Lab 5
Classy Chat

XMPP Client Implementation --- Part 2
Due: Wed. 3/9 at 11PM (for Mon. aft.), Thurs. 3/10 at 5PM (for Mon. evening), or Thurs. 3/10 at 11 (for Tues. aft.)

In this week’s lab we will finish work on the chat client programs from the last lab. Most real chat clients create a separate window for each friend with whom messages are exchanged. We would like you to extend your chat client from last week so that it uses multiple windows in this way. Each of these windows will have its own JTextArea for displaying the messages exchanged and a JTextField in which a user can enter messages to be sent to the friend associated with the window. In addition, your program will display a control window used to log in, log out, and to start a conversation with a selected friend. A sample of what some of these windows might look like is shown below.

Class Structure
The primary goal of this week’s lab is to introduce techniques for defining several new classes to implement a single program.

A critical step in the process of implementing a program using several classes is deciding which portions of the information needed by the entire program will be kept in each class. This in turn determines which instance variables and which methods are declared in each class. To simplify the process of writing your first multi-class program, we will provide you with suggestions for the instance variables and methods for each of the classes you should define.

You will define three separate classes:

- **XMPPControl**: This class will create the control window that displays the fields for entering the user’s login information, the login and logout buttons, the menu of friends, and a button to start a new conversation with the friend selected in the menu.

- **ChatWindow**: This class will be used to create a separate window for each conversation.

- **XMPPStanza**: The objects described by this class will not appear as new windows on your screen. Instead, the purpose of this class is to provide a better way to organize some of the code you wrote for the last lab. In particular, this class will provide methods that do the work of extracting JabberIDs and other subfields from the XML sent by Google’s XMPP server. You will create a new object of this class to represent each stanza your program receives from the server.
XMPPControl
Your XMPPControl class will be quite similar to the class that was your entire program last week. The class will contain most of the instance variables for the GUI components used in last week’s lab. You will still need the text fields and buttons used for logging in and out along with the menu to select a friend. XMPPControl will also still have an instance variable for the GTalkConnection to Google’s server. The components used to send messages and to see the messages received, however, will be moved to the ChatWindow class.

The only significant addition to the instance variables declared in XMPPControl will be a variable of type HashMap to keep track of which ChatWindow is associated with each ongoing conversation. The details of using a HashMap are discussed later in this handout.

You will make three main changes to the methods you defined last time:

1. The if statement in buttonClicked should no longer check whether the user clicked the “Send” button. Instead, it should check whether the user clicked the “Start Chat” button. If so, it should display a ChatWindow so the user can begin a conversation.
2. In dataAvailable, replace the code that displayed an incoming message with code to notify the appropriate ChatWindow that a message has been received and should be displayed in that window.
3. Revise the code in dataAvailable that examines the subparts of packets received from the server to use the methods of the XMPPStanza class.

Additionally, you will add one private method named getChatWindow to XMPPControl. This method will take a base JabberId and return the ChatWindow in which messages sent to or from that user should be displayed. It will be used when the user presses the “Start Chat” button and when an incoming message needs to be displayed.

ChatWindow
Your ChatWindow class should be designed to create a separate window for each ongoing conversation.

The constructor for a new ChatWindow should expect three parameters:
- the full JabberID of your program’s user,
- the base JabberID of the other person involved in the conversation, and
- the GTalkConnection your program has established with Google’s server.

The ChatWindow class will extend GUIManager. As a result, you can create a new window by invoking createWindow in its constructor. Its constructor should also create the appropriate GUI components and add them to its content pane. The methods you define will need to use the two JabberIDs and the network connection that are passed as parameters to the constructor. Accordingly, you should include assignment statements within your constructor to associate instance variable names with the values of these parameters.

You will only need to define two methods in this class:

- textEntered
  The code in this method will be automatically executed whenever the user presses return after typing a message into the window’s text field. It should send the contents of the window’s text field to Google’s server as a message stanza and display it in the window’s text area.

- displayIncomingMessage
  The code in this method will display the content from a message stanza received from Google’s server. The message stanza will first be accessed by invoking in.nextStanza within the dataAvailable method of your XMPPControl class. The code in dataAvailable will determine which ChatWindow should display the message and then invoke the displayIncomingMessage method of that ChatWindow, passing the text found between the <body> and </body> tags as a parameter.

