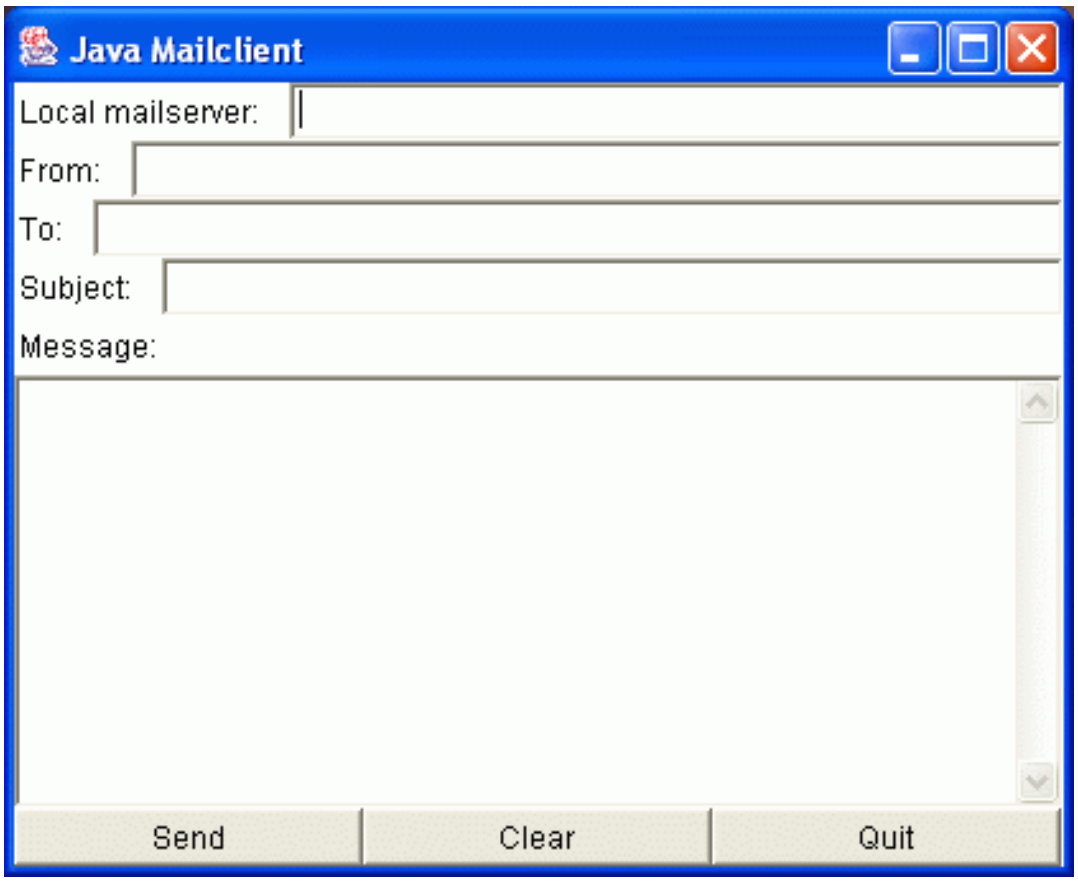
Programming Assignment 2: A Mail User Agent in Java

In this lab you will implement a mail user agent that sends mail to other users. Your task is to program the SMTP interaction between the MUA and the local SMTP server. The client provides a graphical user interface containing fields for entering the sender and recipient addresses, the subject of the message and the message itself. Here's what the user interface looks like:



With this interface, when you want to send a mail, you must fill in complete addresses for both the sender and the recipient, i.e., `user@someschool.edu`, not just simply `user`. You can send mail to only one recipient. You will also need to give the name (or IP address) of your local mailserver. If you do not know the name of your local mailserver, see Querying the DNS below for more information on how to obtain the address of the local mailserver.

When you have finished composing your mail, press *Send* to send it.

The Code

The program consists of four classes:

MailClient	The user interface
Message	Mail message

Envelope	SMTP envelope around the Message
SMTPConnection	Connection to the SMTP server

You will need to complete the code in the `SMTPConnection` class so that in the end you will have a program that is capable of sending mail to any recipient. The code for the `SMTPConnection` class is at the end of this page. The code for the other three classes is provided on this page.

