The Demo Programs in Folder Assignment4

Strings and Ciphers

Cryptography, derived from the Greek word *κρυπτοσ* meaning *hidden,* is the science of creating and decoding secret messages whose meaning cannot be understood by others who might intercept the message. One of the most important episodes in the history of cryptography was the breaking of the German "Engima" code during World War II. The breaking of Enigma not only was essential to the Allied victory in the Battle of the Atlantic, but also was one of the first applications of digital computers. And while the Enigma code itself is beyond the scope of this course, cryptography turns out to be an ideal domain for talking about string operations.

Two of the demo programs in the folder for Assignment 4 are simple encryption and decryption programs for ciphers less sophisticated than Enigma. To read them, one need to know some terminology. In cryptography, the message you are trying to send is called the **plaintext**; the message that you actually send is called the **ciphertext**. Unless your adversaries know the secret of the encoding system, which is usually embodied in some privileged piece of information called a **key**, intercepting the ciphertext should not enable them to discover the original plaintext version of the message. On the other hand, the intended recipient, who is in possession of the key, can easily translate the ciphertext back into its plaintext counterpart.

Caesar ciphers

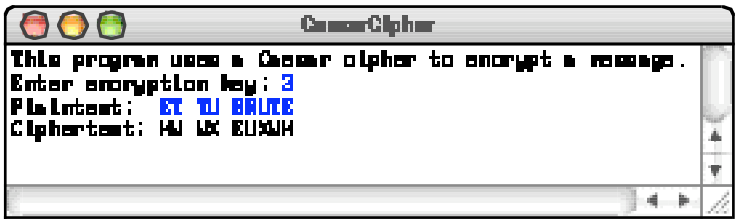
One of the earliest documented uses of ciphers is by Julius Caesar. In his *De Vita Caesarum,* the Roman historian Suetonius describes Caesar's encryption system like this:

> If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely **D**, for **A**, and so with the others.

Even today, the technique of encoding a message by shifting letters a certain distance in the alphabet is called a **Caesar cipher**. According to the passage from Suetonius, each letter is shifted three letters ahead in the alphabet. For example, if Caesar had had time to translate the final words Shakespeare gives him, **ET TU BRUTE** would have come out as **HW WX EUXWH**, because **E** gets moved three letters ahead to **H**, **T** gets moved three to **W**, and so on. Letters that get advanced past the end of the alphabet wrap around back to the beginning, so that **X** would become **A**, **Y** would become **B**, and **Z** would become **C**.

Caesar ciphers have been used in modern times as well. The "secret decoder rings" that used to be given away as premiums in cereal boxes were typically based on the Caesar cipher principle. In early electronic bulletin board systems, users often disguised the content of postings by employing a mode called **ROT13**, in which all letters were cycled forward 13 positions in the alphabet. And the fact that the name of the **HAL** computer in Arthur C. Clarke's *2001* is a one-step Caesar cipher of **IBM** has caused a certain amount of speculation over the years.

The **CaesarCipher** program encodes or decodes a message using a Caesar cipher. It reads a numeric key and a plaintext message from the user and then displays the ciphertext message that results when each of the original letters is shifted the number of letter positions given by the key. A sample run of the program looks like this:



For the Caesar cipher, decryption does not require a separate program as long as the implementation is able to accept a negative key, as follows:

