Assignment #4—Cryptography

YEAH hours: 7:00–8:30 P.M., Wednesday, February 10, TBA
Due: Wednesday, February 17, 5:00 P.M.
Note: This assignment may be done in pairs

Last year’s Hollywood blockbuster The Imitation Game (which won the Academy Award for Best Adapted Screenplay even though the writers got most of the technical details wrong) showed movie audiences how a group of cryptographers in England managed to break the German military code, which was called Enigma. Your task in this assignment is to replicate some of their work. You, however, are fortunate in having access to modern computing, allowing you to use Java instead of primitive electronics.

This assignment is divided into three parts, which are substantially independent even though you will be able to reuse some code. This handout focuses on what you have to do to complete the assignment. The background you need on cryptography and the Enigma machine appears in Handout #29.

Part 1: Checking keys in a letter-substitution cipher

Your first task consists of adding a method to the code for the letter-substitution cipher, which we developed together in class on February 1. The starter project includes a working version of the program file LetterSubstitutionCipher.java—reproduced in full as Figure 1 on the next page—which is primarily intended to give you a start on the rest of the assignment. As long as you enter a valid key, the program does exactly what you would want. It accepts a key and a plaintext message and then turns that message into the corresponding encrypted ciphertext. A sample run of the program, just as you have it in the starter folder, might look like this:

![Figure 1: Letter-substitution cipher. Enter 26-letter key: QWERTYUIOPASDFGHJKLMZCVBNM
Plaintext: WORKERS OF THE WORLD UNITE!
Ciphertext: VGKAYKL GY ZIT VKKSR XFOZT!](image)

The key in a letter-substitution cipher is a string that shows the enciphered counterpart of each of the 26 letters of the alphabet in order. In this example, the user has entered the key "QWERTYUIOPASDFGHJKLMZCVBNM" (rather unimaginatively generated by typing the keys in order on the keyboard), which corresponds to the following mapping:

```
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
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
The translation of an uppercase letter into its ciphertext equivalent is implemented in the following line from the `encrypt` method:

```java
ch = key.charAt(ch - 'A');
```

The expression `ch - 'A'` subtracts the character code for the letter A from the character stored in the variable `ch`, which is guaranteed by the logic of the program to be an uppercase character. The result is therefore an integer between 0 and 25 indicating the index of the character `ch` from the beginning of the alphabet. Calling `key.charAt` with this index returns the character at the corresponding position in the key.
As the comments at the beginning of Figure 1 indicate, the only thing you have to do for
this part of the assignment is add code to check whether the key is legal. A key string in
a letter-substitution cipher is valid only if it meets the following two conditions:

1. The key is exactly 26 characters long.
2. Every uppercase letter appears in the key.

These conditions automatically rule out the possibility that the key contains invalid
characters or duplicated letters. After all, if all 26 uppercase letters appear and the string
is exactly 26 characters long, there isn’t room for anything else.

Your submitted version of the code for LetterSubstitutionCipher.java must ask the
user for a key, check the key to make sure it is legal. If it is not, your program should tell
the user that the key is invalid and allow the user to try again. Your code, for example,
should be able to duplicate the following sample run:

The first key entered by the user is only eight characters long and is therefore too short to
be a legal key. The second attempted key is the right length but is missing the letter Z.
The user gets it right on the third try.

Part 2: Inverting a letter-substitution key

Letter-substitution ciphers require the sender and receiver to use different keys, one to
encrypt the message and one to decrypt it when it reaches its destination. Your task in
this part of the assignment is to write a method that takes an encryption key as described
in Part 1 and returns the corresponding decryption key. In cryptography, that operation is
called inverting the encryption key.

The idea of inverting a key is most easily illustrated by example. Suppose, for example,
that the encryption key is "QWERTYUIOPASDFGHJKLZXCVBNM" as in the first example from
Part 1. That key represents the following translation table:

| 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 |

The translation table shows that A maps into Q, B maps into W, C maps into E, and so on.
To turn the encryption process around, you have to read the translation table from bottom
to top, looking to see what plaintext letter gives rise to each possible letter in the
ciphertext. For example, if you look for the letter A in the ciphertext, you discover that
the corresponding letter in the plaintext must have been K. Similarly, the only way to get
a B in the ciphertext is to start with an X in the original message. The first two entries in the inverted translation table therefore look like this:

\[
\begin{array}{c c c}
A & B \\
\downarrow & \downarrow \\
K & X
\end{array}
\]

If you continue this process by finding each letter of the alphabet on the bottom of the original translation table and then looking to see what letter appears on top, you will eventually complete the inverted table, as follows:

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

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

The inverted key is simply the 26-letter string on the bottom row, which in this case is "KXVMCNOPHQRSZYIJADLEGWBUFT".

To complete Part 2 of the assignment, you have to implement the method

