Project 2 - A Simple Chat Program

Minimum Effort Due: (Part 1) 2/11/2007 (Sunday)
Complete Project Due: (Part 2) 2/23/2007 (Friday)
Late Project Due: (Last Chance) 2/24/2007 (Saturday)

Overview

In this project, you will write a program to allow a limited form of interactive text messaging like AOL™, Instant Messenger™, or ICQ™.

Objectives

1. Write a program that utilizes multiple threads.
2. Develop a simple GUI.
3. Gain further experience working with event driven programming.
4. Obtain additional experience writing programs in Java.
5. Gain more experience writing Java classes.
6. Work with a program that uses a network.

Activities

1. Read the program requirements.
2. Understand the program specifications.
3. Review the design guidelines.
4. Get the files you need to complete the project.
5. Write your code.
6. Submit your project.

Getting Help

You may get help from your instructor(s) and the teaching assistants. Anything else is not allowed and is subject to the penalties listed in the DCS Policy on Academic Dishonesty. This includes but is not limited to:

- obtaining help from any other people
- providing help to other people
- sharing source code with anyone, by any means or medium.

We certainly do not expect there to be absolutely no communication between the students of this class; we know that people tend to learn very well that way. If you have any doubt if what you would like to do might violate our rules, please see your lecture instructor for clarification.

Project 2 - Requirements

Overview

Your program will be a chat program with a graphical user interface. Users of the chat system are identified by an unique string called a *handle*. Before using the system a user must register her handle with the server.

Your program must be capable of displaying two types of windows:

1. **User List**. Displays a list of the handles of all users who are currently registered with the chat server. There only needs to be one of these. The list must be automatically updated as new users register with the server and registered users sign off.
2. **Conversation Window**. A window containing a current conversation between two users. Your program must be capable of handle multiple conversations at the same time. Each conversation will be handled by a separate window.

Your program must be capable of performing the following actions:

1. When the program is started, the User List window should immediately appear.
2. There must be a *simple* way to start a conversation with another user (or with yourself). Presumably this would be from the User List, but that is up to you.
3. Your program should instantly notify the user when a remote user attempts to start a chat session. You could do this by popping up a window, or changing the user list in someway to indicate which user is calling. The local user should have the option of either accepting the invitation, denying the invitation, or ignoring it all together.
4. After either #2 or #3 (accept) occurs, a window must appear for the local user to enter text and view messages sent by both users. Again, the exact format of this window, and how often the text is updated, is up to you.
5. It must be possible to terminate a conversation from the conversation window. Again you are free to choose how your program will accomplish this, but it could be as simple as closing the conversation window. Note that either user may terminate the connection at any time, and that the program should inform the local user when the remote user terminates the connection.
6. There must be a way to terminate the entire program. This would include terminating all active conversations (gracefully) and closing the window that contains the user list.

Note that your program must be capable of displaying and updating the user list at the same time it is accepting incoming chat requests, and initiating outgoing chat requests.

Project 2 - Specification

Overview

The chat program should be started by using the following command:

```
java CSChat myhandle
```

where *myhandle* is chosen by the user. A handle is any valid Java string. Note that if you choose to use spaces or special characters in your handle you may have to enclose it in quotes on the command line. If the handle is already registered by another chat program user, the following error message will be printed on standard error and the program will gracefully terminate without generating any additional output:

```
myhandle is already in use
```

where *myhandle* is the handle the user specified on the command line.

Note: the handle that you specify will be seen by all students taking CS2. Do not use handle names that would be considered unprofessional or offensive. Failure to follow this rule may result in a failing grade for this project or in the course.

If the program is started without the correct arguments, the following error message will be printed on standard error and the program will gracefully terminate without generating any additional output:

```
Usage: java CSChat myhandle
```

Design Constraints

You are not allowed to use a GUI building tool like Forte, JBuilder or VisualJ++ for this project. The Swing code must be manually written.

Project 2 - Design

Overview

There are three parts to the chat system: a **name service**, a **chat server**, and a **chat application**.

- The **name service** runs on a dedicated server machine (we will be using *holly*) and keeps track of registered clients.
- Users of the system do not communicate directly with the name service but use a client interface (the **network chat server**), that runs as part of their application. This interface allows clients to discover other registered clients and communicate with them.
- The **chat application** is a graphical user interface that allows users to see who else is registered with the name service and to establish chat connections with them.