Last week, we had you write a loop to replace sequences of the form &amp; found in incoming messages with single ampersands. We did this to give you practice writing loops. In fact, however, you can process &amp; and other XML &-escape sequences ( &lt; and &gt;) without writing any loops. A built-in String method named replaceAll makes this easy. If message is a String variable containing a text message and you want to replace all ampersands in the message with &amp; then you can just execute the statement
message = message.replaceAll( "&", "&amp;" );

To do the reverse (replace &amp; with a single ampersand), you would say
message = message.replaceAll( "&amp;", "&" );

You should use this method three times in textEntered and three times in displayIncomingMessage to ensure that you correctly handle messages containing &, < and > symbols.

**XMPPStanza**

The XMPPStanza class is intended to provide convenient access to the fields of a packet received from the Google XMPP server. Your XMPPStanza class should provide a constructor and the seven public methods outlined below.

- **public XMPPStanza( String packetContents )**

  The constructor takes the text of a packet from the server as its parameter. It will store that text in an instance variable for later processing. That is all the constructor will do, and the variable it associates with its parameter is the only instance variable you will need to declare in the class.

- **public boolean isPresence()**
  **public boolean isRoster()**
  **public boolean isMessage()**

  These methods produce boolean values. They will be used to determine the type of stanza received. For example, if the following text is received from the server:

  <message to="pb@gmail.com" from="tm@gmail.com" type="chat"><body>Hi</body></message>

  and this String is used to create an XMPPStanza named stanza, then

  stanza.isMessage() produces true, while stanza.isRoster() and stanza.isPresence() return false.

- **public String getType()**
  **public String getFromJid()**

  As their names suggest, these methods should extract the appropriate attribute values from the opening tags of a stanza and return the requested value as a String. The getType method should return the value of the “type” attribute. For example, assuming that stanza refers to the packet shown above, the invocation

  stanza.getType()

  should produce “chat”. If the stanza contains no “type” attribute, the method should return the empty string (“”). You will apply getType to presence stanzas to see whether a stanza contains the type="unavailable" attribute used to indicate that a user is offline.

  The getFromJid method should return the JabberID included as the value of the from attribute after removing any resource identifier (i.e., anything after a “/”). Therefore, the invocation

  stanza.getFromJid()

  should produce “tm@gmail.com” when applied to the sample message stanza shown above. This method should return the complete value of the from attribute if it contains no “/”. If the stanza contains no “from” attribute, this method should return the empty string.

- **public String getMessageBody()**
  **public String getRosterItems()**

  Each of these methods returns the contents found between a pair of matching XML tags within a certain type of stanza. The getMessageBody method should only be used when processing a message stanza. It should return the text found within a matching pair of body and /body tags found within the stanza. If no such tags exist, it should return null. For example, still assuming that stanza is the message stanza above, the invocation
stanza.getMessageBody() should produce the String “Hi”.

The getRosterItems method performs a similar operation on the stanzas used to deliver a user’s roster of friends. It should return all of the text found between a pair of query and /query tags. Like getMessageBody it should return null if such tags are not found. We emphasize that this method returns all of the roster items as a String. You will still need to use a loop employing indexOf and substring in your dataAvailable method to extract the ids included in the roster.

Since these two methods are very similar, they can be implemented by designing a single private method that takes the name of the tags that surround the desired text as a parameter. This optional approach is discussed in Appendix 1 in the online version of this handout.

The interface getRosterItems provides is not ideal. It would be nicer if the method could return the items in a form that would require no String processing in dataAvailable. Appendix 1 also discusses how this can be done.

The methods described above will be the only public methods defined in XMPPStanza. You should be able to implement these methods by copying and slightly revising segments of code you wrote last week.

Using HashMaps
To keep track of which ChatWindow goes with which conversation, you will use a class named HashMap. The HashMap class is included within the standard java.util library. To use this class you will need to add the following line to the top of the file containing the definition of your XMPPControl class:

```java
import java.util.*;
```

A HashMap is like a dictionary. You can use it to “look up” the ChatWindow associated with a friend’s JabberID. HashMap isn’t included in Java just for implementing XMPP clients. It can be used to associate values of one type with values of any other type. Because of its flexibility, you have to tell Java what sort of “words” you want to “define” in your dictionary and what sort of values will be used as “definitions”. We do this by including the types of the values to be used in angle brackets when declaring the HashMap. Since you want to associate JabberIDs (Strings) with ChatWindows, the declaration you use should look like:

```java
private HashMap<String, ChatWindow> windowDictionary;
```

When constructing a HashMap with new, you need to include those types again:

```java
windowDictionary = new HashMap<String, ChatWindow>();
```

Once a HashMap is constructed, it is very easy to use. To add a JabberID and its associated ChatWindow, you will use the put method as in:

```java
windowDictionary.put( jid, newWindow );
```

You can later ask which window is associated with a given name by using the invocation

```java
windowDictionary.get( jid )
```

If no window is associated with the name you provide as an argument to get, the value null will be returned as its result.