```java
private String invertKey(String key)
```

that generates a decryption key for the encryption key passed as the argument. For example, if you were to call

```java
invertKey("QWERTYUIOPASDFGHJKLMNBVNCXZM")
```

you should get back "KXVMCNOPHQRSZYIJADLEGWBUFT", as shown in the earlier example.

You also need to write a run method that asks the user for an encryption key, displays the inverted decryption key, and then validates the result by taking advantage of the fact that inverting a key twice gives you back the original key. The following sample run produces the last line by calling invertKey on the value shown on the preceding line:

Make sure your solution checks that the key is legal, just as you did in Part 1.

**Part 3: Simulating the Enigma machine**

Your main task in the assignment—and the one that is the most important in terms of its historical significance—is simulating the function of the Enigma machine. Although the Enigma machine seems complex, it is really just a series of letter-substitution ciphers implemented using the rotors in a way that forces the letter permutation to change on each key press.

This part of the assignment requires you to implement two classes: EnigmaSimulator and EnigmaModel. The subsections that follow describe how the classes needed for Part 3 fit together and what you need to do to implement them.
The Enigma Simulator class

The Enigma simulator is the first program in which you need implement more than one class. The main program class is `EnigmaSimulator`, which contains the `run` method. The purpose of `EnigmaSimulator` is to interact with the user. The program begins by asking the user for the rotor order, which allows the user to choose three of the five rotors supplied with the Enigma machine (these five rotors are called the stock rotors) and determine the order in which these rotors are inserted into the machine. For example, if the user enters 513 as the rotor order, that means that the Enigma machine should be set up to use stock rotor 5 as the slow rotor, stock rotor 1 as the medium rotor, and stock rotor 3 as the fast rotor. The rotor numbers read from left to right because that’s how they are positioned in the physical machine.

The program then asks the user for the rotor setting, which is the three-letter string that indicates how the rotors should be rotated so that they correspond to the codebook settings for the day. For example, if the rotor setting is the string "JLY", the rotors should be set so that the letters J, L, and Y appear in the windows on the Enigma panel. The program then asks the user to enter a plaintext string to be encrypted. Once the program has all the necessary data, it calls the necessary methods in the `EnigmaModel` class to simulate the encryption of the plaintext into the corresponding ciphertext, which is displayed on the console.

The following sample run of the `EnigmaSimulator` program illustrates these steps:

```
Enter rotor order: 513
Enter rotor setting: JLY
Enter a plaintext line: HELLOWORLD
The encoded ciphertext: OYCEMFIYHY
```

This example shows that, given the rotor order 513 and rotor setting "JLY", the plaintext "HELLOWORLD" gets enciphered into "OYCEMFIYHY". That string seems entirely meaningless, but the magic of the Enigma machine is that typing the enciphered message into a machine that starts with the same settings reproduces the original plaintext string, as follows:

```
Enter rotor order: 513
Enter rotor setting: JLY
Enter a plaintext line: OYCEMFIYHY
The encoded ciphertext: HELLOWORLD
```

The `EnigmaSimulator` class itself is simple to write because the `EnigmaModel` class does most of the hard work. The only requirement for this class beyond implementing the steps described earlier in this section is that your program should allow the user to reenter the rotor order and rotor setting values if the user enters an illegal value. The actual checking for legality is done by the `EnigmaModel` class, but your implementation has to notice that `EnigmaModel` class has rejected one of these values and then allow the user to try again.
The EnigmaModel class

Up to now, most of the classes you’ve written—including the EnigmaSimulator class in Part 1—have been complete programs that extend one of the Program subclasses. In this part of the assignment, your task is to implement a class that is used as a resource for a program class that you implement independently. The EnigmaSimulator class therefore functions as a client of the EnigmaModel class. In coding EnigmaModel, you are taking on the role of implementer, which means that you need to understand how the Enigma machine actually works and then translate your understanding into Java.

The public entries in the EnigmaModel class appear in tabular form in Figure 3. The first entry is the constructor, and the rest are the public methods. The EnigmaModel.java file that comes with the starter project includes a full set of javadoc comments for the constructor and the public methods but the implementations of those methods have been left up to you. Those incomplete methods—which in some cases include temporary code to ensure that the methods return a value of the appropriate type—are generally called stubs. For example, the stub for getRotorSetting is

