Lab 6 – Cryptography

CSCI 1101B – Spring 2015

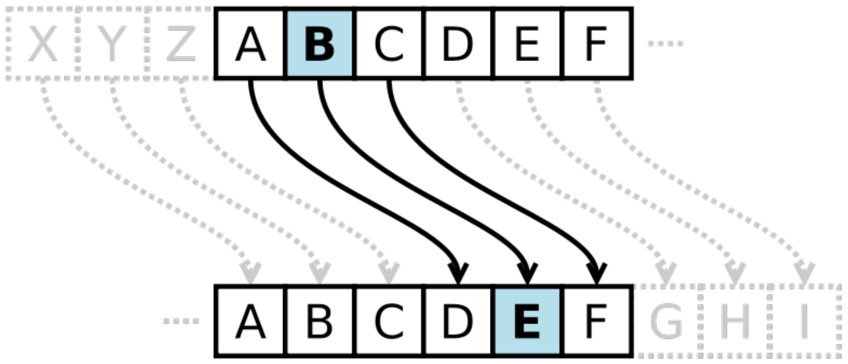
Due: April 10, 10 pm (note extra days)

Objective: To gain experience using `Strings` and `chars`.

The Scenario: You have a message you would like to send to someone else so that no one else can read it. *Cryptography* provides a solution to this problem (and many others like it) in which data must be protected from access by unauthorized parties. Cryptography (and security in general) is an important issue in many areas of computing, such as allowing you safe access to websites that require you to supply a username and password.

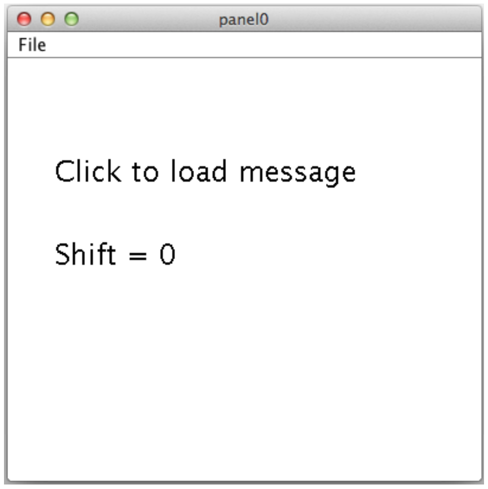
In this lab, you will design a program that implements a very simple cryptographic method known as the Caesar cipher, named after Julius Caesar who (supposedly) used it to communicate with his generals. In this type of cipher, letters are simply shifted a fixed number of letters. For example, if the shift amount were 3 (as in the illustration below), `A` would become `D`, `B` would become `E`, `C` would become `F`, etc. Note that if the letter being encrypted is shifted beyond the end of the alphabet, this technique "wraps around" and continues the shift from the beginning if the alphabet, e.g., `Y` shifted by 3 would become `B`.

In order to decrypt a message (i.e., recover the original message from the encrypted message), you would need to know the shift amount and then apply the cipher in reverse to the encrypted message. Of course, you could pretty easily break this particular cryptographic method by just trying every possible shift amount until you guessed the correct number, which is why real systems use much more sophisticated encryption methods than the Caesar cipher. But this remains a useful exercise in the basic principle of cryptography!



Program Behavior

The window when your program first starts should look like this (font size = 24):



Click to load message

Shift = 0

The behavior of the program should comprise three phases:

1. Loading the (unencrypted) input message.
2. Setting the shift number (how many places to shift each letter during encryption).
3. Encrypting the input message using the Caesar cipher.

Clicking on the "Click to load message" text will read the input (unencrypted) message from a text file called `message.txt` located in your lab folder and display the message in the window (replacing the "**Click to load message**" instructions with "**Message: secret message here**", if `message.txt` contains "secret message here"). The message should contain **only** upper-case letters and spaces and should not be empty (i.e., "" is not a valid input message). If the message does not comply with these rules, loading the message should simply display the error message "**Invalid input, try again**" and allow the user to load the message again by clicking on the text, thus allowing the user to first fix the contents of the `message.txt` file.

Once the user has loaded a (valid) message, the program should allow the user to set the shift number. Initially, the shift number is zero (i.e., no encryption). Clicking on the Shift message will increment the counter once per click, up to a maximum shift amount of 25.

