CST 8221 – JAP - Assignment #2, Part 2

Due Date: prior or on 9 April 2020

Earnings: 7% of your total course mark

Purpose: Writing the Client and Server code
The purpose of Assignment 2 is to build a simple socket based multi-threaded client/server chat application. In Part 1 of the assignment you have built the GUI component of the application. In Part 2 of the assignment you are to write the client and the server code. The application must behave exactly as specified below and shown in the **ClientServerChatScreenCaptures_W20 .pdf.**

Requirements Specification:

You have to implement the following classes and interfaces: ***Accessible***, ***ChatProtocolConstants***, ***ConnectionWrapper***, ***ChatRunnable***, ***Client***, ***ClientChatUI***, ***Server***, ***ServerChatUI***.

Swing Implementation

Interface Accessible

The interface must contain the following methods:

```
JTextArea getDisplay()
void closeChat()
```

Class ChatProtocolConstants

The class must only contain the following static string constants:

```
CHAT_TERMINATOR="Bye";
DISPLACMENT="\t\t";
LINE_TERMINATOR="\r\n";
HANDSHAKE="hello";
```

Class ConnectionWrapper

The class must contain the following fields, constructor, and methods:

Fields
```
ObjectOutputStream outputStream
ObjectInputStream inputStream
Socket socket
```

Constructor

```
ConnectionWrapper(Socket socket)
```
The constructor initializes the *socket* field only.

Methods

```
Socket getSocket()
ObjectOutputStream getOutputStream()
ObjectInputStream getInputStream()
```

`ObjectInputStream createObjectIStreams() throws IOException`
The method instantiates an object of *ObjectInputStream* using the input stream of the socket, assigns the reference to the *inputStream* field and returns the *inputStream* reference.

`ObjectOutputStream createObjectOStreams() throws IOException`
The method instantiates an object of *ObjectOutputStream* using the output stream of the socket , assigns the reference to the *outputStream* field and returns the *outputStream* reference.

`void createStreams() throws IOException`
The method instantiates an object of *ObjectOutputStream* and assigns the reference to the *outputStream* field. Then it instantiates an object of *ObjectInputStream* and assigns the reference to the *inputStream* field.

`public void closeConnection()throws IOException`
The method closes the output stream, the input stream, and the socket. Make sure that you do not call the *close()* method on **null** references. Also make sure that you do not call *close()* on a closed socket.

Class ChatRunnable
The class ***ChatRunnable*** has the following declaration:

```
ChatRunnable <T extends JFrame & Accessible> implements Runnable
```

The class must contain the following fields, constructor, and methods:

Fields
The class must have the following *final* fields

```
final T ui
final Socket socket
final ObjectInputStream inputStream
final ObjectOutputStream outputStream
final JTextArea display
```

Constructor

`ChatRunnable (T ui, ConnectionWrapper connection)`
The constructor uses the *connection* parameter and its get methods to initialize the *socket, inputStream*, and *outputStream* fields. It uses the *ui* parameter to initialize the *display* and the *ui* fields.

Methods
`void run()`

The method declares a local variable *strin* of type *String*.
Then in an endless loop it performs the following:

1. If the *socket* is not closed it uses the *inputStream* to read an object and assign it to the *strin* variable. If the socket has been closed it breaks the loop.
2. Trim the *strin* and compare it to the CHAT_TERMINATOR string. If the trimmed *strin* equal to the `CHAT_TERMINATOR` string, a *final* string *terminate* is declared. Then *terminate* is assigned a string made of the following substrings: DISPLACMENT, current date and time, LINE_TERMINATOR, and *strin*.
   For the current date and time you must use the ***DateTimeFormmater*** class to format the date and time returned by the ***LocalDateTime*** *now()* method. The format must display the name of the month, the day of the month followed by a comma (,), the time, and the AM or PM. For example: `October 31, 13:13 PM`
   Finally, it uses the *display* field *append()* method to append the *terminate* string and then it breaks the loop.
3. If during the above operations exceptions are thrown they must be caught and the loop must be broken.
4. If the strin is not equal to the `CHAT_TERMINATOR` string, a final string *append* is declared. Then *append* is assigned a string made of the same substrings as described in item 2 above. Finally, it uses the *display* field to append the *append* string.

