Computer Laboratory 8

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday November 8st at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs have explicit instructions about useful topics to review/overview which will help you excel in this lab.
- For more rules see the lab rules document on canvas

2 Learning Objectives

This lab is designed with a few learning objectives in mind. By doing this lab you will:

- Practice manipulating the primitive char data type
- Practice reading about useful functions in official java documentation
- Build your first java classes.
  - This includes practicing documentation a **CLASS**
  - As well as thinking carefully about your choice of visibility modifiers!
- Have an opportunity to write code of non-trivial algorithmic complexity inside Java
- Get further practice using java classes – this time with a class you've created.

3 Introduction

In this lab we will be working for a secret client. Our client works for an agency in charge of detecting a secret spy ring sending coded messages in the U of M's computer science department. They've asked us to implement a few things:

First, our client's agency has determined that the spies are using a Caesar cipher, and have asked us to implement code for encrypting and decrypting text based on a Caesar cipher. Then our client would like us to build some core parts of their message tracking and spy tracking operations.

The Caesar cipher is a classic piece of cryptography history. It's classified as a "rotation" cipher. This means that the algorithm works by replacing each letter with a different letter by "rotating" it around the alphabet. For example the string "abe" rotated by 3 would be "deh" ('a' +3 = 'd', 'b' +3 = 'e', and 'e' +3 = 'h'). This process works for letters later in the alphabet by "rotating" around the alphabet: ('z' +1 = 'a', 'x'+5 = 'c', etc.) While relatively simple, this cipher can be difficult to decode without knowing exactly how far the letters were rotated.

A few further notes: First, decoding this cipher is likewise simple to do if you know the rotation amount, simply rotate backwards, so ('d' -3 = 'a', 'c' -5 = 'x', etc.) Secondly, when encoding or decoding you do have to be careful: it's possible for a letter to be "rotated" from the end of the alphabet back to the beginning. ('x'+5 = 'c' as above.) For a human this "wrap around" rotation is easy – but it's common to mess up this detail when working in a computer.

Applied in modern computing, this cipher is relatively easy to break (English letters are not used at the same rate, in long text, it's a safe bet that whatever letter shows up the most is 'e', and you can work backwards from there). Despite this limit, it seems our client's agency is tracking spies who still think this is a useful cipher. For computer text we will want this algorithm to only deal with lowercase letters. Any uppercase letter should be replaced with an equivalent lowercase letter before encryption or decryption. Non-letters (numbers, spaces, etc) should be ignored.

For example:

`"Thursday's operation is prepared. Pick up your parcel before departing."`

rotated +20 is

`"nbolmxus'm ijyluncih cm jlyjulyx. jcwe oj siol julwyf vyzily xyjulncha."`

(note the capitol T goes is turned into a lowercase t and then rotated to n. Likewise the capitol P is made lowercase and punctuation and spaces are left unmodified.

4 A warning about Academic Integrity

While it is quite reasonable to want to google for more information about the Caesar cipher, and possibly find examples or an online encoding tool, you should be cautious. This is a classic encryption algorithm and is used as a code example on various educational websites. Be careful which websites you visit, and what you google to ensure you do not see any

programmed solutions to this problem, as that would be a breach of our academic integrity policy.

This is your reminder. Googling "Java Caesar cipher" is cheating even if you don't find anything useful. Looking at reference solutions for this algorithm is cheating even if you don't copy code. And of course, copying code is cheating). If you feel like you need this level of help designing your solution I ask that you contact course staff and set up a one-on-one meeting so we can help you build your skills to meet this challenge. At the end of the day, cheating will hurt you the most – as you will be less prepared for future labs.

That said, I **strongly encourage** you to read more about Caesar ciphers themselves. Here is a list of further readings I've reviewed and can certify as "safe" to reference:

- `https://en.wikipedia.org/wiki/Caesar_cipher` (wiki article)
- `https://simple.wikipedia.org/wiki/Caesar_cipher` (simple-English wiki article, a bit less academic and more to the point)
- `https://cryptii.com/pipes/caesar-cipher` (an online tool for encrypting and decrypting)
- `http://practicalcryptography.com/ciphers/caesar-cipher/` (a longer form article about the cipher)
- `https://www.xarg.org/tools/caesar-cipher/` (an online tool for encrypting and decrypting)

5 Useful Ideas and Concepts

The Caesar Cipher algorithm involves a lot of String and Character manipulations. This section contains specific recommendations, hints, and help.

5.1 Functions worth researching

This subsection lists specific functions and methods on the String and Character class that might be helpful to you.

You are expected to: look into the following functions online to understand what they are, and how to use them.

