Lab 1
Exploring Internet Email Protocols, Java and BlueJ

GOALS: Learn about three important Internet email standards: SMTP (Simple Mail Transfer Protocol), which is the standard protocol used to send email messages, POP (Post Office Protocol), which is often used by mail clients when reading email, and MIME (Multipurpose Internet Mail Extensions), which determines how mail programs format the contents of the messages they deliver. 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 kerry.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 2020 then your email address is 20jrl2@kerry.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 kerry.cs.williams.edu.

**TCPCapture:** This is a program which allows you to intercept and inspect packets 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.

---

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.
MAIL ACCOUNT CONFIGURATION:
To begin, you should configure the Thunderbird email client to access your account on kerry (our mail server) by completing the following steps:

- Working alone, log in to your computer.
- Press the leftmost icon in the “Dock” at the bottom of the screen to activate the “Finder” and ensure that a finder window is displayed on the screen as shown in the image below.

- Click on the name “Applications” in the list of “Favorites” found in the leftmost section of the finder window to make the system display the available applications in the main section of the finder window.
- Find the icon for Thunderbird (highlighted with a rectangle in the image above).
- Double-click on the Thunderbird icon to launch the Thunderbird mail client. This may take a while.
- Decline if Thunderbird requests permission to access your contacts or to update your version of Thunderbird.
- A dialog box will appear. Uncheck the “Always perform this check …” option and then select Skip Integration. If given the choice, tell it not to “always check” for this at startup.
- 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., 20jrl2@kerry.cs.williams.edu) in Email Address but DO NOT enter your password in Password. Also uncheck the “Remember password” checkbox.
- Click Continue. Thunderbird will take a few seconds to contact the server.
• The Mail Account Setup dialog will change. Make sure that “Remember Password” is unchecked in the new dialog box. Now 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 dialog](image.png)

- Press “Advanced config”. Select “Server settings” (the second item in the list on the left side of the dialog box) and make sure that “Check for new messages at startup” and “Check for new messages every 10 minutes” are not selected. Click “OK”.
- If prompted, select Done. (Most of you won’t need to do this.) A warning dialog box may 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. If Thunderbird complains, check the box I understand the risks, then select Done.

SEND YOURSELF SOME MAIL:

Using your Williams Gmail account (either through http://email.williams.edu or http://www.gmail.com) send a message to your kerry.cs.williams.edu email address with a “Subject” line that includes the fact that the message was sent using Gmail and at least two lines of text as content.

Using TCPCapture

- You should start working in pairs now. Choose either of your two accounts for the next few steps. It does not matter which account you choose now. You will switch to the other one later.
- Within the finder window displaying the contents of the Applications folder, find the sub-folder named Utilities. Double-click on the icon for Utilities. The Utilities folder will open.
- Open TCPCapture by double-clicking its icon within the Utilities folder. A window like the one on the right should appear. (If you get a warning about it being damaged, just try opening it again.)
• 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 the program 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 end with kerry.cs.williams.edu. As you do this, Thunderbird may request access to your contacts. Decline this request.

• Enter text in the Subject field that mentions that the message was sent using Thunderbird.

• Add any short message you would like but stick to simple text (i.e., don’t change the font or make anything bold). Then click on the 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 on the right. The program’s window is now filled with several lines of text. Each of these lines corresponds to a packet sent between the Thunderbird program and our email server. The lines that start with kerry.cs.williams.edu -> CLIENT correspond to packets that traveled 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 [137.165.19.90]
```

which might translate into English roughly as “Hello, I’m 137.165.19.90.” (This happens to be the network address of the machine used while preparing this handout. Yours will be slightly different.)

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 Enter mail, end with "." on a line by itself
```
If you click on the line immediately below this packet, the program’s window should display something like the image shown on the right. Some of the text near the bottom 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 contains the text of your email message. It should be clear that Thunderbird sent a lot more than what you actually typed.

Email Headers
The contents of the complete packet probably looks something like the text below:

The message you sent appears as a few lines at the bottom of the text. It is preceded by a collection of lines created by Thunderbird that describe various properties of the message. The purpose of many of these lines is obvious. The lines that start with “To:” and “From:” identify the sender and the recipient. A line that starts with “Subject:” includes the subject you typed when you sent the message. The “Date:” line indicates when the message was sent. The remaining lines are more mysterious. The last three lines describe how the message is encoded relative to an Internet standard called “MIME” (Multipurpose Internet Mail Extensions) which we will discuss in more detail later. The “User-Agent:” line describes the software used to send the message (Thunderbird on a Mac). The “Message-ID:” line provides a serial number for the message.
All of these “lines” are called mail headers. They follow a common format. Each header begins with an identifying string (“To”, “Date”, etc.) followed by a colon. This is followed by details associated with the identifying string. If the contents of the header exceed one line, each of the subsequent lines must be indented. The first completely blank line in a mail message separates the headers from the body of the message. As we will see later, even if a line that looks like a header appears after this blank line, it will be treated as part of the message body.

