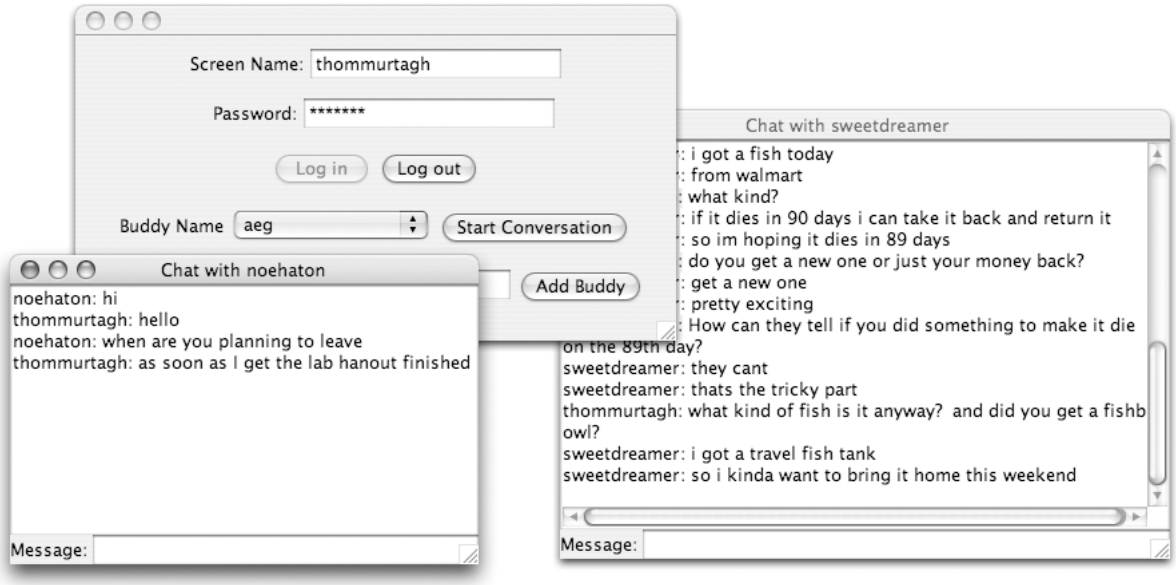
Lab 5
TOC to Me

IM Client Implementation --- Part 2
Due Oct. 21/22, 11PM

In this week's lab we will finish work on the IM client programs from the last lab. The primary goals are:

- to use multiple windows to manage more than one chat session at a time, and
- to design a program that uses more than one class.

The chat client you constructed in the last lab displayed all messages sent and received in a single `JTextArea`. Most real IM clients create a separate window for each buddy with whom messages are exchanged. We would like you to extend your chat client 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 your program's user can enter messages to be sent to the buddy associated with the window. In addition to these chat windows, your program will display a control window used to log in, log out, and to start a conversation with a selected buddy. A sample of what some of these windows might look like is shown below.



Class Structure

One major goal when dividing a program into classes is to make the structure of the program as clear and logical as possible. A critical step in this process is deciding what functionality to include in each class. This requires deciding which portions of the information needed by the entire program will be kept in each class. This in turn determines what instance variables and what 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:

- **`IMControl`**: This class will create the control window that displays the fields used to enter the user's screen name and password, the login and logout buttons, the menu of buddies, the field and button used to add names to the buddy menu. At least until the end of the lab, it should also contain a text area used to display diagnostic information and warnings.
- **`ChatWindow`**: This class will be used to create a window for each conversation that takes place while running your program.
- **`TOCServerPacket:`** 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 screen names and other subfields from the packets sent by AOL's IM server.

IMControl

Your `IMControl` class will be quite similar to the class that was your entire program last week. The class will still contain most of the instance variables used to refer to GUI components in last week's lab. You will still need the text fields and buttons used for entering login information and new buddy names along with the menu used to select a buddy to talk to. The components used to enter messages you want to send and to see the messages received, however, will be moved to the `ChatWindow` class. `IMControl` will also still have an instance variable associated with the connection to the AOL server.

The only significant addition to the instance variables declared in `IMControl` will be a variable of type `HashMap` used to keep track of which `ChatWindow`s are associated with which ongoing conversations. **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` that specified how to send a message when the user clicked the send button will be replaced by code to create a new `ChatWindow` when the user clicks the button to start a new conversation.
2. The code in `dataAvailable` that displayed an incoming message will be replaced by code to notify the appropriate `ChatWindow` that a message has been received.
3. The code within `dataAvailable` that examines the subparts of packets received from the server will be revised to use the methods of the `TOCServerPacket` class.

