CHAPTER 21
Networking

As all readers know, Java is practically a synonym for Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, cross-platform, portable code. However, one of the most important reasons that Java is the premier language for network programming are the classes defined in the **java.net** package. They provide an easy-to-use means by which programmers of all skill levels can access network resources.

This chapter explores the **java.net** package. It is important to emphasize that networking is a very large and at times complicated topic. It is not possible for this book to discuss all of the capabilities contained in **java.net.** Instead, this chapter focuses on several of its core classes and interfaces.

Networking Basics

Before we begin, it will be useful to review some key networking concepts and terms. At the core of Java's networking support is the concept of a *socket*. A socket identifies an endpoint in a network. The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term *Berkeley socket* is also used. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use 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.

Socket communication takes place via a protocol. *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 retransmitting them as necessary to reliably

transmit 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.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for FTP; 23 is for Telnet; 25 is for e-mail; 43 is for whois; 80 is for HTTP; 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is a quite simple protocol for a basic page-browsing web server. Here's how it works. When a client requests a file from an HTTP server, an action known as a *hit,* it simply sends 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 to tell the client whether or not the request can be fulfilled and why.

A key component of the Internet is the *address*. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-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, organized into eight 16-bit chunks. 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, when using Java, you won't normally need to worry about whether IPv4 or IPv6 addresses are used because Java handles the details for you.

Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its *domain name,* describes a machine's location in a name space. For example, **www.HerbSchildt.com** is in the *COM* top-level domain (reserved for U.S. commercial sites); it is called *HerbSchildt*, and *www* identifies the server for web requests. An Internet domain name is mapped to an IP address by the *Domain Naming Service (DNS)*. This enables users to work with domain names, but the Internet operates on IP addresses.

The Networking Classes and Interfaces

Java supports TCP/IP both by extending the already established stream I/O interface introduced in Chapter 19 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 classes contained in the **java.net** package are shown here:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager	MulticastSocket	StandardSocketOption (Added by JDK 7.)
DatagramPacket	NetPermission	URI
DatagramSocket	NetworkInterface	URL
DatagramSocketImpl	PasswordAuthentication	URLClassLoader
HttpCookie	Proxy	URLConnection
HttpURLConnection	ProxySelector	URLDecoder
IDN	ResponseCache	URLEncoder
Inet4Address	SecureCacheResponse	URLStreamHandler

The **java.net** package's interfaces are listed here:

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily (Added by JDK 7.)	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption (Added by JDK 7.)	

In the sections that follow, we will examine the main networking classes and show several examples that apply to them. Once you understand these core networking classes, you will be able to easily explore the others on your own.

InetAddress

The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address. You 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. **InetAddress** can handle both IPv4 and IPv6 addresses.

Factory Methods

The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you

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 **InetAddress**es 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.

InetAddress 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 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("www.HerbSchildt.com");
    System.out.println(Address);

    InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
    for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
  }
}
```

Here is the output produced by this program. (Of course, the output you see may be slightly different.)

