9 Networking

What's in This Set of Notes ?

One of the more advanced topics in programming is that of networking and client/server communications. This topic considers multiple applications communicating with one another over a network (such as the internet). We will find out here that JAVA has some nice communication packages that allow us to build programs that communicate with one another in a fairly simply manner.

Here are the individual topics found in this set of notes (click on one to go there):

- 9.1 Networking Basics
- 9.2 URLs
- 9.3 Client/Server Communications
- 9.4 Client/Server Example
- 9.5 Datagram Sockets
- 9.6 Auction Example

9.1 Networking Basics

Networking allows you to create multiple JAVA applications and have them communicate with one another. So, we can set up what is known as ***distributed*** applications in which there are client/server relationships. A ***server*** is an application that provides a "service" to various ***clients*** who request the service. There are many client/server scenarios in real life:

- Bank tellers (server) provides a service for the account owners (client)
- Waitresses (server) provides a service for customers (client)
- Travel agents (server) provide a service for people wishing to go on vacation (client)

In some cases, servers themselves may become clients at some times.

- For example, the travel agent will become a client when they phone the airline to make a reservation or contact a hotel to book a room.

In the general networking scenario, everybody can either be a client or a server at any time. This is known as ***peer-to-peer*** computing. In terms of writing java applications it is similar to having many applications communicating among one another.

- For example, Napster worked this way. Thousands of people all act as clients (trying to download songs from another person) as well as servers (in that they allow others to download their songs).

There are many different strategies for allowing communication between applications. JAVA technology allows:

- internet clients to connect to servlets or back-end business systems (or databases).
- applications to connect to one another using sockets.
- applications to connect to one another using RMI.
- some others

We will look at the simplest strategy of connecting applications using sockets.

A ***Protocol*** is:

- a standard pattern of exchanging information.
- like rules/steps for communication

The simplest example of a protocol is a phone conversation:
JIM dials a phone number
MARY says "Hello..."
JIM says "Hello..."
The conversation ensues ...
JIM says "Goodbye"
MARY says "Goodbye"
Perhaps another person gets involved:
JIM dials a phone number
MARY says "Hello..."
JIM says "Hello" and perhaps asks to speak to **FRED**
MARY says "Just a minute"
FRED says "Hello..."
JIM says "Hello..."
The conversation ensues ...
JIM says "Goodbye"
FRED says "Goodbye"

Either way, there is an "expected" set of steps or responses involved during the initiation and conclusion of the conversation. If these steps are not followed, confusion occurs (like when you phone someone and they pick up the phone but do not say anything).

Computer protocols are similar in that a certain amount of "handshaking" goes on to establish a valid connection between two machines. Just as we know that there are different ways to shake hands, there are also different protocols. There are actually layered levels of protocols in that some low level layers deal with how to transfer the data bits, others deal with more higher-level issues such as "where to send the data to".

Computers running on the Internet typically use one of the following high-level **Application Layer** protocols to allow applications to communicate:

- **HyperText Transfer Protocol** (HTTP)
- **File Transfer Protocol** (FTP)
- **Telnet**

This is analogous to having multiple strategies for communicating with someone (in person, by phone, through electronic means, by post office mail etc...).

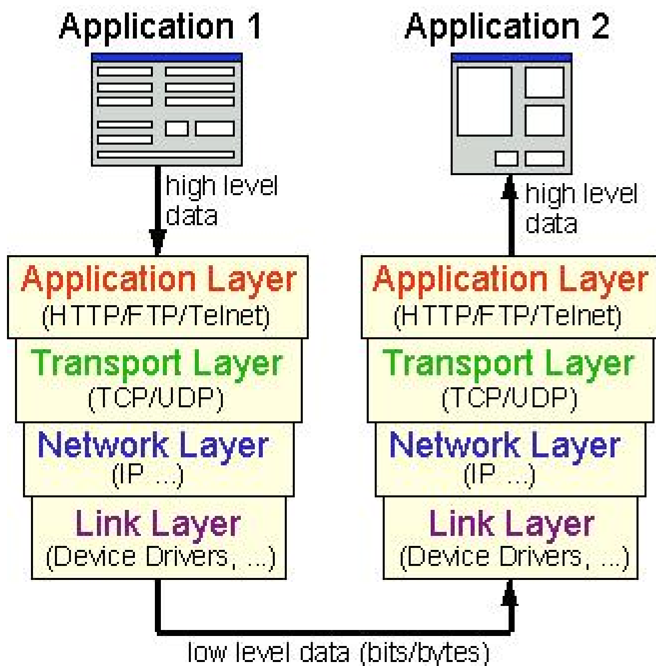
In a lower **Transport Layer** of communication, there is a separate protocol which is used to determine how the data is to be transported from one machine to another:

- **Transport Control Protocol** (TCP)
- **User Datagram Protocol** (UDP)

This is analogous to having multiple ways of actually delivering a package to someone (Email, Fax, UPS, Fed-Ex etc...)

Beneath that layer is a **Network Layer** for determining how to locate destinations for the data (i.e., address). And at the lowest level (for computers) there is a **Link Layer** which actually handles the transferring of bits/bytes.

So, internet communication is built of several layers:

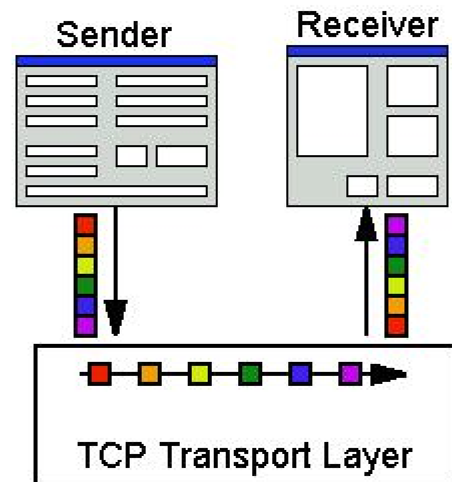


When you write JAVA applications that communicate over a network, you are programming in the Application Layer.

JAVA allows two types of communication via two main types of **Transport Layer** protocols:

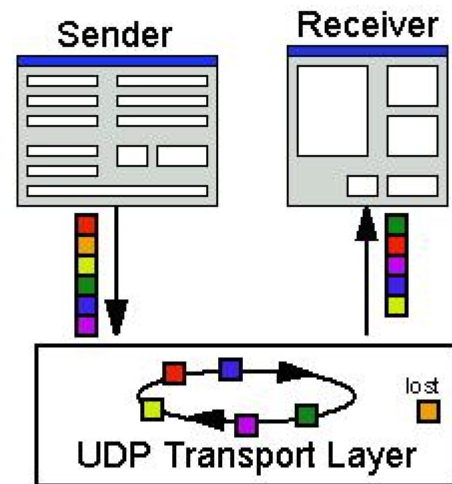
TCP

- a ***connection-based*** protocol that provides a reliable flow of data between two computers.
- guarantees that data sent from one end of the connection actually gets to the other end and in the same order
  - similar to a phone call. Your words come out in the order that you say them.
- provides a point-to-point channel for applications that require **reliable communications**.
- **slow overhead time** of setting up an end-to-end connection.



UDP

- a protocol that sends independent packets of data, called ***datagrams***, from one computer to another.
- no guarantees about arrival. UDP is not connection-based like TCP.
- provides **communication that is not guaranteed** between the two ends
    - sending packets is like sending a letter through the postal service
    - the order of delivery is not important and not guaranteed
    - each message is independent of any other.
- **faster** since no overhead of setting up end-to-end connection
- many firewalls and routers have been configured NOT TO allow UDP packets.



Why would anyone want to use UDP protocol if information may get lost ? Well, why do we use email or the post office ? We are never guaranteed that our mail will make it to the person that we send them to, yet we still rely on them. It may still be quicker than trying to contact a person via phone (i.e., like a TCP protocol).

One more important definition we need to understand is that of a ***port***. Although a computer usually has a single physical connection to the network, data sent by different applications or delivered to them do so through the use of ***ports*** configured on same physical connection. A ***port*** is used as a gateway or "entry point" into an application.

Data transmitted over the internet to an application requires the address of the destination computer and the application's port number. A computer's address is a 32-bit IP address. The port number is a 16-bit number ranging from 0 to 65,535, with ports 0-1023 restricted by well-known applications like HTTP and FTP.

9.2 URLs

URL is an acronym for **Uniform Resource Locator** and is a reference (an address) to a resource on the Internet. So, it is used to represent the "location" of a webpage or web-based application.

URLs:

- are Strings that describe how to find a resource on the Internet
- represent names of resources which can be files, databases, applications, etc..
- resource names contain a host machine name, filename, port number, and other information.
- may also specify a ***protocol identifier*** (e.g., http, ftp)

Here is an example of a full URL:
http://www.scs.carleton.ca/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs

- http:// is the protocol identifier which indicates the protocol that will be used to obtain the resource.
- the remaining part is the ***resource name***, and its format depends on the protocol used to access it.

The complete list of components that can be found in a URL resource name are as follows:

Host Name	The name of the machine on which the resource lives: **http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs**
Port # (optional)	The port number to which to connect: **http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs**
Filename	The pathname to the file on the machine: **http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs**
Reference (optional)	A reference to a named *anchor* (a.k.a. *target*) within a resource that usually identifies a specific location within a file: **http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs**

In JAVA, there is a **URL** class defined in the **java.net** package. We can create our own URL objects as follows:

```
URL carleton = new URL("http://www.scs.carleton.ca/~courses/COMP1006/");
```

JAVA will "dissect" the given String in order to obtain information about protocol, hostName, file etc.... Due to this, JAVA may throw a **MalformedURLException** ... so we will need to do this:

```
try {
    URL carleton = new URL("http://www.scs.carleton.ca/~courses/COMP1006/");
} catch(MalformedURLException e) {
    ...
}
```

Another way to create a URL is to break it into its various components:

```
      try {
          URL theseNotes = new URL("http", "www.scs.carleton.ca", 80,

      "/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html");
```

```
    } catch(MalformedURLException e) {
        ...
    }
```

If you take a look at the JAVA API, you will notice some other constructors as well.

The URL class also supplies methods for extracting the parts (protocol, host, file, port and reference) of a URL object. Here is an example that demonstrates what can be accessed. Note that this example only manipulates a URL object, it does not go off to grab any webpages :) :

```
import java.net.*;
public class URLExample {
public static void main(String[] args) {
URL theseNotes = null;
try {
    theseNotes = new URL("http", "www.scs.carleton.ca", 80,

"/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html#URLs");
} catch(MalformedURLException e) {
    e.printStackTrace();
}
System.out.println(theseNotes);
System.out.println("protocol = " + theseNotes.getProtocol());
System.out.println("host = " + theseNotes.getHost());
System.out.println("filename = " + theseNotes.getFile());
System.out.println("port = " + theseNotes.getPort());
System.out.println("ref = " + theseNotes.getRef());
    }
}
```

Here is the output:

```
http://www.scs.carleton.ca:80/~courses/COMP1006/Notes/COMP1406_9/1406No
tes9.html#URLs
protocol = http
host = www.scs.carleton.ca
filename = /~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html
port = 80
ref = URLs
```

After creating a URL object, you can actually connect to that webpage and read the contents of the URL by using its **openStream()** method which returns an **InputStream**. You actually read from the webpage as if it were a simple text file. If an attempt is made to read from a URL that does not exist, JAVA will throw an **UnknownHostException**

Here is an example that reads a URL directly. It actually reads the file representing this set of notes and displays it line by line to the console. Notice that it reads the file as a text file, so we simply get the HTML code. Also, you must be connected to the internet to run this code:

```
import java.net.*;
import java.io.*;
public class URLReaderExample {
    public static void main(String[] args) {
        URL theseNotes = null;
        try {
            theseNotes = new URL("http", "www.scs.carleton.ca",
80,
```

```
"/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html");
            BufferedReader in = new BufferedReader(
                                    new
InputStreamReader(theseNotes.openStream()));

            // Now read the webpage file
            String lineOfWebPage;
            while ((lineOfWebPage = in.readLine()) != null)
                System.out.println(lineOfWebPage);

            in.close(); // Close the connection to the net
        } catch(MalformedURLException e) {
            System.out.println("Cannot find webpage " +
theseNotes);
        } catch(IOException e) {
            System.out.println("Cannot read from webpage " +
theseNotes);
        }
    }
}
```

The output should look something like this, assuming you could connect to the webpage:

```
<!DOCTYPE html PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
  <meta http-equiv="Content-Type"
 content="text/html; charset=iso-8859-1">
  <meta name="Author" content="Mark Lanthier">
  <meta name="GENERATOR"
 content="Mozilla/4.7 [en]C-CCK-MCD EBM-Compaq1  (Win95; U) [Netscape]">
  <title>COMP1006/1406 Notes 9 - Networking</title>
</head>
<body>
 
<table width="100%">
  <tbody>
    <tr>
      <td><i><font color="#006600">COMP1406/1006 - Design and
Implementation of Computer
Applications</font></i></td>
...
```

Example:

Here is a modification to the above example that reads the URL by making a **URLConnection** first. Since the tasks of opening a connection to a webpage and reading the contents may both generate an IOException, we cannot distinguish the kind of error that occured. By trying to establish the connection first, if any IOExceptions occur, we know they are due to a connection problem. Once the connection has been established, then any further IOException errors would be due to the reading of the webpage data.

```
import java.net.*;
import java.io.*;
public class URLConnectionReaderExample {
    public static void main(String[] args) {
        URL theseNotes = null;
        BufferedReader in = null;
        try {
            theseNotes = new URL("http", "www.scs.carleton.ca",
80,
"/~courses/COMP1006/Notes/COMP1406_9/1406Notes9.html");
        } catch(MalformedURLException e) {
            System.out.println("Cannot find webpage " +
theseNotes);
            System.exit(-1);
        }
        try {
            URLConnection  aConnection =
theseNotes.openConnection();
            in = new BufferedReader(
                        new
InputStreamReader(aConnection.getInputStream()));
        }
        catch (IOException e) {
            System.out.println("Cannot connect to webpage " +
theseNotes);
            System.exit(-1);
        }
        try {
            // Now read the webpage file
            String lineOfWebPage;
            while ((lineOfWebPage = in.readLine()) != null)
                System.out.println(lineOfWebPage);
            in.close(); // Close the connection to the net
        } catch(IOException e) {
            System.out.println("Cannot read from webpage " +
theseNotes);
        }
    }
}
```

9.3 Client/Server Communications

Many companies today sell services or products.   In addition, there are a large number of companies turning towards E-business solutions and various kinds of webserver/database technologies that allow them to conduct business over the internet as well as over other networks.