You will need to use the HashMap in two situations in your program:

- **Processing incoming messages.** When your program receives a message stanza, the message is probably part of an ongoing conversation for which a ChatWindow has already been created. In this case, you will find the existing window by applying the HashMap’s get method to the sender’s base JabberID. When a friend initiates a new conversation, however, you will have no existing window. Instead, you will create a new ChatWindow and add it and the sender’s JabberID to the HashMap so that you can find it later.
Responding when the user presses the “Start Conversation” button. When the user starts a new conversation, your program should look up the base JabberID selected from the Friends menu in the HashMap, creating a new window only if none already exists.

Because the steps you will perform in both of these situations are nearly identical, you should avoid repeating the needed code by defining a private method named `getChatWindow` and invoking that method from both `dataAvailable` and `buttonClicked`. The method will take a base JabberID as a parameter and return a `ChatWindow` as a result. (Hint: This method is not `void`!) It will use the `HashMap` `get` method to see if you have already created a window for its parameter id. If such a window exists, the method should use `setVisible` to make sure it can be seen. If not, it should create a new `ChatWindow` and place that window and the associated id in the `HashMap` using `put`. Either way, it should return the window as its result.

Getting Started
You should begin your work this week by making a new copy of your project folder from last week, renaming it so that its name contains your name and “Lab5” (but no blanks). Open this project with BlueJ. Then, change the name of your main class from the last lab to “XMPPControl” by simply changing it in the class header and the constructor and saving the file.

Implementation Plan
As usual, 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 our suggested plan for such an approach.

Implement the `XMPPStanza` class
1. Since this is the first time you will be writing a class that does not extend `GUIManager`, we will provide a separate “Working with Multiple Classes” handout to guide you through the process of using BlueJ to construct and test such classes. This handout will lead you through the implementation of four of the methods of the `XMPPStanza` class. For anyone who wants to start early, this handout can be downloaded from the “Labs” page of the course web page. Start by following the instructions in that handout.

2. Add definitions for the remaining methods of the `XMPPStanza` class: `getFromJid`, `getMessageBody` and `getRosterItems`. Test these methods as you tested the other methods of this class in the code pad.

Modify `dataAvailable` to use the `XMPPStanza` class
3. Modify the code of your `dataAvailable` method to take advantage of your `XMPPStanza` class. When you complete this step, the revised program should do exactly what your program did last week. The purpose of defining and using `XMPPStanza` is not to extend what your program can do but instead to reorganize it so that the individual methods included in your program are shorter and simpler.
   a. As soon as you retrieve a `String` using the `in.nextStanza` method at the beginning of `dataAvailable`, you should pass this `String` as a parameter in an `XMPPStanza` construction and associate the `XMPPStanza` constructed with a local variable.

   b. Your `dataAvailable` method from last week had a three-way if statement that used `startsWith`, `contains`, and/or other `String` methods to distinguish the three types of stanzas your program might receive from the server. Replace the conditions in this three-way if with conditions that invoke the `isMessage`, `isPresence`, and `isRoster` methods of your `XMPPStanza` class.

   c. Replace as many as possible of the invocations of `indexOf`, `substring` and other `String` methods in `dataAvailable` with invocations of the `getMessageBody`, `getRosterItems`, `getFromJid`, and `getType` methods of the `XMPPStanza` class. You should be able to get rid of such code in everything except the processing of the roster. Test your program to make sure everything still works as it did before.

Define a preliminary version of the `ChatWindow` class
4. Define a `ChatWindow` class with a constructor that expects no parameters. At this step, just write the code to create the GUI interface. You should test this by running it as if it was an independent program.