For those who want to know more, http://www.owlriver.com/spam/stop-spam.html provides a (slightly dated but) reasonable discussion of mail headers.

SMTP Basics
The email message’s body and all of the headers that accompany it appear in just one of many packets intercepted by TCPCapture. Now that we have discussed the details of this packet, 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 one containing 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. This is a variation of HELO which is short for “Hello”.
- **RCPT** - provides the destination address to which the message should be delivered. The command name must be followed by “TO: <user@somemachinename>” where user@somemachinename is the email destination address for the message. This is what really determines the recipients of the message (rather than the “To:” header in the message packet).
- **MAIL** - provides the return address for the message’s sender. The command name must be followed by “FROM: <sendername@somemachinename>” where sendername@somemachine-name should be the email address of the sender.
- **DATA** - indicates that the message itself will follow immediately.
- **QUIT** - Terminates the connection.

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 care 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 above should appear. Near the top of the window there is a text field labeled Server name. You should type the name of our server, kerry.cs.williams.edu, into this field.

A single machine might be running several server programs at the same time. For example, kerry.cs.williams.edu is constantly running a POP server program and an SMTP server program. When this machine receives a packet it needs a way to decide which of these two servers 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.

NetTap is now ready for you to connect to and interact with our mail server. There is one catch. The server can be very impatient. If you take too long to complete any step in the process required to send a message through the server, the server may disconnect you. If this happens, you will need to connect again and start over a little faster. To avoid this, it is a good idea to first read through the steps described in the next section of the lab (everything up to the “Experimenting with POP” header) without performing any of them. Then, once you have the big picture, go through the section again actually performing the steps described.

**Sending an Email with NetTap**

Arrange the TCPCapture and NetTap windows on your screen so that you can see them both easily.

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 bottom of your TCPCapture window where you intercepted 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 is sent from the SMTP server to the NetTap program as soon as you press Connect. 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.

After receiving the DATA command, the server expects you to send the email message you want to send. As discussed above, the Thunderbird email client (and most real email clients) includes many header lines in addition to the text of the message. You don’t need to mimic all this, but just to better understand how headers are used, we want you enter two header lines.

First enter a line of the form

Subject: a message sent using NetTap

and press “Send” to deliver it to the server. Next, send the server a blank line by clearing the contents of the field next to the “Send” button and then pressing the button. Finally, enter and send a line that contains “From: ” followed by your kerry.cs.williams.edu email address.

If you were paying attention while reading our explanation of headers above, you will realize that we just had you deliberately send a “From:” header incorrectly. All headers should precede the first blank line in the message but we told you to send a blank line before the “From:” header. You will see the impact of this “mistake” later when you use Thunderbird to examine the messages sent to your account.

Finally, enter some creative message line-by-line pressing Send after each line has been typed. When you are done, enter and send a line containing just a single period to inform the server that the message has been completed.

Now, send the server a “QUIT” command to terminate your conversation.

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.

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. Just as it intercepted and displayed the packets Thunderbird sent to and received from the server, it now shows everything you sent or received 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 space and then your user name (such as 20jrl2).

PASS - The second packet sent to the server should be composed of the word “PASS” followed by a space and then 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. Deleting requests don’t actually take effect until (and unless) you later send a QUIT command.

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. Enter the server name kerry.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, using different “words” (220 vs. +OK) for a single concept.

Once you are connected, send “USER” and “PASS” commands as described above 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. We will discuss more of this control information shortly. Finally, send a “QUIT” command to the server and also quit NetTap.

**How Thunderbird Uses POP**

As the next 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 *Stop, Clear and Start*. Next, switch to Thunderbird and press the *inbox* icon (Get Message) 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 as well as the messages you sent using Gmail and the Thunderbird program. In the list of messages shown in the Thunderbird window the message sent using NetTap should stand out in one way. For each of the messages other than the one sent with NetTap, Thunderbird should display both the sender’s account or name and the message’s subject. For the message sent using NetTap, only the subject should appear. This is because we had you deliberately break the email message format rules when sending this message. We told you to send a blank line between the “Subject:” header and the “From:” header. As soon as Thunderbird sees this blank line, it assumes everything else is part of the message body rather than part of the collection of headers. If you click once on the summary line for the NetTap message in the Thunderbird window, you will see that the “From:” line you typed appears as the first line of the message body.

