Lab 4
Jabbering AIMlessly

Google Chat Client Implementation Project --- Part 1
Due: Wed. 3/2 at 11PM (for Mon. aft. lab), Thurs. 3/3 at 5PM (for Mon. evening), or Thurs. 3/3 at 11 (for Tues. aft.)

Over the next two weeks of lab, you will construct your own Google chat client. The goals for Part 1 are:
• to explore the XMPP chat protocol that has been used by Google, Facebook and many other servers, and
• to implement a program that uses conditional statements, loops, and string operations.

This week’s program will have several limitations. For example, all chat messages you exchange with others will be displayed in one window. Next week, you will add a few extensions to produce a fully functional chat program.

Pre-lab Preparation
You should be getting pretty good at constructing buttons, text fields and labels by now. We would like to take advantage of this by having you type in the code that creates the GUI interface for this week’s program before your lab period. That is, before you come to lab, we would like you to have entered all of the code for (only!) your program’s constructor and all of the instance variable declarations that go with this code. We also would like you to create two Google accounts (beyond any you might already have) for yourself.

We are asking you to do this so that you can make more efficient use of our support during the lab period. If you have completed the “easy” parts of your program before you come to lab, you will have the full lab period to ask for assistance with the more challenging parts. At the same time, if you do need assistance with the construction of the interface, we still want to help you. So we are not requiring that your code for constructing the interface be complete before you start lab. We are only asking that you do your best to write the code and type it in to BlueJ before lab.

As you develop your chat client, you will need to test its code by exchanging messages with another client. It will be possible (and even enjoyable!) to do this by exchanging messages with other students in the lab or with friends who are nowhere near our lab. At first, however, it will be simpler if you can exchange messages with yourself by running a standard chat client on your machine at the same time you are testing your own program. We recommend that you use a webpage logged into Gmail as the standard chat client for this lab.

To do this, you will have to log in with different account names on your lab chat program and on the other client. To enable you to do this, please register for 2 new Gmail accounts before coming to lab. We recommend that you create new accounts rather than using existing Gmail accounts you may have, because we will have to configure these account specially to ensure that they interact with your client program correctly.

Creating Google Accounts
Go to www.google.com/accounts1. If you find yourself logged in to an existing Google account, sign off from it. Follow the steps needed to create a new account. When the account has been created, select Gmail to go to the Gmail page of your new account. On the left side of the page you will see a pull-down menu with your new user name. From this menu, select “Revert to old chat” (this will disable the Hangouts feature). When you have carried out these steps, sign off from your newly created account and repeat the steps above to create a second account.

Once you have created both accounts, use the gmail.com web interface to have one of your accounts issue an invitation to chat to the other (https://support.google.com/chat/answer/159497?hl=en) and then to accept this invitation. This will ensure that the two accounts belong to each other’s rosters of potential chat partners. Make sure that this is the case by chatting with your (other) self using two web pages each logged into one of your Gmail accounts. Note that you can have both of your new Gmail accounts open simultaneously in separate windows by clicking on your name in the upper right-hand corner of the Gmail page of one of your accounts and selecting “add account”.

Important Note: To fully test some features of your program, you will want your account to have more than a single friend. To accomplish this, once you are in lab you should use the Gmail web interface to exchange chat messages with at least two of your class/lab mates so that their accounts are added to your account’s list of friends.

1 If you are reading this document on a computer, just click the link that this is footnoting!
Client Interface
A sample of what the program’s interface might look like is shown on the right.

Across the top of the program’s window, there are fields used to enter the user’s account name and password and buttons used to log in and out. Obviously, the user is supposed to enter an account and password before pressing the login button. Also, until login is completed successfully, the buttons for logging out and sending messages should be disabled. They should become enabled once the user logs in and then be disabled again upon logout.

Immediately below these items, there is a text area that occupies most of the program’s window. This area is used to display the messages sent between the program’s user and his or her friends. When the program’s user sends a message, the program should display the user’s account name together with the message in this large text area. Similarly, when a message is received, the sender’s account and the message should be displayed.

Next, there is a pop-up menu labeled “Friends”. The menu will include the names of the user’s friends (as registered with Google). Friends who are logged in will appear first in this menu followed by friends who are currently not logged in. The two groups should be separated by a menu item of the form “--------- offline ---------”.

