Table of Contents

Chapter 6. Looking Up Internet Addresses

Devices connected to the Internet are called *nodes*. Nodes that are computers are called *hosts* . Each node or host is identified by at least one unique number called an Internet address or an IP address. Most current IP addresses are four bytes long; these are referred to as IPv4 addresses. However, a small but growing number of IP addresses are 16 bytes long; these are called IPv6 addresses. (4 and 6 refer to the version of the Internet Protocol, not the number of the bytes in the address.) Both IPv4 and IPv6 addresses are ordered sequences of bytes, like an array. They aren't numbers, and they aren't ordered in any predictable or useful sense.

An IPv4 address is normally written as four unsigned bytes, each ranging from 0 to 255, with the most significant byte first. Bytes are separated by periods for the convenience of human eyes. For example, the address for *hermes.oit.unc.edu* is 152.2.21.2. This is called the *dotted quad* format.

An IPv6 address is normally written as eight blocks of four hexadecimal digits separated by colons. For example, at the time of this writing, the address of www.ipv6.com.cn is *2001:0250:02FF:0210:0250:8BFF:FEDE:67C8*. Leading zeros do not need to be written. Thus, the address of www.ipv6.com.cn can be written as *2001:250:2FF:210:250:8BFF:FEDE:67C8*. A double colon, at most one of which may appear in any address, indicates multiple zero blocks. For example, *FEDC:0000:0000:0000:00DC:0000:7076:0010* could be written more compactly as *FEDC::DC:0:7076:10*. In mixed networks of IPv6 and IPv4, the last four bytes of the IPv6 address are sometimes written as an IPv4 dotted quad address. For example, *FEDC:BA98:7654:3210:FEDC:BA98:7654:3210* could be written as *FEDC:BA98:7654:3210:FEDC:BA98:118.84.50.16*. IPv6 is only supported in Java 1.4 and later. Java 1.3 and earlier only support four byte addresses.

IP addresses are great for computers, but they are a problem for humans, who have a hard time remembering long numbers. In the 1950s, it was discovered that most people could remember about seven digits per number; some can remember as many as nine, while others remember as few as five. ("The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," by G. A. Miller, in the *Psychological Review*, Vol. 63, pp. 81-97.) This is why phone numbers are broken into three- and four-digit pieces with three-digit area codes. Obviously, an IP address, which can have as many as 12 decimal digits, is beyond the capacity of most humans to remember. I can remember about two IP addresses, and then only if I use both daily and the second is on the same subnet as the first.

To avoid the need to carry around Rolodexes full of IP addresses, the Internet's designers invented the Domain Name System (DNS). DNS associates hostnames that humans can remember (such as *hermes.oit.unc.edu*) with IP addresses that computers can remember (such as 152.2.21.2). Most hosts have at least one hostname. An exception is made for computers that don't have a permanent IP address (like many PCs); because these computers don't have a permanent address, they can't be used as servers and therefore don't need a name, since nobody will need to refer to them.

Colloquially, people often use "Internet address" to mean a hostname (or even an email address). In a book about network programming, it is crucial to be precise about addresses and hostnames. In this book, an address is always a numeric IP address, never a human-readable hostname.

Some machines have multiple names. For instance, www.ibiblio.org and helios.metalab.unc.edu are really the same Linux box in Chapel Hill. The name www.ibiblio.org really refers to a web site rather than a particular machine. In the past, when this web site moved from one machine to another, the name was reassigned to the new machine so it always pointed to the site's current server. This way, URLs around the Web don't need to be updated just because the site has moved to a new host. Some common names like *www* and *news* are often aliases for the machines providing those services. For example, news.speakeasy.net is an alias for my ISP's news server. Since the server may change over time, the alias can move with the service.

On occasion, one name maps to multiple IP addresses. It is then the responsibility of the DNS server to randomly choose machines to respond to each request. This feature is most frequently used for very high traffic web sites, where it splits the load across multiple systems. For instance, www.oreilly.com is actually two machines, one at 208.201.239.36 and one at 208.201.239.37.

Every computer connected to the Internet should have access to a machine called a *domain name server*, generally a Unix box running special DNS software that knows the mappings between different hostnames and IP addresses. Most domain name servers only know the addresses of the hosts on their local network, plus the addresses of a few domain name servers at other sites. If a client asks for the address of a machine outside the local domain, the local domain name server asks a domain name server at the remote location and relays the answer to the requester.

Most of the time, you can use hostnames and let DNS handle the translation to IP addresses. As long as you can connect to a domain name server, you don't need to worry about the details of how names and addresses are passed between your machine, the local domain name server, and the rest of the Internet. However, you will need access to at least one domain name server to use the examples in this chapter and most of the rest of this book. These programs will not work on a standalone computer. Your machine must be connected to the Internet.

6.1. The InetAddress Class

The `java.net.InetAddress` class is Java's high-level representation of an IP address, both IPv4 and IPv6. It is used by most of the other networking classes, including `Socket`, `ServerSocket`, `URL`, `DatagramSocket`, `DatagramPacket`, and more. Generally, it includes both a hostname and an IP address.

```
public class InetAddress extends Object implements Serializable
```

In Java 1.3 and earlier, this class is final. In Java 1.4, it has two subclasses. However, you should not subclass it yourself. Indeed, you can't, because all constructors are package protected.

