *either* call the one-parameter constructor or call the no-parameter constructor followed by calling `setFile`. Calling `readLine` without opening the file first should throw an appropriate exception `getTokens` should simply

return a zero-length array of Strings if called with an empty String
- The `setFile` method needs to arrange for the file to be read one line at a time. This will make it convenient for the input to be setup as a `BufferedReader` object. A BufferedReader requires a `FileReader` to construct it. A `FileReader` is constructed from a File Also, note that it is possible the file does not exist, or perhaps cannot be read. Thus, your `setFile` may want to call a `setInput` function like so:

```
private void setInput(String filename) throws FileNotFoundException {
    try {
        FileReader f = new FileReader(filename);
        input = new BufferedReader(f); // Assumes input is the field name for the BufferedReade
    } catch (FileNotFoundException fnfe) {
        System.err.println("File "+file+" not found");
        throw fnfe; // rethrow the exception
    }
}
```

- The `readLine` method can use delegation return `input.readLine()`, just like any other `BufferedReader`. However, you may also need to catch a possible `IOException` it may raise.
- The String split method provides an efficient way to parse input strings.
    - The class `StringTokenizer` provides another very flexible way to parse input strings.
        - Note that, each time you find a "{" token, you next need look for a "}" token, to find the end of the `DataItemEntry` name.
        - The basic StringTokenizer methods are `hasMoreTokens()`, which returns true when there is another token to read, and `nextToken(delimiter)`, which returns the next String bounded by that delimiter.
        - When constructing a `StringTokenizer`, you may provide the default token delimiter, and a boolean indicating whether you'd like to get the delimiters themselves back as tokens.
    - Also, you can just find the tokens using the `String` `indexOf` and `substring` methods.
    - It is also possible to use the `Scanner` class
- When accumulating your array of results, you may find it useful to temporarily store them in a `List<String>`, since it is easy to add Strings to it. A `LinkedList` of `String` is a good implementation class. To turn a `List` into an array, remember to use the `toArray` method of the collection object, and pass a `new String[0]` as a parameter to coerce the return type.
- `Properties` can be loaded directly, given a file name.
- While you will create your own form letter, a sample form letter file might have content like so (or even be a web page):

```
{date}
Dear {name},

BREAKING: {newsHeadline}

This is an ALL HANDS ON DECK SITUATION:
If we don't fight back, the {otherParty} will get their way.

Donate to {thisParty} TODAY so we can finally put an end to the {otherParty} shenanigans!

Give {amount}£ now

Or, donate another amount

Paid for by the {thisParty} PAC, not authorized by any candidate or candidate's committee.
```

- While you will create your own properties file, a sample file, useful for the letter above, might have content like so:

```
name=Loyal Party Supporter
newsHeadline=Rt. Hon. Lord North calls for OUTRAGEOUS INCREASE IN TAXES on TEA!
thisParty=Sons Of Liberty
otherParty=British
date=May 8, 1773
amount=1
```

Evaluation

Criteria	Weight
FormLetterFileReader, test, and output	35%
FormLetterFile program and test output	35%
Your own FormLetter template, and its output run	30%