Lab 1
Exploring Internet Email Protocols, Java and BlueJ
Report due: In class on Wednesday, February 10, 2016

GOALS: Learn about two important Internet email protocols: SMTP (Simple Mail Transfer Protocol), which is the standard protocol used to send email messages, and POP (Post Office Protocol), which is often used by mail clients when reading email. Learn to use BlueJ to edit and run Java programs.

REQUIREMENTS:

• Read Chapter 1 and the first section of Chapter 4 of Programming with Java, Swing and Squint before coming to lab.

• In every lab except this one you must read the entire lab handout before arriving at lab. This lab is a tutorial and is intended to be read during the lab period.

INFORMATION:

• We have created an email account for you on fuji.cs.williams.edu. Your username is your class year followed by your OIT username. So if your OIT username is jrl2 and your class year is 2019 then your email address is 19jrl2@fuji.cs.williams.edu.

• We will give you your email account password at the beginning of lab. Your password will be displayed in plain text during the lab. At the end of lab, please change your password. Information on how to change your password appears at the end of this document.

• Part of this lab involves working in pairs. Please find a partner and work adjacent to one another. This handout will let you know when you should work in pairs and when you should work alone.

Part I: Exploring Email Protocols
You will use three programs to explore how Internet mail protocols function:

Thunderbird: You will use the Thunderbird mail client to send and read email messages using your mail account on fuji.cs.williams.edu.

TCPCapture: This is a program which allows you to intercept and inspect packets1 sent to or from your machine through the Internet. You will use it to determine exactly what packets are exchanged between the Thunderbird email client and our email server.

NetTap: This is a program that allows you to exchange packets directly with a remote server rather than depending on a client program like a mail program or a web browser.

MAIL ACCOUNT CONFIGURATION:
To begin, you should configure the Thunderbird email client to access your account on fuji by completing the following steps:

1 In order to send an email message, a client program sends several messages of its own. It is confusing to use the word “message” for both sorts of communications. So in the rest of this handout we will use the word “message” only when talking about actual email messages. We will use the term “packet” to refer to the smaller, individual communications sent between clients and servers.
• Working alone, log in to your computer.
• Press Command+Spacebar to activate Spotlight and type Thunderbird then press Return.
• A dialog box will appear. Select Skip Integration.
• Another dialog will appear. Select Skip this and use my existing email.
• The Mail Account Setup dialog will appear. Enter your name in Name, full email address (e.g., 19jrl2@fujl.cs.williams.edu) in Email Address and your password in Password. Click Continue. Thunderbird will take a few seconds to contact the server.
• The Mail Account Setup dialog will change. For security reasons, deselect Remember Password. Now select POP3, then select Manual config. Adjust the settings under Incoming and Outgoing so that they are the same as those below (but use your own username):

![Mail Account Setup](image)

• Select Done. A warning dialog box will appear. To enable you to examine the packets exchanged between the mail client and server we have had you configure Thunderbird to send packets without using any encryption. This would usually be a bad way to configure your mail client. Check the box I understand the risks, then select Done.

SEND YOURSELF SOME MAIL:

Using your favorite webmail interface (e.g. Gmail, etc) send two messages to your fujl.cs.williams.edu email address.
Using TCPCapture