Below the menu is a text field where the user can enter a message to send. Once a message has been entered and a friend selected from the Friends menu, the user can send the message by clicking on the “Send” button. Note that messages can be sent to friends whether they are online or offline. If a message is sent to someone who is offline, the XMPP server delivers it when that user logs back in.

Jabbering Away
The Google chat system and many other chat facilities (include Facebook’s) use a protocol named XMPP (eXtensible Messaging and Presence Protocol). XMPP is a descendant of an earlier protocol known as Jabber. Jabber was developed as part of an open-source project whose goal was to produce an open alternative to the proprietary AIM system provided by AOL. The first Jabber server was released by Jeremie Miller in January of 1999. The protocol gained popularity quickly. Equally quickly, its designers began working to seek recognition of the protocol as a standard from the body that approves Internet protocols, the Internet Engineering Task Force (IETF). The IETF issued standard documents for the protocol in 2004. In the process, its name changed from Jabber to XMPP.

The client you will implement for this lab is based on XMPP. With a little luck, the brief introduction provided below will provide all the information you need to complete this lab. If you want to know more, however, the complete standard documents can be found at http://xmpp.org/xmpp-protocols/rfc/

Like POP and SMTP, XMPP is a text-based protocol. All it requires is sending packets full of text between the client and server. Unlike SMTP and POP, the messages sent between XMPP clients and servers are highly structured. They are composed using a language known as XML (eXtensible Markup Language). XML will seem familiar to those of you who know HTML, the language used to describe web pages.

An XML document is composed of content and markup. Text that should be interpreted as markup is surrounded by matching pairs of less than (“<”) and greater than (“>”) symbols (also called angle brackets). Each piece of markup is referred to as a tag. Anything that is not markup is content.

For example, to send the message “Hi.” shown in the first line of text displayed in the sample interface above, a chat client program might send the text:
to an XMPP server. This text includes four examples of markup tags:

```xml
<message to="pinbawl@gmail.com" from="thommurtagh@gmail.com/wk" type="chat"><body>Hi.</body></message>
```

and

```xml
<message to="pinbawl@gmail.com" from="thommurtagh@gmail.com/wk" type="chat">
<body>
</body>
</message>
```

and one piece of content:

```
Hi.
```

The first word that appears after the “<” that opens a markup tag is the type of the tag. Thus, the first two tags in our example would be described as a message tag and a body tag. In Java, you have seen that punctuation marks often come in matching pairs. The “{” goes with the “}” and each “(” has a matching “)”. In XML, tags often form matching pairs. In general, a tag whose name starts with a “/” matches a tag whose name is formed by removing the “/”. Thus, a /body tag matches a body tag and a /message tag matches a message tag. A pair of matching tags together with all of the enclosed tags and content is known as an **element**.

Some markup tags are intended to stand by themselves. They have no matching “/” tag. To indicate this in XML, we place a “/” before the “>” that ends such a tag. For example, when a user first logs in through a client program, the server will send the client a series of item tags of the form

```xml
<item jid="pinbawl@gmail.com" name="Tommy Walker" subscription="both"/>
```

describing all of the friends associated with the user’s account. The “/” near the end tells us that there will be no matching </item> tag.

In the simplest case, the only text between the “<” and “>” is the tag’s name. The “<body>” tag shown above is an example of this. In many other cases, however, pairs of names and values separated by equal signs will be included in a tag. These are called **attribute specifications**. For example, the item tag shown above specifies values for the attributes jid, name, and subscription. The XMPP standard describes which attributes may or must be specified in each type of tag and how they should be interpreted. For example, the jid attribute of an item tag should be the account of one of the user’s friends, the name attribute provides a familiar name for the other user, and the subscription attribute indicates whether or not the “friend” status between the two accounts is mutual (“both”) or not. The values associated with attribute names should be surrounded by quotes as shown in the example. The order in which attribute specifications appear within a tag is not fixed by the protocol. Thus, the tag shown above would be considered equivalent to the tag

```xml
<item name="Tommy Walker" subscription="both" jid="pinbawl@gmail.com"/>
```

A correctly implemented client should interpret these tags identically.

When the XMPP protocol is used, certain XML elements function as complete commands while others only appear as subparts of larger elements. For example, the body tags shown in the example above can only be sent to the server as the contents of a larger message element. Similarly, the item tag shown above must be enclosed as part of a larger “iq” (information query) element. XMPP uses the term **stanza** for complete tags that can be sent to the server independently.

