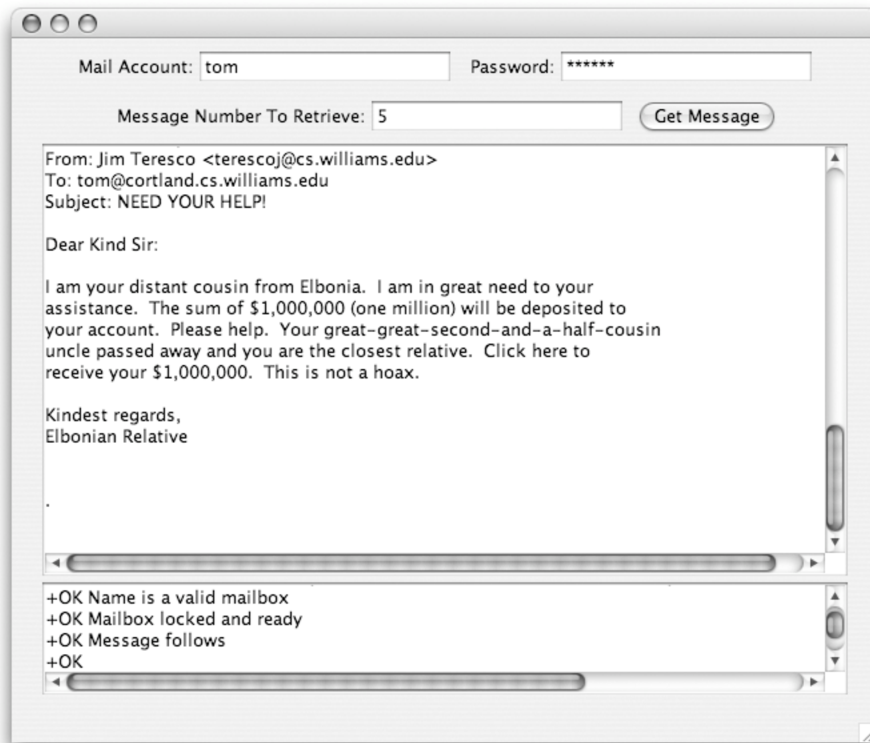
Using BlueJ's Debugging Tools: Part I

The BlueJ environment provides tools to help you identify errors that occur in your program. Knowing how to use these tools can make it much easier to understand and correct a programming mistake. For the first part of this lab, we would like you to become familiar with BlueJ's debugger by completing the following tutorial which simulates the debugging of a Java program.

The program we will work with is a not quite correct version of the POP client we had you construct in lab last week. A complete listing of the code for this program is attached to this handout. A sample of the interface that should be provided by the program you will be debugging is shown below.



At the top of the window, the program displays text fields in which the user should enter an account identifier and a password to log in. There is also a field in which the user can enter a message number to retrieve. When the user presses the "Get Message" button, the program logs in to cortland using the account message provided and displays the requested message in its window.

To download a copy of this program into your own account:

- Launch Safari (you can use another browser if you prefer, but the download instructions below are specific to Safari) and go to the "Lab Schedule and Handouts" page of the CS 134 web site.
- Find the link that indicates it can be used to download the `IncompleteLab2` program.
- Point at the link. Hold down the control key and depress the mouse button to make a menu appear.
- While still holding the control key and the mouse button, depress the option key. The item that had said "Download Linked File" should now read "Download Linked File As ..." Select this item.

- Using the dialog box that appears, navigate to your "`Documents`" folder and save the `IncompleteLab2.zip` file in that folder. (You may want to click on the arrow to the right of the "Save As:" text field to make the navigating process easier.)
- Return to the Finder, locate the `IncompleteLab2.zip` file in your Documents folder, and double-click on it to create a `IncompleteLab2` folder.
- Launch BlueJ. By default, BlueJ will open the project you created for last week's lab. DO NOT close this project quite yet.
- Using the "Open Project" item in the "File" menu, open the `IncompleteLab2` project. Once you have done this, close your project from last week's lab.

