Chapter 04
Networking Basics

Contents:

Introduction[Ref. 2]

Sun Microsystems, the developer of Java having their motto as, 'The Network is Computer'. So they made the Java programming language, more appropriate for writing networked programs than, say, C++ or FORTRAN. What makes Java a good language for networking are the classes defined in the java.net package.

These networking classes encapsulate the socket paradigm pioneered in the Berkeley Software Distribution (BSD) from the University of California at Berkeley. No discussion of Internet networking libraries would be complete without a brief recounting of the history of UNIX and BSD sockets.

Networking Basics

Ken Thompson and Dennis Ritchie developed UNIX in concert with the C language at Bell Telephone Laboratories, in 1969. For many years, the development of UNIX remained in Bell Labs and in a few universities and research facilities that had the DEC-PDP machines it was designed to be run on. In 1978, Bill Joy was leading a project at Cal Berkeley to add many new features to UNIX, such as virtual memory and full-screen display capabilities. By early 1984, just as Bill was leaving to found Sun Microsystems, he shipped
4.2BSD, commonly known as Berkeley UNIX. 4.2 BSD came with a fast file system, reliable signals, inter-process communication, and, most important, networking. The networking support first found in 4.2 eventually became the de facto standard for the Internet. Berkeley's implementation of TCP/IP remains the primary standard for communications within the Internet. The socket paradigm for inter-process and network communication has also been widely adopted outside of Berkeley. Even Windows and the Macintosh started talking Berkeley sockets in the late 80s.

The OSI Reference Model

A formal OSI - Open System Interconnection - model has 7 layers but this one shows the essential layer definitions. Each layer has its own standardized protocols and applications programming interface (API), which refers to the functions, and their arguments and return values, called by the next higher layer. Internally, the layers can be implemented in different ways as long as externally they obey the standard API.

For example, the Network Layer does not know if the Physical Layer is Ethernet or a wireless system because the device drivers respond to the function calls the same way. The Internet refers primarily to the Network Layer that implements the Internet Protocol (IP) and the Transport Layer that implements the Transmission Control Protocol (TCP). In fact, we often here people refer to the "TCP/IP" network rather than calling it the Internet.

The application layer also includes various protocols, such as FTP (File Transport Protocol) and HTTP (Hypertext Transfer Protocol) for the Web, that rely on the TCP/IP layers. Most users never look below the application layer. Most application programmers never work below the TCP/IP layers.

Socket Overview

A network socket is a lot like an electrical socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands the standard protocol can plug into the socket and communicate. With electrical sockets, it doesn't matter if you plug in a lamp or a toaster; as long as they are expecting 50Hz, 115-volt electricity, the devices will work. Think how our electric bill is created. There is a meter some where between our house and the rest of the network. For each kilowatt of power that goes through that meter, we are billed. The bill comes to our address. So even though the electricity flows freely around the power grid, all of the sockets in our house have a particular address. The same idea applies to network sockets, except we talk about TCP/IP packets and IP addresses rather than electrons and street addresses. Internet Protocol (IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. Transmission Control Protocol (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and re-transmitting them as necessary to reliably transmit our data. A third protocol, User Datagram Protocol (UDP), sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Client/Server

A server is anything that has some resource that can be shared. There are compute servers, which provide computing power; print servers, which manage a collection of printers; disk servers, which provide networked disk space; and web servers, which store web pages. A client is simply any other entity that wants to gain access to a particular server. The interaction between client and server is just like the interaction between a lamp and an electrical socket. The power grid of the house is the server, and the lamp is a power client. The server is a permanently available resource, while the client is free to unplug after it is has been served.

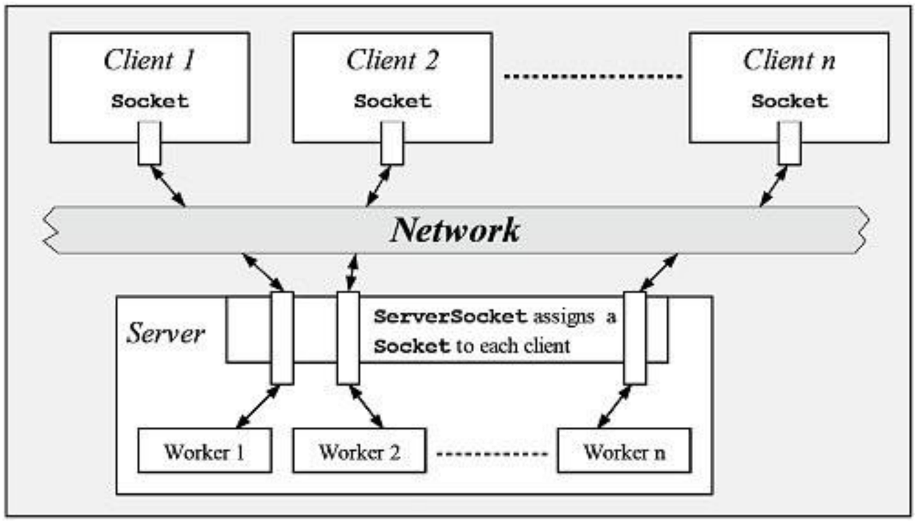

Fig. Client-Server Communication

In Berkeley sockets, the notion of a socket allows a single computer to serve many different clients at once, as well as serving many different types of information. This feat is managed by the introduction of a port, which is a numbered socket on a particular machine. A server process is said to listen to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Reserved Sockets

Once connected, a higher-level protocol ensues, which is dependent on which port we are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to us if we have spent any time surfing the Internet. Port number 21 is for FTP, 23 is for Telnet, 25 is for e- mail, 79 is for finger, 80 is for HTTP, 119 is for net-news and the list goes on. It is up to each protocol to determine how a client should interact with the port.

Table- Well-known port assignments

Protocol	Port	Protocol	Purpose
daytime	13	TCP/UDP	Provides an ASCII representation of the current time on the server.
FTP data	20	TCP	FTP uses two well-known ports. This port is used to transfer files.
FTP	21	TCP	This port is used to send FTP commands like put and get.
SSH	22	TCP	Used for encrypted, remote logins.
telnet	23	TCP	Used for interactive, remote command-line sessions.
smtp	25	TCP	The Simple Mail Transfer Protocol is used to send email between machines.
time	37	TCP/UDP	A time server returns the number of seconds that have elapsed on the server since midnight, January 1, 1900, as a four-byte, signed, big-endian integer.
whois	43	TCP	A simple directory service for Internet network administrators.
finger	79	TCP	A service that returns information about a user or users on the local system.
HTTP	80	TCP	The underlying protocol of the World Wide Web.
POP3	110	TCP	Post Office Protocol Version 3 is a protocol for the transfer of accumulated email from the host to sporadically connected clients.

NNTP	119	TCP	Usenet news transfer; more formally known as the "Network News Transfer Protocol".
IMAP	143	TCP	Internet Message Access Protocol is a protocol for accessing mailboxes stored on a server.
RMI Registry	1099	TCP	The registry service for Java remote objects.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is quite a simple protocol for a basic page-browsing web server. When a client requests a file from an HTTP server, an action known as a hit, it simply prints the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code number to tell the client whether the request can be fulfilled and why. Here's an example of a client requesting a single file, /index.html, and the server replying that it has successfully found the file and is sending it to the client:

Server	**Client**
Listens to port 80	Connects to port 80
Accepts the connection	Writes GET /index.html HTTP/1.0\n\n.
Reads up until the second end-of-line (\n)	
Sees that GET is a known command and that HTTP/1.0 is a valid protocol version.	
Reads a local file called /index.html	
Writes HTTP/1.0 200 OK\n\n.	"200" means here comes the file
Copies the contents of the file into the socket.	Reads the contents of the file and displays it.
Hangs up.	Hangs up.

