CHAPTER 1

Java Socket Programming

- Exploring the world of sockets
- Learning how to program your network
- Java Stream and filter Programming
- Understanding client sockets
- Discovering server sockets

The Internet is built of many related protocols, and more complex protocols are layered on top of system level protocols. A *protocol* is an agreed-upon means of communicating used by two or more systems. Most users think of the Web when they think of the Internet, but the Web is just a protocol built on top of the Hypertext Transfer Protocol (HTTP). HTTP, in turn, is built on top of the Transmission Control Protocol/Internet Protocol (TCP/IP), also known as the sockets protocol.

Most of this book will deal with the Web and its facilitating protocol, HTTP. But before we can discuss HTTP, we must first examine TCP/IP socket programming.

Frequently, the terms *socket* and *TCP/IP programming* are used interchangeably both in the real world and in this chapter. Technically, socket-based programming allows for more protocols than just TCP/IP. With the proliferation of TCP/IP systems in recent years, however, TCP/IP is the only protocol that is commonly used with socket programming.

The World of Sockets

Spiders, bots, and aggregators are programs that browse the Internet. If you are to learn how to create these programs, which is one of the primary purposes of this book, you must first learn how to browse the Internet. By this, I don't mean browsing in the typical sense as a user does; instead, I mean browsing in the way that a computer application, such as Internet Explorer, browses.

Browsers work by requesting documents using the Hypertext Transfer Protocol (HTTP), which is a documented protocol that facilitates nearly all of the communications done by a browser. (Though HTTP is mentioned in connection with sockets in this chapter, it is discussed in more detail in Chapter 2, "Examining the Hypertext Transfer Protocol.") This chapter deals with *sockets*, the protocol that underlies HTTP.

Sockets in Hiding

When sockets are used to connect to TCP/IP networks, they become the foundation of the Internet. But because sockets function beneath the surface, not unlike the foundation of a house, they are often the lowest level of the network that most Internet programmers ever deal with. In fact, many programmers who write Internet applications remain blissfully ignorant of sockets. This is because programmers often deal with higher-level components that act as intermediaries between the programmer and the actual socket commands. Because of this, the programmer remains unaware of the protocol being used and how sockets are used to implement that protocol. In addition, these programmers remain unaware of the layer of the network that exists below sockets—the more hardware-oriented world of routers, switches, and hubs.

Sockets are not concerned with the format of the data; they and the underlying TCP/IP protocol just want to ensure that this data reaches the proper destination. Sockets work much like the postal service in that they are used to dispatch messages to computer systems all over the world. Higher-level protocols, such as HTTP, are used to give some meaning to the data being transferred. If a system is accepting a HTTP-type message, it knows that that message adheres to HTTP, and not some other protocol, such as the Simple Mail Transfer Protocol (SMTP), which is used to send e-mail messages.

The Bot package that comes with this book (see the companion CD) hides this world from you in a manner similar to the way in which networks hide their socket commands behind intermediaries—this package allows the programmer to create advanced bot applications without knowing what a socket is. But this chapter does cover the lower-level aspects of how to actually communicate at the lowest "socket level." These details show you exactly how an HTTP request can be transmitted using sockets, and how the server responds. If, at this time, you are only interested in creating bots and not how Internet protocols are constructed, you can safely skip this chapter.

TCP/IP Networks

When you are using sockets, you are almost always dealing with a TCP/IP network. Sockets are built so that they could abstract the differences between TCP/IP and other low-level network protocols. An example of this is the Internetwork Packet Exchange (IPX) protocol. IPX is the protocol that Novell developed to create the first local area network (LAN). Using sockets, programs could be constructed that could communicate using either TCP/IP or IPX. The socket protocol isolated the program from the differences between IPX and TCP/IP, thus making it so a single program could operate with either protocol.

NOTE Although other protocols can be used with sockets, they have very limited Internet browsing capabilities, and therefore, they will not be discussed in this book.

When it was first introduced, TCP/IP was a radical departure from existing network structures because it did not follow the typical hierarchical pattern that was used before. Unlike other network structures, such as Systems Network Architecture (SNA), TCP/IP makes no distinction between client and server at the machine level, instead, it has a single computer that functions as client, server, or both. Each computer on the network is given a single address, and no address is greater than another. Because of this, a supercomputer running at a government research institute has an IP address, and a personal computer sitting in a teenager's bedroom also has an IP address; there is no difference between these two.

The name for this type of network is a *peer-to-peer network*. All computers on a TCP/IP network are considered peers, and it is very common for machines on this network to function

both as client and server. In a peer-to-peer network, a *client* is the program that sent the first network packet, and a *server* is the program that received the first packet. A *packet* is one network transmission; many packets pass between a client and server in the form of requests and responses.

Network Programming

You will now see how to actually program sockets and deal with socket protocols. Collectively, this is known as *network programming*. Before you learn the socket commands to affect such communications, however, you will first need to examine the protocols. It makes sense to know what you want to transmit before you learn how to transmit it.

You will begin this process by first seeing how a server can determine what protocol is being used. This is done by using common network ports and services.

Common Network Ports and Services

Each computer on a network has many sockets that it makes available to computer programs. These sockets, which are called *ports*, are numbered, and these numbers are very important. (A particularly important one is port 80, the HTTP socket that will be used extensively throughout this book.) Nearly every example in this book will deal with web access, and therefore makes use of port 80. On any one computer, the server programs must specify the numbers of the ports they would like to "listen to" for connections, and the client programs must specify the numbers of the ports they would like to seek connections from.

You may be wondering if these ports can be shared. For instance, if a web user has established a connection to port 80 of a web server, can another user establish a connection to port 80 as well? The answer is yes. Multiple clients can attach to the same server's port. However, only one program at a time can listen on the same server port. Think of these ports as television stations. Many television sets (clients) can be tuned to a broadcast on a particular channel (server), but it is impossible for several stations (servers) to broadcast on the same channel.

Table 1.1 lists common port assignments and their corresponding Request for Comments (RFC) numbers. RFC numbers specify a document that describes the rules of this protocol. We will examine RFCs in much greater detail later in this chapter.

TABLE 1.1: Common Port Assignments and Corresponding RFC Numbers

Port	Common Name	RFC#	Purpose
7	Echo	862	Echoes data back. Used mostly for testing.
9	Discard	863	Discards all data sent to it. Used mostly for testing.

Continued on next page

TABLE 1.1 CONTINUED: Common Port Assignments and Corresponding RFC Numbers

Port	Common Name	RFC#	Purpose
13	Daytime	867	Gets the date and time.
17	Quotd	865	Gets the quote of the day.
19	Chargen	864	Generates characters. Used mostly for testing.
20	ftp-data	959	Transfers files. FTP stands for File Transfer Protocol.
21	ftp	959	Transfers files as well as commands.
23	telnet	854	Logs on to remote systems.
25	SMTP	821	Transfers Internet mail. Stands for Simple Mail Transfer Protocol.
37	Time	868	Determines the system time on computers.
43	whois	954	Determines a user's name on a remote system.
70	gopher	1436	Looks up documents, but has been mostly replaced by HTTP.
79	finger	1288	Determines information about users on other systems.
80	http	1945	Transfer documents. Forms the foundation of the Web.
110	pop3	1939	Accesses message stored on servers. Stands for Post Office Protocol, version 3.
443	https	n/a	Allows HTTP communications to be secure. Stands for Hypertext Transfer Protocol over Secure Sockets Layer (SSL).

What Is an IP Address?

The TCP/IP protocol is actually a combination of two protocols: the Transmission Control Protocol (TCP) and the Internet Protocol (IP). The IP component of TCP/IP is responsible for moving packets of data from node to node, and TCP is responsible for verifying the correct delivery of data from client to server.

An IP address looks like a series of four numbers separated by dots. These addresses are called IP addresses because the actual address is transferred with the IP portion of the protocol. For example, the IP address of my own site is 216.122.248.53. Each of these four numbers is a byte and can, therefore, hold numbers between zero and 255. The entire IP address is a 4-byte, or 32-bit, number. This is the same size as the Java primitive data type of `int`.

Why represent an IP address as four numbers separated by periods? If it's really just an unsigned 32-bit integer, why not just represent IP addresses as their true numeric identities? Actually, you can: the IP address 216.122.248.53 can also be represented by 3631937589. If you point a browser at `http://216.122.248.53` it should take you to the same location as if you pointed it to `http://3631937589`.

If you are not familiar with the byte-order representation of numbers, the transformation from 216.122.248.53 to 3631937589 may seem somewhat confusing. The conversion can easily be accomplished with any scientific calculator or even the calculator that comes with

Windows (in scientific mode). To make the conversion, you must convert each of the byte components of the address 216.122.248.53 into its *hexadecimal* equivalent. You can easily do the conversion by switching the Windows calculator to decimal mode, entering the number, and then switching to hexadecimal mode. When you do this, the results will mirror these:

Decimal	Hexadecimal
216	D8
122	7A
248	F8
53	35

Now that each byte is hexadecimal, you must create one single hexadecimal number that is the composite of all four bytes concatenated together. Just list each byte one right after the other, as shown here:

D8 7A F8 35 or D87AF835

You now have the numeric equivalent of the IP address. The only problem is that this number is in hexadecimal. No problem, your scientific calculator can easily convert hexadecimal back into decimal. When you do so, you will get the number 3,631,937,589. This same number can now be used in the URL: `http://3631937589`.