Don't Hurry!

Before any program you write in Java can be run, it must first be translated into a simpler language called Java Virtual Machine code. BlueJ performs this translation when you press the "Compile" button. By default, BlueJ tries its best to perform this translation in a way that will maximize the speed with which the computer can perform the steps in your code. This process is call *optimization*. As part of optimization, BlueJ may do things like reorder some of the steps you specified if it determines that such a reordering will speed up execution without changing the perceived behavior of your program. Unfortunately, the "perceived behavior" BlueJ avoids changing is the behavior observed by someone running your program, not the behavior observed by someone using debugging tools to examine the step-by-step execution of the code. As a result, to use the debugger effectively, we must start by telling BlueJ not to attempt to optimize the code it produces.

To do this:

- Select "Preferences..." from the "BlueJ" menu.
- Click on the "Miscellaneous" tag.
- Under "Compiler and runtime settings" make sure that "Use optimization" is not clicked.
- Click "OK" as often as necessary (twice?), and then quit and restart BlueJ.

Seeing Red

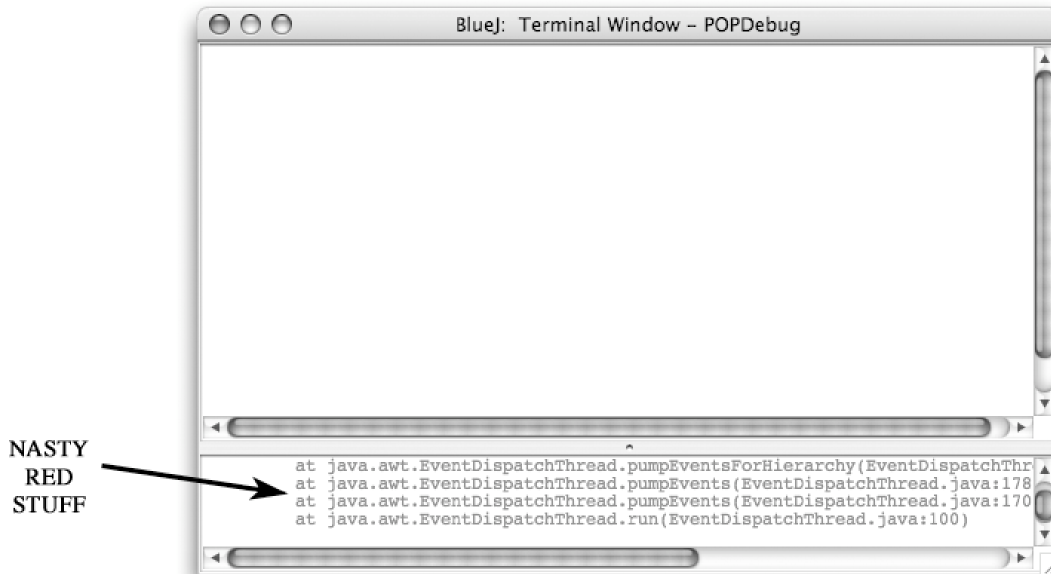
Many of you have probably already made some sort of mistake while programming that made BlueJ pop up a window full of indecipherable messages displayed in a frightening shade of red. One of your main goals during this tutorial will be to learn a bit about how to interpret these messages, which are known as *run-time error messages*.

To produce a sample of one of these messages:

