MV Cipher

Objective: To use various data types and arithmetic to create a MV Cipher.

Background:
Ciphers have been used for thousands of years to disguise messages from prying eyes. Modern encryption has its roots in cipher technology. The Caesar Cipher is one of the oldest and simplest of ciphers. It was called a "shift" cipher because letters of the original message are replaced by letters a certain number of letters up or down the alphabet.

The MV Cipher is a variation on the Caesar cipher and a much stronger encryption. The stronger the encryption, the more difficult it is to "break" the code. The MV Cipher uses a series of Caesar ciphers based on the letters of a keyword.

Suppose the plaintext message is "Attack at 04:00 hours". You select a keyword and repeat it until it matches the alphabet characters of the plaintext message. For example, using the keyword "FRUIT" it would be:

```
A t t a c k <sp> a t <sp> 0 4 : 0 0 <sp> h o u r s
F R U I T F      R U                     I T F R U
```

The first letter "A" would use the key character "F" for encryption. Since "F" is the 6th character in the alphabet, then you shift "A" six characters up the alphabet to "G" (see encrypted version below). The second letter "t" of the original message would use the key character "R". Since "R" is the 18th character in the alphabet, then you shift "t" eighteen characters up the alphabet with wrap-around which eventually lands on "l", and this becomes the second cncrypted letter. You continue this pattern until you get the encrypted message:

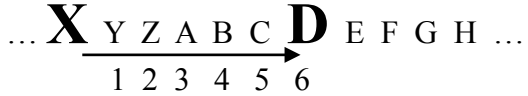
```
G l o j w q <sp> s o <sp> 0 4 : 0 0 <sp> q i a j n
```

Notice that uppercase and lowercase is preserved, and non-alpha characters are preserved which includes spaces (<sp>). To decrypt the message, you use the same keyword but shift "down" the alphabet with wrap-around. Your assignment will be to implement a MV Cipher encryption/decryption program.

Discussion:
The MV Cipher program can take advantage of Java's ability to use characters as arithmetic operands. To shift the letter 'A' up six characters, we could write:

```
char oldChar = 'A';
char newChar = (char)(oldChar + 6);
```

Be sure to cast your values properly when you do assignments so you do not get "Possible loss of precision" compiler errors. When you compute the new character, you must mathematically wrap-around the ASCII alphabet. For example, to shift 'X' up six characters, count two letters up to 'Z' then count four letters from 'A' to the encrypted character 'D'.



Be careful! The Java modulus operator (%) does not work with ASCII character values because the alphabet goes from 65 to 90 (uppercase) and 97 to 122 (lowercase). Check the ASCII chart on the wall. Using the modulus operator could potentially wrap-around to ASCII **0** which is an unprintable character (yikes!). You need to come up with a solid algorithm to avoid this potential problem. (Welcome to APCS A!)

To avoid keyword confusion, all keywords will be alphabetic characters only. This means your program must reject keywords the user enters with non-alpha characters. In addition, your program should store the keyword as uppercase alphabetic characters only. This will help calculate the cipher shift. Finally, the keyword must contain at least 3 characters.

You will need to create one algorithm that performs encryption and another algorithm for decryption. There are many clever ways to do it, and you should try out your ideas.

In architecting your program, use this decomposition strategy.
Create:

(1) a method that reads each line of an input file into a string
(2) a method that encrypts or decrypts a string one character at a time and returns the converted string
(3) a method that encrypts or decrypts a single lowercase alpha character and returns that character
(4) a method like (3) that handles uppercase alpha characters

Testing:
A good first test of your encryption algorithm is to use the key "**ZZZ**". Since **Z** is the 26th letter of the alphabet, your algorithm will map the letter A to A, B to B, C to C, etc making the output look exactly like the input.

A good second test is to use the key "**AAA**". Since **A** is the 1st letter of the alphabet, your algorithm will map letter A to B, B to C, C to D, etc in a very easy to recognize pattern. Here is a sample run ("%" is the Linux prompt and **bold** is user input).

```
% cat testInput.txt
Yes this young man is a xerox problem.
This is just a 543, to one file.
% java MVCipher

 Welcome to the MV Cipher machine!

Please input a word to use as key (letters only) -> ap
ERROR: Key must be all letters and at least 3 characters long
Please input a word to use as key (letters only) -> apcs

Encrypt or decrypt? (1, 2) -> 1

Name of file to encrypt -> testInput.txt
Name of output file -> encrypt.txt

The encrypted file encrypt.txt has been created using the keyword -> APCS

% cat encrypt.txt
Zuv miyv rpkqz nqq bt q axsea iseeefc.
```

```
Waji ll kkvm b 543, jr hou ibmu.
```

You must also test your decryption algorithm that reverses the process and restores the original file contents. The following sample run uses as input the encrypted file created above.

```
% java MVCipher

 Welcome to the MV Cipher machine!

Please input a word to use as key (letters only) -> apcs

Encrypt or decrypt? (1, 2) -> 2

Name of file to decrypt -> encrypt.txt
Name of output file -> decrypt.txt

The decrypted file decrypt.txt has been created using the keyword -> APCS

% cat decrypt.txt
Yes this young man is a xerox problem.
This is just a 543, to one file.
% diff testInput.txt decrypt.txt
%
```

The Linux **diff** command compares character-for-character the original text file **testInput.txt** to the decrypted file **testFinal.txt**. If both files are exactly the same then the **diff** command returns nothing (as shown above). If there are differences, **diff** will print messages to your screen. Here is an example of corresponding lines not matching:

```
% diff testInput.txt decrypt.txt
2c2
< This is just a 543, to one file.
---
> Blur hm gust a 543, to pne pile.
%
```

Assignment:
Download the **MVCipher.zip** file from Mr Greenstein's web site and unzip. It will create an **MVCipher** directory in which you will do all your work.

Complete the provided **MVCipher.java** to encrypt and decrypt a plaintext file. Two test input files **cp.txt** and **Macbeth.txt** are also provided.

You must compare your encrypted results of your program to the benchmark program inside **MVCipher.jar**. The JAR file was included in Mr Greenstein's zip file. To run Mr Greenstein's cipher program, enter the command:

```
java -cp MVCipher.jar MVCipher
```

Compare your encrypted output to Mr Greenstein's MVCipher.jar encrypted output. Your program will have succeeded if the two encrypted files match using **diff**. Mr Greenstein will be performing these tests when grading.