Proxy Servers

A proxy server speaks the client side of a protocol to another server. This is often required when clients have certain restrictions on which servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in turn communicate for the client. A proxy server has the additional ability to filter certain requests or cache the results of those requests for future use. A caching proxy HTTP server can help reduce the bandwidth demands on a local network's connection to the Internet. When a popular web site is being hit by hundreds of users, a proxy server can get the contents of the web server's popular pages once, saving expensive internet work transfers while providing faster access to those pages to the clients.

Internet Addressing

Every computer on the Internet has an address. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128- bit value to represent an address. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, IPv6 is downwardly compatible with IPv4. Currently, IPv4 is by far the most widely used scheme, but this situation is likely to change over time.

Because of the emerging importance of IPv6, Java 2, version 1.4 has begun to add support for it. However, at the time of this writing, IPv6 is not supported by all environments. Furthermore, for the next few years, IPv4 will continue to be the dominant form of addressing. As mentioned, IPv4 is, loosely, a subset of IPv6, and the material contained in this chapter is largely applicable to both forms of addressing.

There are 32 bits in an IPv4 IP address, and we often refer to them as a sequence of four numbers between 0 and 255 separated by dots (.). This makes them easier to remember; because they are not randomly assigned they are hierarchically assigned. The first few bits define which class of network, lettered A, B, C, D, or E, the address represents. Most Internet users are on a class C network, since there are over two million networks in class C. The first byte of a class C network is between 192 and 224, with the last byte actually identifying an individual computer among the 256 allowed on a single class C network. This scheme allows for half a billion devices to live on class C networks.

Domain Naming Service (DNS)

The Internet wouldn't be a very friendly place to navigate if everyone had to refer to their addresses as numbers. For example, it is difficult to imagine seeing http://192.9.9.1/ at the bottom of an advertisement. Thankfully, a clearing house exists for a parallel hierarchy of names to go with all these numbers. It is called the Domain Naming Service (DNS). Just as the four numbers of an IP address describe a network hierarchy from left to right, the name of an Internet address, called its domain name, describes a machine's location in a name space, from right to left. For example, www.google.com is in the COM domain (reserved for commercial sites), it is called Google (after the company name), and www is the name of the specific computer that is Google's web server. www corresponds to the rightmost number in the equivalent IP address.

Java and the Net

Now that the stage has been set, let's take a look at how Java relates to all of these network concepts. Java supports TCP/IP both by extending the already established stream I/O interface and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model.

The Networking Classes and Interfaces

The classes contained in the java.net package are listed here:

Authenticator	InetSocketAddress	SocketImpl
ContentHandler	JarURLConnection	SocketPermission
DatagramPacket	MulticastSocket	URI
DatagramSocket	NetPermission	URL
DatagramSocketImpl	NetworkInterface	URLClassLoader
HttpURLConnection	PasswordAuthentication	URLConnection
InetAddress	ServerSocket	URLDecoder
Inet4Address	Socket	URLEncoder
Inet6Address	SocketAddress	URLStreamHandler

Some of these classes are to support the new IPv6 addressing scheme. Others provide some added flexibility to the original java.net package. Java 2, version 1.4 also added functionality, such as support for the new I/O classes, to several of the preexisting networking classes. The java.net package's interfaces are listed here:

ContentHandlerFactory	SocketImplFactory	URLStreamHandlerFactory
FileNameMap	SocketOptions	
DatagramSocketImplFactory		

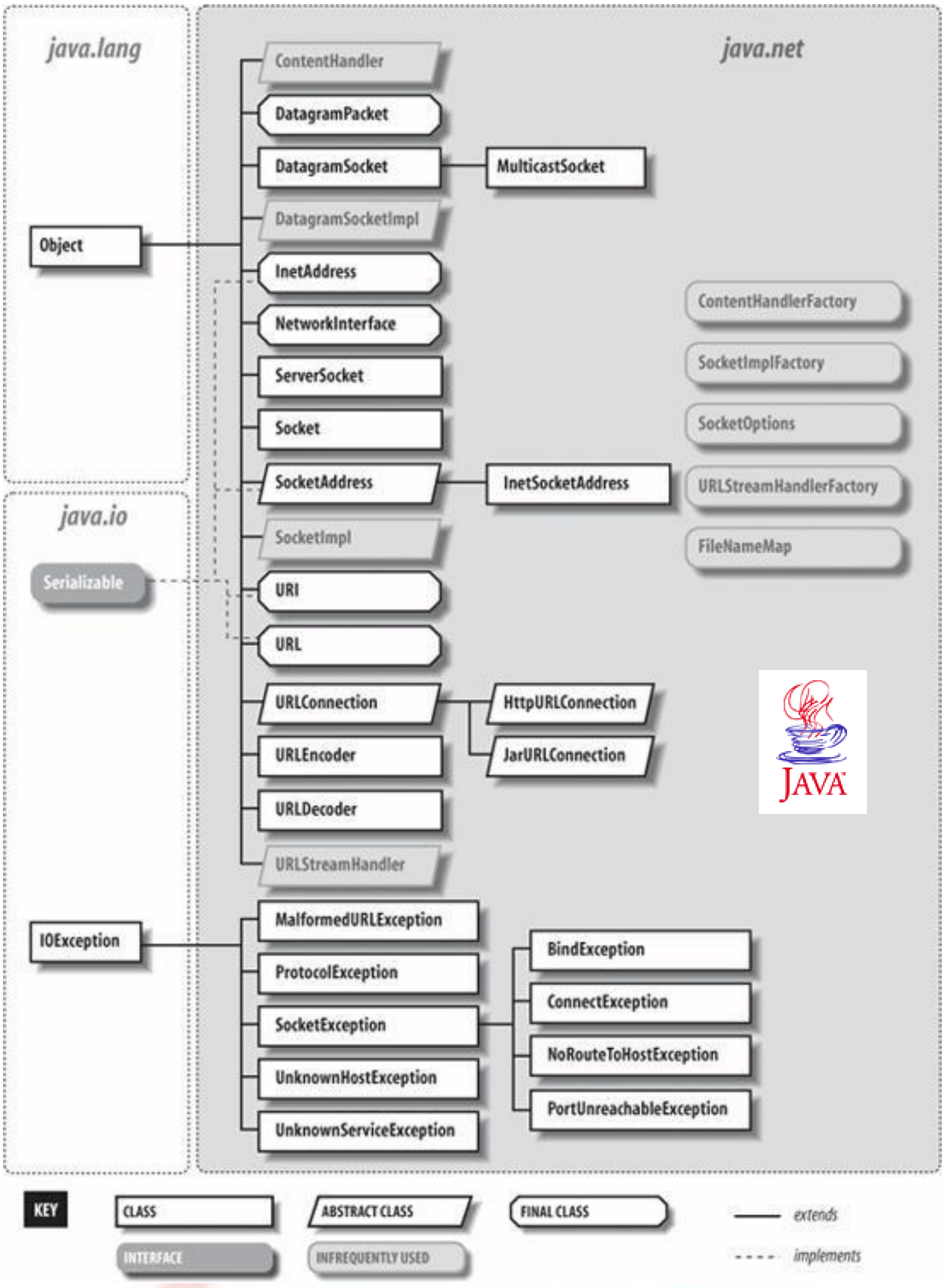

Fig. java.net package

InetAddress

Whether we are making a phone call, sending mail, or establishing a connection across the Internet, addresses are fundamental. The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address. We interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside. As of Java 2, version 1.4, InetAddress can handle both IPv4 and IPv6 addresses. This discussion assumes IPv4.

