CSC 115: Fundamentals of Programming II
Assignment #1: Introduction to Interfaces

Due date

Thursday, May 28th, 2015 at 9:00 pm via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #1 link on the CSC 115 conneX site. Please make sure you follow all the required steps for submission – *including confirming your submission!*

Learning outcomes

When you have completed this assignment, you should have become re-acquainted with:

- How to create a *one-dimensional (i.e., 1D) array*.
- How to *read from and write to array elements*, using both *explicit and computed index values*.
- How to understand coding involving *simple file input and output* using *streams*.
- Your code indented and appropriately document via comments. (Please follow the guidelines in coding_conventions.pdf available in the "Lectures & stuff" folder on conneX.)

A (partial) sequence alignment program

Problem description

An important task in bioinformatics is the alignment of DNA and RNA sequences. For our purposes a *nucleic acid sequence* (or *sequence* for short) is a string of letters: *A* for *adenine*, *C* for *cytosine*, *G* for *guanine*, and *T* for *thymine*. Here is one possible string:

```
TATGCCTCCGGTACATCAAC
```

In a *pairwise sequence alignment*, two such strings are compared in order to determine how best one aligns with the other such that the nucleic acid sequences match each other. Real sequence data as used by biochemists and bioinformatics research consist of very long strings of A, C, G and T. Determining alignment for the general case can require the use of complex algorithms.

However, for this assignment we will do something much simpler. We will attempt to align two strings where the second string is shorter than the first. Here is an example of a second string:

```
TCAA
```

In aligning the second string to the first string provided above, we try to find the character position in the first string (or *offset*) at which the characters of the second string also appear. In our example, that offset would be 15 (i.e., an offset of 0 would mean the beginning of the first string is identical to the second string). We could visualize the alignment of the two strings as follows (where periods are used as filler characters):

```
TATGCCTCCGGTACATCAAC
...............TCAA.
```

Unfortunately we cannot always expect the second string to appear exactly in the first string. Sequence alignment is made more difficult as the alignment must take into account differences between the sequences. For example, consider the following two strings where the second is to be aligned to the first.

```
ACGGTACGGAGTCTTGAAGT

GTTA
```

The second string GTTA does not appear anywhere in the first string. However, we still wish to attempt the best possible alignment. As a thought experiment, imagine we try to align the two sequences by using an offset of 0. This would give us:

```
ACGGTACGGAGTCTTGAAGT
~~GTTA~~..............
```

Unfortunately this is a poor alignment. No characters match between the sequences., and I have indicated this with the use of ~~strikethroughs~~. Using an offset of one (1) is no better:

```
ACGGTACGGAGTCTTGAAGT
.~~GTTA~~.............
```

However, if we use an offset of two (2) then situation begins to improve:

```
ACGGTACGGAGTCTTGAAGT
..G~~T~~TA............
```

In the first two attempted alignments we have four *errors* (i.e., mismatched characters between the strings). In the third attempted alignment we have one *error*. We slightly change our visualization by using an "x" in the second string for a mismatched character:

```
ACGGTACGGAGTCTTGAAGT
```

```
..GxTA..............
```

Therefore we can restate our goal for alignment: We find the (rightmost) offset where the number of *errors* is minimized. In our running example (which also corresponds to test case in02.txt provided with this assignment) the offset happens to be 2.

Shown above is a snapshot of an execution of a finished version of a program named *SeqAlignment.java*.

```
$ java SeqAlignment --file tests/in02.txt
13
0
```

The first number printed is an offset, and the second is the number of errors in the match. Here is another run of the program but this time using the --verbose flag:

```
$ java SeqAlignment --file tests/in02.txt --verbose
AGAAACAGTCATAGCTTGGT
............GCTT...
13
0
```

The structure of input files is simple. As an example, here are the contents of tests/in02.txt is:

```
2
AGAAACAGTCATAGCTTGGT
GCTT
```

The first line denotes the number of strings in the rest of the file. Strings then follow, one per line.

You are given the code for *SeqAlignment.java*, and this Java class is written assuming the existence of an implementation of the *Aligner.java* interface. Given an implementation of the *Aligner* interface, such as *MyAligner.java*, the following four methods would be found in *MyAligner.java*:

- *int getOffset(int index):* Returns the number of characters by which the string indicated by index must be shifted in order to align with the other strings making up the alignment. The first string is always index 0, the second string is always index 1. For this assignment, the first string will always have an offset of 0 (i.e., it is the second string that is shifted to perform the alignment).
- *String getSequence(int index):* Returns the string corresponding to the sequence stored at index. If index is 0, then the first string is returned; if index is 1, then the second string is returned.
- *int getNumErrors():* Return the number of mismatched characters present in the most recent alignment.