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

If you examine these packets starting at the bottom, things should look familiar. Thunderbird uses the RETR command repeatedly to retrieve all of the messages sent to your kerry account so that it has them on hand to display as soon as you click on any of the message summary lines in the Thunderbird window. Once it has them all, it send a QUIT command.

However, if you start at the beginning of TCPCapture’s summary of the dialog between kerry.cs.williams.edu and the Thunderbird program, things look quite different from what you did to communicate with the mail server using NetTap. While you started by sending USER and PASS commands, Thunderbird starts by sending AUTH and CAPA commands. These commands ask the server to tell Thunderbird about the “AUTHenticaion” methods the server accepts and about other “CAPAbilities” it supports. The server responds with lists of terms that identify its features. This is necessary since the POP protocol allows different servers to provide different options to mail program.

Given the server’s responses to the initial AUTH and CAPA commands, Thunderbird sends an “AUTH PLAIN” command to the server indicating that it wants to use the “plaintext” authentication method. Oddly, instead of sending your account id and password in plaintext, it sends a string that probably looks something like “AHRlc3RzMTcAc3ByaW5nMjAxNw==” a few lines later. This is an encoding of your id and password using a technique called base 64 encoding. The server then confirms that the login is accepted.
Next, before sending any RETR commands, Thunderbird sends the three commands STAT, LIST and UIDL. We already discussed STAT above. It asks the server to respond with a count of the number of messages stored on the account. LIST and UIDL ask for information about the individual messages stored. LIST sends back the size of each message. UIDL sends back a serial number that will remain unchanged as long as the message is stored on the server even if other messages are added or deleted.

To appreciate why Thunderbird sends these commands first clear your TCPCapture window (but leave the program actively capturing packets) and then quit and restart Thunderbird. Thunderbird will contact the server again when it restarts to see if any new messages arrived. It may ask you to re-enter your password to make this possible.

Now, look at the TCPCapture window. The dialog between the server and Thunderbird starts out with AUTH and CAPA commands as it did before, but something is missing. There are no RETR commands this time. Before you quit Thunderbird, it stored a copy of the email messages it retrieved. When you restart the program, it does no have to retrieve those messages again. It does, however, need to check that no new messages have arrived. The LIST and UIDL commands enable it to check for new messages.

Send a new message to yourself using Thunderbird. As you have done in the past, mention the fact that it was send using Thunderbird in the message’s subject. To make things a bit more interesting, make one of the words in the message body bold or italics.

Now, clear the TCPCapture window and then tell Thunderbird to check for new mail by clicking on its “Get Messages” button. Examine the packets shown in the TCPCapture window. How many RETR commands are issued?

Next, what commands do you think Thunderbird sends to the POP server when you tell it to delete a message? Think about it for a moment. Then, to find out, first make sure that you can see both the TCPCapture window and the Thunderbird window on your screen. Select the summary line in the Thunderbird window for the message you sent using NetTap so that its contents are displayed. While keeping an eye on the TCPCapture window, delete the message by pressing the delete key or selecting “Delete Message” from the Edit menu. Are you surprised by what you see in the TCPCapture window?

To explore what just happened (or didn’t happen) a bit more thoroughly, quit Thunderbird for a moment. Go back to NetTap (restart it if necessary). Connect to the POP server on kerry, log in by issuing USER and PASS commands. Send a STAT command to see how many messages you have. Use the RETR command to retrieve the next to last message. This should be the message you sent using NetTap. It is still on the server even though you deleted it using Thunderbird!

What you are seeing here is that Thunderbird (and most other similar mail programs) keeps its own copies of the messages from your mail account. At some points, Thunderbird’s version of what is in your mail account may be different from what is actually on the server.

To emphasize this, let’s make it worse.

Using NetTap, send a DELE command to delete the first message in your mail account (this should be the message you sent using Gmail). Then send a QUIT command using NetTap to terminate your POP connection to kerry.

Now, again keeping an eye on the TCPCapture window, restart Thunderbird, re-entering your password if asked to do so. Two interesting things should be observed. First, as expected, Thunderbird will again connect to kerry to check for new mail. While doing so, however, it also sends a DELE command. What message is it deleting?
Second, if you scan through the messages displayed in the Thunderbird message list, you should still see the message you sent using Gmail, even though you just deleted it using NetTap. Thunderbird is aware that the message from Gmail has somehow been removed from your account on kerry, but since it has made its own copy of the message it will continue to display it until you delete it using Thunderbird’s interface. (But don’t delete it from Thunderbird now! We will use it a bit later in this tutorial.)