Factory Methods

The InetAddress class has no visible constructors. To create an InetAddress object, we have to use one of the available factory methods. Factory methods are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used InetAddress factory methods are shown here.

```
static InetAddress getLocalHost( ) throws UnknownHostException
static InetAddress getByName(String hostName) throws
                         UnknownHostException
static InetAddress[ ] getAllByName(String hostName) throws
                         UnknownHostException
```

The getLocalHost( ) method simply returns the InetAddress object that represents the local host. The getByName( ) method returns an InetAddress for a host name passed to it. If these methods are unable to resolve the host name, they throw an UnknownHostException.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The getAllByName( ) factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to. It will also throw an UnknownHostException if it can't resolve the name to at least one address. Java2, version1.4 also includes the factory method getByAddress(), which takes an IP address and returns an InetAddress object. Either an IPv4 or an IPv6 address can be used. The following example prints the addresses and names of the local machine and two well-known Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;
class InetAddressTest
{
	public static void main(String args[])
		throws UnknownHostException
	{
		InetAddress Address = InetAddress.getLocalHost();
		System.out.println(Address);
			Address = InetAddress.getByName("google.com");
		System.out.println(Address);
			InetAddress SW[] =
			InetAddress.getAllByName("www.yahoo.com");
		for (int i=0; i<SW.length; i++)
			System.out.println(SW[i]);
	}
}
```

```
itdept-server/194.168.1.75
google.com/209.85.171.100
www.yahoo.com/87.248.113.14
```

Instance Methods

The InetAddress class also has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the most commonly used.

`boolean equals(Object other)`

It returns true if this object has the same Internet address as other.

`byte[ ] getAddress( )`

It returns a byte array that represents the object’s Internet address in network byte order.

`String getHostAddress( )`

It returns a string that represents the host address associated with the InetAddress object.

`String getHostName( )`

It returns a string that represents the host name associated with the InetAddress object.

`boolean isMulticastAddress( )`

It returns true if this Internet address is a multicast address. Otherwise, it returns false.

`String toString( )`

It returns a string that lists the host name and the IP address for convenience.

Internet addresses are looked up in a series of hierarchically cached servers. That means that our local computer might know a particular name-to- IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn't have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server, called InterNIC (internic.net). This process might take a long time, so it is wise to structure our code so that we cache IP address information locally rather than look it up repeatedly.

TCP/IP Client Sockets

TCP/IP sockets are used to implement t reliable, bidirectional, persistent, point-to-point, and stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

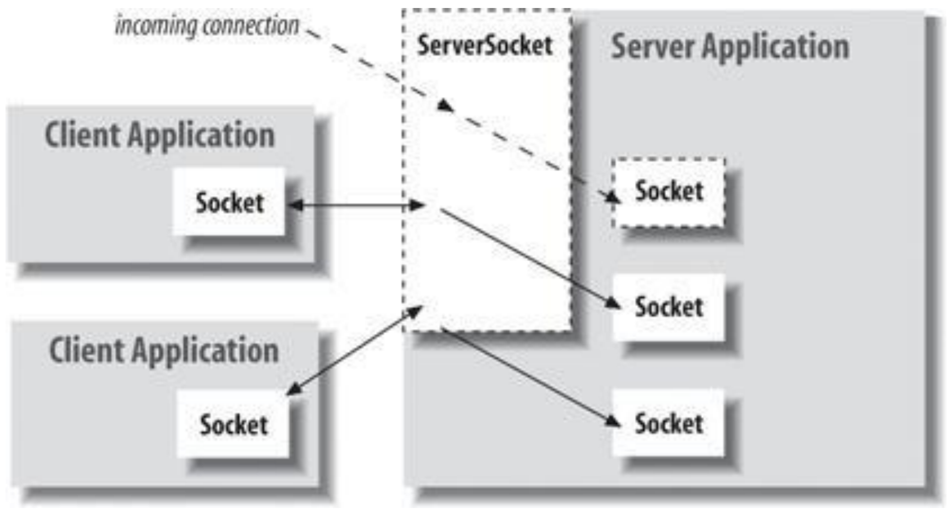

Fig. Clients and servers, Sockets and ServerSockets

Applets may only establish socket connections back to the host from which the applet was downloaded. This restriction exists because it would be dangerous for applets loaded through a firewall to have access to any arbitrary machine. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The ServerSocket class is designed to be a listener, which waits for clients to connect before doing anything. The Socket class is designed to connect to server sockets and initiate protocol exchanges.

The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

```
Socket(String hostName, int port) throws UnknownHostException, IOException
```

Creates a socket connecting the local host to the named host and port; can throw an UnknownHostException or an IOException.

```
Socket(InetAddress ipAddress, int port)throws UnknownHostException, IOException
```

Creates a socket using a preexisting InetAddress object and a port; can throw an IOException. A socket can be examined at any time for the address and port information associated with it, by use of the following methods:

```
InetAddress getInetAddress( )
```

It returns the InetAddress associated with the Socket object.

```
int getPort( )
```

It returns the remote port to which this Socket object is connected.

```
int getLocalPort( )
```

It returns the local port to which this Socket object is connected.

Once the Socket object has been created, it can also be examined to gain access to the input and output streams associated with it. Each of these methods can throw an IOException if the sockets have been invalidated by a loss of connection on the Net. These streams are used exactly like the other I/O streams to send and receive data.

```
InputStream getInputStream( )
```

This returns the InputStream associated with the invoking socket.

```
OutputStream getOutputStream( )
```

This returns the OutputStream associated with the invoking socket.

Find out which of the first 1,024 ports seem to be hosting TCP servers on a specified host

```
import java.net.*;
import java.io.*;
public class LowPortScanner
{
     public static void main(String[] args)
     {
          String host = "localhost";
          for (int i = 1; i < 1024; i++)
          {
               try {
               Socket s = new Socket(host, i);
```

```
              System.out.println("There is a server on port " + i
                   + " of " + host);
      }
      catch (UnknownHostException ex)
      {
        System.err.println(ex);
        break;
      }
      catch (IOException ex)
      {
        // must not be a server on this port
      }
    } // end for
  }  // end main
}  // end PortScanner
```

Here's the output this program produces on local host. Results will vary, depending on which ports are occupied. As a rule, more ports will be occupied on a Unix workstation than on a PC or a Mac:

```
java LowPortScanner

There is a server on port 21 of localhost
There is a server on port 80 of localhost
There is a server on port 110 of localhost
There is a server on port 135 of localhost
There is a server on port 443 of localhost
```

A daytime protocol client [Ref. 1]

```
import java.net.*;
import java.io.*;
public class DaytimeClient
{
  public static void main(String[] args)
  {
    String hostname;
    try
    {
      Socket theSocket = new Socket("localhost", 13);
      InputStream timeStream = theSocket.getInputStream( );
      StringBuffer time = new StringBuffer( );
      int c;
      while ((c = timeStream.read( )) != -1) time.append((char) c);
      String timeString = time.toString( ).trim( );
      System.out.println("It is " + timeString + " at "
          + hostname);
    }  // end try
    catch (UnknownHostException ex)
    {
      System.err.println(ex);
    }
    catch (IOException ex)
    {
      System.err.println(ex);
    }
  }  // end main
} // end DaytimeClient
```

DaytimeClient reads the hostname of a daytime server from the command line and uses it to construct a new Socket that connects to port 13 on the server. Here the National Institute of Standards and Technology's time server at time.nist.gov is used as a host name. The client then calls theSocket.getInputStream( ) to get theSocket's input stream, which is stored in the variable timeStream. Since the daytime protocol specifies ASCII, DaytimeClient doesn't bother chaining a reader to the stream. Instead, it just reads the bytes into a StringBuffer one at a time, breaking when the server closes the connection as the protocol requires it to do. Here's what happens:

```
java DaytimeClient

It is 52956 03-11-13 04:45:28 00 0 0 706.3 UTC(NIST) * at
time.nist.gov
```

Whois

The very simple example that follows opens a connection to a whois port on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to lookup the argument as a registered Internet domain name, then send back the IP address and contact information for that site.