```
default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/216.66.31.161
www.nba.com/216.66.31.179
```

Instance Methods

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

boolean equals(Object *other*)	Returns **true** if this object has the same Internet address as *other*.
byte[ ] getAddress( )	Returns a byte array that represents the object's IP address in network byte order.
String getHostAddress( )	Returns a string that represents the host address associated with the **InetAddress** object.
String getHostName( )	Returns a string that represents the host name associated with the **InetAddress** object.
boolean isMulticastAddress( )	Returns **true** if this address is a multicast address. Otherwise, it returns **false**.
String toString( )	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 your 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. This process might take a long time, so it is wise to structure your code so that you cache IP address information locally rather than look it up repeatedly.

Inet4Address and Inet6Address

Beginning with version 1.4, Java has included support for IPv6 addresses. Because of

this, two subclasses of **InetAddress** were created: **Inet4Address** and **Inet6Address**. **Inet4Address** represents a traditional-style IPv4 address. **Inet6Address** encapsulates a new-style IPv6 address. Because they are subclasses of **InetAddress,** an **InetAddress** reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use **InetAddress** when working with IP addresses because it can accommodate both styles.

TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, 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.

NOTE

As a general rule, 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. Thus, **ServerSocket** is for servers. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined here.

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 connected to the named host and port.
Socket(InetAddress *ipAddress*, int *port*) throws IOException	Creates a socket using a preexisting **InetAddress** object and a port.

Socket defines several instance methods. For example, 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( )	Returns the **InetAddress** associated with the **Socket** object. It returns **null** if the socket is not connected.
int getPort( )	Returns the remote port to which the invoking **Socket** object is connected. It returns 0 if the socket is not connected.
int getLocalPort( )	Returns the local port to which the invoking **Socket** object is bound. It returns –1 if the socket is not bound.

You can gain access to the input and output streams associated with a **Socket** by use of the **getInputStream( )** and **getOuptutStream( )** methods, as shown here. Each can throw an **IOException** if the socket has been invalidated by a loss of connection. These streams are used exactly like the I/O streams described in Chapter 19 to send and receive data.

InputStream getInputStream( ) throws IOException	Returns the **InputStream** associated with the invoking socket.
OutputStream getOutputStream( ) throws IOException	Returns the **OutputStream** associated with the invoking socket.

Several other methods are available, including **connect( ),** which allows you to specify a new connection; **isConnected( ),** which returns true if the socket is connected to a server; **isBound( ),** which returns true if the socket is bound to an address; and **isClosed( ),** which returns true if the socket is closed. To close a socket, call **close( ).** Closing a socket also closes the I/O streams associated with the socket. Beginning with JDK 7, **Socket** also implements **AutoCloseable**, which means that you can use a **try**-with-resources block to manage a socket.

The following program provides a simple **Socket** example. It opens a connection to a "whois" port (port 43) on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, and 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;

    // Create a socket connected to internic.net, port 43.
    Socket s = new Socket("whois.internic.net", 43);

    // Obtain input and output streams.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construct a request string.

    String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
    // Convert to bytes.
    byte buf[] = str.getBytes();

    // Send request.
    out.write(buf);

    // Read and display response.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
    s.close();
  }
}
```

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

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: MHPROFESSIONAL.COM
   Registrar: MELBOURNE IT, LTD. D/B/A INTERNET NAMES WORLDWIDE
   Whois Server: whois.melbourneit.com
   Referral URL: http://www.melbourneit.com
   Name Server: NS1.MHEDU.COM
   Name Server: NS2.MHEDU.COM
.
.
.
```

Here is how the program works. First, a **Socket** is constructed that specifies the host name "whois.internic.net" and the port number 43. **Internic.net** is the InterNIC web site that handles whois requests. Port 43 is the whois port. Next, both input and output streams are opened on the socket. Then, a string is constructed that contains the name of the web site you want to obtain information about. In this case, if no web site is specified on the command line, then "MHProfessional.com" is used. The string is converted into a **byte** array and then sent out of the socket. The response is read by

inputting from the socket, and the results are displayed. Finally, the socket is closed, which also closes the I/O streams.

In the preceding example, the socket was closed manually by calling **close( )**. If you are using JDK 7 or later, then you can use a **try**-with-resources block to automatically close the socket. For example, here is another way to write the **main( )** method of the previous program:

```
// Use try-with-resources to close a socket.
public static void main(String args[]) throws Exception {
  int c;

  // Create a socket connected to internic.net, port 43. Manage this
  // socket with a try-with-resources block.
  try ( Socket s = new Socket("whois.internic.net", 43) ) {

    // Obtain input and output streams.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construct a request string.
    String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
    // Convert to bytes.
    byte buf[] = str.getBytes();

    // Send request.
    out.write(buf);

