

## Chapter 9: Networking

In this chapter we will introduce Java's networking features that will allow us to communicate with other computers connected to the Internet. Java adheres to the Internet standard protocol called TCP/IP that specifies how computers talk to each other over the Internet. Therefore, you can easily create Java programs to interface with existing services available on the Internet, even if those services are provided by programs written in a language other than Java. In fact, you could even write an entire web browser programs similar to Netscape or Internet Explorer in Java and connect to all services offered on the World Wide Web.<sup>1</sup> On the other hand, you could create a web server system in Java as well, using the many build-in network classes to facilitate the creation of that program.

This build-in support for Internet connectivity is perhaps the most exiting part of Java. To be sure, there are a lot of details to take care of before successfully creating a program that connects to other programs via the Internet. But Java provides an organized and flexible set of classes in the `java.net` package that will take care of many details automatically. You are free to concentrate on the task at hand without worrying about the technical details that are handled automatically by the appropriate Java classes. Still, a detailed understanding of how the Internet works is essential for the creation of "net-savvy" programs.

In section 10.1 we will introduce some background information about the Internet and the way it functions as a whole. Section 10.2 will show how to create client programs to connect to existing servers and give several examples. Section 10.3 will describe how to create complete client/server solutions where a server can handle multiple simultaneous clients. Section 10.4, which is optional, will give an extended example of creating a complete and useful chat client/server package. Section 10.5 will conclude the chapter with a brief discussion of some of the security issues you are faced with when creating client/server packages.

This chapter will bring us close to the forefront of current technology. Java does provide additional "hot" features that we will be unable to cover, but the ability to create programs utilizing the Internet will give us the necessary tools to create modern and exciting software that is in line with current demands.

### Quick View

Here is a quick overview of the topics covered in this chapter.

---

<sup>1</sup> Such a browser already exists: it is the "Hot Java" web browser and is written by Sun Microsystems.

**10.1. The Internet**

What's in a Name: IP Names and Numbers; Sockets, Ports, and Protocols; Client/Server Programs; Using Telnet to Connect to Servers

**10.2. Creating Client Programs**

Simple Client Programs; A `WebViewer` Client

**10.3. Creating Server Programs**

A Server for Multiple Simultaneous Clients; (\*) Directory Client/Server System to bypass Applet Restrictions; (\*) Multi-Client Synchronized Server for Online Ordering

**(\*\*) 10.4. The ChatterBox Client/Server Example**

The `ChatterBox` Protocol; The `ChatterBox` Server Classes; The `ChatterBox` Client Class; The `ChatterBox` Client as an Applet

**10.5. Security Issues**

(\*) These sections are optional but recommended

(\*\*) These sections are optional

## 9.1. The Internet

With the Internet's rapid growth during the mid to late 90's the ability of programs to utilize the Internet became increasingly important. In fact, the latest versions of most major application packages such as Microsoft Office now contain build-in "Internet support". In some cases that support consists of accessing online information from within a program. In other cases programs can automatically downloading updates or patches, or even create collaborative work environments where people in different locations can work together on a particular project via the Internet. Using Java we can create programs with similar features.

One of the uses of this technology that will be most interesting to us is that it will allow us to bypass the security restrictions imposed on Java applets: applets can not write data to disk, but they *are* allowed to send data over the network to other computers. An applet we create can, for example, send data to another stand-alone program that is also created by us and that program *can* write data to disk (because as a stand-alone program it does not fall under the applet security restrictions). Hence, we *can* enable Java applets to "write" data to disk, taking a detour over the Internet. Another use of this technology is to create client/server application where one "server" program can provide and synchronize resources to many "client" programs.

Before exploring Java's networking capabilities we need to understand what the Internet is and how it works. It is particularly important to understand from the outset that the Internet is much more than the "World Wide Web" even though today the web is by far the most utilized component of the Internet.

### Definition 10.1.1: The Internet

*The Internet is a (huge) collection of wires and computers that collectively transfer digital information wrapped in individual data packets. In effect, it ties a large collection of computer*

networks together through a mutually agreed upon standard called TCP/IP (Transmission Control Protocol/ Internet Protocol). The computers attached to the Internet fall into four general categories: routers, gateways, servers, and clients.

- Routers are large computers that are directly connected to other routers via high-speed data connections or satellite links. Routers pass data packets along to other routers to bring them closer to their delivery address. Routers form the backbone of the Internet and are "aware" of the topology of parts of the net.
- Gateways are special computers that mediate traffic between local area networks (local collections of servers and clients) and Internet routers.
- Servers are medium-sized computers such as Unix workstations connected to one (or more) router or gateway. Servers provide services to client computers via mutually agreed upon methods of communication called protocols.
- Clients are smaller computers such as individual PC's that are (usually) connected to a router or gateway. Clients send data packets through gateways and routers to servers, requesting certain services via specific protocols.

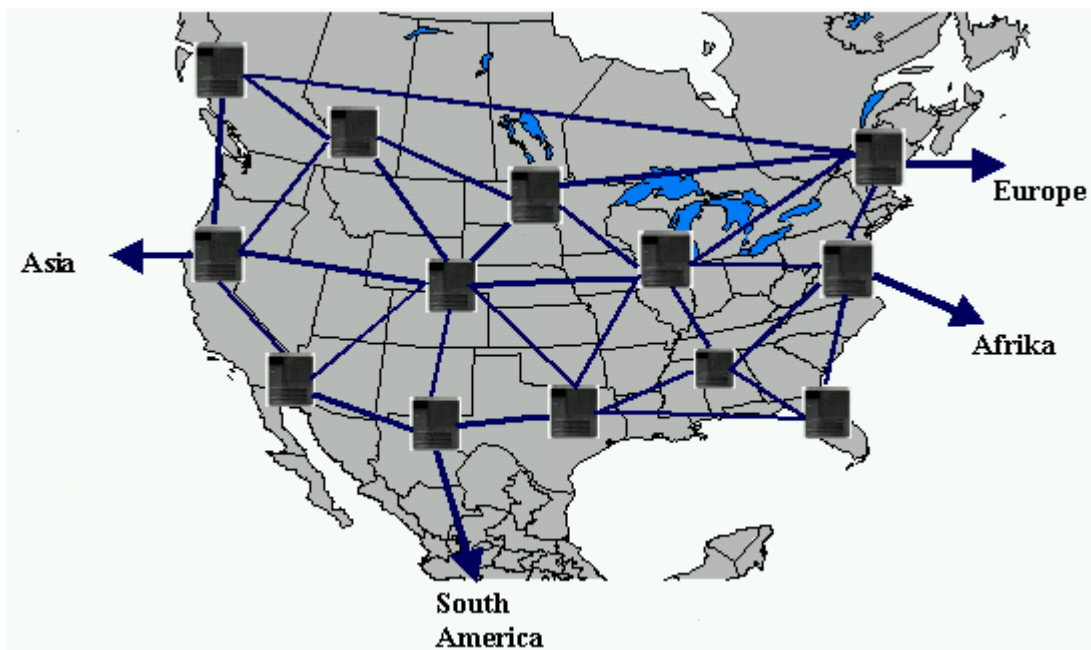


Figure 10.1.1: Interconnected Routers form the backbone of the Internet and span the world

Frequently client computers are connected to other networks in addition to the Internet. Many university campuses and businesses, for example, have their own network that allows users to access common disk space and share printers. The same wires that serve to connect these computers to their own private network also connect the computers to the Internet. That works because data traveling through a network can be encoded in multiple ways. Data that is encoded according to the TCP/IP protocol is "Internet data" and is allowed outside the private network, data that is encoded in other ways remains inside the private network. Such private networks have so-called "gateway computers" that make sure that only data that is specifically for the Internet is forwarded to an Internet router, while other data packets are restricted to the private network. Other computers are connected to the Internet via dialup connections to servers or gateways that in turn forwarding data packets to the nearest Internet router. Since the Internet can utilize existing networks and connect them with each other it is sometimes referred to as the "network of networks".

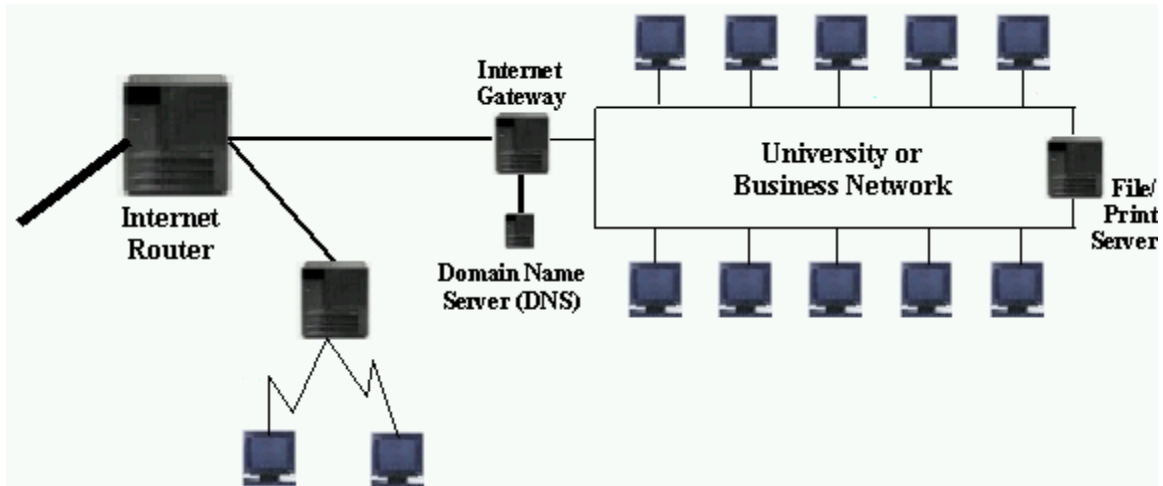


Figure 10.1.2: PC's connected to Internet via Gateway Computer or Modem

The Internet actually derived from another network called ARPANet, created by the Defense Department with taxpayer's money. That network was created in the late 60's to allow military command-and-control facilities to continue to communicate even if some of the nodes of the network were destroyed (by acts of nuclear war, as the plans went). The main feature of this network was that routers could independently redirect data traffic to other nodes if there was a problem with a few nodes between two points of communications. As the thread of war decreased in 70's and 80's, ARPANet was opened up to allow universities to participate in the data traffic. In the 80's businesses joined the network, but it was mainly used by technically and scientifically minded participants who had to be rather skilled to use the net. In the 90's the World Wide Web was created as part of the Internet, and its ease of use and visual impact sparked a huge increase in Internet participants. Today the Internet is mainly run by private companies that continuously add new routing capacities to allow for the huge increases in world-wide data traffic the Internet has experienced in the last years.

## What's in a Name: IP Names and Numbers

The first fundamental problem of the Internet is how to uniquely identify all connected systems. Just as every person who wants to participate in the worldwide telephone system needs a unique phone number, all computers connected to the Internet need a unique classification tag. Every computer that is part of the Internet gets a unique identification number and name. That applies also to computers that only connect temporarily to the Internet via a modem connection.

### Definition 10.1.2: IP Number and Name

*Every computer attached to the Internet is assigned a unique address called IP number. An IP number consists of 4 groups of integers between 0 and 255, separated by dots.<sup>2</sup> Each number also has a unique name associated with it, consisting of multiple strings separated by dots. The last two parts of the name, called domain name, identify a computer as part of the same local network and are associated with the first two groups of numbers. The last part of the name provides some*

<sup>2</sup> Thus the Internet can support a maximum of  $255^4 = 4,228,250,625$  computers. But the first two integers often identify a domain and the domain owner is free to administer their last two integers as they please. Therefore per domain  $255^2 = 65,025$  IP numbers are unavailable even if only 10 of those numbers are used.

*information about the nature of the machine. Every data packet sent over the Internet contains the IP number of the sender as well as the IP number of the intended recipient in addition to any data transmitted.*

*If a computer can not obtain a valid IP number it resorts to the standard default value of 127.0.0.1 with the associated name of localhost. This name can be used by a machine to connect to itself even when it is not connected to the Internet.*

For example, all computers connected to Seton Hall University's campus network are identified by numbers starting with 149.150 and associated names ending in .shu.edu. Thus, Seton Hall has the domain name shu.edu and individual computers have names such as www.shu.edu, pirate.shu.edu, and dhcp-143.shu.edu. The IP numbers associated to these names would look like 149.150.2.1, 149.150.2.6, and 149.150.87.143. A few names and numbers are permanent, i.e. they always refer to the same physical computer. The majority of the names and numbers are associated dynamically and refer to different computers at different times.<sup>3</sup>

Knowing the name of a computer usually reveals something about its origin and purpose, as illustrated in table 10.1.3. For example, names ending in .edu are usually associated with computers at US educational institutions, those ending in .de are reserved for computers physically located in Germany, etc.

IP Name Ending	Usual Meaning
.edu	educational institution in the US
.com	commercial site or network
.net	site associated with the functioning of the Internet
.org	non-profit organization
.mil	military institution in the US
.de, .ca, .fr, .it	sites in Germany (.de), Canada (.ca), France (.fr), and Italy (.it)

Table 10.1.3: Meanings associated with Internet names

Although the IP *numbers* are required to transmit data packets over the Internet, the IP *names* are much easier for users to remember. Therefore, there are special computers available on the Internet that will look up the number associated with a name. These directory assistance computers are called Domain Name Service (DNS) servers and each computer on the Internet must know the IP number of at least one DNS system to resolve Internet names to numbers. This is somewhat analogous to the telephone system: it is much easier for people to remember names rather than phone numbers. As long as you know the phone number of at least one directory assistance service you can usually find out the number of anyone whose name you know.

### Definition 10.1.3: Domain Name Service, or DNS

*Special computers called Domain Name Service (DNS) servers act as "directory assistance" utilities to find the associated IP number for computers referred to by name. That number is required before a computer can send any packets over the net.*

<sup>3</sup> Seton Hall, for example, could support 65,025 computers but in reality connects at most 2-4% of that figure. If that ratio were representative for other domains, a more realistic limit on the number of possible Internet computers is closer to 3% of 255<sup>4</sup> or approximately 130 million machines worldwide.

*Internet names and numbers are maintained, at least at this time, by an organization called INTERNIC ([www.internic.net](http://www.internic.net)). The directory information that the DNS servers maintain is constantly and automatically updated.<sup>4</sup>*

At this point we can piece together a simplified but reasonably accurate scheme of how the Internet works.

**Example 10.1.4:**

Try to describe what happens when a user establishes a dialup connection to "the Internet", starts his or her favorite web browser, and enters a URL to view a particular web page.

**Step 1:** A user turns on a PC: The operating system of that PC starts and loads - among many other things - a program that can handle the TCP/IP (Transmission Control Protocol/ Internet Protocol) protocol.<sup>5</sup> At this time the IP number of the PC is 127.0.0.1 and the associated IP name is localhost (see figure 10.1.4).

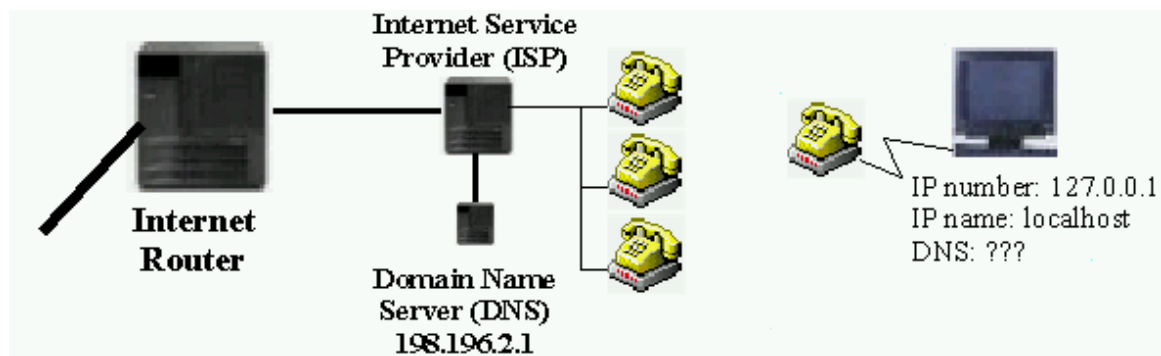


Figure 10.1.4: Individual PC waiting to dialup to the Internet via an ISP

**Step 2:** The user starts a dialup connection to an Internet Service Provider (ISP) to connect to the Internet: The dialup software calls a particular phone number and waits until some computer answers. When a computer answers, the host computer and the local PC exchange authentication information (such as username and password). If successful, the host machine determines a free IP number within its allocated range and sends that to the client PC. The client PC is then known on the Internet via this IP number and the associated IP name, and traffic to and from the Internet is routed through the dialup host machine to the PC. The host computer also sends the client PC a particular IP number that is to be used as the client's DNS server (see figure 10.1.5).

<sup>4</sup> It usually takes a day or more for a new Internet name to trickle through all DNS systems around the world. A new Internet name is universally available only after all DNS systems are updated.

<sup>5</sup> On Windows-based PCs, this program is, essentially and simplified, called "winsock".

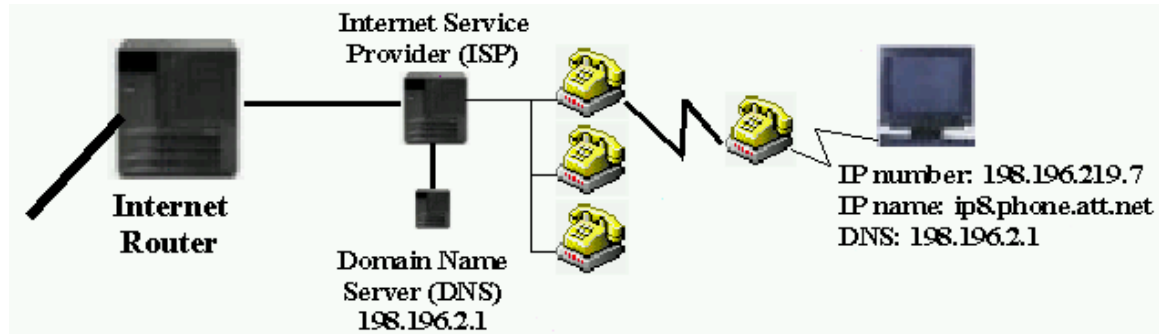


Figure 10.1.5: Individual PC after successful connection to the Internet via an ISP

**Step 3:** A user starts a web browser and enters a particular URL to view, such as `http://www.cat.shu.edu/`. The web browser sends a request to its DNS server (defined during the initial phases of dialing up), asking for the IP number corresponding to the name "www.cat.shu.edu". The DNS server searches its database, inquires at other DNS servers if necessary, and returns the appropriate IP number to the client PC. The client PC then sends a request through its host and attached gateways/routers to load a particular web page. The data packet contains the IP number of the client PC (established during dialup), the IP number of the web server (determined by the DNS server) and some information that spells "please give me web page X"<sup>6</sup>. The data packets are passed around various gateways and routers until it reaches the destination IP number where it is received by a web server. That web server locates the requested web page on its disk, wraps it into one or more Internet data packets, adds in particular its own IP number as sender and the IP number of the original requestor as recipient, and sends the packet on its way. The packet is again transmitted via gateways and routers<sup>7</sup> along to the original requestor. There it is decoded and the web page is displayed for the user to look at (see figure 10.1.6).

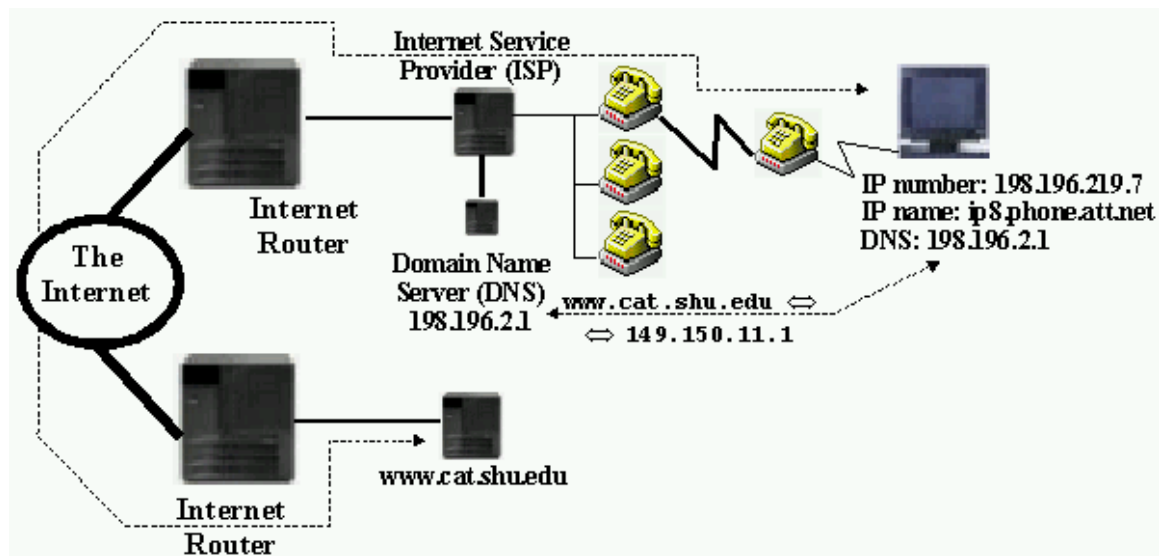


Figure 10.1.6: PC requesting a web page from web server

The only part of this picture that we did not explain in detail is how the web server system (`www.cat.shu.edu` in this example) can understand the request "please give me web page X", and how in turn the PC can decode the answer received. After all, as we saw in chapter 9, if data is

<sup>6</sup> There really is no need to say "please" to a web server ...

<sup>7</sup> The specific path the return packets take may very well be different from the original path.

isolated from structured types inside a program we need to know *something* about the data before we can make sense of it.<sup>8</sup>

In addition, we need to know how exactly data can flow into and out of a system connected to the Internet.