- You should start working in pairs now. Choose either of your two email accounts for the next few steps. It does not matter which account you choose now. You will switch to the other one later.
- Open Spotlight by typing Command+Spacebar and type Utilities. Select Utilities from the list of folders displayed by double-clicking. The Utilities folder will open.
- Open TCPCapture by double-clicking its icon.
- A window like the one on the right should appear. Near the top of the window is a menu that controls the type of packets TCPCapture will intercept. It will initially display HTTP. These are the initials for “Hypertext Transfer Protocol,” the protocol used to talk to web servers. We want to look at the packets sent when the SMTP protocol is used to send mail. So, you should change the setting of this menu to SMTP. In the next menu, select en0 (the name of the computer’s Ethernet interface). Then, click Start. The program will now display any SMTP packets sent to or from your machine. The cow icon at the top of the program’s window will start running to indicate that it is monitoring network traffic.
- Let’s send a mail message. Bring Thunderbird back to the foreground by clicking on its icon in the dock. Next, click on the Write icon (it’s the pencil with sheet of paper).
- Fill in the To field with your partner’s email address. The address you enter should ends with @fuji.cs.williams.edu As you do this, Thunderbird may request access to your contacts. Decline this request.
- Add any short message or subject you would like. Then click on the (kinda lame) stamp-shaped icon Send to send the message.
- Once this is done, bring the TCPCapture window back to the top of your desktop. It should now look something like the image shown below. The program’s window is now filled with several lines of text. Each of these lines corresponds to a packet sent between the mail client program and our email server. The lines that start with
  fuji.cs.williams.edu -> CLIENT correspond to packets that travelled from our server to your machine. Lines that start with
  CLIENT -> ...
  correspond to lines your machine sent to our server. Obviously, sending just a single email message involves sending many packets through the network.
The remaining text on each line shows the contents of the packet that was actually sent. For example, the first packet sent from the client to the server was:

```
EHLO tcl217a-21.cs.williams.edu
```

which might translate into English roughly as “Hello, I’m tcl217a-21.cs.williams.edu” (this happen to be the network address of the machine used while preparing this handout).

Some of the packets sent between the client and server are too long to fit on one line of the screen. They are truncated in the summary display, but there is a way to see the entire contents of such packets. If you click once on the summary line for a packet, its complete contents will be displayed in the region at the bottom of the TCPCapture window. For example, near the end of the exchange between the client and server there should be a packet containing the text

```
354 End data with <CR><LF> . <CR><LF>
```

If you click on the line immediately below this packet, the program’s window should display something like the image shown on the right. The text should look familiar. It probably isn’t surprising that one of the packets exchanged between a client and a server when sending an email message actually contains the email itself!

You can use the scroll bar to move through the contents of the packet that appears at the bottom of the window. If you scroll to the top, you will find three lines that are not actually part of the packet. Instead they specify who sent the packet, who received the packet, and what protocol was used.

Now that we have found the packet that contains the email being sent, we should try to figure out what the other packets are for.

First, note that each of the packets sent from the client to the server other than the email message itself starts with a four letter code. Each code is a command name. The commands you should find in your TCPCapture window and their uses are explained below:

- **EHLO** - identify the client machine to the server.

- **MAIL** - provides the return address for the message’s sender. The command name must be followed by “FROM: <sendername@somemachine name>” where sendername@somemachine name should be the email address of the sender.

- **RCPT** - provides the destination address to which the message should be delivered. The command name must be followed by “TO: <user@somemachine name>” where user@somemachine name is the email destination address for the message.

- **DATA** - indicates that the message itself will follow immediately.

- **QUIT** - Terminates the connection.

Similarly, if you look at the lines sent to the client from the server, they all start with three digit codes like 220 and 250. The 220 code means ready while 250 means that the server accepted the last client packet. In fact, these numerical codes are the only critical information contained in the lines sent from the server.
The client program doesn’t really need to look at any of the text that follows. The text is there to help any human who happens to be reading the packets exchanged.

There are many other commands and responses recognized by SMTP servers, but these are the most important. If you want to learn about the others, visit:

http://en.wikipedia.org/wiki/SMTP

or read the complete and official description of the SMTP protocol at:

http://www.faqs.org/rfcs/rfc2821.html

(You will not want to read the complete description, but you may find it interesting to take a peek.) Do NOT close or clear your TCPCapture window just yet.

Using NetTap
When a mail server receives packets like those you intercepted using TCPCapture, it doesn’t know what program they came from. In our case, we used the Thunderbird email client, but the mail server would accept the packets just as happily from Microsoft Outlook, Apple Mail, or any other mail client. In fact, the server doesn’t even care whether they came from a mail client. All it cares about is that the packets somehow got sent to it through the network. To illustrate this, we have constructed a program that isn’t a mail client but can be used to send packets to an SMTP server. The program is named NetTap.

- Open NetTap.jar in the Utilities folder by double-clicking on its icon.
- A window like the one shown below should appear. Near the top of the window there is a text field labeled Server name. You should type the name of our server, fuji.cs.williams.edu, into this field.