If the loop is broken and if the socket is not closed, the *outputStream* is used to write the following string:

```
ChatProtocolConstants.DISPLACMENT+
ChatProtocolConstants.CHAT_TERMINATOR+
ChatProtocolConstants.LINE_TERMINATOR
```

Finally, before *run()* method returns it calls the *closeChat()* method of *ui*.

Class Server

The ***Server*** class is responsible for creating a server socket and starting a server side chat GUI that will communicate with each individual client.

The ***main()*** method should perform the following tasks:

- If command line string is supplied at launch, the method converts the string to an integer port number. Otherwise, the server must use 65535 as a default port number and prints on the console that the default port is used.
- The method creates a TCP/IP server socket bound to the specified port.
- The method declares a local variable *friend* and initializes it to 0.
- In an endless loop the method calls ***accept()*** on the server socket instance. Once a connection with the client is established, the *accept()* method will return a Socket instance. Assign that instance to a local Socket variable *socket*. Use the following lines statements to set the socket.

  ```
  if(socket.getSoLinger()!= -1) socket.setSoLinger(true,5)
  if(!socket.getTcpNoDelay()) socket.setTcpNoDelay(true);
  ```

  The method prints the socket information on the console screen and increments the *friend* variable.
- Next, still in the same loop, a *final* variable *title* of type string is declared and following string is assigned to it: `"YourName's Friend "+friend`

- Finally, it call its *launchClient()* (see Assignment 2 Part 1) with the Socket instance *socket* and the *title*.

The server does not have a GUI. It must started at the command prompt. In order to stop the server the user must terminate the Java Virtual Machine (JVM) (Ctrl-C).

Class ServerChatUI
The ***ServerChatUI*** must inherit from ***JFrame*** and ***Accessible***.

Fields
The class must have at least the following fields
```
JTextField message
JButton sendButton
JTextArea display
ObjectOutputStream outputStream
Socket socket
ConnectionWrapper connection
```

Constructor

```
ServerChatUI(Socket socket)
```
See Assignment 2 Part 1.

Methods

```
JTextArea getDisplay()
```

```
closeChat()
```
The method tries to close the connection and then disposes the frame.

```
JPanel createUI()
```
See Assignment 2 Part 1.

```
runClient()
```
The method initializes the *connection* field with a new *ConnectionWrapper.* Next, it uses the *connection* to create streams and initialize the *outputStream* field.
Then it creates an object of type *Runnable* using the *ChatRunnable* constructor, creates a thread passing the runnable reference to the Thread constructor and starts the thread.

Inner Classes

```
Controller
```
See Assignment 2 Part 1.

Methods
```
actionPerformed()
```
In the *actionPerformed()* if the Send button is clicked the method call the private *send()* method of the *Controller*.

`send()`
The method gets the text from the *message* text field assigns it to a local variable *sendMessage*, appends it to the *display* adding a line terminator, and then uses the *outputStream* to write the following string object:

```
ChatProtocolConstants.DISPLACMENT
+ sendMessage
+ChatProtocolConstants.LINE_TERMINATOR
```

If some run-time errors occur during the operation of the method, it must display the errors on the chat display text area.

`WindowController`
See Assignment 2 Part 1.

Methods

`windowClosing()`
First, the method prints on the console the following message:
`ServerUI Window closing!`
Second, using the *outputStream* it tries to write the following object:

```
ChatProtocolConstants.DISPLACMENT
+ChatProtocolConstants.CHAT_TERMINATOR
+ChatProtocolConstants.LINE_TERMINATOR
```

If an exception occurs during the writing, in the **finally** clause of the **try-catch** statement it disposes the frame.
Third, the method prints on the console the following message:
`Closing Chat!`
Fourth, using the *connection* it tries to close the connection. If an exception occurs, in the **finally** clause of the **try-catch** statement it disposes the frame.
Fifth, the method disposes the frame and prints on the console the following message:
`Chat closed!`

`windowClosed()`
The method prints on the console the following message:
`Server UI Closed!`

Class Client
See Assignment 2 Part 1.