```
public String getRotorSetting() {
    return ""; // Replace this line with the actual implementation
}
```

The stub returns the empty string just to make sure that the program compiles correctly.

Figure 2. Public entries in the EnigmaModel class

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new EnigmaModel()</td>
<td>Creates a new instance of the EnigmaModel class, which simulates the operation of the Enigma machine. By default, the constructor uses the first three stock rotors as the slow, medium, and fast rotors.</td>
</tr>
<tr>
<td>setRotorOrder(order)</td>
<td>Sets the rotor order to order, which is a three-digit integer giving the numbers of the stock rotors to use. For example, calling setRotorOrder(513) uses stock rotor 5 as the slow rotor, stock rotor 1 as the medium rotor, and stock rotor 3 as the fast rotor. The method returns true if the rotor order is legal (the digits are between 1 and 5 and no digit appears more than once) and false otherwise.</td>
</tr>
<tr>
<td>setRotorSetting(str)</td>
<td>Sets the rotor setting to str, which must consist of three uppercase letters. The method returns true if the rotor setting is legal and false otherwise.</td>
</tr>
<tr>
<td>getRotorSetting()</td>
<td>Returns the three-letter string that corresponds to the current rotor setting, which advances every time a key is pressed.</td>
</tr>
<tr>
<td>encrypt(plaintext)</td>
<td>Encrypts the plaintext string by passing each letter through the various rotors of the Enigma machine. All letters in the string are converted to uppercase, and the rotors of the Enigma machine are advanced before translating each letter. If a character in the plaintext string is not a letter, the rotors do not advance and the character is simply copied to the output unchanged.</td>
</tr>
</tbody>
</table>
The starter file also includes private definitions for the five stock rotors, each of which is a 26-character string that specifies the permutation that rotor performs when a signal flows through it from right to left. For example, the definition of `STOCK_ROTOR_1` is