The program you construct for this lab will send or receive three types of XML stanzas. As you may have already guessed, **message** stanzas will be used to send messages to other users and to deliver messages from other users to your client. **Presence** stanzas will be used to deliver information to your client about which of its user’s friends are online and offline. Finally the server will provide a list of item tags of the form shown above to your client as part of an **iq** stanza soon after a user first logs in. We will discuss the relevant details of each form of XML stanza below.

The most important thing to know about the stanzas the server will send to your client is that you will be able to ignore most of the information they include. XML is a protocol designed to support many features that will not be included in your client. If you look at the text of the stanzas sent to your client you will see information included in the XML designed to support video chat, provide detailed information about each friend’s status, and manage chat.
rooms. Fortunately, it is possible to implement a simple client that ignores this extra information. For each type of stanza, we will describe the relevant tags and attributes. You can safely ignore everything else.

**Jabber IDs**

The attribute specification:

```
jid="pinbawl@gmail.com"
```

shown in the sample item tag above is an interesting anachronism. The letters jid stand for Jabber identifier. Even though the IETF changed the name of the chat protocol from Jabber to XMPP when it was standardized, they could not change this attribute name from jid to xid or XMPPid without making the new protocol incompatible with existing server and client software. So, they let the name Jabber live on in this feature of the protocol. The strings that servers and clients use as addresses for messages in the XMPP protocol are still called JabberIDs.

A basic JabberID looks exactly like an email address. The user’s account name is followed by an at sign and what looks like a server name. In fact, the component after the at sign is an XMPP domain name rather than a server name. In particular, the Google server name is different from the domain name. The Google XMPP server is named talk.google.com while the domain name included in all Google JabberIDs is gmail.com.

A user can log in to an XMPP server from different computers at the same time. To give each login a unique address, each login is assigned a unique *resource name* that becomes part of what is called that login’s full JabberID. The resource name follows the basic JabberID and is separated from the domain name by a slash. For example, if the account thommurtagh logged in from home and work, one login might be assigned the full JabberID thommurtagh@gmail.com/work and the other might be assigned thommurtagh@gmail.com/home.

**Communicating with Google’s XMPP Server**

To communicate with Google’s XMPP server you will use a class named `GTalkConnection` in place of the `NetConnection` class you have used for your POP clients. The construction of a `GTalkConnection` will not only establish a connection to Google’s XMPP server, talk.google.com, it will also complete the exchange of all of the XML stanzas required to log in to the server. As a result, instead of providing a hostname and port number as you did when creating a `NetConnection`, when you create a `GTalkConnection` you will provide a user’s Gmail account and password. For example, you could create a connection by executing

```
private GTalkConnection toGoogle;
...

toGoogle = new GTalkConnection( “thommurtagh@gmail.com”, “somethingsecret” );
```

The server will assign a resource name to the new login. This resource name becomes part of the full JabberID that identifies the new login. If the login performed when you create a new `GTalkConnection` is successful, you can retrieve the full JabberID using the method `fullJid`. For example, after executing the construction shown above you might execute

```
String fullId;
fullId = toGoogle.fullJid();
```

If the login was unsuccessful, `fullJid` will return the empty string. Therefore, you can use `fullJid` to determine if the login was accepted or rejected.

Once you have created a `GTalkConnection` and verified that the login process was successful, there are three methods you will use to exchange messages with the server. All three are very similar to methods associated with the `NetConnection` class.

You can use an invocation of the form

```
toGoogle.out.println( someString );
```

to send text to the server. You can either send an entire stanza with one such invocation or send pieces of a stanza using several `printlns`. For example, you could send the message tag used as an example above:

```
<message to="pinbawl@gmail.com" from="thommurtagh@gmail.com/wk" type="chat"><body>Hi.</body></message>
```
by executing the instructions:

toGoogle.out.println("<message to="pinbawl@gmail.com" from="thommurtagh@gmail.com/wk" type="chat">");
toGoogle.out.println("<body>Hi.</body>");
toGoogle.out.println("</message>");

Note that any quotes that are to be sent as part of a message to the server must be preceded by backslashes when included in a quoted String literal in a Java program.