When reading the official Java documentation for these functions, pay special attention to if the functions are static or not, and if the functions return a value. This effects how you should call these functions. For example, String function `toLowerCase` is not static and returns a String so you would probably need to call it:

```
text = text.toLowerCase();
```

While the Character method isAlphabetic is static and returns a boolean so you might use it:

```
if (Character.isAlphabetic(c)) {...}
```

- String class: `https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`
  - `toLowerCase`
  - `charAt`
  - `length`
  - `split` (this one can be a bit tricky to read about due to the advanced idea of "Regular Expressions" – you may want to "test" this function out on your own before building it into a bigger program)
  - `contains` (This one can be a bit tricky to read because it claims it's parameter is "CharSequence" – you can call this function with a string.)
- Character class: `https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html`
  - `isAlphabetic`
  - `isLowerCase`
  - `isUpperCase`
  - `toLowerCase`
  - `toUpperCase`

While this will reduce the amount of "char math" you need to use, it will not get rid of it.

5.2 Char math

Every programming language has a way to perform "char math" this is to say, mathematical style manipulation of char values. The "heart" of this is having a way to convert from char values to int values and back again.

Given a char `c`, you can use the following code to get an integer equivalent: `int i = (int) c;` Likewise given an integer `i` you can use the following code to get a char equivalent: `char c = (char) i;` The `(int)` and `(char)` syntax is called a type conversion – it requests the computer perform a possibly unsafe conversion between types. In this case it will convert a char to it's raw Unicode value.

Chars also support addition and subtraction so given a lowercase English letter stored in variable `c` then the code: `int i = (int)(c - 'a')` will set i to be a number indicating the position of `c` in the English alphabet (a = 0, b=1, c=2, etc.) This can be converted back like so: `char c = (char)(i + 'a')`.

This will be specifically useful for implementing the Caesar cipher. Caesar ciphers are based on a "mathematic" vision of the alphabet where each letter represents a number – so

you will find it useful to first transform a letter to a number, and then perform much of your "rotation" manipulations on those numbers.

A common issue in implementing rotation ciphers like Caesar cipher is rotating letters like z into non-letter values instead of into early letters in the alphabet. I.E. you need to arrange for 'z' + 2 to go to 'b'. (or in numbers, you need to take 25 +2 and get 1). This operation can be done with careful use of if statement, however a more elegant solution can be found if you make careful use of the modulo operator.

5.3 building strings in loops

A pattern that might be useful for building strings in loops is as follows:

```
String output = "";
<some loop>{
  compute next letter of output string
  output += next letter
}
return output.
```

This basic code outline is well suited to this problem. Simply loop over each letter in the input strings and process them to the correct output string.

6 Files

This lab will involve the following files

- `CaesarCipher.java` – Code that implements the Caesar cipher.
- `CaesarCipherTest.java` – code that tests the Caesar cipher
- `KnownSpy.java` – code that tracks information about known spies, as well as code for managing our database of known spies
- `KnownSpyTest.java` – code that tests the Known Spy test
- `Message.java` – simple java object for tracking messaging information
- `MessageTest.java` – tests for the message class.

You will build Message, CaesarCipher, and Message classes and will need to submit all three for grading.

7 Requirements and Software Design

This lab involves the creation of three Java classes. While you can theoretically do these in any order, I recommend the following order:

- Message - this class is the simplest, making it a good choice to program first.
- CaesarCipher - This class has the most complicated code.
- KnownSpy - this class uses the other two classes, so it really should be last.

These three classes, and their formal requirements are listed below:

7.1 Message

A message object represents one of the messages that our client's agency has intercepted. Many of these messages are innocent, but some are sent by spies and are encrypted with a Caesar cipher.

A complete message must implement the following – all should be public and non-static.

- a three argument constructor, the first argument is the name of the person who sent the message, the second argument is the name of the person who received the message, and the third argument is the message itself.
- a method `getFrom()` which returns a String (the person who sent the message)
- a method `getTo()` which returns a String (the person who revived the message)
- a method `setTo(String)` which returns nothing, and changes the record of who the message was sent to.
- a method `getMessage()` which returns a String (the test of the message itself)
- a method `setMessage(String)` which returns nothing, and changes the record of what text was sent.
- a `toString()` method which returns a String. The format for this should exactly match the examples in the test file: `Message from: <from> to: <to> Message "<message>"`.

You will be in charge of deciding which private attributes are needed to make this possible.

7.2 CaesarCipher

A `CaesarCipher` object represents one version of the Caesar cipher described earlier in this document. Each instance of the cipher is defined by the *key* value – how far it "rotates" around the alphabet. This key value should be a private attribute of the CaesarCipher object, and should be an integer.

A complete CaesarCipher must implement the following methods – all should be public and non-static.