Class ClientChatUI
The ***ClientChatUI*** must inherit from ***JFrame*** and ***Accessible***.

Fields
The class must have at least the following fields

```
JTextField message
JButton sendButton
JTextArea display
ObjectOutputStream outputStream
Socket socket
ConnectionWrapper connection
```

Constructor

`ClientChatUI(String title)`
See Assignment 2 Part 1.

Methods
`JTextArea getDisplay()`

`closeChat()`
If the socket is not closed the method tries to close the connection. Then it calls *enableConnectButton()*.

`JPanel createClientUI()`
See Assignment 2 Part 1.

`runClient()`
See Assignment 2 Part 1.

`void enableConnectButton()`
The private method enables the Connect button, sets the background of the Connect button to red, disables the Send button, and request the focus to the host text field.

Inner Classes

`Controller`
See Assignment 2 Part 1.

Methods
`actionPerformed()`
First, the method declares a boolean variable *connected* and initializes it to ***false***.

Second, if the Connect button has been clicked, the method declares a variable *host* and assigns to it the string displayed currently in the *host* text field.

Third, the method declares a variable *port* of type int. It gets the selected item from the combo box, converts it to integer and assigns it to the *port* variable.
Note: If you use only one event handler that implements the ***ActionListener*** interface, you should use getSelectedItem() to get the port number from the combo box.

Fourth, it calls the method *connect()* and assigns the return value to the *connect* variable. If the *connect* variable is *true*, the method disables the Connect button, makes the background color of the Connect blue, enables the Send button, and requests the focus to the message text field. Then, it creates an object of type *Runnable* using the *ChatRunnable* constructor, creates a thread passing the runnable reference to the Thread constructor and starts the thread. If the connect variable is *false*, it method returns.

Fifth, if the Send button has been clicked, the method calls the private method send().

If some run-time errors occur during the operation of the method, it must display the errors on the chat display and return.

`boolean connect(String host, int port)`

First, the method tries to create a time-out socket. If the socket is successfully created it assigns the instance to the *socket* field and then uses the following line to set the socket:

```
if(socket.getSoLinger()!= -1) socket.setSoLinger(true,5)
if(!socket.getTcpNoDelay()) socket.setTcpNoDelay(true);
```

Next, it appends the socket information on the chat display text area.

Second, the method creates a new ***ConnectionWrapper*** and assigns the returned reference to the *connection* field. Then it uses the connection to create streams, and initializes the *outputStream* field.

Third, if the operation is successful, it returns ***true***; otherwise it returns ***false***.

If some run-time errors occur during the operation of the method, it must display the errors on the chat display.

`send()`
The method gets the text from the *message* text field assigns it to a local variable *sendMessage*, appends it to the *display* adding a line terminator, and then uses the *outputStream* to write the following string object:

```
ChatProtocolConstants.DISPLACMENT
+ sendMessage
+ChatProtocolConstants.LINE_TERMINATOR
```

If some run-time errors occur during the operation of the method, it must first call *enableConnectButton()* method and then must display the errors on the chat display.

`WindowController`
See Assignment 2 Part 1.

Methods
windowClosing()
The method using the *outputStream* tries to write the following object:
`ChatProtocolConstants.CHAT_TERMINATOR`
If exception occurs it calls *System.exit(0);* otherwise it calls *System.exit(0)*.

JavaFX Implementation

Interface Accessible
See Swing Implementation.

Class ChatProtocolConstants
See Swing Implementation.

Class ConnectionWrapper
See Swing Implementation.

Class ChatRunnable
The class ***ChatRunnable*** has the following declaration:

```
ChatRunnable<T extends Application&Accessable>implements Runnable
```
For the rest see the Swing Implementation.

Class Server
See Swing Implementation.
The following modification must be made in the Swing implementation:
Before the endless loop the following line must be included:
```
Platform.setImplicitExit(false);
```

```
launchClient(Socket in, String title)
```
See Assignment 2 Part1.