ChatWindow

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

The constructor for a new `ChatWindow` should expect three parameters:

- the normalized screen name of your program's user,
- the normalized screen name of the other person involved in the conversation, and
- the `FLAPConnection` your program has established with the AOL server.

Thus, the construction of a new `ChatWindow` might look like:

```
ChatWindow newConversation = new ChatWindow(myName, buddy, toAOL);
```

The `ChatWindow` class will extend `GUIManager`. As a result, you can create a new window by invoking `createWindow` in its constructor just as you invoked `createWindow` in the constructors of your earlier programs. Your `ChatWindow` constructor should create a `JTextArea` in which messages can be displayed and a `JTextField` in which message can be entered, and add these components to the content pane. Both of these components will be used by methods defined within the class and should therefore be associated with instance variables. In addition, the methods you define will need to use the two screen names and the network connection that are passed as parameters to the constructor. Accordingly, you should define instance variables to refer to these items and include assignment statements within your constructor to associate the instance variable names with the parameter values.

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. The code in your `textEntered` method should send the contents of the text field as an IM message and display it in the `ChatWindow`'s text area.
- **`messageReceived`**
  The code in this method will handle messages received for this conversation from the AOL server. These messages will actually first be accessed by invoking `in.nextPacket` within the `dataAvailable` method of your `IMControl` class. The code in `dataAvailable` will need to determine which `ChatWindow` should display the message and then invoke the `messageReceived` method of that `ChatWindow`, passing the message as a parameter.

TOCServerPacket

The `TOCServerPacket` class is intended to provide convenient access to the fields of a packet received from the AOL IM server, similar to how the `MailMessage` class provided convenient access to data recieved from the POP server in lecture. Your `TOCServerPacket` class should provide a constructor and the seven public methods outlined below.

- **Constructor.**

  The constructor takes the text of a packet received from the server as its parameter. It will store that text in an instance variable for later processing. That is the only instance variable you will need to declare in the `TOCServerPacket` class.

  To construct a `TOCServerPacket`, you provide a `String` containing the packet received from the server as a parameter in a `TOCServerPacket` construction. For example, if the statement

  ```
  String packetContents = toAOL.in.nextPacket();
  ```

  were used to access a packet from the server, then the construction

  ```
  TOCServerPacket currentPack = new TOCServerPacket( packetContents );
  ```

  could be used to construct a `TOCServerPacket` for the data received.

- **`public boolean isError()`**
  `public boolean isBuddyUpdate()`
  `public boolean isIncomingIM()`

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

  ```
  IM_IN2:somebuddy:T:T:<html><body>still there?</html></body>
  ```

  and this `String` was used to create a `TOCServerPacket` named `currentPack`, then

  ```
  currentPack.isIncomingIM()
  ```

  produces `true`, while `currentPack.isError()` and `currentPack.isBuddyUpdate()` return `false`.

- **`public String getBuddyName()`**
  `public String getErrorCode()`
  `public String getMessage()`

  As their names suggest, each of these methods should extract one of the fields of a packet received from the AOL server and return it as a String. For example, if `currentPack` is the packet above, the invocation

  ```
  currentPack.getBuddyName()
  ```

should produce the String `"somebuddy"`.

The `getMessage` method should do a little more work than the other two methods. It should return the text of the message included in an IM_IN2 packet without any of the HTML that might have been included in the packet. That is, the loop to remove HTML that you included in the last lab's `dataAvailable` method should be moved to the `getMessage` method's definition (unless you choose to modify the rest of your program to handle displaying HTML as described in the Extra Credit appendix). Assuming `currentPack` refers to the packet described above, the invocation

```
currentPack.getMessage()
```

should return "`still there?`".

- **`public boolean isBuddyOnline()`**

  This method returns a boolean value indicating whether the third field in the packet to which it is applied starts with "T". This method is meant to be used with UPDATE_BUDDY2 packets. In these packets, the third field indicates whether the user whose screen name appears as the second parameter is online ("T") or offline ("F").