## Sockets, Ports, and Protocols

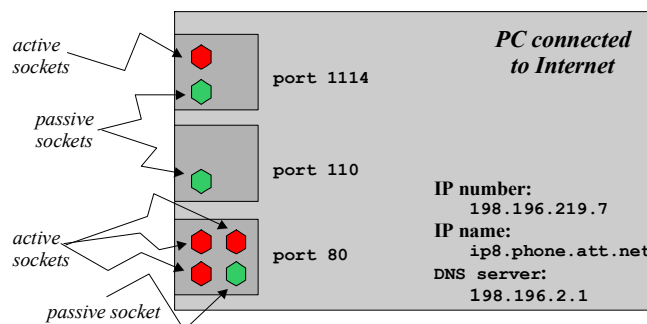
There are two final pieces to the puzzle to finish our basic introduction to the way the Internet works: ports and sockets, which are the physical locations where data is transmitted, and protocols, which are the specifications of how the data is structured:

### Definition 10.1.5: Ports and Sockets

*A port is generally a place where information goes into or out of a computer. Many personal computers, for example, have serial ports, i.e. physical locations to attach input/output devices. Computers attached to a communications network such as the Internet also have ports that are non-physical logical channels identified by a unique port number through which information can flow in and out of the system. Computers on the Internet have over 10,000 logical ports each, where ports with numbers less than 1024 are often reserved for special services.*

*A socket is a network communications endpoint, analogous to a socket to which electrical wires (the network data connection) can be attached. Sockets are associated with a port, and each port can have one passive socket awaiting incoming connections as well as multiple active sockets corresponding to an open connection on that port. An active socket on one machine is connected to a remote active socket on another machine via an open data connection. Closing the connection destroys the active sockets at each endpoint. A passive socket is not connected, but rather awaits an incoming connection, which will spawn a new active socket.*

In other words, when a computer is connected to the Internet, information flows in and out of logical ports identified by number. Each port has a socket associated with it that is initially passive. When a connection is requested to a particular port, the passive socket spawns an active socket on this port that will then serve as one endpoint of the data connection. The remote computer initiating the connection also has an active socket on a particular port, serving as the other endpoint of the data link. Data can then be transmitted through these ports via the active sockets.



<sup>8</sup> Compare example 9.2.14 where we saved two `int` values to disk but retrieved the data as a `double`, which was for all intent and purposes a valid but random `double`.

Figure 10.1.7: Schematic computer with ports and sockets

Now that we know how exactly data gets in and out of a computer (more or less at least), we need to specify what type of data exactly is transmitted, how it is interpreted, and how requests are made to initiate any data transfer. That is specified via protocols, i.e. formal descriptions of rules and message formats that machines must follow to exchange data.

### Definition 10.1.6: Internet Protocols

*A protocol is a mutually agreed upon method of communication between two or more parties that clarifies exactly what type of data exchange will take place and how to initiate it. Many well-known protocols exist for the Internet that govern complex services between two or more computers. Some of the important ones are listed in table 10.1.8 together with their default port numbers, including:*

- `http`: *Hypertext Transfer Protocol* - specifies how to request and transmit web pages
- `ftp`: *File Transfer Protocol* - specifies how to initiate and conduct file transfers
- `smtp`: *Simple Mail Transfer Protocol* - specifies how to send electronic mail
- `pop`: *Post Office Protocol* - specifies how to retrieve and manipulate electronic mail

*Protocols are specified in so-called Requests for Comments (RFC) and are available in written form on the Internet. Protocols are implemented via programming code to perform the data exchange according to the rules and formats of the particular protocol.*

Protocol	Meaning	Standard Port
<code>echo</code>	echoes everything back	7
<code>discard</code>	discards everything	9
<code>daytime</code>	sends date and time	13
<code>ftp</code>	file transfer protocol	21
<code>telnet</code>	terminal protocol	23
<code>smtp</code>	simple mail transfer protocol	25
<code>finger</code>	displays user information	79
<code>http</code>	hypertext transfer protocol	80
<code>pop3</code>	post office protocol (version 3)	110
<code>nntp</code>	network news, or Usenet	119
<code>imap</code>	manages email on server	143
<code>talk</code>	talks to another user	517
<code>kerberos</code>	security services	750

Table 10.1.8: Standard Internet protocol names and associated ports

These protocols are very strict and are discussed and refined for quite some time. Once implemented they are strictly adhered to until a refined version of the protocol is discussed, established, and implemented.<sup>9</sup> Tables 10.1.9 to 10.1.12 show some sample exchanges in various protocols.

Protocol	Default Port	Sample Exchange
<code>http</code>	80	<pre> =&gt; GET / HTTP/1.0 =&gt; ↵ &lt;= HTTP/1.0 200 OK </pre>

<sup>9</sup> The `http` protocol, for example, was recently upgraded from version 1.0 to version 1.1, which means that all web servers and web browsers had to be rewritten to adhere the new `http 1.1` protocol.

		<pre> &lt; Date: Wed, 18 Mar 1999 20:18:59 GMT &lt; Server: Apache/1.0.0 &lt; Content-type: text/html &lt; Content-length: 1579 &lt; Last-modified: Mon, 22 Jan 1999 22:23:34 GMT &lt; &lt; HTML document ... </pre>
--	--	--

Table 10.1.9: Sample client/server exchange using http protocol to request a web page

Protocol	Default Port	Sample Exchange
smtp	25	<pre> &lt; 220 kitten Simple Mail Transfer Service Ready =&gt; HELO mycomputer.mydomain &lt; 250 kitten.shu.edu =&gt; MAIL FROM:&lt;Smith@shu.edu&gt; &lt; 250 OK =&gt; RCPT TO:&lt;Jones@shu.edu&gt; &lt; 250 OK =&gt; DATA &lt; 354 Start mail input; end with &lt;CRLF&gt;.&lt;CRLF&gt; =&gt; Blah blah blah... =&gt; . &lt; 250 OK </pre>

Table 10.1.10: Sample client/server exchange using smtp protocol to send an email message

Protocol	Default Port	Sample Exchange
pop	110	<pre> &lt; +OK POP3 server ready &lt; &lt;1896.697170952@dbc.mtview.ca.us&gt; =&gt; APOP mrose c4c9334bac560ecc979e58001b3e22fb &lt; +OK mrose's maildrop has 2 messages (320 octets) =&gt; LIST &lt; +OK 2 messages (320 octets) &lt; 1 120 &lt; 2 200 &lt; . =&gt; RETR 1 &lt; +OK 120 octets &lt; &lt;the POP3 server sends message 1&gt; &lt; . =&gt; QUIT &lt; +OK dewey POP3 server signing off (maildrop empty) </pre>

Table 10.1.11: Sample client/server exchange using pop protocol to request an email message

Protocol	Default Port	Sample Exchange
echo	7	<pre> =&gt; hello &lt; hello =&gt; Anybody there? &lt; Anybody there? =&gt; This protocol is pretty easy. &lt; This protocol is pretty easy. </pre>

Table 10.1.12: Sample client/server exchange using the echo protocol

## Client/Server Programs

In tables 10.1.9 to 10.1.12 data is exchanged according to the specified protocol, assuming a connection could be established on the default port. Two systems are participating: the system initiating the connection is called client, the system answering the request is called server. Data prefaced by the right-arrow symbol  $\Rightarrow$  is sent from the client to the server, data prefaced by the left-arrow symbol  $\Leftarrow$  is sent from the server to the client. Such a client/server model, in fact, is the underlying software model that governs the Internet.

### Definition 10.1.7: Server Program

*A server program is a stand-alone computer program that often has access to a large set of data or other resources and runs in the background on computers permanently connected to the Internet<sup>10</sup>. It has a minimal user interface and operates, for the most part, without user intervention. Servers most often "listen" on specified ports for a network connection, and once a connection is established they communicate with the requestor using a mutually agreed upon protocol. One server program can usually handle requests by multiple clients simultaneously.<sup>11</sup>*

All major protocols that are in use on the Internet are implemented in particular server programs.

- A web server program, usually called `httpd`, listens by default on port 80 for incoming connections and delivers web pages when requested. It implements the `http` protocol.
- A pop server program, usually called `popd`, listens on port 110 and is ready to deliver email to a user when asked. It implements the `pop` protocol
- The program most frequently used to deliver email between computers is called `sendmail`. It listens on port 25 and implements the `smtp` protocol
- Other frequently used server programs are `ftpd`, listening on port 21, and `telnetd`, listening on port 23.
- A server that may not appear very useful but can be handy for testing purposes is the `echod` server, implementing the `echo` protocol. That server listens on port 7 by default.
- Another server program that can be quite useful is called `msqld` and listens on port 1114 by default. It understands requests in Structured Query Language (SQL) but the actual protocol is more involved than SQL.








	<b>msqld</b> (port 1114)	<b>Unix Server</b>  <b>IP number:</b> <b>149.150.2.6</b>  <b>IP name:</b> <b>pirate.shu.edu</b>  <b>DNS server:</b> <b>149.150.2.1</b>
	<b>popd</b> (port 110)	
	<b>httpd</b> (port 80)	
	<b>sendmail</b> (port 25)	
	<b>telnetd</b> (port 23)	
	<b>ftpd</b> (port 21)	
	<b>echod</b> (port 7)	

Figure 10.1.13: Typical server programs running on a typical Unix system

<sup>10</sup> Server programs can be started on any computer, even on a `localhost` not connected to the Internet. However, in order to connect to a server the client program needs to know the IP name of the server. Therefore that name should not change and the server system should be connected at all times.

<sup>11</sup> Of course multiple requests are not handled truly simultaneously unless the system that runs the server has more than one processor.

The counter-part to server programs are client programs. Those are the programs that the user interacts with and that in turn interact with an appropriate server program who speaks the same protocol as the client.

### Definition 10.1.8: Client Program

*A client program is a program with an extensive user interface or other visual capabilities, but with little access to resources. A client program communicates with a server program via an Internet connection on a specific port to request information or cause action. It presents the data delivered by the server in an appropriate way or displays appropriate messages regarding the actions caused on the server. In many cases multiple client programs can be serviced by one server program. The client must use the same protocol as the server in order to communicate effectively.*

There are many examples of client programs and most modern programs these days include at least a client component:

- Netscape Communicator and Internet Explorer are both web client programs. They both understand the `http` protocol and interface usually with a web server program such as `httpd`.
- Chat client programs are commonly used to communicate with several people at the same time, much like a telephone party line. Client programs interface with appropriate chat servers who negotiate the communication between the participating chat clients.
- Email programs such as Netscape Messenger, Microsoft Outlook, or Eudora are client programs that understand the `pop` protocol or, more and more frequently, another email standard protocol called `imap`. They interface with appropriate `pop` or `imap` servers to retrieve email messages for the user. Most also interface with an `smtp` client program to deliver outgoing email.
- Anti-Virus protection programs are *not* client programs but they can often download updated anti-virus definitions from the Internet. They do that by invoking a client component that interfaces with an appropriate server program to obtain the new information. The protocol used is usually proprietary.<sup>12</sup>

Now we have spent a lot of time explaining some of the inner workings of the Internet. It is high time for some examples. Before we actually write our own Java client and server programs, we will take a hands-on look at some existing servers using a standard universal client program called telnet.

### Using Telnet to Connect to Servers

Telnet is a terminal emulation and communications program that allows you to use a remote computer for text-based applications as if you were sitting in front of it. It is most frequently used to interface with a `telnetd` server program using a specific protocol. However, telnet is very versatile and can be used to interface with many, in fact most, other text-based server programs. Since it is

---

<sup>12</sup> It should be secure enough to prevent a virus program from infecting the downloaded information.

automatically installed on Windows 95/98/NT computers as well as all Unix systems, it is an ideal starting point to explore various servers.<sup>13</sup>

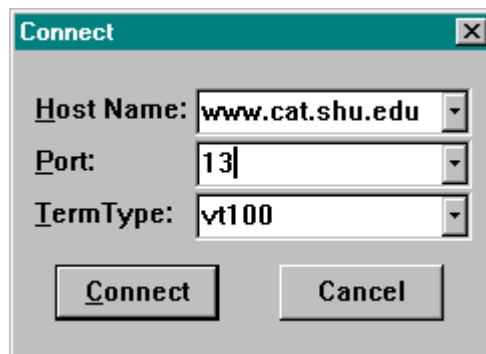
### ***Example 10.1.9:***

The `daytime` protocol (see table 10.1.8) is a particularly simple protocol. It specifies that the server should send the date and time of the server machine to any client connecting to it. Use `telnet` to connect to the standard `daytime` port 13 of a machine such as `www.cat.shu.edu` and describe what happens.

Telnet, by default, tries to connect to a `telnetd` server on port 23, but it can also connect to any other port and attempt to communicate with a server listening on that port. To start a telnet connection to port 13 on the machine `www.cat.shu.edu` using a Unix system you type:

```
telnet www.cat.shu.edu 13
```

On Windows systems, you should open the `C:\Windows` folder and double-click on the program `telnet.exe`. Select `Connect | Remote System ...` and enter the desired host name and port in the dialog box that pops up:



*Figure 10.1.14: Establishing a Windows telnet connection to `www.cat.shu.edu` on port 13*

On Macintosh systems, the procedure is similar. Of course the system you are using should be connected to the Internet prior to initiating a connection. If everything works, you will see a response from the `daytime` server such as the one shown in figure 10.1.15.



*Figure 10.1.15: Typical response from a `daytime` server*

The `daytime` server will deliver the current date and time of its system, which may be different from the current time of the client system.<sup>14</sup> After sending the information, the server immediately closes the connection.

<sup>13</sup> Telnet programs are readily available for free for Macintosh systems (see appendix).

<sup>14</sup> If there is no `daytime` server listening on port 13 of the server system you will get a message saying "request denied" or something similar. You can try any other machine name you know such as `www.sun.com`, `www.ibm.com`, `www.mit.edu`, etc. until you find a `daytime` server that will answer.



Telnet can also be used to connect to servers that speak a more complex protocol such as `http`. It will not make any attempt to interpret the response sent by a server in any way, it simply displays it "as is".

**Example 10.1.10:**

Review the sample session using the `http` protocol and an `httpd` server shown in table 10.1.9, then use `Telnet` (not `Netscape` or `Internet Explorer`) to try to obtain a web page from the web server running on the machine called `www.shu.edu` or from another system such as `www.yahoo.com`. Explain in detail what is happening when you are successful.

First of all, we are trying to connect to another computer on the Internet, so we must be connected to the Internet either via a dialup connection or another network connection. Once your connection to the Internet is established, you can start the `telnet` program to connect to the desired host as in example 10.1.10. Note, however, that based on our previous discussion we now need to connect to port 80 on the specific host, because `httpd` web servers are listening on port 80. To start the `telnet` connection on Unix systems, you type:

```
telnet www.shu.edu 80
```

to connect to the machine called `www.shu.edu` on port 80. On Windows systems, you open the `C:\Windows` folder and double-click on the program `telnet.exe`. Select `Connect | Remote System ...` and enter the desired host name and port in the dialog box that pops up:

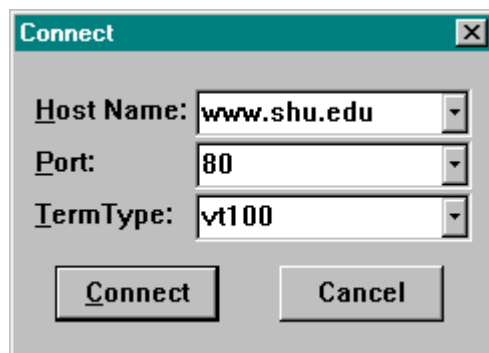


Figure 10.1.16: Establishing a telnet connection to `www.shu.edu` on port 80

On Macintosh systems, the procedure is similar. If you are indeed properly connected to the Internet, you will soon see a prompt informing you that you are connected. However, nothing else will happen. The web server program is now waiting for you - or rather the client program - to issue an appropriate request for a web page as outlined in the `http` protocol.<sup>15</sup> In table 10.1.9 we saw a sample session with an `httpd` server so we know that the appropriate commands to request a web page are:

```
GET / HTTP/1.0
└
```

<sup>15</sup> The web server does not know that a non-web browser has connected as a client.

In other words, type "GET / HTTP/1.0" as indicated, hit ENTER - nothing happens - and then hit ENTER again.<sup>16</sup> Now you have issued the simplest command from the http protocol that the server understands and you will see a "web page" similar to the one shown in figure 10.1.17:

```
Telnet - www.yahoo.com
Connect Edit Terminal Help
er=0 cellspacing=0 cellpadding=0<tr><td height=2></td></tr></table></td></tr></table><table border=0 cellspacing=6 cellpadding=0><tr><td align=right><a href=r/us><small>Yahoo! prefers</small></a></td><td><a href=r/us><img width=37 height=3 border=0 src=http://a1.g.a.yimg.com/7/1/31/000/us.yimg.com/a/ui/visa/sm.gif></td></tr></table><small><a href=r/ad>How to Suggest a Site</a> - <a href=Company Info</a> - <a href=r/pv>Privacy Policy</a> - <a href=r/ts>Terms of Se
</a> - <a href=r/cb>Contributors</a> - <a href=r/hr>Openings at Yahoo!</a><p>ight &copy; 1999 Yahoo! Inc. All rights reserved.<br><a href=r/cy>Copyright
cy</a></small></center></body></html>
```

Figure 10.1.17: Resulting data transmitted when requesting a web page via telnet

Of course this data does not look the way it would when requesting the same page with a web browser. A web browser, in addition to being able to speak http with a web server, also understands the Hypertext Markup Language HTML, which consists of formatting commands enclosed in angular brackets. A web client applies appropriate formatting mechanisms instead of simply displaying the HTML tags as our telnet program does. However, we have achieved our goal: we did get the desired web page from the web server even if it did not look very pretty.

Here is what happens in detail:

- *You establish a dialup connection if necessary:* your Internet service provider assigns an IP name and number to your computer and tells your computer with DNS server to use.
- *You establish a telnet connection to www.shu.edu on port 80:* First, a request is transmitted to the DNS server to find the IP number of the host named www.shu.edu. Once the DNS server supplies that number, a data packet including this number, your own IP number, and some additional information that says "establish connection with port 80" is sent to your dialup server. That server passes the packet along to routers, who in turn deliver it, eventually, to the machine with the IP number associated with www.shu.edu. That machine looks at the packet and opens a connection to port 80. A passive socket on that port spawns an active socket that serves as the termination point of your connection. A connection is now established from a socket on some local port of your system to the socket on port 80 on www.shu.edu.
- *You type GET / HTTP/1.0:* An appropriate data packet is sent through the Internet to the receiving active socket on www.shu.edu. There the web server listening to port 80 picks up the packet and reads your command. It sees "GET", which instructs it to look for a web page. It sees "/" (a single slash), which means to get the top-level web page, which has the actual URL http://www.shu.edu/index.html. It sees "HTTP/1.0", meaning that it is to use version 1.0 of the http protocol. Nothing happens yet, because the web server is waiting for other options that you may want to specify in accordance with the http 1.0 protocol.

<sup>16</sup> Depending on how telnet is configured you may or may not see what you are typing. Even if you do not see anything, type the commands as indicated; they will be sent to the server whether you see them or not.

- *You press ENTER, which sends an empty line to the web server:* That tells the web server that your command is finished. The web server now retrieves the appropriate web page from its hard disk, adds some header information to the page, wraps the data into one or more packets including the IP number of the recipient, and sends the packet back to your computer. There the telnet program receives the information and displays it on the screen.<sup>17</sup>
  - The web server now closes the connection and is ready to service another client request or your telnet client again, should you so choose.
- 

It should be noted that the above scenario is the same regardless of which particular system you are using and independent of the system where the web server program is running. The point of adhering to a specific protocol is that you can implement programs that speak that protocol on pretty much any system you choose (as long as it can establish a TCP/IP Internet connection). This, of course, makes client/server programming in Java particularly attractive: not only is the protocol system independent, but also the actual Java classes are platform independent so that one client or server program can run on pretty much any platform without having to recompile anything. This is one of the reasons why Java is frequently called the "programming language of the Internet".

Now - finally - we are reading to create our own Java client programs in the next section, followed by our own Java-based client/server packages in section 10.3. In fact, after all of this discussion illustrating the rather complex nature of communications on the Internet it will turn out to be surprisingly easy to create Java client and server programs.

## 9.2. Creating Client Programs

Creating client programs in Java is based on two classes in the `java.net` package, `java.net.URL` and `java.net.Socket`. We have already defined the `URL` class in definition 9.5.6 so we need to define the `Socket` class next.

### Definition 10.2.1: The `Socket` Class

*This class is part of the `java.net` package and implements a client socket.<sup>18</sup> Constructing a new `Socket` instance will turn a passive socket on a port into an active socket if a connection to the specified port on the host can be established. The `Socket` class can return input streams to receive information from the host and output streams to send information to the host. The Java API defines `Socket` as follows:*

```
public class Socket extends Object
{ // selected constructor
    public Socket(String host, int port)
        throws UnknownHostException, IOException
    // selected methods
```

<sup>17</sup> A true web client such as Netscape would interpret the response and show you the information formatted according to the HTML formatting tags included in the response.

<sup>18</sup> Recall that sockets are the terminal points for network connections and exist on a port. See definition 10.1.5 for details.

```
public InetAddress getInetAddress()19
public InetAddress getLocalAddress()
public int getPort()
public int getLocalPort()
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
public void close() throws IOException
}
```

The methods `getInetAddress` and `getLocalAddress` refer to the `InetAddress` class, which represents an Internet address in Java.

### Definition 10.2.2: The `InetAddress` Class

*This class is part of the `java.net` package and represents an IP address. The Java API defines `InetAddress` as follows:*

```
public final class InetAddress extends Object implements Serializable
{ // selected methods
    public String getHostName()20
    public String getAddress()
    public boolean equals(Object obj)
}
```

Now we are ready to create our own client programs. We will start with simple clients connecting to servers we have already seen when using the multi-purpose telnet client program in examples 10.1.9 and 10.1.10.

## Simple Client Programs

### Example 10.2.3:

Create a stand-alone Java program to connect to a web server on port 80. The program should display the IP number of the host and of the client, as well as the port used on the host and on the client. The host name should be entered as command line parameter. After displaying the connection information, the program should disconnect.

To establish a connection, all we have to do is instantiate a proper `Socket` object. We can then use the various methods of the `Socket` and `InetAddress` class to display the desired information. Our class will only contain a simple standard `main` method as follows:

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

public class WebConnect21
{ public static void main(String args[])
```

<sup>19</sup> Calls to `getInetAddress` and `getLocalAddress` may result in a call to a DNS server and take additional time.

<sup>20</sup> Calling the `getHostName` method may result in a call to a DNS server to resolve an IP number. That takes additional time so calling this method will delay a program.

<sup>21</sup> You should comment out the calls to `getLocalAddress` and `getInetAddress`, then run the program again to see the performance hit you take when using these methods.