Such applications usually represent a client/server scenario in which one or more servers serve multiple clients.

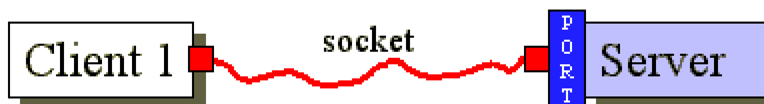
Our definition of a ***server*** here will be: *any application that provides a service and allows clients to communicate with it*. Such services may provide:

- a recent stock quote
- transactions for bank accounts
- an ability to order products
- an ability to make reservations
- a way to allow multiple clients to interact (Auction)

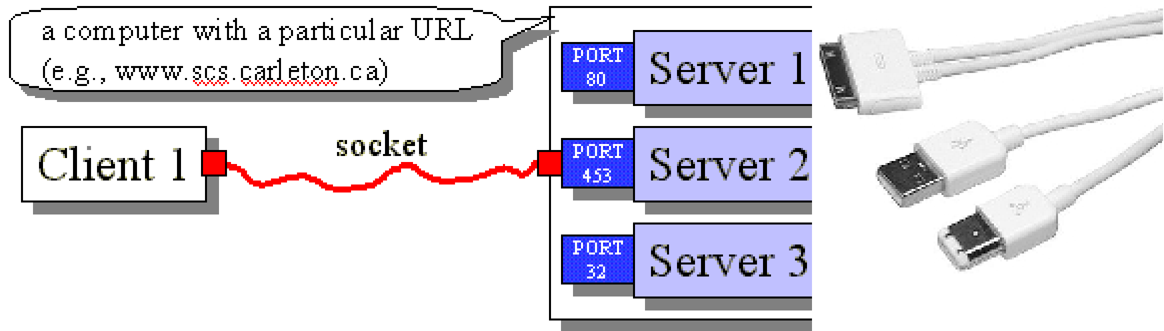
The ***client***, of course, will be: *any application that requests a service from a server*. The client typically "uses" the service and then displays results to the user. Normally, communication between the client and server must be reliable (no data can be dropped or missing):

- stock quotes must be accurate and timely
- banking transactions must be accurate and stable
- reservations/orders must be acknowledged

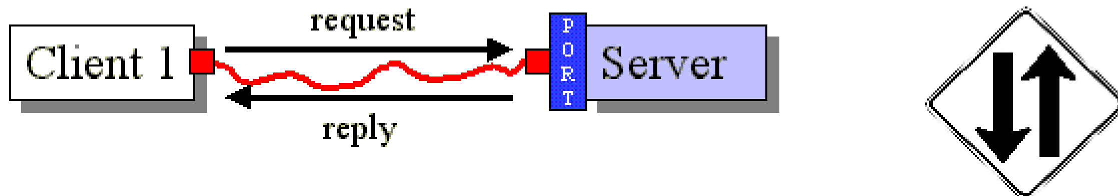
The TCP protocol, mentioned earlier, provides reliable point-to-point communication. Using TCP the client and server must establish a connection in order to communicate. To do this, each program binds a ***socket*** to its end of the connection. A ***socket*** is one endpoint of a two-way communication link between 2 programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application to which the data is to be sent. It is similar to the idea of plugging the two together with a cable.



The ***port number*** is used as the server's location on the machine that the server application is running. So if a computer is running many different server applications on the same physical machine, the port number uniquely identifies the particular server that the client wishes to communicate with:



The client and server may then each read and write to the socket bound to its end of the connection.



In JAVA, the server application uses a **ServerSocket** object to wait for client connection requests. When you create a **ServerSocket**, you must specify a port number (an **int**). It is possible that the server cannot set up a socket and so we have to expect a possible **IOException**. Here is an example:

```
public static int SERVER_PORT = 5000;

ServerSocket serverSocket;
try {
    serverSocket   = new ServerSocket(SERVER_PORT);
} catch(IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot open connection to Server",
                                  "Error", JOptionPane.ERROR_MESSAGE);
}
```

The server can communicate with only one client at a time. The server waits for an incoming client request through the use of the **accept()** message:

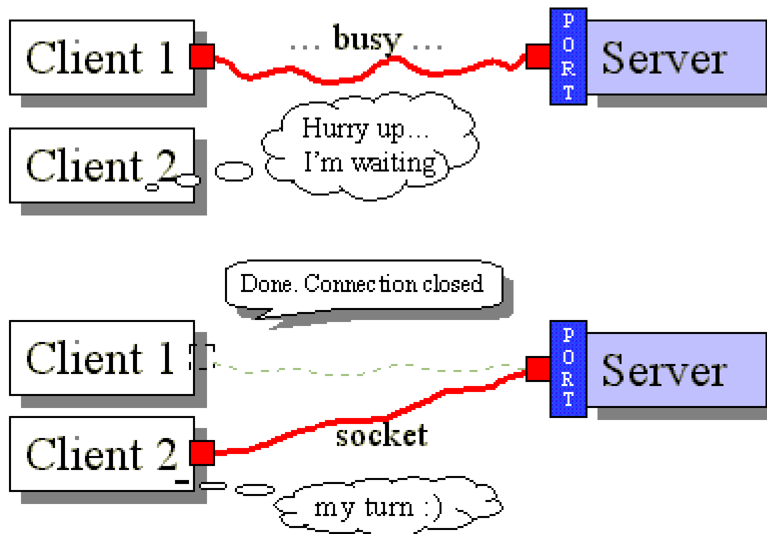
```
Socket aClientSocket;
try {
    aClientSocket = serverSocket.accept();
} catch(IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot accept incoming client
connection",
                                  "Error", JOptionPane.ERROR_MESSAGE);
}
```

When the `accept()` method is called, the server program actually waits (i.e., ***blocks***) until a client becomes available (i.e., an incoming client request arrives. Then it creates and returns a **Socket** object through which communication takes place.

Once the client and server have completed their interaction, the socket is then closed:

```
aClientSocket.close();
```

Only then may the next client open a socket connection to the server. So, remember ... if one client has a connection, everybody else has to wait until they are done:



So how does the client connect to the server ? Well, the client must know the address of the server as well as the PORT number. The server's address is stored as an **InetAddress** object which represents any IP address (i.e., an internet address, an ftp site, local machine etc,...). If the server and client are on the same machine, the static method **getLocatHost()** in the **InetAddress** class may be used to get an address representing the local machine.

```
public static int SERVER_PORT = 5000;

try {
     InetAddress address = InetAddress.getLocalHost();
     Socket socket = new Socket(address, SERVER_PORT);
} catch(UnknownHostException e) {
     JOptionPane.showMessageDialog(null, "Host Unknown",
                                   "Error", JOptionPane.ERROR_MESSAGE);
} catch(IOException e) {
     JOptionPane.showMessageDialog(null, "Cannot connect to server",
                                   "Error", JOptionPane.ERROR_MESSAGE);
}
```

Once again, a socket object is returned which can then be used for communication.
Here is an example of what a local host may look like:

```
cr850205-a/169.254.180.32
```

The **getLocalHost()** method may, however, generate an **UnknownHostException**.
You can also make an **InetAddress** object by specifying the network IP address directly or the machine name directly as follows:

```
InetAddress.getByName("169.254.1.61");
InetAddress.getByName("www.scs.carleton.ca");
```

So how do we actually do communication between the client and the server ? Well, each socket has an **inputStream** and an **outputStream**. So, once we have the sockets, we simply ask for these streams and then reading and writing may occur.

```
try {
    InputStream   in = socket.getInputStream();
    OutputStream  out = socket.getOutputStream();
} catch(IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot open I/O
Streams",
                                  "Error",
JOptionPane.ERROR_MESSAGE);
}
```

Normally, however, we actually wrap these input/output streams with text-based, datatype-based or object-based wrappers:

```
    ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
    ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());

    BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
    PrintWriter out = new PrintWriter(socket.getOutputStream());

    DataInputStream in = new DataInputStream(socket.getInputStream());
    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
```

You may look back at the notes on streams to see how to write to the streams. However, one more point ... when data is sent through the output stream, the **flush()** method should be sent to the output stream so that the data is not buffered, but actually sent right away.

Also, you must be careful when using **ObjectInputStreams** and **ObjectOutputStreams**. When you create an **ObjectInputStream**, it blocks while it tries to read a header from the underlying **SocketInputStream**. When you create the corresponding **ObjectOutputStream** at the far end, it writes the header that the **ObjectInputStream** is waiting for, and both are able to continue. If you try to create both **ObjectInputStreams** first, each end of the connection is waiting for the other to complete before proceeding which results in a deadlock situation (i.e., the programs seems to hang/halt). This behaviour is described in the API documentation for the **ObjectInputStream** and **ObjectOutoutStream** constructors.

9.4 Client/Server Example

Lets now take a look at a real example. In this example, a client will attempt to:

1. connect to a server
2. ask the server for the current time
3. ask the server for the number of requests that the server has handled so far
4. ask the server for an invalid request (i.e., for a pizza)

Here is the client application:

```
import java.net.*;
import java.io.*;
import javax.swing.JOptionPane;
public class Client  {
    private Socket          socket;
    private BufferedReader  in;
    private PrintWriter     out;

    // Make a connection to the server
    private void connectToServer() {
        try {
            socket = new Socket(InetAddress.getLocalHost(),
Server.SERVER_PORT);

            in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()));
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "CLIENT: Cannot connect to
server",
                                          "Error",
JOptionPane.ERROR_MESSAGE);
            System.exit(-1);
        }
    }

    // Disconnect from the server
    private void disconnectFromServer() {
        try {
            socket.close();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "CLIENT: Cannot disconnect
from server",
                                          "Error",
JOptionPane.ERROR_MESSAGE);
        }
    }

    // Ask the server for the current time
    private void askForTime() {
        connectToServer();
```

```
        out.println("What Time is It ?");
        out.flush();
        try {
            String time = in.readLine();
            System.out.println("CLIENT: The time is " + time);
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "CLIENT: Cannot receive time
from server",
                                          "Error",
JOptionPane.ERROR_MESSAGE);
        }
        disconnectFromServer();
    }

    // Ask the server for the number of requests obtained
    private void askForNumberOfRequests() {
        connectToServer();
        out.println("How many requests have you handled ?");
        out.flush();

        int count = 0;
        try {
            count = Integer.parseInt(in.readLine());
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "CLIENT: Cannot receive num
requests from server",
                                          "Error",
JOptionPane.ERROR_MESSAGE);
        }
        System.out.println("CLIENT: The number of requests are " + count);
        disconnectFromServer();
    }

    // Ask the server to order a pizza
    private void askForAPizza() {
        connectToServer();
        out.println("Give me a pizza");
        out.flush();
        disconnectFromServer();
    }

    public static void main (String args[]) {
        Client c = new Client();
        c.askForTime();
        c.askForNumberOfRequests();
        c.askForAPizza();
        c.askForTime();
        c.askForNumberOfRequests();
    }
}
```

Now the server application runs forever, continually waiting for incoming client requests:

```
import java.net.*;   // all socket stuff is in here
import java.io.*;
import javax.swing.JOptionPane;
```

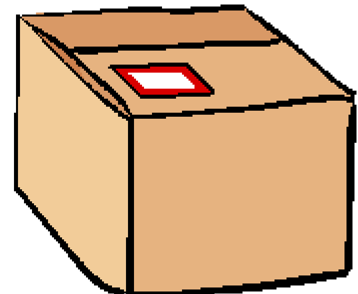
```java
public class Server {
    public static int SERVER_PORT = 6000;  // arbitrary, but above 1023
    private int counter = 0;

    // Helper method to get the ServerSocket started
    private ServerSocket goOnline() {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(SERVER_PORT);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error creating
network connection",
                                          "Error",
JOptionPane.ERROR_MESSAGE);
        }
        System.out.println("SERVER online");
        return serverSocket;
     }

     // Handle all requests
     private void handleRequests(ServerSocket serverSocket) {
        while(true) {
            Socket socket = null;
            BufferedReader in = null;
            PrintWriter out = null;
            try {
                // Wait for an incoming client request
                socket = serverSocket.accept();

                // At this point, a client connection has been made
                in = new BufferedReader(
                            new InputStreamReader(socket.getInputStream()));
                out = new PrintWriter(socket.getOutputStream());
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "SERVER: Error connecting
to client",
                                              "Error",
JOptionPane.ERROR_MESSAGE);
                System.exit(-1);
            }
            // Read in the client's request
            try {
                String request = in.readLine();
                System.out.println("SERVER: Client Message Received: " +
request);
                if (request.equals("What Time is It ?")) {
                    out.println(new java.util.Date());
                    counter++;
                }
                else if (request.equals("How many requests have you handled
?"))
                    out.println(counter++);
                else
                    System.out.println("SERVER: Unknown request: " +
request);
```

```
                // Now make sure that the response is sent
                out.flush();

                // We are done with the client's request
                socket.close();
            } catch(IOException e) {
                JOptionPane.showMessageDialog(null, "SERVER: Error
communicating with client",
                                              "Error",
JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    public static void main (String args[]) {
        Server s = new Server();
        ServerSocket ss = s.goOnline();
        if (s != null) s.handleRequests(ss);
    }
}
```

Note, to run this using JCreator, we will have to execute two different JCreator applications, one for the server and one for the client.