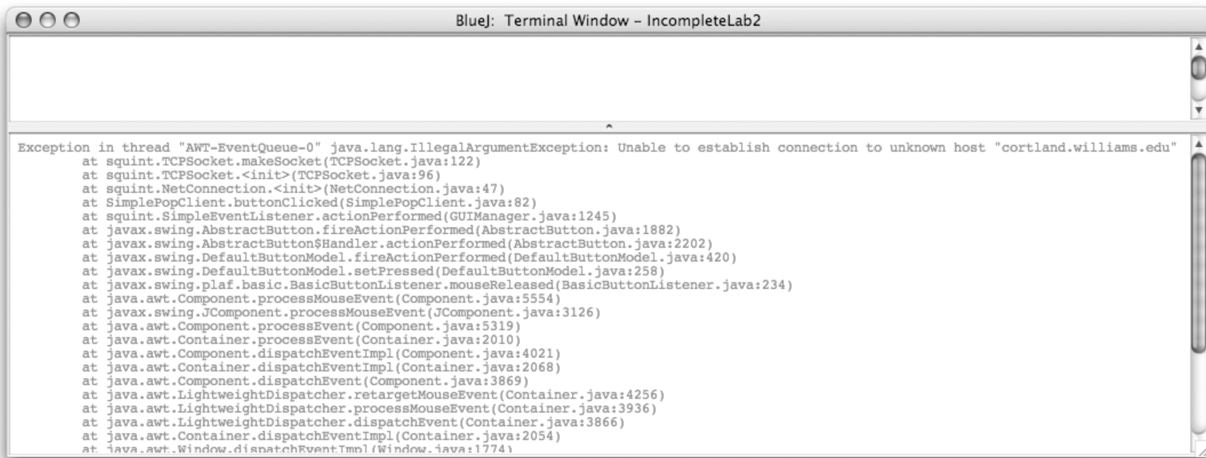
- Run the "`SimplePopClient`" program in the project we had you download (i.e., select "`new SimplePopClient()`").
- Enter your account identifier, password and a small message number.
- Click "Get Message".

That should be enough to get a window full of red ink. (If you don't see such a window, it might be hiding under your program's window. Move some windows around to see if you can find it.)

The window that appears may initially look like the image shown below. The window is divided into two sections with the red stuff that we are interested in hidden in the smaller section on the bottom.



If this is the case, then grab the dot in the middle of the bar between the upper and lower sections and drag the boundary upwards so that the bottom area fills most of the screen. You may also want to stretch the whole window a bit to make it wider. Then use the scroll bar on the bottom right to scroll to the top of the red messages so that your window finally looks like:



Interpreting BlueJ Error Messages

The most important line in the red text that BlueJ has displayed is the first line shown:

```
Exception in thread "AWT-EventQueue-0" java.lang.IllegalArgumentException:
            Unable to establish connection to unknown host "cortland.williams.edu"
```

The first line tells you that a run-time error has occurred. Java calls such errors exceptions. Most of the first half of the line is a typical bit of Java gibberish, but the second half of the line is actually quite informative. It tells us exactly what when wrong. The program was unable to establish a network connection to the maclhine `"cortland.williams.edu"`. This is not surprising since there is no such ma-

chine. The machine the program is supposed to connect to is `"cortland.cs.williams.edu"`. The programmer apparently left out the "cs" while typing in the name of the server.

This program is short enough that we could easily find the place where the server's name was misspelled and correct it. In larger programs, however, this might not be so easy. As a result, Java's red error messages contain additional information to help us identify the precise line where the error occurred. Sometimes, however, you have to work a bit to find this information.

Take a look at the next few lines that Java displayed when the error occured. They should look something like:

```
at squint.TCPSocket.makeSocket(TCPSocket.java:122)
at squint.TCPSocket.<init>(TCPSocket.java:96)
at squint.NetConnection.<init>(NetConnection.java:47)
at SimplePopClient.buttonClicked(SimplePopClient.java:82)
at squint.SimpleEventListener.actionPerformed(GUIManager.java:1245)
```

The first few of these lines should seem pretty useless. You have never even heard of most of the words that appear in these lines including `TCPSocket` and `makeSocket`. You should notice, however, that the punctuation included suggests that `TCPSocket.makeSocket( ... )` might be some sort of method invocation. In the third line, you can at least find a familiar name, `NetConnection`. Finally, in the fourth line you see two names that are actually defined within the program you are trying to debug, `SimplePopClient` and `buttonClicked`.