```

    { try
      { Socket connection = new Socket(args[0], 80);
        System.out.println("Connection established:");
        System.out.println("Local Connection Information:");
        System.out.println("\t address:" +connection.getLocalAddress());
        System.out.println("\t port:" + connection.getLocalPort());
        System.out.println("Remote Connection Information:");
        System.out.println("\t address:" +connection.getInetAddress());
        System.out.println("\t port:" + connection.getPort());
        connection.close();
      }
      catch(UnknownHostException uhe)22
      { System.err.println("Unknown host: " + args[0]); }
      catch(IOException ioe)
      { System.err.println("IOException: " + ioe); }
    }
  }
}

```

We can use this program to establish a connection on port 80 to Yahoo ([www.yahoo.com](http://www.yahoo.com)), as shown in figure 10.2.1.

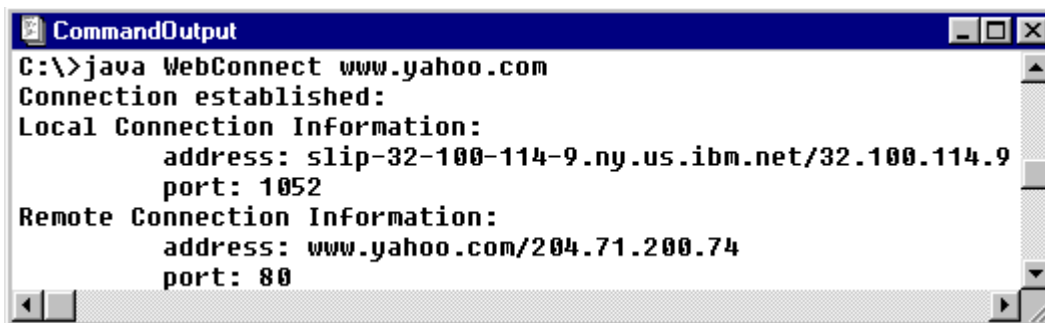


Figure 10.2.1: Using WebConnect to connect to [www.yahoo.com](http://www.yahoo.com)

You can see that the computer that runs the above program has been assigned the IP number 32.100.114.9 and the corresponding name [slip-32-100-114-9.ny.us.ibm.net](http://slip-32-100-114-9.ny.us.ibm.net), which indicates a dialup connection. The computer we have successfully connected to is called [www.yahoo.com](http://www.yahoo.com) and has the IP number 204.71.200.74. The port used on the remote computer is 80 as specified, but on our local computer a “random” port with number 1052 was used to contain the socket.<sup>23</sup>

This class is exceedingly simple even though it really accomplishes quite a lot. The complexities of establishing the connection is handled automatically by the `Socket` class. It is just as simple to receive actual information from the server as the next example will show.

#### ***Example 10.2.4***

Create a program that connects to a daytime server of an Internet system and prints out the date and time received from that server.

Before we start we need to identify a system on the Internet where a daytime server is running. In example 10.1.9 we used `telnet` to connect to a daytime server. Whatever IP name worked for that example should also work fine here.

<sup>22</sup> Note that `UnknownHostException` is a subclass of `IOException`. Therefore it should be caught first so that the `IOException` will not absorb the `UnknownHostException`.

<sup>23</sup> When you run this program, your local information will of course be slightly different.

Our class will establish a connection similar to example 10.2.3. If a connection can be established, we setup an input stream to receive information from the server using the `getInputStream` method of a `Socket`. But a daytime server sends plain text, so that we need a character input stream whereas `getInputStream` returns a byte-level `InputStream`. Therefore we convert the `InputStream` using an `InputStreamReader` to a character-level stream as defined in definition 9.5.9. Then we use the `BufferedReader` class from definition 9.3.1 to have access to a `readLine` method to read the single line of text. Here is the code:

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

public class GetDayTime
{
    public static void main(String args[])
    {
        String host = args[0];
        try
        {
            Socket connection = new Socket(host, 13);
            System.out.println("Connection established:");
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    connection.getInputStream()));
            String daytime = in.readLine();
            System.out.println("DayTime received: " + daytime);
            connection.close();
        }
        catch(UnknownHostException uhe)
        {
            System.err.println("Host not found: " + uhe);
        }
        catch(IOException ioe)
        {
            System.err.println("Error: " + ioe);
        }
    }
}
```

We can now use this class to get the time of various systems on the Internet as long as they have a daytime server listening on port 13. Figure 10.2.2 shows the date and time obtained from two systems.



```
CommandOutput *
C:\Java\10\10.2.4>java GetDayTime www.cat.shu.edu
Connection established:
DayTime received: Tue Sep 7 09:38:54 1999

C:\Java\10\10.2.4>java GetDayTime www.ibm.com
Connection established:
DayTime received: Tue Sep 7 13:38:53 1999
```

Figure 10.2.2: Results of using `GetDayTime` for two daytime servers

As it turns out, sending data to a server is just as easy. In our next client example we will setup both an input and an output stream so that we can send as well as receive data.

### ***Example 10.2.5:***

Create a Java client that connects to an `echo` server on port 7 of some system. It should implement the `echo` protocol, which is particularly easy: an `echo` server waits for

incoming data and returns whatever data it received through the network connection to the client.<sup>24</sup>

This time we need to establish a connection to port 7 because standard `echo` servers listen on that port. The code to connect to the server is similar to example 10.2.4 but we also need to create a proper input and output stream. The input stream is used to send information to the `echo` server, the output stream will receive the information from the server. The `echo` protocol is simple:

- every line sent is immediately echoed back

That means for our program that every “send” should be followed by a “read” until the user closes the connection. The actual code starts out as before to establish the connection. Then we create a new `DataInputStream` using the `InputStream` returned by the socket’s `getInputStream` method and a new `DataOutputStream` using the `OutputStream` returned by the `getOutputStream` method.

We will use the `Console` class from section 2.4, example 5, to ask the user for input. Then we send each input string to the `echo` server and wait for the answer. Once the answer has been received we continue the process until the `echo` server echoes `.QUIT`. Here is the code:

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

public class EchoClient
{   public static void main(String args[])
    {   try
        {   Socket connection = new Socket(args[0], 7);
            System.out.println("Connection established:");
            DataInputStream in = new DataInputStream(
                connection.getInputStream());
            DataOutputStream out = new DataOutputStream(
                connection.getOutputStream());
            String line = new String("");
            while (!line.toUpperCase().equals(".QUIT"))
            {   System.out.print("Enter string: ");
                line = Console.readString();
                System.out.println("\tSending string to server ... ");
                out.writeUTF(line);
                System.out.println("\tWaiting for server response ...");
                line = in.readUTF();
                System.out.println("Received: " + line);
            }
            in.close();
            out.close();
            connection.close();
        }
        catch(UnknownHostException uhe)
        {   System.err.println("Unknown host: " + args[0]); }
        catch(IOException ioe)
        {   System.err.println("IOException: " + ioe); }
    }
}
```

Figure 10.2.3 shows our `EchoClient` in action, communicating with an `echo` server over the Internet.

---

<sup>24</sup> Many Unix systems on the Internet do run an `echo` server and the majority of web servers are Unix systems. Therefore, chances are high that you can simply use the name of a web server system to try to connect to the `echo` port. If you can not find a host that works, try using the name `www.shu.edu`.

```

CommandOutput *
C:\temp>java EchoClient www.shu.edu
Connection established:
Enter string: Anybody out there?
Sending string to server ...
Waiting for server response ...
Received: Anybody out there?
Enter string: I understand this, next example!
Sending string to server ...
Waiting for server response ...
Received: I understand this, next example!
Enter string: .quit
Sending string to server ...
Waiting for server response ...
Received: .quit

```

Figure 10.2.3: The EchoClient class in action

Note that it really does not matter what type of input and output stream we use as long as they are compatible with each other. After all, the `echo` server will faithfully echo whatever we send it, either byte or character based information. In the next example we will use a character-based output stream to send text to an `smtp` server, which means that we need to convert an `OutputStream` to a `Writer`.

### Definition 10.2.6: The `OutputStreamWriter` Class

*An `OutputStreamWriter` is a bridge from byte streams to character streams: It writes characters to a stream, translating them to bytes. It uses the platform's default encoding, but another encoding may be specified. The Java API defines `OutputStreamWriter` as follows:*

```

public class OutputStreamWriter extends Writer
{ // selected constructor
    public OutputStreamWriter(OutputStream out)
}

```

*As with a `FileWriter`, an `OutputStreamWrite` can be wrapped inside a `BufferedWriter` for better efficiency.*

Now we can construct our next example, a class that sends electronic mail through an `smtp` server.

### Example 10.2.7:

Review the sample session listed in table 10.1.10 describing parts of the `smtp` protocol. Then create a class that can send email to a specified user, using an `smtp` server of some Internet system.<sup>25</sup>

According to table 10.1.10 it should be easy to send email via an `smtp` server. We will use the following scheme:

<sup>25</sup> Most Unix systems run an `smtp` server, but not all `smtp` servers will deliver email from foreign systems. The safest bet is the `smtp` server provided by your Internet Service Provider.

- define the `smtp` server to use via command line parameter
- bring up a frame where the user can enter sender, recipient, and message
- when the user clicks on `Send`, establish a connection to the `smtp` server
- define character input and output stream through the connection socket
- receive identification message from server
- send `HELO` `computername.domain` information, receive answer from server
- send `MAIL FROM:<yourname@youremail.domain>`, receive answer from server
- send `RCPT TO:<recipient@address.domain>`, receive answer from server
- send `DATA`, receive answer from server
- send actual message followed by a single line with a period, receive answer from server
- close all streams and disconnect

Based on that scheme it is easy to create the class as follows:

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

public class MailMessage extends Frame implements ActionListener
{
    private TextField sender      = new TextField("wachsmut@shu.edu", 50);
    private TextField recipient  = new TextField(50);
    private TextArea message    = new TextArea(6, 50);
    private Button   send       = new Button("Send Message");
    private Button   cancel     = new Button("Cancel Message");
    private String   host       = null;
    private BufferedReader in    = null;
    private PrintWriter out     = null;
    public MailMessage(String _host)
    {
        super("Mail Message");
        host = _host;
        Panel labels = new Panel(new GridLayout(2,1));
        labels.add(new Label("Sender: "));
        labels.add(new Label("Recipient: "));
        Panel fields = new Panel(new GridLayout(2,1));
        fields.add(sender); fields.add(recipient);
        Panel head = new Panel(new BorderLayout());
        head.add("West", labels); head.add("Center", fields);
        Panel buttons = new Panel(new FlowLayout());
        buttons.add(send); buttons.add(cancel);
        setLayout(new BorderLayout());
        add("North", head); add("Center", message); add("South", buttons);
        send.addActionListener(this); cancel.addActionListener(this);
        validate(); pack(); show();
    }
    private void cancel()
    {
        System.exit(0);
    }
    private void sendWithAnswer(String msg) throws IOException
    {
        out.println(msg); flush();26
        System.out.println("Server: " + in.readLine());
    }
    private void send()
    {
        try
        {
            Socket connection = new Socket(host, 25);
```

---

<sup>26</sup> When sending text through a buffered output stream you need to call `flush` to ensure that data is actually sent through the stream. If you do not call `flush`, data will be sent when the buffer is full, which means that the server will probably never receive the command you want to send.

```

        in = new BufferedReader(new InputStreamReader(
            connection.getInputStream()));
        out = new PrintWriter(new OutputStreamWriter(
            connection.getOutputStream()));
        System.out.println("Server says: " + in.readLine());
        sendWithAnswer("HELO my.system.com");
        sendWithAnswer("MAIL FROM:<" + sender.getText().trim() + ">");
        sendWithAnswer("RCPT TO:<" + recipient.getText().trim() + ">");
        sendWithAnswer("DATA");
        out.println(message.getText()); out.println(".");
        System.out.println("Server says: " + in.readLine());
        in.close(); out.close(); connection.close();
        System.out.println("Message sent. Quitting.");
    }
    catch(UnknownHostException uhe)
    { System.err.println("Unknown host: " + uhe); }
    catch(IOException ioe)
    { System.err.println("Error: " + ioe); }
    finally
    { System.exit(0); }
}
public void actionPerformed(ActionEvent ae)
{ if (ae.getSource() == send)
    send();
  else if (ae.getSource() == cancel)
    cancel();
}
public static void main(String args[])
{ MailMessage mm = new MailMessage(args[0]); }
}

```

This is a very simple exchange with an `smtp` server. For one thing we assume that the messages returned by the server do not contain any error message. But the program will work, assuming you can find an `smtp` server that is willing to transfer your message.<sup>27</sup> Figures 10.2.4 and 10.2.5 show an email message sent by our class and the messages that the server returned.

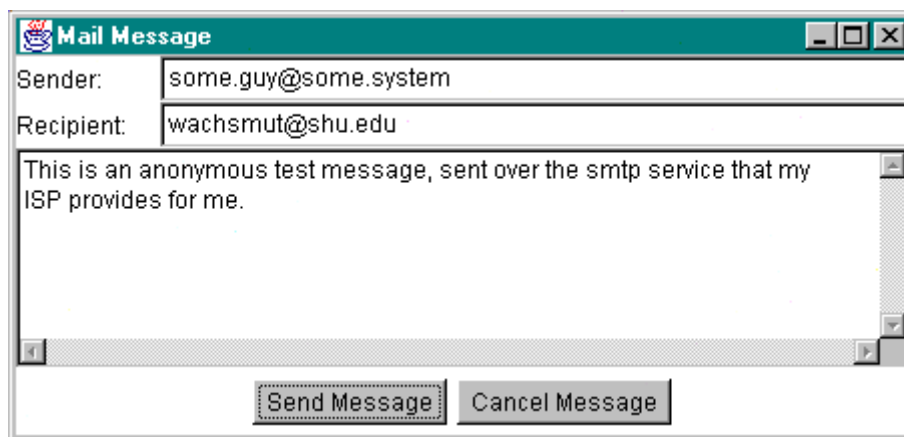
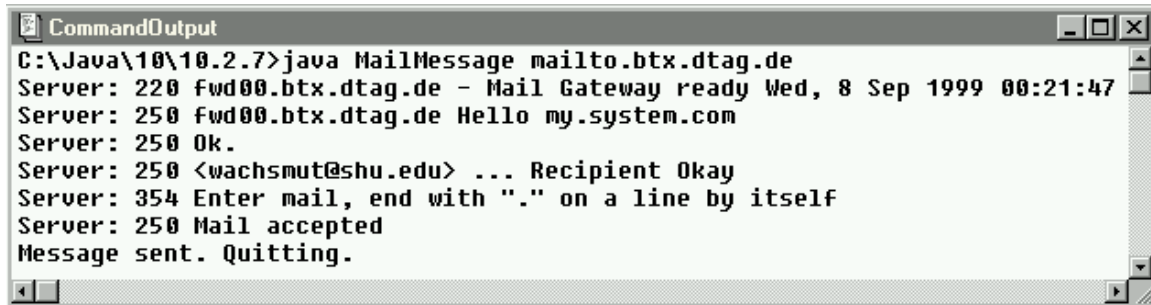


Figure 10.2.4: Message entered into `MailMessage` frame

<sup>27</sup> Using this program you could use `president@whitehouse.gov` as sender and attempt to send your message via the `smtp` server of the machine `whitehouse.gov`. If that `smtp` server did transfer your message, it would look authentic enough to puzzle a good many people. Therefore, many `smtp` servers do not transfer mail from systems outside their own domain. In addition, `smtp` servers often keep a usage log file. It is *not advisable* to attempt to use `whitehouse.gov`.

A screenshot of a Windows-style window titled "CommandOutput". The window contains the following text:

```
C:\Java\10\10.2.7>java MailMessage mailto.btx.dtag.de
Server: 220 fwd00.btx.dtag.de - Mail Gateway ready Wed, 8 Sep 1999 00:21:47
Server: 250 fwd00.btx.dtag.de Hello my.system.com
Server: 250 Ok.
Server: 250 <wachsmut@shu.edu> ... Recipient Okay
Server: 354 Enter mail, end with "." on a line by itself
Server: 250 Mail accepted
Message sent. Quitting.
```

Figure 10.2.5: Messages returned by smtp server while sending above message

Our final example will show a more sophisticated client that will interpret the answers return by the server it connects to.

### A WebViewer Client

As we have seen, creating client programs turned out to be pretty simple. Our next example will be somewhat more elaborate, not because of any network communications problem but rather because what should be accomplished once data has been successfully retrieved.

#### Example 10.2.8:

Create a stand-alone `WebViewer` program that connects to a web server on port 80 and retrieves a specified web page from that server. The user should be able to enter the URL to display in a text field, and the web page should appear in a `TextArea`. Your program should not attempt to properly interpret all `HTML` tags (as Netscape would), but it should:

- display all hyperlinks contained in the web page
- interpret some basic `HTML` formatting tags such as `<P>` and `<BR>` to make the page somewhat readable
- hide all `HTML` formatting tags that it does not interpret

Note that a web server sends the contents of a web page as character data.

This example presents us with several problems:

- How do I connect to a web server?
- What commands, according to the `http` protocol, will cause a web server to relinquish a web page?
- What are `HTML` formatting tags in the first place?
- How do I differentiate between `HTML` formatting tags and standard text?
- How do I interpret some `HTML` tags and ignore others?

The first question is actually the simplest: our program connects to a web server as in example 10.2.4, then establishes input and output streams similar to the example 10.2.5. The only difference to the `echo` server is that we need character-based streams, not data input/output streams.

The second question is also simple since we have already done that in example 10.1.10 using `Telnet`. We know that sending the command

```
GET / HTTP/1.0
```

followed by hitting `ENTER` twice will cause a web server to display a web page. To get a specific page from the web server, we will therefore try the command

```
GET /directory/filename.html HTTP/1.0
```

followed by hitting (or sending) `ENTER` twice.

We don't have the time to explain the details of `HTML` formatting tags, but here is a simple example that explains the basic.<sup>28</sup>

Web pages are stored as plain text file, containing text and a variety of tags that cause specific actions when interpreted by a web client. The `HTML` formatting tags are always enclosed in angular brackets “`<`” and “`>`” and these special characters do not show up otherwise in the text. Formatting tags have a name (one or more letters) and can contain any number of optional parameters. Parameters can stand alone, or are in the form of `PARAM="VALUE"`, where the quotation marks are not necessarily present and could be single or double quotation marks. Most, but not all, formatting tags have an “ending” tag that has the same name as the original tag, prefaced by a forward slash “`/`”. Here is an example, with the formatting tags in bold:

```
<HTML>
<HEAD>
<TITLE>This is a Title</TITLE>
</HEAD>
<BODY>
<H1>This is a Headline</H1>
This is a line of plain text, including some <B>bold</B> and some
<I>italics</I>. The line breaks must be specifically indicated by tags. The
<P>
paragraph tag implies a new paragraph, adding some extra space between them.
The line-break tag
<BR>
breaks a line at that spot, but does not add extra spacing. There are also <A
HREF="http://www.shu.edu/index.html">hyperlinks</A> that make text clickable
and specify which site to go to. Hyperlinks can get optional parameters such as
<A HREF="/index.html" TARGET="_new">the "new page target" option</A> which will
cause the specified page to appear in a new browser window.
<P>
There are different tags for bulleted and numbered lists:
<UL>
  <LI>item 1 of bulleted list
  <LI>item 2 of bulleted list
</UL>
and
<OL>
  <LI>item 1 of numbered list (automatically numbered)
  <LI>item 2 of numbered list (automatically numbered)
</OL>
The end of an HTML document is noted using special ending tags.
</BODY>
</HTML>
```

---

<sup>28</sup> See definition 6.3.21 for a description of `HTML` formatting tags.

This example does not come close to explaining all that can be accomplished with HTML tags, but it should serve as a brief example that should be enough complete this example.<sup>29</sup> Note that even when you use a web editor such as Netscape Composer or Microsoft FrontPage, the documents saved will include HTML formatting characters just as in this example.

The problem of extracting and interpreting the HTML tags seems somewhat related to the problem of interpreting mathematical expression that we solved successfully in the command-line Calculator example in examples 8.2.1, 8.2.3, and 8.2.4. There we used different types of tokens to distinguish between operators and numbers, so we will try a similar approach here. Recall the approach we used for the Calculator program:

- we used a generic Token class as the base class
- we used a TokenNumeric and TokenOperator class to distinguish between operators and numbers
- the TokenOperator class “knew” how to interpret the various operators, i.e. it could perform the mathematical operations represented by the operator

We’ll try the same here: we will use a generic HTML class as the base class, extend it to classes HTMLTag and HTMLText to hold formatting tags or text, respectively, and give the HTMLTag class the necessary know-how to interpret those tags that need interpretation. Assuming we have created these classes already, the main work of loading and interpreting web page content will be done by the class WebPage, structured as follows:

```
public class WebPage
{   private LinkedList tokens = new LinkedList();

    public WebPage(String host, int port, String file)
    public String getContent()
    private void tokenizer(String content)
    private String loader(String host, int port, String file)
}
```

The linked list tokens will contain a list of tokens in the order they appear in the web page.<sup>30</sup>

The constructor calls the private loader method with the appropriate input parameters and passes its output through the tokenizer method:

```
public WebPage(String host, int port, String file)
{   tokenizer(loader(host, port, file)); }
```

The loader, in turn, will connect to the web server and retrieve the web page as a single string with formatting tags and text mixed just as in the original page. It will use character streams to transfer data, using the InputStreamReader and OutputStreamWriter classes as bridges between byte-level input/output streams returned by getOutputStream and getInputStream and the character-level classes we need:

```
private String loader(String host, int port, String file)
{   String content = new String(""), line = new String("");
    try
```

---

<sup>29</sup> An added problem interpreting HTML formatting tags is that not all HTML documents are “well-formed”, i.e. they may not include some tags or options that really should be present according to the HTML standard.

<sup>30</sup> A queue, actually, would be more appropriate, but for simplicity we’ll use the build-in Java class LinkedList instead of a Queue.

```

    { Socket client = new Socket(host, port);
      PrintWriter out = new PrintWriter(
          new OutputStreamWriter(client.getOutputStream()));
      BufferedReader in = new BufferedReader(
          new InputStreamReader(client.getInputStream()));
      out.println("GET " + file + " HTTP/1.0");
      out.println("");
      out.flush();31
      while ((line = in.readLine()) != null)
          content += line;
      in.close(); out.close(); client.close();
    }
    catch(UnknownHostException uhe)
    { content = "<HTML>No content: host " + host + " unknown</HTML>"; }
    catch(IOException ioe)
    { content = "<HTML>No content: host " + host + " refused</HTML>"; }
    return content;
}

```

The smartest method of `WebPage` is the `tokenizer`. It takes as input the raw string obtained by the loader from the web server and breaks it up into `HTMLTag` and `HTMLText` objects. The algorithm used is simple:

- first, we break up the entire raw string into “tokens” according to the “<” character. That will ensure that each “token” will start with an `HTML` tag, but it may also include “standard” text.
- second, we break up each token from the first step according to the “>” character. That will result in two pieces, the first one being an `HTML` tag (with options if present) and the second one in text without formatting tags (could be empty).