9.5 Datagram Sockets

Recall that with datagrams there is no direct socket connection between the client and the server. That is, packets are received "in seemingly random order" from different clients. It is similar to the way email works. If the client requests or server responses are too big, they are broken up into multiple packets and sent one packet at a time. The server is not guaranteed to receive the packets all at once, nor in the same order, nor is it guaranteed to receive all the packets !!



Let us look at the same client-server application, but by now using **DatagramSockets** and **DatagramPackets**. Once again, the server will be in a infinite loop accepting messages, although there will be no direct socket connection to the client. We will be setting up a ***buffer*** (i.e., an array of bytes) which will be used to receive incoming requests. Each message is sent as a ***packet***. Each packet contains:

- the ***data*** of message (i.e., the message itself)
- the ***length*** of the message (i.e., the number of bytes)
- the ***address*** of the sender (as an InetAddress)
- the ***port*** of the sender

The code for packaging and sending an outgoing packet involves creating a **DatagramSocket**

and then constructing a **DatagramPacket**. The packet requires an array of bytes, as well as the address and port in which to send to. The byte array can be obtained from most objects by sending a **getBytes()** message to the object. Finally, a **send()** message is used to send the packet:

```
    DatagramSocket  socket = new DatagramSocket();
    byte[]          sendBuffer = "This is the data (which does
not have to be a String)".getBytes();
    DatagramPacket  packetToSend = new
DatagramPacket(sendBuffer, sendBuffer.length, anInetAddress,
aPort);
    socket.send(packetToSend);
```

The server code for receiving an incoming packet involves allocating space (i.e., a byte array) for the **DatagramPacket** and then receiving it. The code looks as follows:

```
    byte[] recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
    DatagramPacket receivePacket = new DatagramPacket(recieveBuffer,
recieveBuffer.length);
    socket.receive(receivePacket);
```

We then need to extract the data from the packet. We can get the address and port of the sender as well as the data itself from the packet as follows:

```
    InetAddress   sendersAddress = receivePacket.getAddress();
    int           sendersPort = receivePacket.getPort();
    String        sendersData = new
String(receivePacket.getData(), 0, receivePacket.getLength());
```

In this case the data sent was a String, although it may in general be any object.

By using the sender's address and port, whoever receives the packet can send back a reply.

Here is the modified client/server code using the DatagramPackets:

```
import java.net.*;
import java.io.*;
import javax.swing.JOptionPane;
public class PacketServer {
public static int SERVER_PORT = 6000;
private static int INPUT_BUFFER_LIMIT = 500;

private int counter = 0;

// Handle all requests
private void handleRequests() {
    System.out.println("SERVER online");

    // Create a socket for communication
    DatagramSocket socket = null;
    try {
```

```
        socket = new DatagramSocket(SERVER_PORT);
    } catch (SocketException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Cannot connect to
network",
                                      "Error", JOptionPane.ERROR_MESSAGE);
        System.exit(-1);
    }

    // Now handle incoming requests
    while(true) {
        try {
           // Wait for an incoming client request
           byte[] recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
            DatagramPacket receivePacket = new DatagramPacket(
                            recieveBuffer, recieveBuffer.length);
            socket.receive(receivePacket);

           // Extract the packet data that contains the request
            InetAddress address = receivePacket.getAddress();
           int clientPort = receivePacket.getPort();
            String request = new String(receivePacket.getData(), 0,
                                        receivePacket.getLength());
            System.out.println("SERVER: Packet received: \"" + request +
                            "\" from " + address + ":" + clientPort);

           // Decide what should be sent back to the client
           byte[] sendBuffer;
           if (request.equals("What Time is It ?")) {
                System.out.println("SERVER: sending packet with time info");
                sendResponse(socket, address, clientPort,
                          new java.util.Date().toString().getBytes());
                counter++;
            }
           else if (request.equals("How many requests have you handled ?")) {
                System.out.println("SERVER: sending packet with num
requests");
                sendResponse(socket, address, clientPort,
                             ("" + ++counter).getBytes());
            }
           else
                System.out.println("SERVER: Unknown request: " + request);
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error receiving
client requests",
                                      "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
}

// This helper method sends a given response back to the client
private void sendResponse(DatagramSocket socket, InetAddress address,
                          int clientPort, byte[] response) {
    try {
        // Now create a packet to contain the response and send it
        DatagramPacket sendPacket = new DatagramPacket(response,
                            response.length, address, clientPort);
```

```
        socket.send(sendPacket);
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Error sending response
to client" +
                                      address + ":" + clientPort,
                                      "Error", JOptionPane.ERROR_MESSAGE);
    }
}

public static void main (String args[]) {
    new PacketServer().handleRequests();
}

}
```

Notice that only one **DatagramSocket** is used, but that a new **DatagramPacket** object is created for each incoming message.

Now lets look at the client:

```
import java.net.*;
import java.io.*;
import javax.swing.JOptionPane;
public class PacketClient  {
private static int INPUT_BUFFER_LIMIT = 500;
private InetAddress localHost;

public PacketClient() {
    try {
        localHost = InetAddress.getLocalHost();
    } catch(UnknownHostException e) {
        JOptionPane.showMessageDialog(null, "CLIENT: Error
connecting to network",
                                      "Error",
JOptionPane.ERROR_MESSAGE);
        System.exit(-1);
    }
}

// Ask the server for the current time
private void askForTime() {
    DatagramSocket socket = null;
    try {
        socket = new DatagramSocket();
        byte[] sendBuffer = "What Time is It ?".getBytes();
        DatagramPacket sendPacket = new
DatagramPacket(sendBuffer,
                                sendBuffer.length, localHost,
Server.SERVER_PORT);
        System.out.println("CLIENT: Sending time request to
server");
        socket.send(sendPacket);
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "CLIENT: Error
```

```
sending time request to server",
                                        "Error",
JOptionPane.ERROR_MESSAGE);
    }

    try {
        byte[] receiveBuffer = new byte[INPUT_BUFFER_LIMIT];
        DatagramPacket receivePacket = new
DatagramPacket(receiveBuffer, receiveBuffer.length);
        socket.receive(receivePacket);
        System.out.println("CLIENT: The time is " +
                        new String(receivePacket.getData(), 0,
receivePacket.getLength()));
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "CLIENT: Cannot
receive time from server",
                                        "Error",
JOptionPane.ERROR_MESSAGE);
    }
    socket.close();
}

// Ask the server for the number of requests obtained
private void askForNumberOfRequests() {
    DatagramSocket socket = null;
    try {
        socket = new DatagramSocket();
        byte[] sendBuffer = "How many requests have you handled
?".getBytes();
        DatagramPacket sendPacket = new
DatagramPacket(sendBuffer,
                            sendBuffer.length, localHost,
Server.SERVER_PORT);
        System.out.println("CLIENT: Sending request count request
to server");
        socket.send(sendPacket);
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "CLIENT: Error
sending request to server",
                                        "Error",
JOptionPane.ERROR_MESSAGE);
    }

    try {
        byte[] receiveBuffer = new byte[INPUT_BUFFER_LIMIT];
        DatagramPacket receivePacket = new
DatagramPacket(receiveBuffer, receiveBuffer.length);
        socket.receive(receivePacket);
        System.out.println("CLIENT: The number of requests are "
+
                        new String(receivePacket.getData(), 0,
receivePacket.getLength()));
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "CLIENT: Cannot
receive num requests from server",
                                        "Error",
```

```
JOptionPane.ERROR_MESSAGE);
    }
    socket.close();
}

// Ask the server to order a pizza
private void askForAPizza() {
    try {
        byte[] sendBuffer = "Give me a pizza".getBytes();
        DatagramPacket sendPacket = new
DatagramPacket(sendBuffer,
                        sendBuffer.length, localHost,
Server.SERVER_PORT);
        DatagramSocket socket = new DatagramSocket();
        System.out.println("CLIENT: Sending pizza request to
server");
        socket.send(sendPacket);
        socket.close();
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "CLIENT: Error
sending request to server",
                                      "Error",
JOptionPane.ERROR_MESSAGE);
    }
}

public static void main (String args[]) {
    PacketClient c = new PacketClient();
    c.askForTime();
    c.askForNumberOfRequests();
    c.askForAPizza();
    c.askForTime();
    c.askForNumberOfRequests();
}

}
```

9.6 Auction Example

Let us look at a large example now in which a server application represents an auction. Items are put up for auction and clients bid over the network on the items. Clients must first register with the auction server and then they may make bids on the items as they are placed up for auction. The user at the server end, decides when the item is no longer up for auctioning.

The example involves 11 classes:

- Model classes:
    - **Customer** - a customer who will be bidding at the auction:

aCustomer

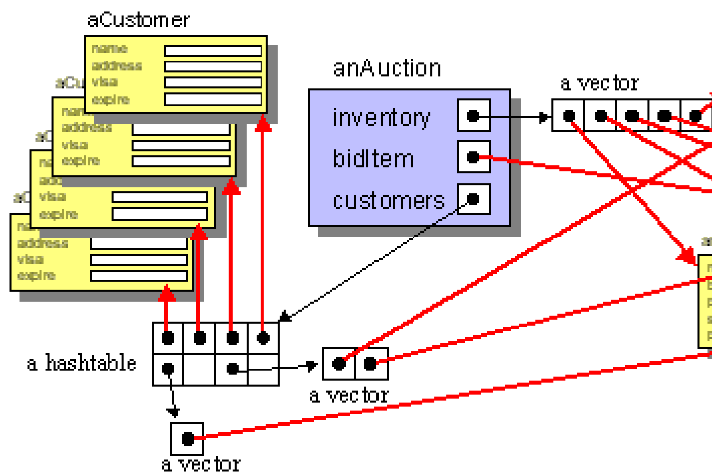
name	Bob Upandown
address	23 Lois Lane
visa	1425 3728 8939 9052
expire	11/05

- **AuctionItem** - an item which is up for Auctioning:

anAuctionItem

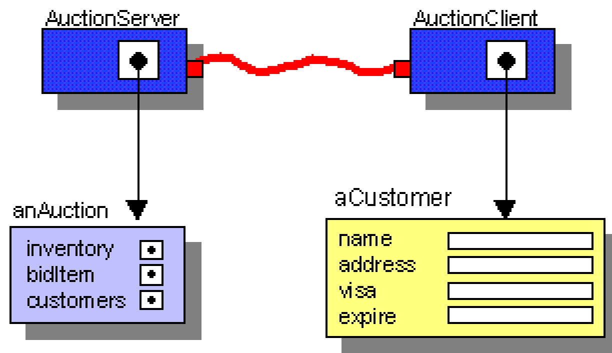
name	Antique Table
bid	150.00
purchaser	• → aCustomer (name, address, visa, expire)
sold	false
picture	table.gif

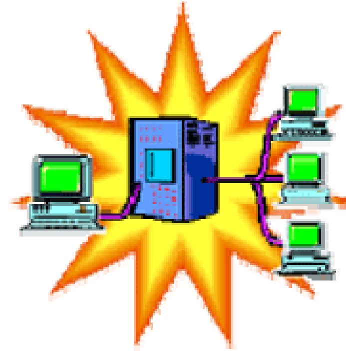
- **Auction** - the auction itself, keeping track of inventory of items to be auctioned, the current item up for auction and the customers (and their purchases) and the auctioning business logic:



- <u>Communication classes:</u>

    - **AuctionServer** - allows communication between the auction and the outside world.
    - **AuctionClient** - allows communication (i.e., bidding/responses) between the client and the server.