The Name Service

The name service runs continuously on a server machine and allows users to register with it and find out others who have registered with it. The name service keeps an up-to-date collection of registered "handles" and broadcasts this collection at regular intervals to interested listeners. A chat server object that wishes to be registered with the name service must obtain a "ticket" from the name service. In order to keep this ticket valid, the network chat server must continually renew this ticket. If it fails to renew the ticket, the name service will drop its registration.

The name service will be running continuously on *holly* and you can count on it being running for your testing. The protocol used by the name service is private. The only access to the name service is through the `ChatServer` interface.

The name service maintains a log that describes what names have been registered and what names have been dropped. You can view the log by clicking here. Note that this file may get to be very big as more and more people work on their project.

Chat Server

The chat server provides the functionality required to communicate with the name service, to initiate chat sessions, and to be notified of incoming chat requests. The `ChatServer` interface defines the behavior provided by a chat server. A chat server will use the name service to register a handle, and to be notified when the list of registered users changes. Since everyone is sharing the same name service, applications will be able to see and communicate with applications written by different people on different machines.

Applications can register a listener with the chat server to receive notifications of registered users and connection attempts from other applications. The chat server also provides mechanisms for initiating and accepting connections between chat applications. All network streams are managed by the chat server.

The Chat Application

The chat application is an application program with a graphical user interface that allows users to register with the name service, discover who else is registered, propose connections with other users, accept connections from other users, and chat with other users through connections that have been both proposed and accepted.

Part of the graphical user interface is a continuously updated display of the handles of the other registered users. Another part of the graphical user interface allows the user to discover proposed connections and accept them if desired. Also, means are provided so that connections can be proposed to any registered user at any time.

Multiple conversations may be established and used simultaneously through the user interface and connections can be closed at any time under control of the user.

Chat Bot System

Adam Strong has written several "bots" that you can use to test the functionality of your client. There is a bot called "EchoBot" which will echo everything you send to it, back to you. There is also a "ChatBot" which can take interactive commands from you that can initiate a chat session with your client, disconnect, etc. The information you send to these bots can be retrieved by going to the directory of your handle underneath here. Keep in mind that everything you send to the bots can be publicly viewed by everyone.

Program Structure

The Name Service

Although the name service code is being provided to you (i.e., you do not have to write it), it is important that you understand its role in this system in order to build your program. The name service runs on *holly* and the actual server and port of this server are defined in the variables `NetworkChatServer.DEFAULT_NS_HOST` and `NetworkChatServer.DEFAULT_NS_PORT`. The name service maintains a collection of registered handles and manages the renewal process so that handles that are not periodically renewed are dropped from the collection. It also broadcasts to all interested registered users the contents of this collection at periodic intervals. It also accepts handle registrations from remote clients and adds them to the collection.

The protocol between the name service and its clients is private and is implemented by an object that implements the `ChatServer` interface. You must use this interface to access the name service.

The `NetworkChatServer`

The `NetworkChatServer` class implements the `ChatServer` interface and provides a means for clients to access the name service and to establish connections with other chat applications. This class will handle registration and the automatic re-registration of your handle. As the class name implies, this class uses a network connection to establish chat sessions This code has been written for you and is being provided for you as part of this project.

Calling the constructor for the `NetworkChatServer` starts the `NetworkChatServer` running. If the

`NetworkChatServer` is unable to "find" the machine on which the name service is running (by default, *holly.cs.rit.edu*), then it will throw a `java.net.UnknownHostException`, which you should be prepared to deal with. (This may happen if you are running your program behind certain types of firewalls at home/work, or if your/holly's connection to the Internet is down.)

Once the `NetworkChatServer` is started and enabled, it will deliver regular events if you register a listener with it. A `ChatListener` receives notifications of the currently registered handles and notifications of chat requests directed at your handle (if you have registered one). You must enable the `NetworkChatServer` to receive user notifications and chat requests.

The handle that you register with the name service will be published to all registered clients of the name service. A handle is a Java string which must be unique. In other words no other application can register the same handle that you are using, while you are actively using it. If it is not unique the server will refuse to register the handle.

Once you have registered a handle with the `NetworkChatServer` you may attempt to chat with another registered user (even yourself) or accept a connection from another user. No communication is possible until a connection is proposed and accepted. It is possible to set up several distinct connections with the same user.