Letter-substitution ciphers

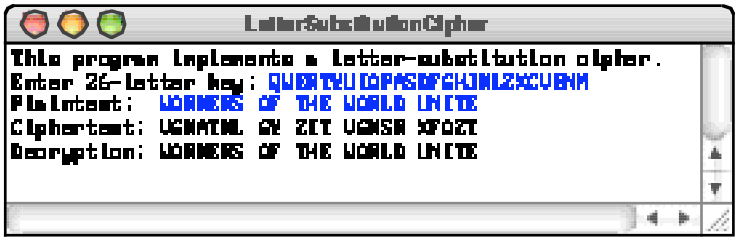
Although they are certainly simple, Caesar ciphers are also extremely easy to break. There are, after all, only 25 nontrivial Caesar ciphers for English text. If you want to break a Caesar cipher, all you have to do is try each of the 25 possibilities and see which one translates the ciphertext message into something readable. A somewhat better scheme is to allow each letter in the plaintext message to be represented by an arbitrary letter instead of one a fixed distance from the original. In this case, the key for the encoding operation is a translation table that shows what each of the 26 plaintext letters becomes in the ciphertext. Such a coding scheme is called a **letter-substitution cipher.** The key in such a cipher can be represented as a 26-character string, which shows the mapping for each character, as shown in the following example:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
Q W E R T Y U I O P A S D F G H J K L Z X C V B N M
```

Letter-substitution ciphers have been used for many, many years. In the 15th century, an Arabic encyclopedia included a section on cryptography describing various methods for creating ciphers as well as techniques for breaking them. More importantly, this same manuscript includes the first instance of a cipher in which several different coded symbols can stand for the same plaintext character. Codes in which each plaintext letter maps into a single ciphertext equivalent are called **monoalphabetic ciphers.** More complex codes—like the Enigma machine—in which the representation for a character changes over the course of the encryption process are called **polyalphabetic ciphers**.

The **LetterSubstitutuionCipher** program implements this more general letter-substitution cipher. The program asks the user to enter a 26-letter key and the plaintext

message. It then displays the ciphertext and, to ensure that the encryption is working, what you get if you decrypt the ciphertext using the same key:



Scrabble scoring

In most word games, each letter in a word is scored according to its point value, which is inversely proportional to its frequency in English words. In Scrabble, the points are allocated as follows:

Points	Letters
1	**A**, **E**, **I**, **L**, **N**, **O**, **R**, **S**, **T**, **U**
2	**D**, **G**
3	**B**, **C**, **M**, **P**
4	**F**, **H**, **V**, **W**, **Y**
5	**K**
8	**J**, **X**
10	**Q**, **Z**

For example, the Scrabble word **FARM** is worth 9 points: 4 for the *F*, 1 each for the *A* and the *R*, and 3 for the *M*. The **ConsoleProgram ScrabbleScore** reads in words and prints out their score in Scrabble, not counting any of the other bonuses that occur in the game. The program ignores any characters other than uppercase letters in computing the score. In particular, lowercase letters are assumed to represent blank tiles, which can stand for any letter but which have a score of 0.

Adding commas to numeric strings

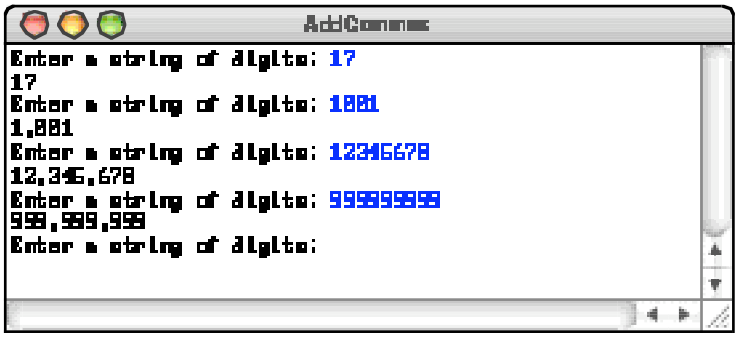
When large numbers are written out on paper, it is traditional—at least in the United States—to use commas to separate the digits into groups of three. For example, the number one million is usually written in the following form:

```
1,000,000
```

To make it easier for programmers to display numbers in this fashion, the program **AddCommas** implements a method

```
private String addCommasToNumericString(String digits)
```

that takes a string of decimal digits representing a number and returns the string formed by inserting commas at every third position, starting on the right. For example, a sample run looks something like:



Deleting characters from a string

The program **RemoveChar** implements a method

```
public String removeAllOccurrences(String str, char ch)
```

that removes all occurrences of the character **ch** from the string **str**. For example:

```
removeAllOccurrences("This is a test", 't')  returns  "This is a es"
removeAllOccurrences("Summer is here!", 'e') returns  "Summr is hr"
removeAllOccurrences("---0---", '-')         returns  "0"
```

Displaying the lines of a file in reverse order

The program **ReverseFile** does this.

Counting the words in a file

The program **WordCount** reads a file and reports how many lines, words, and characters appear in it. Suppose, for example, that the file **lear.txt** contains the following passage from Shakespeare's *King Lear*, which might serve as a guide to how government officials responded to the devastation of Hurricane Katrina:

```
Poor naked wretches, wheresoe'er you are,
That bide the pelting of this pitiless storm,
How shall your houseless heads and unfed sides,
Your loop'd and window'd raggedness, defend you
From seasons such as these?  O, I have ta'en
Too little care of this!
```

Given this file, your program should be able to generate the following sample run:

For the purposes of this program, a word consists of a consecutive sequence of letters and/or digits, which can be tested using the static method **Character.isLetterOrDigit**. Also, the program does not try to count the characters that mark the end of a line, which will have different values depending on the type of computer.