The places where you need to complete the code have been marked with the comments `/* Fill in */`. Each of the places requires one or more lines of code.

The `MailClient` class provides the user interface and calls the other classes as needed. When you press *Send*, the `MailClient` class constructs a `Message` class object to hold the mail message. The `Message` object holds the actual message headers and body. Then the `MailClient` object builds the SMTP envelope using the `Envelope` class. This class holds the SMTP sender and recipient information, the SMTP server of the recipient's domain, and the `Message` object. Then the `MailClient` object creates the `SMTPConnection` object which opens a connection to the SMTP server and the `MailClient` object sends the message over the connection. The sending of the mail happens in three phases:

1. The `MailClient` object creates the `SMTPConnection` object and opens the connection to the SMTP server.
2. The `MailClient` object sends the message using the function `SMTPConnection.send()`.
3. The `MailClient` object closes the SMTP connection.

The `Message` class contains the function `isValid()` which is used to check the addresses of the sender and recipient to make sure that there is only one address and that the address contains the @-sign. The provided code does not do any other error checking.

Reply Codes

For the basic interaction of sending one message, you will only need to implement a part of SMTP. In this lab you need only to implement the following SMTP commands:

Command	Reply Code
DATA	354
HELO	250
MAIL FROM	250
QUIT	221
RCPT TO	250

The above table also lists the accepted reply codes for each of the SMTP commands you need to implement. For simplicity, you can assume that any other reply from the server indicates a fatal error and abort the sending

of the message. In reality, SMTP distinguishes between transient (reply codes 4xx) and permanent (reply codes 5xx) errors, and the sender is allowed to repeat commands that yielded in a transient error. See Appendix E of RFC 821 for more details.

In addition, when you open a connection to the server, it will reply with the code 220.

Note: RFC 821 allows the code 251 as a response to a RCPT TO-command to indicate that the recipient is not a local user. You may want to verify manually with the `telnet` command what your local SMTP server replies.

Hints

Most of the code you will need to fill in is similar to the code you wrote in the WebServer lab. You may want to use the code you have written there to help you.

To make it easier to debug your program, do not, at first, include the code that opens the socket, but use the following definitions for `fromServer` and `toServer`. This way, your program sends the commands to the terminal. Acting as the SMTP server, you will need to give the correct reply codes. When your program works, add the code to open the socket to the server.

```
fromServer = new BufferedReader(new InputStreamReader(System.in));
toServer = System.out;
```

The lines for opening and closing the socket, i.e., the lines `connection = ...` in the constructor and the line `connection.close()` in function `close()`, have been commented out by default.

Start by completing the function `parseReply()`. You will need this function in many places. In the function `parseReply()`, you should use the `StringTokenizer`-class for parsing the reply strings. You can convert a string to an integer as follows:

```
int i = Integer.parseInt(argv[0]);
```

In the function `sendCommand()`, you should use the function `writeBytes()` to write the commands to the server. The advantage of using `writeBytes()` instead of `write()` is that the former automatically converts the strings to bytes which is what the server expects. Do not forget to terminate each command with the string CRLF.

You can throw exceptions like this:

```
throw new Exception();
```

You do not need to worry about details, since the exceptions in this lab are only used to signal an error, not to give detailed information about what went wrong.

Optional Exercises

You may want to try the following optional exercises to make your program more sophisticated. For these exercises, you will need to modify also the other classes (MailClient, Message, and Envelope).

- **Verify sender address**. Java's System-class contains information about the username and the InetAddress-class contains methods for finding the name of the local host. Use these to construct the sender address for the Envelope instead of using the user-supplied value in the From-header.
- **Additional headers**. The generated mails have only four header fields, From, To, Subject, and Date. Add other header fields from RFC 822, e.g., Message-ID, Keywords. Check the RFC for the definitions of the different fields.
- **Multiple recipients**. Currently the program only allows sending mail to a single recipient. Modify the user interface to include a Cc-field and modify the program to send mail to both recipients. For a more challenging exercise, modify the program to send mail to an arbitrary number of recipients.
- **More error checking**. The provided code assumes that all errors that occur during the SMTP connection are fatal. Add code to distinguish between fatal and non-fatal errors and add a mechanism for signaling them to the user. Check the RFC to see what the different reply codes mean. This exercise may require large modifications to the `send()`, `sendCommand()`, and `parseReply()` functions.