Why do we need two forms of IP addresses? What does 216.122.248.53 add that 3631937589 does not? Mainly, the former is easier to memorize. Though neither number is terribly appealing to memorize, the designers of the Internet thought that period-separated byte notation (216.122.248.53) was easier to remember than the lengthy numeric notation (3631937589). In reality, though, the end user generally sees neither form. This is because IP addresses are almost always tied to hostnames.

What Is a Hostname?

Hostnames are used because addresses such as 216.122.248.53, or 3631937589, are too hard for the average computer user to remember. For example, my hostname, `www.heat-on.com`, is set to point to `216.122.248.53`. It is much easier for a human to remember `www.heat-on.com` than it is to remember `216.122.248.53`.

A hostname should not be confused with a Uniform Resource Locator (URL). A hostname is just one component of a URL. For example, one page on my site may have the URL of `http://www.jeffheaton.com/java/advanced/`. The hostname is only the `www.jeffheaton.com` portion of that URL. It specifies the server that will transmit the requested files. A hostname only identifies an IP address belonging to a server; a URL specifies some specific file on a server. There are other components to the URL that will be examined in Chapter 2.

The relationship between hostnames and IP addresses is not a one-to-one but a many-to-many relationship. First, let's examine the relationship of many hostnames to one IP address. Very often, people want to host several sites from one server. This server can only have one IP address, but it can allow several hostnames to point to it. This is the case with my own site. In addition to `www.heat-on.com`, I also have `www.jeffheaton.com`. Both of these hostnames are set to provide the exact same IP address. I said that the relationship between hostnames and IP addresses was many-to-many. Is there a case where one single hostname can have multiple IP addresses? Usually this is not the case, but very large volume sites will often have large arrays of servers called *webfarms* or *server farms*. Each of these servers will often have its own individual IP address. Yet the entire server farm is accessible through *one* hostname.

It is very easy to determine the IP address from a hostname. There is a command that most operating systems have called Ping. The Ping command has many uses. It can tell you if the specified site is up or down; it can also tell you the IP address of a host. The format of the Ping command is `PING <hostname | IP>`. You can give Ping either a hostname or an IP address. Below is a Ping that was given the hostname of `heat-on.com`. As `heat-on.com` is pinged, its IP address is returned.

```
C:\>ping heat-on.com

Pinging heat-on.com [216.122.248.53] with 32 bytes of data:

Reply from 216.122.248.53: bytes=32 time=150ms TTL=241
Reply from 216.122.248.53: bytes=32 time=70ms TTL=241
Reply from 216.122.248.53: bytes=32 time=131ms TTL=241
Reply from 216.122.248.53: bytes=32 time=120ms TTL=241
```

This command can also be used to prove that my site with the hostname `jeffheaton.com` really has the same address as my site with the hostname `heat-on.com`. The following Ping command demonstrates this:

```
C:\>ping jeffheaton.com

Pinging jeffheaton.com [216.122.248.53] with 32 bytes of data:

Reply from 216.122.248.53: bytes=32 time=80ms TTL=241
Reply from 216.122.248.53: bytes=32 time=80ms TTL=241
Reply from 216.122.248.53: bytes=32 time=90ms TTL=241
Reply from 216.122.248.53: bytes=32 time=70ms TTL=241
```

The distinction between hostnames and URLs is very important when dealing with Ping. Ping only accepts IP addresses or hostnames. A URL is not an acceptable input to the Ping command. Attempting to ping `http://www.heat-on.com` will not work, as demonstrated here:

```
C:\>ping http://www.heat-on.com/
Bad IP address http://www.heat-on.com/.
```

Ping does have some programming to make it more intelligent. If you were to just ping `http://www.heat-on.com` without the trailing "/" and other path specifiers, the Windows version of Ping will take the hostname from the URL.

WARNING Like nearly every example in this book, the Ping command requires that you be connected to the Internet for this example to work.

How DNS Resolves a Hostname to an IP Address

Socket connections can only be established using an IP address. Because of this, it is necessary to convert a hostname to an IP address. How exactly is a hostname resolved to an IP address? Depending on how your computer is configured, it could be done in several ways, but most systems use domain name service (DNS) to provide this translation. In this section, we will examine this process. First, we will explore how DNS transforms a hostname into an IP address.

DNS and IP Addresses

DNS servers are server machines that return the IP addresses associated with particular hostnames. There is not just one central DNS server, however; resolving hostnames is handled by a huge, diverse array of DNS servers that are set up throughout the world.

When your computer is configured to access the Internet, it must be given the IP addresses of two DNS servers. Usually these are configured by your network administrator or provided by your Internet service provider (ISP). The DNS servers may have hostnames too, but you cannot use these when you are configuring the servers. Your computer must have a DNS server in order to resolve an IP address. If the DNS server you have was presented using a hostname, however, you're in trouble. This is because the computer doesn't have a DNS server to use to look up the IP address of the one DNS server you do have. As you can see, it's really a chicken and egg–type of problem.

But requiring computer users to enter two DNS servers as IP addresses can be cumbersome. If the user enters any piece of this information incorrectly, they will be unable to connect to any sites using a hostname. Because of this, the *Dynamic Host Configuration Protocol (DHCP)* was created.

Using the Dynamic Host Configuration Protocol

Very often, computer systems use DHCP instead of forcing the user to specify most network configuration information (such as IP addresses and DNS servers). The purpose of DHCP is to enable individual computers on an IP network to obtain their initial configurations from a DHCP server or servers, rather than making users perform this configuration themselves. The network administrator can set up all the DNS information on one central machine, the DNS

server. The DHCP server then disseminates this configuration information to all user computers. This provides conformity and alleviates the users from having to enter network configuration information. The DHCP server has no exact information about the individual computers until they request this configuration information. The user computers will request this information when they first connect to the network. The overall purpose of this is to reduce the work necessary to administer a large IP network. The most significant piece of information distributed in this manner is the DNS servers that the user computer should use.

DHCP was created by the Internet Architecture Board (IAB) of the Internet Engineering Task Force (IETF; a volunteer organization that defines protocols for use on the Internet). Because of this, the definition of DHCP is recorded in an Internet RFC, and the IAB is asserting its status as to Internet Standardization.

Many broadband ISPs, such as cable modems and DSL, use DHCP directly from their broadband modem. When the broadband modem is connected to the computer using Ethernet, the DHCP server can be built into the broadband modem so that it can correctly configure the user's computer.

Resolving Addresses Using Java Methods

Earlier, you saw that Ping could be used to determine the IP address of a hostname. In order for this to work, you will need a way for a Java program to programmatically determine the IP address of a site, without having to call the external Ping command. If you know the IP address of the site, you can validate it, or differentiate it from other sites that may be hosted at the same computer. This validation can be completed by using methods from the Java `InetAddress` class.

The most commonly used method in the `InetAddress` class is the `getByName` method. This static method accepts a String parameter that can be an IP address (`216.122.248.53`) or a hostname (`www.heat-on.com`). This is shown in Listing 1.1, which also shows how an IP address can be converted to a hostname or vice versa.

Listing 1.1 Lookup Addresses (Lookup.java)

```
import java.net.*;

/**
 * Example program from Chapter 1
 * Programming Spiders, Bots and Aggregators in Java
 *
 * A simple class used to lookup a hostname using either
 * an IP address or a hostname and to display the IP
 * address and hostname for this address. This class can
 * be used both to display the IP address for a hostname,
 * as well as do a reverse IP lookup and  * give the host
```

```
 * name for an IP address.

 *
 * @author Jeff Heaton
 * @version 1.0
 */
public class Lookup {

  /**
   * The main function.
   *
   * @param args The first argument should be the
   * address to lookup.
   */
  public static void main(String[] args)
  {
    try {
      if ( args.length==0 ) {
        System.out.println(
                          "Call with one parameter that specifies the host " +
                          "to lookup.");
      } else {
        InetAddress address = InetAddress.getByName(args[0]);
        System.out.println(address);
      }
    } catch ( Exception e ) {
      System.out.println("Could not find " + args[0] );
    }
  }
}
```

The actual address resolution in Listing 1.1 occurs during the execution of the following two lines:

```
InetAddress address = InetAddress.getByName(args[0]);
System.out.println(address);
```

First, the input address (held by `arg[0]`) is passed to `getByName` to construct a new `InetAddress` object. This will create a new `InetAddress` object, based on the host specified by `args[0]`. The program should be called by specifying the address to resolve. For example, looking up the IP address for `www.heat-on.com` will result in the following:

```
C:\Lookup>java Lookup www.heat-on.com
www.heat-on.com/216.122.248.53
```

Reverse DNS Lookup

Another very powerful ability that is contained in the `InetAddress` class is *reverse DNS lookup*. If you know only the IP address, as you do in certain network operations, you can pass this IP address to the `getByName` method, and from there, you can retrieve the associated hostname.