You can use an invocation of the form

    toGoogle.in.nextStanza()

to retrieve a complete stanza of XML sent from the server to your client. Typically, you will assign the result of such an invocation to a variable as in:

    String stanza;
    stanza = toGoogle.in.nextStanza();

Finally, an invocation of the form:

    toGoogle.addMessageListener( this );

can be used to tell Java that you want the code you place in your dataAvailable method to be executed whenever a stanza arrives from the server and the method connectionClosed is to be executed if the server terminates the connection. When you use addMessageListener in this lab, the versions of dataAvailable and connectionClosed you define must expect no parameters. That is, you must say

    public void dataAvailable() {
    

rather than using a method header of the form “public void dataAvailable( NetConnection connection )” or “public void dataAvailable( GTalkConnection connection )”.

One important difference between a GTalkConnection and a NetConnection is that all the data sent through a GTalkConnection is encrypted. This makes it impossible to use a program like TCPCapture as a debugging tool when working with a GTalkConnection. Anything TCPCapture might display would be indecipherable.

To provide you with debugging help, the GTalkConnection class includes a method named debugEnabled. Executing a command of the form

    toGoogle.debugEnabled( true );

after you create a GTalkConnection will cause the connection to display all XML sent through the connection in the upper half of the BlueJ terminal window (the same window that the ugly error messages appear in). Like the error messages, this debugging information will not disappear automatically between runs of your program. This can be a source of confusion. Executing a statement of the form

    System.out.print( "\f" );

in your program will clear the upper half of the window (the “f” in “\f” stands for “form feed”). Therefore, the code you use to create your GTalkConnection while you are debugging should probably look like:

    System.out.print( "\f" );
    toGoogle = new GTalkConnection( ... );
    toGoogle.debugEnabled( true );

Although BlueJ does not automatically clear its terminal window, it may remove older lines displayed in the window if it becomes too full. To avoid this, make sure that the “Unlimited buffering” item is selected in the “Option” menu that appears after you click on the terminal window.
Sending Chat Messages
To send a message, your program will embed the text of the message within a pair of matching body and /body tags and then embed this element within an enclosing pair of message and /message tags. The message tag should include attribute specifications for the to and from addresses. The to address can be a partial JabberID. The from address must be the full JabberID assigned to your client during login (and accessible using the fullJid method). A type attribute with the value “chat” should also be included.

The Java code to actually produce the correct String to encode a message stanza can get quite ugly. It can help to create the desired String and assign it to a variable before using a println to send it. It can also help if you break that sequence of concatenations that construct the String into separate lines corresponding to the components of the XML tag. For example, the code to send the opening line of a message stanza might look like:

```java
String messageTag = "<message to=" + friend + "" + " from=" + fullJid + "" + " type="chat">
    toGoogle.out.println( messageTag );
```

Receiving Chat Messages
When another user sends a chat message to the user running your program, the server will deliver it as an XMPP stanza very similar to the stanza the other client used to send the message. The stanza you receive will consist of a pair of message and /message tags. Within these tags there will be a pair of body and /body tags surrounding the text of the message. The from attribute of the opening message tag will tell you who sent the message. Simple! Right?

Alas, there are a few complications.

First, the message stanza your client receives is likely to contain many tags and attributes in addition to the ones described above. The text below, for example, was a message stanza received from Google's chat server. We have tried to indent the tags to make it as readable as possible.

```xml
<message to="thommurtagh@gmail.com/TCLE6F71892" from="pinbawl@gmail.com/gmail.2CD97743" type="chat">
    <body>Hello</body>
    <met:google-mail-signature xmlns:met="google:metadata">
        zXyFdCeF0h2WRcAW-VvPzApK00A
    </met:google-mail-signature>
    <cha:active xmlns:cha="http://jabber.org/protocol/chatstates"/>
    <nos:x value="disabled" xmlns:nos="google:nosave"/>
    <arc:record otr="false" xmlns:arc="http://jabber.org/protocol/archive"/>
</message>
```

It should be clear that this stanza contains many items we haven’t mentioned. The good news is that you can simply ignore them. You can use the String indexOf method to find the key components (the body tag, the from attribute, etc.) and process these components alone. When you write the code to do this, however, remember to base your code on the rules of XML and XMPP, not on the specific example above.

For example, in the example shown above, the "type" attribute follows the “from” attribute. You might be tempted to write code that determines where the “from” id ends by searching for the word “type” and then backing up two spaces. The rules of XML, however, allow the server to send these attributes in any order it wants. Therefore, the correct way to extract the “from” id is to search for "from=" to discover where the from id starts and to then search for the closing quote (the first "" found after "from=") to determine where the id ends.