Querying the DNS

The Domain Name System (DNS) stores information in resource records. Normal name to IP-address mappings are stored in type A (Address) resource records. Type NS (NameServer) records hold information about nameservers and type MX (Mail eXchange) records tell which server is handling the mail delivery of the domain.

The server you need to find is the server handling the mail for your school's domain. First, you must find the nameserver of your school and then query this nameserver for the MX-host. Assuming you are at Someschool and your domain is `someschool.edu`, you would do the following:

1. Find the address of a nameserver for the top-level domain `.edu` (NS query)
2. Query the nameserver for .edu about the nameserver for the domain `someschool.edu` to get the address of Someschool's nameserver. (NS query)
3. Query Someschool's nameserver for MX-records for the domain `someschool.edu`. (MX query)

Ask your local system administrator how to perform DNS queries manually.

Under Unix you can query DNS manually with the `nslookup`-command. The syntax of the `nslookup`-command is as follows. Note that the argument `host` can also be a domain.

Normal query	`nslookup host`

Normal query using a given server	`nslookup host server`
NS-query	`nslookup -type=NS host`
MX-query	`nslookup -type=MX host`

The reply to the MX-query may contain multiple mail exchangers. Each of them is preceded by a number which is the preference value for this server. Lower preference values indicate preferred servers so you should use the server with the lowest preference value.

SMTPConnection.java

This is the code for the SMTPConncetion class that you will need to complete. The code for the other three classes is provided on this page.

```
import java.net.*;
import java.io.*;
import java.util.*;

/**
 * Open an SMTP connection to a mailserver and send one mail.
 *
 */
public class SMTPConnection {
    /* The socket to the server */
    private Socket connection;

    /* Streams for reading and writing the socket */
    private BufferedReader fromServer;
    private DataOutputStream toServer;

    private static final int SMTP_PORT = 25;
    private static final String CRLF = "\r\n";

    /* Are we connected? Used in close() to determine what to do. */
    private boolean isConnected = false;

    /* Create an SMTPConnection object. Create the socket and the
       associated streams. Initialize SMTP connection. */
    public SMTPConnection(Envelope envelope) throws IOException {
        // connection = /* Fill in */;
        fromServer = /* Fill in */;
        toServer =   /* Fill in */;
```

```
        /* Fill in */
        /* Read a line from server and check that the reply code is 220.
           If not, throw an IOException. */
        /* Fill in */

        /* SMTP handshake. We need the name of the local machine.
           Send the appropriate SMTP handshake command. */
        String localhost = /* Fill in */;
        sendCommand( /* Fill in */ );

        isConnected = true;
    }

    /* Send the message. Write the correct SMTP-commands in the
       correct order. No checking for errors, just throw them to the
       caller. */
    public void send(Envelope envelope) throws IOException {
        /* Fill in */
        /* Send all the necessary commands to send a message. Call
           sendCommand() to do the dirty work. Do _not_ catch the
           exception thrown from sendCommand(). */
        /* Fill in */
    }

    /* Close the connection. First, terminate on SMTP level, then
       close the socket. */
    public void close() {
        isConnected = false;
        try {
            sendCommand( /* Fill in */ );
            // connection.close();
        } catch (IOException e) {
            System.out.println("Unable to close connection: " + e);
            isConnected = true;
        }
    }

    /* Send an SMTP command to the server. Check that the reply code is
       what is is supposed to be according to RFC 821. */
    private void sendCommand(String command, int rc) throws IOException {
        /* Fill in */
        /* Write command to server and read reply from server. */
        /* Fill in */
```

```
        /* Fill in */
        /* Check that the server's reply code is the same as the parameter
           rc. If not, throw an IOException. */
        /* Fill in */
    }

    /* Parse the reply line from the server. Returns the reply code. */
    private int parseReply(String reply) {
        /* Fill in */
    }

    /* Destructor. Closes the connection if something bad happens. */
    protected void finalize() throws Throwable {
        if(isConnected) {
            close();
        }
        super.finalize();
    }
}
```