A single machine might be running several different server programs at the same time. For example, fuji.cs.williams.edu is constantly running a POP server program, an SMTP server program, and an HTTP (web) server. When this machine receives a packet it needs a way to decide which of these three programs is the intended destination for the packet. This is accomplished by associating a standard “port number” with each protocol.

A port number is a bit like a telephone extension number. When a packet is sent it is addressed not just to a machine but to a particular port on that machine. The port number associated with the SMTP protocol is 25. By default, NetTap assumes this is the port number to which you would like to send packets. Later, when you experiment with the POP protocol, you will have to enter its port number (110) so that the packets sent by the program will be delivered to the POP server program.
Once the server name and port information has been provided, you can connect NetTap to the indicated server by simply pressing the Connect button.

If you look back at the TCPCapture window in which you intercepted the packets sent between the Thunderbird email client and our mail server, you will notice that the first packet was sent from the server. The packet started with the code 220 indicating the server was ready. You will notice that a similar packet has now been sent from the SMTP server to the NetTap program. The program will display this packet in its window preceded by a "<" to indicate that it is an incoming packet.

After sending the “220” packet, the server waits to receive an “EHLO” packet. You can send the expected packet using NetTap by typing its contents into the field to the right of the “SEND” button and then pressing “SEND.” Feel free to cheat and copy/paste the text of the packet from the dialogue you recorded using the TCPCapture program (the copy shortcut is command + “C” key, paste is command + “V” key). The packet will be sent to the server and displayed in the NetTap window preceded by “-> “.

Continue to mimic the dialogue you recorded using TCPCapture until you have sent the “DATA” command. Make sure you receive a 250 response to each line you enter before moving on to the next line. If you take a long time at any step in this process, the server may get impatient and disconnect you. If this happens, just connect again and start over a little faster.

After receiving the DATA command, the server expects you to send the body of the email message you want to send. The Thunderbird email client and most real email clients) includes many things in addition to the text of the message including a subject line and the date. You don’t need to mimic all this. Just enter the message line-by-line and enter a line containing a single period when you are done.

That’s it. You have just sent an email message without the help of an email program! Later on in the lab, we will check to make sure it was actually delivered by using the Thunderbird email client. For now, send the server a “QUIT” command to terminate your conversation.

If you left TCPCapture running while you were using NetTap (you probably did since we never told you to stop it), take a look at the packets it captured while you were using NetTap. After inspecting the packets, clear the TCPCapture window.

**Experimenting with POP**

We can also use TCPCapture and NetTap to explore the function of one of the protocols used to retrieve email, the POP protocol. This time, we will approach the process in the opposite order. First, you will learn about the types of packets that can be sent to a POP server by entering them yourself using NetTap. Then, you will use TCPCapture to see how the Thunderbird email client actually uses this protocol.

Like SMTP, the POP protocol requires that the client start each packet it sends to the server with a four letter command name. The commands you will need are discussed below. As with SMTP, if you wish to learn more, you might either read the Wikipedia discussion of POP:


or read the official description:

http://www.ietf.org/rfc/rfc1939.txt

In this case, the official description is actually short enough that you may find it readable.

The first two commands you will need identify the account whose mail is to be accessed:

**USER** - The first packet sent to the server should be composed of the code “USER” followed by a user name (such as 19jr12).
PASS - The second packet sent to the server should be composed of the word “PASS” followed by the account password.

When you are all done you will enter a very simple command:

QUIT - This command requires no additional information.

In between, you may want to use three other commands:

STAT - Asks the server to send you a concise message indicating how many mail messages are available.

RETR - Asks the server to send you the contents of one of your messages. Each retrieve request should include the number of the message to retrieve. Try a small number like 1 or 2.

DELE - Deletes a message. Each delete request should include the number of the message to delete.

The POP server sends packets back to the client in response to each command it receives. These packets begin with a code indicating whether the command was acceptable. A packet from the server that starts with “+OK” indicates that the server is happy, while a packet starting with “-ERR” indicates distress. A POP server usually includes explanatory information after the “+OK” or “-ERR” summary code.

With this background, you should be able to use NetTap to connect to our department’s POP server and read the mail that has been sent to your account on our server.