Second, the server can include either a partial or full JabberID as the from attribute of each message tag. You only want to display the partial id (i.e., the account name and “gmail.com”) in your program’s text area. You can do this by using indexOf to find the "" that ends the id and to see if it contains any "/". Then you can use substring to extract the partial id.

Third, the server will often send you message stanzas that do not contain a body element. These stanzas are used to provide information about when another user starts sending you a message. You should ignore such stanzas.

Finally, imagine that you are using XMPP to have a chat with someone and you are discussing how XMPP works. You might want to send a message about the </body> tag. For example, you might want to say “What comes after
the </body> tag?” Based on what we have said earlier, the XML stanza you would expect to send for this message would look like:

```xml
<message from="thommurtagh@gmail.com/TCLE6F71892" type="chat" to="pinbawl@gmail.com">
  <body> What comes after the </body> tag? </body>
</message>
```

Unfortunately, this stanza would be unacceptable because each body tag should have exactly one matching /body tag and this message contains more /body tags than body tags.

To avoid such problems, the rules of XML call for using special sequences of characters to represent any “<” or “>” characters that might appear in the content of a stanza. In place of “<”, XML uses the sequence “&lt;”. In place of “>” it uses the sequence “&gt;”. Therefore, to send the message “What comes after the </body> tag.” a client should transmit a stanza of the form:

```xml
<message from="thommurtagh@gmail.com/TCLE6F71892" type="chat" to="pinbawl@gmail.com">
  <body> What comes after the &lt;/body&gt; tag. </body>
</message>
```

To make things worse, XML also has to include a special mechanism to encode ampersands. If not, if you sent a message like “The sequence &lt; represents a less-than sign.” it would be interpreted as “The sequence < represents a less-than sign.” XML therefore uses “&amp;” to encode any ampersands that occur in the content of a message. To send the message “The sequence &lt; represents a less than sign.” you would therefore include the text

```
The sequence &amp;lt; represents a less than sign.
```

in an appropriate message element.

As a result, your code to transmit messages will eventually have to include a loop to replace any “<”, “>”, or “&” characters that appear in a message with the appropriate &-sequence. In addition, your code to handle incoming messages will have to find any of the special codes that begin with ampersands and convert them into the appropriate characters before displaying the incoming message in the program’s text area.

For this week, we will only ask you to perform the conversion of “&amp;” to “&” in incoming messages. This will give you a chance to practice some String processing so that next lab you will be ready to extend your code to handle less than and greater than symbols as well and to handle them in both incoming and outgoing messages.

**Good Buddies**

For each account it supports, an XMPP server maintains a roster of all of the other accounts with which the owner has agreed to exchange messages. After your program creates a new GTalkConnection, the server will send it an iq stanza containing one item tag for each friend in the user’s roster. An example of such a stanza is shown below. We have indented tags to emphasize the structure of the stanza and only shown a subset of the item tags.

```xml
<iq to="thommurtagh@gmail.com/TCLE6F71892" from="thommurtagh@gmail.com" id="jabber2" type="result">
  <query xmlns="jabber:iq:roster">
    <item jid="pinbawl@gmail.com" name="Tommy Watson" subscription="both"/>
    <item jid="kevin.murtagh@gmail.com" subscription="both"/>
    <item jid="folkens@gmail.com" subscription="both"/>
  </query>
</iq>
```

Again, you should notice that the stanza contains many components you can ignore.

Because there are many types of iq stanzas the server can send, your program will have to check each iq stanza received to see if it contains a query tag containing the text “jabber:iq:roster”. This is how you know that the iq stanza you received contains roster information. Then you will have to write a loop to find all of the item tags and extract the value of the jid attribute in each one. You should add each of these JabberIDs to the end of your program’s Friends menu. As with other tags, you should not assume that the attributes in an item tag will always appear in the order shown in our examples. Your code for extracting attribute values should work correctly regardless of the order in which the subparts appear.
Presence Information
The roster information the server sends tells you all of a user’s friends, but it does not tell you which friends are online and which are offline. This information is provided in presence stanzas. After the server finishes sending you the roster information when you first log in, it will send you a presence stanza for each friend who is currently online. Then, when friends log on or off while you remain connected, the server will send you presence stanzas to inform you of the changes in your friends’ status.