- <u>User Interface classes:</u>
    - **AuctionServerApp** - the server GUI (only 1 server runs this).
    - **AuctionClientApp** - the client GUI (each client runs this).
    - **RegistrationDialog** - the dialog box used for getting registration information.
    - **AuctionCatalogDialog** - the dialog box used to display the auction's catalog of items.
    - **AuctionItemDialog** - the dialog box that allows a new AuctionItem to be specified (at the server).
    - **DialogClientInterface** - the interface that allows dialog boxes to inform their owners upon closing.

Let us consider the basic model classes. First, we will examine the **Customer** ... it is quite simple:

```
public class Customer implements java.io.Serializable {
    private String   name;
    private String   address;
    private String   visa;
    private String   expire;

    public Customer() { this("","","",""); }
    public Customer(String n, String a, String v, String e) {
        name = n; address = a; visa = v; expire = e;
    }
    public String getName() { return name; }
    public String getAddress() { return address; }
    public String getVisa() { return visa; }
    public String getExpire() { return expire; }

    public void setName(String n) { name = n; }
    public void setAddress(String a) { address = a; }
    public void setVisa(String v) { visa = v; }
    public void setExpire(String e) { expire = e; }

    public boolean hasMissingInformation() {
        return ((name == null) || (name.length()==0) ||
                (visa == null) || (visa.length()==0) ||
                (address == null) || (address.length()==0) ||
```

```
                (expire == null) || (expire.length()==0));
    }
}
```

The class is straight forward. The **hasMissingInformation()** method will be useful later when we ask the user for the customer information. It checks to make sure that there are non-null, non-zero-length strings for all the data. Of course, we are not validating the visa number anywhere.

The **AuctionItem** class is similarily simple:

```
public class AuctionItem implements java.io.Serializable {
    private String      name;
    private float       bid;
    private Customer    purchaser;
    private boolean     sold;
    private String      picture;

    public AuctionItem() { this("", 0, ""); }
    public AuctionItem(String n, float startingBid, String fileName) {
        name = n;
        bid = startingBid;
        purchaser = null;
        picture = fileName;
        sold = false;
    }
    public String getName() { return name; }
    public float getBid() { return bid; }
    public Customer getPurchaser() { return purchaser; }
    public String getPicture() { return picture; }
    public boolean isSold() { return sold; }

    public void setName(String n) { name = n; }
    public void setBid(float amount) { bid = amount; }
    public void setPicture(String imageName) { picture = imageName; }
    public void setPurchaser(Customer c) { purchaser = c; }
    public void setSold() { sold = true; }

    public String toString() {
        if (sold) return "[SOLD " + name + " ]";
        else return name;
    }
}
```

Notice that the picture is actually just a filename. We use the following images for our auction items:

Now what about the **Auction** itself ? It is more complicated since it must deal with all auction-type behaviour such as placing items up for bid, handling/validating bids, registering clients, handling purchases etc... So what kind of methods do we need to write ? Consider first the Auction's state, and think about how it changes hen the users interact with it.

The auction will maintain an inventory of **AuctionItem** objects:

```
private ArrayList<AuctionItem>   inventory;
```

We will need to know which item is currently up for bid (initially **null**):

```
private AuctionItem   bidItem;
```

Finally, we will want to keep track of the **Customers** and their purchases. We can use a **HashMap** where the keys are the **Customer** objects themselves and the corresponding value will be an **ArrayList** of all **AuctionItems** purchased so far by that **Customer**:

```
private HashMap<Customer,ArrayList<AuctionItem>>  customers;
```

Customers should interact this way with the auction:

- Register with the auction - note that we keep an **ArrayList** of purchases for each **Customer**.

```
public void registerCustomer(Customer c) {
    customers.put(c, new ArrayList<AuctionItem>());
}
```

- Ask for a catalog (i.e., a list) of items that will be bidded on

```
public ArrayList<AuctionItem> getInventory() { return inventory;
}
```

- Ask if an item is currently up for bidding

```
public boolean hasItemUpForBid() { return bidItem != null; }
```

- Ask what item is currently up for bidding

```
public AuctionItem getBidItem() { return bidItem; }
```

- Ask what the latest bid is - return 0 if no item is up for bid

```java
public float latestBid() {
    if (bidItem == null) return 0;
    return bidItem.getBid();
}
```

- Make a bid for the latest item

```java
public boolean acceptBidFrom(String name, float amount) {
    // If nothing is up for bidding, don't accept the bid
    if (bidItem == null) return false;

    // First make sure the bid is actually valid
    if (amount <= latestBid()) return false;

    // Now make sure the customer is valid
    Customer c = customerWithName(name);
    if (c != null) {
        // Store the bid amount AND the bidder
        bidItem.setPurchaser(c);
        bidItem.setBid(amount);
        return true;
    }
    return false;
}
```

- Ask who made the latest bid (as confirmation when the customer makes a bid)

```java
public Customer latestBidder() {
    if (bidItem == null) return null;
    return bidItem.getPurchaser();
}
```

What about the person running the auction ?   What kind of behaviour does he/she need ?

- Add to the inventory of items to be auctioned

```java
public void add(AuctionItem item) {
    inventory.add(item);
}
```

- Get a list of all customers and their purchases

```java
public HashMap<Customer, ArrayList<AuctionItem>> getCustomers() {
return customers; }
```

- Place an item up for bidding - make sure it was not already sold :)

```java
public void placeUpForBid(AuctionItem item) {
    if (item.isSold()) return;
    bidItem = item;
}
```

- Stop the bidding process for an item - call when the item is considered sold

```
public void stopBidding() {
    if (latestBidder() != null) {
        ArrayList<AuctionItem> purchases =
customers.get(latestBidder());
        purchases.add(bidItem);
        bidItem.setSold();
        inventory.remove(bidItem);
    }
    bidItem = null;
}
```

So, you can see that the methods are all quite simple. They are all public so that the **AuctionServer** can communicate with this model class when either customer requests come in, or when the server user interacts with his/her GUI.

Here is the combined code:

```
import java.util.ArrayList;
import java.util.HashMap;
public class  Auction {
private ArrayList<AuctionItem>                     inventory;
private AuctionItem                                bidItem;
private HashMap<Customer,ArrayList<AuctionItem>>   customers;

public Auction() { this(new ArrayList<AuctionItem>()); }
public Auction(ArrayList<AuctionItem> initInventory) {
    inventory = initInventory;
    bidItem = null;
    customers = new HashMap<Customer,ArrayList<AuctionItem>>();
}
public ArrayList<AuctionItem> getInventory() { return inventory;
}
public AuctionItem getBidItem() { return bidItem; }
public HashMap<Customer,ArrayList<AuctionItem>> getCustomers() {
return customers; }

// Return the latest bid
public float latestBid() {
    if (bidItem == null) return 0;
    return bidItem.getBid();
}

// Return the latest bidder
public Customer latestBidder() {
    if (bidItem == null) return null;
    return bidItem.getPurchaser();
}

// Return the name of the latest bidder
public String latestBidderName() {
    if ((bidItem == null) || (bidItem.getPurchaser() == null))
        return "";
```

```java
        return bidItem.getPurchaser().getName();
    }

    // Add the given item to the inventory
    public void add(AuctionItem item) {
        inventory.add(item);
    }

    // Register the Customer with the given information to Auction
    public void registerCustomer(String name, String address,
        String visa, String expire) {
        registerCustomer(new Customer(name, address, visa, expire));
    }

    // Register the given Customer with the Auction
    public void registerCustomer(Customer c) {
        customers.put(c, new ArrayList<AuctionItem>());
    }

    // Place the given item up for bidding, customers can now bid on
it
    public void placeUpForBid(AuctionItem item) {
        if (item.isSold()) return;
        bidItem = item;
    }

    // Return whether or not there is currently an item up for
bidding
    public boolean hasItemUpForBid() { return bidItem != null; }

    // Find the Customer with the given name
    public Customer customerWithName(String name) {
        for (Customer c: customers.keySet())
          if (c.getName().equals(name)) return c;
        return null;
    }

    // Accept an incoming bid for the item up for bid.
    // If it is a valid bid, remember who made the bid
    // and increase the latest bid amount.
    public boolean acceptBidFrom(String name, float amount) {
        // If nothing is up for bidding, don't accept the bid
        if (bidItem == null) return false;

        // First make sure the bid is actually valid
        if (amount <= latestBid()) return false;

        // Now make sure the customer is valid
        Customer c = customerWithName(name);
        if (c != null) {
            bidItem.setPurchaser(c);
            bidItem.setBid(amount);
            return true;
        }
```

```
        return false;
    }

    // Once the bidding has stopped, the item is considered
    // to be sold to the last bidder, if there was one.
    public void stopBidding() {
        if (latestBidder() != null) {
            ArrayList<AuctionItem> purchases =
customers.get(latestBidder());
            purchases.add(bidItem);
            bidItem.setSold();
            inventory.remove(bidItem);
        }
        bidItem = null;
    }

    public static Auction example1() {
        Auction      a = new Auction();
        AuctionItem  first = new AuctionItem("Antique
Table",150.0f,"table.jpg");
        a.add(first);
        a.add(new AuctionItem("JVC VCR",65.0f,"vcr.jpg"));
        a.add(new AuctionItem("Antique
Cabinet",400.0f,"cabinet.jpg"));
        a.add(new AuctionItem("5-piece
Drumset",190.0f,"drumset.jpg"));
        a.add(new AuctionItem("Violin & Case",100.0f,"violin.jpg"));
        a.add(new AuctionItem("13\" TV/VCR
Combo",100.0f,"tvvcr.jpg"));
        a.add(new AuctionItem("486Dx2-66 Laptop",125.0f,
"486laptop.jpg"));
        a.add(new AuctionItem("Rocking Chair",80.0f,
"rockingchair.jpg"));
        a.add(new AuctionItem("1996 Mazda
Miata",6500.0f,"miata.jpg"));
         a.placeItemUpForBid(first);
        return a;
    }

    }
```

Now, what about the server itself ? Well, we have seen earlier how to make a simple server that can **accept()** incoming messages from a client via a **ServerSocket** object. We use the same approach. We will simply get the server started, and then dispatch any incoming messages to an appropriate helper method.

What kinds of client messages should the server accept ?

- `'r'`: register a client
- `'b'`: handle an incoming bid;
- `'c'`: handle a catalog request
- `'u'`: handle an update request (i.e., latest bid info)

So, the server should wait in an infinite loop, accepting client messages forever. Here is the basic framework, we will add the helper methods later:

```
import java.io.*;
import java.net.*;
import javax.swing.JOptionPane;
public class  AuctionServer  extends Thread {
// These variables are required for communication with clients
public static int SERVER_PORT = 6000;
private ServerSocket       serverSocket;
private ObjectInputStream  inputStreamFromClient;
private ObjectOutputStream outputStreamToClient;
private boolean            online;

// This is the model on which we are auctioning
private Auction     auction;

// Keep the appl. too so we can update it when we change info
private AuctionServerApp serverApplication;

public Auction getAuction() { return auction; }
public AuctionServer(Auction a) {
    auction = a;  online = false;
}

// Allow a server application to register for updates to the model
public void registerForUpdates(AuctionServerApp app) {
    serverApplication = app;
}

// Attempt to bring the server online
public boolean goOnline() {
    online = false;
    try {
        serverSocket = new ServerSocket(SERVER_PORT);
        online = true;
        System.out.println("SERVER Auction Server Online");
        start(); // Starts the server by calling run()
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Error Getting Server
Online",
                                      "Error", JOptionPane.ERROR_MESSAGE);
        System.exit(-1);
    }
    return online;
}

// Do what is necessary to shut down the server connection
public boolean goOffline() {
    try {
        if (online){
            serverSocket.close();
            online = false;
            System.out.println("SERVER Auction Server Offline");
        }
    } catch(IOException e) {
```

```java
        JOptionPane.showMessageDialog(null, "SERVER: Error Going Offline",
                                      "Error", JOptionPane.ERROR_MESSAGE);
    }
    return online;
}

// Try disconnecting from the client
private boolean closeClientConnection(Socket s) {
    try {
        if (s != null) s.close();
        return true;
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Error During Client
Disconnect",
                                      "Error", JOptionPane.ERROR_MESSAGE);
        return false;
    }
}

// Accept incoming messages from clients forever
public void run() {
    // Accept messages forever
    while (online) {
        Socket socket = null;
        try {
           // Wait for an incoming message
            socket = serverSocket.accept();
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error Contacting
Client",
                                      "Error", JOptionPane.ERROR_MESSAGE);
        }
        try {
            // Make object streams for the socket
            inputStreamFromClient =
                new ObjectInputStream(socket.getInputStream());
            outputStreamToClient =
                new ObjectOutputStream(socket.getOutputStream());

            // Now handle the message
            try {
                String aLine; // Will hold initial command message
                aLine = (String)inputStreamFromClient.readObject();
                System.out.println("SERVER Received: " + aLine);

              if (aLine != null) {
                  // Dispatch to the helper methods
                  char command = aLine.charAt(0);
                  switch(command) {
                      case 'r': registerClient(); break;
                      case 'b': handleIncomingBid(); break;
                      case 'c': handleCatalogRequest(); break;
                      case 'u': handleUpdateRequest(); break;
                      default: System.out.println("SERVER Error:
                                    Invalid Message " + command);
                    }
```

```
                }
              else
                    System.out.println("SERVER Error: Invalid
                                    Client Message Command");
            } catch(ClassNotFoundException e) {
                System.out.println("SERVER Error in Client
                                    Message Data");
            }
        } catch(IOException e) {
            JOptionPane.showMessageDialog(null, "SERVER: Error Receiving
Client Message",
                                    "Error", JOptionPane.ERROR_MESSAGE);
        } finally {
            // Now close the connection to this client
            System.out.println("SERVER Closing Client Connection");
            closeClientConnection(socket);
        }
    }
}
// Test a simple AuctionServer
public static void main(String args[]) {
    AuctionServer server = new AuctionServer(Auction.example1());
    server.goOnline();
}

}
```

