Programming Assignment 1: Email Client with Browser Capabilities

For this assignment you are allowed to work and submit in groups of two.

This assignment counts for 10% of the total module mark. Failure in this assessment may be compensated for by higher marks in other components of the module. The purpose of the assignment is to learn the SMTP and HTTP protocols and socket programming. It partly assesses the following learning outcome:

- Students should be able to program applications and protocols for computer networks.

The assignment is due Friday, 18 October 2019, 17:00.

Late submissions are subject to the University standard system of penalties. (cf. Section 6 in Code of Practice on Assessment)

In this assignment you will implement a mail client that sends mail to other users. You can use any operating system when programming this assignment. However, your programm will be tested on one of the csc linux machines (lxfarm01-lxfarm08). **Make sure that your programm works on those machines**. In particular your program should compile from the command line. In the lab you can open a **MobaXterm** session to work on those machines.

There are two parts to this assignment, part one is worth up to 60%, part two is worth up to 20% (by completing parts one and two you could score up to 80%). The remainder of the marks are awarded for any optional exercises you decide to tackle. Therefore, the only way to score 100% is to code parts one and two and some optional extras.

Submission Instructions

Make sure that all your submitted files contain your names and student-IDs. Please submit only from one student account but make sure to include both names in the header.

No extras implemented

If you implemented only part one and part two of the assignment (and NO extras) then you have to submit two files:

- `SMTPConnect.java`
- `HttpInteract.java`

With extras

You will have to first submit a no extras version of `SMTPConnect.java` and `HttpInteract.java`. This will be used for assessing part 1 and 2 of the assignment.

If you implemented some extras you have to write a small text file called `extras.txt` where you briefly describe the implemented extras and how to test them. So you have to submit:

- `SMTPConnect.java` (no extras version)
- `HttpInteract.java` (no extras version)
- `extras.txt`
- A self contained zip-archive that includes all java files for running the altered EmailClient (even the unaltered java files).

Important Instructions

- Make sure that all your submitted java files contain your names and student-IDs in the header.
- Submit only from one student account but make sure to include both names in the header.
- Upload only java and txt-files. The submission server only allows you to do this one after the other.

If you finished your assignment click here to SUBMIT.

Overview

In this programming assignment you are going to implement a mail client that sends mail to other users. Here's what the user interface looks like:

[Interface]

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 and portnumber of your local SMTP mailserver. For security reason, we will not use a live SMTP server. Instead we use **MailCatcher** (https://mailcatcher.me), a simple SMTP server that catches any message sent to it and displays it a web interface. MailCatcher is installed on the main student webserver. The web interface can be accessed via http://student.csc.liv.ac.uk:1080. To send emails to MailCatcher, you can use the SMTP server **student.csc.liv.ac.uk** on port **1025**, which is already filled in. Emails delivered to this SMTP server will be displayed in the web frontend. Defaults values for the fields in the mailclient can be specified in `Emailclient.java`. You can also specify default email addresses here, which will save you from a lot of typing.

To use the described MailCatcher setup, you will have to run your code on a machine in the CS network. If you want to test your code at home I suggest that you use a fake SMTP server, which will act like a normal SMTP server but instead of sending the email it will just display it. You can also use MailCatcher or some other fake SMTP server; e.g. fakeSMTP (easier to run). When running this pick a random port number between 1024 and 65535. In the EmailClient you will need to use the same port number and **localhost** as SMTP server name.

In the "HTTP://" field you can provide a URL (e.g.: "comp211.gairing.com/test.txt"). When you press the "Get" buttom, the source of the corresponding object should appear in the "Message" field.

Your code will be tested on one of the csc linux machines:
lxfarm01 ... lxfarm08.csc.liv.ac.uk

The Code

For the first part you will need to complete the code in the `SMTPConnect` class so that in the end you will have a program that is capable of sending mail to any recipient.
For the second part you will need to complete the code in the `HttpInteract` class so that in the end your program can download the mail message from any web address.

The program consists of five classes:

EmailClient	The user interface
EmailMessage	Email message
SMTPConnect	Connection to the SMTP server
HttpInteract	Interaction with HTTP server

The code for the `SMTPConnect` class and the `HttpInteract` class is at the end of this page. The code for the other classes is provided on this page.
A zip-archive of all files is here.

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.

Comments within the java-files also give additional instructions/hints that you should follow.

The `EmailClient` class provides the user interface and calls the other classes as needed.

Part One (SMTPConnect)

The first 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.