    // Read and display response.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
  }
  // The socket is now closed.
}
```

In this version, the socket is automatically closed when the **try** block ends.

So the examples will work with versions of Java prior to JDK 7, and to clearly illustrate when a network resource can be closed, subsequent examples will continue to call **close( )** explicitly. However, in your own code, you should consider using automatic resource management since it offers a more streamlined approach. One other point: In this version, exceptions are still thrown out of **main( )**, but they could be handled by adding **catch** clauses to the end of the **try**-with-resources block.

NOTE

For simplicity, the examples in this chapter simply throw all exceptions out of **main(**

). This allows the logic of the network code to be clearly illustrated. However, in real-world code, you will normally need to handle the exceptions in an appropriate way.

URL

The preceding example was rather obscure because the modern Internet is not about the older protocols such as whois, finger, and FTP. It is about WWW, the World Wide Web. 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 scalable way to locate all of the resources of the Net. Once you 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. Within Java's network class library, the **URL** class provides a simple, concise API to access information across the Internet using URLs.

All URLs share the same basic format, although some variation is allowed. Here are two examples: **http://www.MHProfessional.com/** and **http://www.MHProfessional.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 you leave off the "http://" from your 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.MHProfessional.com/** is the same as **http://www.MHProfessional.com/index.htm**.

Java's **URL** class has several constructors; 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*) throws MalformedURLException

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

URL(String *protocolName,* String *hostName,* int *port,* String *path*)
throws MalformedURLException

URL(String *protocolName,* String *hostName,* String *path*)
throws MalformedURLException

Another frequently used constructor allows you 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*) throws MalformedURLException

The following example creates a URL to **HerbSchildt.com**'s articles page and then examines its properties:

```
// Demonstrate URL.
import java.net.*;
class URLDemo {
  public static void main(String args[]) throws MalformedURLException {
    URL hp = new URL(http://www.HerbSchildt.com/Articles");

    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());
  }
}
```

When you run this, you will get the following output:

```
Protocol: http
Port: -1
Host:  www.HerbSchildt.com
File: /Articles
Ext:http://www.HerbSchildt.com/Articles
```

Notice that the port is −1; this means that a port was not explicitly set. Given a **URL** object, you can retrieve the data associated with it. To access the actual bits or content information of a **URL,** create a **URLConnection** object from it, using its **openConnection( )** method, like this:

```
urlc = url.openConnection()
```

openConnection( ) has the following general form:

URLConnection openConnection( ) throws IOException

It returns a **URLConnection** object associated with the invoking **URL** object. Notice that it may throw an **IOException.**

URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use **URLConnection** to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for **URL** objects that are using the HTTP protocol.

URLConnection defines several methods. Here is a sampling:

int getContentLength( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned.
long getContentLengthLong( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned. (Added by JDK 7.)
String getContentType( )	Returns the type of content found in the resource. This is the value of the **content-type** header field. Returns **null** if the content type is not available.
long getDate( )	Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.
long getExpiration( )	Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.

String getHeaderField(int *idx*)	Returns the value of the header field at index *idx*. (Header field indexes begin at 0.) Returns **null** if the value of *idx* exceeds the number of fields.
String getHeaderField(String *fieldName*)	Returns the value of header field whose name is specified by *fieldName*. Returns **null** if the specified name is not found.
String getHeaderFieldKey(int *idx*)	Returns the header field key at index *idx*. (Header field indexes begin at 0.) Returns **null** if the value of *idx* exceeds the number of fields.
Map<String, List<String>> getHeaderFields( )	Returns a map that contains all of the header fields and values.
long getLastModified( )	Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable.
InputStream getInputStream( ) throws IOException	Returns an **InputStream** that is linked to the resource. This stream can be used to obtain the content of the resource.

Notice that **URLConnection** defines several methods that handle header information. A header consists of pairs of keys and values represented as strings. By using **getHeaderField( )**, you can obtain the value associated with a header key. By calling **getHeaderFields( )**, you can obtain a map that contains all of the headers. Several standard header fields are available directly through methods such as **getDate( )** and **getContentType( )**.

The following example creates a **URLConnection** using the **openConnection( )** method of a **URL** object and then uses it to examine the document's properties and content:

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
  public static void main(String args[]) throws Exception {
    int c;
    URL hp = new URL(http://www.internic.net");
    URLConnection hpCon = hp.openConnection();

    // get date
    long d = hpCon.getDate();
    if(d==0)
      System.out.println("No date information.");
    else
      System.out.println("Date: " + new Date(d));

    // get content type
    System.out.println("Content-Type: " + hpCon.getContentType());

    // get expiration date
    d = hpCon.getExpiration();
    if(d==0)
      System.out.println("No expiration information.");
    else
      System.out.println("Expires: " + new Date(d));

    // get last-modified date
    d = hpCon.getLastModified();
    if(d==0)
      System.out.println("No last-modified information.");
    else
      System.out.println("Last-Modified: " + new Date(d));


    // get content length
    long len = hpCon.getContentLengthLong();
    if(len == -1)
      System.out.println("Content length unavailable.");
    else
      System.out.println("Content-Length: " + len);

    if(len != 0) {
      System.out.println("=== Content ===");
      InputStream input = hpCon.getInputStream();
      while (((c = input.read()) != -1)) {
        System.out.print((char) c);
      }
      input.close();
    } else {
      System.out.println("No content available.");
    }
  }
}
```