6.1.1. Creating New InetAddress Objects

There are no public constructors in the `InetAddress` class. However, `InetAddress` has three static methods that return suitably initialized `InetAddress` objects given a little information. They are:

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

All three of these methods may make a connection to the local DNS server to fill out the information in the `InetAddress` object, if necessary. This has a number of possibly unexpected implications, among them that these methods may throw security exceptions if the connection to the DNS server is prohibited. Furthermore, invoking one of these methods may cause a host that uses a PPP connection to dial into its provider if it isn't already connected. The key thing to remember is that these methods do not simply use their arguments to set the internal fields. They actually make network connections to retrieve all the information they need. The other methods in this class, such as `getAddress( )` and `getHostName( )`, mostly work with the information provided by one of these three methods. They do not make network connections; on the rare occasions that they do, they do not throw any exceptions. Only these three methods have to go outside Java and the local system to get their work done.

Since DNS lookups can be relatively expensive (on the order of several seconds for a request that has to go through several intermediate servers, or one that's trying to resolve an unreachable host) the `InetAddress` class caches the results of lookups. Once it has the address of a given host, it won't look it up again, even if you create a new `InetAddress` object for the same host. As long as IP addresses don't change while your program is running, this is not a problem.

Negative results (host not found errors) are slightly more problematic. It's not uncommon for an initial attempt to resolve a host to fail, but the immediately following one to succeed. What has normally happened in this situation is that the first attempt timed out while the information was still in transit from the remote DNS server. Then the address arrived at the local server and was immediately available for the next request. For this reason, Java only caches unsuccessful DNS queries for 10 seconds.

In Java 1.4 and later, these times can be controlled by the `networkaddress.cache.ttl` and `networkaddress.cache.negative.ttl` system properties. `networkaddress.cache.ttl` specifies the number of seconds a successful DNS lookup will remain in Java's cache. `networkaddress.cache.negative.ttl` is the number of seconds an unsuccessful lookup will be cached. Attempting to look up the same host again within these limits will only return the same value. -1 is interpreted as "never expire".

Besides locale caching inside the `InetAddress` class, the local host, the local domain name server, and other DNS servers elsewhere on the Internet may also cache the results of various queries. Java provides no way to control this. As a result, it may take several hours for the information about an IP address change to propagate across the Internet. In the meantime, your program may encounter various exceptions, including `UnknownHostException`, `NoRouteToHostException`, and `ConnectException`, depending on the changes made to the DNS.

Java 1.4 adds two more factory methods that do not check their addresses with the local DNS server. The first creates an `InetAddress` object with an IP address and no hostname. The second creates an `InetAddress` object with an IP address and a hostname.

```
public static InetAddress getByAddress(byte[] address)
 throws UnknownHostException  // 1.4
public static InetAddress getByAddress(String hostName, byte[] address)
 throws UnknownHostException  // 1.4
```

Unlike the other three factory methods, these two methods make no guarantees that such a host exists or that the hostname is correctly mapped to the IP address. They throw an `UnknownHostException` only if a byte array of an illegal size (neither 4 nor 16 bytes long) is passed as the `address` argument.

6.1.1.1. public static InetAddress getByName(String hostName) throws UnknownHostException

`InetAddress.getByName( )` is the most frequently used of these factory methods. It is a static method that takes the hostname you're looking for as its argument. It looks up the host's IP address using DNS. Call `getByName( )` like this:

```
java.net.InetAddress address =
 java.net.InetAddress.getByName("www.oreilly.com");
```

If you have already imported the `java.net.InetAddress` class, which will almost always be the case, you can call `getByName( )` like this:

```
InetAddress address = InetAddress.getByName("www.oreilly.com");
```

In the rest of this book, I assume that there is an `import java.net.*;` statement at the top of the program containing each code fragment, as well as any other necessary `import` statements.

The `InetAddress.getByName( )` method throws an `UnknownHostException` if the host can't be found, so you need to declare that the method making the call throws `UnknownHostException` (or its superclass, `IOException`) or wrap it in a `try` block, like this:

```
try {
  InetAddress address = InetAddress.getByName("www.oreilly.com");
  System.out.println(address);
}
catch (UnknownHostException ex) {
  System.out.println("Could not find www.oreilly.com");
}
```

Example 6-1 shows a complete program that creates an `InetAddress` object for www.oreilly.com and prints it out.

Example 6-1. A program that prints the address of www.oreilly.com

```
import java.net.*;

public class OReillyByName {

  public static void main (String[] args) {

    try {
      InetAddress address = InetAddress.getByName("www.oreilly.com");
      System.out.println(address);
    }
    catch (UnknownHostException ex) {
      System.out.println("Could not find www.oreilly.com");
    }

  }

}
```

Here's the result:

```
% java OReillyByName
www.oreilly.com/208.201.239.36
```

On rare occasions, you will need to connect to a machine that does not have a hostname. In this case, you can pass a `String` containing the dotted quad or hexadecimal form of the IP address to `InetAddress.getByName( )`:

```
InetAddress address = InetAddress.getByName("208.201.239.37");
```

Example 6-2 uses the IP address for www.oreilly.com instead of the name.