The seven methods described above will be the only public methods defined in `TOCServerPacket`. You could implement these methods by simply copying and revising segments of code you wrote last week. Instead, however, we want you to define a private method

```
private String getSuffix(int n)
```

and then use this method to simplify the definitions of the public methods. The `getSuffix` method will be quite simple. It will return a String that is the suffix of the text of the packet that was passed to the `TOCServerPacket` constructor. The method will take an `int` value as a parameter. It should return the suffix of the packet that remains after removing all of the text up to and including the nth colon where n is the value of the parameter. For example, assuming `this` represents the packet shown above, we have:

Invocation	Result
`this.getSuffix(1)`	`somebuddy:T:T:<html><body>still there?</html></body>`
`this.getSuffix(2)`	`T:T:<html><body>still there?</html></body>`
`this.getSuffix(4)`	`<html><body>still there?</html></body>`

The definition of this method will include a simple loop. Each of the public "get" methods and the `isBuddyOnline` method of the `TOCServerPacket` class can then be easily defined using `getSuffix`.

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. Therefore, to use this class you will need to add the line

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

to the top of the file containing the definition of your `IMControl` class. Although you'll use `HashMaps` to keep track of `ChatWindows`, they will only be used in the `IMControl` class.

The `HashMap` class is like a dictionary. You can use it to "look up" the `ChatWindow` associated with a buddy name. Of course, `HashMap` isn't included in Java just for implementing IM clients. It can be used to

associate values of one type with values of any other type. Because HashMap is flexible in this way, you have to tell Java what sort of "keywords" 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 as keys and definitions in angle brackets when declaring the `HashMap`. Since you want to associate buddy names (Strings) with `ChatWindow`s, the declaration you use should look like:

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

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

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

Once a `HashMap` is created, it is very easy to use. To add an item to the collection of "definitions", you pass the name you want to associate with the item and the item itself as arguments to the `put` method as in:

```
windowDictionary.put( buddyName, newWindow );
```

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

```
windowDictionary.get( buddyName )
```

(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 IM messages**. When your program receives a new `IM_IN2` packet in its `dataAvailable` method, you will use the `get` method to see if a `ChatWindow` for that buddy already exists. If no window exists, you will create a new one, and add it to the `HashMap` using the `put` method. You should then pass the message received to the `ChatWindow`'s `messageReceived` method so that it will display.
- **Responding when the user presses the "Start Conversation" button**. When the user starts a new conversation, your program should lookup the name selected from the buddy menu in the `HashMap`, creating a new window only if none already exists. To avoid creating multiple windows for a single buddy, use normalized screen names when you invoke `put` and `get`.

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 "`IMControl`" 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 a possible plan for such an approach.

Implement the TOCServerPacket class

1. Begin implementing the constructor, the three boolean-valued methods used to determine packet types, and the private `getSuffix` method of the `TOCServerPacket` class as outlined in the appendix "Working with Multiple Classes" below.
2. Add definitions of the remaining public methods of the TOCServerPacket class: `getBuddyName`, `getErrorCode`, `getMessage`, and `isBuddyOnline`. Test each of these methods by creating an appropriate `TOCServerPacket` just as you tested the other methods of this class while following the instructions in the "Working with Multiple Classes" appendix.

Modify dataAvailable to use the TOCServerPacket class

3. Modify the code of your `dataAvailable` method to take advantage of your `TOCServerPacket` class:
   a. As soon as you retrieve a String using the `in.nextPacket` method at the beginning of the `dataAvailable` method you should pass this String as a parameter in a `TOCServerPacket` construction and associate the `TOCServerPacket` constructed with a local variable.
   b. Replace the invocations of `startsWith` in the branches of the `if` statement that forms the body of your `dataAvailable` methods with conditions that invoke the `isIncomingIM`, `isBuddyUpdate`, and `isError` methods of your `TOCServerPacket` class.
   c. Replace the invocations of `indexOf` and `substring` you used to extract the relevant fields of an incoming message with invocations of the `getBuddyName`, `isBuddyOnline`, `getMessage`, and `getErrorCode` methods of the `TOCServerPacket` class.
   d. Test your revised program to make sure everything still works as it did before.

   Note: At this point, the revised program will provide no functionality beyond what your program provided last week. The purpose of the process of defining and using `TOCServerPacket` was not to extend what your program could do but instead to reorganize its code so that each individual method included in your program would be shorter and simpler.

