CS2133: Computer Science II
Assignment 4

Prof. Christopher Crick

1 Caesar Cipher (30 points)

The Caesar cipher is a (very insecure) method for encrypting text dating back to the Romans. It is the same alphabetic shift cipher as is used in the Secret Decoder Rings that no longer come as prizes in breakfast cereals and Cracker Jacks. Each letter in a text is replaced by the letter that occurs some fixed number of positions later in the alphabet. For instance, if the cipher implemented a shift of 3 positions, 'A' would be transformed to 'D', 'B' to 'E', 'L' to 'O' and so forth. Letters at the end of the alphabet wrap around to the beginning, so 'Z' would get coded as 'C'. To decode the message, simply reverse the shift.

Java characters are represented in Unicode, which is a complex character representation that is supposed to allow writing in the alphabet of every human language, including Cyrillic, Tagalog, Hmong, Egyptian hieroglyphics, Chinese and (ahem) Klingon. However, the first 128 characters in Unicode are the same as the English-only ASCII representation that has been around since the 1950s. In ASCII, printable English characters start at 32 (*space*) and end at 126 (*tilde*), with all of the alphabetic characters in between. The letter 'A' is 65, 'B' is 66, 'a' is 97, etc. We can generalize the Caesar cipher to handle all of these characters, using an arbitrary shift size as our key. The algorithm is, then:

```
if charValue < 32 or charValue > 126 outputChar = (char)charValue // leave alone
charValue = charValue + key
// We include both of these so that we can encode by using a positive number
// and decode using the same number as a negative
if charValue > 126 charValue = charValue - 95 //(127 - 32)
if charValue < 32 charValue = charValue + 95
outputChar = (char)charValue
```

Write a command-line Java program (no GUI this time!) that can be run as follows:

```
java Caesar key infile [outfile]
```

The program should open the file *infile* and encrypt each character according to *key*, a number. It should write the encrypted text to *outfile*, if provided, or to the screen if not.

The program should exit gracefully and print out a useful error message if it runs into any problems, including:

- Incorrect arguments

- *infile* not found or not readable
- *outfile* not writeable

Extra Credit Implement the Vignere cipher, which takes a word instead of a number as a key. Use the ASCII value of each letter of the key in turn as the shift, starting over from the beginning of the key when you run out of letters. Thus if the key is "hot", the first letter of the message should be shifted by 104 (ASCII 'h'), the second by 111 (ASCII 'o'), the third by 116, the fourth by 104 again, and so on. Also make sure that there is a way to tell the program to decode as well as encode.

2 Web Browser (35 points)

Write a *very simple* GUI web browser. Your program should have a text edit box at the top of the window for the user to type a URL. Provide a listener that activates when the user types something in and presses "Enter".

The HTTP specification is fairly simple. Create a Socket object from the URL's web address, connecting on port 80 (the default HTTP port). You will probably want to attach a PrintWriter and a BufferedReader to the Socket. Send the following lines of text, ending in both carriage return and newline characters (yes, HTTP follows Windows line-ending conventions, rather than Unix):

```
GET <filepath> HTTP/1.1
Host: <web address>
<blank line>
```

In other words, if the user typed "http://cs.okstate.edu/students.html", your program should send these three lines to the Socket:

```
GET /students.html HTTP/1.1\r\n
Host: cs.okstate.edu\r\n
\r\n
```

Don't forget to flush your output afterwards, or your request will never be sent.

The web server will respond with a bunch of text including the web page, which you should read into a String. In the program panel, display (as plain text) the body of the webpage (explained below). Make sure that you set up the display so that it includes a scroll bar when needed. You must also handle exceptions and errors gracefully. If the user enters a nonexistent URL, for instance, or if the HTTP response is formatted incorrectly, then the program should inform the user.

A proper HTTP response will look like the following:

```
HTTP header junk
Several lines
User doesn't
want to see this stuff
<html some other formatting>
<head maybe some additional stuff>
Javascript definitions
Other stuff the user doesn't care about.
```

```
<title>The Title of the Webpage</title>
</head>
<body onload="main()" might be telling the browser to run Javascript>
Here's all the stuff the user really cares about.
<h1>Along with a bunch of formatting,</h1>
<a href="http://another.webpage.org">links,</a>
<img src="http://another.webpage.org/lolcat.jpg" and some other image info>
and images.
</body>
</html>
```

Browsers display HTML by following the instructions provided by tags delineated by angle brackets. The first word inside a bracket specifies the kind of tag; tags can then have other information inside, and then end with a closing bracket. All of the text after a tag is formatted according to the tag's instructions, until a closing tag is encountered. These tags are also delineated by angle brackets, and include a slash and then the name of the tag. Thus, in the example, the line "Along with a bunch of formatting," is enclosed by the "h1" tag, which instructs the browser to display it with a font size indicating that it is a first-level heading. According to the HTML specification, browsers are not required to implement all of the scores of tags that exist, and they should ignore any tags they do not recognize. Your browser will ignore almost all of them. It will be able to do the following:

- Extract the webpage's title from between the ⟨title⟩⟨/title⟩ tags and change the title of your program frame appropriately.
- Display all of the text located between the ⟨body ... ⟩ and ⟨/body⟩ tags, and nothing outside of the body.
- Ignore every other tag. Don't print out any angle brackets or any of the text located inside of them.

Needless to say, once you have the webpage loaded into a String, you will be using a lot of substring() and charAt() calls to massage the response into the webpage you will display to the user.

Note: Java has a few interesting classes that will render HTML themselves, such as the JEditorPane. They're fun to play around with, and might even be useful to you as a first step to getting your program working. However, for the purposes of this assignment, you must implement your own simple renderer, working from the simple string of plain text you get from a web server.