```java
private static final String STOCK_ROTOR_1 =
    "EKMFLGDQVZNTOWYHXUSPAIBRCJ";
```

which means that `STOCK_ROTOR_1` corresponds to the following permutation:

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

Similarly, the starter file contains a definition of the permutation for the Enigma reflector.

Your primary task in Part 3 is to implement the constructor and the four public methods in the `EnigmaModel` class. To do so, you need to decide what instance variables are required to store the state of the Enigma machine and what private methods would help make your implementation easier to write.

**Tracing the Enigma encryption path**

As described in Handout #29, each of the three rotors in the Enigma machine implements a permutation, just as that concept is used in the first two parts of the assignment. By default, the `EnigmaSimulator` constructor initializes the machine to use the first three stock rotors as the slow, medium, and fast rotors, respectively. At the beginning, the rotors are all set to their default starting position in which the A character is visible through the window on top of the machine, as shown in Figure 4.

![Figure 4. Initial setting of the Enigma machine using the first three stock rotors](image)
Ignoring for the moment the fact that the rotors advance whenever a key is pressed, each character runs through a fixed series of seven encryption steps. Pressing a key causes an electrical signal to be fed into the corresponding wire at the right edge of the machine. That signal then passes through the fast rotor, where it gets shifted to some other position in the alphabet. From there, the signal flows through the medium rotor, the slow rotor, and the reflector. It then makes a return trip through the rotors in left-to-right order. The complete path is shown in Figure 5, which shows the signal starting at A and ending at R.

Each individual stage in the translation is nothing more than a simple letter-substitution cipher. The first stage, for example, energizes the A wire at the right of the fast rotor, which implements the following permutation:

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

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

The signal emerges from the fast rotor at the B position and then moves on through the medium rotor, where it is translated into J:

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

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

The slow rotor then maps the J into Z:

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

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

Figure 5. Path taken by the signal when input A is energized
These three translation steps take the signal up to the reflector, which you can see in the highlighted path in Figure 5.

The reflector is also a letter-substitution cipher, which is defined as a private constant in the `EnigmaModel` starter file. The reflector takes the \( Z \) emerging from the slow rotor and maps it into \( G \):

\[
\begin{align*}
A & B C D E F G H I J K L M N O P Q R S T U V W X Y Z \\
\text{I} & X U H F E Z D A O M T K Q J W N S R L C Y P B V G
\end{align*}
\]

The signal then goes back through the three rotors in the opposite direction. When the signal is running from left to right, however, you can’t use the permutations directly because the translation has to happen “backwards.” What you need, therefore, is the inverse of the original permutation. The good news is that you’ve already written `invertKey` for Part 2 and can use the same code here. The return trip through the slow, medium, and fast rotors therefore looks like this:

\[
\begin{align*}
A & B C D E F G H I J K L M N O P Q R S T U V W X Y Z \\
W & Y G A D E P V Z B E C K M T H X S L R I N Q O J
\end{align*}
\]

\[
\begin{align*}
A & B C D E F G H I J K L M N O P Q R S T U V W X Y Z \\
J & P C Z W R L F B D K O T Y U Q G E N H X M I V S
\end{align*}
\]

\[
\begin{align*}
A & B C D E F G H I J K L M N O P Q R S T U V W X Y Z \\
T & A G B P C S D O E U F V N Z H Y I X J W L R K O M
\end{align*}
\]

Just as Figure 5 shows, the signal ends up at \( R \).

**Advancing the Enigma rotors**

Despite the fact that the Enigma machine goes through these seven letter-substitution steps, the encoding scheme you’ve built so far is still an easily decrypted cipher. What makes the Enigma code hard to break is that the permutations implemented by the rotors change for every letter. Whenever the operator presses down on any key, that action advances the fast rotor by one position, which occurs before the letter is translated. If that rotation carries the setting of the fast rotor past \( Z \), the fast rotor cycles back to \( A \) and then advances the medium rotor one step. When the medium rotor makes a complete revolution past \( Z \), the slow rotor advances. Thus, the fast rotor changes on every character, the medium rotor advances once every 26 characters, and the slow rotor advances once every 676 (26 \times 26) characters.

Physically, the operation of advancing a rotor one notch corresponds to turning the rotor one position so that the next letter appears in the window. Simulating that operation in the underlying implementation is a two-step process. In the first step, the permutation string associated with the rotor shifts by one character position, cycling the original first character to the end of the string. In the second step, every character in the new string must move one step closer to the beginning of the alphabet, so that \( Z \) becomes \( Y \), \( Y \) becomes \( X \), and so on. This operation is also cyclical, which means that the \( A \) in the permutation string becomes \( Z \).
For example, the first stock rotor uses the following permutation string when it is in the A position, which is simply the definition of the \texttt{STOCK\_ROTOR\_1} constant:

"EKMFGLDQVZNTOWYHXUSPAIBRCJ"

If you rotate this rotor to the B position, the first step is to move the first letter to the end of the string, which produces the following permutation:

"KMFLGDQVZNTOWYHXUSPAIBRCJE"

The second step in the process is to create the correct permutation string by subtracting one from the internal values of each of these characters so that it becomes the preceding letter, letting A wrap around to Z. The updated permutation string is therefore

"JLEKFCPUYMSNVXGWTROZHAQBID"

These operations are straightforward to implement using the methods in Java’s \texttt{String} class, but you are unlikely to get this part right unless you test your code as you go. You need to create a method

\begin{verbatim}
private String advanceRotor(String permutation)
\end{verbatim}

that takes a permutation string and returns the new permutation string that results from performing these two operations. Write a test program and make sure that calling

\begin{verbatim}
advanceRotor("EKMFGLDQVZNTOWYHXUSPAIBRCJ")
\end{verbatim}