Example 6-2. A program that prints the address of 208.201.239.37

```
import java.net.*;

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

    try {
      InetAddress address = InetAddress.getByName("208.201.239.37");
      System.out.println(address);
    }
    catch (UnknownHostException ex) {
      System.out.println("Could not find 208.201.239.37");
    }

  }

}
```

Here's the result in Java 1.3 and earlier:

```
% java OReillyByAddress
www.oreilly.com/208.201.239.37
```

When you call `getByName( )` with an IP address string as an argument, it creates an `InetAddress` object for the requested IP address without checking with DNS. This means it's possible to create `InetAddress` objects for hosts that don't really exist and that you can't connect to. The hostname of an `InetAddress` object created from a string containing an IP address is initially set to that string. A DNS lookup for the actual hostname is performed only when the hostname is requested, either explicitly via `getHostName( )` or implicitly through `toString( )`. That's how www.oreilly.com was determined from the dotted quad address 208.201.239.37. If at the time the hostname is requested and a DNS lookup is finally performed the host with the specified IP address can't be found, then the hostname remains the original dotted quad string. However, no `UnknownHostException` is thrown.

The `toString( )` method in Java 1.4 behaves a little differently than in earlier versions. It does not do a reverse name lookup; thus, the host is not printed unless it is already known, either because it was provided as an argument to the factory method or because `getHostName( )` was invoked. In Java 1.4, Example 6-2 produces this output:

```
/208.201.239.37
```

Hostnames are much more stable than IP addresses. Some services have lived at the same hostname for years but have switched IP addresses several times. If you have a choice between using a hostname like www.oreilly.com or an IP address like 208.201.239.37, always choose the hostname. Use an IP address only when a hostname is not available.

6.1.1.2. public static InetAddress[ ] getAllByName(String hostName) throws UnknownHostException

Some computers have more than one Internet address. Given a hostname, `InetAddress.getAllByName()` returns an array that contains all the addresses corresponding to that name. Its use is straightforward:

```
InetAddress[] addresses = InetAddress.getAllByName("www.apple.com");
```

Like `InetAddress.getByName( )`, `InetAddress.getAllByName( )` can throw an `UnknownHostException`, so you need to enclose it in a `try` block or declare that your method throws `UnknownHostException`. Example 6-3 demonstrates by returning a complete list of the IP addresses for www.microsoft.com.

Example 6-3. A program that prints all the addresses of www.microsoft.com

```
import java.net.*;

public class AllAddressesOfMicrosoft {

  public static void main (String[] args) {

    try {
      InetAddress[] addresses =
       InetAddress.getAllByName("www.microsoft.com");
      for (int i = 0; i < addresses.length; i++) {
        System.out.println(addresses[i]);
      }
    }
    catch (UnknownHostException ex) {
      System.out.println("Could not find www.microsoft.com");
    }

  }

}
```

Here's the result:

```
% java AllAddressesOfMicrosoft
www.microsoft.com/63.211.66.123
www.microsoft.com/63.211.66.124
www.microsoft.com/63.211.66.131
www.microsoft.com/63.211.66.117
www.microsoft.com/63.211.66.116
www.microsoft.com/63.211.66.107
www.microsoft.com/63.211.66.118
www.microsoft.com/63.211.66.115
www.microsoft.com/63.211.66.110
```

www.microsoft.com appears to have nine IP addresses. Hosts with more than one address are the exception rather than the rule. Most hosts with multiple IP addresses are very high-volume web servers. Even in those cases, you rarely need to know more than one address.

6.1.1.3. public static InetAddress getByAddress(byte[ ] address) throws UnknownHostException // Java 1.4public static InetAddress getByAddress(String hostName, byte[] address) throws UnknownHostException // Java 1.4

In Java 1.4 and later, you can pass a byte array and optionally a hostname to `getByAddress()` to create an `InetAddress` object with exactly those bytes. Domain name lookup is not performed. However, if byte array is some length other than 4 or 16 bytes—that is, if it can't be an IPv4 or IPv6 address—an `UnknownHostException` is thrown.

This is useful if a domain name server is not available or might have inaccurate information. For example, none of the computers, printers, or routers in my basement area network are registered with any DNS server. Since I can never remember which addresses I've assigned to which systems, I wrote a simple program that attempts to connect to all 254 possible local addresses in turn to see which ones are active. (This only took me about 10 times as long as writing down all the addresses on a piece of paper.)

`getByAddress(byte[] address)` really doesn't do anything `getByAddress(String address)` doesn't do. In a few cases, it might be marginally faster because it doesn't have to convert a string to a byte array, but that's a trivial improvement. `getByAddress(String hostName, byte[] address)` does let you create `InetAddress` objects that don't match or even actively conflict with the information in the local DNS. There might occasionally be a call for this, but the use case is pretty obscure.