```
//Demonstrate Sockets.
import java.net.*;
import java.io.*;
class Whois
{
     public static void main(String args[]) throws Exception
     {
          int c;
          Socket s = new Socket("internic.net", 43);
          InputStream in = s.getInputStream(); OutputStream out
          = s.getOutputStream(); Stringstr=(args.length==0?
          "google.com":args[0])+"\n";
```

```
            byte buf[] = str.getBytes();
            out.write(buf);
            while ((c = in.read()) != -1)
                  System.out.print((char) c);
            s.close();
      }
}
```

If, for example, you obtained information about osborne.com, we'd get something similar to the following:

TCP/IP Server Sockets [Ref. 2]

Java has a different socket class that must be used for creating server applications. The ServerSocket class is used to create servers that listen for either local or remote client programs to connect to them on published ports. Since the Web is driving most of the activity on the Internet, this section develops an operational web (http) server.

ServerSockets are quite different from normal Sockets. When we create a ServerSocket, it will register itself with the system as having an interest in client connections. The constructors for ServerSocket reflect the port number that we wish to accept connections on and, optionally, how long we want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50.

The ServerSocket class contains everything needed to write servers in Java. It has constructors that create new ServerSocket objects, methods that listen for connections on a specified port, methods that configure the various server socket options, and the usual miscellaneous methods such as toString().

In Java, the basic life cycle of a server program is:

1. A new ServerSocket is created on a particular port using a ServerSocket() constructor.
2. The ServerSocket listens for incoming connection attempts on that port using its accept( ) method. accept( ) blocks until a client attempts to make a connection, at which point accept( ) returns a Socket object connecting the client and the server.
3. Depending on the type of server, either the Socket's getInputStream() method, getOutputStream( ) method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both close the connection.
6. The server returns to step 2 and waits for the next connection.

The constructors might throw an IOException under adverse conditions. Here are the constructors:

```
ServerSocket(int port) throws BindException, IOException
```

It creates server socket on the specified port with a queue length of 50.

```
ServerSocket(int port, int maxQueue) throws BindException,
     IOException
```

This creates a server socket on the specified port with a maximum queue length of maxQueue.

```
ServerSocket(int port, int maxQueue, InetAddress localAddress)
     throws IOException
```

It creates a server socket on the specified port with a maximum queue length of maxQueue. On a multi-homed host, local Address specifies the IP address to which this socket binds. ServerSocket has a method called accept( ),
which is a blocking call that will wait for a client to initiate communications, and then return with a normal Socket that is then used for communication with the client.

Scanner for the server ports: [Ref. 1]

```
import java.net.*;
import java.io.*;
public class LocalPortScanner
{
  public static void main(String[] args)
  {
    for (int port = 1; port <= 65535; port++)
    {
      try
      {
        // the next line will fail and drop into the catch block if
        // there is already a server running on the port
        ServerSocket server = new ServerSocket(port);
      }
      catch (IOException ex)
      {
        System.out.println("There is a server on port " + port
                            + ".");
      } // end catch
    } // end for
  }
}
```

Accepting and Closing Connections [Ref. 1]

A ServerSocket customarily operates in a loop that repeatedly accepts connections. Each pass through the loop invokes the accept( ) method. This returns a Socket object representing the connection between the remote client and the local server. Interaction with the client takes place through this Socket object. When the transaction is finished, the server should invoke the Socket object's close() method. If the client closes the connection while the server is still operating, the input and/or output streams that connect the server to the client throw an InterruptedIOException on the next read or write. In either case, the server should then get ready to process the next incoming connection. However, when the server needs to shut down and not process any further incoming connections, we should invoke the ServerSocket object's close( ) method.

```
public Socket accept( ) throws IOException
```

When server setup is done and we're ready to accept a connection, call the ServerSocket's accept() method. This method "blocks"; that is, it stops the flow of execution and waits until a client connects. When a client does connect, the accept( ) method returns a Socket object. We use the streams returned by this Socket's getInputStream( ) and getOutputStream( ) methods to communicate with the client. For example:

```
ServerSocket server = new ServerSocket(5776);
while (true)
{
  Socket connection = server.accept( );
  OutputStreamWriter out
   = new OutputStreamWriter(connection.getOutputStream( ));
  out.write("You've connected to this server. Bye-bye now.\r\n");
  connection.close( );
}
```

If we don't want the program to halt while it waits for a connection, put the call to accept( ) in a separate thread.

When exception handling is added, the code becomes somewhat more convoluted. It's important to distinguish between exceptions that should probably shut down the server and log an error message, and exceptions that should just close that active connection. Exceptions thrown by accept( ) or the input and output streams generally should not shut down the server. Most other exceptions probably should. To do this, we'll need to nest our try blocks.

Finally, most servers will want to make sure that all sockets they accept are closed when they're finished. Even if the protocol specifies that clients are responsible for closing connections, clients do not always strictly adhere to the protocol. The call to close( ) also has to be wrapped in a try block that catches an IOException. However, if we do catch an IOException when closing the socket, ignore it. It just means that the client closed the socket before the server could. Here's a slightly more realistic example:

```
try
{
  ServerSocket server = new ServerSocket(5776);
  while (true)
  {
    Socket connection = server.accept( );
    try
    {
      Writer out
      = new OutputStreamWriter(connection.getOutputStream( ));
      out.write("You've connected to this server. Bye-bye now.");
      out.flush( );
      connection.close( );
    }
    catch (IOException ex)
    {
     // This tends to be a transitory error for this one connection;
     // e.g. the client broke the connection early. Consequently,
     // we don't want to break the loop or print an error message.
     // However, we might choose to log this exception in an error log.
    }
    finally
    {
     // Guarantee that sockets are closed when complete.
     try
     {
       if (connection != null) connection.close( );
     }
     catch (IOException ex) {}
   }
}
catch (IOException ex)
{
  System.err.println(ex);
}
```

```
     public void close( ) throws IOException
```

If we're finished with a server socket, we should close it, especially if the program is going to continue to run for some time. This frees up the port for other programs that may wish to use it. Closing a ServerSocket should not be confused with closing a Socket. Closing a ServerSocket frees a port on the local host, allowing another server to bind to the port; it also breaks all currently open sockets that the ServerSocket has accepted.

```
public InetAddress getInetAddress( )
```

This method returns the address being used by the server (the local host). If the local host has a single IP address (as most do), this is the address returned by InetAddress.getLocalHost( ). If the local host has more than one IP address, the specific address returned is one of the host's IP addresses. we can't predict which address we will get. For example:

```
ServerSocket httpd = new ServerSocket(80);
InetAddress ia = httpd.getInetAddress( );
```

If the ServerSocket has not yet bound to a network interface, this method returns null.

```
public int getLocalPort( )
```

The ServerSocket constructors allow us to listen on an unspecified port by passing 0 for the port number. This method lets us find out what port we're listening on.

Example: A Daytime server for daytime client: [Ref. 1]

```
public class DaytimeServer
{
  public static void main(String[] args)
  {
      try
   {
     ServerSocket server = new ServerSocket(13);
     Socket connection = null;
     while (true)
     {
       try
       {
         connection = server.accept( );
         Writer out = new
               OutputStreamWriter(connection.getOutputStream( ));
         Date now = new Date( );
         out.write(now.toString( ) +"\r\n");
         out.flush( );
         connection.close( );
       }
       catch (IOException ex) {}
       finally
       {
         try
         {
           if (connection != null) connection.close( );
         }
         catch (IOException ex) {}
       }
     }  // end while
   }  // end try
   catch (IOException ex)
   {
     System.err.println(ex);
   } // end catch
  } // end main
}
```

URL [Ref. 2]

The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scaleable way to locate all of the resources of the Net. Once we can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. In fact, the Web is really just that same old Internet with all of its resources addressed as URLs plus HTML. Within Java's network class library, the URL class provides a simple, concise API to access information across the Internet using URLs.

Format [Ref. 2]

Two examples of URLs are http://www.rediff.com/ and http://www.rediff.com:80/index.htm/. A URL specification is based on four components. The first is the **protocol** to use, separated from the rest of the locator by a colon (:). Common protocols are http, ftp, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if we leave off the http:// from our URL specification). The second component is the **host name** or **IP address** of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the **port number**, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus:80 is redundant.) The fourth part is the actual **file path**. Most HTTP servers will append a file named index.html or index.htm to URLs that refer directly to a directory resource. Thus, http://www.rediff.com/ is the same as http://www.rediff.com/index.htm. Java's URL class has several constructors, and each can throw a MalformedURLException. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

```
URL(String urlSpecifier)
```

The next two forms of the constructor allow you to break up the URL into its component parts:

```
URL(String protName, String hostName, int port, String path)
URL(String protName, String hostName, String path)
```

Another frequently used constructor allows us to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it's really quite easy and useful.