In the second step we also create the appropriate `HTMLTag` or `HTMLText` objects and insert them into the list tokens:

```

private void tokenizer(String content)
{ StringTokenizer tags = new StringTokenizer(content, "<");
  while (tags.hasMoreTokens())
  { StringTokenizer split = new StringTokenizer(tags.nextToken(), ">");
    if (split.hasMoreTokens())
        tokens.addLast(new HTMLTag(split.nextToken()));
    if (split.hasMoreTokens())
        tokens.addLast(new HTMLText(split.nextToken()));
  }
}

```

Finally, the `getContent` method will iterate through the elements of the `tokens` list and get their string representations. Any formatting will be performed by the `HTML` classes through their `toString` methods<sup>32</sup>:

```

public String getContent()
{ String content = new String("");
  for (Iterator iterator = tokens.iterator(); iterator.hasNext();)
      content += iterator.next().toString();
}

```

<sup>31</sup> As we mentioned, when using character output streams we must `flush` the streams to cause our command(s) to be sent to the server.

<sup>32</sup> Our approach of letting the `HTMLTag` class decide on the formatting is much too simple. To properly format `HTML` documents we must take *other* formatting tags into account, not only the current one. For example, to properly decide on the right indentation of a paragraph we must take into account tags such as `<BLOCKQUOTE>` and `<UL>` that occurred *before* the current tag, and to properly format a table requires knowledge of tags *after* the current one.

```

    return content;
}

```

To handle the two type of HTML objects we want to deal with we'll create the following classes:

- class HTML: base class for the two different types of information contained in a web page
- class HTMLText: extends HTML and stores a text piece of a web page
- class HTMLTag extends HTML and stores a formatting tag; also knows how to interpret some tags and ignore others

Each class is simple, so here is the source code for all of them:

```

public class HTML
{   protected String token;
    public String toString()
    {   return token; }
}

public class HTMLText extends HTML
{   public HTMLText(String _token)
    {   token = _token.trim(); }
}

public class HTMLTag extends HTML
{   public HTMLTag(String _token)
    {   token = _token.trim(); }
    public String toString()
    {   String tag = token.toUpperCase();
        if ((tag.startsWith("P"))    || (tag.startsWith("DIV")) ||
            (tag.startsWith("H"))    || (tag.startsWith("UL")) ||
            (tag.startsWith("DL"))    || (tag.startsWith("/UL")) ||
            (tag.startsWith("/DL"))  || (tag.startsWith("BR")))
            return "\n";
        else if (token.startsWith("LI"))
            return "\n  * ";
        else if (tag.startsWith("A "))
            return " [" + token + " ] ";
        else
            return "";
    }
}

```

The HTMLTag class, in particular, class will perform some non-standard interpretation of some standard HTML tags, mainly adding line breaks to make the resulting text somewhat readable. It will also display the hyperlinks inside square brackets and cause every other tag not specifically mentioned to be ignored in the final else statement.

What's left is the final class containing the GUI elements and the button to load a particular web page as specified by the user in a standard full URL<sup>33</sup>:

```

import java.awt.*;
import java.awt.event.*;
import java.net.*;

public class WebViewer extends Frame implements ActionListener
{   private TextField address = new TextField();
    private TextArea display = new TextArea();
}

```

---

<sup>33</sup> See definition 9.5.4 for the definition of a full URL.

```

private Button go = new Button("View Page");
private class WindowCloser extends WindowAdapter
{ public void windowClosing(WindowEvent we)
  { System.exit(0); }
}
public WebViewer()
{ super("Web Viewer Lite");
  Panel north = new Panel();
  north.setLayout(new BorderLayout());
  north.add("West", new Label("URL:"));
  north.add("Center", address);
  north.add("East", go); go.addActionListener(this);
  Panel center = new Panel();
  setLayout(new BorderLayout());
  add("North", north);
  add("Center", display);
  addWindowListener(new WindowCloser());
  validate(); pack(); setVisible(true);
}
public void actionPerformed(ActionEvent ae)
{ if (ae.getSource() == go)
  showURL(address.getText());
}
public void showURL(String address)
{ try
  { setCursor(new Cursor(Cursor.WAIT_CURSOR));
    URL url = new URL(address);
    String host = url.getHost();
    int port = url.getPort();
    if (port <= 0)
      port = 80;
    WebPage page = new WebPage(host, port, url.getFile());
    display.setText(page.getContent());
  }
  catch(MalformedURLException murle)
  { display.setText("Invalid URL: " + address); }
  finally
  { setCursor(new Cursor(Cursor.DEFAULT_CURSOR)); }
}
public static void main(String args[])
{ WebViewer viewer = new WebViewer();
}

```

When all classes are compiled, the `WebViewer` will produce results similar to the one shown in figure 10.2.6 (only when you are connected to the Internet, of course):

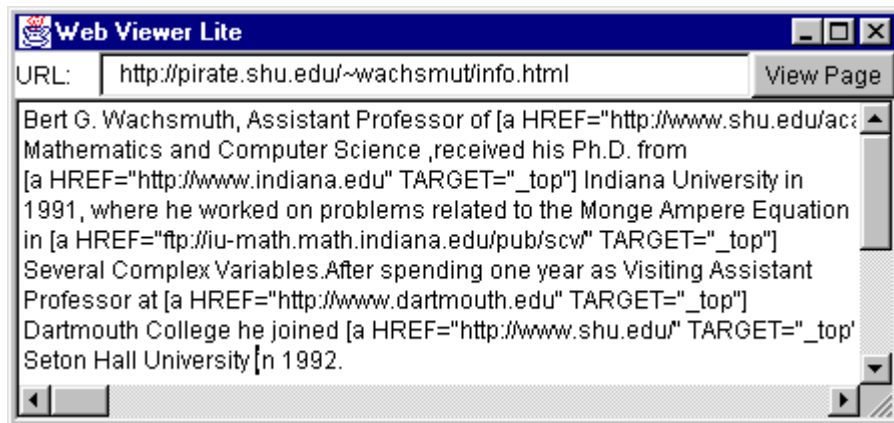


Figure 10.2.6: The `WebViewer` client displaying a text-only web page

This program can reveal interesting responses coming from a web server that you would not see using a regular web client.

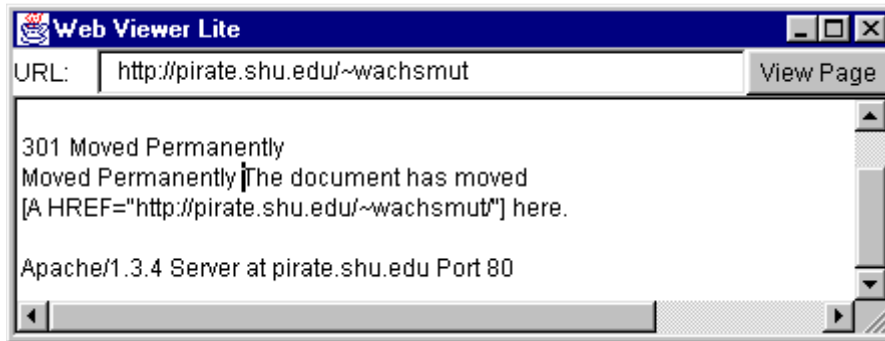


Figure 10.2.7: The `WebViewer` client program displaying a error message received from an `httpd` server

Figure 10.2.7 shows a so-called "301 redirect message". The URL `http://pirate.shu.edu/~wachsmut` is almost correct, but it is missing the last "slash". That will cause the Apache web server to issue a "301" redirect code, which in turn would cause a regular web client to automatically issue another call to the web server for the URL mentioned in the "moved permanently" message. Thus, you would not see this message in a regular web client, but you also would not be made aware of the fact that forgetting the final slash will cause a web client to issue *two* calls, slowing it down. ■

This client, while simple, provides lots of opportunities that we will pursue in the exercises. As a straightforward improvement we could extract the hyperlinks and make them clickable by adding them to a `List`. We could also implement "Backward" and "Forward" buttons similar to a real web client by using appropriate stacks. Finally we could load a web page asynchronously in a separate thread. But there are plenty of less obvious possibilities as well: we could use this client as blueprint for a data mining program that explores a web site and extracts some or all web pages linked from its main web page. Or we could use a similar client to extract particular information such as stock quote or weather information from some web site, format it appropriately, and save it to some other site to embed the information into that site's web pages.

If nothing else, we have learned at least one thing from this example: don't forget the final slash in a URL, it will slow down the web server's response time unnecessarily.

### 9.3. Creating Server Programs

Now we are ready to create our own server programs, together with associated client programs. Creating client programs is based, for the most part, on the `Socket` class. Creating server programs, as it turns out, is based on a similar class called `ServerSocket` that is also part of the `java.net` package. The class is defined as follows:

**Definition 10.3.1: The `ServerSocket` Class**

*This class represents a socket that can act as a server. A server socket binds itself to a particular port of the machine and usually waits for an incoming connection via the `accept` method. That method blocks until a connection is received, in which case it returns a new socket that can serve as an endpoint for the requested connection. The Java API defines `ServerSocket` as follows:*

```
public class ServerSocket extends Object
{ // selected constructor
  public ServerSocket(int port) throws IOException
  // selected methods
  public InetAddress getInetAddress()
  public int getLocalPort()
  public Socket accept() throws IOException
  public void close() throws IOException
}
```

The basic idea of setting up a server program is therefore simple: we create an instance of a `ServerSocket`, call on its `accept` method and wait. When a client connects, the `accept` method returns a `Socket`, and in the previous section we have learned everything about *that* class.

**Example 10.3.2:**

Create a “`LoggingServer`” that simply prints out (logs) all requests that a client makes. Then create a simple `LoggingClient` that connects to the same port that the server is running and sends strings of text obtained from the user. Run both classes on one computer, using the host name `localhost` (with IP number `127.0.0.1`) and any mutually known port number.<sup>34</sup> Note that you do *not* have to be connected to the Internet to run this client/server combination but your networking software must be loaded. If you have problems, do connect to the Internet even if you are not using an outside connection.

Following the basic idea mentioned we will instantiate a `ServerSocket` on, say, port 1234, then call on the `accept` method. After a connection is established, we’ll setup a `DataInputStream` using the `Socket` returned by `accept`. Then we enter a loop, listening on the input stream for data sent to our server until we read the string `.QUIT`:

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

public class LoggingServer
{ public static void main(String args[])
  { try
    { ServerSocket server = new ServerSocket(1234);
      System.out.println("Server started: " + server.getLocalPort());
      Socket connection= server.accept();
      System.out.println("Connection established.");
      DataInputStream in = new DataInputStream(
        connection.getInputStream());
      String line = new String("");
      while (!line.toUpperCase().equals(".QUIT"))
      { line = in.readUTF();
        System.out.println("<= " + line);
      }
      in.close(); connection.close(); server.close();
    }
  }
}
```

<sup>34</sup> If you are using a Unix systems, make sure to choose ports higher than 1024. Using port numbers less than 1024 require special rights that you may not have on a Unix system.

```

    }
    catch(IOException ioe)
    { System.err.println("Error: " + ioe); }
}
}

```

The corresponding client is almost identical to our `EchoClient` from example 10.2.5, but we only need to establish a `DataOutputStream` in this case. We will use the `Console` class from section 2.4, example 5, to ask the user for input:

```

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

public class SimpleClient
{ public static void main(String args[])
  { try
    { Socket connection = new Socket(args[0], 1234);
      System.out.println("Connection established:");
      String line = new String("");
      DataOutputStream out = new DataOutputStream(
        connection.getOutputStream());
      while (!line.equalsIgnoreCase(".QUIT"))
      { System.out.print("Enter text (or .quit) => ");
        line = Console.readString();
        out.writeUTF(line);
      }
      out.close();
      connection.close();
    }
    catch(UnknownHostException uhe)
    { System.err.println(uhe); }
    catch(IOException ioe)
    { System.err.println(ioe); }
  }
}

```

Now we can test our client/server combination by running both programs on one system. Both programs, obviously, require some command line input to start properly and the port numbers must match to make a connection. See figure 10.3.1 for a sample session with our logging client/server classes.

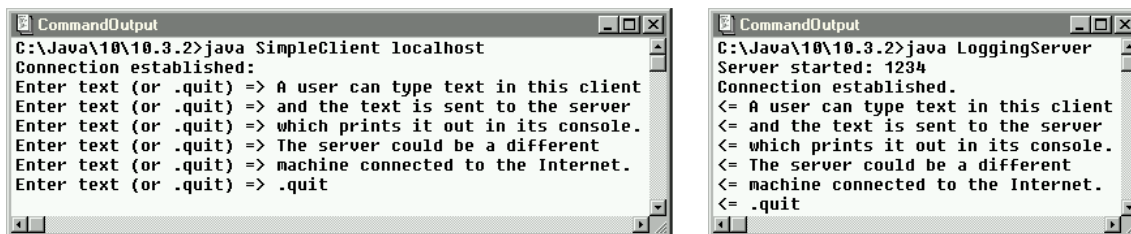


Figure 10.3.1: Sample session with `LoggingServer` and `SimpleClient`

Our server works fine, but it has two principle flaws we must fix before attempting any "real" examples:

- After the client terminates the connection, the server also shuts down. It must be restarted manually before the next client can be handled.
- Our server can not handle more than one connection. As long as one client is connected, no other client can access our server.

## A Server for Multiple Simultaneous Clients

To make client/server programming more useful, we need to create servers that can handle multiple clients simultaneously, so another reasonably simple example is in order before moving on to "real" examples.

### ***Example 10.3.3:***

Create an echo server and test it against the echo client we created before. Recall that the echo protocol specifies that the server should wait for incoming connection, receive a line of text, then send that line back to the client without changes, until the client terminates the connection. Your echo server should not use the "standard echo" port because that port may already be in use. Instead, both your echo server and client should use some mutually agreed upon port with number higher than, say, 2000. Make sure that the server does not terminate when a client is done, and that it can handle multiple clients simultaneously.

We will solve this task in several stages. The first stage will be a program very similar to the `LoggingServer` in example 10.3.2, but it will use an additional `DataOutputStream` to echo any strings received back to the client instead of printing them out on the console.

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

public class EchoServer
{   public static void main(String args[])
    {   try
        {   ServerSocket server = new ServerSocket(
            Integer.parseInt(args[0]));
            System.out.println("Server started on " +server.getLocalPort());
            Socket connection= server.accept();
            System.out.println("Connection established.");
            DataInputStream in = new DataInputStream(
                connection.getInputStream());
            DataOutputStream out = new DataOutputStream(
                connection.getOutputStream());
            String line = new String("");
            while (!line.equalsIgnoreCase(".QUIT"))
            {   line = in.readUTF();
                System.out.println("Echoing: " + line);
                out.writeUTF(line);
            }
            in.close(); out.close(); connection.close(); server.close();
        }
        catch(IOException ioe)
        {   System.err.println("Error: " + ioe); }
    }
}
```

This simple `EchoServer` will work just fine against the `EchoClient` we created in example 10.2.5, because client and server speak the same protocol. Actually, the `EchoClient` from example 10.2.5 uses port number 7, so we need to change that number to match the port where the current server is listening. We will leave this small modification to you. Of course this server can still handle only one client and quits when that client is done.

In our second stage we will modify the `EchoServer` so that it stays active if a client quits. The idea is simple: the server uses the `accept` method to wait for a connection. Since `accept` is called exactly once, the server can only handle exactly one connection. To change this, we embed the call to `accept` in a loop as follows (the new code is in bold and italics):

```
public class EchoServer
{ private static boolean running = true;
  public static void main(String args[])
  { try
    { ServerSocket server = new ServerSocket(
      Integer.parseInt(args[0]));
      System.out.println("Server started on " +server.getLocalPort());
      while (running)
      { Socket connection= server.accept();
        System.out.println("Connection established.");
        DataInputStream in = new DataInputStream(
          connection.getInputStream());
        DataOutputStream out = new DataOutputStream(
          connection.getOutputStream());
        String line = new String("");
        while (!line.equalsIgnoreCase(".QUIT"))
        { line = in.readUTF();
          System.out.println("Echoing: " + line);
          out.writeUTF(line);
        }
        in.close(); out.close(); connection.close();
        System.out.println("Connection closed.");
      }
    }
    catch(IOException ioe)
    { System.err.println("Error: " + ioe); }
  }
}
```

This, indeed, will work, but there are two problems: First, we were somewhat too successful because our server will now run all the time and can only be shut down by killing the process (by pressing CONTROL-C, for example). We really want a server that stays active for additional connections but also shuts down gracefully when told. The second problem is more serious: if a client connects properly but disconnects improperly (by pressing CONTROL-C in the client program, for example), then the server throws an `IOException` and quits after all. This is a situation that any Internet server must handle: the Internet is, for a variety of reasons, unreliable and connections can terminate at any time.<sup>35</sup>

To fix these problems, we'll modify the code as follows:

- First, we will modify the protocol that both client and server need to understand: if a client sends the string `.SHUTDOWN` then the server should terminate the connection to that client and shut down.
- Second, we'll add another try-catch block inside the while loop to ensure that errors in communicating to one client will not terminate the server.

Here is the new code, with the changes shown in bold and italics

```
public class EchoServer
```

---

<sup>35</sup> The easiest scenario for a connection failure is that the `EchoClient` uses a dialup connection and which is terminated before properly closing the client.

```

{ private static boolean running = true;
  public static void main(String args[])
  { try
    { ServerSocket server = new ServerSocket(
      Integer.parseInt(args[0]));
      System.out.println("Server started on " +server.getLocalPort());
      while (running)
      { Socket connection= server.accept();
        System.out.println("Connection established.");
        DataInputStream in = new DataInputStream(
          connection.getInputStream());
        DataOutputStream out = new DataOutputStream(
          connection.getOutputStream());
        String line = new String("");
        try
        { while (!line.equalsIgnoreCase(".QUIT"))
          { line = in.readUTF();
            System.out.println("Echoing: " + line);
            if (line.toUpperCase().equals(".SHUTDOWN"))
            { running = false;
              line = ".QUIT";
            }
            out.writeUTF(line);
          }
          in.close(); out.close(); connection.close();
          System.out.println("Connection closed.");
        }
        catch(IOException ioe)
        { System.out.println("Connection closed unexpectedly."); }
      }
      System.out.println("Server closed");
    }
    catch(IOException ioe)
    { System.err.println("Error: " + ioe); }
  }
}

```

This server will stay active reliable and can handle multiple clients, but only one client at a time.<sup>36</sup>

Our third and final version will overcome this last restriction by handling clients in independent threads:

- we will create a class `EchoServerThread` extending `Thread` and move the code that actually handles a connection into the `run` method of that thread
- we will modify the `EchoServer` so that it starts a new thread every time a connection is made

Here is the new `EchoServerThread` class, containing virtually the same code that handled a connection as before, just relocated into the `run` method of the thread:

```

public class EchoServerThread extends Thread
{ private Socket connection;
  public EchoServerThread(Socket _connection)
  { connection = _connection;
    start();
  }
  public void run()

```

---

<sup>36</sup> If a second client tries to connect while the first client is still active, it will be put on hold until the first client quits. The second client will not see any error message or throw any exceptions, it will simply wait until the first one is done.

```

    { try
      { DataInputStream in = new DataInputStream(
        connection.getInputStream());
        DataOutputStream out = new DataOutputStream(
        connection.getOutputStream());
        String line = new String("");
        while (!line.equalsIgnoreCase(".QUIT"))
        { line = in.readUTF();
          System.out.println("Echoing: " + line);
          out.writeUTF(line);
        }
        in.close(); out.close(); connection.close();
        System.out.println("Connection closed.");
      }
      catch(IOException ioe)
      { System.out.println("Connection closed unexpectedly."); }
    }
}

```

Thus, the original `EchoServer` class is no longer responsible for managing a connection. Instead, it will simply start a new `EchoServerThread` whenever a connection is made so that it is free to handle the next connection while leaving the details of the currently active connection to the thread.

```

public class EchoServer
{ private static boolean running = true;
  public static void main(String args[])
  { try
    { ServerSocket server = new ServerSocket(
      Integer.parseInt(args[0]));
      System.out.println("Server started on " +server.getLocalPort());
      while (running)
      { Socket connection= server.accept();
        System.out.println("New connection moved to thread.");
        EchoServerThread handler = new EchoServerThread(connection);
      }
    }
    catch(IOException ioe)
    { System.err.println("Error: " + ioe); }
  }
}

```

Finally, we have achieved our goal: with these two relatively simple classes we have created a stable Internet server that can handle multiple clients simultaneously.<sup>37</sup> ■

The client/server system from example 10.3.3 can serve as a model for other server programs that need to handle multiple clients at the same time.

#### Definition 10.3.4: Rule of Thumb to Create a Multi-Client Server

*To create a server program that can service multiple clients simultaneously, you could follow these guidelines:*

<sup>37</sup> We have lost the ability to shutdown the server gracefully. In principle, any of the threads could be used to set the field `running` of the server to `false`. That would shutdown the server, but not right away. Therefore the server is best terminated by pressing `CONTROL-C`. Alternatively, the server itself could use a thread to wait for incoming connections.

*Create two classes. The first class waits for connections and passes active connections to the second class. The second class extends `Thread` and handles the communication with one client.*

- *The first class instantiates a new `ServerSocket` on a given port. The `accept` method of that `ServerSocket` should be inside an infinite loop. If a connection is established, the class passes the `Socket` returned by `accept` to a new instance of the second class extending `Thread`.*
- *The thread class uses the `Socket` received as input parameter during instantiation to setup input and output streams to the active client. It then handles all communications according to the agreed-upon protocol inside its `run` method.*

Client/server solutions can be particularly useful to overcome the security restrictions imposed on applets. As outlined in definition 9.5.2, applets can not directly manipulate files, not even list files in a given directory. In our next example we will create a client/server combination where an applet can direct a server to perform actions that the applet by itself is not allowed to do, such as obtain a directory listing.

## Directory Client/Server System

### Example 10.3.5

In example 9.2.13 we created a standalone program to list the contents of a given directory. Create a client/server combination that will allow an applet to list files and directories on the server system. The user should be able to move up and down the directory structure of the server, listing the contents of each directory as well as a brief summary of each directory.<sup>38</sup> In this example, construct only the server, making sure to clearly specify the protocol that the server understands.