This collection of lines traces the major steps the program was trying to perform when the error occurred. The fourth line says that the program was executing the instruction at line 82 within the definition of the `SimplePopClient` class. Furthermore, it tells you that that line was part of the `buttonClicked` method defined within that class. (Since any slight change to the program is likely to change the numbering of its lines, the number displayed on your screen may not be exactly 82. If so, just note the number that does appear and use it in place of 82 in the instructions that follow.)

Now that you know that the error occurred in the `buttonClicked` method on line 82 (or very nearby):

- Return to the the project window and double click on the icon for `SimplePopClient` to display the source code for the class.
- Select "Go to line..." from the "Tools" menu.
- Enter the line number that appeared in the `buttonClicked` line of the error message in the dialog box and click "OK".
- BlueJ will now indicate which line in the program was the immediate source of the error by placing a red cursor at the left end of the line. The indicated line should appear near the beginning of the buttonClicked method.

If you find using "Go to line..." a little clumsy, there is a way to tell BlueJ that you would like it to always display line numbers next to the lines of your program. To do this:

- Select "Preferences" from the "BlueJ" menu.
- Click on the "Editor" tab.
- Click to check the box next to the "Display line numbers" option.

- Click “OK”.

In the remainder of this handout, when we show images of a BlueJ program editing window, they will include such line numbers at the left edge of the window.

Once you get to the right line, you should discover that it contains the construction of a `NetConnection` and that the machine name specified is indeed missing the “cs”. Correct the machine name.

Before trying to run the corrected program, take another look at the lines BlueJ displayed after the first line of its error message:

```
at squint.TCPSocket.makeSocket(TCPSocket.java:122)
at squint.TCPSocket.<init>(TCPSocket.java:96)
at squint.NetConnection.<init>(NetConnection.java:47)
at SimplePopClient.buttonClicked(SimplePopClient.java:82)
at squint.SimpleEventListener.actionPerformed(GUIManager.java:1245)
```

We now know that the fourth line describes the instruction our program was executing when the error occured, a line that constructs a new `NetConnection`. To construct a `NetConnection`, Java executes code provided within the Squint library. The third line tells us that the 47th line of the code within the library that describes how to `<init>`ialize a `NetConnection` was being executed when the error occured. Just as your code depends on code in the library, the instructions that initialize a `NetConnection` depend on other code provided within the library. In particular, the second line above indicates that as part of initializing the NetConnection, the code to initialize something called a `TCPSocket` was used.

Basically, when you construct an object defined within any of the Java libraries or invoke a method on such an object, the computer will execute code that you did not actually write. This code is instead provided within the library. Frequently, errors in your code are actually detected while executing instructions provided within the libraries. When this happens, the lines that BlueJ includes after the first line of an error message will describe steps performed within the libraries. In this case, to get information that will actually help you determine where the error occurred, you must read through the message BlueJ displays until you find the name of a method or class that you actually defined. The line number provided at the end of that message will lead you to the line in your code that was executing when the error happened.

Breakpoints

Make sure that you have “fixed” the program by adding “cs.” to the machine name.

Next, before running the program again, make sure to clear the red error messages from BlueJ’s terminal window. If you don’t, they will stay in the window and it is easy to get confused and think that a problem you have fixed is still occurring.

To clear the messages:

- Make sure the window containing the error message is active (click on it once to be sure),
- Select “Clear” from the “Options” menu (or just press “K” while holding down the command key).

Now, compile the program, run it again, and try to access an email message. You will discover that the program isn’t actually fixed yet. More red ink!

This time, the first line displayed should say

```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
```