6.1.1.4. public static InetAddress getLocalHost( ) throws UnknownHostException

The `InetAddress` class contains one final means of getting an `InetAddress` object. The static method `InetAddress.getLocalHost()` returns the `InetAddress` of the machine on which it's running. Like `InetAddress.getByName( )` and `InetAddress.getAllByName( )`, it throws an `UnknownHostException` when it can't find the address of the local machine (though this really shouldn't happen). Its use is straightforward:

```
    InetAddress me = InetAddress.getLocalHost( );
```

Example 6-4 prints the address of the machine it's run on.

Example 6-4. Find the address of the local machine

```
import java.net.*;

public class MyAddress {

  public static void main (String[] args) {

    try {
      InetAddress address = InetAddress.getLocalHost( );
      System.out.println(address);
    }
    catch (UnknownHostException ex) {
      System.out.println("Could not find this computer's address.");
    }

  }

}
```

Here's the output; I ran the program on *titan.oit.unc.edu*:

```
% java MyAddress
titan.oit.unc.edu/152.2.22.14
```

Whether you see a fully qualified name like *titan.oit.unc.edu* or a partial name like *titan* depends on what the local DNS server returns for hosts in the local domain. If you're not connected to the Internet, and the system does not have a fixed IP address or domain name, you'll probably see *localhost* as the domain name and 127.0.0.1 as the IP address.

6.1.2. Security Issues

Creating a new `InetAddress` object from a hostname is considered a potentially insecure operation because it requires a DNS lookup. An untrusted applet under the control of the default security manager will only be allowed to get the IP address of the host it came from (its *codebase*) and possibly the local host. Untrusted code is not allowed to create an `InetAddress` object from any other hostname. This is true whether the code uses the `InetAddress.getByName( )` method, the `InetAddress.getAllByName()` method, the `InetAddress.getLocalHost( )` method, or something else. Untrusted code can construct an `InetAddress` object from the string form of the IP address, though it will not perform DNS lookups for such addresses.

Untrusted code is not allowed to perform arbitrary DNS lookups for third-party hosts because of the prohibition against making network connections to hosts other than the codebase.

Arbitrary DNS lookups would open a covert channel by which a program could talk to third-party hosts. For instance, suppose an applet downloaded from *www.bigisp.com* wants to send the message "macfaq.dialup.cloud9.net is vulnerable" to *crackersinc.com*. All it has to do is request DNS information for *macfaq.dialup.cloud9.net.is.vulnerable.crackersinc.com*. To resolve that hostname, the applet would contact the local DNS server. The local DNS server would contact the DNS server at *crackersinc.com*. Even though these hosts don't exist, the cracker can inspect the DNS error log for *crackersinc.com* to retrieve the message. This scheme could be considerably more sophisticated with compression, error correction, encryption, custom DNS servers that email the messages to a fourth site, and more, but this version is good enough for a proof of concept. Arbitrary DNS lookups are prohibited because arbitrary DNS lookups leak information.

Untrusted code is allowed to call `InetAddress.getLocalHost()`. However, this method returns a hostname of *localhost* and an IP address of 127.0.0.1. This is a special hostname and IP address called the *loopback address*. No matter which machine you use this hostname or IP address on, it always refers to the current machine. No specific DNS resolution is necessary. The reason for prohibiting the applet from finding out the true hostname and address is that the computer on which the applet is running may be deliberately hidden behind a firewall. In this case, an applet should not be a channel for information the web server doesn't already have. (Some older browsers, including Netscape 4.x, do allow a little more information about the local host to leak out, including its IP address, but only if no DNS lookup is required to get this information.)

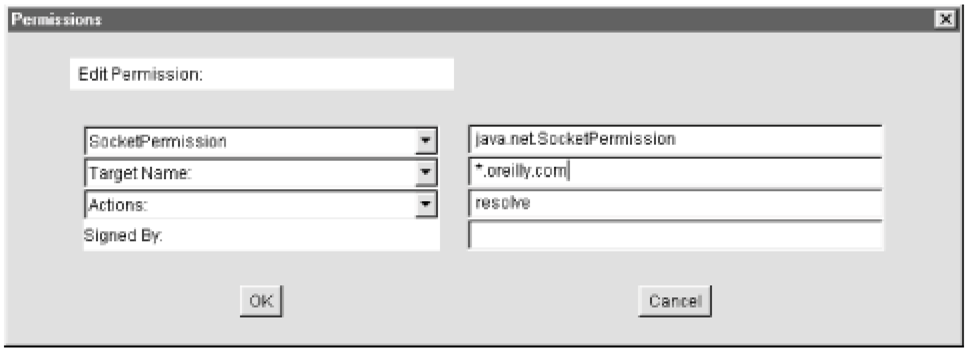
Like all security checks, prohibitions against DNS resolutions can be relaxed for trusted code. The specific `SecurityManager` method used to test whether a host can be resolved is `checkConnect( )`:

```
    public void checkConnect(String hostname, int port)
```

When the `port` argument is -1, this method checks whether DNS may be invoked to resolve the specified `host`. (If the `port` argument is greater than -1, this method checks whether a connection to the named host on the specified port is allowed.) The `host` argument may be either a hostname like www.oreilly.com, a dotted quad IP address like 208.201.239.37, or, in Java 1.4 and later, a hexadecimal IPv6 address like *FEDC::DC:0:7076:10*.