5. Next, add a `textEntered` method that appends the text typed into the text field at the bottom of the `ChatWindow` to the contents of its text area.
6. Add two parameters to the constructor for the JabberIDs of the sender and recipient of the messages that appear in the window. Modify the invocation of createWindow in the constructor so that the recipient's name will be displayed in the title bar (pass the desired title as a third parameter to createWindow) and modify textEntered so that it places the sender's name before each message it displays. You should still be able to test this as if it was a single class program. After you select “new ChatWindow( ... )” from BlueJ’s menu it will let you type in parameter values for it to use. Just type in two names in quotes.

Modify XMPPControl to create and remember ChatWindows
7. Modify your XMPPControl class by removing the message text field and the “Send” button. (At this point, it will be temporarily impossible to send messages using your program. You will fix that once you have implemented the ability to manage multiple ChatWindows.)
8. When a user logs in successfully, create a HashMap and associate it with an instance variable.
9. Add a “Start Chat” button to the XMPPControl window. Define a version of getChatWindow that simply creates and returns a ChatWindow when invoked. Invoke this method when “Start Chat” is pressed.
10. Modify getChatWindow to use the HashMap to keep track of the windows it creates. Use the HashMap get method to look for an existing window before creating a new one. If the get method finds a matching ChatWindow in the HashMap, just make it visible. If the get method returns null, create a new window and put the window and its base JabberID into the HashMap. Either way, return the window.

Modify ChatWindow (and XMPPControl) so that ChatWindow can send messages
11. Add a GTalkConnection parameter to the ChatWindow constructor. Associate the parameter value with an instance variable in the ChatWindow class, and add code to textEntered to actually send a message when the user types one into the window’s text field. Remember that the message stanza you send to the server must include the full JabberID of the sender. Modify the code in getChatWindow that creates ChatWindows to pass the GTalkConnection as a parameter. Test your program. It should now let you send messages through individual ChatWindows (but will still display all messages received in the main window).

Modify ChatWindow and XMPPControl to display incoming messages
12. Add the definition of a displayIncomingMessage method to ChatWindow. Now, modify the code in your dataAvailable method to send the contents of each incoming message to the appropriate ChatWindow's displayIncomingMessage method. This will involve using your getChatWindow method.
13. At this point, the JTextArea in your main window should not be used to display anything other than login errors. You might want to make it much smaller or replace it with a JTextField or a JLabel.

Use replaceAll to implement support for &-sequences
14. Modify the code in displayIncomingMessage and textEntered to use replaceAll to convert back and forth between the special symbols &, <, and > and their XML representations: “&amp;”, “&lt;”, and “&gt;”. Be careful about the order in which you use replaceAll. For example, if you first convert a < into &lt; and then convert all &’s into &amp;, the single < will become &amp;&lt; which is incorrect. To avoid this, you must handle incoming ampersands last and outgoing ampersands first.

A final thought: Now that your program is working, use it to start a chat session and then log out. What happens if you try to send a message? Why? You do not need to worry about fixing the behavior (although one easy solution is to just remove the logout button and a better solution is discussed in Appendix 1).

Clean Up
Make sure to take a final look through your code, checking its correctness and style. Review the comments you received on the work you submitted in previous weeks and make sure you address the issues raised. Check over the style guide accessible through the course web page and make sure you have followed its guidelines. Make sure you have included your name and lab section in a comment in each class definition.

You should now be ready to submit your program.
Appendix 1: Extra Credit Ideas
If you are interested in doing a bit more, the following optional items can increase the functionality or improve the quality of your program.

1. Use a private method to simplify the implementations of getMessageBody and getRosterItems. This method (which might be named getTagContents) would take the name of a tag as a parameter and return the text found between matching tags with this name (or null if no such tags could be found). For example, getTagContents("body") would be equivalent to invoking the getMessageBody method. However, rather than simply eliminating getMessageBody and getRosterItems from your program what we are suggesting is that you implement getTagContents as a private method and then simplify your implementations of the other two methods by using it. Your method should be general enough to work with both tags that have not attribute values specified (like body) and tags where many attributes may be specified before the ending “>” (like query).

You should be able to make a similar improvement by writing a private getAttribute method and then using it to implement getType and getFromJid.