For example, if you know the address `216.122.248.53` accessed your web server but you don't know to whom this IP address belongs, you could pass this address to the `InetAddress` object for reverse lookup:

```
C:\Lookup>java Lookup 216.122.248.53
heat-on.com/216.122.248.53
```

With the basics of Internet addressing out of the way, you are now almost ready to learn how to program sockets, but first you must learn a bit of background information about sockets' place in Java's complex I/O handling system. You will first be shown how to use the Java I/O system and how it relates to sockets.

Java I/O Programming

Java has some of the most complex input/output (I/O) capabilities of any programming language. This has two consequences: first, because it is complex, it is quite capable of many amazing things (such as reading ZIP and other complex file formats); second, and somewhat unfortunately, because it is complex, it is somewhat difficult for a programmer to learn, at least initially.

But don't be put off by this initial difficulty because Java has an extensive array of I/O support classes, which are all contained in the `java.io` package. Java's I/O classes are made up of *input streams*, *output streams*, *readers*, *writers*, and *filters*. These are merely categories of object, and there are several examples of each type. These categories will now be examined in detail.

NOTE Because the primary focus of this book is to teach you the Java network communication you will need in order to program spiders, bots, and aggregators, we will examine Java's I/O classes as they relate to network communications. However, much of the information could also easily apply to file-based I/O under Java. If you are already familiar with file programming in Java, much of this material will be review. Conversely, if you are unfamiliar with Java file programming, the techniques learned in this chapter will also directly apply to file programming.

Output Streams

There are many types of output streams provided by Java. All output streams share a common base class, `java.io.OutputStream`. This base class is declared as abstract and, therefore, it cannot be directly instantiated. This class provides several fundamental methods that are needed to write data. This section will show you how to create, use, and close output streams.

Creating Output Streams

The `OutputStream` class provided by Java is abstract, and it is meant only to be overridden to provide `OutputStreams` for such things as socket- and disk-based output. The `OutputStream` provided by Java provides the following methods:

```
public abstract void write(int b)
  throws IOException

public void write(byte[] b)
  throws IOException

public void write(byte[] b, int off, int len)
  throws IOException

public void flush()
  throws IOException

public void close()
  throws IOException
```

NOTE Other Java output streams extend this class to provide functionality. If you would like to create an output stream or filter, you will need to extend this class as well.

We will first see how the abstract `write` method can be used to create an output stream of your own. After that, the next section describes how to use the other methods.

Creating an output stream is relatively easy. You should create an output stream any time you would like to implement a *data consumer*. A data consumer is any class that accepts data and does something with that data. What is done with the data is left up to the implementation of the output stream.

Creating an output stream is easy if you keep in mind what an output stream does—it outputs bytes. This is the only functionality that you must provide to create an output stream. To create the new output stream, you must override the single byte version of the `write` method (`void write(int b)`). This method is used to consume a single byte of data. Once you have overridden this method, you must do with that byte whatever makes sense for the class you are creating (examples include writing the byte to a file or encrypting the byte).

An example of using an output stream to encrypt will be shown in Chapter 3, "Securing Communications with HTTPS." In Chapter 3, we will need to create a class that implements a base64 encoder. *Base64* is a method of encoding text so that it is not easily recognized. We will create a filter that will accept incoming text and output it as encoded base64 data. This encoder works by creating an output stream (actually a filter) capable of outputting base64-encoded text. This class works by providing just the single byte version of `write`.

There are many other examples of output streams provided by Java. When you open a connection to a socket, you can request an output stream to which you can transmit information. Other streams support more traditional I/O. For instance, Java supports a `FileOutputStream` to deal with disk files. Other `OutputStream` descendants are provided for other output streams. Now, you will be shown how to use output streams using some of the other methods of the `OutputStream` class.

Using Output Streams

Output streams exist to allow data to be written to some data consumer; what sort of consumer is unimportant because the output stream objects define methods that allow data to be sent to *any* sort of data consumer.

The `write` method only works with the `byte` data type. Bytes are usually an inconvenient data type to deal with because most data types are larger numbers or strings. Most programmers deal with the higher-level data types that are composed of bytes. Later in this chapter, we will examine filters, which will allow you to write higher-level data types, such as strings, to output streams without the need to manually convert these data types to bytes.

NOTE Even though the `write` methods specify that they accept `int`s, they are actually accepting bytes. Only the lower 8 bytes of the `int` are actually used.

The following example shows you how to write an array of bytes to an output stream. Assume that the variable output is an output stream. You will be shown how to actually obtain an output stream later in this chapter.

```
byte b = new byte[100]; // creates a byte array
output.write( b ); // writes the byte array
```

Now that you have seen how to use output streams, you will be shown how to read them more efficiently. By adding buffering to an output stream, data can be read in much larger, more efficient blocks.

Handling Buffering in Output Streams

It is very inefficient for a programming language to write data out in very small blocks. A considerable overhead occurs every time a `write` method is invoked. If your program uses many `write` method calls, each of which writes only a single byte, much time will be lost just dealing with the overhead of writing each byte independently. To alleviate this problem, Java uses a technique called *buffering*, which is the process of storing bytes for later transmission.

Buffering takes many small `write` method calls and combines them into one large block of data to be written. The size of this eventual block of data is system defined and controlled by Java. Buffering occurs in the background, without the programmer being directly aware of it.

But sometimes the programmer *must* be directly aware of buffering. Sometimes it is necessary to make sure that the data has actually been written and is not just sitting in a buffer. Writing data without regard to buffering is not practical when you are dealing with network streams such as sockets. This is because the server computer is waiting for a complete message from the client before it responds. But how can it ever respond if the client is waiting to send more data? In fact, if you just write the data, you can quickly enter a deadlock situation with each of the components acting as follows:

Client Has just sent some data to the server and is now waiting for a response.

Output Stream (*buffered*) Received the data, but it is now waiting for a bit more information before it transmits the data it has already received over the network.

Server Waiting for client to send the request; will time out soon.

To alleviate this problem, the output stream provides a `flush` method, which allows the programmer to force the output stream to write any data that is stored in the buffer. The `flush` method ensures that data is definitely written. If only a few bytes are written, they may be held in a temporary buffer before being transmitted. These bytes will later be transmitted when there is a certain, system-defined amount. This allows Java to make more efficient use of transfer bandwidth. Programmers should explicitly call the `flush` method when they are working with `OutputStream` objects. This will ensure that any data that has not been transmitted yet will be transmitted.

If you're dumping a certain amount of data to a file object, buffering is less important. For disk-based output, you simply dump the data to the file and then close it. It really does not matter when the data is actually written—you just know that it is all written once you issue the `close` command on the file output stream.

Closing an Output Stream

A `close` method is also provided to every output stream. It is important to call this method when you are done with the `OutputStream` class to ensure that the stream is properly closed and to make sure any file data is flushed out of the stream. If you fail to call the `close` method, Java will discard the memory taken by the actual `OutputStream` object when it goes out of scope, but Java will not actually close the object.

WARNING Not calling the `close` method can often cause your program to leak resources. Resource leaks are operating system objects, such as sockets, that are left open if the `close` method is not called.

If an output stream is an abstract class, where does it come from? How do you instantiate an `OutputStream` class? `OutputStream` objects are never obtained directly by using the *new*

operator. Rather, `OutputStream` objects are usually obtained from other objects. For example, the `Socket` class contains a method called `getOutputStream`. Calling the `getOutputStream` method will return an `OutputStream` object that will be used to write to the socket. Other output streams are obtained by different means.

Input Streams

Like output streams, there are many types of input streams provided by Java, which share a common base class, `java.io.InputStream`. This base class is declared as abstract and, therefore, cannot be directly instantiated. This class provides several fundamental methods that are needed to read data. This section will show how to create, use, and close input streams.

Creating Input Streams

The `InputStream` class provided by Java is abstract, and it is only meant to be overridden to provide `InputStream` classes for such things as socket- and disk-based input. The `InputStream` provided by Java provides the following methods:

```
public abstract int read()
  throws IOException

public int read(byte[] b)
  throws IOException

public int read(byte[] b, int off, int len)
  throws IOException

public long skip(long n)
  throws IOException

public int available()
  throws IOException

public void close()
  throws IOException

public void mark(int readlimit)

public void reset()
  throws IOException

public boolean markSupported()
```

We will first see how the abstract `read` method can be used to create an input stream of your own. After that, the next section describes how to use the other methods.

Creating an input stream is relatively easy. You should create an input stream any time you would like to implement a *data producer*. A data producer is any class that provides data that it got from somewhere. Where this data comes from is left up to the implementation of the output stream.

Creating an input stream is easy if you keep in mind what an input stream does—it reads bytes. This is the only functionality that you must provide to create an input stream. To create the new input stream, you must override the single byte version of the `read` method (`int read()`). This method is used to produce a single byte of data. Once you have overridden this method, you must do with that byte whatever makes sense for the class you are creating (examples include writing the byte to a file or encrypting the byte).

Usually you will be using input streams rather than creating them. The next section describes how to use input streams.