- *void performAlignment():* Execute the actual algorithm for aligning the strings.

(Note: The code provided for you in SeqAlignment.java performs all of the file input and the output of results. That is, you are not expected to write this functionality yourselves.)

What you are to write

You are to:

- Write a Java class named *BasicAligner.java* which will be an implementation of the *Aligner* interface.
- Simultaneously write a Java class named *TestBasicAligner.java* which will indicate to you whether or not your implementation effort is succeeding.
- Modify a single line in the provide *SeqAlignment.java* – line 50 – so that instead of using the provided *NullAligner.java* the program uses your *BasicAligner.java*. (The line to be modified is also indicated by a comment in the file.) ***You are otherwise forbidden to modify any other part of SeqAlignment.java***

Here are steps to follow:

1. Write a bare-bones class named *BasicAligner.java* (you may base it on *NullAligner.java*) that implements the Aligner interface. Ensure the file compiles correctly. Note that the constructor's signature is passed in an array of strings and therefore more than two strings could be aligned. However, for this assignment you need only store two strings as instance variables.

2. Write a very simple program (having a "main()") named *TestBasicAligner.java* which will instantiate a *BasicAligner* class and be able to send messages to the object. (Use this for initial testing rather than the testing the *SeqAlignment* program directly.)

3. In the constructor of *BasicAligner* convert the strings into character arrays. (You'll want to keep the strings and the character arrays as instance variables.) These arrays can then be used to implement this method:

   ```
   private int matchAt(int firstI, int secondI)
   ```

   which returns the number of mismatched characters when comparing the first string at character-array index *firstI* with the second string at character-array index *secondI*. For example, if the first string is "AAAGGGC", and the second string is "GGC", then *match(3, 0)* would return 1, but *match (4, 0)* would return 0. In order to test this method, add a main() method to BasicAligner as this class and place your testing code here (i.e., this main

method will have access to the private method that is otherwise inaccessible from TestBasicAligner).

4. Write an implementation of:

   ```
   public void performAlignment()
   ```

   which makes use of *matchAt().* The algorithm can be one that simply loops through (nearly) every character in the first string, and in each loop iteration attempts to match the second string starting at the first's string character indicated by that iteration. The algorithm needs to keep track of the first-string character index that yields a match (actually, the right-most match) having the lowest error count (i.e., lowest value returned by *matchAt*). Once alignment is completed the results are to be stored in some instance variable for later retrieval.

5. Write implementations of *getOffset*, *getSequence* and *getNumErrors*.

6. While completing steps 4 and 5, write tests in *TestBasicAligner.java* that will exercise code in your implementation. The tests should be simple and straightforward (i.e., do not replicate in your test class any of the ten provided test cases).

7. Write any other methods needed to support your implementation of *BasicAligner.java*.

8. Modify *SeqAlignment.java* in order to use your new *BasicAligner* class. Test the result with the ten tests provided with this assignment. (The tests are in the tests/ directory provided with the assignment. Each of the ten tests consists of an input file and a file containing the output expected -- but note that SeqAlignment.java outputs to the console and not to a file.)

Files to submit (four in total):

a. *BasicAligner.java*
b. *TestBasicAligner.java*
c. *Your modified SeqAlignment.java (note: only one line is to be changed)*
d. *A file named test_output.txt (which also includes a description of what each test is checking followed by the test input and output)*

Grading scheme

- "A" grade: An exceptional submission demonstrating creativity and initiative going above and beyond the assignment requirements. The program runs

without any problems and uses arrays, streams, exceptions and the specified methods. Methods are tested with output showing tests and their results. Any extra work appears in class files having the name *<file>Extra.java*, and identified within the file (i.e., Class comment) are details of your extension to the assignment demonstrating creativity and initiative.

- "B" grade: A submission completing the requirements of the assignment. The program runs without any problems and uses arrays, interfaces and the specified methods. Methods are tested with output showing tests and their results.

- "C" grade: A submission completing most of the requirements of the assignment. The program runs with some problems but uses arrays, interfaces and the specified methods, and yet might not have the expected output. Methods are tested with output showing tests and their results.

- "D" grade: A serious attempt at completing requirements for the assignment. The program runs with major problems, or does not use arrays, interfaces, or the specified methods. Methods are tested with output showing tests and their results.

- "F" grade: Either no submission given, the submission does not compile, or submission represents very little work.