Class ServerChatUI
The ***ServerChatUI*** must inherit from ***Application*** and ***Accessible***.

Fields
The class must have at least the following fields
```
TextField message
Button sendButton
TextArea display
ObjectOutputStream outputStream
Socket socket
ConnectionWrapper connection
Stage primaryStage
String title
```

Constructor
```
ServerChatUI(Socket socket, String title)
```
See Assignment 2 Part 1.

Methods
```
TextArea getDisplay()
```

```
closeChat()
```
First the method tries close the connection.
Next, the method tries to close the *primaryStage*.
Note: You must use *Platform.runLater()* to close the stage.

`Scene createScene()`
See Assignment 2 Part 1.

`start(Stage primaryStage)`
See Assignment 2 Part 1.
Additionally, before showing the stage, the *primaryStage* must call *setOnCloseRequest()*. As a parameter you must provide a lambda expression. In the body of the lambda expression, you must first print on the console the following message: `Server UI Closed!`. Next, you must use the *outputStream* field to write the following object:

```
ChatProtocolConstants.DISPLACMENT,
+ChatProtocolConstants.CHAT_TERMINATOR
+ChatProtocolConstants.LINE_TERMINATOR);
```

After showing the stage the method call the *runClient()* method.

`runClient()`
See Swing implementation.

Inner Classes

`BorderedTitledPane`
See Assignment 2 Part 1.

`Controller`
See Assignment 2 Part 1.

Methods
`handle()`
In the *handle()* method, if the Send button is clicked the method call the private *send()* method.

`send()`
The method gets the text from the *message* text field assigns it to a local variable *sendMessage*, appends it to the *display* adding a line terminator, and then uses the *outputStream* to write the following string object:

```
ChatProtocolConstants.DISPLACMENT
+ sendMessage
+ChatProtocolConstants.LINE_TERMINATOR
```

If some run-time errors occur during the operation of the method, it must display the errors on the chat display text area.

Class Client
See Assignment 2 Part 1.

Class ClientChatUI
The ***ClientChatUI*** must inherit from ***Application*** and ***Accessible***.

Fields

The class must have at least the following fields

```
TextField message
Button sendButton
TextArea display
ObjectOutputStream outputStream
Socket socket
ConnectionWrapper connection
```

Methods

`TextArea getDisplay()`

`closeChat()`

See Swing Implementation.

`Scene createScene()`

See Assignment 2 Part 1.

`start()`

See Assignment 2 Part 1.

`stop()`

If the socket is not closed, the method tries to write the following object `ChatProtocolConstants.CHAT_TERMINATOR` to the output stream.

`void enableConnectButton()`

The private method enables the Connect button, sets the background of the Connect button to red, and disables the Send button.

Inner Classes

`Controller`

See Assignment 2 Part 1.

Methods

`handle()`

See Swing Implementation.

`send()`

See Swing Implementation.

INPORTANT NOTE:

You are allowed (but not required) to work in a team on this part of the assignment (Assignment 2 Part 2). A team can have two members only. Both members must be officially registered in **the same theory section** of the course and must have demonstrated their Assignment 2 Part 1. If you decide to work in a team, one member of the team must send me a notification e-mail with the names (first, last) and the student numbers of the team members. Without a proper and timely (at least two week before the assignment is due) notification the team work will not be accepted.

What to Submit:

Paper submission:

No paper submission is required for this assignment.

Code submission:

Compress in one **.zip** file all relevant to the assignment **.java** and **.class**. Upload the assignment **zip** file to Brightspace prior to or on the due date. The name of the zip file must have the following structure: Student's family name followed by the last three digits of the student ID number followed by _JAP_A2P2 , and finally, followed by your lab section number (for example, s301). **Teams** must submit one .zip file only. The name of the file must contain the names of both members e.g. Name345_Name123_JAP_A2P2_lab section(s).zip.

The submission must follow the course submission standards. The ***Assignment Submission Standard*** and the ***Assignment Marking Guide*** are posted on Brightspace.

Enjoy the assignment. And do not forget that:

"To have a chat you ought to have a friend first." Society Rule #1

CST8221 – JAP, 27 February 2020, S^R