Using Input Streams

There are many examples of overridden input streams provided by Java. For example, when you open a connection to a socket, you can request an input stream from which you can receive information. Java also supports a `FileInputStream` to deal with disk files. Still other `InputStream` descendants are provided for other input streams.

The `InputStream` class uses several methods to transmit data. By using these methods, you can transmit data to a data consumer. The exact nature of this data consumer is unimportant to the input stream; the input stream is only concerned with the function of moving the data. What is done with the data is left up to which type of input stream you're using, such as a socket- or disk-based file. These methods will now be described.

The `read` methods allow you to read data in bytes. Even though the abstract `read` method shown in the previous section returns an `int`, the method is only reading a byte at a time. For performance reasons, whenever reasonably possible, you should try to use the `read` methods that accept an array. This will allow more data to be read from the underlying device at a time.

NOTE Note even though the `read` methods specify that they return `int`s, they are actually returning bytes. Only the lower 8 bytes of the `int` are actually used.

The `skip` method allows a specified number of bytes to be skipped. This is often more efficient than just reading bytes and discarding their values. The `available` method is also provided to show how many bytes are available to be read.

Java also supports two methods called `mark` and `reset`. I do not generally recommend their use because they have two weaknesses that are hard to overcome. Specifically, not all streams

support `mark` and `reset`, and those streams that do support them generally impose range limitations that restrict how far you can "rewind." The idea is that you can call a `mark` at some point as you are reading data from the `InputStream` and then you continue reading. If you ever need to return to the point in the stream when the `mark` method was called, you can call `reset` and return to that position. This would allow your program to reread data it has already seen. In many ways, this is a rewind feature for an input stream.

Closing Input Streams

Just like output streams, input streams must be closed when you are done with them. Input streams do not have the buffering issues that output streams do, however. This is because input streams are just reading data, not saving it. Since the data is already saved, the input stream cannot cause any of it to be lost. For example, reading only half of a file won't in anyway change or damage that file.

Input streams do share the resource-leaking issues of output streams, though. If you do not explicitly close an input stream, you run the risk of the underlying operating system resource not being closed. If this is done enough, your program will run out of streams to allocate.

Filter streams are built on the concept of input and output streams. Filter streams can be layered on top of input and output streams to provide additional functionality. Filters will be discussed in the next section.

Filter Streams, Readers, and Writers

Any I/O operation can be accomplished with the `InputStream` and `OutputStream` classes. These classes are like atoms: you can build anything with them, but they are very basic building blocks. The `InputStream` and `OutputStream` classes only give you access to the raw bytes of the connection. It's up to you to determine whether the underlying meaning of these bytes is a string, an IEEE754 floating point number, Unicode text, or some other binary construct.

Filters are generally used as a sort of attachment to the `InputStream` and `OutputStream` classes to hide the low-level complexity of working solely with bytes. There are two primary types of filters. The first is the *basic filter*, which is used to transform the underlying binary numbers into meaningful data types. Many different basic filters have been created; there are filters to compress, encrypt, and perform various translations on data. Table 1.2 shows a listing of some of the more useful filters available.

TABLE 1.2: Some Java Filters

Read Filter	Write Filter	Purpose
BufferedInputStream	BufferedOutputStream	These filters implement a buffered input and output stream. By setting up such a stream, an application can read/write bytes from a stream without necessarily causing a call to the underlying system for each byte that is read/written. The data is read/written by blocks into a buffer. This often produces more efficient reading and writing. This is a normal filter and can be used in a chain.
DataInputStream	DataOutputStream	A data input/output stream filter allows an application to read/write primitive Java data types from an underlying input/output stream in a machine-independent way.
GZIPInputStream	GZIPOutputStream	This filter implements a stream filter for reading or writing data compressed in the GZIP format.
ZipInputStream	ZipOutputStream	This filter implements input/output filter streams for reading and writing files in the ZIP file format. This class includes support for both compressed and uncompressed entries.
n/a	PrintWriter	This filter prints formatted representations of objects to a text-output stream. This class implements all of the print methods found in `PrintStream`. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

The second type of filter is really a set of filters that work together; the filters that compose this set are called *readers* and *writers*. The remainder of this section will focus on readers and writers. These filters are designed to handle the differences between various methods of text encoding. Readers and writers, for example, can handle text encoded in such formats as ASCII Encoding (UTF-8) and Unicode (UTF-16).

Filters themselves are extended from the `FilterInputStream` and `FilterOutputStream` classes. These two classes inherit from `InputStream` and `OutputStream` classes respectively. Because of this, filters function exactly like the low-level `InputStream` and `OutputStream` classes. Every `FilterInputStream` must implement at least a `read` method. Likewise, every `FilterOutputStream` must implement at least a `write` method. By overriding these methods, the filters may modify data, as it is being read or written. Many filter streams will provide many more methods. But some, for example the `BufferedInputStream` and `BufferedOutputStream`, provide no new methods and merely keep the same interface as `InputStream` and `OutputStream`.

Chaining Filters Together

One very important feature of filters is their ability to chain themselves together. A basic filter can be layered on top of either an input/output stream or another filter. A reader/writer can be layered on top of an input/output stream or another filter but never on another reader/writer. Readers and writers must always be the last filter in a chain.

Filters are layered by passing the underlying filter or stream into the constructor of the new stream. For example, to open a file with a `BufferedInputStream`, the following code should be used:

```
FileInputStream fin = new FileInputStream("myfile.txt");
BufferedInputStream bis = new BufferedInputStream(fin);
```

It is very important that the underlying `InputStream` not be discarded. If the `fin` variable in the preceding code were reassigned or set to null, an error would result when the `BufferedInputStream` was used.

Proxy Issues

One very important aspect of TCP/IP networking is that no two computers can have the same IP address. Proxies and firewalls allow many computers to access the Internet through one single IP address, though. This is often the situation in large corporate environments. The users will access one single computer, called a *proxy server*, rather than directly connecting to the Internet. This access is generally sufficient for most users.

The primary difference between a direct connection and this type of connection is that when a computer is directly connected to the Internet, that computer has one or more IP addresses all to itself. In a proxy situation, any number of computers could be sharing the same outbound proxy IP address. When the computer hooked to the proxy is using client-side sockets, this does not present a problem. The server that is acting as the proxy server can conceivably support any number of outbound connections.

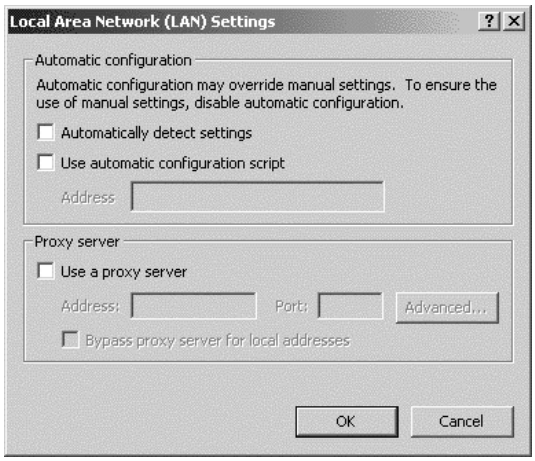
Problems occur when a computer connected through the proxy wants to become a server. If the computer hooked to the proxy network sets itself to become a server on a specific port, then it can *only* accept connections on the internal proxy network. If a computer from the outside attempts to connect back to the computer behind the proxy, it will end up trying to connect to the proxy computer, which will likely refuse the connection.

Most of the programs presented in this book are clients. Because of this, they can be run from behind a proxy server with little trouble. The only catch is that they have to know that they are connected through a proxy. For example, before you can use Microsoft Internet Explorer (IE) from behind a proxy server, you must configure it to know that it is being run

in this configuration. In the case of IE, you can select Tools and then Internet Options to do this. From the resulting menu, select Connections and then choose the LAN Settings button. A screen similar to the one in Figure 1.1 will appear. This screen shows you how to configure IE for the correct proxy settings.

NOTE This book assumes that you have a working Internet connection. You will need the information presented here to allow Java to use your proxy server. Just having the settings in IE does not configure every network service on your computer to use the proxy server. Each application must generally be configured separately.

FIGURE 1.1: Proxy settings in Internet Explorer



Configuring Java to Use a Proxy Server

There are two ways to configure Java to use a proxy server. The proxy configuration can be either set by the Java code itself, or it can be set as parameters to the Java Virtual Machine (JVM) when the application is first started. The proxy settings for Java are contained in system properties and can be specified from the command line or can be set by the program. Table 1.3 shows a list of some of the more common proxy-related system properties. Like any system property, proxy-related properties can be set in two different ways. The first is by specifying them on the command line to the JVM. For example, to execute a program called `UseProxy.class`, you could use the following command:

```
java –Dhttp.ProxyHost=socks.myhost.com -Dhttp.ProxyPort=1080 UseProxy
```

If you would prefer to set the proxy information programmatically from your program, you can use the following section of code to accomplish the same thing. You do not need to use both methods—one will suffice.

```
public class UseProxy
{
  public static void main(String args[])
  {
   System.setProperty("http.proxySet",true);
   System.setProperty("http.proxyHost","socks.myhost.com");
   System.setProperty("http.proxyPort","1080");
    // program continues here
  }
}
```

WARNING If you are connecting to the Internet through a proxy server, you *must* use one of the above methods to let Java know about your proxy settings. If you fail to do this, the programs in this book will not be able to connect to the Internet.