```
URL(URL urlObj, String urlSpecifier)
```

In the following example, we create a URL to cric-info's news page and then examine its properties:

```
// Demonstrate URL. [Ref. 2]
import java.net.*;
class URLDemo
{
    public static void main(String args[])
        throws MalformedURLException
    {
        URL hp = new URL("http://content-
          ind.cricinfo.com/ci/content/current/story/news.html");
        System.out.println("Protocol: " + hp.getProtocol());
        System.out.println("Port: " + hp.getPort());
        System.out.println("Host: " + hp.getHost());
        System.out.println("File: " + hp.getFile());
        System.out.println("Ext:" + hp.toExternalForm());
    }
}
```

Output

```
Protocol: http
Port: -1
Host: content-ind.cricinfo.com
File: /ci/content/current/story/news.html
Ext:http://content-ind.cricinfo.com/ci/content/current/story/news.html
```

Notice that the port is 1; this means that one was not explicitly set. Now that we have created a URL object, we want to retrieve the data associated with it. To access the actual bits or content information of a URL, we create a URLConnection object from it, using its openConnection( ) method, like this:

```
url.openConnection()
```

openConnection( ) has the following general form:

```
URLConnection openConnection( )
```

It returns a URLConnection object associated with the invoking URL object. It may throw an IOException.

URLConnection [Ref. 1]

URLConnection is an abstract class that represents an active connection to a resource specified by a URL. The URLConnection class has two different but related purposes. First, it provides more control over the interaction with a server (especially an HTTP server) than the URL class. With a URLConnection, we can inspect the header sent by the server and respond accordingly. We can set the header fields used in the client request. We can use a URLConnection to download binary files. Finally, a URLConnection lets us send data back to a web server with POST or PUT and use other HTTP request methods.

A program that uses the URLConnection class directly follows this basic sequence of steps:

1. Construct a URL object.
2. Invoke the URL object's openConnection( ) method to retrieve a URLConnection object for that URL.
3. Configure the URLConnection.
4. Read the header fields.
5. Get an input stream and read data.
6. Get an output stream and write data.
7. Close the connection.

We don't always perform all these steps. For instance, if the default setup for a particular kind of URL is acceptable, then we're likely to skip step 3. If we only want the data from the server and don't care about any meta-information, or if the protocol doesn't provide any meta-information, we'll skip step 4. If we only want to receive data from the server but not send data to the server, we'll skip step 6. Depending on the protocol, steps 5 and 6 may be reversed or interlaced.

The single constructor for the URLConnection class is protected:

```
protected URLConnection(URL url)
```

Consequently, unless we're sub-classing URLConnection to handle a new kind of URL (that is, writing a protocol handler), we can only get a reference to one of these objects through the openConnection( ) methods of the URL and URLStreamHandler classes. For example:

```
try {
  URL u = new URL("http://www.greenpeace.org/");
  URLConnection uc = u.openConnection( );
}
catch (MalformedURLException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Reading Data from a Server

Here is the minimal set of steps needed to retrieve data from a URL using a URLConnection object:

1. Construct a URL object.
2. Invoke the URL object's openConnection( ) method to retrieve a URLConnection object for that URL.
3. Invoke the URLConnection's getInputStream( ) method.
4. Read from the input stream using the usual stream API.
5. The getInputStream() method returns a generic InputStream that lets you read and parse the data that the server sends.
6. public InputStream getInputStream( )

Example- Download a web page with a URLConnection

```
import java.net.*;
import java.io.*;
public class SourceViewer2 {
  public static void main (String[] args) {
    if  (args.length > 0) {
      try {
        //Open the URLConnection for reading
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection( );
        InputStream raw = uc.getInputStream( );
        InputStream buffer = new BufferedInputStream(raw);
        // chain the InputStream to a Reader
        Reader r = new InputStreamReader(buffer);
        int c;
        while ((c = r.read( )) != -1) {
          System.out.print((char) c);
        }
      }
      catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    } //  end if
  } // end main
}  // end SourceViewer2
```

The differences between URL and URLConnection aren't apparent with just a simple input stream as in this example. The biggest differences between the two classes are: [Ref. 1]

1. URLConnection provides access to the HTTP header.
2. URLConnection can configure the request parameters sent to the server.
3. URLConnection can write data to the server as well as read data from the server.

Reading the Header

HTTP servers provide a substantial amount of information in the header that precedes each response. For example, here's a typical HTTP header returned by an Apache web server:

```
HTTP/1.1 200 OK
Date: Mon, 18 Oct 1999 20:06:48 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Last-Modified: Mon, 18 Oct 1999 12:58:21 GMT
ETag: "1e05f2-89bb-380b196d"
Accept-Ranges: bytes
Content-Length: 35259
Connection: close
Content-Type: text/html
```

1. `public String getContentType( )`

This method returns the MIME content type of the data. It relies on the web server to send a valid content type.
For Example:
`text/plain, image/gif, application/xml, and image/jpeg.`

```
Content-type: text/html; charset=UTF-8
```
or
```
Content-Type: text/xml; charset=iso-2022-jp
```

2. `public int getContentLength( )`

The getContentLength( ) method tells us how many bytes there are in the content. Many servers send Content-length headers only when they're transferring a binary file, not when transferring a text file. If there is no Content-length header, getContentLength() returns -1. The method throws no exceptions. It is used when we need to know exactly how many bytes to read or when we need to create a buffer large enough to hold the data in advance.

3. `public long getDate( )`

The getDate( ) method returns a long that tells us when the document was sent, in milliseconds since midnight, Greenwich Mean Time (GMT), January 1, 1970. We can convert it to a java.util.Date. For example:

```
Date documentSent = new Date(uc.getDate( ));
```

This is the time the document was sent as seen from the server; it may not agree with the time on our local machine. If the HTTP header does not include a Date field, getDate( ) returns 0.

4. `public long getExpiration( )`

Some documents have server-based expiration dates that indicate when the document should be deleted from the cache and reloaded from the server. getExpiration( ) is very similar to getDate( ), differing only in how the return value is interpreted. It returns a long indicating the number of milliseconds after 12:00 A.M., GMT, January 1, 1970, at which point the document expires. If the HTTP header does not include an Expiration field, getExpiration( ) returns
0, which means 12:00 A.M., GMT, January 1, 1970. The only reasonable interpretation of this date is that the document does not expire and can remain in the cache indefinitely.

5. `public long getLastModified( )`

The final date method, getLastModified( ), returns the date on which the document was last modified. Again, the date is given as the number of milliseconds since midnight, GMT, January 1, 1970. If the HTTP header does not include a Last-modified field (and many don't), this method returns 0.