If you are working in a pair, you should switch from the account you have been using to send messages to the other partner’s account (the account to which you sent the messages) for this portion of the lab. To begin, start up NetTap and TCPCapture on this second account. (If you are working alone, just press Clear in the NetTap and TCPCapture windows.)

Change the NetTap Protocol Port Number/Name to 110 by either selecting the appropriate menu item or just typing in the number. Leave the server name set as fuji.cs.williams.edu. Then, click Connect to start your conversation with our POP server. Rather than receiving a “220” packet to let you know that the server is ready, you will instead receive a packet that starts with the code “+OK.” POP and SMTP are like different languages.

Once you are connected, send “USER” and “PASS” commands to the server to log in (remember to use the account to which you just finished sending messages). As we warned you, your password will now appear in plain text on the screen. The system should respond with a message telling you that your mailbox is ready.

Now, pick a message number between 1 and the total number of messages available and send an “RETR” command to the server requesting that message. (If you are not sure how many messages are there, use the “STAT” command.) Access a few of your messages this way. You may have to look carefully to recognize them since mail programs surround the simple text of your email messages with lots of control information. Finally, send a “QUIT” command to the server and quit NetTap.

How Thunderbird Uses POP
As a final step in this investigation of the SMTP and POP protocols, we would like you to use TCPCapture to record the packets exchanged between Thunderbird and our POP server when you use Thunderbird to read email from our server.

First, go back to the TCPCapture program. Set the TCPCapture protocol menu to POP and press Clear and Start. Next, switch to Thunderbird and press the envelope icon to retrieve your mail. The program
may ask for your password. In a few moments, the Thunderbird window should fill with a list of the headers of all the messages waiting in your mailbox.

Look at the end of this list of messages to make sure the message you sent using NetTap arrived just as well as the message you sent using the Thunderbird program.

Now, go back to the TCPCapture window and examine the exchange of packets that flowed between Thunderbird and our POP server.

We would like you to use TCPCapture to explore two questions about how the Thunderbird program uses the POP protocol. Reading the online references on POP described above should provide useful information to help you, but you should expect to have to do some experimenting with Thunderbird, NetTap and TCPCapture to try to figure out how Thunderbird is using POP. We do not expect you to provide complete answers to these questions. We do, however, want you to explain what you have been able to learn and how.

1. First, when you look at the TCPCapture window you will notice that the Thunderbird program uses two commands we have not discussed, “UIDL” and “LIST”. How and why do you think Thunderbird uses these commands? It may help if you send yourself some more messages (or delete some messages by issuing DELE commands through NetTap), quit and restart the Thunderbird application, and use TCPCapture to see what messages are exchanged when you get your new mail again. Try selecting Get Mail.

2. Second, what does the Thunderbird program do when you click on the delete button in its interface after selecting one of your message in the message list? Delete a message in this way while running TCPCapture to see what happens. Again, use a combination of the references we suggested and your ability to experiment with NetTap and TCPCapture to explain what you observe.

As part of your submission for this lab, we ask that you turn in during class on Wednesday, a brief, typed report with your thoughts on these two questions. If you are working with a partner a single, joint report is fine.

At this point you can quit TCPCapture, NetTap and Thunderbird.

**Part II: Java Programming with BlueJ**

Today you will construct two simple Java programs. The first will display the following dialog box:

The dialog box will not actually function as an authentication mechanism, it is just a facade with no machinery behind it. Creating this facade, however, will introduce you to many aspects of Java programming.
If you are working with a partner, you can do this part of the lab on either account. We do, however, suggest that you take turns doing the typing. In fact, we will remind you to switch a bit later in case you forget.

**Using BlueJ to Enter Program Text**

Begin by launching the BlueJ application: Activate Spotlight by pressing Command+Spacebar, then type BlueJ and press Return. Once BlueJ is running:

- Say “No Thanks” to the dialog box inviting you to participate in BlueJ data gathering.
- Select New Project from the BlueJ Project menu and use the dialog box that appears to navigate to the Documents folder in your account directory.
- Name your project. If your name happens to be Herb Gritz and your partner is Sally Floyd, then GritzFloydLab1 would be a great project name. Make sure that your project folder name **ALWAYS** includes your name and **NEVER** contains blanks or special characters. Enter this name in the field labeled File: at the top of the dialog box. Click Create.
- BlueJ should now display a project window with your project name in its title bar. Click the New Class... button in this window. In the context of this lab, “class” just means “program”. That is, when you tell BlueJ to create a new class you are telling it that you want to start writing a new program.
- BlueJ will display another dialog box asking you to name your class and to tell it what kind of class it is. Select **GUIManager** for the type of the class. Name your class LoginWindow. Press OK.
- At this point, an icon with the name you picked for your class will appear in the project window. Double-click on this icon to display a window showing the text of your class.