TABLE 1.3: Common Command Line Proxy Settings in Java

System Property	Values	Purpose
FtpProxySet	true/false	Set to true if a proxy is to be used for FTP connections.
FtpProxyHost	hostname	The host address for a proxy server to be used for FTP connections.
FtpProxyPort	port number	The port to be used on the specified hostname to be used for FTP connections.
gopherProxySet	true/false	Set to true if a proxy is to be used for Gopher connections.
gopherProxyHost	hostname	The host address for a proxy server to be used for Gopher connections.
gopherProxyPort	port number	The port to be used on the specified hostname to be used for Gopher connections.
http.proxySet	true/false	Set to true if a proxy is to be used for HTTP connections.
http.proxyHost	hostname	The host address for a proxy server to be used for HTTP connections.
http.proxyPort	port number	The port to be used on the specified hostname to be used for HTTP connections.
https.proxySet	true/false	Set to true if a proxy is to be used for HTTPS connections.
https.proxyHost	hostname	The host address for a proxy server to be used for HTTPS connections.
https.proxyPort	port number	The port to be used on the specified hostname to be used for HTTPS connections.

Socket Programming in Java

Java has greatly simplified socket programming, especially when compared to the requirements and constructs of many other programming languages. Java defines two classes that are of particular importance to socket programming: `Socket` and `ServerSocket`. If the program you are

writing is to play the role of server, it should use `ServerSocket`. If the program is to connect to a server, and thus play the role of client, it should use the `Socket` class.

The `Socket` class, whether server (when done through the child class `ServerSocket`) or client, is only used to initially start the connection. Once the connection is established, input and output streams are used to actually facilitate the communication between the client and server. Once the connection is made, the distinction between client and server is purely arbitrary. Either side may read from or write to the socket.

All socket reading is done through a Java `InputStream` class, and all socket writing is done through a Java `OutputStream` class. These are low-level streams provide only the most rudimentary input methods. All communication with the `InputStream` and the `OutputStream` must be done with bytes—bytes are the only data type recognized by these classes. Because of this, the `InputStream` and `OutputStream` classes are often paired with higher-level Java input classes. Two such classes for `InputStream` are the `DataInputStream` and the `BufferedReader`. The `DataInputStream` allows your program to read binary elements, such as 16- or 32-bit integers from the socket stream. The `BufferedReader` allows you to read lines of text from the socket. For `OutputStream`, the two possible classes are `DataOutputStream` and the `PrintWriter`. The `DataOutputStream` allows your program to write binary elements, such as 16- or 32-bit integers from the socket stream. The `PrintWriter` allows you to write lines of text from the socket.

As mentioned earlier, sockets form the lowest-level protocol that most programmers ever deal with. Layered on top of sockets are a host of other protocols used to implement Internet standards. These socket protocols are documented in RFCs. You will now learn about RFCs and how they document socket protocols.

Socket Protocols and RFCs

Sockets merely define a way to have a two-way communication between programs. These two programs can write any sort of data, be it binary or textual, to/from each other. If there is to be any order to this, though, there must be an established protocol. Any protocol will define how each side should communicate and what is to be accomplished by this communication.

Every Internet protocol is documented in a RFC—RFCs will be quoted as sources of information throughout this book. RFCs are numbered; for example, HTTP is documented in RFC1945. A complete set of RFCs can be found at `http://www.rfc-editor.org`.

RFC numbers are never reused or edited. Once an RFC is published, it will not be modified. The only way to effectively modify an RFC is to publish a new RFC that makes the old RFC obsolete.

NOTE To see which RFC number applies to protocols such as HTTP, SMTP, FTP, and other Internet protocols, refer back to Table 1.1.

For the remainder of the chapter, we will be examining client sockets and server sockets. These will be described in detail through the use of two RFCs. First, you'll look at RFC821, which defines SMTP and shows a client implementation. Second, you will examine RFC1945, which defines HTTP and shows a simple web server implementation.

Client Sockets

Client sockets are used to establish a connection to server sockets, and they are the type of sockets that will be used for the majority of socket examples throughout this book. To demonstrate client sockets, we will look at an example of SMTP. You will be shown SMTP through the use of an example program that sends an e-mail.

The Simple Mail Transfer Protocol

The *Simple Mail Transfer Protocol (SMTP)* forms the foundation of all e-mail delivery by the Internet. As you can see from Table 1.1, SMTP uses port 25 and is documented by RFC821.

When you install an Internet e-mail program, such as Microsoft Outlook Express or Eudora Mail, you must specify a SMTP server to process outgoing mail. This SMTP server is set up to receive mail messages formatted by Eudora or similar programs. When an SMTP server receives an e-mail, it first examines the message to determine who it is for. If the SMTP server controls the mailbox of the receiver, then the message is delivered. If the message is for someone on another SMTP server, then the message is forwarded to that SMTP server.

NOTE For the purposes of this chapter, you do not care whether the SMTP server is going to forward the e-mail or handle the e-mail itself. Your only concern is that you have handed the e-mail off to an SMTP server, and you assume that the server will handle it appropriately. You will not be aware of it if the e-mail needs to be forwarded or processed.

The SMTP protocol that RFC821 defines is nothing more than a series of requests and responses. The SMTP client opens a connection to the server. Once the connection is established, the client can issue any of the commands shown in Table 1.4.

NOTE Table 1.4 does not show a complete set of SMTP commands, just the commands needed for this chapter. For a complete list of commands refer to RFC821.

TABLE 1.4: Selected SMTP Commands

Command	Purpose
HELO [*client name*]	Should be the first command, and should identify the client computer.
MAIL FROM [*user name*]	Should specify who the message is from, and should be a valid e-mail address.
RCPT TO [*user name*]	Should specify the receiver of this message, and should be a valid e-mail address.
DATA	Should be sent just before the body of the e-mail message. To end this command, you must send a period (".") as a single line.

Here, you can see a typical communication session, including the commands discussed in Table 1.4, between an RFC client and the RFC server:

1. The client opens the connection. The server responds with

   ```
   220 heat-on.com ESMTP Sendmail 8.11.0/8.11.0; Mon, 28 May 2001 15:41:26
   -0500 (CDT)
   ```

2. The client sends its first command (the HELO command) to identify itself, followed by the hostname:

   ```
   HELO JeffSComputer
   ```

   Sometimes the hostname is used for security purposes, but generally it is just logged. By convention, the hostname of the client computer should be displayed after the HELO command as seen here.

3. The server responds with

   ```
   250 heat-on.com Hello SC1-178.charter-stl.com [24.217.160.175], pleased to
   meet you
   ```

4. The client sends its second command:

   ```
   MAIL FROM: thesender@senderhost.com
   ```

   It is here that the e-mail sender is specified. Some SMTP severs will verify that the person the e-mail is from is a valid user for this system. This is to prevent certain bulk e-mailers from fraudulently sending large quantities of unwanted e-mail from an unsuspecting SMTP server.

5. The server responds with

   ```
   250 2.1.0 thesenderj@senderhost.com... Sender ok
   ```

6. The client sends its third command:

   ```
   RCPT TO: touser@tohost.com
   ```

This command specifies to whom the e-mail is being sent. The SMTP server looks at this command to determine what to do with the e-mail. If the user specified here is in the same domain handled by the SMTP server, then it sends the message to the correct mailbox. If the user specified here is elsewhere, then it forwards the mail message to the server that handles mail for that user.

7. The server responds with:

```
250 2.1.5 touserj@tohost.com... Recipient ok
```

8. The client now begins to send data:

```
DATA
```

9. The server responds with

```
354 Enter mail, end with "." on a line by itself
```

10. The client sends its data and ends it with a single "." on a line by itself:

```
This is a test message.
.
```

11. Finally, the server responds with

```
250 2.0.0 f4SKfQH59504 Message accepted for delivery
```

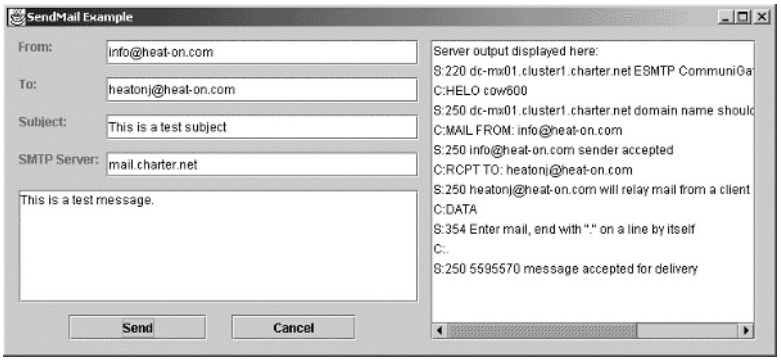
12. The session is complete and the connection is closed.