Notice that the code looks huge ... but its actually mostly error checking. For example, here is the run() method without the clutter of the error checking and comments (although the code below won't compile):

```
public void run() {
    while (online) {
        Socket socket = serverSocket.accept();
        inputStreamFromClient = new
ObjectInputStream(socket.getInputStream());
        outputStreamToClient = new
ObjectOutputStream(socket.getOutputStream());

        String aLine = (String)inputStreamFromClient.readObject();
        if (aLine != null) {
            char command = aLine.charAt(0);
            switch(command) {
              case 'r': registerClient(); break;
              case 'b': handleIncomingBid(); break;
              case 'c': handleCatalogRequest(); break;
              case 'u': handleUpdateRequest(); break;
              default: System.out.println("SERVER Error: Invalid Message " +
                                    command);
            }
        }
        closeClientConnection(socket);
    }
}
```

So what about handling the different messages ? Below are the helper methods.

When a registration message is received, we need to read in a **Customer** object (the client will have to make a **Customer** object with his/her name, address, visa & expiry date. We will take this object, make sure that no information is missing and then send a reply back to the client customer. If the registration is successful, we will send back `"Registration received"` otherwise we will send back `"Registration Error: information is missing"`.

```
// Handle an incoming request for a client to be registered
private void registerClient() {
    Customer c = null;
    try {
        c = (Customer)inputStreamFromClient.readObject();
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Error
Receiving Client Registration Information",
                                      "Error",
JOptionPane.ERROR_MESSAGE);
    } catch(ClassNotFoundException e) {
        System.out.println("SERVER Error In Client Registration
Data");
    }
    try {
        if (c != null) {
            if (c.hasMissingInformation())
                outputStreamToClient.writeObject("Registration
Error: information is missing");
            else {
                auction.registerCustomer(c);
                outputStreamToClient.writeObject("Registration
received");
            }
        }
        outputStreamToClient.flush();
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Error
Sending Registration Response",
                                      "Error",
JOptionPane.ERROR_MESSAGE);
    }
}
```

Now what about when the customer wants to make a bid ? In this case, we will need to know the name of the **Customer** as well as the amount being bid. We will need to verify that this customer is indeed registered and also that his/her bid is valid. If he/she is not registered, we will send back: `"Error: You MUST register first"`. If the bid is invalid, we will send back: `"Error: Your bid is invalid"`. Otherwise we will record the bid and send back: `"Bid received"`.

```
// Handle an incoming request for a client to make a bid
private void handleIncomingBid() {
    String name = "";
    float bid = 0;
    try {
        // Expect the client's name and bid
        name = (String)inputStreamFromClient.readObject();
        bid = ((Float)inputStreamFromClient.readObject()).floatValue();
    } catch(IOException e) {
```

```
        JOptionPane.showMessageDialog(null, "SERVER: Error Receiving Client
Bid Information",
                                      "Error", JOptionPane.ERROR_MESSAGE);
    } catch(ClassNotFoundException e) {
        System.out.println("SERVER Error In Client's Bid Data");
    }
    try {
        if (auction.customerWithName(name) == null)
            outputStreamToClient.writeObject("Error: You MUST register
first");
        else if (auction.acceptBidFrom(name, bid))
            outputStreamToClient.writeObject("Bid received");
        else
            outputStreamToClient.writeObject("Error: Your bid is invalid");
        outputStreamToClient.flush();
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Error Sending Bid
Response",
                                      "Error", JOptionPane.ERROR_MESSAGE);
    }
    // Update the server application to reflect the new bid information
    if (serverApplication != null)
        serverApplication.update();
}
```

Now what about requests for a client to have a catalog ? In this case, we simply send back the inventory **ArrayList**:

```
// Handle an incoming request for a client to have a catalog
private void handleCatalogRequest() {
    try {
        outputStreamToClient.writeObject(auction.getInventory());
        outputStreamToClient.flush();
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Error Sending Catalog to
Client",
                                      "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

That one was easy :). Our last message that we need to handle is an **update()**. The client application will repeatedly ask for the latest bid information so that it's screen can be refreshed. Each time the server receives this message, it should simply send back the **AuctionItem** that is currently up for bid. Note that this item contains all necessary update information including the name of the item, its latest bid value and its latest bidder. Note that if nothing is currently up for bid when this message is received, we simply send back **null**.

```
// Handle an incoming request for the latest bid information
private void handleUpdateRequest() {
    try {
        outputStreamToClient.writeObject(auction.getBidItem());
        outputStreamToClient.flush();
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null, "SERVER: Error Sending Update to
Client",
                                      "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Well, that is it for the **Server**. Now what about the **Client** ? Here is the framework

```
import java.io.*;
import java.net.*;
import javax.swing.JOptionPane;
import java.util.ArrayList;
public class AuctionClient {
// These variables hold the necessary socket information
// for communincating with the Auction Server
private Socket                socket;
private ObjectInputStream     inputStreamFromServer;
private ObjectOutputStream    outputStreamToServer;
private Object                serverReply;

// This is the Customer who is attached to this client as its model
private Customer     customer;
private AuctionItem  latestAuctionItem;

public AuctionClient(Customer c) { customer = c; }

// Return the server's latest reply, useful for GUI status pane
public Object getServerReply() { return serverReply; }

// Return the Customer information pertaining to this client
public Customer getCustomer() { return customer; }

// Return the AuctionItem being bid on
public AuctionItem getLatestAuctionItem() {
    return latestAuctionItem;
}

// Try connecting to the auction server
private boolean establishServerConnection() {
    try {
        socket = new
Socket(InetAddress.getLocalHost(),AuctionServer.SERVER_PORT);
        outputStreamToServer = new
ObjectOutputStream(socket.getOutputStream());
        inputStreamFromServer = new
ObjectInputStream(socket.getInputStream());
        return true;
    } catch(IOException e) {
        handleError("CLIENT: Error Connecting to Server");
        return false;
    }
}

// Try disconnecting from the auction server
private boolean closeServerConnection() {
    try {
        socket.close();
        return true;
    } catch(IOException e) {
        handleError("CLIENT: Error Disconnecting from Server");
        return false;
```

```
    }
}

// Use this to display errors and also to record them for the GUI
private void handleError(String s) {
    serverReply = s;
    JOptionPane.showMessageDialog(null, s, "Error",
JOptionPane.ERROR_MESSAGE);
}

// Test by registering, sending bid and requesting catalog
public static void main(String args[]) {
    AuctionServer server = new AuctionServer(Auction.example1());
    server.goOnline();
    AuctionClient client = new AuctionClient(
           new Customer("Mark", "23 Oak St.", "4256 4878 0439 2387",
"12/04"));
    System.out.println(client.register());
    System.out.println(client.getServerReply());
    System.out.println(client.sendBid("Mark", 120.23f));
    System.out.println(client.getServerReply());
    System.out.println(client.sendBid("Bob", 300));
    System.out.println(client.getServerReply());
    System.out.println(client.sendBid("Mark", 5000));
    System.out.println(client.getServerReply());
    System.out.println(client.sendForCatalog());
    System.out.println("\"" + client.sendForUpdate() + "\"");
    server.goOffline();
}

}
```

Notice in the main method that the interesting methods are missing. Let us create those now.

To register, the client must connect to the server, send a `"register request"` message and then send the **Customer** information for this client. We will wait for the server's reply and store it in the serverReply variable (which will be shown in the status pane of the GUI). Then, we close the server connection.

```
// Send a request to be registered for the auction by the server
public boolean register() {
    if (!establishServerConnection()) return false;

    // Output the appropriate registration information
    try {
        outputStreamToServer.writeObject("register request");
        outputStreamToServer.writeObject(customer);
        outputStreamToServer.flush();
    } catch(IOException e) {
        handleError("CLIENT: Error sending Registration");
    }

    // Now wait to see if the registration went through
    boolean result = false;
    try {
```

```
            serverReply = (String)inputStreamFromServer.readObject();
            if (serverReply != null)
                result = serverReply.equals("Registration received");
        } catch(IOException e) {
            handleError("CLIENT: Error receiving registration
response");
        } catch(ClassNotFoundException e) {
            handleError("CLIENT: Error in Server reply data");
        } finally {
            closeServerConnection();
            return result;
        }
    }
```

Now to bid on an item, we need to send a `"bid request"` message as well as the **Customer** name and the bid value. We then need to wait for the reply to see if it worked and simply store this reply in the serverReply for the GUI to display.

```
// Send a bid to the server
public boolean sendBid(String name, float bid) {
    if (!establishServerConnection()) return false;

    // Output the appropriate bid information
    try {
        outputStreamToServer.writeObject("bid request");
        outputStreamToServer.writeObject(name);
        outputStreamToServer.writeObject(new Float(bid));
        outputStreamToServer.flush();
    } catch(IOException e) {
        handleError("CLIENT: Error sending bid");
    }

    // Now wait to see if the bid went through
    boolean result = false;
    try {
        serverReply = (String)inputStreamFromServer.readObject();
        if (serverReply != null)
            result = serverReply.equals("Bid received");
    } catch(IOException e) {
        handleError("CLIENT: Error receiving bid response");
    } catch(ClassNotFoundException e) {
        handleError("CLIENT: Error in Server reply data");
    } finally {
        closeServerConnection();
        return result;
    }
 }
```

When a catalog is required, we simply send a simple `"catalog request"` message to the server and get back the result (stored in serverReply). The method will also return the catalog **ArrayList** ... but we will let the GUI figure out what to do with it :).

```
// Send a request for a catalog of auction items to the server
public ArrayList<AuctionItem> sendForCatalog() {
    if (!establishServerConnection())
   return new ArrayList<AuctionItem>();
```

```
    // Send the request
    try {
        outputStreamToServer.writeObject("catalog request");
        outputStreamToServer.flush();
    } catch(IOException e) {
        handleError("CLIENT: Error sending catalog request");
    }

    // Now wait to see if the request went through
    serverReply = new ArrayList<AuctionItem>();
    try {
        serverReply =
(ArrayList<AuctionItem>)inputStreamFromServer.readObject();
        if (serverReply == null)
            serverReply = new ArrayList<AuctionItem>();
    } catch(IOException e) {
        handleError("CLIENT: Error receiving catalog");
    } catch(ClassNotFoundException e) {
        handleError("CLIENT: Error in Server reply data");
    } finally {
        closeServerConnection();
        return (ArrayList<AuctionItem>)serverReply;
    }
}
```

OK ... Now for the update request. We will send `"update request"` to the server and then wait for the latest **AuctionItem** to be returned. We will return it from this method.

```
// Send a request for the latest bid information to the server
public AuctionItem sendForUpdate() {
    if (!establishServerConnection()) return null;

    // Send the request
    try {
        outputStreamToServer.writeObject("update request");
        outputStreamToServer.flush();
    } catch(IOException e) {
        handleError("CLIENT: Error sending update request");
        return null;
    }
    // Now wait to see if the request went through
    serverReply = null;
    latestAuctionItem = null;
    try {
        serverReply = (AuctionItem)inputStreamFromServer.readObject();
        latestAuctionItem = (AuctionItem)serverReply;
    } catch(IOException e) {
        handleError("CLIENT: Error receiving update");
    } catch(ClassNotFoundException e) {
        handleError("CLIENT: Error in Server reply data");
    } finally {
        closeServerConnection();
        return latestAuctionItem;
    }
}
```

Wow! That sure was fun. Actually, we can run the main method shown above since it runs the server and a client as well. It is a good idea to run these to make sure that everything works fine. Here is our test case:

```
    AuctionServer server = new AuctionServer(Auction.example1());
    server.goOnline();
    AuctionClient client = new AuctionClient(
           new Customer("Mark", "23 Oak St.", "4256 4878 0439 2387",
"12/04"));
    System.out.println(client.register());
    System.out.println(client.getServerReply());
    System.out.println(client.sendBid("Mark", 120.23f));
    System.out.println(client.getServerReply());
    System.out.println(client.sendBid("Bob", 300));
    System.out.println(client.getServerReply());
    System.out.println(client.sendBid("Mark", 5000));
    System.out.println(client.getServerReply());
    System.out.println(client.sendForCatalog());
    System.out.println("\"" + client.sendForUpdate() + "\"");
    server.goOffline();
```

Here is the output which server output in blue, client output in red:

```
SERVER Auction Server Online
SERVER Received a Command: register request
SERVER Closing Client Connection
true
Registration received
SERVER Received a Command: bid request
SERVER Closing Client Connection
false
Error: Your bid is invalid
SERVER Received a Command: bid request
SERVER Closing Client Connection
false
Error: You MUST register first
SERVER Received a Command: bid request
SERVER Closing Client Connection
true
Bid received
SERVER Received a Command: catalog request
SERVER Closing Client Connection
[Antique Table, JVC VCR, Antique Cabinet, 5-piece Drumset,
Violin & Case, 13" TV/VCR Combo, 486Dx2-66 Laptop, Rocking
Chair, 1996 Mazda Miata]
SERVER Received a Command: update request
SERVER Closing Client Connection
"Antique Table"
SERVER Auction Server Offline
```

Notice that there are some errors when closing because the sockets are still set up when the client quits :).