It should be clear from the outset that this client/server package, if implemented, may compromise the security of the server system. After all, applets have security restrictions for a good reason, and while technically possible it may not be advisable to bypass these restrictions with a client/server solution.

Be that as it may, to proceed with this example you should first review example 9.2.13 where the `File` class was used extensively to list the contents of a directory. Code similar to this will be embedded in the server that sends the contents of a directory to an applet instead of displaying it on the console. The first question we need to solve is exactly what should our server be able to do, and exactly how do we get it to perform that action. In other words, we need to specify the protocol that the server can understand. We decide to use the capabilities outlined in table 10.3.2.

Command	Server Action
<code>pwd</code>	Sends the current directory
<code>chdir ..</code>	Changes directory on the server to the parent of the current directory if possible
<code>chdir newdir</code>	Changes directory on the server to the directory specified in <code>newDir</code> if possible
<code>list</code>	Sends a list of files and directories inside the current directory

<sup>38</sup> This example could serve as the first stage of an FTP-like client/server package that can be used to transfer data from the applet to the server.

<code>pass passwd</code>	Checks to see if <code>passwd</code> is the password that the server expects. If so, sends "okay" otherwise sends "error".
<code>anything else</code>	Terminates the connection

*Table 10.3.2: The outline of the protocol for our directory server*

This is not precise enough. What exactly is the data type of the commands sent to the server? What type does the server return when it sends the "current directory" or a "list of files"? Therefore we need to clarify the protocol outlined in table 10.3.2 as follows:

- The server accepts commands consisting of exactly two strings. The first string must be one of the commands "pwd", "chdir", "list", or "pass". The second string is a parameter to the first command. If the first command string does not need a parameter, the second string will be ignored but must be present. Any other command and parameter will cause the server to disconnect.
- If the server receives the "pwd" command and an arbitrary parameter it will send the current directory as a string.
- If the server receives the "chdir" command with parameter "." it will attempt switch to the parent of the current directory if possible and send the new current directory as a string.
- If the server receives the "chdir" command with a parameter different from "." it will attempt to switch to the directory specified by the parameter and send the new current directory as a String.
- If the server receives the "list" command with an arbitrary parameter it will send an array of type `File[]` consisting of the contents of the current directory.
- If the server receives the "pass" command it will check whether the following parameter is equal to a preset password. If so it will send the string "okay", otherwise it will send the string "error" and disconnect.
- The first command the server expects is the "pass" command. Only after sending the correct password will the server allow further commands.

This protocol implicitly dictates the type of streams that the server needs to setup:

- since it receives only `String` types, the input stream can be a `DataInputStream`
- since it needs to send `String` types as well as an array of `File` the output stream must be an `ObjectOutputStream`

Once we have carefully specified the protocol, it is not hard to create the server class. Our server, like any good server, should be able to handle multiple clients simultaneously. Therefore definition 10.3.4 provides the basic blueprint for our classes.

We will create a `DirServer` class that establishes a `ServerSocket` and uses a `DirThread` class to handle any incoming connection. The details of the connection are left to the `DirThread` class, according to definition 10.3.4. The class also defines the password that the client needs to send, as well as a version string that will be used to ensure that only proper clients can connect to this server. The code for `DirServer` is quite simple:

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

public class DirServer
{   protected static final String PASSWD = "jbd007";
    protected static final String VERSION = "dirserver 1.0";
    private static final int PORT = 8765;
    public static void main(String args[])
    {   try
```

```

        { ServerSocket server = new ServerSocket(PORT);
          System.out.println("Server started: " + server.getLocalPort());
          while (true)
            { DirThread dd = new DirThread(server.accept()); }
        }
        catch(IOException ioe)
        { System.err.println("Error: " + ioe); }
    }
}

```

Our next class will be `DirThread` which will extend `Thread`. The basic outline for the class is dictated by the specifications of our protocol is as follows:

```

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

public class DirThread extends Thread
{
    private ObjectOutputStream out = null;
    private DataInputStream in = null;
    private Socket sock = null;
    private boolean running = false;
    private File currentPath = new File(System.getProperty("user.dir"));
    public DirThread(Socket _sock)
    { /* Sets up input and output stream and starts the thread. */ }
    public void run()
    { /* Processes input commands and parameters as long as running
       is true, catching any exceptions that might occur. If an
       exception occurs, disconnects the client by calling hangup */ }
    private void send(File list[]) throws IOException
    { /* Sends the array of File list through the ObjectOutputStream */ }
    private void send(String msg) throws IOException
    { /* Sends the String msg through the ObjectOutputStream */ }
    private void changeDirectory(String newDir) throws IOException
    { /* If newDir equals "..", makes the parent of currentPath
       the new currentPath if possible. Otherwise checks if newDir
       is a valid directory and sets currentPath to newDir. Then it
       sends currentDir as a String */ }
    private void authenticate()
    { /* Reads password. If it matches DirServer.PASSWD sends "okay",
       otherwise sends "error" and disconnects */ }
    private void hangup()
    { /* Closes all open streams and sockets and sets running to
       false */ }
    private void processCommand(String cmd, String option)
        throws IOException
    { /* determines which of the above methods to call, depending on
       the value of cmd */ }
}

```

Now let's implement the methods as outlined. First, here is the constructor, which is straightforward:

```

public DirThread(Socket _sock)
{
    System.out.println("Starting directory connection ...");
    sock = _sock;
    try
    {
        in = new DataInputStream(sock.getInputStream());
        out = new ObjectOutputStream(sock.getOutputStream());
        running = true;
        start();
    }
}

```

```

        catch(IOException ioe)
        { System.err.println("Error setting up streams: " + ioe); }
    }

```

Next comes the `run` method, which is started by the constructor calling `start`. It first calls `authenticate` to check for a valid password. If the password checks out, the main loop of the `run` method starts. It reads two strings from the input stream and passes them along to `processCommand` to decide what to do:

```

public void run()
{
    authenticate();
    while (running)
    {
        try
        {
            processCommand(in.readUTF(), in.readUTF());
            catch(IOException ioe)
            {
                hangup();
            }
        }
    }
}

```

The two `send` methods send an array of `File` and a `String`, respectively, through the `ObjectOutputStream`, followed by a call to `flush` to ensure that the data is really sent.

```

private void send(File list[]) throws IOException
{
    out.writeObject(list);
    out.flush();
}
private void send(String msg) throws IOException
{
    out.writeUTF(msg);
    out.flush();
}

```

The next method, `changeDirectory`, receives as input the parameter `newDir`. It tries to reset the current directory either to its parent or to the input variable `newDir`. When it is finished, it sends the value of `currentPath`.

```

private void changeDirectory(String newDir) throws IOException
{
    if ((newDir.equals("..")) &&
        (currentPath.getParentFile() != null))
        currentPath = currentPath.getParentFile();
    else
    {
        File newPath = new File(newDir);
        if (newPath.exists() && newPath.isDirectory())
            currentPath = newPath;
    }
    send(currentPath.toString());
}

```

The `authenticate` method reads two strings from the input stream. The first must be the "pass" command and the second must be the correct version – password combination. If the password is correct, it sends "okay". Otherwise it sends "false" and calls `hangup` to terminate the connection.

```

private void authenticate()
{
    try
    {
        String cmd = in.readUTF();
        String passwd = in.readUTF();
        if (passwd.equals(DirServer.VERSION + "|" + DirServer.PASSWD))
            send("okay");
        else
        {
            send("error");
            hangup();
        }
    }
}
catch(IOException ioe)

```

```

    { hangup(); }
}

```

The `hangup` method closes all streams and the open socket if possible and sets `running` to `false` to terminate the `run` loop of the thread.

```

private void hangup()
{ try
  { System.out.println("Hanging up");
    in.close(); out.close(); sock.close();
  }
  catch(IOException ioe)
  { System.err.println("Error disconnecting: " + ioe); }
  finally
  { running = false; }
}

```

The final method to implement is `processCommand`. It simply checks the value of `cmd` and chooses the correct method to handle the command.

```

private void processCommand(String cmd, String option) throws IOException
{ if (cmd.equals("pwd"))
  send(currentPath.toString());
  else if (cmd.equals("chdir"))
  changeDirectory(option);
  else if (cmd.equals("list"))
  send(currentPath.listFiles());
  else
  hangup();
}

```

That's our server class. It should compile and when executed it would wait for a client that speaks the correct protocol. But as it turns out there is a subtle problem with our server that we would discover later when testing the server with our client applet. The problem is that our server thread sends an array of `File` objects as the contents of the current directory. That works in principle, but if the client tries to execute any of the methods of the `File` class such as `isDirectory`, the client, *not* the server, will at that time try to determine whether a particular path is a directory. Since applets are not allowed to inquire such things, a security exception will be thrown. Therefore our server needs to send a list of objects that already include all information about the files that the client might be interested in.<sup>39</sup> Therefore we create a `FileInfo` class that stores all relevant information about a `File` in fields. If objects of that type are transferred to the client the client can inquire about the values of the fields without causing invalid calls to the file system of the applet. The class needs to implement `Serializable` so that it can be transferred through an `ObjectOutputStream`:

```

import java.io.*;

public class FileInfo implements Serializable
{ private String shortName;
  private String fullName;
  private long size;
  private boolean isDir;
  public FileInfo(File file)
  { shortName = file.getName();

```

---

<sup>39</sup> There is an additional problem when sending objects of type `File`. The representation of the path leading to the file will be determined at the time the `toString` method is called, which will be in the client. Therefore, the client will use the particular representation of a path specific to its operating system. If the server runs on another operating system, the path representation might be different, causing problems.

```

        fullName = file.toString();
        size      = file.length();
        isDir     = file.isDirectory();
    }
    public String getName()
    { return shortName; }
    public String toString()
    { return fullName; }
    public long length()
    { return size; }
    public boolean isDirectory()
    { return isDir; }
}

```

Finally we need to change the `send` method in `DirServer` so that it first constructs appropriate `FileInfo` objects and then transfers an array of those objects instead of an array of `File`. In other words, we need to change `private void send(File list[])` as follows:

```

private void send(File list[]) throws IOException
{
    FileInfo info[] = new FileInfo[list.length];
    for (int i = 0; i < list.length; i++)
        info[i] = new FileInfo(list[i]);
    out.writeObject(info); out.flush();
}

```

That is also a change in our protocol, so we should be sure to modify it as follows:

- If the server receives the "list" command with an arbitrary parameter it will *not* send an array of type `File[]` consisting of the contents of the current directory, as originally specified. Instead it will construct an array of `FileInfo` objects, one per `File` in the current directory, and send that array instead. ■

The server classes will compile and the `DirServer` can be started. It will wait for a connection from an appropriate client, which we will tackle next.

### ***Example 10.3.6***

In example 9.2.13 we created a standalone program to list the contents of a given directory. In example 10.3.5 we created a server class that can service multiple clients and understand commands such as `pwd`, `chdir`, and `list`. Create a client applet that uses the server created in example 10.3.5 to list files and directories on the server system. The user should be able to move up and down the directory structure of the server, listing the contents of each directory as well as a brief summary of each directory. The host to be used should be transferred into the applet via an appropriate `PARAM` tag in the `HTML` document. Make sure to test your client/server package extensively, in particular by moving the server to an appropriate web server, which must be the same machine containing the client applet.<sup>40</sup>

<sup>40</sup> You should be aware that this particular server may compromise the security of the entire web server system since it allows basically anyone to view the contents of all directories. We do ask the user to enter a password, to be sure, but that mechanism is much less secure than standard security mechanisms of a web server. You should therefore remove our directory server from the web server system after testing it.

At this stage we do not need to know exactly how the server works. All we need is the exact protocol that the server follows so that we can create our client accordingly. Let's review the protocol from example 10.3.5 briefly:

- The server accepts commands consisting of exactly two strings, commands and parameters. Commands must be "pwd", "chdir", "list", or "pass", parameters as necessary for the command. Anything else will close the connection.
- The "pwd" command with arbitrary parameter causes server to send the current directory as a string.
- The "chdir" command with parameter "." causes server to switch to the parent of the current directory, or to the directory specified by the parameter if different from ".". It will send the new current directory as a string.
- If the server receives the "list" command with an arbitrary parameter will construct an array of `FileInfo` objects and send that array.
- The first command the server expects is the "pass" command. Server will check whether the parameter is equal to a preset password. If so it will send the string "okay", otherwise it will send the string "error" and disconnect.

Next let's get the layout of our applet out of the way. We will use the `PanelBox` class introduced in example 9.6.1 to enhance the layout of our class, so we need `PanelBox` in the same directory as the client applet listed next:

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
import java.text.DecimalFormat;

public class DirClientApplet extends Applet implements ActionListener
{
    private Socket connection = null;
    private DataOutputStream out = null;
    private ObjectInputStream in = null;
    private List dirList = new List();
    private List fileList = new List();
    private TextField passwd = new TextField(10);
    private Label status = new Label("Click 'Connect' ...");
    private Button connect = new Button("Connect");
    private Button disconnect = new Button("Disconnect");
    private boolean connected = false;
    private DecimalFormat formatTool = new DecimalFormat("#,###");41
    private String host = "localhost";
    public void init()
    {
        if (getParameter("host") != null)
            host = getParameter("host");
        Panel buttons = new Panel(new FlowLayout());
        buttons.add(new Label("Password: ")); buttons.add(passwd);
        buttons.add(connect); buttons.add(disconnect);
        passwd.setEchoChar('*');
        Panel lists = new Panel(new BorderLayout());
        lists.add("North", new PanelBox(dirList, "Directories"));
        lists.add("Center", new PanelBox(fileList, "Files"));
        setLayout(new BorderLayout());
        add("North", new PanelBox(status, "Status"));
        add("Center", lists);
        add("South", buttons);
    }
}
```

---

<sup>41</sup> See section 1.5 for a review on formatting decimals.

```

        connect.addActionListener(this);
        disconnect.addActionListener(this);
        dirList.addActionListener(this);
    }
    private void send(String cmd, String option) throws IOException
    { /* sends cmd and option to the server */ }
    private void connect()
    { /* connects to the server, sets up appropriate streams, and sends
        authentication information */ }
    private void showDirectory(String cmd, String path)
    { /* sends the list command to the server, receives the array of
        FileInfo object and add them to the appropriate lists. Also
        computes the number of directories, files, and the total number
        of bytes in the directory to put into the status line */ }
    public void stop()
    { /* sends bye to the server to disconnect, closes all streams and
        sockets, and clears the file and directory list */ }
    public void actionPerformed(ActionEvent ae)
    { /* reacts appropriately to clicks on the connect and disconnect
        buttons as well as to double-clicks on items in the directory
        list. */ }
    }
}

```

Now we need to implement the methods according to these specifications. The first is `send`, which is completely easy:

```

private void send(String cmd, String option) throws IOException
{ out.writeUTF(cmd); out.writeUTF(option); out.flush(); }

```

Next there is the `connect` method. It needs to establish the connection to the server, setup appropriate input and output stream through the socket, and send the password. The password will be constructed by concatenating the string `"dirserver 1.0|"` with whatever the user entered into the `passwd` field of the applet. If the server responds with `"okay"` the password was correct, the connection is established, and we call `showDirectory` with command `"pwd"` and an arbitrary parameter as input to get a listing of the current directory.

```

private void connect()
{ try
  { connection = new Socket(host, 8765);
    out = new DataOutputStream(connection.getOutputStream());
    in = new ObjectInputStream(connection.getInputStream());
    send("pass", "dirserver 1.0|" + passwd.getText());
    if (in.readUTF().equals("okay"))
    { status.setText("Authenticating ... connected");
      connected = true;
      showDirectory("pwd", "dummy");
    }
    else
      status.setText("Authenticating ... REFUSED");
  }
  catch(Exception ex)
  { status.setText("Error: " + ex); }
}

```

The next method is `showDirectory`. It requires two input parameters. The first one should be either `"pwd"` or `"chdir"`, while the second one should indicate the directory to list, if necessary. After sending the command to the server and receiving the current directory, the method sends the list command and receives an array of `Object` types from the server that can be typecast into an array of `FileInfo`

objects. Then the method loops through the available `FileInfo` objects, adding them to the file or directory list as appropriate and computes the summary information for this directory. Before adding any data to the directory list it adds the string `".."` so that the user can always double-click on that entry to view the parent directory of the current one.

```
private void showDirectory(String cmd, String path)
{ try
  { send(cmd, path);
    path = in.readUTF();
    send("list", "dummy");
    FileInfo list[] = (FileInfo[])in.readObject();
    dirList.removeAll(); fileList.removeAll(); dirList.add("..");
    long total = 0, nDirs = 0, nFiles = 0;
    for (int i = 0; i < list.length; i++)
    { if (list[i].isDirectory())
      { nDirs++;
        dirList.add(list[i].toString());
      }
      else
      { nFiles++;
        total += list[i].length();
        fileList.add(list[i].toString());
      }
    }
    status.setText(path+ " ", "+ nDirs+ " dirs, "+ nFiles+ " files, "+
                  formatTool.format(total)+ " bytes]");
  }
  catch(Exception ex)
  { status.setText("Error: " + ex); }
}
```

Then there is the `stop` method. Because we chose this particular method header, the `stop` method is automatically called when the user leaves the page containing our applet. That seems appropriate because if the user does not look at our applet the client should terminate the connection to the server. Other than that the method is straightforward.

```
public void stop()
{ if (connected)
  { try
    { send("bye", "dummy");
      dirList.removeAll(); fileList.removeAll();
      in.close(); out.close(); connection.close();
    }
    catch(IOException ioe)
    { System.err.println("Error: " + ioe); }
    finally
    { connected = false; passwd.setText("");
      status.setText("Disconnected. Click 'Connect' ...");
    }
  }
}
```

Finally there is `actionPerformed`. It will call on `connect` and `stop` if the appropriate buttons are clicked. It will also handle a double-click on items in the directory list. In that case it calls on the `showDirectory` method using the currently selected entry from the directory list as parameter to get a listing of the selected directory.

```
public void actionPerformed(ActionEvent ae)
{ if ((!connected) && (ae.getSource() == connect))
  connect();
}
```

```

else if (connected && (ae.getSource() == disconnect))
    stop();
else if (connected && (ae.getSource() == dirList))
    showDirectory("chdir", dirList.getSelectedItem());
}

```

Now we are done and we can view the fruits of our labor. Figure 10.3.3 shows how an applet can now view the contents of an entire web server, even though the standard security restrictions would not allow it to do so directly.

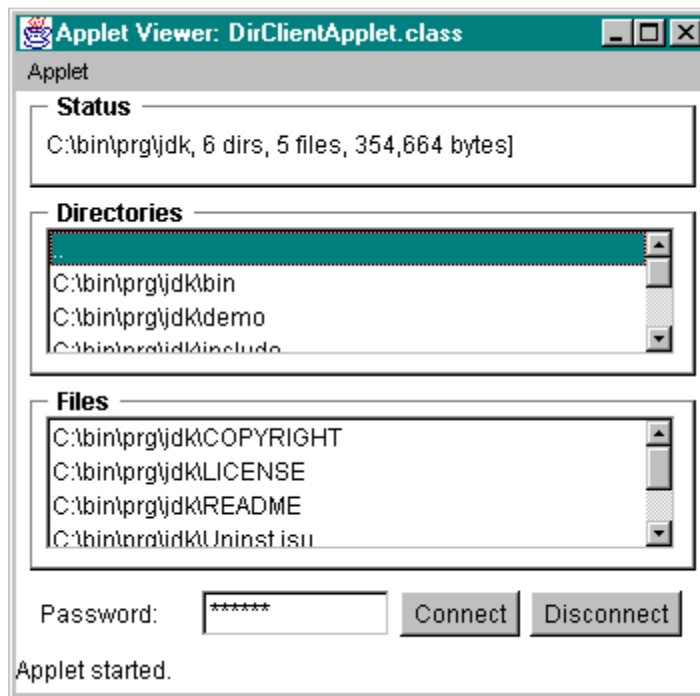


Figure 10.3.3: DirClientApplet showing a directory listing by using a DirServer

## Online Ordering Synchronized Client/Server System

As our final client/server solution aside from the extensive chat client/server system in section 10.4 we will finish the online ordering system that we started in example 9.6.1. That example will illustrate how to add a synchronization feature to a server handling multiple clients simultaneously to make sure that concurrent access to a resource such as a file will not destroy the integrity of that resource.

Recall that we created an applet in example 9.6.1 that reads a "catalog" from a data file, lets the user browse through the catalog and choose items to order. When done, the user was supposed to be able to submit their order electronically, but because of the security restrictions imposed on applets we had to postpone that last part of the puzzle. Now we have all the tools necessary to complete that program.

### ***Example 10.3.7:***

In example 9.6.1 we created a (reasonably) professional looking `Catalog` applet that could read a data file representing items in a catalog, allowed the user to browse through the catalog, and let the user put together an order. The user could try to submit their order, but the actual submission process was not implemented. Now we want to finish that example so that the user *can* submit an order. To do that, create a server program that receives the order from the `Catalog` applet and appends it to a data file on disk. Note that you need to be careful not to corrupt the order data file, especially since multiple clients may attempt to submit orders simultaneously.