From this description, it should be obvious that security is at a minimum with SMTP. You can specify essentially any address you wish with the `MAIL FROM` command. This makes it very easy to forge an e-mail. Of course, a savvy Internet user can spot a forgery by comparing the e-mail headers to a known valid e-mail from that person. SMTP servers will always show the path that the e-mail went through in the headers. But to an unsuspecting user, such e-mails can be very confusing and misleading. Bulk e-mailers, who seek to hide their true e-mail addresses, often use such tactics. This is why when you attempt to reply to a bulk e-mail, the message usually bounces.

Using SMTP

Now that we have reviewed SMTP, we will create an example program that implements an SMTP client. This example program will allow the user to send an e-mail using SMTP. This program is shown running in Figure 1.2, and its source code is show in Listing 1.2. The source code is rather extensive; we'll review it in detail following the code listing.

FIGURE 1.2: SMTP example program



Listing 1.2 A Client to Send SMTP Mail (SendMail.java)

```
import java.awt.*;
import javax.swing.*;

/**
 * Example program from Chapter 1
 * Programming Spiders, Bots and Aggregators in Java
 * Copyright 2001 by Jeff Heaton
 *
 * SendMail is an example of client sockets.  This program
 * presents a simple dialog box that prompts the user for
 * information about how to send a mail.
 *
 * @author Jeff Heaton
 * @version 1.0
 */
public class SendMail extends javax.swing.JFrame {

  /**
   * The constructor.  Do all basic setup for this
   * application.
   */
  public SendMail()
  {
    //{{INIT_CONTROLS
    setTitle("SendMail Example");
    getContentPane().setLayout(null);
    setSize(736,312);
    setVisible(false);
    JLabel1.setText("From:");
    getContentPane().add(JLabel1);
    JLabel1.setBounds(12,12,36,12);
    JLabel2.setText("To:");
    getContentPane().add(JLabel2);
```

```
    JLabel2.setBounds(12,48,36,12);
    JLabel3.setText("Subject:");
    getContentPane().add(JLabel3);
    JLabel3.setBounds(12,84,48,12);
    JLabel4.setText("SMTP Server:");
    getContentPane().add(JLabel4);
    JLabel4.setBounds(12,120,84,12);
    getContentPane().add(_from);
    _from.setBounds(96,12,300,24);
    getContentPane().add(_to);
    _to.setBounds(96,48,300,24);
    getContentPane().add(_subject);
    _subject.setBounds(96,84,300,24);
    getContentPane().add(_smtp);
    _smtp.setBounds(96,120,300,24);
    getContentPane().add(_scrollPane2);
    _scrollPane2.setBounds(12,156,384,108);
    _body.setText("Enter your message here.");
    _scrollPane2.getViewport().add(_body);
    _body.setBounds(0,0,381,105);
    Send.setText("Send");
    Send.setActionCommand("Send");
    getContentPane().add(Send);
    Send.setBounds(60,276,132,24);
    Cancel.setText("Cancel");
    Cancel.setActionCommand("Cancel");
    getContentPane().add(Cancel);
    Cancel.setBounds(216,276,120,24);
    getContentPane().add(_scrollPane);
    _scrollPane.setBounds(408,12,312,288);
    getContentPane().add(_output);
    _output.setBounds(408,12,309,285);
    //}}


    //{{INIT_MENUS
    //}}

    //{{REGISTER_LISTENERS
    SymAction lSymAction = new SymAction();
    Send.addActionListener(lSymAction);
    Cancel.addActionListener(lSymAction);
    //}}

    _output.setModel(_model);
    _model.addElement("Server output displayed here:");
    _scrollPane.getViewport().setView(_output);
    _scrollPane2.getViewport().setView(_body);
  }

  /**
   * Moves the app to the correct position
```

```
 * when it is made visible.
 *
 * @param b True to make visible, false to make
 * invisible.
 */
public void setVisible(boolean b)
{
  if ( b )
    setLocation(50, 50);
  super.setVisible(b);
}

/**
 * The main function basically just creates a new object,
 * then shows it.
 *
 * @param args Command line arguments.
 * Not used in this application.
 */
static public void main(String args[])
{
  (new SendMail()).show();
}

/**
 * Created by VisualCafe.  Sets the window size.
 */
public void addNotify()
{
  // Record the size of the window prior to
  // calling parents addNotify.
  Dimension size = getSize();

  super.addNotify();

  if ( frameSizeAdjusted )
    return;
  frameSizeAdjusted = true;

  // Adjust size of frame according to the
  // insets and menu bar
  Insets insets = getInsets();
  javax.swing.JMenuBar menuBar =
    getRootPane().getJMenuBar();
  int menuBarHeight = 0;
  if ( menuBar != null )
    menuBarHeight = menuBar.getPreferredSize().height;
  setSize(insets.left
          + insets.right
          + size.width,
          insets.top
          + insets.bottom
```

```
          + size.height
          + menuBarHeight);
}

// Used by addNotify
boolean frameSizeAdjusted = false;

//{{DECLARE_CONTROLS

/**
 * A label.
 */
javax.swing.JLabel JLabel1 =
  new javax.swing.JLabel();

/**
 * A label.
 */
javax.swing.JLabel JLabel2 =
  new javax.swing.JLabel();

/**
 * A label.
 */
javax.swing.JLabel JLabel3 =
  new javax.swing.JLabel();

/**
 * A label.
 */
javax.swing.JLabel JLabel4 =
  new javax.swing.JLabel();

/**
 * Who this message is from.
 */
javax.swing.JTextField _from =
  new javax.swing.JTextField();

/**
 * Who this message is to.
 */
javax.swing.JTextField _to =
  new javax.swing.JTextField();

/**
 * The subject of this message.
 */
javax.swing.JTextField _subject =
  new javax.swing.JTextField();

/**
```

```
 * The SMTP server to use to send this message.
 */
javax.swing.JTextField _smtp =
  new javax.swing.JTextField();

/**
 * A scroll pane.
 */
javax.swing.JScrollPane _scrollPane2 =
  new javax.swing.JScrollPane();

/**
 * The body of this email message.
 */
javax.swing.JTextArea _body =
  new javax.swing.JTextArea();

/**
 * The send button.
 */
javax.swing.JButton Send =
  new javax.swing.JButton();

/**
 * The cancel button.
 */
javax.swing.JButton Cancel =
  new javax.swing.JButton();

/**
 * A scroll pain.
 */
javax.swing.JScrollPane _scrollPane
= new javax.swing.JScrollPane();

/**
 * The output area.  Server messages
 * are displayed here.
 */
javax.swing.JList _output =
  new javax.swing.JList();
//}}

/**
 * The list of items added to the output
 * list box.
 */
javax.swing.DefaultListModel _model
= new javax.swing.DefaultListModel();

/**
 * Input from the socket.
```

```
 */
java.io.BufferedReader _in;

/**
 * Output to the socket.
 */
java.io.PrintWriter _out;

//{{DECLARE_MENUS
//}}


/**
 * Internal class created by VisualCafe to
 * route the events to the correct functions.
 *
 * @author VisualCafe
 * @version 1.0
 */
class SymAction
  implements java.awt.event.ActionListener {

  /**
   * Route the event to the correction method.
   *
   * @param event The event.
   */
  public void actionPerformed
    (java.awt.event.ActionEvent event)
  {
    Object object = event.getSource();
    if ( object == Send )
      Send_actionPerformed(event);
    else if ( object == Cancel )
      Cancel_actionPerformed(event);
  }
}

/**
 * Called to actually send a string of text to the
 * socket.  This method makes note of the text sent
 * and the response in the JList output box.  Pass a
 * null value to simply wait for a response.
 *
 * @param s A string to be sent to the socket.
 * null to just wait for a response.
 * @exception java.io.IOException
 */
protected void send(String s) throws java.io.IOException
{
  // Send the SMTP command
  if ( s!=null ) {
```

```
    _model.addElement("C:"+s);
    _out.println(s);
    _out.flush();
  }
  // Wait for the response
  String line = _in.readLine();
  if ( line!=null ) {
    _model.addElement("S:"+line);
  }
}

/**
 * Called when the send button is clicked.  Actually
 * sends the mail message.
 *
 * @param event The event.
 */
void Send_actionPerformed(java.awt.event.ActionEvent event)
{
  try {

    java.net.Socket s
    = new java.net.Socket( _smtp.getText(),25 );
    _out = new java.io.PrintWriter(s.getOutputStream());
    _in = new java.io.BufferedReader(
      new java.io.InputStreamReader(s.getInputStream()));

    send(null);
    send("HELO " +
         java.net.InetAddress.getLocalHost().getHostName() );
    send("MAIL FROM: " + _from.getText() );
    send("RCPT TO: " + _to.getText() );
    send("DATA");
    _out.println("Subject:" + _subject.getText());
    _out.println( _body.getText() );
    send(".");
    s.close();

  } catch ( Exception e ) {
    _model.addElement("Error: " + e );
  }

}

/**
 * Called when cancel is clicked.  End the application.
 *
 * @param event The event.
 */
void Cancel_actionPerformed(java.awt.event.ActionEvent event)
{
```

```
        System.exit(0);
      }
    }
```

Using the SMTP Program

To use the program in Listing 1.2, you must know the address of an SMTP server—usually provided by your ISP. If you are unsure of your SMTP server, you should contact your ISP's customer service. In order for outbound e-mail messages to be sent, your e-mail program must have this address. Once it does, you can enter who is sending the e-mail (if you are sending it, you would type your e-mail address in) and who will be on the receiving end. This is usually entered under the Reply To field of your e-mail program. Both of these addresses *must* be valid. If they are invalid, the e-mail may not be sent. After you have entered these addresses, you should continue by entering the subject, writing the actual message, and then clicking send.