The ChatSession Object

Once a connection is both proposed and accepted, both ends of the connection will obtain a reference to a `ChatSession` object whose state describes the connection. Specifically, the `ChatSession` object will maintain a reference to a `Reader` and a `Writer` that can be used to receive and send character-based data across the chat connection. Anything written to the `Writer` on one end of a connection can be read with the `Reader` at the other end of the connection. It is the responsibility of the chat application to appropriately read and write these streams so that meaningful communication can be accomplished. All chat applications should be capable of dealing with plain text and exchanging data with other chat applications on a line-by-line basis (i.e., reading/writing a full line of text at a time), although other formats might be supported.

Closing the ChatSession object closes both the `Reader` and `Writer` associated with it. If one end of the connection is closed, any attempts to use the other end will result in an `IOException` being thrown.

Use of the Server

Listed below are some typical interactions that a chat application program might have with the server.

Register with the server

1. Create a new NetworkChatServer.
2. Add a `ChatListener` to receive notifications of registered users and connection attempts.
3. Enable the server to enable connection attempts.
4. Register a handle with the server for others to see.

Initiating a connection

1. Invoke `chat(handle)` to propose a connection with user *handle*.

2. Use the `ChatSession` object returned in step 1 to obtain a a `Reader` and a `Writer` for communicating with the other user.
3. When done, or when the `Reader` or `Writer` is closed from the other end, close the session object.

Accepting a connection

1. When a notification of a connection attempt is received by the `ChatListener`, you will receive a `ChatSession` object.
2. The `ChatListener` will need to invoke the `accept()` method on the `ChatSession` object in order to signal to the `ChatServer` that the connection request has been accepted by the user (if appropriate).
3. The `ChatSession` object may then be used to obtain a `Reader` and `Writer` for communicating with the other user.
4. When done, or when the `Reader` or `Writer` is closed from the other end, close the session object.

The Chat Application

The chat application you will write will be started from the command line with a single "handle" argument. Unless there are errors, the application will bring up a graphical user interface and all further interaction with the application will be through the GUI. No messages will be printed to `System.out` and no data will be read from `System.in`. The GUI will continuously display the currently registered handles. The user will have the ability to initiate connections with other users and accept or reject connections proposed by other users. If a connection is made with another user, then the application will provide a means of interacting with this other user. Multiple connections with other users should be possible simultaneously. There should also be the ability to close any open connection without affecting other connections, as well as a means of shutting down the entire application.

The user receiving a request to chat should be given the opportunity to either:

1. **Accept** the incoming request, in which case a `ChatListener` must invoke the `accept` method on the `ChatSession` object provided by the server, and then provide the GUI to support an interactive chat.
2. **Deny** the request, in which case a `ChatListener` should explicitly close the channel to the other user by invoking the `close` method on the `ChatSession` object. (Note: you may want to accept the chat for the sole purpose of sending a message to the other user that says something like, "No, I really don't want to talk to you right now" on a separate thread before closing the session.)
3. **Ignore** the request, in which case none of your `ChatListener` objects should invoke the `accept` and `close` methods on the `ChatSession` instance that they receive from the server.

Since many things are happening simultaneously and some things might take a long time, the application should start separate threads for any operation that might be delayed. In particular, you should not do opens, reads, or writes in a listener thread (either a window listener or a `ChatListener`) but rather create a new thread to do the work.

Project 2 - Getting the Files You Need

The files that you need to complete this project are contained in the `jar` file located here. Download the `jar` file and unpack it in an appropriate location in your account.

Do not change the source code provided to you in the `jar` file, or else your system will not work when we test it.

After you have unpacked the jar file you should see that a directory named `project2` has been created. This directory contains an RCS directory, a directory named `NameService` that contains the class files you will need to use the name server, and class files for all of the classes that you will need to complete this project.

If you are using eclipse, download the zip file located here and store it on your local machine. To add the classes to your project workspace, right click on the workspace and select `Build Path->Add External Archives` and select the downloaded file.

The RCS directory contains the source code for the following classes:

- ChatException
- ChatListener
- ChatAdapter
- ChatServer
- ChatSession
- NetworkChatServer
- NetworkChatSession

This code is complete and you do not need to change it in any way. These files are being provided to you so that you can look at the source and see how these classes are implemented. You can also use these source files to recompile any of the class files you were given in case you should delete one by accident.