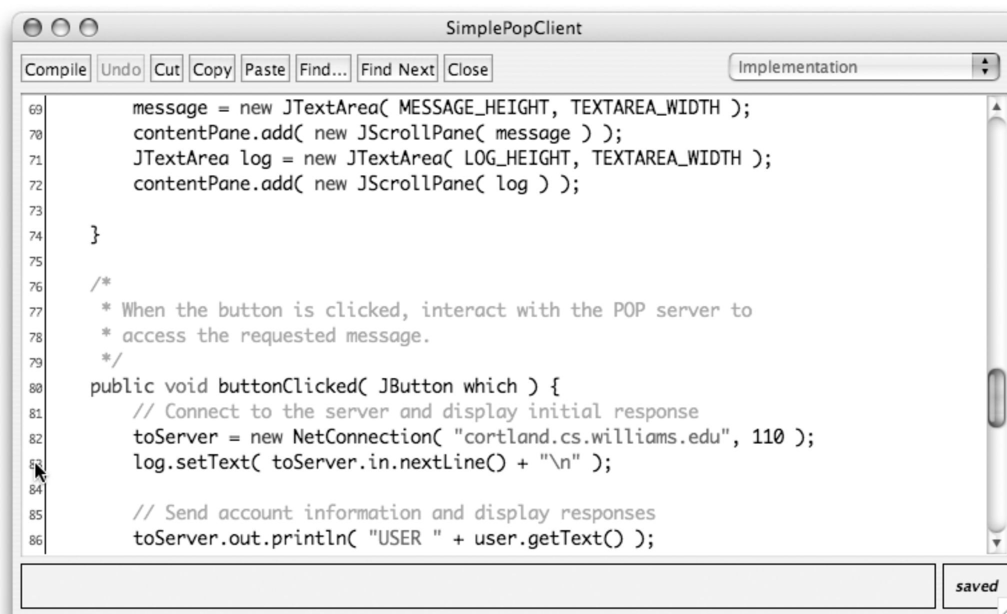
This indicates that BlueJ has encountered what it calls a null pointer exception. What this means is that the program tried to use the value associated with some variable name before any value had actually been assigned to the variable. That is, the variable is correctly declared, but no assignment statement that would associate a meaning with the name had yet been executed.

As with the previous error message, BlueJ includes a long list of lines indicating what the program was doing. These lines can be used to determine the exact line where the error occurred. All you have to do is find the first of these lines that includes the name of the class `SimplePopClient`. It is easy to do in this case because the name `SimplePopClient` appears in the very first line. Using the contents of this message, find the line within the `SimplePopClient` program that caused the error.

The line you find should contain references to two variables, `log` and `toServer`. Given that the error is a null pointer exception, the next step is to figure out which of these variables has never been assigned a meaning. In this case, by process of elimination, we can tell that `log` must be the undefined variable because an assignment to `toServer` immediately precedes the line where the error occurred.

In general, it is not always so easy to determine which name's meaning is undefined when a null pointer exception occurs. In such cases, it would be nice if there was a way to ask BlueJ to tell you the values of the variables used. Unfortunately, after an error like this has occurred, BlueJ cannot provide you with the values of variables. It is possible, however to arrange things so that when we run the program again, BlueJ will stop execution just before executing the troublesome line and let us examine the values of the variables at that point.

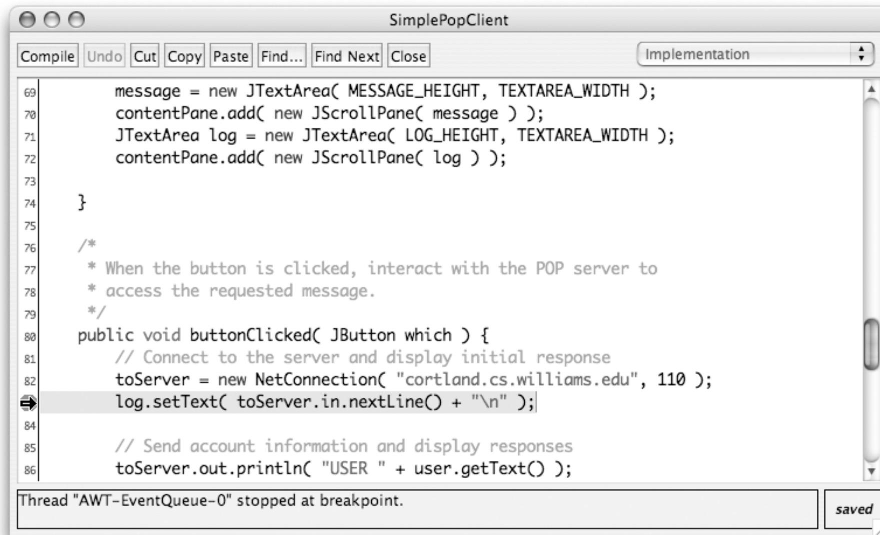
To do this, we set what is called a *breakpoint* before executing the program. This is quite easy to do. Simply position the mouse cursor in the narrow channel along the window's left edge and on the line that caused the error as shown below:



