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

Due date

Monday, January 29, 2018 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*.
- Identify coding involving *simple file input and output* using *streams* in Java.
- Identify the use of *exceptions* in Java.

Encryption and decryption of text

Problem description

Our modern world's combination of massive amounts of digital data (both private and public) and ubiquitous computing devices has resulted in *cryptography* becoming an important application for computers. Cryptography helps protect our privacy and secure our financial transactions from prying (e.g., phishing) eyes. In this assignment you will implement *encryption* and *decryption* algorithms for a simplified *polyalphabetic substitution cipher* (also known as the *Vigenère cipher*).

An *encryption algorithm* accepts a string—also called the *plaintext*—plus another string called a *key,* and converts the plaintext into a string called the *ciphertext*. A *decryption algorithm* performs the reverse: it accepts a ciphertext along with the key used to generate the ciphertext, and produces the original plaintext.

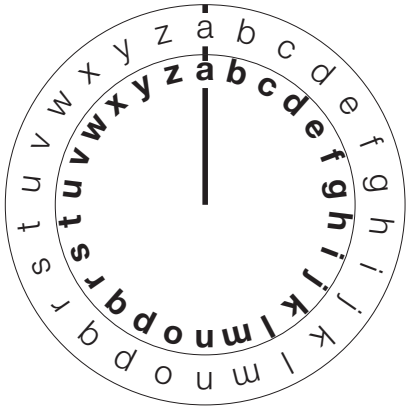
As an example consider the following short plaintext:

`themessage`

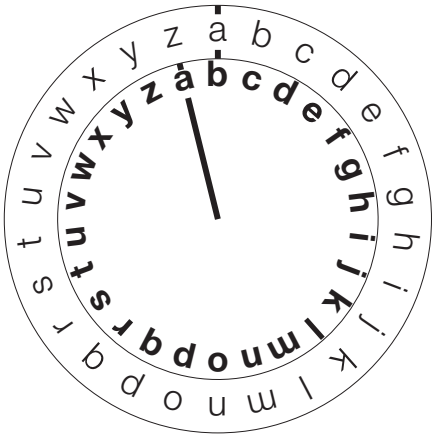
and this three-character key:

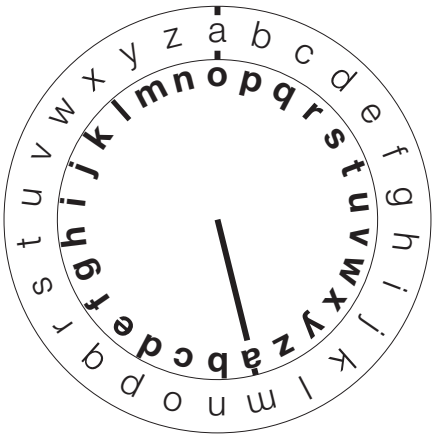
bob

To encrypt the message we will use the special *code*disc shown below. The outermost ring of letters is used for plaintext characters, and the innermost ring is used for ciphertext characters.



The inner ring of the codedisc may be rotated. Below are two snapshots of the state of the codedisc where the inner ring is rotated. The state on the left is set to encrypt text based on the key "b"; the right is set to encrypt text based on the key "o". These states are set by aligning the key character (on the inner ring) to the 'a' of the outer ring. The left state will encrypt an 'a' into 'b', an 'f" into "g", a 'y' into 'z', etc. The right will encrypt an 'a' into 'o', an 'f" into 't', a 'y' into 'm', etc.





To encrypt our plaintext above ("`themessage`") with the key ("*bob*") we use the two example states of the codedisc above. First we write the key above the plaintext repeating it as many times as is needed to match the length of the plaintext:

bobbobbobb
`themessage`

Then for each plaintext letter we must use the appropriate state to encrypt the letter. The first plaintext letter 't' is under the letter 'b' in the repeated-key row. That key-letter 'b' indicates that a plaintext 'a' will become a ciphertext '**b**'. Therefore to encrypt our first plaintext letter we examine the left state outer ring, and find that the ciphertext letter on the inner ring is to be '**u**'.