You should not need to do anything with the `NameService` directory or its contents. Just make sure that when you compile your program the `NameService` directory is on your class path. If you do your compiles within the `project2` directory that was created when you unpacked the `jar` you should be all set.

Project 2 - Writing Your Code

DO NOT write any code until you understand the program requirements, and have written your specification and design documents for this program. Writing code first will simply be a waste of your time, and will result in a program that does not work!!

You will probably want to start this project by building a prototype of your GUI. The prototype will not have the desired functionality, but should contain all of the various components in the final GUI. You can use the prototype as a framework on which the rest of the program can be built. Building the prototype will also help you to define the look and feel of your program. Your program at this point will meet the minimum effort requirements.

After completing the GUI prototype start working on writing the code that will start your program and display the user window. Then start working with the `ChatServer` interface to register with the server and to update the user list.

Start working on the rest of the parts of this program that you feel are the easiest ones, and work your way to the hardest ones. As you implement the individual pieces of your program be sure that the program compiles without errors. You should also test each piece of the program as it is completed. This may require that you create special test files or add code for testing the intermediate functionality.

Develop your code in small pieces, and test each part of your code as you write it. It you wait until you have all the code written to start testing, it will be almost impossible to locate and correct errors in your program.

Part of your grade in this project will depend on the functionality of the GUI and its ease of use. When your lecture instructor runs your program, they will ask the question, "How simple is it for me to figure out the program and use it?"

We have kept the requirements and the specifications for this program to a minimum in order to leave you the freedom to implement new features that you think may be appropriate. Keep in mind that even though you want to write a very fancy chat client, your program must work in order to get a good grade. Write a program that meets the basic requirements first and **submit it!**. Then if you have time, you may add the additional functionality while making sure to test your program *before* submitting the new version.

As always, part of your grade will also be based on how you use RCS, so make sure that you check in your code whenever you have completed substantial changes to it.

Project 2 - Submission and grading

Minimum Submission

In order to satisfy the minimal reasonable effort requirements for this project (see the CS2 syllabus for details), you must submit a program which contains the following:

- Your program must register the name specified on the command line with the network server. If the name is already in use, you should display an error message and not start up the GUI.
- You must have a runnable GUI prototype for the project. Minimally it should display two windows - one for the list of users on the server, and one for a chat window.
- The main window should display the name of your handle prominently.
- The user list does not have to receive the real users from the server, but it should be populated with some dummy data.
- The chat window does not have to function properly, or display real information. You can use dummy information, but it should contain the following: the name of the person being chatted with, an area to type msgs and subsequently send them, an area to receive messages.
- The send area of your chat window should be editable, and the receive area should not be modifiable.

If you have questions about what is an acceptable minimal submission, please speak with your instructor before the deadline.

This submission must be made by midnight on the Minimum Effort due date (see first page). This portion of the project is worth a maximum of 20 points.

You will submit code for this project using the following **try** command:

```
try 232-grd project2-min CSChat.java *.java *.gif *.jpg *.wav
```

where `*.java *.gif *.jpg *.wav` are any **additional** files used in your program. (Your program may not need any extra files, but your 'main' file must be CSChat.java.)

If you do not submit a working program by minimal effort due date, you will automatically receive a grade of "0" for the project.

Project Submission

You will submit code for the full version of this project using the following **try** command:

```
try 232-grd project2 README CSChat.java *.java *.gif *.jpg *.wav
```

where `*.java *.gif *.jpg *.wav` are any additional files used in your program. (Your program may not need any extra files.)

The file `README` is a text file that will be used by your instructor when your project is graded. It should

include a user guide on how to run your program and differences between the program you submitted for the minimum submission and the implementation you are now submitting. The content and format of this file will be provided to you by your lecture instructor.

If you submit a fully-functional version of the project by midnight on the Final Project due date listed on the first page, your code will be eligible to receive full credit for this portion of the project. For this portion of the project, we will also allow for late submission. The Late Final Project date is also shown on the first page. Submissions received during that late period will lower your final grade by 10 points.

When you feel that you have reached an important milestone in the program's development, *submit* it. This way if you fail to get any further before the due date at least you will have submitted something.