Once the shift counter is at least 1, clicking on the "Message: ..." text should start encryption. Once encryption has started, pressing on the shift counter should no longer have any effect. Each click on the message text should encrypt a single letter, starting with the first letter, and display the updated, partially encrypted message with each click. Spaces should be encrypted as question marks rather than shifted. As mentioned above, if the shift of a letter would take it beyond `Z`, it should "wrap around," i.e. go back to the beginning of the alphabet and continue counting from there. For example, if we were encoding an `X` with the shift of 5, it would be encrypted as a `C`.

Once the message has been completely encrypted, pressing anywhere should do nothing.

Refer to the example sequence of program states at the end of the handout.

Getting Input

So far we have not read textual input from the user. In this lab, we need to read a message from a text file. This is easy to do using the built-in `Scanner` and `File` classes. To use these classes, first add these two lines to your `import` statements:

```
    import java.util.Scanner;
    import java.io.File;
```

Then, to read the first line of the file called `message.txt` as a `String` variable called `input`, you can simply do the following:

```
    String input = new Scanner(new File("message.txt")).nextLine();
```

You can assume that the entire message to encrypt is contained in the first line of the input file (i.e., don't worry about files containing multiple lines).

However, one potential issue we have to deal with is that the file `message.txt` may be missing, in which case we will not be able to read it. Java provides a feature called **exceptions** to deal with these kinds of unexpected situations. While you do not need to understand the details of exceptions here, you will need to wrap your file reading in a `try/catch` block like this:

```
   String input = "";
   try {
       input = new Scanner(new File("message.txt")).nextLine();
   } catch (Exception e) {
       // could not read message.txt
   }
   // now do something with input
```

In the above code, your program will try to read the `message.txt` file, and if it fails (likely due to the file being missing), then the code inside the `catch` block will execute. If that happens, then the `input` variable will not get assigned the message, which is why we need to give the `input` variable a default value (above, simply the empty string “”).

If your program cannot read the `message.txt` file, then you should simply show the standard error message “**Invalid input, try again**”, the same as if you *could* read the message but it was invalid.

Program Structure

Most of the code of your program will go in the `onMousePress` method, since clicking is what drives the behavior of your program. However, you should write an extra method called `isCorrectFormat` that will assist by checking whether the input message is valid. The `isCorrectFormat` should be sent a `String` object and return `true` if that String is in the correct format (only upper-case letters and spaces and has at least one character) and `false` otherwise.

This method will be called from within your `onMousePress` method. While you *could* simply write the code of the `isCorrectFormat` method directly inside `onMousePress`, writing this code inside a separate method cleanly separates components of your program, makes it easy to reuse the `isCorrectFormat` method, and simplifies the `onMousePress` method. This type of method is called a *helper method*, since it is designed to be used by other methods in the same class (the `Events` class, in this case), rather than called externally from other classes.

Helper methods are written in exactly the same way as all the methods we’ve written so far, except that they are marked `private` instead of `public` (i.e., the first word in the method declaration), which simply indicates that the method can *only* be called from within the same class. Note that instance variables are all marked `private` as well, since instance variables are also only accessed from within the class itself (you cannot refer to an instance variable of another class because they are labeled `private`).

Tips

You can easily create the `message.txt` file by renaming the default `README.TXT`

Remember that each character of a `String` is represented by a `char` primitive, which is really just a number that identifies that character. For example, the number 5 might represent a capital A, the number 6 might represent a capital B, and so forth. Thus, to shift a `char` variable using the Caesar cipher, you can just add the shift amount to the `char` variable (though you will need to account for the wrapping behavior as well).

However, one issue is that whenever you perform arithmetic on a `char` variable (e.g., adding an `int` value to a `char` value, or even adding two `char` values together), Java transforms the result into an `int`. What this means is that if `c1` and `c2` are `char` variables and you write something like this:

```
    // shift c1 by c2 places and store the shifted character into c3
    char c3 = c1 + c2;
```

you will get a compiler error complaining about trying to store an `int` expression in a `char` variable. To fix this, you need to force Java to treat the right-side expression as a `char` by **casting** it to the appropriate type:

```
    // force Java to treat (c1 + c2) as a char instead of an int
    char c3 = (char) (c1 + c2);
```

In the above example, we are computing the value `c1 + c2` (which is an `int`) and then forcing Java to treat the summed expression as a `char` expression.