Because you told BlueJ that your class would be a **GUIManager**, the text that initially appears is a template with skeletal definitions of many common components of programs you will write this semester. It will save you a little typing to use the skeletal code provided in the template. Let’s explain, from top to bottom what this code does.

The first two lines

```java
import squint.*;
import javax.swing.*;
```

inform BlueJ that your program depends on two libraries of code designed to support Java programmers. Swing is a standard library provided by Oracle (but originally developed by Sun Microsystems; Oracle acquired Sun in 2010). Squint is a library developed specifically for this course.

The next few lines

```java
/**
 * Class LoginWindow – write a description of your class here
 *
 * Enter your name here
 *
 */
```

form what is called a comment. The computer ignores all the text sandwiched between `/* ... */` when it interprets your program. Such text, however, can be very helpful to the grader or any other person who reads your program. Edit this comment to provide the information it suggests. At a minimum, enter your name.
GUIManager: A Framework for a New Program

The next line

    public class LoginWindow extends GUIManager {

marks the beginning of the text that actually describes how your new class should behave. It is called the class header.

The class header itself provides two important pieces of information about your program. It indicates that the program’s name will be LoginWindow and that the program will depend on library mechanisms to manage a graphical user interface (i.e., it will be a GUI manager). Note that when programming in Java, capitalization matters! In other words, Java is case-sensitive.

The description of how the new class should behave follows this header and is surrounded by a matching pair of opening and closing braces. Note the open brace that appears at the end of this line. If you scroll to the very bottom of the text in the window you will find the matching closing brace at the very end. All the lines in between are part of the description of the LoginWindow class.

The description of the class can be broken down into three sub-parts: variable name declarations, the constructor, and method definitions. We describe each of these parts below.

Variable Name Declarations

The line

    private final int WINDOW_WIDTH = 400, WINDOW_HEIGHT = 400;

declares two names, WINDOW_WIDTH and WINDOW_HEIGHT, and assigns them integer values. In Java, a name that refers to information like a number or an objects like a button is called a variable. The word private means that no code outside of the LoginWindow class may see these variables. The word final means that no other integers may be assigned to WINDOW_WIDTH and WINDOW_HEIGHT (i.e., their assignment is final). Because the values of these variables can not change, we call them constant values, and by convention, capitalize their names.

The line before this variable declaration that starts with two forward slashes is another form of comment. Anything that appears on a line after // will be ignored by Java.

The Constructor

The next lines of code

    public LoginWindow() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
    }

define a constructor. The constructor describes the steps that should be performed immediately when your program is run. Just as a pair of braces surround our entire class description, the constructor instructions are placed between a pair of braces following the constructor’s header. In this case, the single instruction

    this.createWindow( ... , ... )
which creates a visible window when the program starts, appears within these braces.

Shortly, we will add more instructions to the constructor. These instructions will add labels and text fields to the window. But let’s first see what this very simple constructor makes the computer do by running the program before you even edit it.

Before you can run a program, you must compile it into instructions that the computer can understand. To compile your program simply click on the Compile button in the window containing the text of your LoginWindow program. Make sure the phrase “no syntax errors” appears at the bottom of your program window. If not, check things over or ask an instructor or teaching assistant for help until it does.