Remember to submit early, and often. The labs get *very* busy the night a project is due! Although you may start to run `try` before midnight, it may not finish before midnight. The best strategy is to finish before the labs get crazy, then you can sit back and watch the others panic.

Project Grade

The project grade will be computed as follows:

1. You cannot get a high grade for a program that does not run. Therefore, your grade is first computed *solely* on the functionality and correctness of your program. You can get up to 70 points for this. After this preliminary grade is computed, ...
2. You will receive up to 10 points for the quality of your graphical user interface.
3. You can lose up to 20 points if your algorithms and/or implementation are not clear or of low quality, or if any other issues are identified in your code (e.g., race conditions in threading code, etc.), an additional penalty may be assessed.
4. You can lose up to 20 points for violating the programming style standards. This includes incorrect use of RCS.
5. You can lose up to 10 points for submitting an incomplete or inaccurate README file, as specified by your lecture instructor.
6. Any "cap" on your maximum grade due to late submissions will be applied.

Note that a program that produces wrong answers will get a low grade no matter how good its algorithms are or how well it adheres to the style standards.

Class ChatException

java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ **ChatException**

All Implemented Interfaces:
Serializable

```
public class ChatException
extends Exception
```

Used to record information about an exception within the chat system.

See Also:
Serialized Form

Constructor Summary

ChatException() Create a new exception without a message.
ChatException(String message) Create a new ChatException with the specified message.
ChatException(String message, Throwable cause) Create a new chat exception with the specified message and the specified cause.

Method Summary

Methods inherited from class java.lang.Throwable
fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

ChatException

```
public ChatException()
```

Create a new exception without a message.

ChatException

```
public ChatException(String message)
```

Create a new ChatException with the specified message.

Parameters:

`message` - the message to associate with the exception.

ChatException

```
public ChatException(String message,
                     Throwable cause)
```

Create a new chat exception with the specified message and the specified cause.

Parameters:

`message` - the message for this exception.
`cause` - the cause of this exception.

See Also:

`Exception.Exception( String, Throwable )`

Interface ChatListener

All Known Implementing Classes:

ChatAdapter

```
public interface ChatListener
```

A listener for the chat server. Defines two methods. One method is used to pass periodic user registration information to the client. The second is used to notify the client when a chat request has been received.

Method Summary

void	**chatRequest**(ChatSession session) Invoked by the server when a chat request is received.
void	**registeredUsers**(Set<String> users) Invoked by the server when it receives registration information from the name server.

Method Detail

registeredUsers

```
void registeredUsers(Set<String> users)
```

Invoked by the server when it receives registration information from the name server.

Parameters:

`users` - a set containing the names of the registered users (stored as Java `String` values).

---

chatRequest

```
void chatRequest(ChatSession session)
```

Invoked by the server when a chat request is received. At least one listener must invoke the `accept` method on the `session` object, or else the request is considered to have been refused.

The server will wait until all listeners have finished handling this event before determining if the request has been refused. If a listener blocks (e.g., pending a user's response), then the server object will wait until it is done blocking and finishes handling the event before proceeding to the next listener (or deciding if the request has been refused).

Parameters:

`session` - the parameters for the requested chat session.

See Also:

`ChatSession.accept()`

Class ChatAdapter

java.lang.Object
  └ **ChatAdapter**

All Implemented Interfaces:
ChatListener

```
public class ChatAdapter
extends Object
implements ChatListener
```

An adapter class, used to simplify the task of defining listeners for the chat server. Defines trivial (empty) versions of the two methods from the ChatListener interface.

Constructor Summary

ChatAdapter()

Method Summary

void	**chatRequest**(ChatSession session) Invoked by the server when a chat request is received.
void	**registeredUsers**(Set<String> users) Invoked by the server when it receives registration information from the name server.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ChatAdapter

```
public ChatAdapter()
```

Method Detail

registeredUsers

```
public void registeredUsers(Set<String> users)
```

Invoked by the server when it receives registration information from the name server.

Specified by:
registeredUsers in interface ChatListener
Parameters:
users - a set containing the names of the registered users.

---

chatRequest

```
public void chatRequest(ChatSession session)
```

Invoked by the server when a chat request is received.