An Introduction to MIME

When email was first introduced, computers had much simpler displays that did not provide the ability to display text in multiple fonts or with symbols displayed in bold or italics. Each symbol to be displayed could be described by simply providing its value in the ASCII code. As a result, the initial standards for email limited the contents of a message to be simple ASCII text.

Today, email messages can include text containing multiple fonts, symbols that are in italics or underlined and, in addition to this text, attachments including images, audio file, movies, etc. Since the original email standards only allowed simple text, a standard had to be developed that could be used to encode all the fancy features required by modern email. To ensure backward compatibility, this new standard had to be designed to use only the symbols that were provided by the ASCII encoding. This standard is called MIME which is short for Multipurpose Internet Mail Extensions. As our final step in exploring the email system, we want you to observe how MIME provides the ability to include several sub-parts of different types in a single message.

During this lab, we have had you send two messages to the account you are accessing now using Thunderbird. One contained just simple text. For the other, we instructed you to make at least one word bold or italics. Select the plain text message summary line in the Thunderbird window so that Thunderbird displays the contents of this message. Then, select “View Source” from the “More” button that appears near the right edge of the Thunderbird window between the message summaries and the contents of the selected message. A new window containing the full text of the email message should appear. The new window shows much more than was displayed in the bottom section of the Thunderbird window. All of the header lines that were included with the message are displayed in the “Source” window. One of the last of the header lines starts with “Content-Type:” and includes the specification “text/plain”. This header informs Thunderbird that the content of this particular email is simple text. That is, nothing but ASCII. These contents are displayed below the last header.

Now, let’s look at the message containing bold or italic text in the same way. Select the summary for this message so that it is displayed at the bottom of the Thunderbird window. Choose “View Source” from the menu that appears when you press on the “More” button.

This time, the message’s encoding is a bit more complicated. In the simple message, there was only one “Content-Type:” header. In this message there may appear to be three such headers. Start with the first such header line. Note that it appears just a few lines before a blank line and recall that such a blank line indicates the end of the message’s headers. Therefore, this first “Content-Type:” header is the only one that is really a header.

Rather than containing the specification “text/plain”, this “Content-Type:” header says “multipart/alternative” and is followed by a line of the form “boundary=...”. This tells Thunderbird that the message body has several subparts which are separated from one another by lines contains the odd sequence of characters specified in the “boundary=...” line.

The reason this message body has multiple parts is that when faced with a message containing text that is not simple (because it involves font changes or bolded words), Thunderbird tries to have it both ways. To be backward compatible with any old email program that has to process the message, it wants to send the message as plain text. To give newer email software a chance to display the message in all its glory, it wants to use a richer system called HTML to encode the message. HTML is also the language used to encode web pages. So, it sends both a plain text and an HTML version.
If you scan the rest of the message source you will see that right after every line that matches the “boundary=…” specification, there are what look like header lines describing the content type. The first says “text/plain” and is followed by a version of the message you sent where only plain text character are used. The second includes the specification “text/html” indicating that it will be followed by an encoding of the message using the HTML markup language to describe the special formatting desired. If you look carefully, you should find all the words you included in your message surrounded with lots of < and > signs and text that describes (at least to the computer) the desired way to display the text.

Most message sent using modern email software will employ MIME to encode your message contents in multiple formats. You might want to take a look at the source of the message you sent using your Gmail account. How many MIME subparts does this message contain? How many of its headers can you explain?

This concludes our guided tour of email protocols and standards. Obviously, there is much more to explore than we have covered. Feel free to experiment more at the end of lab if you have extra time. For now, however, you should move on to the second component of today’s lab by quitting TCPCapture, Net-Tap and Thunderbird.
Part II: Java Programming with BlueJ

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

![Authenticate 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.

The rest of this lab should be completely individually with each student using their own account.