A similar procedure follows for our second plaintext letter 'h' which is under the key-letter 'o'. For this second letter, the codedisc on the right (set to encrypt an 'a' as an '**o**') indicates the ciphertext letter is to be '**v**'.

Finally after applying this procedure to the plaintext, the resulting ciphertext is as shown below in boldface type:

bobbobbobb
themessage
uvfnsttohf

Decryption is simply the reverse. With the key given to us for the ciphertext and the inner ring correctly rotated, we locate ciphertext letters on the inner ring, and read corresponding letters on the outer ring. In this way we obtain the original plaintext. To decrypt a ciphertext we *must have **both** the ciphertext **and** the original key* used during encryption of the corresponding plaintext.

A simple computer program to simulate the Vigenère cipher

We will not use codediscs, but we can instead use a combination of *letter-to-number mappings* and *modulo arithmetic*. (Modulo arithmetic is often explained using an analog clock, and our codediscs are a little bit like the numbers on such clocks.) For example, we can map "a" to the number 0, 'b' to 1, ..., 'z' to 25. To encrypt plaintext 't' when the corresponding key letter is 'b':

- Convert '**t**' into a number (which will be 19).
- Convert 'b' into a number (which will be 1).
- Add the two numbers modulo 26 (e.g., (19 + 1) % 26).
- Convert the result back into a letter (20, which is '**u**').

The ciphertext is therefore '**u**'. Modulo arithmetic helps us when the result of adding the two numbers is greater than 25.

If the plaintext letter is 'h' while the key-letter is 'o', the arithmetic expression will be ((7 + 14) % 26) or 21, and this yields the ciphertext letter '**v**'.

Transforming ciphertext to plaintext is similar, with the only difference being the key value is ***subtracted*** from the letter value before modulo 26 is computed. *However, Java handles modulo arithmetic with negative numbers in a way that is arithmetically valid but causes problems for us!* For example, if the ciphertext character is '**c**' and the key character is '**m**', then the following expression in Java

(2 - 12) % 26

yields the value -10. However, our codedisc configured for that key character would have produced a plaintext "q" (which is value 16). To handle this situation our modulo expression for decrypting will always need to be slightly modified as shown below to prevent applying % to a negative number:

(**26 +** 2 - 12) % 26

This will produce the desired result (i.e., the number 16).

Programming requirements

You will write the source code for the *VigenereCipher* class. This class is responsible for the actions that simulate the encryption and decryption as described above.

Implementation details for VigenereCipher: The details for the source code are specified in a specification document (available at `http://bit.ly/2CUw6p3`). **The preceding direct link requires you to be logged onto conneX.** Note that this document only describes the class declaration and the names of the methods. It is up to you to determine if data fields are necessary; however, the methods must be exactly as specified in the *Method Detail* section of the specification document. We recommend that you copy and paste the class and method headers into your source code.

Normally, the specification document does not include private helper methods. These methods are usually created to help break down tasks into smaller subtasks and are not accessible to any external programs. You may add any number of extra private methods if you wish, but you must include all of the methods that are detailed for this assignment.

Getting started: We provide the following basic steps, and recommended order, to get you started:

1. Download the files associated with this assignment into a fresh *working* directory (*csc115/assn1* for example).
2. Copy and paste the class description and each of the public and private method headers into a new file called *VigenereCipher.java* (in the working directory).
3. Create minimal method bodies by placing the curly braces in the correct places. If a method returns an array or string, add the following *temporary* line inside the curly braces: `return null;`
4. Compile the file until there are no errors.
5. Start filling in the source code for the simplest method. We recommend dumpArray to start. Make sure that the method works correctly (see details in the next section). Once a method works, then continue with the next simple method. We recommend you implement the private methods first. When

they are subsequently *called* in later methods, we can be confident that a call to a previously tested method will behave as it should.