Specified by:
chatRequest in interface ChatListener
Parameters:
session - the parameters for the requested chat session.
See Also:
ChatSession.accept()

Interface ChatServer

All Known Implementing Classes:

NetworkChatServer

```
public interface ChatServer
```

This interface defines the behavior provided by a chat server.

Method Summary

void	**addChatListener**(ChatListener client) Add a listener to the set of listeners for this object.
ChatSession	**chat**(String handle) Initiate a chat session with a registered user.
boolean	**isEnabled**() Determine whether or not the server is accepting incoming chat requests.
void	**register**(String handle) Register a new user with the chat server.
void	**removeChatListener**(ChatListener client) Remove a listener from the set of registered listeners for this object.
void	**setEnabled**(boolean enable) Either enable or disable the server's ability to accept incoming chat requests.

Method Detail

setEnabled

```
void setEnabled(boolean enable)
```

Either enable or disable the server's ability to accept incoming chat requests.

Parameters:

`enable` - if true the server will accept incoming requests.

---

isEnabled

```
boolean isEnabled()
```

Determine whether or not the server is accepting incoming chat requests.

Returns:
true if the server is accepting requests and false otherwise.

---

register

```
void register(String handle)
              throws ChatException
```

Register a new user with the chat server. The server will not accept requests until a user has registered. Only one handle can be registered.

Parameters:
`handle` - the handle by which this user will be identified.
Throws:
`ChatException` - if the handle is not unique or the user cannot be registered.

---

chat

```
ChatSession chat(String handle)
                 throws ChatException
```

Initiate a chat session with a registered user.

Note: specifying the user's own handle is supported, and will result in a chat request being sent to this application's registered listeners. (This will allow the user to chat with themselves, if the client application supports it.)

Parameters:
`handle` - the handle of the user to chat with.
Returns:
an object that contains information about the new chat session, or null if none of the other user's listeners accepted the request.
Throws:
`ChatException` - if the handle is invalid, or if an internal error occurs.
See Also:
`addChatListener(ChatListener)`, `ChatSession.accept()`

---

addChatListener

`void` **`addChatListener`**`(ChatListener client)`

Add a listener to the set of listeners for this object. Listeners will receive periodic updates of the registered user list and will be notified when a chat request is received.

Parameters:

`client` - the listener to add.

---

removeChatListener

`void` **`removeChatListener`**`(ChatListener client)`

Remove a listener from the set of registered listeners for this object. Equality is determined on a shallow (i.e., by reference) basis.

Parameters:

`client` - the listener to remove.

Interface ChatSession

All Known Implementing Classes:

NetworkChatSession

```
public interface ChatSession
```

This interface defines what information needs to be maintained regarding a chat session.

Method Summary

void	**accept**() Invoked by a ChatListener during execution of chatRequest() to indicate that the request will be handled by the listener.
void	**close**() Close any resources associated with this session.
Reader	**getReader**() Return a reader that can be used to receive text data from the user at the other end of the connection.
Date	**getRequestTime**() Return the time that the request for this chat session was received.
Writer	**getWriter**() Return a writer that can be used to transmit text data to the user at the other end of the connection.
String	**localHandle**() Return the handle of the local user.
String	**remoteHandle**() Return the handle of the remote user.

Method Detail

localHandle

```
String localHandle()
```

Return the handle of the local user.

Returns:
the local handle.

---

remoteHandle

`String` **`remoteHandle`**`()`

Return the handle of the remote user.

Returns:
the remote handle.

---

getRequestTime

`Date` **`getRequestTime`**`()`

Return the time that the request for this chat session was received.

Returns:
the time the request for this session was received.

---

getWriter

`Writer` **`getWriter`**`()`

Return a writer that can be used to transmit text data to the user at the other end of the connection.

Returns:
a stream used to send information to the user.

---

getReader

`Reader` **`getReader`**`()`

Return a reader that can be used to receive text data from the user at the other end of the connection.

Returns:
a stream used to receive information from the user.

---

accept

```
void accept()
            throws ChatException
```

Invoked by a ChatListener during execution of chatRequest() to indicate that the request will be handled by the listener.

If this method is not invoked by the receiver, then the ChatServer providing the session will assume that the "request to chat" has been declined, and the user at the other end of the connection will be appropriately notified.

Throws:

ChatException

---

close

```
void close()
```

Close any resources associated with this session.