- Go to the project window and point the mouse at the icon for the LoginWindow class. Now click the mouse button while pressing the control key (i.e., control-click the LoginWindow icon). A new menu will appear.
- You can run your program by selecting the first item in this menu. It should look like “new LoginWindow()”. Then click OK in the create object window that appears. This tells BlueJ to create an instance of your class and to execute the instructions you placed in its constructor. A red icon for this new instance will appear in the bottom of the project window. A square empty window should appear on the screen. (You may have to work a bit to find it because it may appear behind some of the other windows.) The appearance of this empty window is the result of running your program. Congratulations!
- Press the mouse while pointing at the “BlueJ Virtual Machine” item in the menu bar and select the “Quit BlueJ Virtual Machine” item. This is the preferred way to remove the window your program created from the screen. Try to get in the habit of using the “Quit” menu item when you are finished running one of your programs rather than just clicking on the little red circle in the upper-left corner of the window.
- Modify your program by replacing the number 400 in the line declaring the WINDOW_WIDTH so that it creates the window with width 200. Then compile and run your program again. The window that appears this time should be tall and narrow (though still empty). The two numbers you type in the “createWindow” line determine the width and height of the window.
- Select the “Quit BlueJ Virtual Machine” item again to remove this new window from your screen.

Now that you have the framework required to define your constructor, you can place more instructions there. The areas in which a user can enter a user name and/or password in the completed program are called JTextFields in Java. The simple text labels that appear in the window to identify these fields are called JLabels. To get these items into the window, we have to add commands telling the computer to make new fields and labels and to add these to the window. For the first pair, the correct commands are

```java
contentPane.add( new JLabel( "Username: " ) );
contentPane.add( new JTextField( 8 ) );
```

The “8” in the second command specifies how wide the field should be.

- Add these lines to your program. They belong in the constructor after the “createWindow” line.
- Press “Compile”. If there are any errors, fix them and press “Compile again”; repeat as often as necessary.
- Once again go to the project window and point the mouse at the icon for the LoginWindow class. Now press the control key, and then press the mouse button to make a menu appear.
- Select the first item in the menu ( “new LoginWindow()”). and then click OK in the create object window that appears. Your program’s window should now display a field labeled Username.
If you are working with a partner and have not done so already, this would be a good time to pass the keyboard from one partner to the other.

With just a bit of thought, you should now be able to figure out how to add the labeled field for entering the password. Give it a try. Add the extra lines required and then compile and run your program to see that they do what they should.

Now, we need to add an “Authenticate” button. The four lines you just added to the program all look like

```java
contentPane.add( ... );
```

All that changes from one command to the next is the “new” component we are adding to the window. In Java, buttons are called JButtons. The phrase you should use to create an “Authenticate” button is just

```java
new JButton( "Authenticate" )
```

- Add a `contentPane.add` command to add such a button.
- Compile and run the program to see if it works.

**Methods**

If you ignore all of the single-line comments, the remaining lines in this program have the form

```java
public void buttonClicked() {
}

public void textEntered() {
}

public void menuItemSelected() {
}
```

This text defines three **methods**. A method is a group of instructions associated with a name. Like the constructor and the class itself, the framework for a method consists of a header and a pair of curly braces that enclose the instructions. The names of the three methods in this program clearly hint at their functionality. When a button is clicked, the instructions between the braces that follow the `buttonClicked` header are executed. Likewise for `textEntered` and `menuItemSelected`.

Right now, since there are no instruction between the braces of the `buttonClicked` method, if you press the button in your program’s window, the computer does nothing. This may be appropriate, since we can’t really verify the user id and password in any way, but it would be nice to make it react in some way. One simple possibility is to have it announce “Login Rejected!” as soon as you press the button. To do this, add the instruction

```java
contentPane.add( new JLabel( "Login Rejected!" ) );
```

within the braces that delimit the body of the `buttonClicked` method. Once again, compile and run your program to see that it behaves as expected. See what happens if you try to authenticate twice.

Next, try adding a line to display “Login Accepted!” in the body of the `textEntered` method. Compile and run your program and try to get it to display this message. Hint: You need to press “Enter.”
When you are all done, the `menuItemSelected` method will still be empty. To keep your program tidy, delete the entire method by removing all the text from the comments that precede the `menuItemSelected` header to the “}” that ends the method body. Deleting such unnecessary components of the program template that BlueJ gives you when you start a program is a matter of good style.

**A More Responsive Program**

The program you just created produced a nice looking interface including several text fields, but it did not do anything with the information a person running the program typed into the text fields it displayed. For your second programming experience, we would like you to create another program that uses information entered by its user in a very simple way.