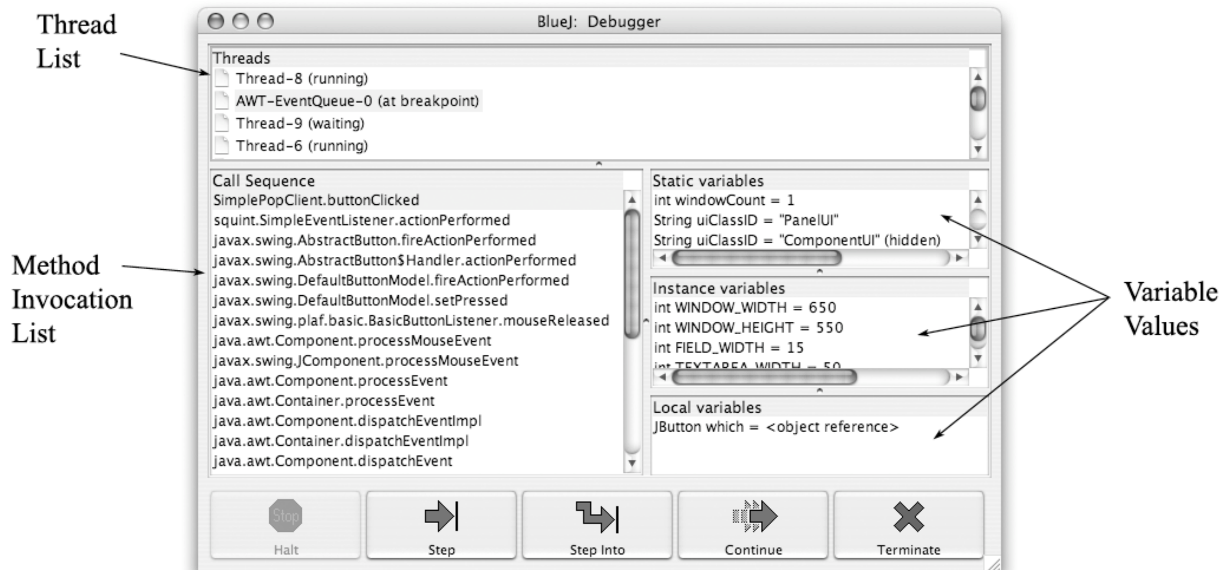
Then, click the mouse once and a little red stop sign will appear in the channel indicating that a breakpoint has been set on that line. If you place the breakpoint on the wrong line by accident, just click again on the stop sign to remove it and then click on the correct line to place another stop sign. In fact, you can

place stops signs on as many lines as you like and the computer will stop as soon as it reaches any of these lines.

Now, run the program again, enter your account id, your password and a message number then click “Get Message”. Then attempt to retrieve a message. This time, your program should stop just before the error would occur. BlueJ will let you know that this has happened in two ways. First, in the window displaying the program’s text, it will indicate the line where execution was stopped by highlighting it and placing an arrow in the channel with the stop signs as shown below:



In addition, a new “Debugger” window will appear. An image of the debugger window is shown below.



Chances are that the window that appears on your screen and its sub-panels won’t be quite big enough to see things well. If so, take a little time to resize the components of your debugger window until they look more like the image above.

In the upper portion of the window, it lists the “Threads” that are currently executing under BlueJ’s control. Each independent activity going on inside your machine is called a thread. The only thread you really care about is the one running your code. In this case, BlueJ has indicated that this is being done by the thread named `AWT-EventQueue-0`.