Class NetworkChatServer

java.lang.Object
  └ **NetworkChatServer**

All Implemented Interfaces:
ChatServer, Observer

```
public class NetworkChatServer
extends Object
implements ChatServer, Observer
```

A network implementation of a chat server. Every server has a port where it will accept incoming connections. Actual chat sessions are carried out on separate connections. This server uses a NameService to record handles and port values for registered servers.

Field Summary

static int	**CONNECT_TIMEOUT**
static String	**DEFAULT_NS_HOST**
static int	**DEFAULT_NS_PORT**

Constructor Summary

NetworkChatServer() Create a new chat server that uses the default host and port for the name service.
NetworkChatServer(String host, int port) Create a new chat server that uses the specified host and port for the name service.

Method Summary

void	**addChatListener**(ChatListener client) Add a listener to the set of listeners for this object.
ChatSession	**chat**(String handle) Initiate a chat session with a registered user.

boolean	**isEnabled**() Determine whether or not the server is accepting incoming chat requests.
void	**register**(String handle) Register a new user with the chat server.
void	**removeChatListener**(ChatListener client) Remove a listener from the set of registered listeners for this object.
void	**setEnabled**(boolean enable) Either enable or disable the server's ability to accept incoming chat requests.
void	**update**(Observable o, Object data) This method will be invoked everytime that an update packet is received by the ticket manager.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

DEFAULT_NS_HOST

```
public static final String DEFAULT_NS_HOST
```

See Also:
Constant Field Values

DEFAULT_NS_PORT

```
public static final int DEFAULT_NS_PORT
```

See Also:
Constant Field Values

CONNECT_TIMEOUT

```
public static final int CONNECT_TIMEOUT
```

See Also:
Constant Field Values

Constructor Detail

NetworkChatServer

```
public NetworkChatServer()
                   throws IOException
```

Create a new chat server that uses the default host and port for the name service.

Throws:

`IOException` - if an IO error occurs while communicating with the name service.

NetworkChatServer

```
public NetworkChatServer(String host,
                         int port)
                   throws UnknownHostException,
                          IOException
```

Create a new chat server that uses the specified host and port for the name service.

Parameters:

`host` - name/address of the name server being targeted
`port` - port number on the name server to be connected to

Throws:

`UnknownHostException` - if the host name is invalid.
`IOException` - if an IO error occurs while communicating with the name service.

Method Detail

setEnabled

```
public void setEnabled(boolean enable)
```

Either enable or disable the server's ability to accept incoming chat requests.

Specified by:

`setEnabled` in interface `ChatServer`

Parameters:

`enable` - if true the server will accept incoming requests.

---

isEnabled

```
public boolean isEnabled()
```

Determine whether or not the server is accepting incoming chat requests.

Specified by:
isEnabled in interface ChatServer

Returns:
true if the server is accepting requests and false otherwise.

---

register

```
public void register(String handle)
              throws ChatException
```

Register a new user with the chat server.

Specified by:
register in interface ChatServer

Parameters:
handle - the handle by which this user will be identified.

Throws:
ChatException - if the handle is not unique or the user cannot be registered.

---

chat

```
public ChatSession chat(String handle)
                 throws ChatException
```

Initiate a chat session with a registered user.

Specified by:
chat in interface ChatServer

Parameters:
handle - the handle of the user to chat with.

Returns:
an object that contains information about the new chat session, or null if the other user did not accept the request.

Throws:
ChatException - if the handle is invalid, or if an internal error occurs.

See Also:
ChatServer.addChatListener(ChatListener), ChatSession.accept()

---

addChatListener

```
public void addChatListener(ChatListener client)
```

Add a listener to the set of listeners for this object. Listeners will receive periodic updates of the registered user list and will be notified when a chat request is received.

Specified by:

addChatListener in interface ChatServer

Parameters:

client - the listener to add.

---

removeChatListener

```
public void removeChatListener(ChatListener client)
```

Remove a listener from the set of registered listeners for this object. Equality is determined on a shallow (i.e., by reference) basis.

Specified by:

removeChatListener in interface ChatServer

Parameters:

client - the listener to remove.