Example:

```
import java.net.*;
import java.io.*;
import java.util.*;
public class HeaderViewer
{
  public static void main(String args[])
  {
      try
      {
        URL u = new URL("http://www.rediffmail.com/index.html");
        URLConnection uc = u.openConnection( );
        System.out.println("Content-type: " +
               uc.getContentType( ));
        System.out.println("Content-encoding: "
         + uc.getContentEncoding( ));
        System.out.println("Date: " + new Date(uc.getDate( )));
        System.out.println("Last modified: "
         + new Date(uc.getLastModified( )));
        System.out.println("Expiration date: "
         + new Date(uc.getExpiration( )));
        System.out.println("Content-length: " +
               uc.getContentLength( ));
      }  // end try
      catch (MalformedURLException ex)
      {
        System.out.println("I can't understand this URL...");
      }
      catch (IOException ex)
      {
        System.err.println(ex);
      }
      System.out.println( );
  }  // end main
}  // end HeaderViewer
```

Sample output:

```
Content-type: text/html
Content-encoding: null
Date: Mon Oct 18 13:54:52 PDT 1999
Last modified: Sat Oct 16 07:54:02 PDT 1999
Expiration date: Wed Dec 31 16:00:00 PST 1969
Content-length: -1
```

Sample output for: http://www.oreilly.com/graphics/space.gif

```
Content-type: image/gif
Content-encoding: null
Date: Mon Oct 18 14:00:07 PDT 1999
```

```
Last modified: Thu Jan 09 12:05:11 PST 1997
Expiration date: Wed Dec 31 16:00:00 PST 1969
Content-length: 57
```

Retrieving Arbitrary Header Fields

The last six methods requested specific fields from the header, but there's no theoretical limit to the number of header fields a message can contain. The next five methods inspect arbitrary fields in a header. Indeed, the methods of the last section are just thin wrappers over the methods discussed here; we can use these methods to get header fields that Java's designers did not plan for. If the requested header is found, it is returned. Otherwise, the method returns null.

```
public String getHeaderField(String name)
```

The getHeaderField() method returns the value of a named header field. The name of the header is not case-sensitive and does not include a closing colon. For example, to get the value of the Content-type and Content-encoding header fields of a URLConnection object uc, we could write:

```
String contentType = uc.getHeaderField("content-type");

String contentEncoding = uc.getHeaderField("content-encoding"));
```

To get the Date, Content-length, or Expires headers, you'd do the same:

```
String data = uc.getHeaderField("date");

String expires = uc.getHeaderField("expires");

String contentLength = uc.getHeaderField("Content-length");
```

These methods all return String, not int or long as the getContentLength( ), getExpirationDate(), getLastModified( ), and getDate( ) methods of the last section did. If we're interested in a numeric value, convert the String to a long or an int.

Do not assume the value returned by getHeaderField() is valid. We must check to make sure it is non-null.

```
public String getHeaderFieldKey(int n)
```

This method returns the key (that is, the field name: for example, Content-length or Server) of the nth header field. The request method is header zero and has a null key. The first header is one. For example, to get the sixth key of the header of the URLConnection uc, we would write:

```
String header6 = uc.getHeaderFieldKey(6);
```

```
public String getHeaderField(int n)
```

This method returns the value of the nth header field. In HTTP, the request method is header field zero and the first actual header is one. Example below uses this method in conjunction with getHeaderFieldKey( ) to print the entire HTTP header.

```
//Print the entire HTTP header
import java.net.*;
import java.io.*;
public class AllHeaders {
  public static void main(String args[]) {
    for (int i=0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection( );
        for (int j = 1; ; j++) {
          String header = uc.getHeaderField(j); if (header ==
          null) break;
          System.out.println(uc.getHeaderFieldKey(j) + ": " +
               header);
        }  // end for
      }  // end try
      catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a URL I
               understand.");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
      System.out.println( );
    }  // end for
  }  // end main
}  // end AllHeaders
```

For example, here's the output when this program is run against http://www.oreilly.com:

```
java AllHeaders http://www.oreilly.com

Server: WN/1.15.1
Date: Mon, 18 Oct 1999 21:20:26 GMT
Last-modified: Sat, 16 Oct 1999 14:54:02 GMT
Content-type: text/html
Title: www.oreilly.com -- Welcome to O'Reilly &amp; Associates!
```

```
-- computer  books, software, online publishing
Link: <mailto:webmaster@oreilly.com>; rev="Made"
```

Besides Date, Last-modified, and Content-type headers, this server also provides Server, Title, and Link headers. Other servers may have different sets of headers.

```
public long getHeaderFieldDate(String name, long default)
```

This method first retrieves the header field specified by the name argument and tries to convert the string to a long that specifies the milliseconds since midnight, January 1, 1970, GMT. getHeaderFieldDate() can be used to retrieve a header field that represents a date: for example, the Expires, Date, or Last-modified headers. To convert the string to an integer, getHeaderFieldDate() uses the parseDate( ) method of java.util.Date. The parseDate() method does a decent job of understanding and converting most common date formats, but it can be stumped—for instance, if we ask for a header field that contains something other than a date. If parseDate( ) doesn't understand the date or if getHeaderFieldDate( ) is unable to find the requested header field, getHeaderFieldDate( ) returns the default argument. For example:

```
    Date expires = new Date(uc.getHeaderFieldDate("expires", 0));
    long lastModified = uc.getHeaderFieldDate("last-modified", 0);
    Date now = new Date(uc.getHeaderFieldDate("date", 0));
```

We can use the methods of the java.util.Date class to convert the long to a String.

```
public int getHeaderFieldInt(String name, int default)
```

This method retrieves the value of the header field name and tries to convert it to an int. If it fails, either because it can't find the requested header field or because that field does not contain a recognizable integer, getHeaderFieldInt( ) returns the default argument. This method is often used to retrieve the Content-length field. For example, to get the content length from a URLConnection uc, you would write:

```
int contentLength = uc.getHeaderFieldInt("content-length", -1);
```

In this code fragment, getHeaderFieldInt( ) returns -1 if the Content- length header isn't present.

Configuring the Connection

The URLConnection class has seven protected instance fields that define exactly how the client makes the request to the server. These are:

```
protected URL     url;
protected boolean doInput = true;
protected boolean doOutput = false;
protected boolean allowUserInteraction =
                         defaultAllowUserInteraction;
protected boolean useCaches = defaultUseCaches;
protected long    ifModifiedSince = 0;
protected boolean connected = false;
```

For instance, if *doOutput* is true, we'll be able to write data to the server over this URLConnection as well as read data from it. If *useCaches* is false, the connection bypasses any local caching and downloads the file from the server afresh.

Since these fields are all protected, their values are accessed and modified via obviously named setter and getter methods:

```
public URL     getURL( )
public void    setDoInput(boolean doInput)
public boolean getDoInput( )
public void    setDoOutput(boolean doOutput)
public boolean getDoOutput( )
public void    setAllowUserInteraction(boolean allow)
public boolean getAllowUserInteraction( )
public void    setIfModifiedSince(long since)
public long    getIfModifiedSince( )
```

User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) is an alternative protocol for sending data over IP that is very quick, but not reliable. That is, when you send UDP data, you have no way of knowing whether it arrived, much less whether different pieces of data arrived in the order in which you sent them. However, the pieces that do arrive generally arrive quickly.

The difference between TCP and UDP is often explained by analogy with the phone system and the post office. TCP is like the phone system. When we dial a number, the phone is answered and a connection is established between the two parties. As we talk, we know that the other party hears our words in the order in which we say them. If the phone is busy or no one answers, we find out right away. UDP, by contrast, is like the postal system. We send packets of mail to an address. Most of the letters arrive, but some may be lost on the way. The letters probably arrive in the order in which we sent them, but that's not guaranteed. The farther away we are from our recipient, the more likely it is that mail will be lost on the way or arrive out of order. If this is a problem, we can write sequential numbers on the envelopes, then ask the recipients to arrange them in the correct order and send us mail telling us which letters arrived so that we can resend any that didn't get there the first time. However, we and our correspondent need to agree on this protocol in advance. The post office will not do it for us.