2. Make the text wrap:

The default behavior of the JTextArea is to display the lines un-wrapped. If conversation is the name of your JTextArea, you can tell it to wrap lines that are too long to fit within the window by executing

```java
conversation.setLineWrap( true );
```

3. Make the JTextArea stretch and shrink as you resize each ChatWindow:

To do this, include the invocation

```java
this.setLayout( new BorderLayout() );
```

in the ChatWindow constructor before you add components to the content pane. To use BorderLayout, you will have to add "import java.awt.*;" at the beginning of your ChatWindow file and add a second parameter when you invoke the content pane’s add method to specify where each component should be placed. There are 5 choices: BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.WEST, BorderLayout.EAST, and BorderLayout.CENTER. Only one item can be placed in each of these five areas. The item in the center is stretched to fill any space not used by the other four regions. So, placing your text area by executing

```java
contentPane.add( conversation, BorderLayout.CENTER );
```

would ensure that it would grow if you increased the window size. Place a JPanel holding the JTextField and the label “Message:” at the bottom of the window using BorderLayout.SOUTH.

4. Close all conversation windows when the user logs out:

We mentioned at the end of our implementation plan that the design of this program has a little flaw. When a user logs out, the chat windows that had been created remain on the screen and think they are still connected to Google. If a user tries to send a message using one of these windows, an error will result. It would be best if all of the conversation windows were closed when the user logs out.

We did not want to include this as part of the basic requirements for this week’s lab, but it really is not too hard to do. It just requires learning a little more about HashMaps:

- The HashMap class provides a method named values that returns a collection of all of the objects placed in the HashMap. In addition, there is a special type of loop designed to tell Java to execute some instructions once for every item in such a collection. In particular, if your HashMap is named windowMap, then a loop of the form shown below will tell Java to execute the statements in the
loop once for each window after associating the variable `aWindow` with the window.

```java
for ( ChatWindow aWindow : windowMap.values() ) {
    one or more statements
}
```

- Use a loop of this form in your code that handles logging off to make all of the windows your program has created invisible.
- Make sure you also create a new `HashMap` each time someone logs in so that you won’t try to reuse the old windows when a new user logs in.

5. Modify `getRosterItems` so that it returns a collection of roster items instead of the entire query string. Java includes a class called `HashSet` that can be used to represent a set of objects. `HashSet` is very similar to `HashMap`. You can add new items to an instance of `HashSet` using the `add` method. You can also use the for loop syntax to iterate through the items (although unlike `HashMap`, you do not use the `values` method). Consider the following example which builds a `HashSet` of `String`s and then iterates through them, adding each one out to a `JComboBox` named `menu` in turn.

```java
HashSet<String> set = new HashSet<String>();
set.add("John");
set.add("Ringo");
set.add("Paul");
set.add("George");
for (String s : set) {
    menu.addItem(s);
}
```

- Change `getRosterItems` so that it returns a `HashSet<String>` instead of a `String`. Each element of the set returned will be one of the JabberIDs found in the roster.
- Add code inside `getRosterItems` to parse the JIDs, add them to a new `HashSet`, and then return the `HashSet`.
- Change your `XMPPControl` code to add friends to your combo box by iterating through the `HashSet` using the for loop syntax.

6. Show bold and colored messages by replacing `JTextArea` with `JEditorPane`:
   You have seen that the messages received from Google may contain odd sequences like `<` and `&gt;`. These sequences are a shared feature of XML and HTML. We had you write code to convert these sequences into the symbols they represent. An alternative is to use a GUI component that knows how to display HTML tags. This will automatically ensure that things like `<` get displayed correctly and also let you add HTML tags to do things like use different colors to display incoming and outgoing messages. The Java `JEditorPane` is such a component.

   This will be easy to do if you already know HTML. If not, feel free to ask for help.

   To use a `JEditorPane` you will need to:
   - Use a `BorderLayout` as described under item 3 above: “Make the `JTextArea` resizable”.
   - Change the variable you used to refer to your `JTextArea` to a `JEditorPane` variable and construct a `JEditorPane` instead of a `JTextArea` (the constructor expects no parameters).
   - Assuming the `JEditorPane` variable’s name is `conversation`, include the invocations
conversation.setEditable( false );
conversation.setContentType( "text/html" );
in your constructor (the first one should really be there already).

- Remove the code that converted &-sequences into the corresponding symbols.
- Use `setText` instead of `append` to place text in the `JEditorPane`. This means you will have to keep a separate `String` variable containing the contents of the entire conversation.
- Make sure that the argument to `setText` begins with `<html><body>`, ends with `</body></html>`, and includes either `<br>` or `<p>` tags between messages.
- Add any other HTML tags you want to make the text appear in color, bold font, etc.