NOTE For more information on how to compile examples in this book, see Appendix E "How to Compile Examples Under Windows."

As stated earlier, to send an e-mail with this program, you must enter who is sending the message. You may be thinking that you could enter any e-mail address you want here, right? Yes, this is true; as long as the SMTP server allows it, this program will allow you to impersonate anyone you enter into the To address field. However, as previously stated, a savvy Internet user can tell whether the e-mail address is fake.

After the mention of possible misrepresentation of identity on the sender's end, you may now be asking yourself, "Is this program dangerous?" This program is no more dangerous than any e-mail client (such as Microsoft Outlook Express or Eudora) that also requires you to tell it who you are. In general, all e-mail programs must request both your identity and that of the SMTP server.

Examining the SMTP Server

You will now be shown how this program works. We will begin by looking at how a client socket is created. When the client socket is first instantiated, you must specify two parameters. First, you must specify the host to connect to; second, you must specify the port number (e.g., 80) you would like to connect on. These two items are generally passed into the constructor. The following line of code (from Listing 1.2) accomplishes this:

```
java.net.Socket s =new java.net.Socket( _smtp.getText(),25 );
```

This line of code creates a new socket, named `s`. The first parameter to the constructor, `_smtp.getText()`, specifies the address to connect to. Here it is being read directly from a text field. The second parameter specifies the port to connect to. (The port for SMTP is 25.) Table 1.1

shows a listing of the ports associated with most Internet services. The hostname is retrieved from the `_smtp` class level variable, which is the `JTextField` control that the SMTP hostname is entered into.

If any errors occur while you are making the connection to the specified host, the `Socket` constructor will throw an `IOException`. Once this connection is made, input and output streams are obtained from the `Socket.getInputStream` and `Socket.getOutputStream` methods. This is done with the following lines of code from Listing 1.2:

```
_out = new java.io.PrintWriter(s.getOutputStream());
_in = new java.io.BufferedReader(new
  java.io.InputStreamReader(s.getInputStream()));
```

These low-level stream types are only capable of reading binary data. Because this data is needed in text format, filters are used to wrap the lower-level input and output streams obtained from the socket.

In the code above, the output stream has been wrapped in a `PrintWriter` object. This is because `PrintWriter` allows the program to output text to the socket in a similar manner to the way an application would write data to the `System.out` object—by using the `print` and `println` methods. The application presented here uses the `println` method to send commands to the SMTP server. As you can see in the code, the `InputStream` object has also been wrapped; in this case, it has been wrapped in a `BufferedReader`. Before this could happen, however, this object must first have been wrapped in an `InputStreamReader` object as shown here:

```
_in = new java.io.BufferedReader(new
  java.io.InputStreamReader(s.getInputStream()));
```

This is done because the `BufferedReader` object provides reads that are made up of lines of text instead of individual bytes. This way, the program can read text up to a carriage return without having to parse the individual characters. This is done with the `readLine` method.

You will now be shown how each command is sent to the SMTP server. Each of these commands that is sent results in a response being issued from the SMTP server. For the protocol to work correctly, each response must be read by the SMTP client program. These responses start with a number and then they give a textual description of what the result was. A full-featured SMTP client should examine these codes and ensure that no error has occurred.

For the purposes of the SendMail example, we will simple ignore these responses because most are informational and not needed. Instead, for our purposes, the response will be read in and displayed to the *`_output`* list box. Commands that have been sent to the server are displayed in this list with a `C:` prefix to indicate that they are from the client. Responses returned from the SMTP server will be displayed with the `S:` prefix.

To accomplish this, the example program will use the `send` method. The `send` method accepts a single String parameter to indicate the SMTP command to be issued. Once this

command is sent, the `send` method awaits a response from the SMTP host. The portion of Listing 1.2 that contains the `send` method is displayed here:

```
protected void send(String s) throws java.io.IOException
{
  // Send the SMTP command
  if(s!=null)
  {
    _model.addElement("C:"+s);
    _out.println(s);
    _out.flush();
  }
  // Wait for the response
  String line = _in.readLine();
  if(line!=null)
  {
    _model.addElement("S:"+line);
  }
}
```

As you can see, the `send` method does not handle the exceptions that might occur from its commands. Instead, they are thrown to the `calling` method as indicated by the `throws` clause of the function declaration. The variable *`s`* is checked to see if it is null. If *`s`* is null, then no command is to be sent and only a response is sought. If *`s`* is *not* null, then the value of *`s`* is logged and then sent to the socket. After this happens, the `flush` command is given to the socket to ensure that the command was actually sent and not just buffered. Once the command is sent, the `readLine` method is called to await the response from the server. If a response is sent, then it is logged.

Once the socket is created and the input and output objects are created, the SMTP session can begin. The following commands manage the entire SMTP session:

```
send(null);
send("HELO " +
  java.net.InetAddress.getLocalHost().getHostName() );
send("MAIL FROM: " + _from.getText() );
send("RCPT TO: " + _to.getText() );
send("DATA");
_out.println("Subject:" + _subject.getText());
_out.println( _body.getText() );
send(".");
s.close();
```

TIP Refer to Table 1.4 in the preceding section to review the details of what each of the SMTP commands actually means.

The rest of the SendMail program (as seen in Listing 1.2) is a typical Swing application. The graphical user interface (GUI) layout for this application was created using VisualCafé. The VisualCafé comments have been left in to allow the form's GUI layout to be edited by VisualCafé if you are using it. If you are using an environment other than VisualCafé, you may safely delete the VisualCafé comments (lines starting in //). The VisualCafé code only consists of comments and does not need to be deleted to run on other platforms.

Server Sockets

Server sockets form the side of the TCP/IP connection to which client sockets connect. Once the connection is established, there is little distinction between the server sockets and client sockets. Both use exactly the same commands to send and retrieve data. Server sockets are represented by the `ServerSocket` class, which is a specialized version of the `Socket` class. The `Socket` class is the same class that was discussed in the earlier section about client sockets.

The Hypertext Transfer Protocol

Unlike SMTP, which is used to send e-mail messages, HTTP forms the foundation of all web browsing on the Internet. HTTP differs from SMTP in one very important way: SMTP is made up of a series of single-line packets (or communications) between the client and server, but the typical HTTP request has only two packets—the request and the response. In HTTP, the client sends a series of lines that specify what the client is requesting, and the server then responds with the response as one single packet. Listed below, you can see a typical HTTP client request for the page `http://www.heat-on.com`,

```
GET / HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-
excel, application/msword, application/vnd.ms-powerpoint, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: WWW.HEAT-ON.COM
```

which is followed by the corresponding server response:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Thu, 02 Nov 2000 02:30:16 GMT
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGGGGQRZC=KCGDKDABODIEPLJPHAMBMOFB; path=/
Cache-control: private

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">

<HTML>
```

```
<HEAD>
   <TITLE>Jeff Heaton</TITLE>
…. The rest of the HTML document …..
```

The request and response packets shown here have similar formats. Each one has two areas: the header and the body. The first blank line is the border between the header area and the body area. Usually requests will not have a body, so they end with the first blank line. The body portion of the response is generally the only portion that is actually seen by the user. The headers control information that the client and server send to each other. The body contains the actual HTML code that will be displayed, and it begins immediately after the first blank line.

NOTE The meanings of the headers in HTTP are important. We will discuss them in greater detail in Chapter 2. The section in Appendix B, "HTTP Headers," lists most of the HTTP headers that will be used throughout this book.

The first line of the request is the most important because it specifies the document that is being requested. In the previous example, the server assumes that the browser has been asked to retrieve the URL `http://www.heat-on.com`. The first line `GET / HTTP/1.0` says three important things about this request: what type of request it is, what document is being requested, and what version of HTTP is needed. This line will usually be either `GET`, `POST`, or `HEAD`; in this case, this is a `GET` request. `GET` simply requests a document and sends a little information to the server. This request is mostly used when you click any link you may find on a web page. The `POST` request is used when you submit a form, but this rule does not always apply because JavaScript can alter this behavior (JavaScript can allow `POST`s to be linked to nearly any user event, such as clicking a hyperlink). The `/` indicates the document that being requested. Specifying `/` means that the root document is being requested, not a specific document. Finally `HTTP/1.0` just specifies the HTTP version that is needed.

NOTE For more information on the `HEAD` request, refer to Chapter 2.

After the `GET` or `POST` request is received by the web server, a response is sent back. A sample response is shown in the second part of the previous example. There are two parts to this response. The first is the mention of HTTP headers, with the first line of the HTTP headers specifying the status of the request. The second part of the response is the body, the HTML returned from the server to the browser. The status is shown here:

```
HTTP/1.1 200 OK
```

This first line of the HTTP headers starts with a numeric error code; some of the common ones are listed here. (The section in Appendix B, "HTTP Status Codes," lists the standard meanings of each of these responses.)

100-199 Is an informational message and is not generally used.

200-299 Means a successful request.

300-399 Indicates that the requested resource has been moved. These are often used for redirection.

400-499 Indicates client errors.

500-599 Indicates server errors.

The remaining lines of the header, repeated here, comprise the actual message.