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 simply displays anything typed into the text field at the bottom of the `ChatWindow` in its text area.
6. Now add parameters to the constructor for the screen names of the sender and recipient of the messages in the window. Modify the invocation of `createWindow` in the constructor so that the recipient's name will be displayed in the title bar (just pass the 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 `IMControl` to create and remember `ChatWindows`

7. Modify your `IMControl` 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. Add a "Start Conversation" button to the IMControl window. Add code to create a new `ChatWindow` when this button is pressed.
9. Create a `HashMap` for `ChatWindows` when a user logs in successfully, and save it in an instance variable. Add code to use the `HashMap` to keep track of the windows created when the "Start Conversation" button is pressed. Add an instruction to `get` the window associated with the screen name selected in your buddy menu each time "Start Converstation" is pressed. 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 screen name into the `HashMap`.

Modify `ChatWindow` (and `IMControl`) so that `ChatWindow` can send messages

10. Add a `FLAPConnection` parameter to the `ChatWindow` constructor. Associate the parameter value with an instance variable, and add code to `textEntered` to actually send a message when the user types one into the window's text field. Modify the instruction(s) in `IMControl` that create `ChatWindows` to pass the `FLAPConnection` to AOL 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 `IMControl` to correctly display incoming messages

11. Add the definition of a `messageReceived` method to `ChatWindow` and modify the code in your `dataAvailable` method to redirect incoming IM_IN2 messages to the appropriate `ChatWindow`'s `messageReceived` method. To do this, you will have to extract the sender's name from the IM_IN2 mes-

sage and try to get the associated `ChatWindow` from the `HashMap`. If no window already exists, you should create a new one and add it to the `HashMap`.

12. 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`.

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...).

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 included your name and lab section in a comment in each class definition.

Grading

This labs will be graded on the following scale:

++	An absolutely fantastic submission of the sort that will only come along a few times during the semester.
+	A submission that exceeds our standard expectation for the assignment. The program must reflect additional work beyond the requirements or get the job done in a particularly elegant way.
✓+	A submission that satisfies all the requirements for the assignment --- a job well done.
✓	A submission that meets the requirements for the assignment, possibly with a few small problems.
✓-	A submission that has problems serious enough to fall short of the requirements for the assignment.
-	A submission that is significantly incomplete, but nonetheless shows some effort and understanding.
--	A submission that shows little effort and does not represent passing work.

Completeness / Correctness

- GUI layout
- Creating window to start conversation
- Receive message into right window
- ChatWindow class
- TOCServerPacket method definitions
- dataAvailable uses TOCPackets, and not Strings

Style

- Commenting
- Good variable names
- Good, consistent indentation
- Good use of blank lines
- Removing unused methods
- Uses public and private methods appropriately

Submission Instructions

Find the folder that BlueJ created for your project. Its name should be the one you picked for your project (something like FloydLab5).

- Click on the Desktop, then go to the "Go" menu and "Connect to Server."
- Type "cortland" in for the Server Address and click "Connect."
- Select **Guest**, then click "Connect."
- Select the volume "Courses" to mount and then click "OK." (and then click "OK" again)
- A Finder window will appear where you should double-click on "cs134",
- Drag your project's folder into either "Dropoff-Monday" or "Dropoff-Tuesday".
- Log off of the computer before you leave.

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Lab, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. The Mac will not let you submit again unless you change the name of your folder slightly. Just add something to the folder name (like the word "revised") and the resubmission will work fine.

Appendix 1: Extra Credit Ideas

These optional items can increase the functionality of your program if you are interested in doing a bit more.

1. Make the text wrap:

   The default behavior of the `JTextArea` is to display the line un-wrapped. If `conversation` is the name of your `JTextArea`, change to wrapping with

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

2. Make the `JTextArea` resizable:

   Change the layout manager for your `ChatWindow` to stretch the `JTextArea` to fill the window. To do this, include the invocation

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

   in your `ChatWindow` constructor before you add components to the content pane. To use `BorderLayout`, you will have to "`import java.awt.*;`" at the beginning of your `ChatWindow` class file. Once you do this, you will have to add a second parameter when you invoke the content pane's add method to specify where the component being added 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

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

   would ensure that it would grow if you increased the window size. Place a panel holding the `JTextField` and the label "Message:" using `Borderlayout.SOUTH`.

3. Show bold and colored messages by replacing `JTextArea` with `JEditorPane`:

   You have seen that the IM messages received from AOL are actually encoded using HTML. We suggested that you should write code to remove this HTML. An alternative is to use a GUI component that knows how to display HTML. The Java `JEditorPane` is such a component.