The program establishes an HTTP connection to www.internic.net over port 80. It then displays several header values and retrieves the content. Here are the first lines of the output (the precise output will vary over time).

```
Date: Mon Oct 04 15:53:24 CDT 2010
Content-Type: text/html; charset=UTF-8
```

```
No expiration information.
Last-Modified: Thu Sep 24 15:22:52 CDT 2009
Content-Length: 7316
=== Content ===
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>InterNIC | The Internet's Network Information Center</title>
.
.
.
```

HttpURLConnection

Java provides a subclass of **URLConnection** that provides support for HTTP connections. This class is called **HttpURLConnection.** You obtain an **HttpURLConnection** in the same way just shown, by calling **openConnection( )** on a **URL** object, but you must cast the result to **HttpURLConnection.** (Of course, you must make sure that you are actually opening an HTTP connection.) Once you have obtained a reference to an **HttpURLConnection** object, you can use any of the methods inherited from **URLConnection.** You can also use any of the several methods defined by **HttpURLConnection.** Here is a sampling:

static boolean getFollowRedirects( )	Returns **true** if redirects are automatically followed and **false** otherwise. This feature is on by default.
String getRequestMethod( )	Returns a string representing how URL requests are made. The default is GET. Other options, such as POST, are available.
int getResponseCode( ) throws IOException	Returns the HTTP response code. –1 is returned if no response code can be obtained. An **IOException** is thrown if the connection fails.
String getResponseMessage( ) throws IOException	Returns the response message associated with the response code. Returns **null** if no message is available. An **IOException** is thrown if the connection fails.
static void setFollowRedirects(boolean *how*)	If *how* is **true,** then redirects are automatically followed. If *how* is **false,** redirects are not automatically followed. By default, redirects are automatically followed.
void setRequestMethod(String *how*) throws ProtocolException	Sets the method by which HTTP requests are made to that specified by *how*. The default method is GET, but other options, such as POST, are available. If *how* is invalid, a **ProtocolException** is thrown.

The following program demonstrates **HttpURLConnection.** It first establishes a connection to www.google.com. Then it displays the request method, the response code, and the response message. Finally, it displays the keys and values in the response header.

```
// Demonstrate HttpURLConnection.
import java.net.*;
import java.io.*;
import java.util.*;

class HttpURLDemo
{

public static void main(String args[]) throws Exception {
  URL hp = new URL(http://www.google.com");

  HttpURLConnection hpCon = (HttpURLConnection) hp.openConnection();

  // Display request method.
  System.out.println("Request method is " +
                     hpCon.getRequestMethod());

  // Display response code.
  System.out.println("Response code is " +
                     hpCon.getResponseCode());

  // Display response message.
  System.out.println("Response Message is " +
                     hpCon.getResponseMessage());


  // Get a list of the header fields and a set
  // of the header keys.
  Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
  Set<String> hdrField = hdrMap.keySet();

  System.out.println("\nHere is the header:");

   // Display all header keys and values.
   for(String k : hdrField) {
     System.out.println("Key: " + k +
                        "  Value: " + hdrMap.get(k));
   }
  }
}
```