Below the thread list on the left side of the window, BlueJ provides a list of method names similar to the one provided in the error message we examined earlier. At the top of the list is `buttonClicked`, because that is the method whose code was executing when the breakpoint was encountered. Below `buttonClicked` is `actionPerformed`, the method which invoked `buttonClicked`, an so on.

On the right side of the window, BlueJ displays the variable names being used and the values associated with them. These are divided into three sections: the static variables (which we don’t even know about yet), the instance variables, and the local variables.

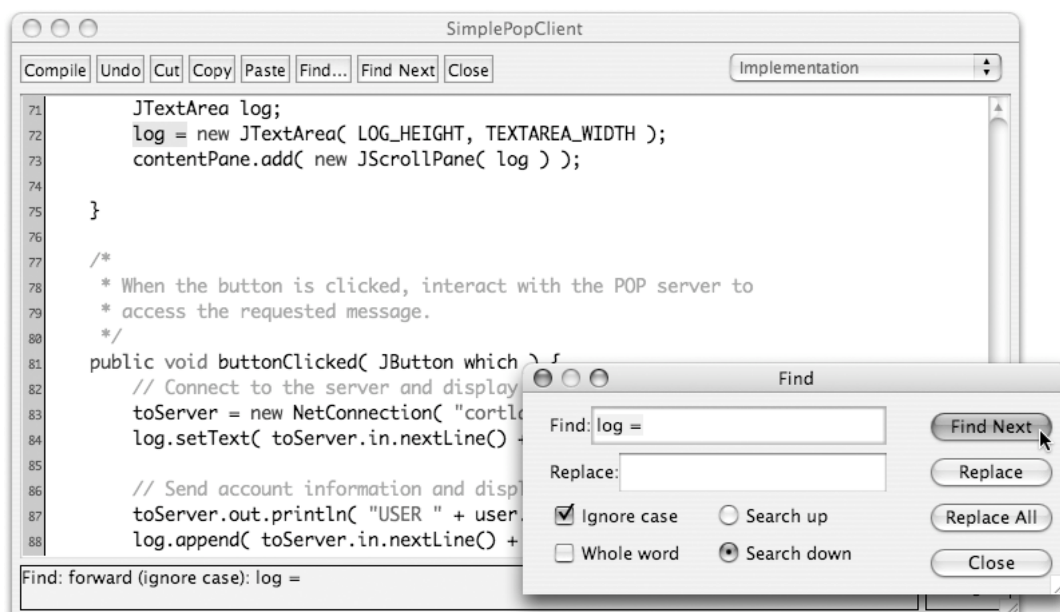
One of the two variables `log` and `toServer` must be undefined. So use the scroll bar to the right of the “Instace variables” pane of the debugger window to find them in this list. They will probably be near the top. The value of one of them will be described as “null” within the debugger window. This is the variable that was never assigned a value. The other variable will be described as an “object reference”.

As predicted, you should find that the `log` variable is undefined or “null”. Now, the hard part begins, you have to look through the program to see how this could happen. The most obvious explanation would be that the programmer simply forgot to assign a value to the variable. The BlueJ debugger cannot really help you determine if this is the case, but the BlueJ editor can.

You can use the editor to search for any assignment statements involving `log`:

- Select “Find” from the “Tools” menu (or just press command-F).
- Enter “log =” in the dialogue that appears.
- Press “Find Next” once or twice.

Doing this should be enough to find the assignment to log that appears in the constructor as shown below.



If you look at the line before the assignment, you will discover the source of the problem. The constructor includes a declaration for `log`. `log` is also declared as an instance variable by the declaration

```
// +OK and -ERR messages from the server are displayed in this area
JTextArea log;
```

which appears before the constructor. Java considers the instance variable `log` and the local variable `log` to be distinct and separate. The assignment in the constructor associates a meaning with the local variable `log`, but not with the instance variable. All the references to `log` found within the buttonClicked method are interpreted as references to the instance variable `log`.

To fix this problem, all you need to do is delete the declaration of `log` that appears right before the assignment to `log` in the constructor.