   To use a `JEditorPane` you will need to:

   - Use a `BorderLayout` as described above.
   - 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).
   - Revise your code to only remove <`html`>, <`body`>, </`body`> and </`html`> tags
   - Use `setText` instead of `append` to place text in the `JEditorPane`. This means you will have to keep a separate String variable of the entire conversation.
   - Make sure that the argument to setText begins with <`html`><`body`>, ends with </`html`></`body`>, and includes either <`br`> or <`p`> tags between messages.

Appendix 2: Working with Multiple Classes

Since this is the first time you will define several distinct classes as part of a single program, we will lead you through the initial steps in the definition of the `TOCServerPacket` class to show you how you can edit and test the separate classes that make up a multi-class program independently.

Add a TOCServerPacket class to your program

1. To create the `TOCServerPacket` class, click on BlueJ's "New Class..." button, but do not select `GUI-Manager` as the type of class. Instead, use the setting "`Class`". BlueJ will respond by creating a class definition with a skeletal constructor and method definition.
   Note: The "Class" template provided by BlueJ does not include any import directives. This particular class will not use any of the features of Squint or Swing, so it is not necessary to add imports. In general, however, you will have to add imports for any libraries on which the class you are defining depends.
2. Begin editing this class template by replacing the comments in the first few lines with comments that briefly describe the function of the `TOCServerPacket` class and include your name. Next, replace the sample instance variable declaration for "`x`" with a declaration for a String variable that will refer to the text of the TOC packet represented by a particular object of this class. Update the comment that describes the variable while you are at it.
3. Now, revise the definition of the `TOCServerPacket` constructor. The constructor in the template includes no formal parameter declarations and sets the (now non-existent) instance variable `x` to 0. Your constructor should expect a String consisting of the contents of a TOC packet as a parameter and should associate the value of this parameter with the `String` instance variable we told you to define in step 2.
4. Next, replace the definition of `sampleMethod` with a definition of `isIncomingIM`. This method's definition should look like:

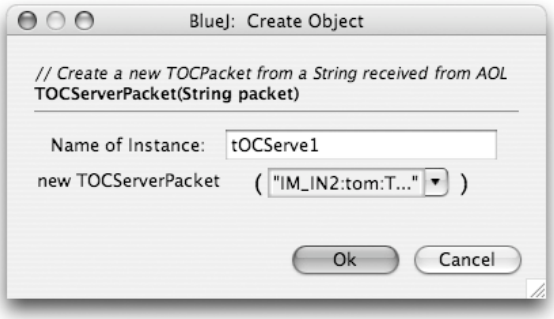
```
public boolean isIncomingIM() {
      return your-instance-variable's-name.startsWith( "IM_IN2" );
  }
```

   with the italicized text replaced by the name you selected for the class' instance variable.
5. Provide a similar definition for `isBuddyUpdate`. Compile your class, fixing any syntax errors reported until it compiles correctly. Now, even though the code you have included is quite simple, let's test that it works correctly so that you can learn how to test a class like this before trying to use it as part of a larger program.

Creating an instance of your TOCServerPacket class

6. The process of creating an object of a class like `TOCServerPacket` is similar to the process you have used to run programs in past weeks. You point your mouse at the tan rectangle in your project window that represents the `TOCServerPacket` class, depress the "ctrl" key, and then select the "`new TOCServerPacket(...`" item from the menu that appears when the mouse button is depressed.
7. After you ask BlueJ to construct a `TOCServerPacket` it will display a dialog box like the one shown on the right.

   In the dialog box that appears, the text field above the "OK" button will be empty. BlueJ wants you to fill it in with the `String` that should be provided as a parameter to the `TOCServerPacket` constructor. Type in something that looks like the beginning of a TOC IM_IN2 packet as we have shown in the image on the right. The packet does not have to be complete (typing in exactly what we have shown will actually work), but it must be included in quotes. When you have filled in the parameter value, click "OK".



In past labs, when you have clicked "OK" in such a dialog box, your program's window has appeared on the screen. This is because in the constructors of those classes, the code has started with an invocation of createWindow. The `TOCServerPacket` class does not invoke createWindow. As a result, after you click "OK", no new window will appear. Instead, all that will happen is that a little red rectangle will appear near the bottom left of your BlueJ project window.