Now what about the GUI's ? Let us examine the dialog box that requests for a new item to be added to the auction inventory. It should allow the user to specify the item name, starting bid and picture file. Here is what it will look like, and its code is shown below:

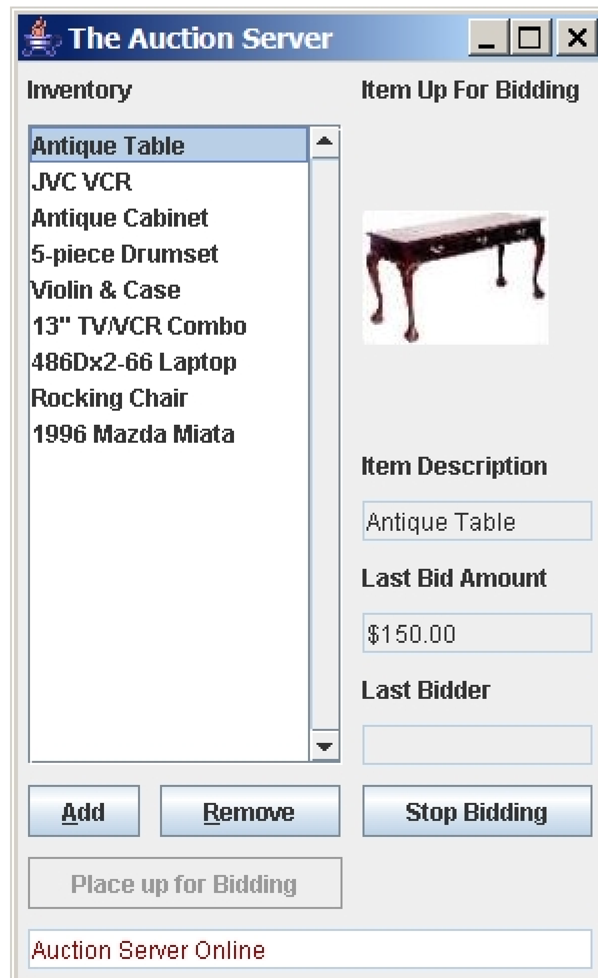
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class AuctionItemDialog extends JDialog {

      // Store a pointer to the model for changes later
      private AuctionItem item;

      private JTextField nameTextField;
      private JTextField bidTextField;
      private JTextField pictureField;

      // The buttons and main panel
      private JButton okButton;
      private JButton cancelButton;

      public AuctionItemDialog(Frame owner, AuctionItem ai){

          super(owner,"New Auction Item", true);

          item = ai;  // Store the model

          // Make a panel with two buttons (placed side by side)
          JPanel buttonPanel = new JPanel();
          buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT, 5,
      5));
          buttonPanel.add(okButton = new JButton("OK"));
          buttonPanel.add(cancelButton = new JButton("CANCEL"));

          // Make a panel with auction item information
          JPanel itemPanel  = new JPanel();

          nameTextField = new JTextField(item.getName(), 15);
          bidTextField = new JTextField(""+item.getBid(), 15);
          bidTextField.setHorizontalAlignment(JTextField.RIGHT);
          pictureField = new JTextField(item.getPicture(), 15);
```

```
        // Set the layoutManager and add the components
        itemPanel.setLayout(new GridLayout(3,2,5,5));
        itemPanel.add(new JLabel("Item Name:"));
        itemPanel.add(nameTextField);
        itemPanel.add(new JLabel("Starting Bid ($):"));
        itemPanel.add(bidTextField);
        itemPanel.add(new JLabel("Picture file (gif/jpg):"));
        itemPanel.add(pictureField);

        // Make the dialog box by adding the two panels
        setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5));
        add(itemPanel);
        add(buttonPanel);

        // Prevent the window from being resized
        setSize(365, 150);
        setResizable(false);

        // Listen for ok button click
        okButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event){
                okButtonClicked();
        }});

        // Listen for cancel button click
        cancelButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event){
                cancelButtonClicked();
        }});

        // Listen for window closing: treat like cancel button
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                cancelButtonClicked();
        }});
    }

    public AuctionItem getAuctionItem() { return item; }

    private void okButtonClicked(){
        // Update model to show changed owner name
        item.setName(nameTextField.getText());
        item.setBid(Float.parseFloat(bidTextField.getText()));
        item.setPicture(pictureField.getText());

        // Tell the client that ok was clicked
        if (getOwner() != null)
            ((DialogClientInterface)getOwner()).dialogFinished(this);
        dispose();
    }

    private void cancelButtonClicked(){
        // Tell the client that cancel was clicked
        if (getOwner() != null)
```

```
((DialogClientInterface)getOwner()).dialogCancelled(this);
    dispose();
}

}
```

Notice how the item variable (i.e., the model) is updated when the OK is clicked. Other than that, there is nothing new here.

OK, now let us look at the Server's GUI code. The **AuctionServerApp** class represents the application that is used by the person who is running the auction. It should be connected to an **Auction** object and have the ability to list/add/remove **AuctionItems** as well as place one up for bidding. In addition, the user will want to be able to stop the bidding when no clients have responded to the latest bid for a while. The image to the right is what the GUI will look like.



Below is the basic framework for the code. To keep things simple, the update methods and the event handlers are discussed afterwards.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.*;
import java.net.*;

public class  AuctionServerApp extends JFrame implements
DialogClientInterface {

    // This image gets shown when there is nothing up for
    bid
```

```
private static ImageIcon BLANK_IMAGE =  new
ImageIcon("blankItem.jpg");

private JTextField   statusField;
private JTextField   bidItemField;
private JTextField   bidAmountField;
private JTextField   bidderField;
private JList        itemsList;
private JLabel       pictureLabel;
private JButton      bidButton, stopButton;
private JButton      addButton, removeButton;

// This interface connects to an AuctionServer that
handles all the work
private AuctionServer     auctionServer;
private int               selectedItemIndex;
private AuctionItem       newAuctionItem; // item
being added to auction

private ListSelectionListener
itemsListListener;

public AuctionServerApp() { this(new
AuctionServer(new Auction())); }
public AuctionServerApp(AuctionServer a) {
    super("The Auction Server");
    auctionServer = a;
    auctionServer.registerForUpdates(this);
    if
(auctionServer.getAuction().getInventory().size() >
0)
        selectedItemIndex = 0;
    else
        selectedItemIndex = -1;
    initializeComponents();
    addListeners();
    update();
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(300,490);
    setVisible(true);
}

private void addListeners() {
    // When the window is first OPENED, go online
    addWindowListener(new WindowAdapter() {
        public void windowOpened(WindowEvent event) {
           goOnline(); }});

    // When the window is CLOSED, go offline
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent event)
{
           goOffline(); }});
```

```
    // Add a listener for the ADD button
    addButton.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent
theEvent) {
            handleAddAuctionItem(); }});

    // Add a listener for the REMOVE button
    removeButton.addActionListener(new
ActionListener() {
        public void actionPerformed(ActionEvent
theEvent) {
            handleRemoveAuctionItem(); }});

    // Add a listener for the PLACE UP FOR BIDDING
button
    bidButton.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent
theEvent) {
            handlePlaceForBid(); }});

    // Add a listener for the STOP BIDDING button
    stopButton.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent
theEvent) {
            handleStopBid(); }});

    // Add a selection listener for the inventory
list
    itemsList.addListSelectionListener(
    itemsListListener = new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent
theEvent) {
            handleSelectAuctionItem(); }});
}

// Cause the Auction Server to go online
private void goOnline() {
    if (auctionServer.goOnline()) {
        updateStatus("Auction Server Online");
        auctionServer.start(); // start the thread
    }
    else
        updateStatus("Error: Problem Getting Auction
Server Online");
}

// Cause the Auction Server to go offline
private void goOffline() {
    if (auctionServer.goOffline())
        updateStatus("Auction Server Offline");
    else
```

```
        updateStatus("Error: Problem Going Offline");
}

// Build the frame by adding all the components
private void initializeComponents() {
    GridBagLayout layout = new GridBagLayout();
    GridBagConstraints layoutConstraints = new
GridBagConstraints();

    JPanel panel = new JPanel();
    panel.setLayout(layout);
    setContentPane(panel);

    JLabel aLabel = new JLabel("Inventory");
    layoutConstraints.gridx = 0;
    layoutConstraints.gridy = 0;
    layoutConstraints.gridwidth = 2;
    layoutConstraints.gridheight = 1;
    layoutConstraints.fill = GridBagConstraints.NONE;
    layoutConstraints.insets = new Insets(5,5,5,5);
    layoutConstraints.anchor =
GridBagConstraints.NORTHWEST;
    layoutConstraints.weightx = 0.0;
    layoutConstraints.weighty = 0.0;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);

    aLabel = new JLabel("Item Up For Bidding");
    layoutConstraints.gridx = 2;
    layoutConstraints.gridwidth = 1;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);

    pictureLabel = new JLabel();
    layoutConstraints.gridy = 1;
    layoutConstraints.fill = GridBagConstraints.BOTH;
    layoutConstraints.weightx = 1.0;
    layoutConstraints.weighty = 1.0;
    layout.setConstraints(pictureLabel,
layoutConstraints);
    panel.add(pictureLabel);

    aLabel = new JLabel("Item Description");
    layoutConstraints.gridy = 2;
    layoutConstraints.fill = GridBagConstraints.NONE;
    layoutConstraints.weightx = 0.0;
    layoutConstraints.weighty = 0.0;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);

    aLabel = new JLabel("Last Bid Amount");
    layoutConstraints.gridy = 4;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);
```

```
    aLabel = new JLabel("Last Bidder");
    layoutConstraints.gridy = 6;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);

    bidItemField = new JTextField();
    bidItemField.setEditable(false);
    layoutConstraints.gridx = 2;
    layoutConstraints.gridy = 3;
    layoutConstraints.gridwidth = 1;
    layoutConstraints.gridheight = 1;
    layoutConstraints.fill =
GridBagConstraints.HORIZONTAL;
    layoutConstraints.anchor =
GridBagConstraints.NORTHWEST;
    layoutConstraints.weightx = 1.0;
    layoutConstraints.weighty = 0.0;
    layout.setConstraints(bidItemField,
layoutConstraints);
    panel.add(bidItemField);

    bidAmountField = new JTextField();
    bidAmountField.setEditable(false);
    layoutConstraints.gridy = 5;
    layout.setConstraints(bidAmountField,
layoutConstraints);
    panel.add(bidAmountField);

    bidderField = new JTextField();
    bidderField.setEditable(false);
    layoutConstraints.gridx = 2;
    layoutConstraints.gridy = 7;
    layout.setConstraints(bidderField,
layoutConstraints);
    panel.add(bidderField);

    addButton = new JButton("Add");
    addButton.setMnemonic('A');
    layoutConstraints.gridx = 0;
    layoutConstraints.gridy = 8;
    layoutConstraints.fill = GridBagConstraints.NONE;
    layoutConstraints.weightx = 0.0;
    layout.setConstraints(addButton,
layoutConstraints);
    panel.add(addButton);

    itemsList = new JList();
    JScrollPane scrollPane = new
JScrollPane(itemsList,

ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,

ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    layoutConstraints.gridx = 0;
    layoutConstraints.gridy = 1;
    layoutConstraints.gridwidth = 2;
```

```
    layoutConstraints.gridheight = 7;
    layoutConstraints.fill = GridBagConstraints.BOTH;
    layoutConstraints.anchor =
GridBagConstraints.CENTER;
    layoutConstraints.weighty = 1.0;
    layout.setConstraints(scrollPane,
layoutConstraints);
    panel.add(scrollPane);

    removeButton = new JButton("Remove");
    removeButton.setMnemonic('R');
    layoutConstraints.gridx = 1;
    layoutConstraints.gridy = 8;
    layoutConstraints.gridwidth = 1;
    layoutConstraints.gridheight = 1;
    layoutConstraints.fill = GridBagConstraints.NONE;
    layoutConstraints.ipadx = 10;
    layoutConstraints.ipady = 0;
    layoutConstraints.anchor =
GridBagConstraints.NORTHWEST;
    layoutConstraints.weighty = 0.0;
    layout.setConstraints(removeButton,
layoutConstraints);
    panel.add(removeButton);

    bidButton = new JButton("Place up for Bidding");
    layoutConstraints.gridx = 0;
    layoutConstraints.gridy = 9;
    layoutConstraints.gridwidth = 2;
    layout.setConstraints(bidButton,
layoutConstraints);
    panel.add(bidButton);

    stopButton = new JButton("Stop Bidding");
    layoutConstraints.gridx = 2;
    layoutConstraints.gridy = 8;
    layoutConstraints.gridwidth = 1;
    layout.setConstraints(stopButton,
layoutConstraints);
    panel.add(stopButton);

    statusField = new JTextField();
    statusField.setEditable(false);
    statusField.setBackground(new
Color(255,255,255));
    statusField.setForeground(new Color(160,0,0));
    layoutConstraints.gridx = 0;
    layoutConstraints.gridy = 10;
    layoutConstraints.gridwidth = 4;
    layoutConstraints.fill =
GridBagConstraints.HORIZONTAL;
    layoutConstraints.weightx = 1.0;
    layout.setConstraints(statusField,
layoutConstraints);
    panel.add(statusField);
}
```

```
        // Update all the components
        public void update() {

        itemsList.removeListSelectionListener(itemsListListen
        er);
            updateItemsList();
            updateRemoveButton();
            updatePlaceButton();
            updateStopButton();
            updateBidItemField();
            updateBidAmountField();
            updateBidderField();
            updatePictureLabel();

        itemsList.addListSelectionListener(itemsListListener)
        ;
        }

        // Test it by bringing up the server and some clients
        public static void main(String[] args) {
            new AuctionServerApp(new
        AuctionServer(Auction.example1()));
            new AuctionClientApp();
            new AuctionClientApp();
            new AuctionClientApp();
        }

    }
```