```
Server: Microsoft-IIS/4.0
Date: Thu, 02 Nov 2000 02:30:16 GMT
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGGGGQRZC=KCGDKDABODIEPLJPHAMBMOFB; path=/
Cache-control: private

... message continues here ...
```

Using HTTP

Listing 1.3 shows the example of server sockets for this chapter. In this listing, you are introduced to a very simple web server that would not be practical for any use because it only displays one page. This example does demonstrate the use of a server socket, however. It also shows a simple use of HTTP; more complex uses of HTTP will be discussed in Chapter 2.

Listing 1.3 A Simple Web Server (WebServer.java)

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

/**
 * Example program from Chapter 1
 * Programming Spiders, Bots and Aggregators in Java
 * Copyright 2001 by Jeff Heaton
 *
 * WebServer is a very simple web-server.  Any request
 * is responded with a very simple web-page.
 *
 * @author Jeff Heaton
 * @version 1.0
 */
public class WebServer {

  /**
   * WebServer constructor.
   */
  protected void start()
  {
    ServerSocket s;
```

```
    System.out.println("Webserver starting up on port 80");
    System.out.println("(press ctrl-c to exit)");
    try {
      // create the main server socket
      s = new ServerSocket(80);
    } catch ( Exception e ) {
      System.out.println("Error: " + e );
      return;
    }

    System.out.println("Waiting for connection");
    for ( ;; ) {
      try {
        // wait for a connection
        Socket remote = s.accept();
        // remote is now the connected socket
        System.out.println("Connection, sending data.");
        BufferedReader in = new BufferedReader(
          new InputStreamReader(remote.getInputStream()) );
        PrintWriter out
        = new PrintWriter(remote.getOutputStream());

        // read the data sent.  We basically ignore it,
        // stop reading once a blank line is hit.  This
        // blank line signals the end of the client HTTP
        // headers.
        String str=".";
        while ( !str.equals("") )
          str = in.readLine();

        // Send the response
        // Send the headers
        out.println("HTTP/1.0 200 OK");
        out.println("Content-Type: text/html");
        out.println("Server: Bot");
        // this blank line signals the end of the headers
        out.println("");
        // Send the HTML page
        out.println(
         "<H1>Welcome to the Ultra Mini-WebServer</H2>");
        out.flush();
        remote.close();
      } catch ( Exception e ) {
        System.out.println("Error: " + e );
      }
    }
  }

  /**
   * Start the application.
   *
```

```
     * @param args Command line parameters are not used.
     */
    public static void main(String args[])
    {
      WebServer ws = new WebServer();
      ws.start();
    }
  }
```

Listing 1.3 implements this very simple web server that is shown in Figure 1.3 below. This listing demonstrates how server sockets are used to listen for requests and then fulfill them.

TIP To use the program in Listing 1.3, you must execute it on a computer that does not already have a web server running. If there is already a web server running, an error will be displayed and the example program will terminate. For more information on how to compile and execute a program, see Appendix E.

Because a full-featured web server would be beyond the scope of this book, the program in Listing 1.3 will ignore any requests and simply respond with the page shown in Figure 1.3. Because this program is a web server, to see its output, you must access it with a browser. To access the server from the same machine that the server is running on, select `http://127.0.0.1` as the address that the browser is to look at. The IP address 127.0.0.1 always specifies the local machine. Alternatively, you can view this page from another computer by pointing its browser at the IP address of the computer running the web server program.

FIGURE 1.3: The mini web server

Examining the Mini Web Server

Server sockets use the `ServerSocket` object rather than the `Socket` object that client sockets use. There are several constructors available with the `ServerSocket` object. The simplest

constructor accepts only the port number on which the program should be listening. *Listening* refers to the mode that a server is in while it waits for clients to connect. The following lines of code are used in Listing 1.3 to create a new `ServerSocket` object and reserve port 80 as the port number on which the web server should listen for connections:

```
try
{
   // create the main server socket
   s = new ServerSocket(80);
}
catch(Exception e)
{
  System.out.println("Error: " + e );
  return;
}
```

The `try` block is necessary because any number of errors could occur when the program attempts to register port 80. The most common error that would result is that there is already a server listening to port 80 on this machine.

WARNING This program will not work on a machine that already has a web server, or some other program, listening on port 80.

Once the program has port 80 registered, it can begin listening for connections. The following line of code is used to wait for a connection:

```
Socket remote = s.accept();
```

The `Socket` object that is returned by `accept` is exactly the same class that is used for client sockets. Once the connection is established, the difference between client and server sockets fade. The primary difference between client and server sockets is the way in which they connect. A client sever connects to something. A server socket waits for something to connect to it.

The `accept` method is a *blocking* call, which means the current thread will wait for a connection. This can present problems for your program if there are other tasks it would like to accomplish while it is waiting for connections. Because of this, it is very common to see the `accept` method call placed in a `worker` thread. This allows the main thread to carry on other tasks, while the `worker` thread waits for connections to arrive.

Once a connection is made, the `accept` method will return a `socket` object for the new socket. After this point, reading and writing is the same between client and server sockets. Many client server programs would create a new thread to handle this new connection.

Now that a connection has been made, a new thread could be created to handle it. This new `worker` thread would process all the requests from this client in the background, which allows the `ServerSocket` object to wait for and service more connections. However, the example

program in Listing 1.3 does not require such programming. As soon as the socket is accepted, input and output objects are created; this same process was used with the SMTP client. The following lines from Listing 1.3 show the process of preparing the newly accepted socket for input and output:

```
// remote is now the connected socket
System.out.println("Connection, sending data.");
BufferedReader in = new BufferedReader(
  new InputStreamReader(remote.getInputStream()) );
PrintWriter out = new PrintWriter(remote.getOutputStream());
```

Now that the program has input and output objects, it can process the HTTP request. It first reads the HTTP request lines. A full-featured server would parse each line and determine the exact nature of this request, however, our ultra-simple web server just reads in the request lines and ignores them, as shown here:

```
// read the data sent. We basically ignore it,
// stop reading once a blank line is hit. This
// blank line signals the end of the
// client HTTP headers.
String str=".";
while(!str.equals(""))
  str = in.readLine();
```

These lines cause the server to read in lines of text from the newly connected socket. Once a blank line (which indicates the end of the HTTP header) is reached, the loop stops, and the server stops reading. Now that the HTTP header has been retrieved, the server sends an HTTP response. The following lines of code accomplish this:

```
// Send the response
// Send the headers
out.println("HTTP/1.0 200 OK");
out.println("Content-Type: text/html");
out.println("Server: Bot");
// this blank line signals the end of the headers
out.println("");// Send the HTML page
out.println(
  "<H1>Welcome to the Ultra Mini-WebServer</H2>");
```

Status code 200, as shown on line 3 of the preceding code, is used to show that the page was properly transferred, and that the required HTTP headers were sent. (Refer to Chapter 2 for more information about HTTP headers.) Following the HTTP headers, the actual HTML page is transferred. Once the page is transferred, the following lines of code from Listing 1.3 are executed to clean up:

```
out.flush();
remote.close();
```

The `flush` method is necessary to ensure that all data is transferred, and the `close` method is necessary to close the socket. Although Java will discard the `Socket` object, it will not generally close the socket on most platforms. Because of this, you must close the socket or else you might eventually get an error indicating that there are no more file handles. This becomes very important for a program that opens up many connections, including one to a spider.

Summary

Socket programming is an area of Java that many programmers are unaware of. Sockets are used to implement bidirectional communication channels between programs that are typically running on different computers. All support for sockets is directly built into the JDK and does not require the use of third-party class libraries. Sockets are divided into two categories: client sockets and server sockets. Client sockets initiate the communication between programs; server sockets wait for clients to connect. Once the user has connected, both types function in the same manner and both can send and receive data packets.

Keep in mind that server sockets must specify a port on which to listen for connections. Each computer on the Internet has numeric ports assigned to it, and no two services on the same machine may share a port number. However, two clients may connect to the same port on the same machine.

The protocol by which a client and server communicate must also be well defined. The client and server programs are rarely made by the same vendor, so open standards are very important. Most Internet standards are documented in Request for Comments (RFCs). RFCs are never altered or removed once they have been published; instead, to change information in a RFC, a new one is published that is said to make the old one obsolete.

Now that you know the basics of socket communication, you can begin to explore HTTP. Chapter 2 focuses exclusively on implementing the routines necessary to communicate with a web server. (Web servers are also known as HTTP servers.) The `GET` and `POST` methods of the HTTP protocol will be examined in detail.