Test the methods of your incomplete TOCServerPacket class

8. Next, point the mouse at the red rectangle that represents the object you just created and depress first the control key and then the mouse button. A menu will appear. The menu should include items that look like "`boolean isIncomingIM()`" and "`boolean isBuddyUpdate()`". Select one of these items. BlueJ will display a dialog box presenting the result of invoking the method you selected, `true` or `false`. Click "OK". Now, follow the same procedure to select the menu item for the other method. The box displayed this time should contain the opposite result. If both results were correct, you should move on and define the third boolean-valued method required, `isError`. Once this method is defined, compile the class and test the new method just as we had you test the other methods.

Define the `getSuffix` method to your class

9. Enter the code to define the `getSuffix` method. It will require a bit more code than the boolean-valued methods you have already defined. It must contain a loop. The body of this loop should use `indexOf` to find the first colon and then use `substring` to remove the colon and the text before this colon. The loop should also count how many colons have been removed in this way and stop when the count equals the method's parameter value.

Make your `getSuffix` method public

10. Ultimately, `getSuffix` should be a `private` method. We will not be able to test its behavior using BlueJ's menus, however, if it is defined this way initially. Therefore, define it as a `public` method now and then change its definition to `private` after testing it as described below.

Testing the `getSuffix` method

11. `getSuffix` is a bit harder to test than the methods defined earlier mainly because is expects parameters.
    a) Select "`new TOCServerPacket`" from the menu that appears when you control-press the mouse while pointing at the `TOCServerPacket` icon.
    b) In the dialog box that appears, type some example text that looks like a possible server packet. At the least, make sure your example contains several segments of text separated by colons.
    c) After the red icon representing the new `TOCServerPacket` appears, control-press the mouse on its icon. The menu that appears should contain an item like "`String getSuffix( int ... )`". Select this item.
    d) Fill the field in the dialog box that appears with some number less than the number of colons in your example and click "OK". Check that the answer is correct. If not, check your code and try again.

Once you are finished testing, make the method `private`, so that it is hidden inside the class.

Complete the `TOCServerPacket` class

12. Now, continue in the same manner to define and test each of the other methods required in the `TOCServerPacket` class following the instructions in the implementation plan.

Appendix 3: Sharing Your Program

In past semesters, a number of students have asked how they could send a copy of their completed IM program to someone else so that the other person could run it without a copy of BlueJ or Squint. Instructions for how to do this can found at the end of the PDF version of this handout available through the course web page. These instructions can also be applied to any of the other programs you write this semester.

Appendix 3: Sharing Your Program

In past semesters, a number of students have asked how they could send a copy of their completed IM program to someone else so that the other person could run it without a copy of BlueJ or Squint. The following instructions should provide a way to do this (although the other machine will probably have to have an appropriate version of Java installed). These instructions can also be applied to any of the other programs you write this semester.

1. Open your program in BlueJ so that its project window is displayed on the screen.
2. Open your main class ("IMControl") class and add a new method of the form:

```
public static void main( String [ ] args ) {
    new IMControl();
}
```

3. To verify that you added the new method correctly, recompile your program and then point at the main class ("IMControl") icon in the project window, press the control key and the mouse button and select "void main(String [ ] args)" from the menu that appear. Click "OK" in the dialog box that appears. If everything is fine, your program should begin to run.
4. Now, select "Create Jar File..." from the "Project" menu.
5. Select "IMControl" in the "Main Class" menu, then click "Continue".
6. Use the "New Folder" button to create a new folder to hold the standalone version of your program. Then, enter a name like "MyAIM.jar" (that ends in .jar) in the "File:" field for the file you will create and click "Create"
7. Go to "http://www.cs.williams.edu/~cs134/squint.html" in a web browser and download a copy of "squintV2.12.jar" into the folder that contains the .jar file you created in the previous step
8. Go to the labs page for the course and download a copy of TOCtools.jar. Place this in the folder with your other jar files.
9. Test that everything works by quitting BlueJ, and then finding the folder you created in step 7 and double-clicking on the jar file for your program. The program should run.
10. Point at the finder icon for the folder you created in step 7, depress control and the mouse button and select the "Create Archive" item from the menu that appears. This will create a .zip file out of your folder.
11. Send the .zip file to anyone you want. With a bit of luck, if they unpack the .zip and then click on your .jar file your program will run.