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

When you press *Send*, the `EmailClient` class constructs a `EmailMessage` class object to hold the mail message. The `EmailMessage` object holds the actual message headers and body, SMTP sender and recipient information, snd the SMTP server of the recipient's domain. Then the `EmailClient` object creates the `SMTPConnect` object which opens a connection to the SMTP server and the `EmailClient` object sends the message over the connection. The sending of the mail happens in three phases:

1. The `EmailClient` object creates the `SMTPConnect` object by calling the constructor of `SMTPConnect`. This opens the connection to the SMTP server (given in the `DestHost` and `DestHostPort`variables of the `EmailMessage` object.).
2. The `EmailClient` object sends the message using the function `SMTPConnect.send()`.
3. The `EmailClient` object closes the SMTP connection.

The `EmailMessage` 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 more involved 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 `telnet` or `nc` 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 *HTTP and SMTP 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.

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 IO exceptions like this:

```
            throw new IOException();
```

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.

Part Two (HttpInteract)

For part two you have to uncommend a block of code in the `GetListener` method of the `EmailClient` class.

The graphical interface contains a field marked "HTTP://" which can be used to enter a URL.

When you press *Get*, the following happens:

1. The `EmailClient` constructs a `HttpInteract` class object to hold the request message.
2. The `EmailClient` sends the request using the function `HttpInteract.send()`.
   This function returns a `String` containing the requested object, or an error message, if one occured.
3. The `EmailClient` updates the message field to the returned `String`.

Your job is to complete the code of the `HttpInteract` constructor and the function `HttpInteract.send()`.

The constructor splits the URL into its *host* and *path* part, contruct the `requestMessage` and assign the corresponding class variables.

In the function `HttpInteract.send()` you have to:

1. Open a TCP connection to the *host* and assign the appropriate input and output streams to the connection.
2. Request the object by sending the `requestMessage` into the OuputStream.
3. Read the status line from the InputStream and extract the status code.
4. If the request was successful (status code = 200) then read the headers from the InputStream and extract the lenght of the body. This information is stored in the `"Content-Length:"` (or sometimes `"Content-length:"`) field. For reading the status line and header you can use the `readLine()` of the `BufferedReader`
5. Read the body from the InputStream and return a `String` containing the body. Here you should use one of the `read()` methodes of the `BufferedReader`.

If you now press *Send* and you specified sender and recepient addresses, you should be able to send the downloaded object as the body of an E-mail.

Remark: This setup will only work for unsecured webservers (i.e., no HTTPS). More and more webservers are now secured and use SSL sockets. For simplicity (and because we haven't covered SSL yet), we will restrict to unsecure HTTP. Here are some URLs that we will use to test your code:

- comp211.gairing.com/test.txt
- comp211.gairing.com/foo/bar/page2.html
- comp211.gairing.com/test404/index.php
- comp211.gairing.com/nolength.php

For the extras this might also be useful:

- comp211.gairing.com/redirect/index.php

Optional Exercises

As meantioned above, to get a mark >80% you have to implement some extras. It is up to you what you implement. If you decide to do some extras you also have to write a short textfile called `extras.txt` with a short description of the implemented extras and how one can check them.

Important: If you implement extras you will have to submit 2 versions of this assignment:

- One without the extras, which should work with the standard version of the other java files,
- and one version with the extras as zip-archive that includes all java files for running the EmailClient, even those that were not altered.

Here are some **suggestions** of things you might wanna try: For these exercises, you will need to modify also the other classes.

- **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.
    - For example try to set MIME headers to include the **"Content-Type"** specified in the downloaded object also in the E-mail. On a HTML enabled email client, you can then see a intepreted html file, not just the source.
    - At the moment you can only download and send ASCII files. Use some encoding to do the same for **binary files**, e.g. jpg-images. Again MIME headers can help you.
- **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.
- **Follow Redirections** If the http server replies with a 301 or 302 status code then extract new location of object from the header or body of the reply and download object from there. Change also the URL field in the EmailClient accordingly.