You can grant an applet permission to resolve a host by using the Policy Tool to add a `java.net.SocketPermission` with the action connect and the target being the name of the host you want to allow the applet to resolve. You can use the asterisk wildcard (*) to allow all hosts in particular domains to be resolved. For example, setting the target to **.oreilly.com* allows the applet to resolve the hosts www.oreilly.com, java.oreilly.com, perl.oreilly.com, and all others in the *oreilly.com* domain. Although you'll generally use a hostname to set permissions, Java checks it against the actual IP addresses. In this example, that also allows hosts in the *ora.com* domain to be resolved because this is simply an alias for *oreilly.com* with the same range of IP addresses. To allow all hosts in all domains to be resolved, just set the target to *. Figure 6-1 demonstrates.

Figure 6-1. Using the Policy Tool to grant DNS resolution permission to all applets



SocketPermission: java.net.SocketPermission
Target Name: *.oreilly.com
Actions: resolve
Signed By:

6.1.3. Getter Methods

The `InetAddress` class contains three getter methods that return the hostname as a string and the IP address as both a string and a byte array:

```
public String getHostName( )
public byte[] getAddress( )
public String getHostAddress( )
```

There are no corresponding `setHostName( )` and `setAddress( )` methods, which means that packages outside of `java.net` can't change an `InetAddress` object's fields behind its back. Therefore, Java can guarantee that the hostname and the IP address match each other. This has the beneficial side effect of making `InetAdddress` immutable and thus thread-safe.

6.1.3.1. public String getHostName( )

The `getHostName( )` method returns a `String` that contains the name of the host with the IP address represented by this `InetAddress` object. If the machine in question doesn't have a hostname or if the security manager prevents the name from being determined, a dotted quad format of the numeric IP address is returned. For example:

```
InetAddress machine = InetAddress.getLocalHost( );
String localhost = machine.getHostName( );
```

In some cases, you may only see a partially qualified name like *titan* instead of the full name like *titan.oit.unc.edu*. The details depend on how the local DNS behaves when resolving local hostnames.

The `getHostName( )` method is particularly useful when you're starting with a dotted quad IP address rather than the hostname. Example 6-5 converts the dotted quad address 208.201.239.37 into a hostname by using `InetAddress.getByName( )` and then applying `getHostName( )` on the resulting object.

Example 6-5. Given the address, find the hostname

```
import java.net.*;

public class ReverseTest {

  public static void main (String[] args) {

    try {
      InetAddress ia = InetAddress.getByName("208.201.239.37");
      System.out.println(ia.getHostName( ));
    }
    catch (Exception ex) {
      System.err.println(ex);
    }

  }

}
```

Here's the result:

```
% java ReverseTest
www.oreillynet.com
```

6.1.3.2. public String getHostAddress( )

The `getHostAddress()` method returns a string containing the dotted quad format of the IP address. Example 6-6 uses this method to print the IP address of the local machine in the customary format.

Example 6-6. Find the IP address of the local machine

```
import java.net.*;


public class MyAddress {

  public static void main(String[] args) {

    try {
      InetAddress me = InetAddress.getLocalHost( );
      String dottedQuad = me.getHostAddress( );
      System.out.println("My address is " + dottedQuad);
    }
    catch (UnknownHostException ex) {
      System.out.println("I'm sorry. I don't know my own address.");
    }

  }

}
```

Here's the result:

```
% java MyAddress
My address is 152.2.22.14.
```

Of course, the exact output depends on where the program is run.

6.1.3.3. public byte[] getAddress( )

If you want to know the IP address of a machine (and you rarely do), `getAddress( )` returns an IP address as an array of bytes in network byte order. The most significant byte (i.e., the first byte in the address's dotted quad form) is the first byte in the array, or element zero—remember, Java array indices start with zero. To be ready for IPv6 addresses, try not to assume anything about the length of this array. If you need to know the length of the array, use the array's `length` field:

```
InetAddress me = InetAddress.getLocalHost( );
byte[] address = me.getAddress( ));
```

The bytes returned are unsigned, which poses a problem. Unlike C, Java doesn't have an unsigned byte primitive data type. Bytes with values higher than 127 are treated as negative numbers. Therefore, if you want to do anything with the bytes returned by `getAddress ( )`, you need to promote the bytes to `int`s and make appropriate adjustments. Here's one way to do it:

```
int unsignedByte = signedByte < 0 ? signedByte + 256 : signedByte;
```

Here, `signedByte` may be either positive or negative. The conditional operator `?` tests whether `signedByte` is negative. If it is, 256 is added to `signedByte` to make it positive. Otherwise, it's left alone. `signedByte` is automatically promoted to an `int` before the addition is performed so wraparound is not a problem.

One reason to look at the raw bytes of an IP address is to determine the type of the address. Test the number of bytes in the array returned by `getAddress( )` to determine whether you're dealing with an IPv4 or IPv6 address. Example 6-7 demonstrates.