Checking Out Your Code And Using BlueJ

Begin by launching the BlueJ application. It can be found in the Applications folder. Once BlueJ is running:

- If prompted, say “No Thanks” to the dialog box inviting you to participate in BlueJ data gathering.
- Select Tools -> Team -> Checkout Project from the BlueJ menu. A window as shown on the right should appear.
- Make sure to select the tab labeled “Git” at the top of the window.
- For Repository URI, enter text similar to what is shown in the image to the right with the account information near the end, “22ew1”, replaced with your own CS account id. For example you would type https://evolene.cs.williams.edu/cs134-f19/lab1/23ab7.git if your id was 23ab7.
- Enter your name and your @williams.edu email address.
- Finally, enter your CS username in the User field and your CS password in the Password field. Check everything and then click “OK”. If you receive any kind of error message, carefully double-check everything you typed and then try “OK” again.
- The next window (also shown on the right) asks you to specify a location on your local disk to save your files. We recommend making a folder for cs134 in your Documents folder, and then storing your labs there. Name your lab “lab1.”
- BlueJ should now display a project window with “lab1 (Shared Project)” 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
 * *
 * /
```

form what is called a comment. The computer ignores all the text sandwiched between an opening /* matched with a closing */ 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. **Do not enter your name.** We try to grade your work anonymously and do not want your code to include any identifying information.

**GUIManager: A Framework for a New Program**

The next line

```java
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 the class header 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

```java
private final int WINDOW_WIDTH = 400, WINDOW_HEIGHT = 400;
```

declares two names, WINDOW_WIDTH and WINDOW_HEIGHT, and associates them with 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. 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

```java
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

```java
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 BlueJ 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 very first Java program. Congratulations!
- Click on the empty window. Then press the mouse while pointing at the “BlueJ Virtual Machine” item in the menu bar (at the top of the screen) 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 this “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 \texttt{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 specify 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 to add things to your currently empty window. The areas in which a user can enter a user name and/or password in the completed program are called \texttt{JTextFields} in Java. The simple text labels that appear in the window to identify these fields are called \texttt{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

\begin{verbatim}
    contentPane.add( new JLabel( "Username:" ) );
    contentPane.add( new JTextField( 8 ) );
\end{verbatim}

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 “this.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 \texttt{Username}. 

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

\begin{verbatim}
    contentPane.add( ... );
\end{verbatim}

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

\begin{verbatim}
    new JButton( "Authenticate" )
\end{verbatim}

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

\textbf{Methods}

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

\begin{verbatim}
    public void buttonClicked( ) {
......
    }
\end{verbatim}
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 button-Clicked header are executed. Likewise for textEntered and menuItemSelected.

Right now, since there are no instructions 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 an instruction to add a new JLabel displaying “Login Rejected” to the contentPane. This instruction should be placed 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, add 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 https://www.madtakes.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:
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:

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

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”.

In the editor window for the new class, change the comments included at the top of the template to include a short description of the program. As before, do not include your name. 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:

and when it is run it should produce a result like:
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 how to build a program whose interface looks like:

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.

**Using Variables**

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 other than the first.)

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.

The declaration you just entered tells Java that you plan to use the name `noun1` to refer to a text field, but it does not create a text field or associate the name with any particular text field. 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" ) );
cContentPane.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 an operation 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 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 per-
fect, 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 win-
dow, you will probably find that your interface looks pretty messy, particularly after you press the COM-
POSE 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 declar-
ing 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 con-
tentPane. 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 appropri-
ate declarations for FINAL_WINDOW_WIDTH and FINAL_WINDOW_HEIGHT in this case.

**Submission Instructions**

Once your programs work correctly or you are finished working for this session, you should save your
code to our server. This process involves committing and pushing your code. To do this, you should:

- Start by finding the "Commit/Push..." button in BlueJ’s window. If it is not visible, click the arrow icon
  at the bottom of the space below “Compile” to display the Teamwork buttons.
- Click the "Commit/Push..." button.
- A window will open showing you the file(s) that have been modified. Type a brief comment that de-
scribes your edits in the "Commit comment" box, and click Commit.
• After writing a comment and committing, the bottom part of the dialog box will become active. Just click the “Push” button. You may need to enter your password again (in a window that might be obscured by the dialog box). At this point all of your code has been saved. You may click Close to exit the window.

• **Note:** If you want to continue working on your code at later time from a different computer, you will need to **checkout** your code again (as you did back on page 14). Similarly, if you make changes on a different computer and then open an old version, you should **update** (by clicking the Update button under Teamwork) to the latest version before making new edits. We will go over this process in more detail in Lab 2 next week. If you are curious, for more instructions on retrieving/saving your code from/to our server using BlueJ and git, check out [http://www.cs.williams.edu/~cs134/submission2.html](http://www.cs.williams.edu/~cs134/submission2.html).

**Finishing up**

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

• Open the Terminal from the Utilities folder (in the Applications folder).

• Click in the window that opens. To the right of the text that appears, type `kpasswd`.

• When prompted, type your current password and press enter. Don’t be alarmed if nothing shows up as you type your password. That is normal!

• When prompted, type your new password and press enter (twice).

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.