The program is inspired by the web site [http://www.madglibs.com](http://www.madglibs.com) which was in turn clearly inspired by the word game Mad Libs. When you go to this web site, you get to pick one of several Mad Libs stories to complete. The web site then presents you with a web page containing a collection of labeled text fields that look something like:

![Be Kind Mad Lib](image)

After you fill in the text fields with words of the types indicated (as we have with “worm”, “worlds”, etc.), the program displays the text of a story obtained by substituting the words you provided into a standard template. For example, our words produced:

![Be Kind Mad Lib](image)

For your program, you may use the “Be Kind” story shown above, any other Mad Lib from the madglibs.com site, or any Mad Lib that strikes your fancy. In our instructions however, we will assume you are using the “Be Kind” story.

**Using Variables**

Begin, as you did for your first program by pressing the “New Class...” button in the BlueJ project window. Again select “GUIManager” as the class type. Since “angry” is pretty close to “mad” and Mad Libs is a “word” game, enter “AngryWords” as your class name and press “OK”.

13
In the editor window for the new class, change the comments included at the top of the template to include your name(s). Since you won’t be using them for this program, immediately delete the `textEntered` and `menuItemSelected` methods.

Eventually, your program’s interface should look something like:

![Image of the interface](image1.png)

and when it is run it should produce a result like:

![Image of the output](image2.png)

You could construct a program that produced the desired interface just as you constructed the interface for your `LoginWindow` program. Unfortunately, if you did so, you would be unable to access the words that a user typed in the program’s text fields. Therefore, you have to take a different approach to creating the desired interface. To show you what you need to do, we will lead you through the steps required to produce a program containing just the first text field of the desired interface. That is, we will show you have to build a program whose interface looks like:

![Image of the interface](image3.png)

We will also show you how to include code so that the program will produce just the first phrase of the Mad Lib when a user presses the COMPOSE button. Then, you can use the techniques employed in this simple program to construct a program for a more complete Mad Libs.
To write a program that can access the text typed in a JTextField, you need to associate a variable name with the JTextField. There are several steps required to accomplish this. First, you must pick a name for the JTextField. The name must start with a letter and consist only of letters and numbers. It is a Java convention to start with a lower-case letter. You can use your imagination, but it is helpful to pick a name that will help someone reading your program understand the purpose of the item associated with the name. So, for the first text field names like noun1, nounField, or firstNoun might be appropriate. (Blanks are not allowed within a Java name, but it is common to make names out of multiple words by just capitalizing the first letter of every word.)

Let’s go with noun1.

You must tell Java the name you have chosen by declaring the name. To do this, you would place a line of the form

```java
private JTextField noun1;
```

between your class header and the first line of your class constructor. Putting it right after the line from the GUIManager template that declares WINDOW_WIDTH and WINDOW_HEIGHT will work well.

Next, you must associate a particular JTextField with the name by including the name and the construction of a new text field in an assignment statement like

```java
noun1 = new JTextField( 10 );
```

This line belongs inside the program’s constructor, after the line that uses createWindow. It will create a text field and associate a name with it, but it will not actually add the field to the program’s window because it does not ask the contentPane to add it.

In your previous program, you added a JLabel and a JTextField to the program’s window using commands like

```java
contentPane.add( new JLabel( "NOUN" ) );
contentPane.add( new JTextField( 10 ) );
```

Now, since you have already associated the name noun1 with the text field you want to display, you can instead write

```java
contentPane.add( new JLabel( "NOUN" ) );
contentPane.add( noun1 );
```

Add code similar to what we have just described to your program together with code to add a “COMPOSE” button to the contentPane. Compile and run your program to verify that it produces the desired interface.

**Using Accessor Methods**

Now that you have a name associated with the JTextField, you can use a method named getText to tell the computer to access the field’s contents.