Example 6-7. Print the IP address of the local machine

```
import java.net.*;

public class AddressTests {

  public static int getVersion(InetAddress ia) {

    byte[] address = ia.getAddress( );
    if (address.length == 4) return 4;
    else if (address.length == 16) return 6;
    else return -1;

  }

}
```

6.1.4. Address Types

Some IP addresses and some patterns of addresses have special meanings. For instance, I've already mentioned that 127.0.0.1 is the local loopback address. IPv4 addresses in the range 224.0.0.0 to 239.255.255.255 are multicast addresses that send to several subscribed hosts at once. Java 1.4 and later include 10 methods for testing whether an `InetAddress` object meets any of these criteria:

```
public boolean isAnyLocalAddress( )
public boolean isLoopbackAddress( )
public boolean isLinkLocalAddress( )
public boolean isSiteLocalAddress( )
public boolean isMulticastAddress( )
public boolean isMCGlobal( )
public boolean isMCNodeLocal( )
public boolean isMCLinkLocal( )
public boolean isMCSiteLocal( )
public boolean isMCOrgLocal( )
```

6.1.4.1. public boolean isAnyLocalAddress( )

This method returns true if the address is a *wildcard address*, false otherwise. A wildcard address matches any address of the local system. This is important if the system has multiple network interfaces, e.g. several Ethernet cards or an Ethernet card and a wireless connection. This is normally important only on servers and gateways. In IPv4, the wildcard address is 0.0.0.0. In IPv6 this address is 0:0:0:0:0:0:0:0 (a.k.a ::).

6.1.4.2. public boolean isLoopbackAddress( )

This method returns true if the address is the loopback address, false otherwise. The loopback address connects to the same computer directly in the IP layer without using any physical hardware. Thus, connecting to the loopback address enables tests to bypass potentially buggy or nonexistent Ethernet, PPP, and other drivers, helping to isolate problems. Connecting to the loopback address is not the same as connecting to the system's normal IP address from the same system. In IPv4, this address is 127.0.0.1. In IPv6, this address is 0:0:0:0:0:0:0:1 (a.k.a. ::1).

6.1.4.3. public boolean isLinkLocalAddress( )

This method returns true if the address is an IPv6 link-local address, false otherwise. This is an address used to help IPv6 networks self-configure, much like DHCP on IPv4 networks but without necessarily using a server. Routers do not forward these packets beyond the local subnet. All link-local addresses begin with the eight bytes FE80:0000.0000:0000. The next eight bytes are filled with a local address, often copied from the Ethernet MAC address assigned by the Ethernet card manufacturer.

6.1.4.4. public boolean isSiteLocalAddress( )

This method returns true if the address is an IPv6 site-local address, false otherwise. Site-local addresses are similar to link-local addresses except that they may be forwarded by routers within a site or campus but should not be forwarded beyond that site. Site-local addresses begin with the eight bytes FEC0:0000.0000:0000. The next eight bytes are filled with a local address, often copied from the Ethernet MAC address assigned by the Ethernet card manufacturer.

6.1.4.5. public boolean isMulticastAddress( )

This method returns true if the address is a multicast address, false otherwise. Multicasting broadcasts content to all subscribed computers rather than to one particular computer. In IPv4, multicast addresses all fall in the range 224.0.0.0 to 239.255.255.255. In IPv6, they all begin with byte FF. Multicasting will be discussed in Chapter 14.

6.1.4.6. public boolean isMCGlobal( )

This method returns true if the address is a global multicast address, false otherwise. A global multicast address may have subscribers around the world. All multicast addresses begin with FF. In IPv6, global multicast addresses begin with FF0E or FF1E depending on whether the multicast address is a well known permanently assigned address or a transient address. In IPv4, all multicast addresses have global scope, at least as far as this method is concerned. As you'll see in Chapter 14, IPv4 uses time-to-live (TTL) values to control scope rather than addressing.

6.1.4.7. public boolean isMCOrgLocal( )

This method returns true if the address is an organization-wide multicast address, false otherwise. An organization-wide multicast address may have subscribers within all the sites of a company or organization, but not outside that organization. Organization multicast addresses begin with FF08 or FF18, depending on whether the multicast address is a well known permanently assigned address or a transient address.

6.1.4.8. public boolean isMCSiteLocal( )

This method returns true if the address is a site-wide multicast address, false otherwise. Packets addressed to a site-wide address will only be transmitted within their local site. Organization multicast addresses begin with FF05 or FF15, depending on whether the multicast address is a well known permanently assigned address or a transient address.

6.1.4.9. public boolean isMCLinkLocal( )

This method returns true if the address is a subnet-wide multicast address, false otherwise. Packets addressed to a link-local address will only be transmitted within their own subnet.

Link-local multicast addresses begin with FF02 or FF12, depending on whether the multicast address is a well known permanently assigned address or a transient address.

6.1.4.10. public boolean isMCNodeLocal( )

This method returns true if the address is an interface-local multicast address, false otherwise. Packets addressed to an interface-local address are not sent beyond the network interface from which they originate, not even to a different network interface on the same node. This is primarily useful for network debugging and testing. Interface-local multicast addresses begin with the two bytes FF01 or FF11, depending on whether the multicast address is a well known permanently assigned address or a transient address.