SMTPConnect.java

This is the code for the SMTPConncetion class that you will need to complete for part 1. The code for the other classes is provided on this page.

```
/*************************************
 * Filename:  SMTPConnect.java
 * Names:
 * Student-IDs:
 * Date:
 *************************************/
import java.net.*;
```

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

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

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

    private static final String CRLF = "\r\n";

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

    /* Create an SMTPConnect object. Create the socket and the
       associated streams. Initialise SMTP connection. */
    public SMTPConnect(EmailMessage mailmessage) throws IOException {
        // Open a TCP client socket with hostname and portnumber specified in
        // mailmessage.DestHost and  mailmessage.DestHostPort, respectively.
 connection = /* Fill in */;

        // attach the BufferedReader fromServer to read from the socket and
        // the DataOutputStream toServer to write to the socket
        fromServer = /* Fill in */;
 toServer =   /* Fill in */;

 /* Fill in */
 /* Read one 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 = InetAddress.getLocalHost().getHostName();
 sendCommand( /* Fill in */ );

 isConnected = true;
    }

    /* Send message. Write the correct SMTP-commands in the
       correct order. No checking for errors, just throw them to the
       caller. */
    public void send(EmailMessage mailmessage) 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 SMTP 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 */
    }

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

HttpInteract.java

This is the code for the HttpInteract class that you will need to complete for part 2. The code for the other classes is provided on this page.

```
/*************************************
 * Filename:  HttpInteract.java
 * Names:
 * Student-IDs:
 * Date:
 *************************************/

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

/**
 * Class for downloading one object from http server.
 *
 */
public class HttpInteract {
 private String host;
 private String path;
 private String requestMessage;


 private static final int HTTP_PORT = 80;
 private static final String CRLF = "\r\n";
 private static final int BUF_SIZE = 4096;
 private static final int MAX_OBJECT_SIZE = 102400;

  /* Create HttpInteract object. */
 public HttpInteract(String url) {

  /* Split "URL" into "host name" and "path name", and
   * set host and path class variables.
   * if the URL is only a host name, use "/" as path
   */

  /* Fill in */


  /* Construct requestMessage, add a header line so that
   * server closes connection after one response. */

  /* Fill in */

  return;
 }


 /* Send Http request, parse response and return requested object
  * as a String (if no errors),
  * otherwise return meaningful error message.
  * Don't catch Exceptions. EmailClient will handle them. */
 public String send() throws IOException {

  /* buffer to read object in 4kB chunks */
  char[] buf = new char[BUF_SIZE];

  /* Maximum size of object is 100kB, which should be enough for most objects.
   * Change constant if you need more. */
  char[] body = new char[MAX_OBJECT_SIZE];

  String statusLine=""; // status line
  int status;  // status code
  String headers=""; // headers
  int bodyLength=-1; // lenghth of body

  String[] tmp;

  /* The socket to the server */
  Socket connection;

  /* Streams for reading from and writing to socket */
  BufferedReader fromServer;
  DataOutputStream toServer;
  System.out.println("Connecting server: " + host+CRLF);

  /* Connect to http server on port 80.
   * Assign input and output streams to connection. */
  connection = /* Fill in */;
  fromServer = /* Fill in */;
  toServer = /* Fill in */;

  System.out.println("Send request:\n" + requestMessage);


  /* Send requestMessage to http server */
  /* Fill in */

  /* Read the status line from response message */
  statusLine= /* Fill in */;
  System.out.println("Status Line:\n"+statusLine+CRLF);

  /* Extract status code from status line. If status code is not 200,
   * close connection and return an error message.
   * Do NOT throw an exception */
  /* Fill in */

  /* Read header lines from response message, convert to a string,
    * and assign to "headers" variable.
   * Recall that an empty line indicates end of headers.
   * Extract length  from "Content-Length:" (or "Content-length:")
   * header line, if present, and assign to "bodyLength" variable.
  */
  /* Fill in */   // requires about 10 lines of code
  System.out.println("Headers:\n"+headers+CRLF);


  /* If object is larger than MAX_OBJECT_SIZE, close the connection and
   * return meaningful message. */
  if (/* Fill in */) {
   /* Fill in */
   return(/* Fill in */ +bodyLength);
  }

  /* Read the body in chunks of BUF_SIZE using buf[] and copy the chunk
   * into body[]. Stop when either we have
   * read Content-Length bytes or when the connection is
   * closed (when there is no Content-Length in the response).
   * Use one of the read() methods of BufferedReader here, NOT readLine().
   * Make sure not to read more than MAX_OBJECT_SIZE characters.
   */
  int bytesRead = 0;

  /* Fill in */   // Requires 10-20 lines of code

  /* At this points body[] should hold to body of the downloaded object and
   * bytesRead should hold the number of bytes read from the BufferedReader
   */

  /* Close connection and return object as String. */
  System.out.println("Done reading file. Closing connection.");
  connection.close();
  return(new String(body, 0, bytesRead));
 }
}
```