- Make this change.
- Compile and run the program again to verify that it now works correctly.

You can now begin working on the improved email program described in the lab 3 handout. We should warn you that when we wrote the "Getting Started" section of Lab 3, we did not account for the fact that you would be completing this tutorial first. The lab handout says that when you start BlueJ, it will open your Lab 2 project. Now, a) you will already have started BlueJ, and b) it will have the program you just debugged open. This may be a bit confusing, but it should not cause any real problems.

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

/*
 * SimplePopClient --- This program allows its user to view
 * mail messages accessed through a POP client
 */
public class SimplePopClient extends GUIManager {
    // Change these values to adjust the size of the program's window
    private final int WINDOW_WIDTH = 650, WINDOW_HEIGHT = 550;

    // Size of all of the text fields used in the program
    private final int FIELD_WIDTH = 15;

    // Dimensions for text areas used to display messages and server responses
    private final int TEXTAREA_WIDTH = 50;
    private final int LOG_HEIGHT = 5;
    private final int MESSAGE_HEIGHT = 20;

    // Used to enter the POP account identifier
    JTextField user;

    // Used to enter the POP account password
    JPasswordField pass;

    // Used to enter the number of the message to retrieve
    JTextField messageNum;

    // Email messages are displayed in this area
    JTextArea message;

    // +OK and -ERR messages from the server are displayed in this area
    JTextArea log;

    // The current connection to the server
    NetConnection toServer;

    /*
     * Install all of the required GUI components
     */
    public SimplePopClient() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

        // Each JLabel/JTextArea pair is placed together in a pane of their own
        JPanel curPanel;

        // Create fields for entering the account information
        curPanel = new JPanel();
        curPanel.add( new JLabel( "Mail Account:" ) );
        user = new JTextField( FIELD_WIDTH );
        curPanel.add( user );
        contentPane.add( curPanel );

        curPanel = new JPanel();
        curPanel.add( new JLabel( "Password:" ) );
        pass = new JPasswordField( FIELD_WIDTH );
        curPanel.add( pass );
        contentPane.add( curPanel );

        // Create a field for the message number
        curPanel = new JPanel();
        curPanel.add( new JLabel( "Message Number To Retrieve:" ) );
        messageNum = new JTextField( FIELD_WIDTH );
        curPanel.add( messageNum );
        contentPane.add( curPanel );

        // Install the button and text areas in the window
        contentPane.add( new JButton( "Get Message" ) );
        message = new JTextArea( MESSAGE_HEIGHT, TEXTAREA_WIDTH );
        contentPane.add( new JScrollPane( message ) );
        JTextArea log;
        log = new JTextArea( LOG_HEIGHT, TEXTAREA_WIDTH );
        contentPane.add( new JScrollPane( log ) );

    }

    /*
     * When the button is clicked, interact with the POP server to
     * access the requested message.
     */
    public void buttonClicked( JButton which ) {
        // Connect to the server and display initial response
        toServer = new NetConnection( "cortland.williams.edu", 110 );
        log.setText( toServer.in.nextLine() + "\n" );

        // Send account information and display responses
        toServer.out.println( "USER " + user.getText() );
        log.append( toServer.in.nextLine() + "\n" );
        toServer.out.println( "PASS " + new String( pass.getPassword()) );
        log.append( toServer.in.nextLine() + "\n" );

        // Retrieve and display the specified email message
        toServer.out.println( "RETR " + messageNum.getText() );
        log.append( toServer.in.nextLine() + "\n" );
        toServer.addMessageListener( this );

        // Terminate the connection
        toServer.out.println( "QUIT" );
    }

    public void dataAvailable( ) {
        String serverResponse = toServer.in.nextLine();
        if ( serverResponse.startsWith( "+OK" ) ) {
            log.append( serverResponse + "\n" );
            toServer.close();
        } else {
            message.append( serverResponse + "\n" );
        }
    }

    public void connectionClosed( ) {
        log.append( "Server has closed connection\n" );
        toServer.close();
    }
}
```