The method name is out of sync with current terminology. Earlier drafts of the IPv6 protocol called this type of address "node-local", hence the name "isMCNodeLocal". The IPNG working group actually changed the name before Java 1.4 was released. Unfortunately, Java 1.4 uses the old terminology.

Example 6-8 is a simple program to test the nature of an address entered from the command line using these 10 methods.

Example 6-8. Testing the characteristics of an IP address (Java 1.4 only)

```
import java.net.*;

public class IPCharacteristics {

  public static void main(String[] args) {

    try {
      InetAddress address = InetAddress.getByName(args[0]);

      if (address.isAnyLocalAddress( )) {
        System.out.println(address + " is a wildcard address.");
      }
      if (address.isLoopbackAddress( )) {
        System.out.println(address + " is loopback address.");
      }

      if (address.isLinkLocalAddress( )) {
        System.out.println(address + " is a link-local address.");
      }
      else if (address.isSiteLocalAddress( )) {
        System.out.println(address + " is a site-local address.");
      }
      else {
        System.out.println(address + " is a global address.");
      }

      if (address.isMulticastAddress( )) {
        if (address.isMCGlobal( )) {
          System.out.println(address + " is a global multicast address.");
        }
        else if (address.isMCOrgLocal( )) {
          System.out.println(address
           + " is an organization wide multicast address.");
        }
        else if (address.isMCSiteLocal( )) {
          System.out.println(address + " is a site wide multicast
                             address.");
        }
        else if (address.isMCLinkLocal( )) {
          System.out.println(address + " is a subnet wide multicast
                             address.");
        }
        else if (address.isMCNodeLocal( )) {
          System.out.println(address
           + " is an interface-local multicast address.");
        }
        else {
          System.out.println(address + " is an unknown multicast
                             address type.");
        }

      }
      else {
        System.out.println(address + " is a unicast address.");
      }

    }
    catch (UnknownHostException ex) {
      System.err.println("Could not resolve " + args[0]);
    }

  }

}
```

Here's the output from an IPv4 and IPv6 address:

```
$ java  IPCharacteristics 127.0.0.1
/127.0.0.1 is loopback address.
/127.0.0.1 is a global address.
/127.0.0.1 is a unicast address.
$ java  IPCharacteristics 192.168.254.32
/192.168.254.32 is a site-local address.
```

```
/192.168.254.32 is a unicast address.
$ java  IPCharacteristics www.oreilly.com
www.oreilly.com/208.201.239.37 is a global address.
www.oreilly.com/208.201.239.37 is a unicast address.
$ java  IPCharacteristics 224.0.2.1
/224.0.2.1 is a global address.
/224.0.2.1 is a global multicast address.
$ java  IPCharacteristics FF01:0:0:0:0:0:0:1
/ff01:0:0:0:0:0:0:1 is a global address.
/ff01:0:0:0:0:0:0:1 is an interface-local multicast address.
$ java  IPCharacteristics FF05:0:0:0:0:0:0:101
/ff05:0:0:0:0:0:0:101 is a global address.
/ff05:0:0:0:0:0:0:101 is a site wide multicast address.
$ java  IPCharacteristics 0::1
/0:0:0:0:0:0:0:1 is loopback address.
/0:0:0:0:0:0:0:1 is a global address.
/0:0:0:0:0:0:0:1 is a unicast address.
```

6.1.5. Testing Reachability // Java 1.5

Java 1.5 adds two new methods to the `InetAddress` class that enable applications to test whether a particular node is reachable from the current host; that is, whether a network connection can be made. Connections can be blocked for many reasons, including firewalls, proxy servers, misbehaving routers, and broken cables, or simply because the remote host is not turned on when you try to connect. The `isReachable( )` methods allow you to test the connection:

```
public boolean isReachable(int timeout) throws IOException
public boolean isReachable(NetworkInterface interface, int ttl, int timeout)
  throws IOException
```

These methods attempt to connect to the echo port on the remote host site to find out if it's reachable. If the host responds within `timeout` milliseconds, the methods return true; otherwise, they return false. An `IOException` will be thrown if there's a network error. The second variant also lets you specify the local network interface the connection is made from and the "time-to-live" (the maximum number of network hops the connection will attempt before being discarded).

In practice, these methods aren't very reliable across the global Internet. Firewalls tend to get in the way of the network protocols Java uses to figure out if a host is reachable or not. However, you may be able to use these methods on the local intranet.

6.1.6. Object Methods

Like every other class, `java.net.InetAddress` inherits from `java.lang.Object`. Thus, it has access to all the methods of that class. It overrides three methods to provide more specialized behavior:

```
public boolean equals(Object o)
public int hashCode( )
public String toString( )
```

6.1.6.1. public boolean equals(Object o)

An object is equal to an `InetAddress` object only if it is itself an instance of the `InetAddress` class and it has the same IP address. It does not need to have the same hostname. Thus, an `InetAddress` object for *www.ibiblio.org* is equal to an `InetAddress` object for *www.cafeaulait.org* since both names refer to the same IP address. Example 6-9 creates `InetAddress` objects for *www.ibiblio.org* and *helios.metalab.unc.edu* and then tells you whether they're the same machine.