Like the other stanza types, presence stanzas contain far more information you can ignore than information you have to process. The following is an example of a presence stanza

```xml
<presence from="pinbawl@gmail.com/gmail.2CD90813" to="thommurtagh@gmail.com/TCLD9496921">
  <priority>24</priority>
  <caps:c node="http://mail.google.com/XMPP/client/caps" ver="1.1" ext="pmuc-v1 sms-v1 camera-v1 video-v1 voice-v1"
    xmlns:caps="http://jabber.org/protocol/caps"/>
  <status/>
  <x xmlns="vcard-temp:x:update">
    <photo/>
  </x>
</presence>
```

A presence stanza consists of a pair of presence and /presence tags surrounding a collection of other tags. All of the information you care about can be found in the attributes of the opening presence tag. The from attribute tells you the full JabberID of the account associated with this update. If the presence tag contains a type attribute whose value is “unavailable” then you know that the user associated with the from JabberID is not online. If it does not (i.e., it either contains no type attribute or a type attribute with the value “available”), then the user is online.

When your program receives a presence stanza it should extract the JabberID associated with the from attribute using indexOf and substring. This will be a full JabberID. You should remove the resource identifier to get the JabberID of the account involved rather than a particular login. Next, you should remove this JabberID from your program’s Friends menu. Then, you should search for a type attribute in the presence stanza. If you find a type “unavailable” attribute (i.e., the user is offline), you should add the JabberID back to the end of the menu using addItem. If not, you should add the JabberID back to the beginning of the menu using the insertItemAt method. As a result, all of the online users will end up at the beginning of your menu and the offline users will be placed at the end (i.e., below the “--------- offline ---------” entry in the menu).

The only danger with this approach is that if you receive a presence update for the user whose JabberID is the item currently selected in your program’s menu, you may end up with a different item selected. To prevent this, save the name of the selected item before processing a presence stanza and then restore it (using the setSelectedItem method) when you are done.

Saying Goodbye
When you first create a GTalkConnection, the code that handles the process of logging in sends an opening tag of the form

```xml
<stream:stream ... />
```

All of the stanzas you send should be nested between this tag and an appropriate closing tag. Therefore, once your program sends a closing tag of the form

```xml
</stream:stream>
```

the Google server will assume you are done and close the connection. This is how you log out.

Other Server Packets
There are several other types of stanzas the server may send to you. You will not need to interpret or process such stanzas.
**Getting Started**

As usual, you should start your work by launching BlueJ. BlueJ will open last week’s project by default. You should create a new project with a name that includes “Lab4” and your last name. Remember, do NOT include blanks or special characters in your project name. Then, close last week’s project. Finally, create a new “GUIManager” class with a name like ChatClient. The use of JComboxes in this lab will cause the compiler to display an annoying warning button when you compile your program. To avoid this, select “Preferences” from the BlueJ menu, click on the “Miscellaneous” tab and then make sure that “Show compiler warnings when unsafe collections are used” is not enabled.

**Program Structure**

If you think back to last week’s lab, you needed to write two main pieces of code: 1) a constructor containing a sequence of instructions that would execute sequentially, and 2) a buttonClicked method whose code was itself divided by an if statement into three main parts: a) code to login, b) code to logout, c) code to retrieve a message.

The structure of the program you write this week will be similar. There will be a sequence of instructions in the constructor to create the necessary GUI components. There will be a buttonClicked method whose code is divided by an if statement into three main parts: a) code to login, b) code to logout, c) code to send a message to a buddy. In addition, however, there will be a dataAvailable method to handle XMPP stanza’s received from the server. Like the buttonClicked method, the code you write in dataAvailable should be divided by an if statement into three main part: a) code to handle incoming message stanzas, b) code to handle incoming iq stanzas, and c) code to handle incoming presence stanzas.

**Implementation Plan**

You should plan to add code to implement one feature at a time and to test each feature before moving on to the next step. The following gives a possible plan for such an approach. We have tried to order the steps of this plan so that you get some practice writing simple code for String processing before taking on the harder tasks.

**Construct the GUI Interface**

1. As usual, a good place to start is to simply write the code for a constructor that will build the desired interface. As explained in the introduction, we strongly encourage you to write and enter this code before coming to lab. You will probably want to use JPanels to keep things like the login and logout buttons together.