The output produced by the program is shown here. (Of course, the exact response returned by **www.google.com** will vary over time.)

```
Request method is GET
Response code is 200
Response Message is OK
Here is the header:
Key: null  Value: [HTTP/1.1 200 OK]
Key: Date  Value: [Mon, 04 Oct 2010 21:11:53 GMT]
Key: Transfer-Encoding  Value: [chunked]
Key: Expires  Value: [-1]
Key: X-XSS-Protection  Value: [1; mode=block]
Key: Set-Cookie  Value: [NID=39=uAS1-
DdTfLelHcxkEiRy7xNtExX3zJaKS9mjdTy8_XejjBkpjWvcqyMXgC4Ha4VT_5IZN2pnxsloo-
NlGHvcK0AIqXPhFcnCd1R1Ww4WgbiY7KrthNXCQxfXbHJwNgue; expires=Tue, 05-Apr-2011
21:11:53 GMT; path=/; domain=.google.com; HttpOnly,
PREF=ID=6644372b1f96120c:TM=1286226713:LM=1286226713:S=iNeZU0xRTrGPxg2K;
expires=Wed, 03-Oct-2012 21:11:53 GMT; path=/; domain=.google.com]
Key: Content-Type  Value: [text/html; charset=ISO-8859-1]
Key: Server  Value: [gws]
Key: Cache-Control  Value: [private, max-age=0]
```

Notice how the header keys and values are displayed. First, a map of the header keys and values is obtained by calling **getHeaderFields( )** (which is inherited from **URLConnection**). Next, a set of the header keys is retrieved by calling **keySet( )** on the map. Then the key set is cycled through by using a for-each style **for** loop. The value associated with each key is obtained by calling **get( )** on the map.

The URI Class

The **URI** class encapsulates a *Uniform Resource Identifier (URI)*. URIs are similar to URLs. In fact, URLs constitute a subset of URIs. A URI represents a standard way to identify a resource. A URL also describes how to access the resource.

Cookies

The **java.net** package includes classes and interfaces that help manage cookies and can be used to create a stateful (as opposed to stateless) HTTP session. The classes are **CookieHandler, CookieManager,** and **HttpCookie.** The interfaces are **CookiePolicy** and **CookieStore.** All but **CookieHandler** were added by Java SE 6. (**CookieHandler** was added by JDK 5.) The creation of a stateful HTTP session is beyond the scope of this book.

NOTE

For information about using cookies with servlets, see Chapter 32.

TCP/IP Server Sockets

As mentioned earlier, 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. **ServerSocket**s are quite different from normal **Socket**s. When you 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 you want to accept connections on and, optionally, how long you 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 constructors might throw an **IOException** under adverse conditions. Here are three of its constructors:

ServerSocket(int *port*) throws IOException	Creates server socket on the specified port with a queue length of 50.
ServerSocket(int *port*, int *maxQueue*) throws IOException	Creates a server socket on the specified port with a maximum queue length of *maxQueue*.
ServerSocket(int *port*, int *maxQueue*, InetAddress *localAddress*) throws IOException	Creates a server socket on the specified port with a maximum queue length of *maxQueue*. On a multihomed host, *localAddress* 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.

Datagrams

TCP/IP-style networking is appropriate for most networking needs. 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 **DatagramPacket**s. Each is examined here.

DatagramSocket

DatagramSocket defines four public constructors. They are shown here:

DatagramSocket( ) throws SocketException

DatagramSocket(int *port*) throws SocketException

DatagramSocket(int *port,* InetAddress *ipAddress*) throws SocketException

DatagramSocket(SocketAddress *address*) throws SocketException

The first creates a **DatagramSocket** bound to any unused port on the local computer. The second creates a **DatagramSocket** bound to the port specified by *port*. The third constructs a **DatagramSocket** bound to the specified port and **InetAddress**. The fourth constructs a **DatagramSocket** bound to the specified **SocketAddress**. **SocketAddress** is an abstract class that is implemented by the concrete class **InetSocketAddress**. **InetSocketAddress** encapsulates an IP address with a port number. All can throw a **SocketException** if an error occurs while creating the socket.