6. Repeat step 5 with the remaining methods, until every method is complete. Verify each method's correctness by testing it. We describe how to do that in the next section.

Testing VigenereCipher:

If all the parts (methods) of *VigenereCipher* behave as they are supposed to, then very little needs to be done in putting together the final product. To inspire you to practice good programming techniques, you are required to test each of the methods that you write.

You do this in the main method of *VigenereCipher*. In the software-development industry, before submitting the final product, a programmer may elect to remove the main method in a class. In this assignment, you must leave it in the file, as a demonstration that you have individually tested each method. You may comment out lines as you go, but we recommend not deleting them.

The following is an example of some statements (inside the main method of *VigenereCipher*) that will produce output for the programmer to verify whether *dumpArray* and *stringToIntArray* are working:

```
// the following statement creates an object
// this object provides the access to all the methods
// (even the private ones)
VigenereCipher vc = new VigenereCipher("dd");
System.out.println("converting 'blog' to an array of ints");
int[] toNums = vc.stringToIntArray("blog");
vc.dumpArray(toNums, "result:");
```

Once we run *VigenereCipher*, if the output does not look like:

```
converting 'blog' to an array of ints
result:1,11,14,6
```

then we need to debug the code in the targeted method (either *dumpArray* or *stringToIntArray* in this case). If the output looks fine, then we can move on, confident that when we call either of these methods from any other method, it will work correctly.

Putting it all together (the finished product)

The completed *EncryptDecryptText* class manages the flow of input and output by accessing the information in a command-line run of the program. You are not required to understand all of the completed code inside this file, but you may find it a helpful reference for file input and output, and for Java's exception-handling. Note that this class also contains a main method, and makes reference to both the *Cipher* and *NullCipher* classes in line number 58.

Without making any changes to the file, you can compile *EncryptDecryptText.java* and then type the following line on the console, inside your working directory:

```
$ java EncryptDecryptText -k bob -e in01.txt -o output.txt
```

This accesses the plain text inside the file "in01.txt" (themessage), encrypts it against the key "bob", and places the resulting encrypted string into the (newly created) file called "output.txt".

However, the resulting encrypted message when using the unchanged Java file is not very cryptic. It is the same string! Your final programming task is to find the single line that needs to be modified in *EncryptDecryptText*, so that the actual Cipher that does the work is your newly created *VigenereCipher*, instead of the not-very-cryptic NullCipher.

Compile the updated *EncryptDecryptText* and re-run the above command on the console. Note the result. If it matches the string in the file called "out01.txt", then it worked!

The command for decryption is similar. Type the following line on the console:

```
$ java EncryptDecryptText -k bob -d output.txt -o somefile.txt
```

After this command, the newly created "somefile.txt" is expected to contain the same string as the original plaintext in "in01.txt".

There are five pairs of "txt" files that are provided to test the correctness of the whole process. These files are paired (plaintext, ciphertext) by their matching numbers. The corresponding key is also referenced by this number in the README.txt file. For instance, if I want to check the encryption of the contents of "in05.txt", I find the appropriate key in README.txt that corresponds to Test 05, which is "shakespeare", I type :

```
java EncryptDecryptText -k shakespeare -e in05.txt -o somefile.txt
```

on the console, then look at the contents of "somefile.txt", which I expect to be exactly the same as the provided file named "out05.txt".

Files to submit:

a. *VigenereCipher.java*
b. *Your modified EncryptDecryptText.java (note: only one line is to be changed)*

Grading

- Submissions are expected to follow all of the requirements detailed in the specification (link is `http://bit.ly/2CUw6p3`).
- The coding conventions, specified in the "Coding Conventions for CSC 115" document on conneX (link is `http://bit.ly/2D4f8oh`) must be followed.