---

update

```
public void update(Observable o,
                   Object data)
```

This method will be invoked everytime that an update packet is received by the ticket manager. It will notify registered listeners that the update has been received. Notification is done in a separate thread

Specified by:

update in interface Observer

Class NetworkChatSession

java.lang.Object
  └ **NetworkChatSession**

All Implemented Interfaces:
ChatSession

```
public class NetworkChatSession
extends Object
implements ChatSession
```

This class holds information regarding a chat session.

Nested Class Summary

class	**NetworkChatSession.ChatRequestDeniedException** Defines a custom exception type used when an attempt to connect to a remote user fails because the remote user does not accept the request.

Constructor Summary

NetworkChatSession(String handle, Socket sock) Establish a chat session with the user on the other end of the specified socket.
NetworkChatSession(String handle, String host, int port) Actively establish a new chat session with the user on the specified host.

Method Summary

void	**accept**() Invoked by a ChatListener during execution of chatRequest() to indicate that the request will be handled by the listener.
void	**close**() Close this session.
Reader	**getReader**() Return a reader that can be used to receive text data from the user at the other end of the connection.
Date	**getRequestTime**()

	Return the time that the request for this chat session was received.
Writer	**getWriter**() Return a writer that can be used to transmit text data to the user at the other end of the connection.
String	**localHandle**() Return the handle of the local user.
String	**remoteHandle**() Return the handle of the remote user.
String	**toString**() Return a string representation of this chat session.
boolean	**wasAccepted**() Invoked by NetworkChatServer after allowing all listeners a chance to accept a chat request.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

NetworkChatSession

```
public NetworkChatSession(String handle,
                          String host,
                          int port)
                   throws UnknownHostException,
                          IOException,
                          NetworkChatSession.ChatRequestDeniedException
```

Actively establish a new chat session with the user on the specified host.

Parameters:

`handle` - the handle of the user establishing the connection.
`host` - the remote host for the chat session
`port` - the port where the remote server is accepting requests.

Throws:

`UnknownHostException` - if the host name is invalid.
`IOException` - if an error occurs during setup.
`NetworkChatSession.ChatRequestDeniedException` - if the remote user did not accept the chat request.

NetworkChatSession

```
public NetworkChatSession(String handle,
```

```
                Socket sock)
         throws IOException
```

Establish a chat session with the user on the other end of the specified socket.

Parameters:

`handle` - the handle of the user creating this session.
`sock` - the socket used to communicate with the user.

Throws:

IOException

Method Detail

accept

```
public void accept()
            throws ChatException
```

Invoked by a ChatListener during execution of chatRequest() to indicate that the request will be handled by the listener. If it is not invoked, then the ChatServer managing the chat protocol will assume that the "request to chat" has been declined, and the user at the other end of the connection will be appropriately notified.

Specified by:

accept in interface ChatSession

Throws:

ChatException

---

wasAccepted

```
public boolean wasAccepted()
```

Invoked by NetworkChatServer after allowing all listeners a chance to accept a chat request. If it returns false, the ChatServer will simply close() the session, which will cause an I/O exception on the initiating side, which will then be translated into a ChatException (thus fulfilling the requirement for such an exception when the request isn't accepted).

Note that this is **not** a method from ChatSession, but is simply a utility function of the NetworkChatSession type.

---

localHandle

```
public String localHandle()
```

Return the handle of the local user.

Specified by:
localHandle in interface ChatSession
Returns:
the local handle.

---

remoteHandle

```
public String remoteHandle()
```

Return the handle of the remote user.

Specified by:
remoteHandle in interface ChatSession
Returns:
the remote handle.

---

getRequestTime

```
public Date getRequestTime()
```

Return the time that the request for this chat session was received.

Specified by:
getRequestTime in interface ChatSession
Returns:
the time the request for this session was received.

---

getWriter

```
public Writer getWriter()
```

Return a writer that can be used to transmit text data to the user at the other end of the connection.

Specified by:
getWriter in interface ChatSession
Returns:
a stream used to send information to the user.

---

getReader

```
public Reader getReader()
```

Return a reader that can be used to receive text data from the user at the other end of the connection.

Specified by:
`getReader` in interface `ChatSession`

Returns:
a stream used to receive information from the user.

---

close

```
public void close()
```

Close this session.

Specified by:
`close` in interface `ChatSession`

---

toString

```
public String toString()
```

Return a string representation of this chat session.

Overrides:
`toString` in class `Object`

Returns:
a string representation of this chat session.