What about writing all the update methods ?
The inventory list is updated simply by obtaining the inventory **ArrayList** from the auction model:

```
    private void updateItemsList() {
        itemsList.setListData(new
Vector(auctionServer.getAuction().getInventory()));
        itemsList.setSelectedIndex(selectedItemIndex);
    }
```

The REMOVE button is disabled when nothing is selected from the list. The PLACE UP FOR BIDDING button and the STOP BIDDING buttons are disabled when the auction does not have anything up for bidding.

```
    private void  updateRemoveButton() {
        removeButton.setEnabled(selectedItemIndex != -1);
    }
    private void  updatePlaceButton() {
        bidButton.setEnabled(!auctionServer.getAuction().hasItemUpForBid());
    }
    private void  updateStopButton() {
        stopButton.setEnabled(auctionServer.getAuction().hasItemUpForBid());
    }
```

The text fields are all updated according to the latest bid item information:

```
    private void  updateBidItemField() {
        if (auctionServer.getAuction().hasItemUpForBid())

bidItemField.setText(auctionServer.getAuction().getBidItem().getName());
        else bidItemField.setText("");
    }

    private void  updateBidAmountField() {
        if (auctionServer.getAuction().hasItemUpForBid()) {
            java.text.DecimalFormat formatter = new
java.text.DecimalFormat("$0.00");
            bidAmountField.setText(formatter.format(
                      auctionServer.getAuction().latestBid()));
        }
        else bidAmountField.setText("");
    }

    private void  updateBidderField() {
        if (auctionServer.getAuction().hasItemUpForBid())

bidderField.setText(auctionServer.getAuction().latestBidderName());
        else bidderField.setText("");
    }

    private void  updatePictureLabel() {
        if (auctionServer.getAuction().hasItemUpForBid())
            pictureLabel.setIcon(new ImageIcon(
               auctionServer.getAuction().getBidItem().getPicture()));
        else
            pictureLabel.setIcon(BLANK_IMAGE);
    }
```

Finally, the status field shows whatever we pass it. Notice that this is not called from the **update()** method. Instead, whenever there is an important message, we call it:

```
    private void  updateStatus(String s) {
        statusField.setText(s);
    }
```

Now we look at the event handlers for the buttons.

When the ADD button is pressed, create a new **AuctionItem** and open the **AuctionItemDialog** box to edit it. We will set up **dialogFinished()** and **dialogCancelled()** methods to add or ignore the item as necessary.

```
    private void  handleAddAuctionItem() {
        newAuctionItem = new AuctionItem();
        new AuctionItemDialog(this, newAuctionItem).setVisible(true);
    }

    public void dialogFinished() {
        auctionServer.getAuction().add(newAuctionItem);
        selectedItemIndex = auctionServer.getAuction().getInventory().size()-
```

```
1;
        update();
    }
    public void dialogCancelled() {} //do nothing
```

When the REMOVE button is pressed, remove the currently selected item:

```
    private void  handleRemoveAuctionItem() {
        if (selectedItemIndex != -1) {
auctionServer.getAuction().getInventory().remove(selectedItemIndex);
            selectedItemIndex--;
            update();
        }
    }
```

When the PLACE UP FOR BIDDING button is pressed, make the currently selected item to be the one that is placed up for bidding:

```
    private void  handlePlaceForBid() {
        if (selectedItemIndex != -1) {
            auctionServer.getAuction().placeUpForBid(
                        (AuctionItem)auctionServer.getAuction().
                         getInventory().get(selectedItemIndex));
            update();
        }
    }
```

When the STOP BIDDING button is pressed, stop the current bid item from being bid on:

```
    private void  handleStopBid() {
        auctionServer.getAuction().stopBidding();
        update();
    }
```

Lastly, when the item is selected from the list, simply store its index in a local variable:

```
    private void  handleSelectAuctionItem() {
        selectedItemIndex = itemsList.getSelectedIndex();
        update();
    }
```

Well that is it for the server app!! Quite a lot of code, isn't it ? Now let us look at the Client-side GUI. First, we will consider the registration dialog. Notice that this dialog box is fairly straight forward. We pass in the **Customer** in the constructor and use this Customer's information to fill in the initial textFields. Since in our application, customers will only register once, this initial **Customer** object is probably filled with empty information. Nevertheless, we may want to use this dialog box in the future for editing purposes and in this case, our code will work fine. Notice also that when the OK button is clicked, the most recent data in the text fields is used to fill in the **Customer** object.

Auction Registration

Name: Mark

Address: 67 Elm St

VISA #: 7876 3232 8798 3434

Expiry Date: 09/03

OK CANCEL

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class  RegistrationDialog extends JDialog {
// Store a pointer to the model for changes later
private Customer customer;

private JTextField nameTextField;
private JTextField addressTextField;
private JTextField visaTextField;
private JTextField expireTextField;

// The buttons and main panel
private JButton okButton;
private JButton cancelButton;

// A constructor that takes the model and client as parameters
public RegistrationDialog(Frame owner, Customer c){

    // Call the super constructor that does all the work of
setting up the dialog
    super(owner,"Auction Registration",true);

    customer = c;   // Store the model

    // Make a panel with two buttons (placed side by side)
    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT, 5,
5));
    buttonPanel.add(okButton = new JButton("OK"));
    buttonPanel.add(cancelButton = new JButton("CANCEL"));

    // Make a panel with auction item information
    JPanel itemPanel  = new JPanel();

    // Create the textfields initially with the model's contents
    nameTextField = new JTextField(c.getName(), 15);
    addressTextField = new JTextField(c.getAddress(),15);
    visaTextField = new JTextField(c.getVisa(),15);
    expireTextField = new JTextField(c.getExpire(),15);

    // Set the layoutManager and add the components
    itemPanel.setLayout(new GridLayout(4,2,5,5));
```

```
    itemPanel.add(new JLabel("Name:"));
    itemPanel.add(nameTextField);
    itemPanel.add(new JLabel("Address:"));
    itemPanel.add(addressTextField);
    itemPanel.add(new JLabel("VISA #:"));
    itemPanel.add(visaTextField);
    itemPanel.add(new JLabel("Expiry Date:"));
    itemPanel.add(expireTextField);

    // Make the dialog box by adding bank account panel and
button panel
    setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5));
    add(itemPanel);
    add(buttonPanel);

    // Prevent the window from being resized
    setSize(365, 170);
    setResizable(false);

    // Listen for ok button click
    okButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event){
            okButtonClicked(); }});

    // Listen for cancel button click
    cancelButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event){
            cancelButtonClicked(); }});

    // Listen for window closing: treat like cancel button
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent event) {
            cancelButtonClicked(); }});
}

private void  okButtonClicked(){
    // Update model to show changed owner name
    customer.setName(nameTextField.getText());
    customer.setAddress(addressTextField.getText());
    customer.setVisa(visaTextField.getText());
    customer.setExpire(expireTextField.getText());

    // Tell the client that ok was clicked
    if (getOwner() != null)
        ((DialogClientInterface)getOwner()).dialogFinished();
    dispose();
}

private void  cancelButtonClicked(){
    // Tell the client that cancel was clicked
    if (getOwner() != null)
        ((DialogClientInterface)getOwner()).dialogCancelled();
    dispose();
}
```

```
}
```

Now the **AuctionCatalogDialog**, which displays the catalog returned from the server, should display a list of items and their pictures. The user should be able to browse around the list and see the pictures.

Notice that there is no client being passed in. In fact, there is no "response" that needs to be returned to the application. The user simply opens this window and does some browsing. So, there is no OK/CANCEL button combinations, simply a CLOSE button to close the window. Notice as well that the dialog box is non-modal, so the user can open a bunch of them.

A main method is provided to test out the code as well.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class AuctionCatalogDialog  extends JDialog {
// Store a pointer to the model for changes later
private ArrayList<AuctionItem>  inventory;

private JList   itemsList;
private JLabel  picture;
private JButton okButton;

// A constructor that takes the model and client as parameters
public AuctionCatalogDialog(Frame owner, ArrayList<AuctionItem>
items){

    // Call the super constructor that does all the work of
setting up the dialog
    super(owner,"Auction Inventory Catalog",false);
```

```java
    // Store the model and client into instance variables
    inventory = items;

    itemsList = new JList(new Vector<AuctionItem>(items));
    itemsList.setPrototypeCellValue("xxxxxxxxxxxxxxxxxxxxxxxxx");
    JScrollPane scrollPane = new JScrollPane(itemsList,
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    picture = new JLabel("");

    // Set the layoutManager and add the components
    setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));
    itemsList.setAlignmentX(FlowLayout.LEFT);
    add(new JLabel("Inventory"));
    add(itemsList);
    add(picture);
    add(okButton = new JButton("CLOSE"));

    // Prevent the window from being resized
    setSize(200, 450);
    setResizable(false);

    // Listen for CLOSE button click
    okButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event){
            dispose(); }});

    // Add a selection listener for the inventory list
    itemsList.addListSelectionListener(new
ListSelectionListener() {
        public void valueChanged(ListSelectionEvent theEvent) {
           if (itemsList.getSelectedValue() != null)
                picture.setIcon(new
ImageIcon(((AuctionItem)itemsList.

getSelectedValue()).getPicture())); }});

    // Listen for window closing: treat like CLOSE button
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent event) {
                dispose(); }});

    // Now make the dialog box appear
    setVisible(true);
}

// This is used for testing only
public static void main(String args[]) {
    ArrayList<AuctionItem> v = new ArrayList<AuctionItem>();
    v.add(new AuctionItem("Antique Table",150.0f, "table.jpg"));
    v.add(new AuctionItem("JVC VCR",65.0f,"vcr.jpg"));
    v.add(new AuctionItem("Antique
Cabinet",400.0f,"cabinet.jpg"));
    v.add(new AuctionItem("5-piece
Drumset",190.0f,"drumset.jpg"));
```

```
        v.add(new AuctionItem("Violin & Case",100.0f,"violin.jpg"));
        v.add(new AuctionItem("13\" TV/VCR
    Combo",100.0f,"tvvcr.jpg"));
        v.add(new AuctionItem("486Dx2-66
    Laptop",125.0f,"486laptop.jpg"));
        v.add(new AuctionItem("Rocking
    Chair",80.0f,"rockingchair.jpg"));
        v.add(new AuctionItem("1996 Mazda
    Miata",6500.0f,"miata.jpg"));
        JDialog dialog = new AuctionCatalogDialog(null, v);
    }

    }
```

Finally, we will examine the **AuctionClientApp** GUI application. Client users will want to have the ability to register for an auction. This should bring up the **RegistrationDialog**, and then use the **AuctionClient** to send this information to the server. Once registered, the client can then make bids.