In your program’s buttonClicked method, place a single line of the form

```java
contentPane.add( new JLabel( "Be kind to your " + noun1.getText() ) );
```
Like several commands you have already used, this command tells the computer to add a label displaying some text to its display. The command, however, illustrates two new features. First, rather than specifying what the label should display by simply placing the desired text between quotes, this command uses a plus sign to tell the computer that the desired text should be formed by adding together two shorter fragments of text, the phrase “Be kind to your” and the text obtained by asking the text field named noun1 to provide whatever text the program’s user has typed into this field. getText is called a method. The pair of empty parentheses after the name getText are required.

Compile and run this version of the program. If it is correct, then after you press the COMPOSE button, both the original GUI components and the text starting with “Be kind...” should appear in the program window.

Carrying on
The plus sign can be used to combine any number of fragments of text. For example, if you replace the phrase:

“Be kind to your " + noun1.getText()

with the phrase:

“Be kind to your " + noun1.getText() + "-footed"

the program will add the the suffix “-footed” to the text displayed by the new JLabel. Another plus sign could be added after this to add even more text.

With this in mind, we would like you to add more declarations of variables and code to your constructor so that when first run your program includes several labeled text fields in its display. Practice makes perfect, but it can also get tedious. So, you do not need to do a complete Mad Lib if you feel you have gotten the hang of this after adding a total of four or more text fields to your interface.

Next, add code to your buttonClicked method so that the words entered in all of your program’s text fields get incorporated into the Mad Libs story your program displays when the COMPOSE button is pressed. Use a separate JLabel for each line of the Mad Lib story.

Finishing Touches
Given the limited tools you have at this point to control the layout of components in your program’s window, you will probably find that your interface looks pretty messy, particularly after you press the COMPOSE button. You do not need to worry about making your interface beautiful. You don’t have to tools to control the layout of components precisely yet. There are, however, a few things you can to do make things look better.

First, adjusting the size of the program’s window should probably be enough to make the program’s initial display look better. You can do this by grabbing the window’s lower right corner with the mouse and dragging to get an estimate of what size works well. Then, you can change the values in the lines declaring WINDOW_WIDTH and WINDOW_HEIGHT to make the improved size permanent.

If you are not satisfied with this approach, another technique is to join each text field to the JLabel that is supposed to precede it by putting the two together in a JPanel and then adding the JPanel to the contentPane. An explanation of how to use JPanels in this way can be found in section 2.5 of Programming with Java, Swing and Squint which you can access at

http://dept.cs.williams.edu/~cs134/squintchapters/SquintCh2.pdf
Finally, in our sample solution shown above, you may have noticed that we made the COMPOSE button and all of the JLabels and JTextFieldS initially displayed disappear once the COMPOSE button is pressed. To accomplish this, just add the following two lines at the start of your buttonClicked method:

```java
contentPane.removeAll();
this.repaint();
```

Another option you are welcome to play with is adding a line like

```java
this.setSize( FINAL_WINDOW_WIDTH, FINAL_WINDOW_HEIGHT);
```

to the beginning of your buttonClicked method. This will allow you to use different window dimensions for the initial program interface and the display of the final Mad Libs. You will need to include appropriate declarations for FINAL_WINDOW_WIDTH and FINAL_WINDOW_HEIGHT in this case.

**Submission Instructions**

Once your programs work correctly, quit BlueJ, return to the Finder, and look in your Documents folder. You should find the folder that BlueJ created for your project. Its name should be the one you picked for your project (something like GritzFloydLab1).

- Click on the Desktop, then go to the Go menu and select Connect to Server.
- Type “afp://guest@fuji” for the Server Address and click Connect.
- A window will appear where you should select the volume Courses to mount and then click OK.
- A Finder window will appear where you should double-click on cs134.
- Drag your project’s folder (whose name should look like GritzFloydLab1) into either “Monday-Dropoff” if you are in the Monday afternoon lab, “Monday-Evening-Dropoff” for the Monday evening lab, or “Tuesday-Dropoff” if you are in the Tuesday afternoon lab. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click OK.

**Finishing up**

When you are all done, don’t forget to reset your password.

- Open the System Preferences (you can do this using the Apple menu)
- Click on Users & Groups
- Click on Change Password...
- Change your password and click Change Password
- Your password is now changed.

Then, quit all the applications you have been using during the lab and get ready to log out. To log out, press the mouse on the apple icon at the upper left corner of the screen and select the “Log Out” item from the menu that appears.