**Implement login and logout**

2. Get logging in and out to work. To do this, you will need to construct a GTalkConnection when the login button is pressed. You will also want to add your program (i.e., this) as a message listener and enable debugging once the connection is created. Then, you should check that the fullJid method returns a non-empty string. If so, disable the login button and enable the logout button and (for now) the send button. When the logout button is pressed, send a closing /stream:stream tag. This will cause the server to close its end of the connection.

   After the user logs out, you should ensure that the appropriate buttons are enabled, that your text area and menu are cleared, and that the connection to the server is closed. Surprisingly, you should not include the code to do this in your buttonClicked method! Put the code to perform these steps in a separate connectionClosed method. When you send a /stream:stream tag, the server will close your connection. This will cause the code you put in connectionClosed to be executed as desired. Better yet, this way the code to reset the GUI will be executed even if the server closes the connection unexpectedly in some situation. Don’t forget that the method header for your definition of connectionClosed should NOT include any formal parameter declaration.

**Process Presence Updates**

3. Create a dataAvailable method. Again, remember that its header should NOT include any formal parameter declarations. Within this method, retrieve a stanza from the connection and check to see if it is a presence stanza (for now you will ignore all other stanzas received). For each presence stanza, extract the from JabberID and remove and then add it to the appropriate end of the Friends menu. Interestingly, this means you will sometimes remove items from the menu that were not there in the first place. Fortunately, the JComboBox removeItem method does not complain when you do this. Make sure to save the selected item before doing this and to restore it afterwards.
This will only include friends in the menu if they are logged in or log out while your program is running. In step 6, you will also include friends who are offline by processing the roster iq stanza.

Implement the send button
4. Add the code needed to actually send a chat message when the user presses the send button. Remember that these messages should also be displayed in your program’s text area. Within the message stanza, you must place quotes around the attribute values for from, to, and type. Recall that to include a quote in a String literal, you must place a backslash before the quote. For example, the literal "\" describes a string containing just a single double quote.

Make sure you do not send anything if the “--------- offline ---------” menu item is currently selected.

Do not worry about converting less than signs, greater than signs, and ampersands that appear in outgoing messages into the appropriate &-sequences this week. Just don’t send any!

Display incoming message
5. Modify your dataAvailable method so that it also handles incoming message stanzas. Recall that you should ignore any message stanza that does not include a body tag. For each message stanza that does include a body tag, you should extract the content between the body and /body tags and the JabberID provided by the from attribute and display these in your program’s interface.

At this point, don’t worry about handling &; or any of the other sequences starting with ampersands that may be found in incoming messages. You will do this in step 7.

Process the Roster
6. Modify dataAvailable so that it processes the iq tag used to provide your client with its user’s complete roster of friends. This will require a loop that finds all of the item tags included in the iq stanza, extracts the JabberIDs provided within these tags, and adds them to the end of the Friends menu.

Recall that the iq tag containing the roster information includes a query tag with type jabber:iq:roster. You should ignore other iq tags.

Remove &-sequences from incoming message
7. Add a loop to your code for handling incoming messages that finds each &amp; in the message and replaces it with an ampersand. Do this using a loop based on indexOf, substring and concatenation rather than using any of the other String methods Java provides. Do not worry about other &-sequences like &lt; or &gt; this week.

Submission Instructions
Take a final look! When your program seems to be working correctly, take the time to test it thoroughly. See how it behaves when you do unexpected things like leaving text fields empty. After you are confident that your program is correct, you should take a few extra minutes to look over the code before turning it in. Look carefully for any errors that might exist but not have been serious enough to cause your program to malfunction during your testing.

Next, look carefully at your programming style. Make sure your code is formatted in a way that makes it easy to read. Blank lines should be used to separate distinct components of your program from one another. Indentation should be used to distinguish relationships. At a minimum, apply the “Auto-layout” item in the BlueJ Edit menu. Make sure the names you chose for your variables help clarify the functions of those names. Avoid short, cryptic names. Include final instance variable declarations for names associated with values that determine things like the width of program text areas. Use these names in place of their values in your code.

Make sure that you include comments that explain the purposes of the instance variables that you declare. Also provide a comment describing what each method does. If a particular method contains many lines, try to break the body of the method into groups of related instructions and place an explanatory comment before each group. Make sure to include a comment before your class header that includes your name and lab section.

Run your program again to make sure it still works after any changes you made while you were polishing it up.

Now you should be ready to submit your program. You can find instructions describing how to do this on the “Labs” page of the course web site at http://www.cs.williams.edu/~cs134/Labs.html