The CATALOG button can be pressed at any time, and it should get the catalog from the sever, then bring up the **AuctionCatalogDialog** box.

The frame itself should display the information for the latest item which is being bid on. The user should be able to make a bid and press the MAKE BID button to send the bid to the server.

Notice as well that a **Timer** event is set up in the code. Every second, this timer event sends a request to the server for the latest **AuctionItem** information. This information is then returned to this client application and is shown in the window through an update call.

Once again, the code framework is given first, and then the update/event

handler methods are shown
afterwards.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.*;
import java.net.*;

public class  AuctionClientApp  extends JFrame implements
DialogClientInterface {

      // This image gets shown when there is nothing up for
      bid
      private static ImageIcon BLANK_IMAGE =  new
      ImageIcon("blankItem.jpg");

      private JTextField   statusField;
      private JTextField   bidItemField;
      private JTextField   bidAmountField;
      private JTextField   bidderField;
      private JTextField   newBidField;
      private JList        itemsList;
      private JLabel       pictureLabel;
      private JButton      bidButton;
      private JButton      registerButton, catalogButton;

      // This interface connects to an AuctionServer that
      handles all the work
      private AuctionClient       auctionClient;
      private float               bidToMake;
      private javax.swing.Timer   updateTimer;

      public AuctionClientApp() { this(new
      AuctionClient(new Customer())); }
      public AuctionClientApp(AuctionClient c) {
          super("UNREGISTERED Client");
          auctionClient = c;
          bidToMake = 0.0f;

          initializeComponents();
          addListeners();
          update();
          updateTimer.start(); // Start requesting for
      updates
          setSize(200,460);
          setDefaultCloseOperation(EXIT_ON_CLOSE);
          setVisible(true);
      }

      private void addListeners() {
          // Add a listener for the REGISTER button
          registerButton.addActionListener(new
```

```
ActionListener() {
        public void actionPerformed(ActionEvent
theEvent) {
            handleRegistration();  }});

    // Add a listener for the REMOVE button
    catalogButton.addActionListener(new
ActionListener() {
        public void actionPerformed(ActionEvent
theEvent) {
            handleCatalogRequest(); }});

    // Add a listener for the MAKE BID button
    bidButton.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent
theEvent) {
            handleMakeNewBid(); }});

    // Add a listener for key presses in the MAKE BID
textfield
    newBidField.getDocument().addDocumentListener(new
DocumentListener() {
        private void handleBidBeingMade() {
           if (newBidField.getText().length() == 0)
                bidToMake = 0;
           else try {
                bidToMake =
Float.parseFloat(newBidField.getText());
            } catch(NumberFormatException e) {
                bidToMake = 0;
            }
            updateMakeBidButton();
        }
        public void changedUpdate(DocumentEvent e) {
handleBidBeingMade(); }
        public void insertUpdate(DocumentEvent e) {
handleBidBeingMade(); }
        public void removeUpdate(DocumentEvent e) {
handleBidBeingMade(); }});

    //Add a Timer event handler for updates
    updateTimer =  new javax.swing.Timer(1000, new
ActionListener() {
        public void actionPerformed(ActionEvent
theEvent) {
            handleRequestUpdate(); }});
}

// Build the frame by adding all the components
private void initializeComponents() {
    GridBagLayout layout = new GridBagLayout();
    GridBagConstraints layoutConstraints = new
GridBagConstraints();
```

```
    JPanel panel = new JPanel();
    panel.setLayout(layout);
    setContentPane(panel);

    registerButton = new JButton("Register");
    layoutConstraints.gridx = 0;
    layoutConstraints.gridy = 0;
    layoutConstraints.gridwidth = 1;
    layoutConstraints.gridheight = 1;
    layoutConstraints.fill = GridBagConstraints.NONE;
    layoutConstraints.anchor =
GridBagConstraints.NORTHWEST;
    layoutConstraints.weightx = 0.0;
    layoutConstraints.weighty = 0.0;
    layout.setConstraints(registerButton,
layoutConstraints);
    panel.add(registerButton);

    catalogButton = new JButton("Catalog");
    layoutConstraints.gridx = 1;
    layoutConstraints.ipadx = 10;
    layoutConstraints.ipady = 0;
    layoutConstraints.anchor =
GridBagConstraints.NORTHEAST;
    layout.setConstraints(catalogButton,
layoutConstraints);
    panel.add(catalogButton);

    JLabel aLabel = new JLabel("Item Up For
Bidding");
    layoutConstraints.gridx = 0;
    layoutConstraints.gridy = 1;
    layoutConstraints.gridwidth = 2;
    layoutConstraints.fill = GridBagConstraints.BOTH;
    layoutConstraints.insets = new Insets(5,5,5,5);
    layoutConstraints.anchor =
GridBagConstraints.NORTHWEST;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);

    pictureLabel = new JLabel();
    layoutConstraints.gridy = 2;
    layoutConstraints.fill = GridBagConstraints.NONE;
    layoutConstraints.weightx = 1.0;
    layoutConstraints.weighty = 1.0;
    layout.setConstraints(pictureLabel,
layoutConstraints);
    panel.add(pictureLabel);

    aLabel = new JLabel("Item Description");
    layoutConstraints.gridy = 3;
    layoutConstraints.weightx = 0.0;
    layoutConstraints.weighty = 0.0;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);
```

```
    aLabel = new JLabel("Last Bid Amount");
    layoutConstraints.gridy = 5;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);

    aLabel = new JLabel("Last Bidder");
    layoutConstraints.gridy = 7;
    layout.setConstraints(aLabel, layoutConstraints);
    panel.add(aLabel);

    bidItemField = new JTextField();
    bidItemField.setEditable(false);
    layoutConstraints.gridy = 4;
    layoutConstraints.fill =
GridBagConstraints.HORIZONTAL;
    layoutConstraints.weightx = 1.0;
    layout.setConstraints(bidItemField,
layoutConstraints);
    panel.add(bidItemField);

    bidAmountField = new JTextField();
    bidAmountField.setEditable(false);
    layoutConstraints.gridy = 6;
    layout.setConstraints(bidAmountField,
layoutConstraints);
    panel.add(bidAmountField);

    bidderField = new JTextField();
    bidderField.setEditable(false);
    layoutConstraints.gridy = 8;
    layout.setConstraints(bidderField,
layoutConstraints);
    panel.add(bidderField);

    bidButton = new JButton("Make Bid");
    layoutConstraints.gridx = 1;
    layoutConstraints.gridy = 9;
    layoutConstraints.gridwidth = 1;
    layoutConstraints.fill = GridBagConstraints.NONE;
    layoutConstraints.weightx = 0.0;
    layout.setConstraints(bidButton,
layoutConstraints);
    panel.add(bidButton);

    newBidField = new JTextField();
    layoutConstraints.gridx = 0;
    layoutConstraints.gridwidth = 2;
    layoutConstraints.fill =
GridBagConstraints.HORIZONTAL;
    layoutConstraints.anchor =
GridBagConstraints.NORTHEAST;
    layoutConstraints.weightx = 1.0;
    layout.setConstraints(newBidField,
layoutConstraints);
    panel.add(newBidField);
```

```
        statusField = new JTextField();
        statusField.setEditable(false);
        statusField.setBackground(new
    Color(255,255,255));
        statusField.setForeground(new Color(160,0,0));
        layoutConstraints.gridy = 10;
        layoutConstraints.gridwidth = 2;
        layoutConstraints.weightx = 10.0;
        layout.setConstraints(statusField,
    layoutConstraints);
        panel.add(statusField);
    }

    // Update all the components
    private void update() {
        updateMakeBidButton();
        updateBidItemField();
        updateBidAmountField();
        updateBidderField();
        updateNewBidField();
        updatePictureLabel();
    }

    public static void main(String[] args) {
        JFrame frame = new AuctionClientApp();
    }

}
```

Let us look now at the update methods. The MAKE BID button should be disabled when there is nothing to bid on. We will see that the timer updates will eventually get something returned which is the item that is being bid on:

```
private void updateMakeBidButton() {
    bidButton.setEnabled((auctionClient.getLatestAuctionItem() != null) &&
                         (bidToMake != 0));
}
```

Each of the text fields are then updated according to the information from the latest bid item:

```
private void updateBidItemField() {
    AuctionItem item = auctionClient.getLatestAuctionItem();
    if (item == null) bidItemField.setText("");
    else bidItemField.setText(item.getName());
}

private void updateBidAmountField() {
    AuctionItem item = auctionClient.getLatestAuctionItem();
    if (item == null) bidAmountField.setText("");
    else bidAmountField.setText(String.valueOf(item.getBid()));
}

private void updateBidderField() {
    AuctionItem item = auctionClient.getLatestAuctionItem();
    if ((item == null) || (item.getPurchaser() == null))
        bidderField.setText("");
    else
```

```
        bidderField.setText(item.getPurchaser().getName());
}
```

When there is nothing to bid on, we erase the text inside the text field that is being used to make a bid. This is not so important, but it allows us to clear the field in between bids, so that no weird bid amounts will be accidentally submitted to the server:

```
private void updateNewBidField() {
    AuctionItem item = auctionClient.getLatestAuctionItem();
    if (item == null) newBidField.setText("");
}
```

The status field allows us to see what is going on. It displays error messages as well as server replies. We allow any string to be passed in here and we display whatever is passed in:

```
private void updateStatus(String s) {
    statusField.setText(s);
}
```

Lastly, the picture for the item is displayed. The picture to be displayed depends on the item which is currently up for bid. However, the **AuctionItems** only store the picture filename, not the picture itself. In fact, we are "faking" something here. The server actually has all the images on its machine, not the client. So, when **AuctionItem** objects are sent to the client, the client only has the filename, not the files. So it is impossible to be able to display the image !!! However, since our test program has everything running in the same directory, we simply read the .gif files from there based on the name that was given to us by the server :). So ... we are cheating. To implemet things properly in a real system, we would have to transfer the image from the server to the client. However, in java, **Images** are not **Serializable**. Its a real pain! We could however, read the .gif file, send its bytes one at a time to the client and have the client save the bytes back to the file and then create an **ImageIcon** from it. That would work :).

```
private void updatePictureLabel() {
    AuctionItem item = auctionClient.getLatestAuctionItem();
    if (item != null)
        pictureLabel.setIcon(new ImageIcon(item.getPicture()));
    else
        pictureLabel.setIcon(BLANK_IMAGE);
}
```

Now for the event handlers. When the REGISTER button is pressed, we need to create a new **Customer** object and bring up the **RegistrationDialog** box to get its information. Then send a registration request to the **AuctionServer**:

```
private void handleRegistration() {
    new RegistrationDialog(this,
auctionClient.getCustomer()).setVisible(true);
}

public void dialogFinished() {
    if (auctionClient.register())
        setTitle("Client: " + auctionClient.getCustomer().getName());
    updateStatus(auctionClient.getServerReply().toString());
}
public void dialogCancelled() {} //do nothing
```

When the CATALOG button is pressed, send a request to the server for a catalog of items:

```
private void handleCatalogRequest() {
    auctionClient.sendForCatalog();
    new AuctionCatalogDialog(this,
```

```
(ArrayList<AuctionItem>)auctionClient.getServerReply());
}
```

When the MAKE BID button is pressed, send the bid to the Server:

```
private void  handleMakeNewBid() {
    auctionClient.sendBid(auctionClient.getCustomer().getName(),
bidToMake);
    updateStatus(auctionClient.getServerReply().toString());
}
```

When the timer ticks, send an update request to the server, then update the screen. We also detect changes in the **AuctionItem** so that we can display a nice message. For example, if there was nothing being bid on, then suddenly a new **AuctionItem** comes up for bidding, we display the message `"New Item Up For Bidding"`. If the item was already being bid on, then suddenly becomes **null**, we display a message stating `"Item No Longer Up For Bidding"`. Of course if this client made the last bid, then we should inform him/her that he/she now owns the item with a message such as: `"Antique Table SOLD to you for $100.00"` .

```
private void  handleRequestUpdate() {
    AuctionItem prevItem = auctionClient.getLatestAuctionItem();
    AuctionItem newItem = auctionClient.sendForUpdate();
    if (prevItem == null) {
        if (newItem != null)
            updateStatus("New Item Up For Bidding");
    }
    else if (newItem == null) {
           if (prevItem.getPurchaser().getName().equals(
                    auctionClient.getCustomer().getName()))
                updateStatus(prevItem.getName() + " SOLD to you for $" +
                             prevItem.getBid());
           else
                updateStatus("Item No Longer Up For Bidding");
        }
    else if (!prevItem.getName().equals(newItem.getName())) {
           if (prevItem.getPurchaser().getName().equals(
                    auctionClient.getCustomer().getName()))
                updateStatus(prevItem.getName() + " SOLD to you for $" +
                             prevItem.getBid());
           else
                updateStatus("New Item Up For Bidding");
        }
    update();
}
```

Well that is it! There was a LOT of code for this AuctionSystem. You can always add to it if you want to make a nice application.