Note: More and more web pages are using the encrypted "Secure HTTP" protocol, which is denoted "https" in the URL. Your browser will be unable to access these web pages (including common ones like google.com). Don't assume your browser is broken when testing your code until you have verified that you are not trying to access an encrypted page.

Extra Credit Implement another few HTML tags. Some suggestions: Headings (⟨h1⟩, ⟨h2⟩, etc) should change the font size. Images (⟨img src=URL⟩) could be displayed via Toolkit.getImage(URL). Hyperlinks (⟨a href=URL⟩) could be implemented as buttons or clickable text (this is nontrivial, but it would make your browser into a true websurfing app, which would be pretty cool).

3 Robot Teleoperation (35 points)

The robot you will be controlling is a flying quadrotor, the Parrot ARDrone. There is a fair amount of software infrastructure that goes into translating movement commands into differential rotor speeds that drive the robot around, but I (and others) have taken care of that for you. The robot is listening for instructions delivered over the network in a particular format; your task will be to send those instructions based on the controls your application user is sending.

Since not all of you have a robot to test your code with, I have written a robot simulator that sits on the web. You can point a web browser at "http://lear.cs.okstate.edu/robot_sim.html". When you connect with your application and send the correct messages, the robot on the web will move appropriately. Note that there is only one simulator running; if several people are testing their programs at the same time, the simulated robot will be responding to all of their instructions at once. The results may be amusing.

Upon starting up, your program should establish a connection with the robot. This is done by opening a Socket for writing at IP address "lear.cs.okstate.edu" and port "9095". Notice that this port is located at the same address as the web server; that's because the robot simulator is running there and piping its output to the web page. After the assignment is turned in, you will have the opportunity to use your controller to fly a real robot, which might have a different IP address. Make sure it's easy to change.

The messages you will be sending to the robot are encoded in a format called JSON, which is a simple text format for sending objects over the internet. It is simple, but it is not terribly easy to work with in Java, because JSON strings include a lot of quotation marks. And as you well know, quotation marks begin and end Strings in Java, which means that assembling strings with internal quotation marks is a little messy – you have to escape them with backslashes.

A generic JSON message looks like the following:

```
{"name1":"stringvalue1", "name2":numericvalue, "object":{"instance":"value"}}
```

Thus, it is a series of name-value pairs, with curly brackets denoting object nesting levels and quotation marks surrounding every string (but not numbers).

Your program will send three different kinds of messages to the robot: a takeoff instruction, a land instruction and a move instruction. The JSON for the takeoff instruction is:

```
{"op":"publish","topic":"/ardrone/takeoff","msg":{}}
```

In order to construct this string in Java, you will have to write something like the following:

```
String takeoff_msg = "{\"op\":\"publish\",\"topic\":\"/ardrone/takeoff\",\"msg\":{}}";
```

It's messy, but it's the way JSON is formatted and the way Java functions.

The message for landing is very similar:

```
{"op":"publish","topic":"/ardrone/land","msg":{}}
```

The movement message is significantly more complicated, because you are sending actual information. It contains six numbers corresponding to the linear and angular velocities along the robot's various axes. This is what a message telling the robot to stop moving looks like:

```
{"op":"publish",
 "topic":"/cmd_vel",
 "msg":{"linear":{"x":0,
                  "y":0,
                  "z":0},
        "angular":{"x":0,
                   "y":0,
                   "z":0}}}
```

I've inserted formatting line breaks and spaces to make it easier to read; the actual message is all one line just like the takeoff and land messages.

The robot is able to move in four of these six dimensions. A positive linear x number tells the robot to move forward, negative means backward. Linear y is the left/right dimension, and linear z is the up/down (altitude) dimension. All of these values are in meters/second. A positive angular z velocity means rotate to the left, negative means to the right, in radians per second. Although movement messages also contain fields for angular x (roll) and angular y (pitch), this particular robot cannot rotate freely around those axes, so those fields should always be 0.

After you have connected the Socket and sent the handshake message, you can send these JSON messages as often as you like. Make sure you flush your output buffer after you send each message.

Your program should provide the user with a set of controls – exactly how they are displayed and used is up to you. The controls should allow the user to takeoff and land, move forward and backward, up and down, right and left, and turn. When the user indicates that she wants the robot to move in a specific way, the program must construct the appropriate JSON message and send it over the wire. Note that the program must also figure out how and when to send a message for the robot to stop, otherwise it will keep going until it crashes. If the messages are being formatted and sent correctly, the behavior will show up in the robot simulator. Note that you will get no feedback if they are *not* correct (except that the darn robot won't move); think about how you can print out intermediate results for testing. The robot is also a physical object that is subject to physical laws and physical damage; I suggest limiting your linear speeds to +/- 0.25 m/s and your angular speed to +/- 1 radian/s.

Note: The JSON specification is unnecessarily strict about what a number should look like. Decimals between -1 and 1 *must* have a leading zero digit. ".25" is wrong; it must be written "0.25".

Note: The robot in the simulator is facing down when the webpage is first loaded, so a command to move forward should make the image travel down toward the bottom of the screen, and a command to move left will send it to *the robot's* left, which is to the right, viewed from your perspective – at least until you start sending it commands to turn.

Extra Credit: Add adjustable speed controls to your user interface that make the robot speed up and slow down.

Turning in

The files with the main() function that actually execute the programs should be named Caesar.java, Browser.java and Robot.java, so we know which ones we're supposed to run. Those three, plus all of the other .java files that define the classes you need for these programs, should be wrapped up in a zip file called assignment_4_your_name.zip and uploaded to the Dropbox at oc.okstate.edu.

Ensure that everything can be compiled and run from the command line. This assignment is due Wednesday, October 19, at noon.