Remember that a `char` primitive is specified using **single-quotes**, instead of **double-quotes** which are only used for `Strings`:

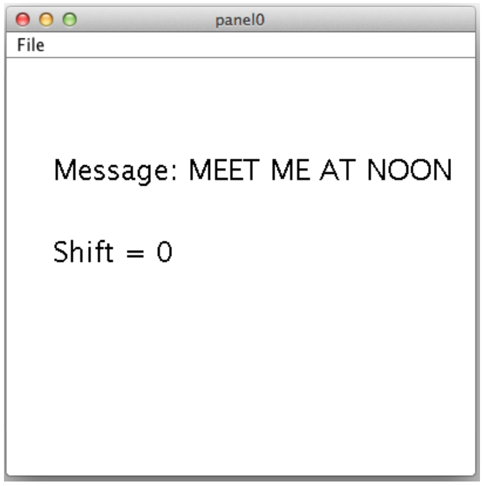
```
char c = '?'; // this is a char (NOT a String)
String s = "?"; // this is a one-character String (NOT a char)
char c2 = "?"; // this is invalid, cannot store a String in a char!
```

Submitting Your Work

Please submit the program in the usual way via Blackboard. Remember that we need your whole project folder (the one with the name composed of your login ID and lab number) and it must be compressed.

Please include a sample `message.txt` file in your submission (the contents can be any reasonably short message).

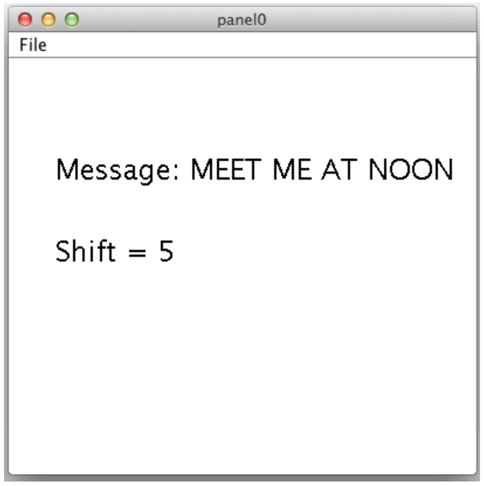
Example Program Sequence



Message: MEET ME AT NOON

Shift = 0

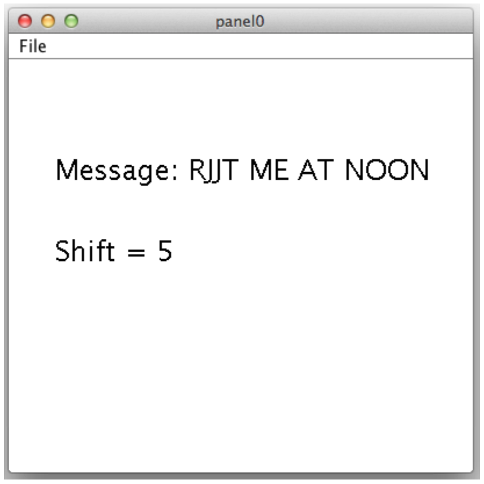
1. After loading the message



Message: MEET ME AT NOON

Shift = 5

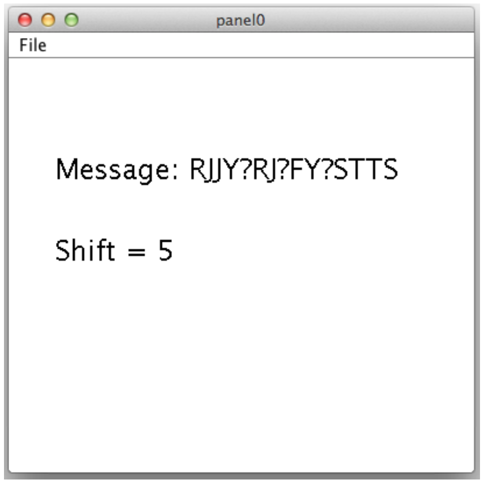
2. After clicking on Shift five times



Message: RJJT ME AT NOON

Shift = 5

3. After clicking on Message three times



Message: RJJY?RJ?FY?STTS

Shift = 5

4. After encrypting the rest of the message