First, let's review the classes that made up the original `Catalog` applet so that we can decide what exactly needs to be done to finish it:

**CatalogItem.java**  
Class to represent an individual catalog item

**CatalogAdmin.java**  
Simple class to create a sample catalog data file

**CatalogOrder.java**  
Class to contain the necessary information representing "an order"

**CatalogImage.java**  
Utility class to store an image from the catalog

**PanelBox.java**  
Utility class to put borders around components in a GUI program or applet

**Catalog.java**  
The main applet

**CatalogSubmission.java**  
The dialog where the user can review their order, enter their mailing and billing information, and submit the actual order

We can ignore the classes `CatalogImage` and `PanelBox` completely, since they will not have an impact on the server we need to create. Also, the actual applet `Catalog` hopefully delegates its work to other classes so that we should focus on the `CatalogItem`, `CatalogOrder`, and `CatalogSubmission` classes. Let's review these classes in detail:

```
import java.io.*;

public class CatalogItem
    implements Serializable
{
    private String ID;
    private String shortName;
    private String longName;
    protected String imageName;
    protected double price;
    public CatalogItem(String _ID)
    public void setName(String _shortName,
                       String _longName)
    public String getName()
    public String toString()
}

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

public class CatalogOrder
    implements Serializable
{
    private String name = null;
    private String credit = null;
    private String address = null;
    private LinkedList list =
        new LinkedList();
    private double totalCost = 0.0;
    public void setInfo(String _name,
                       String _credit,
                       String _address)
    public boolean isComplete()
    public void add(CatalogItem item)
    public String toString()
}
```

The important thing to realize is that both classes implement the `Serializable` interface so they are prime candidates to be written to an object stream. In fact, the `CatalogOrder` class contains a list of `CatalogItems` that the user wishes to order, as well as mailing and billing information, so it is the primary object that we need to transfer. It remains to find out where to add the code that will

eventually write the object to an object stream. If we review the `CatalogSubmission` class that questions is easily answered:

```
import java.awt.*;
import java.awt.event.*;

public class CatalogSubmission extends Frame implements ActionListener
{   private Catalog owner = null;
    private TextField name = new TextField();
    private TextArea address = new TextArea(3,20);
    private TextField credit = new TextField();
    private TextArea orders = new TextArea(6,30);
    private CatalogOrder theBill = null;
    private Button submit = new Button("Submit Order");
    private Button cancel = new Button("Cancel Order");
    private Label status = new Label("Enter personal and mailing info");
    public CatalogSubmission(Catalog _owner, CatalogOrder _theBill)
    public void actionPerformed(ActionEvent ae)
    {   if (ae.getSource() == cancel)
        owner.closeSubmission();
        else if (ae.getSource() == submit)
        {   theBill.setInfo(name.getText(), credit.getText(),
                            address.getText());
            if (theBill.isComplete())
                status.setText("NOT SUBMITTED: feature not implemented.");
            else
                status.setText("NOT SUBMITTED: incomplete information.");
        }
    }
}
```

If the user clicks on the submit button, an error message is generated instead of actually submitting the order to a server. Therefore, we need to modify the `actionPerformed` method in the `CatalogSubmission` class to call a method that will transfer our ordering data. Our strategy will be as follows:

- we modify the `actionPerformed` method of the `CatalogSubmission` class to call on the new method `submitOrder` instead of displaying the text feature not implemented
- the `submitOrder` method will connect to a specific port on the host where the applet originated, establish an `ObjectOutputStream` and attempt to transfer the entire `CatalogOrder` object through that stream
- we will create a server program that listens on the specified port, receive the `CatalogOrder` object, and append it to a text file

The first step is easy: we modify the `actionPerformed` method of `CatalogSubmission` as follows (with the modified code in bold and italics):

```
public void actionPerformed(ActionEvent ae)
{   if (ae.getSource() == cancel)
    owner.closeSubmission();
    else if (ae.getSource() == submit)
    {   theBill.setInfo(name.getText(), credit.getText(),
                        address.getText());
        if (theBill.isComplete())
            submitOrder();
        else
            status.setText("NOT SUBMITTED: incomplete information.");
    }
}
```

Before implementing the other two steps, however, we need to carefully agree on the port to be used and - more importantly - on the protocol we want to use. We arbitrarily pick port 5678 as our "submission port", and we define our protocol as follows:

- the server will listen on port 5678 for an incoming connection
- after connecting, server and client will establish an object stream through their sockets
- the client will send the string "JBDOOrderClient"
- the server will receive that string and reply with the string "JBDOOrderServer"
- the client will send the version number 1.0 as a double
- the server will receive that double and reply with the double 1.0
- the client will send the complete `CatalogOrder` object
- the server will receive an object and type-cast it as `CatalogOrder`
- client and server will close their connection and the server will append the proper information to a plain text file for later processing.

If, at any time the responses do not follow this protocol, the connection will be closed. This protocol will ensure, at least to some degree, that only an authorized client can submit an order to our server. In a "real" system, this protocol should not be made public and some or all of the information should be submitted in an encrypted form to ensure privacy. In our case, we will be content with sending the information "as is", without worrying about proper encryption.<sup>42</sup>

Based on this protocol we add a few constants to our class, then create the `submitOrder` method as follows:

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

public class CatalogSubmission extends Frame implements ActionListener
{ private final static String CLIENT_ID = "JBDOOrderClient";
  private final static double CLIENT_VERSION = 1.0;
  private final static String SERVER_ID = "JBDOOrderServer";
  private final static double SERVER_VERSION = 1.0;
  private final static String HOST = "localhost";
  private final static int PORT = 5670;
  // remaining fields as before
  public CatalogSubmission(Catalog _owner, CatalogOrder _theBill)
  { /* no change */ }
  public void actionPerformed(ActionEvent ae)
  { if (ae.getSource() == cancel)
    owner.closeSubmission();
    else if (ae.getSource() == submit)
    { theBill.setInfo(name.getText(), credit.getText(),
                     address.getText());
      if (theBill.isComplete())
        submitOrder();
      else
        status.setText("NOT SUBMITTED: incomplete information.");
    }
  }
  private void submitOrder()
  { status.setText("Connecting to " + HOST);
    try
```

---

<sup>42</sup> Java certainly provides strong support for sending encrypted data, but it would go beyond the scope of this text to pursue this further.

```

    { Socket connection = new Socket(HOST, PORT);
      ObjectOutputStream out = new ObjectOutputStream(
          connection.getOutputStream());
      ObjectInputStream in = new ObjectInputStream(
          connection.getInputStream());
      out.writeUTF(CLIENT_ID); out.flush();
      if (in.readUTF().equals(SERVER_ID))
      { out.writeDouble(CLIENT_VERSION); out.flush();
        if (in.readDouble() == SERVER_VERSION)
        { status.setText("Connected. Submitting order ...");
          out.writeObject(theBill); out.flush();
          status.setText("Order submitted. Thank you.");
        }
        else
          status.setText("Invalid server version, contact Service");
      }
      else
        status.setText("Invalid server, contact Service");
      out.close(); in.close(); connection.close();
      submit.setEnabled(false);
      cancel.setLabel("Done");
    }
    catch(UnknownHostException uhe)
    { status.setText("Server " + HOST + " unknown.");}
    catch(IOException ioe)
    { status.setText("Error submitting order, please try again."); }
  }
}

```

The method assumes that the proper variables to identify matching clients and servers are the same as defined in the server class. The method connects to the server, sets up object input and output streams through the socket, sends and receives identification strings and numbers according to the protocol, and then sends the `CatalogOrder` object. When everything worked, the method disables the submit button and changes the label of the cancel button to "Done" to prevent people from submitting an order twice.

The interesting part of the example will be the server to receive the data. Our first approach will be to create a relatively simple, on-time server as follows:

```

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

public class JBDDOrderServer
{   protected final static String CLIENT_ID = "JBDDOrderClient";
    protected final static double CLIENT_VERSION = 1.0;
    protected final static String SERVER_ID = "JBDDOrderServer";
    protected final static double SERVER_VERSION = 1.0;
    protected final static String HOST = "localhost";
    protected final static int PORT = 5670;
    public static void main(String args[])
    {   try
        {   ServerSocket server = new ServerSocket(PORT);
            System.out.println("Server started ... waiting ..");
            Socket connection = server.accept();
            System.out.println("Connection established. Identifying ...");
            ObjectInputStream in = new ObjectInputStream(
                connection.getInputStream());
            ObjectOutputStream out = new ObjectOutputStream(
                connection.getOutputStream());
            if (in.readUTF().equals(CLIENT_ID))
            {   out.writeUTF(SERVER_ID); out.flush();

```

```

        if (in.readDouble() == CLIENT_VERSION)
        { out.writeDouble(SERVER_VERSION); out.flush();
          try
          { CatalogOrder theBill = (CatalogOrder)in.readObject();
            System.out.println(theBill);
          }
          catch(ClassNotFoundException cnfe)
          { System.err.println("Invalid class - contact support");
          }
        }
        else
          System.out.println("Invalid version - contact support");
      }
      else
        System.out.println("Invalid client - contact support");
      out.close(); in.close(); connection.close();
      System.out.println("Connection closed.");
      server.close();
    }
  catch(IOException ioe)
  { System.err.println("Error establishing server and streams"); }
}
}

```

Our server first defines the same constants as the applet to be used for identification by the server and the client. Then it binds to the port and waits for a connection. If a connection is established, the server tries to exchange identification information according to the protocol. Then it receives the `CatalogOrder` object and - for now - prints it to the screen. Finally the server closes the connection and exists completely.

This server already works with our client method, but it only handles exactly one connection, and it does not save the ordering information to a text file. To improve the class, we need to solve several problems:

**Problem:** The server should handle multiple client connections simultaneously:

**Solution:** we simply move the code that actually handles the connection to its own class, extending `Thread`, as outlined in definition 10.3.4.

**Problem:** The server needs to save the order to a file instead of printing it to the screen:

**Solution:** We add a method `writeOrder` to the server class (not the thread class we are going to setup) that the various threads can call upon to save the data to a file.

**Problem:** When several clients try to save data simultaneously, we need to make sure that the data file does not get corrupted by uncoordinated write requests.

**Solution:** We mark the `writeOrder` method as `synchronized`. Other threads trying to access `writeOrder` will be put on hold until the currently thread is done.

**Problem:** We need to find a format for the data file so that it can be imported into other programs for easy further processing:

**Solution:** We add a method `exportData` to the `CatalogData` class that returns the order data as a "tab-delimited" string. That way, the data file will consist of one line per order, each line consisting of "fields" separated by `TAB` characters. Such files can be imported directly into programs such as Microsoft Excel or Microsoft Access for further processing.

Let's implement the last solution first: we will add the following `exportData` method to the `CatalogOrder` class:

```

public class CatalogOrder implements Serializable
{ /* Fields as before */
    public void setInfo(String _name, String _credit, String _address)
    { /* no change, just as previously */ }
    public boolean isComplete()
    { /* no change, just as previously */ }
    public void add(CatalogItem item)
    { /* no change, just as previously */ }
    public String toString()
    { /* no change, just as previously */ }
    public String exportData()
    { String s = name + "\t" + credit + "\t";
      StringTokenizer splitter = new StringTokenizer(address, "\n");
      while (splitter.hasMoreTokens())
        s += splitter.nextToken() + "|";
      s += "\t" + list.size() + "\t" + totalCost + "\t";
      for (Iterator iterator = list.iterator(); iterator.hasNext(); )
        s += "Item " + ((CatalogItem)iterator.next()).getName() + "\t";
      return s;
    }
}

```

Next, we will split up our `JBDOrderServer` class so that it can handle multiple simultaneous connections via threads according to definition 10.3.4. The main server class would look as follows (modeled after our final `EchoServer` class):

```

public class JBDOrderServer
{ /* static final constants as before */
    private boolean terminated = false;
    public JBDOrderServer()
    { start(); }
    public void start()
    { try
      { ServerSocket server = new ServerSocket(PORT);
        System.out.println(SERVER_ID+" "+SERVER_VERSION+" ready ...");
        while (!terminated)
        { Socket connection = server.accept();
          JBDOrderServerThread handler =
            new JBDOrderServerThread(connection);
        }
        System.out.println(SERVER_ID + " shutting down.");
        server.close();
      }
      catch(IOException ioe)
      { System.err.println("Error establishing server and streams"); }
    }
    public static void main(String args[])
    { JBDOrderServer server = new JBDOrderServer(); }
}

```

The thread class handling the client connection basically contains the code from the old version of `JBDOrderServer` that used to handle the actual connection:

```

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

public class JBDOrderServerThread extends Thread
{ private Socket connection;
  public JBDOrderServerThread(Socket _connection)

```

```

    { connection = _connection;
      start();
    }
public void run()
    { try
      { System.out.println("Connection established. Identifying.");
        ObjectInputStream in = new ObjectInputStream(
            connection.getInputStream());
        ObjectOutputStream out = new ObjectOutputStream(
            connection.getOutputStream());
        if (in.readUTF().equals(JBDDOrderServer.CLIENT_ID))
        { out.writeUTF(JBDDOrderServer.SERVER_ID); out.flush();
          if (in.readDouble() == JBDDOrderServer.CLIENT_VERSION)
          { out.writeDouble(JBDDOrderServer.SERVER_VERSION);
            out.flush();
            try
            { CatalogOrder theBill = (CatalogOrder)in.readObject();
              System.out.println(theBill);
            }
            catch(ClassNotFoundException cnfe)
            { System.err.println("Invalid class"); }
          }
        }
        else
          System.err.println ("Invalid version");
      }
      else
        System.err.println ("Invalid client");
      out.close(); in.close(); connection.close();
      System.out.println("Connection closed.");
    }
    catch(IOException ioe)
    { System.err.println("IOException !"); }
  }
}

```

That will implement the solution mentioned as the first problem above. Now we are ready to solve the remaining problem of actually appending the data to a text file in the appropriate format. Only a few modifications will be necessary for the `JBDDOrderServerThread` class:

- We need to call on the `JBDDOrderServer` class to write the data to file, so we need a reference field to our server. We define that field in the constructor
- We need to change the line that currently prints the order to the screen so that it calls the `writeOrder` method we will implement in the server class

Specifically, here is the new code, with the changes in bold and italics:

```

public class JBDDOrderServerThread extends Thread
    { private JBDDOrderServer server;
      private Socket connection;
      public JBDDOrderServerThread(JBDDOrderServer _server,
        Socket _connection)
      { server = _server;
        connection = _connection;
        start();
      }
      public void run()
      { /* code as before, but replacing the line that originally
        said System.out.println(theBill); with the following: */
        server.writeOrder(theBill);
        /* all other code is exactly as before */
      }
    }

```

```
    }
```

The final modifications we need to perform involve the `JBDOrderServer` class:

- The class should open a character stream to an output file as soon as the server starts. Since we want to append data to our data file, we need to make sure we open the output stream in "append" mode.
- The class should contain a method `writeOrder` that takes as input a `CatalogOrder` and writes the order to the output stream, formatted by the `exportData` method we previously added to `CatalogOrder`. The method *must* be synchronized to prevent multiple threads from writing to the data file simultaneously.
- When the server shuts down, the output stream needs to be closed as well.

Specifically, our final version of `JBDOrderServer` will look like this, with the changes marked in bold and italics:

```
public class JBDOrderServer
{ /* all fields as before, plus */
  private PrintWriter out;
  public JBDOrderServer()
  { try
    { out = new PrintWriter(new FileWriter("JBDOrder.dat", true)); }
    catch (IOException ioe)
    { System.err.println("Error opening data output file(s)"); }
    start();
  }
  public synchronized void writeOrder(CatalogOrder order)
  { out.println(order.exportData());
    out.flush();
  }
  public void start()
  { try
    { ServerSocket server = new ServerSocket(PORT);
      System.out.println(SERVER_ID+" "+SERVER_VERSION+" ready ...");
      while (!terminated)
      { Socket connection = server.accept();
        JBDOrderServerThread handler =
          new JBDOrderServerThread(this, connection);
      }
      System.out.println(SERVER_ID + " shutting down.");
      server.close();
    }
    catch(IOException ioe)
    { System.err.println("Error establishing server and streams"); }
    finally
    { out.close(); }
  }
  public static void main(String args[])
  { JBDOrderServer server = new JBDOrderServer(); }
}
```

This will finish the code implementation part of our program. Now we need to test everything to make sure it all works. The important thing to remember is that we added a method to an applet that wants to establish a network connection to a server. According to the security restrictions on applets that will only work if the server and the client are located *on the same machine*. We have defined the server in the above classes as "localhost" so that we can run both the server and the client on a local computer for testing purposes. If you move the applet to a web server, you *must also move the server to the same machine* and start it on that machine. Also, *don't forget to change the*

*server name to the name of the machine* where you are moving everything; otherwise the client will not be able to connect to the server.<sup>43</sup>

For now, we will run our complete client/serve online ordering system on a local computer (i.e. you can leave the host name as "localhost"). Make sure that all classes exist in the same directory, then start the `JBDOrderServer` class via the command:

```
java JBDOrderServer
```

The server will start up, waiting for a connection. Next, open another DOS window (if you are using Windows) and start the `Catalog` applet via the `appletviewer` (or use Netscape) on the same local computer you started the server. Fill out a few orders to see if everything works according to plan, as shown in figure 10.3.4.

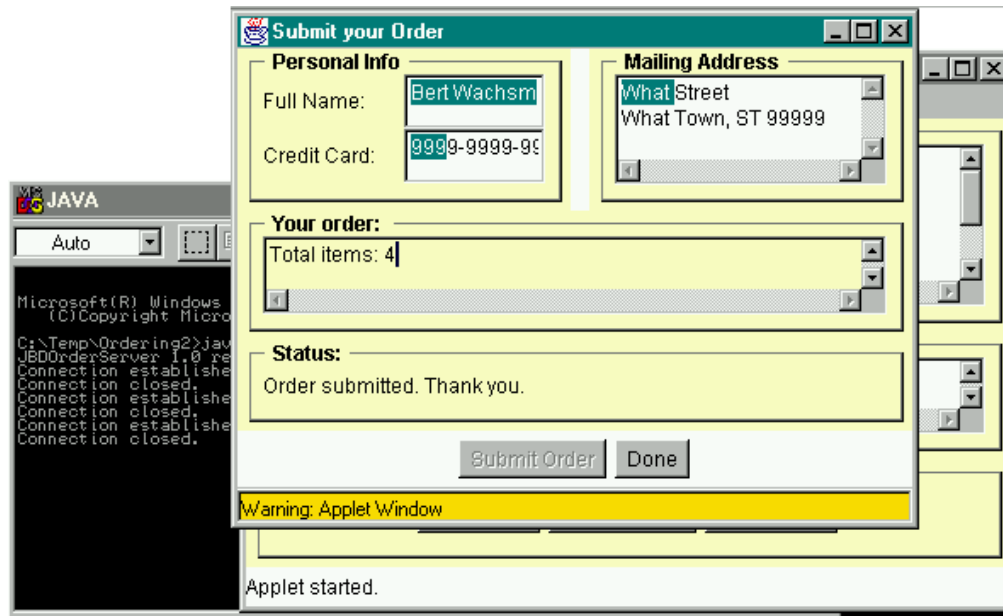


Figure 10.3.4: The `JBDOrderServer` has just received an Order from the `Catalog` applet

In figure 10.3.4 you can see that the server is running in a separate DOS box and it has received three client connections so far. The most recent version of the order client applet is still visible, and has just submitted an order. Next, we'll stop the server to open the data file using, for example, a spreadsheet or any other program that can import tab-delimited data. Note that we *must* shutdown the server before loading the data into another program, otherwise the operating system will not let you open one file with two different programs.

We first close `JBDOrderServer` by pressing `CONTROL-C` while the DOS window is active. Then we open, say, Microsoft Excel to read the data (the process of importing a tab-delimited text file into Excel is left to the reader). The result is shown in figure 10.3.5.

<sup>43</sup> In order to run our `JBDOrderServer` on a web server machine, the Java runtime system must be installed on that machine and you must be able to issue the appropriate command line to start the server (that usually means that you need `telnet` access to the web server).

	A	B	C	D	E	F
1	John Doe	1234-5678-9012	12 Some Street Some Town, ST 12345	1	2.99	Item 001: Pepsi
2	Jane Doe	2109-8765-4321	Some Other Street Some Other Town, \$	3	6.57	Item 001: Pepsi
3	Bert Wachsmuth	9999-9999-9999	What Street What Town, ST 99999	4	7.66	Item 004: Hot Dog

Figure 10.3.5: Data from the `JBDOrderServer` imported to Microsoft Excel

If you have trouble opening the data file with another program such as `Excel`, you can try opening it with the same text editor you are using to create your Java source code files.

To *really* test the client/server solution we came up with, however, we would need to move both the server and the applet to an actual web server and test it with several people simultaneously. Obviously we can not simulate that in this text, so we will leave it as an exercise for the reader. Another approach to test our client/server solution would be to open multiple DOS boxes and start the Catalog applet in each DOS box (if you are using Windows) on one machine - again, this is left as an exercise. ■

There are still several possible improvements to our online ordering system that we could implement. We will leave them as exercises so that we can start the final client/server example of this chapter.

## 9.4. Security Issues

By now it should be clear that every server program that listens to a particular program poses a security risk to the system hosting the server. If a port is open (i.e. a server is running that listens on that port), then *any* client can connect to that port, whether they are using the appropriate protocol or not. The client could then attempt to trick the server into performing actions that may have ill effects on the entire system. This is especially true if the server program performs actions to write legitimate information to disk, and even more true if the server is running as a user that has special privileges on the system hosting the server.

Older versions of the popular SMTP `sendmail` program, for example, contain well-known "security holes". That means that there is a well-known sequence of actions that a malicious client can use to coax the `sendmail` server into performing tasks that will eventually give complete and unrestricted access to the entire system hosting the server. The easiest solution to this problem was to turn off the `sendmail` SMTP server, but that also meant that this particular system could no longer send or receive any electronic mail. More recent versions of `sendmail` have fixed this particular security problem, but Murphy's Law<sup>44</sup> dictates that security issues will be a an ongoing area of concern for *any* server program.

As far as the servers are concerned that we constructed in sections 10.3 and 10.4, the `DirServer` from example 10.3.5 is a particular example of how you can use Java's Internet abilities to compromise the security of an otherwise secure system. That server has two safeguards: it requires a password before it reveals the directory structure of the server system, and it uses byte and object streams so that a Telnet client would not be able to easily communicate with a `DirServer`. Even with

<sup>44</sup> Murphy's Law: Anything that *can* go wrong, *will* go wrong.

these safeguards in place it is not advisable to run the `DirServer` on a production-level system for longer periods of time.

The `DirServer` also illustrates that web server administrators that allow users to start Java programs on that server are faced with the problem that users could create an insecure client/server package and compromise the entire machine. The obvious but restrictive solution is not to allow users to start *any* Java programs on the host system.<sup>45</sup> A more measured approach would be to block access to all ports except those where certified servers are running so that a user could not create a server program that can successfully bind to a port. That approach, in fact, is frequently taken by placing a web server behind a firewall, which blocks access to all but a few registered ports from systems outside the own domain.

It is noteworthy to mention that standard http web servers - in particular the Apache web server - that are providing the content for the World Wide Web have not yet been exposed to any major security problems. That is particularly surprising because web servers, by their very nature, allow any client to connect without asking for any verification such as usernames and passwords, and the http protocol is quite complex, leaving plenty of possibilities for things to go wrong.

In the exercises you will be asked to create a "port sniffer" program that will find all open ports on a particular server. Such a program, of course, exists already (it is not easy to come up with a useful program that does *not* already exist) and is available on the Internet. Port sniffers are a frequent starting point for "hackers" interested in breaking in to a particular system. You should be careful if you execute such a program to look at a particular machine, since your access might be monitored and misunderstood as an attempt to break in. Legal consequences could ensue even if you did not mean to do any harm.

## Case Study: The `ChatterBox` Client/Server Example

We have already seen client/server examples where multiple clients connected to a single server, and in our `JBDOrderServer` class in example 10.3.7 we used a synchronized method to prevent multiple clients to interfere with each other's actions. In the last example of this chapter we will see another multi-client example that uses synchronization, but this time the client will also use threads. In addition, the server will keep track of all running threads and can transfer information from one thread to another.

### ***Example 10.4.1:***

Create a multi-user chat client/server solution. In other words, create a chat server program that can handle multiple clients so that everything that is typed in one client is transmitted to all other connected programs. The client should be based on a `Frame` and work either stand-alone or as an applet. For those familiar with chat rooms, our example only needs to emulate one "room", i.e. people connected to the server will see all messages typed by anyone.

---

<sup>45</sup> This would be difficult to enforce if a web server system also allows Telnet access. After all, a user could download the JDK to their local directory on the server and compile and run programs through that JVM. If Telnet access was not possible, Java server programs can not be started, but the entire system may then be of limited use to advanced users.

In this example we want to create client classes so that any number of users who have loaded the client class via their web browser can type some text at any time. That text should be transmitted to all connected users where it is placed in an appropriate output area for reading. In addition to creating a multi-client server - which we outlined in definition 10.3.4 - this example presents us with three new problems:

- The server needs to keep track of all clients so that a message from one can be transmitted to all. The server, therefore, has to be able to add and remove clients to and from some list. Previously our multi-client server could handle clients in independent thread, i.e. communications only had to flow from one server thread to one client. Now communications needs to flow from any one server thread to all clients.
- The server needs to make sure that all communications happens in a synchronized fashion. For example, if two clients attempt to simultaneously send information, the server needs to determine who goes first and who has to wait so that the messages do not get mixed up. Also, messages should not be sent while a client attempts to connect or disconnect until the new status of the client can be determined.
- The client needs to be ready to receive text at any time from another client. Previously a client only received information from one server thread, and the protocol was easy: the client sent a string and waited for a response. Now the client needs to send strings at any time (as requested by the user) and also be able to receive strings at any time (from any other client)

In addition, this program could be quite useful, and we want to design it so that it is easily extendable. Some features that are pretty standard for chat client/server packages include:

- the ability to talk to only one user instead of the default mode of broadcasting to all users
- the ability to send URL's to clients in such a way that the underlying web browser displays the URL sent
- the ability to organize the chat sessions in multiple rooms, i.e. only users who are in the same chat room will share information with each other
- the ability to exclude people from chat conversations in case they broadcast offensive or inappropriate remarks
- the ability to save a log of a chat conversation for later reference
- the possibility of placing a chat room in "auditorium" mode where only one privileged chat user can speak without being interrupted by others
- the ability to transfer files while conducting a conversation

To be sure, these features and many more are implemented in the Internet Relay Chat service, which allows thousands of users to communicate in hundreds of chat rooms all over the world. IRC (Internet Relay Chat) is one of the standard services provided by many Internet hosts and there is plenty of free software available to connect to IRC. In our case, we want to provide only a subset of features offered by standard IRC software to get a basic understanding of the complexities involved in this mode of communications. We will ask in the exercises to extend our basic example to include some of the features mentioned above, but for now let's start working on the basic client/server solution for this exercise.

As with our `JBDOrderServer` and `EchoServer` example, we need to create three classes:

- **ChatterServer:** The base server class. It will keep track of all client connections, add and remove clients to a list of current clients, and ensure that all communications happens in a synchronized fashion.
- **ChatterThread:** A class extending `Thread` to handle communications with a particular client. It will receive strings from one client and forward them to the `ChatterServer` class for

distribution to other clients. It will also provide a method that can in turn be used by the `ChatterServer` to send strings to the attached client.

- **ChatterBox:** The client class with which the user interacts and that connects to a `ChatterThread`. It needs an appealing interface, including an area for the user to enter new strings, a display area to read strings received from the attached `ChatterThread`, and a `List` to show the names of currently attached users.

As usual, before we can begin implementing these classes, we need to specify exactly what protocol our clients and servers must follow to understand each other.

### The ChatterBox Protocol

We will divide the protocol into two pieces: the first one is how exactly to establish a connection, and the second one is what commands client and server will understand.

To establish a successful connection:

- All information transferred is text-based <sup>46</sup>
- Server waits for a client to connect
- Client connects to server and sends the string "Chatterbox"
- Server receives the string and responds with "ChatterServer"
- Client receives the string and sends a string representing the name of the chat user. Each client must have a unique username, i.e. no two clients can use the same name
- Server receives the string and checks if that user name already exists. If not, the new client connection is accepted. Otherwise, the server sends a "term" command with an appropriate message (see next)

If this protocol is not adhered to, the connection is terminated. If everything is in order, the new connection is added to the current list of valid connections. The information exchange after the initial identification is defined as follows:

- All information transferred is text-based
- All strings transmitted to the client after a successful connection is established consist of two pieces: the first piece is a four-letter command, followed by a colon, then a string representing a parameter or option for the particular command.
- The commands sent by the server that the client understands are:
  - `text:message`  
specifies to display message in the client's output area
  - `term:message`  
tells client to terminate connection (`message` indicates the reason)
  - `list:delimited-user-list`  
indicates that a list of currently connected user is given in `user-list` (the list is delimited by '|' characters)
- The commands sent by the client that the server are:
  - `addu:username`  
specifies to add the user who sent the command to the list of valid users
  - `delu:username`

---

<sup>46</sup> This means that as long as we use the protocol as specified we could use a `Telnet` connection to participate in a chat session with this server.

- specifies to delete the user who sent the command from the list of valid users
- term:username
  - indicates that a term command should be sent to the client who sent the command
- text:message
  - specifies to broadcast message to all valid users

Based on this protocol we can design the basic functionality of our classes as shown in figure 10.4.1.

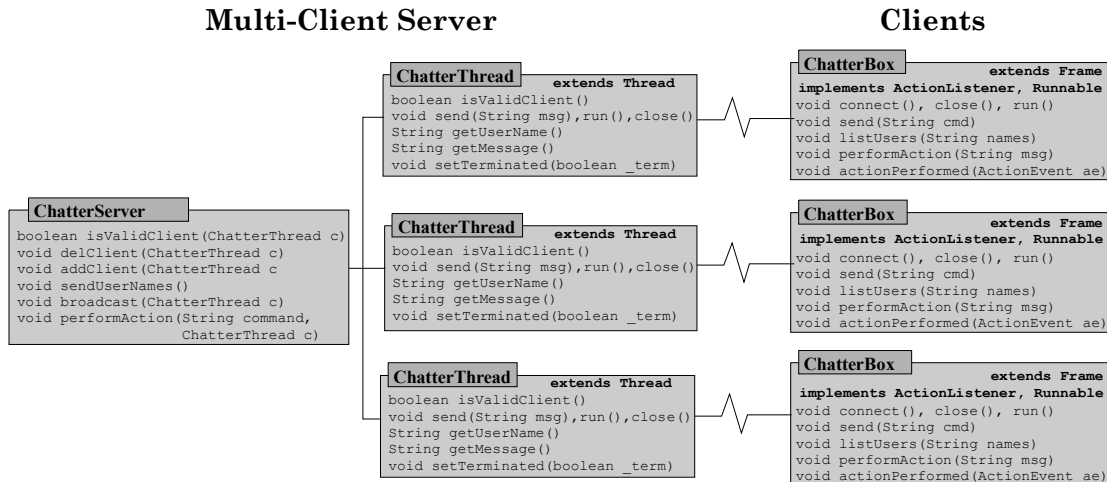


Figure 10.4.1: A representation of the ChatterBox client/server System

### The ChatterBox Server Classes

This time the base class for the server will be more complicated than in examples 10.3.3, 10.3.5, and 10.3.7 because the server needs to manage a list of existing threads. The basic outline for our two classes, as sketched in figure 10.4.1, is as follows:

ChatterServer Class	ChatterThread Class
<pre> import java.io.*; import java.net.*; import java.util.*; public class ChatterServer {     protected static final String ID         = "ChatterServer";     protected static final String CLIENT_ID         = "ChatterBox";     private LinkedList users         = new LinkedList();     private boolean terminated = false;     public ChatterServer(int port)     private boolean isValidClient         (ChatterThread client)     private void delClient         (ChatterThread client)     private void addClient         (ChatterThread client)     private void sendUserNames()     private void broadcast         (ChatterThread client)     public synchronized void performAction         (String action,          ChatterThread client)     public static void main         </pre>	<pre> import java.io.*; import java.net.*; public class ChatterThread     extends Thread {     private ChatterServer server;     private Socket connection;     private PrintWriter out;     private BufferedReader in;     private String user, message;     private boolean terminated         = false;     public ChatterThread         (ChatterServer _server,          Socket _connection)     private boolean isValidClient()     private close();     public void run()     public String getUsername()     public String getMessage()     public void setTerminated         (boolean _terminated)     public void send(String msg)         </pre>

```

    (String args[])
}

```

Table 10.4.2: Outline of `ChatterServer` and `ChatterThread` classes

The `ChatterServer` instantiates a `ServerSocket` that listens to a specified port. It contains a `List` of `ChatterThread` objects, which is initially empty. When a client connects, it starts a `ChatterThread` that exchanges connection information with the client. If successful, the thread stores the user name, asks the `ChatterServer` to add itself to the list of valid clients, and enters its `run` loop. The `run` loop waits for incoming strings and processes them by separating the incoming string into a command and a parameter piece. The command piece is passed to the `ChatterServer` for processing, the option is stored in a message string in the thread. The server acts on the command, calling one of the methods `addClient`, `delClient`, or `broadcast`. To communicate with a client, the `ChatterServer` can use the thread's `send` message. If the server receives a `term` command, it forwards it to the appropriate thread that in turn sends it to the attached client. Also, the server calls its `delClient` method to remove the client from the list of valid clients.

All action of the threads is channeled through the `performAction` method of `ChatterServer`. Hence, if we mark that method as `synchronized`, we can ensure that messages sent by the various threads can not interfere with each other. As soon as one thread uses the `synchronized` `performAction` method, all other threads are put on hold until the method has performed its intended action.

```

ChatterServer
final String ID = "ChatterServer";
LinkedList users = new LinkedList();
boolean terminated = true;

ChatterServer(int port)
boolean isValidClient(ChatterThread c)
void delClient(ChatterThread c)
void addClient(ChatterThread c)
void sendUserNames()
void broadcast(ChatterThread c)
void performAction(String command, ChatterThread c)
static void main(String args[])

```

Figure 10.4.3: A representation of the `ChatterServer` class

So, now that we have established the basic functionality of our classes, we need to implement them. Let's start with the `ChatterServer`.<sup>47</sup>

The constructor instantiates a `ServerSocket` and basically enters an infinite loop, waiting for a connection. If a client connects, it starts a new instance of a `ChatterThread` but without adding it to the list of valid clients. After all, the client may be invalid, attempting to use a name that is already connected to the server:

```

public ChatterServer(int port)
{ ChatterThread cbt = null;
  try
  { ServerSocket server = new ServerSocket(port);
    System.out.println("ChatterBox Server started, waiting ...");
    while (!terminated)
      cbt = new ChatterThread(this, server.accept());
  }
  catch(IOException ioe)

```

<sup>47</sup> Note that all methods but `performAction` and `main` are marked `private` and `performAction` is marked `synchronized`. That way we can ensure that client threads will access the various methods of the `ChatterServer` one client at a time, because all access must be routed through a `synchronized` method.

```

    { System.err.println("Unexpected Error: " + ioe); }
}

```

The method `isValidClient` is provided so that a thread can use it to check other threads already added to the list for a duplicate user name. If the username of the input thread is unique the method returns `true` otherwise it returns `false`:

```

private boolean isValidClient(ChatterThread client)
{
    for (Iterator iterator=users.iterator(); iterator.hasNext(); )
    {
        ChatterThread current = (ChatterThread)iterator.next();
        if (current.getUserName().equals(client.getUserName()))
            return false;
    }
    return true;
}

```

The next methods are simple: `delClient` removes the input client from the list of valid clients, informs the input client that it should terminated the thread's run method, and sends the user names to all remaining clients to inform them of a different user list. The `addClient` method checks whether the client is valid via `isValidClient`, adds the client to the list if it is valid, and sends the new user list. If the client is invalid (i.e. has a duplicate user name) then `addClient` sends a term command with an appropriate message to the client.

```

private void delClient(ChatterThread client)
{
    users.remove(client);
    client.setTerminated(true);
    sendUserNames();
}

private void addClient(ChatterThread client)
{
    if (isValidClient(client))
    {
        users.add(client);
        sendUserNames();
    }
    else
        client.send("term:Duplicate User Name - use other name");
}

```

The `sendUserNames` method iterates through the list of valid clients and constructs a string of user names delimited by the '|' character. After collecting all names, it again iterates through all clients to send them the new user list<sup>48</sup>:

```

private void sendUserNames()
{
    String names = "";
    for (Iterator iter = users.iterator(); iter.hasNext(); )
        names += ((ChatterThread)iter.next()).getUserName() + "|";
    for (Iterator iter = users.iterator(); iter.hasNext(); )
        ((ChatterThread)iter.next()).send("list:" + names);
}

```

Our second to last method `broadcast` iterates through the list of clients, gets the message string stored in the input client thread, and sends that message to all other clients. It prefaces the message by the user name of the input client so that the other clients can tell who sent the message:

```

private void broadcast(ChatterThread client)

```

---

<sup>48</sup> This method could be improved by modifying `addClient` and `delClient` so that those methods ensure that the string representing the user list is always current, which would save one iteration through the list. For now this version will be sufficient (but not efficient).

```

{ String msg = client.getUserName() + "> " + client.getMessage();
  for (Iterator iter = users.iterator(); iter.hasNext(); )
    ((ChatterThread)iter.next()).send("text:" + msg);
}

```

Finally there is the `performAction` method. That method is somewhat similar to the `actionPerformed` method of GUI-based programs. For those programs, action events are generated by various GUI elements such as buttons and the events are passed to the `actionPerformed` method for processing. In our program the "events" are generated by the various client threads and are routed through the `performAction` method for processing. If we mark this method as `synchronized` we will ensure that one thread will be able to finish its action before another thread is allowed access to `performAction`. Thus we can ensure the integrity of our client list as well as make sure that messages sent to the various clients can not interfere with each other. Other than this simple trick, the method is easy:

```

public synchronized void performAction(String cmd, ChatterThread client)
{ if (cmd.equals("addu"))
  addClient(client);
  else if (cmd.equals("delu"))
  delClient(client);
  else if (cmd.equals("text"))
  broadcast(client);
  else if (cmd.equals("term"))
  { client.send("term:Connection terminated - good bye");
    delClient(client);
  }
}

```

And of course there is the obligatory `main` method which instantiates a `ChatterServer`, taking the port to be used from the command line (make sure to enter an integer, preferably above 1024, when starting the program).

```

public static void main(String args[])
{ ChatterServer cbs = new ChatterServer(Integer.parseInt(args[0])); }

```

The `ChatterServer` contains the methods that actually do the "work" of our server. Communication with the individual clients, however, is handled by `ChatterThread` objects.

<b>ChatterThread</b>	<b>extends Thread</b>
<pre> ChatterServer server; Socket connection; PrintWriter out; BufferedReader in; boolean terminated; String user, message;  ChatterThread(ChatterServer _server, Socket _connection) boolean isValidClient() void send(String msg), run(), close() String getUserName() String getMessage() void setTerminated(boolean _term) </pre>	

*Figure 10.4.4: A representation of the ChatterThread class*

These classes extend `Thread` and therefore run independently of each other, but since they call on the `synchronized performAction` method of `ChatterServer` they act in a synchronized fashion. The idea of `ChatterThread` is easy, consisting basically of four major methods:

- **constructor**: initializes some variables and uses `isValidClient` to check whether the client is indeed a proper `ChatterBox` client (if so, it starts the thread)
- **isValidClient**: checks whether the client is valid by exchanging "handshake" messages according to our protocol
- **run**: listens for an incoming string, separates it into command and option part, and passes the command part to `performAction` of `ChatterServer`
- **send**: sends the input string to the attached client

We have already outlined the class in table 10.4.2 and figure 10.4.4 so we can start implementing the various methods. The constructor will define certain variables, then call on `isValidClient` to determine whether the client that connected to the server is the right type. If so, it sends an "addu" command to the server class to add this thread to the list, then it starts the thread:

```
public ChatterThread(ChatterServer _server, Socket _connection)
{ System.out.println("Starting connection thread ...");
  server = _server;
  connection = _connection;
  if (isValidClient())
  { server.performAction("addu", this);
    start();
  }
  else
    System.out.println("Invalid client. Connection closed.");
}
```

The `isValidClient` method defines proper character-level input and output streams through the socket and attempts to exchange identification information according to the connection protocol. Recall that the client will send an ID string, the server - i.e. our thread - checks whether that string is "ChatterBox", the server then sends "ChatterServer" and waits for a string representing the user name. If everything works as advertised, the method returns `true`, otherwise it closes all streams and returns `false`. In particular, if this method returns `true` the proper input and output streams are open:

```
private boolean isValidClient()
{ try
  { out = new PrintWriter(new OutputStreamWriter(
    connection.getOutputStream()));
    in = new BufferedReader(new InputStreamReader(
    connection.getInputStream()));
    String remoteID = in.readLine();
    if ((remoteID != null) &&
        (remoteID.equals(ChatterServer.CLIENT_ID)))
    { send(ChatterServer.ID);
      user = in.readLine();
      return true;
    }
    else
    { close();
      return false;
    }
  }
  catch(IOException ioe)
  { return false; }
}
```

The utility method `close` will close the input and output streams and the socket, and absorb any exceptions that might be thrown in the process:

```

private void close()
{ try
  { in.close(); out.close(); connection.close(); }
  catch(IOException ioe)
  { System.err.println("ERROR closing stream."); }
}

```

The run method is, actually, simple. It waits to receive a string from the input stream, then checks whether the string is valid and has the appropriate length. If not, the method terminates, otherwise it breaks the string into its command and option parts, stores the option information in the field message, and calls the performAction method of the ChatterServer with the command as argument. If an exception occurs, the method prints out an appropriate error message. Also, the method implements the finally phrase of a try-catch statement to send a "delu" command to the server to ensure that when the run method terminates, the client thread is removed from the list of valid clients. Note that if the client properly sends a term command, it should already be removed from the list via the call to server.performAction, and the additional call to server.performAction in finalize should have no effect. But clients may also disconnect involuntarily, so adding a call to server.performAction("delu") will ensure that the client will be removed from the list of valid clients no matter how it terminates.

```

public void run()
{ System.out.println("ChatterThread run method starts");
  try
  { while (!terminated)
    { String line = in.readLine();
      if ((line == null) || (line.length() < 5))
        terminated = true;
      else
      { message = line.substring(5,line.length());
        server.performAction(line.substring(0,4), this);
      }
    }
  }
  catch(IOException ioe)
  { System.err.println("IO Exception in run. Terminating ..."); }
  finally
  { server.performAction("delu", this);
    System.out.println("ChatterThread run done");
  }
}

```

The remaining methods are simple:

- `getUserName` returns whatever string is currently in message,
- `getUserName` returns the user name set in the `isValidClient` method
- `send` transfers the input string through the output stream and flushes the stream to ensure that the data is actually sent and not kept in the stream's buffer
- `setTerminated` changes the field `terminated` which, when set to true, will cause the thread to finish its run method

```

public String getUserName()
{ return user; }
public String getMessage()
{ return message; }
public void setTerminated(boolean _terminated)
{ terminated = _terminated; }
public void send(String msg)
{ out.println(msg); out.flush(); }

```

At this stage, we could already test our server, using `telnet` as the client program. After all, our server uses an entirely text-based protocol, and `telnet` can handle text-based server connections perfectly fine.

First, we need to start our server by typing (e.g. in a DOS box under Windows):

```
java ChatterServer 1234
```

(or some equivalent unused port). Next, we open a `telnet` connection to the `localhost` on port 1234. Once the connection is established, we need to type "ChatterBox", then a user name to connect properly (each line terminated by hitting `ENTER`). After that we can send strings to the server for broadcasting by typing, for example, "term:This line will be sent.". To close the connection, we would type "term:me". Below is a screen shot of two `telnet` connections to our `ChatterServer`.<sup>49</sup>

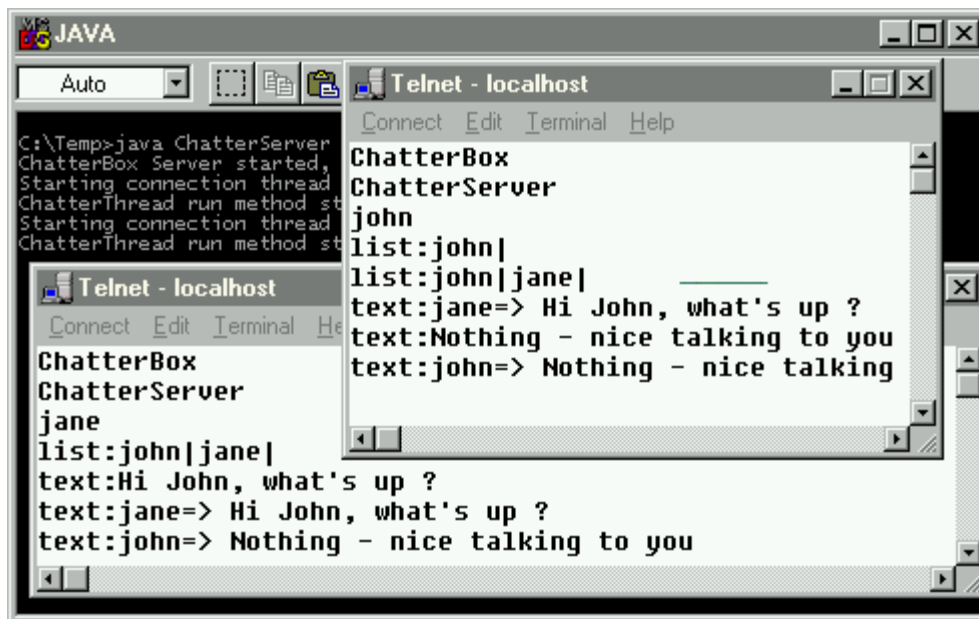


Figure 10.4.5: Two Telnet clients connected to a ChatterServer

While this certainly works it is not a very intuitive way to chat. Therefore, we need to create a client program that will use GUI elements to hide the complexities of our protocol from the user.

### The ChatterBox Client Class

The idea for the client class derives immediately from the way we have setup the server protocol as well as from our need for a graphical user interface:

- The class needs to extend either `Applet` or `Frame` to provide a convenient GUI interface. We will choose a `Frame` and a stand-alone program so that our `ChatterBox` client can connect to servers on any machine.<sup>50</sup>

<sup>49</sup> Note that we have turned "local echo on" in the "Terminal | Preferences" menu of our `Telnet` clients so that you can see what has been typed as well as what the server sent. Otherwise we would not see what is typed in a `Telnet` window but it will be sent to the server nonetheless when pressing `ENTER`.

<sup>50</sup> Applets can only connect to the machine they came from because of the applet security restrictions.

- The class needs text-based input and output streams through a socket that is connected to a `ChatterThread`
- The class needs a method to send outgoing text to the server, or more precisely to an attached `ChatterThread` object
- The class needs to be able to receive incoming text at any time from the attached `ChatterThread` object so it will implement `Runnable` and use a `Thread` to listen to incoming text
- The class needs to interpret the incoming text and act accordingly

As far as the GUI elements are concerned, we will use `TextField` for the user name and the outgoing text, a `TextArea` to display incoming text, a `List` for the names of all attached users, buttons to issue "send text" and "connect/leave" commands, and a `Label` to display status messages. We also use an inner class `WindowCloser` to close the frame.