Both the phone system and the post office have their uses. Although either one could be used for almost any communication, in some cases one is definitely superior to the other. The same is true of UDP and TCP.

Comparing TCP and UDP

No.	TCP	UDP
1	This is connection oriented protocol	This is connection-less protocol
2	The TCP connection is byte stream	The UDP connection is a message stream
3	It does not support multicasting and broadcasting	It supports broadcasting
4	It provides error control and flow control	The error control and flow control is not provided
5	TCP supports full duplex transmission	UDP does not support full duplex transmission
6	It is reliable service of data transmission	This is an unreliable service of data transmission
7	The TCP packet is called as segment	The UDP packet is called as user datagram.

Java's implementation of UDP is split into two classes: DatagramPacket and DatagramSocket. The DatagramPacket class stuffs bytes of data into UDP packets called datagrams and lets us unstuff datagrams that we receive. A DatagramSocket sends as well as receives UDP datagrams. To send data, we put the data in a DatagramPacket and send the packet using a DatagramSocket. To receive data, we receive a DatagramPacket object from a DatagramSocket and then read the contents of the packet. The sockets themselves are very simple creatures. In UDP, everything about a datagram, including the address to which it is directed, is included in the packet itself; the socket only needs to know the local port on which to listen or send.

Datagrams

For most of our internetworking needs, we will be happy with TCP/IP- style networking. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn’t been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: The DatagramPacket object is the data container, while the DatagramSocket is the mechanism used to send or receive the DatagramPackets.

DatagramPacket

DatagramPacket uses different constructors depending on whether the packet will be used to send data or to receive data. This is a little unusual. Normally, constructors are overloaded to let us provide different kinds of information when we create an object, not to create objects of the same class that will be used in different contexts. In this case, all constructors take as arguments a byte array that holds the datagram's data and the number of bytes in that array to use for the datagram's data. When we want to receive a datagram, these are the only arguments we provide; in addition, the array should be empty. When the socket receives a datagram from the network, it stores the datagram's data in the DatagramPacket object's buffer array, up to the length we specified.

Constructors for receiving datagrams

These two constructors create new DatagramPacket objects for receiving data from the network:

```
public DatagramPacket(byte[] buffer, int length)
public DatagramPacket(byte[] buffer, int offset, int length)
```

When a socket receives a datagram, it stores the datagram's data part in buffer beginning at *buffer*[0] and continuing until the packet is completely stored or until *length* bytes have been written into the *buffer*. If the second constructor is used, storage begins at *buffer*[*offset*] instead. Otherwise, these two constructors are identical. *length* must be less than or equal to buffer.length-offset. If we try to construct a DatagramPacket with a length that will overflow the buffer, the constructor throws an *IllegalArgumentException*. This is a RuntimeException, so our code is not required to catch it. It is okay to construct a DatagramPacket with a length less than buffer.length-offset. In this case, at most the first length bytes of buffer will be filled when the datagram is received. For example, this code fragment creates a new DatagramPacket for receiving a datagram of up to 8,192 bytes:

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
```

The constructor doesn't care how large the buffer is and would happily let us create a DatagramPacket with megabytes of data. However, the underlying native network software is less forgiving, and most native UDP implementations don't support more than 8,192 bytes of data per datagram.

Constructors for sending datagrams [Ref. 1]

These two constructors create new DatagramPacket objects for sending data across the network:

```
public DatagramPacket(byte[] data, int length,
               InetAddress destination, int port)
public DatagramPacket(byte[] data, int offset, int length,
               InetAddress destination, int port)
```

Each constructor creates a new DatagramPacket to be sent to another host. The packet is filled with *length* bytes of the *data* array starting at *offset* or
0 if *offset* is not used. If we try to construct a DatagramPacket with a *length* that is greater than *data*.*length*, the constructor throws an *IllegalArgumentException*. It's okay to construct a DatagramPacket object with an *offset* and a *length* that will leave extra, unused space at the end of the

data array. In this case, only *length* bytes of *data* will be sent over the network. The *InetAddress* object destination points to the host we want the packet delivered to; the int argument port is the port on that host.

For instance, this code fragment creates a new DatagramPacket filled with the data "This is a test" in ASCII. The packet is directed at port 7 (the echo port) of the host www.ibiblio.org:

```
String s = "This is a test";
byte[] data = s.getBytes("ASCII");
try {
  InetAddress ia = InetAddress.getByName("www.ibiblio.org");
  int port = 7;
  DatagramPacket dp = new
          DatagramPacket(data, data.length, ia, port);
  // send the packet...
}
catch (IOException ex){ }
```

The get Methods

DatagramPacket has six methods that retrieve different parts of a datagram: the actual data plus several fields from its header. These methods are mostly used for datagrams received from the network.

```
public InetAddress getAddress( )
```

The getAddress( ) method returns an InetAddress object containing the address of the remote host. If the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address). On the other hand, if the datagram was created locally to be sent to a remote machine, this method returns the address of the host to which the datagram is addressed (the destination address).

```
public int getPort( )
```

The getPort( ) method returns an integer specifying the remote port. If this datagram was received from the Internet, this is the port on the host that sent the packet. If the datagram was created locally to be sent to a remote host, this is the port to which the packet is addressed on the remote machine.

```
public SocketAddress getSocketAddress( )
```

The getSocketAddress() method returns a SocketAddress object containing the IP address and port of the remote host. As is the case for getInetAddress( ), if the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address).

```
public byte[] getData( )
```

The getData( ) method returns a byte array containing the data from the datagram. It's often necessary to convert the bytes into some other form of data before they'll be useful to our program. One way to do this is to change the byte array into a String using the following String constructor:

```
public String(byte[] buffer, String encoding)
```

If the datagram does not contain text, converting it to Java data is more difficult. One approach is to convert the byte array returned by getData() into a ByteArrayInputStream using this constructor:

```
public ByteArrayInputStream(byte[] buffer, int offset, int length)
```

'buffer' is the byte array to be used as an InputStream. It's important to specify the portion of the buffer that we want to use as an InputStream using the offset and length arguments.

```
public int getLength( )
```

The getLength( ) method returns the number of bytes of data in the datagram. This is not necessarily the same as the length of the array returned by getData( ), i.e., getData( ).length. The int returned by getLength( ) may be less than the length of the array returned by getData( ).

```
public int getOffset( )
```

This method simply returns the point in the array returned by getData( ) where the data from the datagram begins.

Following program uses all the methods covered above to print the information in the DatagramPacket. This example is a little artificial; because the program creates a DatagramPacket, it already knows what's in it. More often, we'll use these methods on a DatagramPacket received from the network, but that will have to wait for the introduction of the DatagramSocket class in the next section.