- a constructor that takes one parameter – the key value (as an int) that this object will use.
- a method `isValid()` that returns a boolean indicating if the key is valid (between 1 and 25, a rotation of 0 or 26 does nothing and is invalid, rotations above (or below) those limits do the same thing as one of the rotations within those limits)
- a method `encrypt(String)` that returns a String, the result of encrypting the input string with the Caesar cipher algorithm outlined above, rotating by the key amount provided when the object was created.
- a method `decrypt(String)` that returns a String, the result of decrypting the string following the Caesar cipher algorithm. Note, decrypting is simply the opposite of encrypting. Simply rotate by -key instead of key.
- a method `toString()` that returns a String. The format for this should exactly match the examples in the test file: `Caesar(<key>)`.

The encrypt and decrypt methods cannot assume that their inputs will only be lowercase letters. Your functions should deal with:

- lowercase letters
- uppercase letters
- non-alphabetical letters (spaces, symbols, punctuation, numbers, etc.)

Non-letter characters should be left alone and not "rotated" or removed, letters can be returned only in lowercase – you do not need to preserve the case, so the input "ABC" rotated by 1 would be "bcd".

7.3 STRONG RECOMMENDATION

This is recommended, not required, but we know from experience that many students will struggle if they do not do this. As a recommendation – implement a private helper method that takes a char, and an int (the key) and returns that char rotated by the correct amount. This "char math" is quite difficult for many students, and doing this as it's own simple-and-targeted function will make testing and designing this much easier. When you have a

function on it's own like this you can easily test it in isolation for specific letters – then when it works, use it in loops to do full encrypting and decrypting.

So if you name your function "rotate" then:

- `rotate('a', 3)` would be 'd'
- `rotate('a', 10)` would be 'k'
- `rotate('a', -3)` would be 'x'
- `rotate('v', 16)` would be 'f'

You are free to not do this, but if you come to us for help making your char math work and have not structured your code in this way, we may force you to do this before anything else. Good helper functions like this are an essential tool for organization and incremental development, and skipping it can easily make an otherwise possible task insurmountable.

7.4 KnownSpy

A `KnownSpy` object represents a Spy, using a Caesar cipher, which we are aware of. A KnownSpy object is composed of the name of the spy, and a CaesarCipher object that can decode messages from this spy. (Both of which should be private instance attributes.) It should be noted, that this class also has a static method for handling databases of spies.

A complete KnownSpy class must implement the following:

- A public constructor that takes two parameters, the name of the spy and a CaesarCipher object that can decode the Spy's messages.
- A public non-static method `getName()` which returns a String, the name of the spy.
- A public non-static method `decrypt(Message)` which returns nothing. This method takes a message and, if the message was sent by the spy, decrypts that message (changing it's message text string). If the message is from the spy the input message object itself should be modified, otherwise no change should be made.
- A static function `isFromSpy(KnownSpy[], Message)` That returns a boolean. The function takes an array of known spies and a message object. If the sender of the message is one of the known spies the function should return true, otherwise it should return false.
- A public non-static function `tryDecrypt(String encryptedCommonWord, String commonWords)`. This one has specific pseudocode:
  - The goal of this function is to discover the encryption key for a known spy whose cipher key is not known (or appears to have changed) by brute-force decrypting a single common word.

- The first parameter is a single word that is encrypted but appeared frequently in messages from this spy.
- The second parameter might be a string like "cat dog programming algorithm and the above below or" etc. – basically a list of common words that will show up frequently in common English text separated by spaces and might be the encrypted word's decrypted form.
- The function should then attempt to decrypt the encrypted word using every legal key 1-25. If any of the keys lead to a decrypted word that is in the commonWord string, return this word as the "correct decryption". This will be used outside our program to further decrypt longer samples.
- If no key leads to a commonWord when decrypting the encrypted common word, return null.

8 Submission

For this lab you need to turn in:

- `Message.java`
- `CaesarCipher.java`
- `KnownSpy.java`

Grading on this assignment will follow the given general rubric:

- 50 points autograder – This is split somewhat chaotically across the methods.
- 10 points code style:
  - Your name (and the name of any collaborators) should be in a comment at the top of the file
  - Correctly formatted JavaDoc (this is the `/** ... */` comment blocks placed before each function that describes the function
  - You should also have a JavaDoc before each class that describes the class as a whole. You can look at the provided code from lab06 to see examples of this
  - Good variable names
  - Consistent tabbing (while java does not require this the same way python did – we **do** require it. Reading and grading poorly tabbed code is a painful experience)
  - Outside of these explicit requirements, as normal, we will generally be looking for (and giving feedback on) anything that makes the code harder/more painful to read, and will take away points if it's particularly bad.

- 10 points for Message class – note we will be looking for more than direct function requirements, good use of modifiers like public/private will also be checked here.
- 15 Caesar Cipher Class – note we will be looking for more than direct function requirements, good use of modifiers like public/private will also be checked here.
- 15 Known Spy Class – note we will be looking for more than direct function requirements, good use of modifiers like public/private will also be checked here.