DatagramSocket defines many methods. Two of the most important are **send( )** and **receive( )**, which are shown here:

void send(DatagramPacket *packet*) throws IOException

void receive(DatagramPacket *packet*) throws IOException

The **send( )** method sends a packet to the port specified by *packet*. The **receive( )** method waits for a packet to be received from the port specified by *packet* and returns the result.

DatagramSocket also defines the **close**( ) method, which closes the socket. Beginning with JDK 7, **DatagramSocket** implements **AutoCloseable**, which means that a **DatagramSocket** can be managed by a **try**-with-resources block.

Other methods give you access to various attributes associated with a **DatagramSocket**. Here is a sampling:

InetAddress getInetAddress( )	If the socket is connected, then the address is returned. Otherwise, **null** is returned.
int getLocalPort( )	Returns the number of the local port.
int getPort( )	Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port.
boolean isBound( )	Returns **true** if the socket is bound to an address. Returns **false** otherwise.
boolean isConnected( )	Returns **true** if the socket is connected to a server. Returns **false** otherwise.
void setSoTimeout(int *millis*) throws SocketException	Sets the time-out period to the number of milliseconds passed in *millis*.

DatagramPacket

DatagramPacket defines several constructors. Four are shown here:

DatagramPacket(byte *data* [ ], int *size*)

DatagramPacket(byte *data* [ ], int *offset,* int *size*)

DatagramPacket(byte *data* [ ], int *size,* InetAddress *ipAddress,* int *port*)

DatagramPacket(byte *data* [ ], int *offset,* int *size,* InetAddress *ipAddress,* int *port*)

The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a **DatagramSocket.** The second form allows you to specify an offset into the buffer at which data will be stored. The third form specifies a target address and port, which are used by a **DatagramSocket** to determine where the data in the packet will be sent. The fourth form transmits packets beginning at the specified offset into the data. Think of the first two forms as building an "in box," and the second two forms as stuffing and addressing an envelope.

DatagramPacket defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length. In general, the **get** methods are used on packets that are received and the **set** methods are used on packets that will be sent.

InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the **getData( )** method. This may not equal the length of the whole byte array.
int getOffset( )	Returns the starting index of the data.
int getPort( )	Returns the port number.
void setAddress(InetAddress *ipAddress*)	Sets the address to which a packet will be sent. The address is specified by *ipAddress*.
void setData(byte[ ] *data*)	Sets the data to *data*, the offset to zero, and the length to number of bytes in *data*.
void setData(byte[ ] *data*, int *idx*, int *size*)	Sets the data to *data*, the offset to *idx*, and the length to *size*.
void setLength(int *size*)	Sets the length of the packet to *size*.
void setPort(int *port*)	Sets the port to *port*.

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

```
// Demonstrate datagrams.
import java.net.*;

class WriteServer {
  public static int serverPort = 998;
  public static int clientPort = 999;
  public static int buffer_size = 1024;
  public static DatagramSocket ds;
  public static byte buffer[] = new byte[buffer_size];

  public static void TheServer() throws Exception {
    int pos=0;
    while (true) {
    int c = System.in.read();
    switch (c) {
      case -1:
        System.out.println("Server Quits.");
        ds.close();
        return;
      case '\r':
        break;
      case '\n':
        ds.send(new DatagramPacket(buffer,pos,
           InetAddress.getLocalHost(),clientPort));
        pos=0;
        break;
      default:
        buffer[pos++] = (byte) c;
    }
  }
}


public static void TheClient() throws Exception {
  while(true) {
    DatagramPacket p = new DatagramPacket(buffer, buffer.length);
    ds.receive(p);
    System.out.println(new String(p.getData(), 0, p.getLength()));
  }
}

public static void main(String args[]) throws Exception {
  if(args.length == 1) {
    ds = new DatagramSocket(serverPort);
    TheServer();
  } else {
    ds = new DatagramSocket(clientPort);
    TheClient();
  }
}
}
```

This sample program is restricted by the **DatagramSocket** constructor to running between two ports on the local machine. To use the program, run

```
java WriteServer
```

in one window; this will be the client. Then run

```
java WriteServer 1
```

This will be the server. Anything that is typed in the server window will be sent to the

client window after a newline is received.