```
//Construct a DatagramPacket to receive data [Ref. 1]
import java.net.*;
public class DatagramExample1
{
  public static void main(String[] args)
  {
    String s = "This is a test.";
    byte[] data = s.getBytes( );
    try
    {
      InetAddress ia = InetAddress.getByName("www.ibiblio.org");
      int port = 7;
      DatagramPacket dp
       = new DatagramPacket(data, data.length, ia, port);
      System.out.println("This packet is addressed to "
       + dp.getAddress( ) + " on port " + dp.getPort( ));
      System.out.println("There are " + dp.getLength( )
       + " bytes of data in the packet");
      System.out.println(
      new String(dp.getData( ), dp.getOffset( ), dp.getLength( )));
    }
    catch (UnknownHostException e)
    {
      System.err.println(e);
    }
  }
}
```

Output:

```
This packet is addressed to www.ibiblio.org/154.2.254.81 on port 7
There are 15 bytes of data in the packet
This is a test.
```

The Set methods

`public void setData(byte[] data)`

The setData( ) method changes the payload of the UDP datagram. We might use this method if we are sending a large file (where large is defined as "bigger than can comfortably fit in one datagram") to a remote host. We could repeatedly send the same DatagramPacket object, just changing the data each time.

`public void setAddress(InetAddress remote)`

The setAddress( ) method changes the address a datagram packet is sent to. This might allow us to send the same datagram to many different recipients.

`public void setPort(int port)`

The setPort( ) method changes the port a datagram is addressed to.

`public void setLength(int length)`

The setLength( ) method changes the number of bytes of data in the internal buffer that are considered to be part of the datagram's data as opposed to merely unfilled space.

DatagramSocket [Ref. 1]

For sending or receiving a DatagramPacket, we must open a datagram socket. In Java, a datagram socket is created and accessed through the DatagramSocket class. The constructors are listed below:

`public DatagramSocket( ) throws SocketException`

```
public DatagramSocket(int port) throws SocketException
public DatagramSocket(int port, InetAddress interface)
     throws SocketException
```

The first constructor creates a socket that is bound to an anonymous port. Second constructor creates a socket that listens for incoming datagrams on a particular port, specified by the *port* argument. Third constructor is primarily used on multihomed hosts; it creates a socket that listens for incoming datagrams on a specific port and network interface. The port argument is the port on which this socket listens for datagrams. As with TCP sockets, we need to be root on a Unix system to create a DatagramSocket on a
port below 1,024. The address argument is an InetAddress object matching one of the host's network addresses.

//The UDP Port Scanner [Ref. 1].

```
import java.net.*;
public class UDPPortScanner
{
  public static void main(String[] args)
  {
    for (int port = 0; port <= 2000; port++)
    {
      try
      {
        // the next line will fail and drop into the catch block if
        // there is already a server running on port i
        DatagramSocket server = new DatagramSocket(port);
        server.close( );
      }
      catch (SocketException ex)
      {
        System.out.println("There is a server on port " +
                    port + ".");
      } // end try
    } // end for
  }
}
```

Sample output:

```
There is a server on port 123.
There is a server on port 445.
There is a server on port 500.
There is a server on port 1900.
```

Sending and Receiving Datagrams

The primary task of the DatagramSocket class is to send and receive UDP datagrams. One socket can both send and receive. Indeed, it can send and receive to and from multiple hosts at the same time.

```
public void send(DatagramPacket dp) throws IOException
```

Once a DatagramPacket is created and a DatagramSocket is constructed, send the packet by passing it to the socket's send() method. For example, if theSocket is a DatagramSocket object and theOutput is a DatagramPacket object, send theOutput using theSocket like this:

```
theSocket.send(theOutput);
```

If there's a problem sending the data, an IOException may be thrown. However, this is less common with DatagramSocket than Socket or ServerSocket, since the unreliable nature of UDP means we won't get an exception just because the packet doesn't arrive at its destination. We may get an IOException if we are trying to send a larger datagram than the host's native networking software supports, but then again we may not. This method may also throw a SecurityException if the SecurityManager won't let you communicate with the host to which the packet is addressed.

```
public void receive(DatagramPacket dp) throws IOException
```

This method receives a single UDP datagram from the network and stores it in the preexisting DatagramPacket object *dp*. Like the accept( ) method in the ServerSocket class, this method blocks the calling thread until a datagram arrives. If our program does anything besides wait for datagrams, we should call receive() in a separate thread.

The datagram's buffer should be large enough to hold the data received. If not, receive( ) places as much data in the buffer as it can hold; the rest is lost. It may be useful to remember that the maximum size of the data portion of a UDP datagram is 65,507 bytes. (That's the 65,536-byte maximum size of an IP datagram minus the 20-byte size of the IP header and the 8-byte size of the UDP header.) Some application protocols that use UDP further restrict the maximum number of bytes in a packet; for instance, NFS uses a maximum packet size of 8,192 bytes.

If there's a problem receiving the data, an IOException may be thrown. In practice, this is rare. Unlike send( ), this method does not throw a SecurityException if an applet receives a datagram from other than the applet host. However, it will silently discard all such packets.

Example: UDP Client

```
import java.net.*;
import java.io.*;
public class UDPDiscardClient
{
  public static void main(String[] args)
  {
    String hostname = "localhost";
    int port = 9;
    try
    {
      InetAddress server = InetAddress.getByName(hostname);
      BufferedReader userInput
       = new BufferedReader(new InputStreamReader(System.in));
      DatagramSocket theSocket = new DatagramSocket( );
      while (true)
      {
        String theLine = userInput.readLine( );
        if (theLine.equals(".")) break;
        byte[] data = theLine.getBytes( );
        DatagramPacket theOutput
         = new DatagramPacket(data, data.length, server, port);
        theSocket.send(theOutput);
      }  // end while
    }  // end try
    catch (UnknownHostException uhex) {
      System.err.println(uhex);
    }
    catch (SocketException se) {
      System.err.println(se);
    }
    catch (IOException ioex) {
      System.err.println(ioex);
    }
  }  // end main
}
```

//UDP Server

```
import java.net.*;
import java.io.*;
public class UDPDiscardServer
{
  public static void main(String[] args)
  {
    int port = 9;
    byte[] buffer = new byte[65507];
    try
    {
      DatagramSocket server = new DatagramSocket(port);
      DatagramPacket packet = new
               DatagramPacket(buffer, buffer.length);
      while (true)
      {
        try
        {
          server.receive(packet);
          String s = new String(packet.getData( ),
               0, packet.getLength( ));
          System.out.println(packet.getAddress( ) + " at port "
           + packet.getPort( ) + " says " + s);
          // reset the length for the next packet
          packet.setLength(buffer.length);
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
       } // end while
    }  // end try
    catch (SocketException  ex)
    {
      System.err.println(ex);
    }  // end catch
  }  // end main
}
```

Sample outputs:

```
java  UDPDiscardClient
Hi
How
are
You?
Fine?

java  UDPDiscardServer
/127.0.0.1 at port 1027 says Hi
/127.0.0.1 at port 1027 says How
/127.0.0.1 at port 1027 says are
/127.0.0.1 at port 1027 says You?
/127.0.0.1 at port 1027 says Fine?
```

```
     public void close( )
```

Calling a DatagramSocket object's close( ) method frees the port occupied by that socket. For example:

```
try {
  DatagramSocket server = new DatagramSocket( );
  server.close( );
}
catch (SocketException ex) {
  System.err.println(ex);
}
```

It's never a bad idea to close a DatagramSocket when we're through with it; it's particularly important to close an unneeded socket if the program will continue to run for a significant amount of time.

```
public InetAddress getLocalAddress( )
```

A DatagramSocket's getLocalAddress( ) method returns an InetAddress object that represents the local address to which the socket is bound. It's rarely needed in practice. Normally, you either know or don't care which address a socket is listening to.

```
public void connect(InetAddress host, int port)
```

The connect( ) method doesn't really establish a connection in the TCP sense. However, it does specify that the DatagramSocket will send packets to and receive packets from only the specified remote *host* on the specified remote *port*. Attempts to send packets to a different *host* or *port* will throw an IllegalArgumentException. Packets received from a different *host* or a different *port* will be discarded without an exception or other notification.

```
public void disconnect( )
```

The disconnect( ) method breaks the "connection" of a connected DatagramSocket so that it can once again send packets to and receive packets from any host and port.

```
public int getPort( )
```

If and only if a DatagramSocket is connected, the getPort( ) method returns the remote port to which it is connected. Otherwise, it returns -1.

```
public InetAddress getInetAddress( )
```

If and only if a DatagramSocket is connected, the getInetAddress( ) method returns the address of the remote host to which it is connected. Otherwise, it returns null.