gives you back the string \"JLEKFCPUYMSNVXGWTROZHAQBID\".

Keeping track of the current permutation implemented by each rotor does not give you quite enough information to implement all the operations in the \texttt{EnigmaModel} class. Several of the methods need to keep track of what letter appears in the window on the top panel of the Enigma. That information cannot be determined from the current value of the permutation string. After advancing the rotor once, your implementation needs to know that it is in the B position, but there is no way to figure that out just from looking at the string \"JLEKFCPUYMSNVXGWTROZHAQBID\". Keeping track of the current position of each rotor in an instance variable will make your life much easier.

\textbf{Putting it all together}

The operations described in the preceding sections are all you need to implement the \texttt{encrypt} method in the \texttt{EnigmaModel} class. For each letter in the plaintext string passed to \texttt{encrypt}, the implementation has to perform the following operations:

1. Advance the fast rotor one notch, which may in turn advance the medium and slow rotors, depending on whether the process of moving the rotor forward generates a “carry” into the next rotor position.
2. Send the letter to be encoded through the seven-step path through the three rotors from right to left, around the reflector, and then back through the three rotors from left to right.
3. Append the character generated by this process to the output string. (Any characters other than letters that appear in the plaintext string should simply be copied directly into the output string, just as they are in the \texttt{LetterSubstitutionCipher} program.)
The other operations in the `EnigmaModel` class are generally easier to implement than `encrypt`, although you will need to think carefully about your implementation strategies.

**Strategy and tactics**

As with all assignments in CS 106A, the most important advice is to start early. You have two weeks to complete this assignment. If you start the night before it is due, things are not likely to go well. Make milestones for yourself. For example, you might want to get Part 1 working by the end of this week and Part 2 by next Monday. Doing so means that you’ll have more than a week to do Part 3, which accounts for most of the complexity.

The following tips will also probably help you do well on this assignment:

- *Try to get into the spirit of the history.* Although this project is an assignment in 2016, it may help to remember that the outcome of World War II once depended on people solving precisely this problem using tools that were far more primitive that the ones you have at your disposal. Computing sometimes matters a great deal, and there will probably be situations in your lifetimes when the consequences of solving some programming challenge will be just as important. The fate of the world may some day lie in your hands, just as it did for the cryptographers at Bletchley Park.

- *Draw lots of diagrams.* Understanding how the Enigma machine works is an exercise in visualizing how the machine works. A picture will be worth a thousand words here.

- *Test your program in pieces.* Don’t try to get Part 3 of this assignment running all at once. Before you figure out how to advance the rotors on each key press, make sure that you can successfully encrypt a character given a fixed position of the rotors. Once you have that working, you can move on to other tasks. You should also make sure to test the `advanceRotor` method described in the previous section separately before you integrate it into the application as a whole.

- *Debug your program by seeing what values appear in the variables.* When you are debugging a program, it is far more useful to figure out what your program *is* doing than trying to determine why it *isn’t* doing what you want. Every part of this assignment works with strings, and you can get an enormous amount of information about what your program is doing by using printing out the value of the strings you’re using at interesting points. Unfortunately, the `EnigmaModel` class doesn’t have direct access to the program console, but you can nonetheless display data on the Eclipse console by calling `System.out.println`.

- *Check your answers against the demonstration programs.* The class web site includes working versions of each part of this program. Your code should generate the same ciphertext given a particular rotor order, rotor setting, and plaintext string. The web site also has a graphical simulator for the Enigma machine.

- *Come to office hours.* The Enigma simulator is a new assignment this quarter, which means that the section leaders may not know exactly how everything works until they’ve had a chance to play with it. In addition to my regular office hours (Tuesdays from 9:30 to 11:00 and Wednesdays from 4:00 to 5:00), I will drop by the LaIR from time to time to help out.
Possible extensions

There are many things you can implement to make this assignment more challenging. Here are just a few ideas:

• **Add graphics.** If you like working with graphics, you could change the main program so that it simulates the operation of the Enigma machine on the screen. If you do, you shouldn’t have to change the `EnigmaModel` class at all. Whenever you get a mouse click in one of the Enigma keys you’ve displayed on the screen, you can simply pass that character to `encrypt` as a one-character string and then take the character you get back and use that to determine which lamp to light. The overall design of the program uses a programming pattern called MVC (for model-view-controller) in which the responsibilities of different classes are divided among these three roles. The `EnigmaModel` class is (naturally enough) the model, which keeps track of the internal state. The graphics code you write and the main program implement the view and the controller roles.

• **Implement other parts of the Enigma machine.** The German wartime Enigma designs are actually more complicated than the model presented here. In particular, the Enigma machines used by the military branches included a plugboard (*Steckerbrett* in German) that swapped pairs of letters before they were fed into the rotors and after they came out.

• **Simulate the actions of the Bombe decryption machine.** This assignment has you build a simulator for the German Enigma machine. Constructing a mechanical simulator, which the Bletchley cryptographers called a *Typex machine*, was critical to the decryption effort, but was not as important as the Bombe, which went through many possible rotor settings mechanically looking for ones that would generate the correct patterns, as described in the extended notes on the Enigma machine. You’ll have a chance to implement the search for crashes in section, but you might also try to figure out more about how the Bombe worked and build more of its pieces.