```

ChatterBox                                extends Frame
implements ActionListener, Runnable
// GUI Fields:
TextField input, user; Button send, connect;
List users; TextArea output; Label status;
// Client Fields:
Socket connection; PrintWriter out, BufferedReader in;
Thread thread; boolean terminated; String name;
String host; int port
// Inner class:
private class WindowCloser extends WindowAdapter
ChatterBox(String _host, int _port)
void setup(), void connect(), void close(), void run()
void send(String cmd), void listUsers(String names)
void actionPerformed(String msg)
void actionPerformed(ActionEvent ae)
void main(String a[])

```

Figure 10.4.6: A representation of the `ChatterBox` class

The remaining fields (Socket, Thread, streams, etc) should be clear, so here is the outline of the class:

```

import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.util.StringTokenizer;

public class ChatterBox extends Frame implements Runnable, ActionListener
{
    private final static String ID = "ChatterBox";
    private final static String SERVER_ID = "ChatterServer";
    // GUI fields
    private TextField input = new TextField(), user = new TextField();
    private Button send = new Button("Send"), connect = new Button("Connect");
    private List users = new List();
    private TextArea output = new TextArea();
    private Label status = new Label();
    private class WindowCloser extends WindowAdapter { }
    // functional fields to communicate
    private Socket connection = null;
    private PrintWriter out = null;
    private BufferedReader in = null;
    private Thread thread = null;
    private boolean terminated = true;
    private String name, host;
    private int port;
}

```

```

// constructor and methods
public ChatterBox(String _host, int _port)
private void setup()
private void connect()
private void send(String cmd)
private void close()
private void listUsers(String names)
public void run()
public synchronized void performAction(String msg)
public void actionPerformed(ActionEvent ae)
public static void main(String a[])
}

```

Note that the `performAction` method is synchronized. After all, the class contains an independently executing thread that can receive text at any time and pass it to that method for processing. Marking it as synchronized will prevent potential coordination problems. Also, we import a variety of classes, including the `StringTokenizer` class. We will need that class in the `listUsers` method to break apart the delimited string of user names.

Let's start implementing the various methods, starting with the constructor. Actually, the constructor simply delegates its work to the `setup` method:

```

public ChatterBox(String _host, int _port)
{
    super(ID);
    host = _host;
    port = _port;
    setup();
}

```

The `setup` method will layout the various GUI components, using our previously defined `PanelBox` to give the program a more appealing look<sup>51</sup>:

```

private void setup()
{
    Panel inputPanel = new Panel();
    inputPanel.setLayout(new BorderLayout());
    inputPanel.add("Center", input); inputPanel.add("East", send);
    Panel statusPanel = new Panel();
    statusPanel.setLayout(new BorderLayout());
    statusPanel.add("Center", new PanelBox(status, "Status"));
    statusPanel.add("East", new PanelBox(connect, "Action"));
    Panel userPanel = new Panel();
    userPanel.setLayout(new BorderLayout());
    userPanel.add("North", new PanelBox(user, "Your name"));
    userPanel.add("Center", new PanelBox(users, "User list"));
    setLayout(new BorderLayout());
    add("South", new PanelBox(inputPanel, "Input"));
    add("Center", new PanelBox(output, "Output"));
    add("East", userPanel);
    add("North", statusPanel);
    send.addActionListener(this);
    connect.addActionListener(this);
    send.setEnabled(false);
    status.setText("ChatterBox version 1.0, (c) BGW");
    addWindowListener(new WindowCloser());
    validate(); pack(); setVisible(true);
}

```

---

<sup>51</sup> The `PanelBox` class is defined in example 9.6.1 and puts a 3D-border with a title around a component.

Note that the class adds an instance of our inner class `WindowCloser` as a `WindowListener` to the frame. The setup method will produce the look shown in figure 10.4.7:

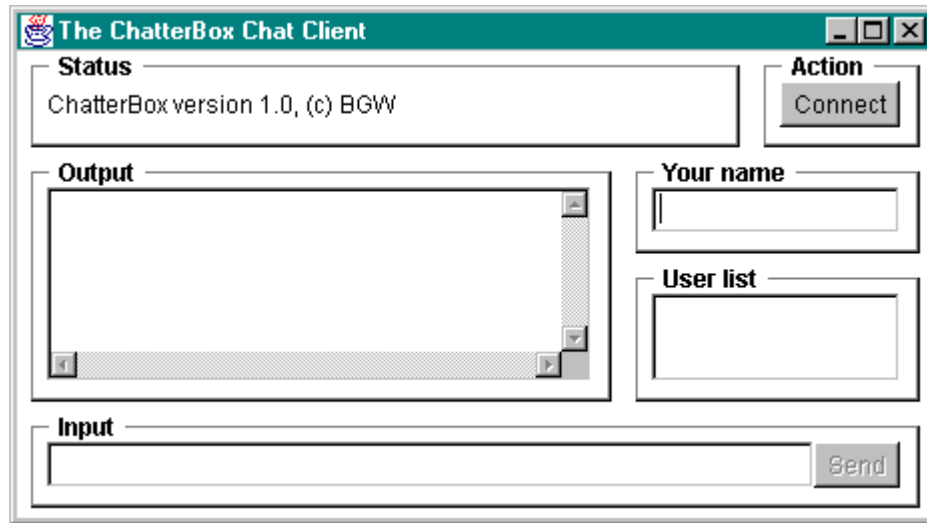


Figure 10.4.7: Layout generated by the setup method of the `ChatterBox` class

Our class contains *two* methods that can cause action. The first is the standard `actionPerformed` method that reacts to a click on the buttons. If the `connect` button is pressed the method checks the label of the button and calls `connect` or sends a `term` command, respectively. The `connect` button, therefore, serves a dual purpose: if currently not connected it allows the user to connect; if currently connected it allows the user to disconnect:

```
public void actionPerformed(ActionEvent ae)
{
    Object src = ae.getSource();
    if ((src == connect) && (connect.getLabel().equals("Connect")))
        connect();
    else if ((src == connect) && (connect.getLabel().equals("Leave")))
        send("term:" + name);
    else if ((src == send) && (!input.getText().equals("")))
        send("text");
}
```

The `actionPerformed` method does not need to be marked as `synchronized` because a user can not press two GUI elements simultaneously, so action events are generated one at a time.

The second method that causes action reacts to the input received from the thread. It might be possible that the thread receives message in quick succession without giving the `performAction` method a chance to finish its work before receiving a new line. Therefore, the message is marked as `synchronized`. It first checks if the input string is sufficiently long to be a valid string according to our protocol (if not, it terminates the thread receiving new messages). Then it interprets the commands:

- if input string contains a `term` command: the method sets `terminated` to `true`, which will terminate the thread receiving new message and shows an appropriate status message
- if input string contains a `list` command: the method calls the `listUser` method to show the user names received in the second "option" part of the input string
- if input string contains a `text` command: the method appends the second part of the input string to the output area:

```

public synchronized void performAction(String msg)
{  if ((msg == null) || (msg.length() < 5))
    terminated = true;
    else if (msg.startsWith("term"))
    {  terminated = true;
        status.setText(ID + ": " + msg.substring(5,msg.length()));
    }
    else if (msg.startsWith("list"))
        listUsers(msg.substring(5,msg.length()));
    else if (msg.startsWith("text"))
        output.append(msg.substring(5,msg.length()) + "\n");
}

```

This method can easily be expanded to understand additional commands if the protocol was modified to allow for more possibilities (see exercises).

What remains is to implement the various utility methods called upon by the above "action" methods. Let's start with the `connect` method. It first checks if a user name has been entered in the appropriate text field. If so, it establishes input and output streams, sends the client ID, waits for and checks the server ID and sends the user name, all according to the connection protocol. If all goes well, the method starts the `run` method of the thread receiving incoming messages, changes the label of the `connect` button, activates the `send` button, and puts the cursor in the input field for the user's convenience:

```

private void connect()
{  if (!user.getText().equals(""))
    {  try
        {  connection = new Socket(host, port);
            out = new PrintWriter(new OutputStreamWriter(
                connection.getOutputStream()));
            in = new BufferedReader(new InputStreamReader(
                connection.getInputStream()));
            out.println(ID); out.flush();
            if (in.readLine().equals(SERVER_ID))
            {  name = user.getText();
                out.println(name); out.flush();
                terminated = false;
                thread = new Thread(this);
                thread.start();
                send.setEnabled(true);
                connect.setLabel("Leave");
                input.requestFocus();
            }
        }
        else
            close();
    }
    catch(UnknownHostException uhe)
    {  status.setText(ID + ": Host unknown"); }
    catch(IOException ioe)
    {  status.setText(ID + ": Connection refused or no server "); }
}
else
    status.setText(ID + ": Please enter your user name first ...");
}

```

The `send` method pushes the string contained in the input field (if any) through the output stream and resets the input field for further input:

```

private void send(String cmd)
{  out.println(cmd + ":" + input.getText()); out.flush();
}

```

```

        input.setText("");
        input.requestFocus();
    }

```

The `run` method, attached to the thread started in the `connect` method, displays information about the status of the connection and enters its main loop. In the loop it waits for text coming in through the input stream and passes any information to the `performAction` method for processing. The method needs a standard `try-catch` block, but also uses a `finalize` part to ensure that all streams are properly closed when the `run` method terminates:

```

public void run()
{
    status.setText(ID+": ["+name+"] connected - type text click SEND");
    try
    {
        while (!terminated)
            performAction(in.readLine());
    }
    catch(IOException ioe)
    {
        status.setText(ID + ": unknown exception in run method !");
    }
    finally
    {
        close();
    }
}

```

When the loop of the `run` method is over, the `finalize` clause calls `close` to properly terminate the connection. That means closing all open streams, erasing the user list, clearing the output area, disabling the `send` button, and changing the label of the `connect` button to "Connect":

```

private void close()
{
    send.setEnabled(false); connect.setLabel("Connect");
    users.removeAll(); output.setText("");
    try
    {
        in.close(); out.close(); connection.close();
    }
    catch(IOException ioe)
    {
        System.err.println("Unexpected error closing connection.");
    }
}

```

The last utility method implement is `listUsers`. That method is called by `performAction` and receives a string of user names terminated by '|' as input. It uses a `StringTokenizer` to separate the names and adds each name to the user list:

```

private void listUsers(String names)
{
    users.removeAll();
    StringTokenizer tokens = new StringTokenizer(names, "|");
    while (tokens.hasMoreTokens())
        users.add(tokens.nextToken());
}

```

Of course we need to implement the standard `main` method to set the whole program in motion. As expected, that method simply instantiates a `ChatterBox` object, using command line parameters for the host name and the port to use:

```

public static void main(String a[])
{
    ChatterBox c = new ChatterBox(a[0], Integer.parseInt(a[1]));
}

```

Actually, we are not quite done. We defined an inner class `WindowCloser` as a field and we are using an instance of that class as a `WindowListener` for our frame. We still need to implement that class, though. Therefore, we replace the "empty" implementation `private class WindowCloser extends WindowAdapter { }` used in the above layout of our class by the "real" version as follows:

```
private class WindowCloser extends WindowAdapter
{   public void windowClosing(WindowEvent we)
    {   if (!terminated)
        send("term:" + name);
        System.exit(0);
    }
}
```

The method implements the `windowClosing` method to react to a click on the window's close box. If the thread is currently active, `terminated` is `false` and the class is connected to a `ChatterThread`. The method sends a `term` command to the server to disconnect, then exits the program. Otherwise, the method does not need to send anything to the server and can quit right away.

Now it's time to play around with our `ChatterBox` client/server system to see if everything works as advertised. Try starting two active `ChatterBox` frames, as well as a `Telnet` connection to the `ChatterServer`, all "talking" to each other. For initial testing purposes, all classes should run on one computer, using the host name `localhost` and the (arbitrary) port `9007`, but you should also test your system by moving the server to a system different from the client. See figure 10.4.8 for a screen shot of the `ChatterBox` system in action.

### The `ChatterBox` Client as an Applet

The `ChatterBox` system should work pretty well and since the client is a stand-alone program it can connect to a server regardless of the host running the server. However, it would also be nice to have an applet version of our client. The obvious advantage is that we can enhance any web page with an interactive chat program without asking the user to install any special programs. The clear disadvantage is that because of the applet security restrictions a `ChatterBox` applet can only connect to a `ChatterServer` if both classes are located on the same web server. But then, if we had both an applet and a stand-alone version, we would have the best of both worlds. Fortunately, with a few simple modifications our existing `ChatterBox` class can be made to start from an `Applet` as well as from the command line, and hence the same class can be used for either purpose.

#### ***Example 10.4.2:***

---

Create a `ChatterBoxlet` class that extends `Applet` and starts an instance of a `ChatterBox`. If any modifications are necessary in the existing `ChatterBox` class, make sure that the class will function perfectly as a stand-alone program as well as an applet. Test the entire `ChatterBox` system by placing the server and the applet classes on a web server, start the server on that machine, and connect to it via a `ChatterBoxlet` applet, a `ChatterBox` stand-alone program, as well as via `Telnet`.

The idea of converting a stand-alone program extending `Frame` to an applet in such a way that the same class can serve both purposes is straight-forward:

- create a "starter" class extending `Applet` that contains (at least) one button
- if the user clicks on that button, an instance of the `Frame` class is instantiated

This approach works, but there are usually two problems with easy solutions:

- if a user clicks multiple times on the applet button, you need to make sure that only one instance of the frame is instantiated
- frames generally quit by calling `System.exit`, while applets are not allowed to call that method

The general solution is to pass an instance of the starter applet to the frame. The frame will then be able to tell whether it was started from an applet, in which case it should not call on `System.exit`, or in stand-alone mode. Also, the frame can notify the applet when it is done, so that the applet will know exactly when an instance is active and can avoid activating a second one.

To see how it works, we will proceed in stages. Here is our first, simplistic attempt to start a `ChatterBox` from an applet:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class ChatterBoxlet extends Applet implements ActionListener
{   private Button start = new Button("Start ChatterBox");

    public void init()
    {   setLayout(new FlowLayout());
        add(start);
        start.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {   if (ae.getSource() == start)
        {   ChatterBox chat = new ChatterBox("localhost", 1234); }
    }
}
```

After creating the appropriate HTML code:

```
<HTML>
<APPLET CODE="ChatterBoxlet.class" WIDTH="200" HEIGHT="50">
</APPLET>
</HTML>
```

we can indeed start a `ChatterBox` frame from a web browser or the `appletviewer`. But, as we said, we can not close the frame because its call to `System.exit` generates a security exception. We can also instantiate multiple instances of the class, which we do not want in this case.

Therefore, we will adjust our applet so that it knows whether a `ChatterBox` is active or not by adding a field of type `ChatterBox` as well as an `exit` method to properly close an existing `ChatterBox` if one is active. The `actionPerformed` method needs to change so that it only instantiates a new `ChatterBox` if none is currently active, otherwise it will activate the current one. We also add the capability to read the host and the port from appropriate `PARAM` tags of the HTML document. The new code is in bold and italics:

```
public class ChatterBoxlet extends Applet implements ActionListener
{   private Button      start = new Button("Start ChatterBox");
    private ChatterBox chat = null;
    private String      host  = "localhost";
    private int         port  = 1234;
    public void init()
    {   setLayout(new FlowLayout());
        add(start);
        start.addActionListener(this);
        if (getParameter("host") != null)
            host = getParameter("host");
        if (getParameter("port") != null)
            port = Integer.parseInt(getParameter("port"));
    }
```

```

    }
    public void actionPerformed(ActionEvent ae)
    { if (ae.getSource() == start)
      { if (chat == null)
        { chat = new ChatterBox("localhost", 1234);
          else
          { chat.setState(Frame.NORMAL);
            chat.setVisible(true);
            chat.toFront();
          }
        }
      }
    }
    public void exit()
    { if (chat != null)
      { chat.setVisible(false);
        chat.dispose();
        chat = null;
      }
    }
  }
}

```

This will compile perfectly, but in order for the `ChatterBox` to use the `exit` method of the `ChatterBoxlet` it needs an appropriate reference to that class. Therefore, we again need to modify `ChatterBoxlet`, and we also need to make some small modifications to the original `ChatterBox` class (as usual, the new code is in bold and italics):

- The change to `ChatterBoxlet` is minimal - we just add the reference `this` to the constructor of a new `ChatterBox` object:

```

public class ChatterBoxlet extends Applet implements ActionListener
{ /* Fields as before */
  public void init()
  { /* no changes */ }
  public void actionPerformed(ActionEvent ae)
  { if (ae.getSource() == start)
    { if (chat == null)
      { chat = new ChatterBox("localhost", 1234, this);
        else
        { chat.toFront();
          }
        }
    }
  }
  public void exit()
  { /* no changes */ }
}

```

- In the `ChatterBox` class there are three changes: first, we need to add an extra field of type `ChatterBoxlet`; second, we need a second (overloaded) constructor that matches our new way of instantiation; and finally we need to adjust the inner class `WindowCloser` to make proper use of its new possibilities:

```

public class ChatterBox extends Frame
    implements Runnable, ActionListener
{ /* all fields as before, plus one additional field */
  private ChatterBoxlet applet = null;
  /* the inner class needs to be modified as follows */
  private class WindowCloser extends WindowAdapter
  { public void windowClosing(WindowEvent we)
    { if (!terminated)
      { send("term:" + name);
        if (applet == null)

```

```

        System.exit(0);
    else
        applet.exit();
    }
}

public ChatterBox(String _host, int _port)
{ /* no changes to this constructor */ }
public ChatterBox(String _host, int _port, ChatterBoxlet _applet)
{
    super(ID);
    host = _host;
    port = _port;
    applet = _applet;
    setup();
}
/* remaining code is unchanged */
}

```

The important change is in the inner class `WindowCloser`: if `applet` is `null`, the class must have been instantiated as a stand-alone program, hence the `System.exit` method is appropriate. If `applet` is not `null`, it must have been instantiated by an applet. Therefore the class calls the applet's `exit` method to let the applet close the frame and set its own reference to the frame back to `null`. Now another instance of `ChatterBox` can be started if the `start` button is clicked again.

This will - finally - create a dual-purpose stand-alone/applet combination:

- you can start a `ChatterBox` from the command line
- you can start a `ChatterBox` using the `ChatterBoxlet` applet

Now we finally have a dual-purpose system, and we can use `PARAM` tags in the `HTML` document to pass the proper host and port into the applet which will forward the information to the `ChatterBox`:

```

<HTML>
<APPLET CODE="ChatterBoxlet.class" WIDTH="200" HEIGHT="50">
  <PARAM NAME="host" VALUE="chat.host.edu">
  <PARAM NAME="port" VALUE="4567">
</APPLET>
</HTML>

```

Of course, the standard security restrictions for applets are still in place, so that the only host that the applet will be able to connect to is the same one from which the applet originates. However, our entire client/server `ChatterBox` system is now flexible enough to be installed on any web server without adjusting and recompiling any code.

Figure 10.4.8 shows a screen shot of one `ChatterBox` started from an applet, one started from the command line, and a `Telnet` client, all connected to one `ChatterServer` running on the local computer:

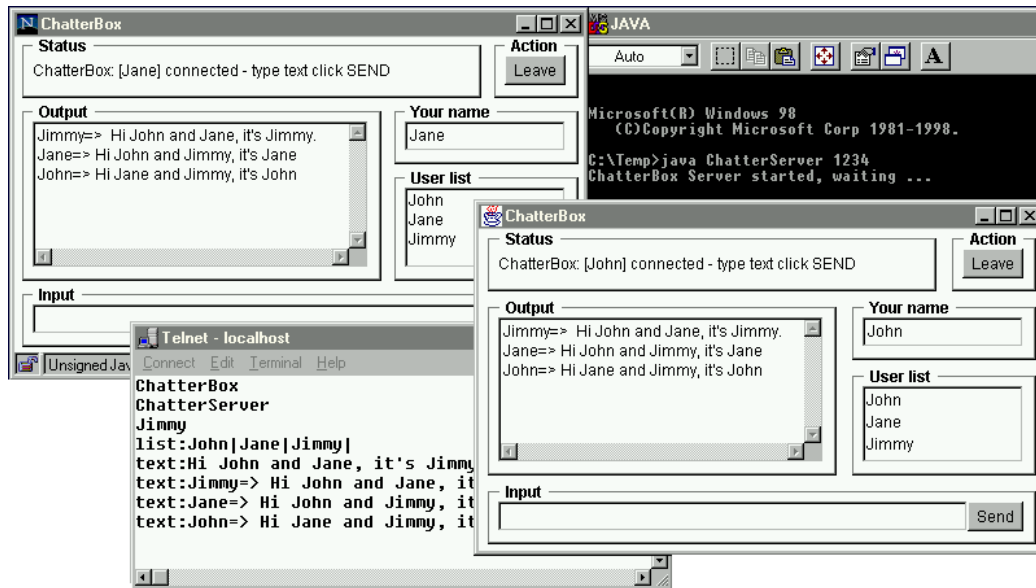


Figure 10.4.8: ChatterBox and Telnet clients connected to a ChatterServer

## Chapter Summary