Example 6-9. Are www.ibiblio.org and helios.metalab.unc.edu the same?

```
import java.net.*;

public class IBiblioAliases {

  public static void main (String args[]) {

    try {
      InetAddress ibiblio = InetAddress.getByName("www.ibiblio.org");
      InetAddress helios = InetAddress.getByName("helios.metalab.unc.edu");
      if (ibiblio.equals(helios)) {
        System.out.println
         ("www.ibiblio.org is the same as helios.metalab.unc.edu");
      }
      else {
        System.out.println
         ("www.ibiblio.org is not the same as helios.metalab.unc.edu");
      }
    }
    catch (UnknownHostException ex) {
      System.out.println("Host lookup failed.");
    }

  }

}
```

When you run this program, you discover:

```
% java IBiblioAliases
www.ibiblio.org is the same as helios.metalab.unc.edu
```

6.1.6.2. public int hashCode( )

The `hashCode( )` method returns an `int` that is needed when `InetAddress` objects are used as keys in hash tables. This is called by the various methods of `java.util.Hashtable`. You will almost certainly not need to call this method directly.

Consistent with the `equals( )` method, the `int` that `hashCode( )` returns is calculated solely from the IP address. It does not take the hostname into account. If two `InetAddress` objects have the same address, then they have the same hash code, even if their hostnames are different. Therefore, if you try to store two objects in a `Hashtable` using equivalent `InetAddress` objects as a key (for example, the `InetAddress` objects for *helios.metalab.unc.edu* and *www.ibiblio.org*), the second will overwrite the first. If this is a problem, use the `String` returned by `getHostName( )` as the key instead of the `InetAddress` itself.

6.1.6.3. public String toString( )

Like all good classes, `java.net.InetAddress` has a `toString( )` method that returns a short text representation of the object. Example 6-1 through Example 6-4 all implicitly called this method when passing `InetAddress` objects to `System.out.println( )`. As you saw, the string produced by `toString( )` has the form:

```
hostname/dotted quad address
```

Not all `InetAddress` objects have hostnames. If one doesn't, the dotted quad address is substituted in Java 1.3 and earlier. In Java 1.4, the hostname is set to the empty string. This format isn't particularly useful, so you'll probably never call `toString()` explicitly. If you do, the syntax is simple:

```
InetAddress thisComputer = InetAddress.getLocalHost( );
String address = thisComputer.toString( );
```

6.2. Inet4Address and Inet6Address

Java 1.4 introduces two new classes, `Inet4Address` and `Inet6Address`, in order to distinguish IPv4 addresses from IPv6 addresses:

```
public final class Inet4Address extends InetAddress
public final class Inet6Address extends InetAddress
```

(In Java 1.3 and earlier, all `InetAddress` objects represent IPv4 addresses.)

Most of the time, you really shouldn't be concerned with whether an address is an IPv4 or IPv6 address. In the application layer where Java programs reside, you simply don't need to know this (and even if you do need to know, it's quicker to check the size of the byte array returned by `getAddress( )` than to use `instanceof` to test which subclass you have). Mostly these two classes are just implementation details you do not need to concern yourself with. `Inet4Address` overrides several of the methods in `InetAddress` but doesn't change their behavior in any public way. `Inet6Address` is similar, but it does add one new method not present in the superclass, `isIPv4CompatibleAddress( )`:

```
public boolean isIPv4CompatibleAddress( )
```

This method returns true if and only if the address is essentially an IPv4 address stuffed into an IPv6 container—which means only the last four bytes are non-zero. That is, the address has the form *0:0:0:0:0:0:0:xxxx*. If this is the case, you can pull off the last four bytes from the array returned by `getBytes( )` and use this data to create an `Inet4Address` instead. However, you rarely need to do this.

6.3. The NetworkInterface Class

Java 1.4 adds a `NetworkInterface` class that represents a local IP address. This can either be a physical interface such as an additional Ethernet card (common on firewalls and routers) or it can be a virtual interface bound to the same physical hardware as the machine's other IP addresses. The `NetworkInterface` class provides methods to enumerate all the local addresses, regardless of interface, and to create `InetAddress` objects from them. These `InetAddress` objects can then be used to create sockets, server sockets, and so forth.

6.3.1. Factory Methods

Since NetworkInterface objects represent physical hardware and virtual addresses, they cannot be constructed arbitrarily. As with the InetAddress class, there are static factory methods that return the NetworkInterface object associated with a particular network interface. You can ask for a NetworkInterface by IP address, by name, or by enumeration.

6.3.1.1. public static NetworkInterface getByName(String name) throws SocketException

The getByName( ) method returns a NetworkInterface object representing the network interface with the particular name. If there's no interface with that name, it returns null. If the underlying network stack encounters a problem while locating the relevant network interface, a SocketException is thrown, but this isn't too likely to happen.

The format of the names is platform-dependent. On a typical Unix system, the Ethernet interface names have the form eth0, eth1, and so forth. The local loopback address is probably named something like "lo". On Windows, the names are strings like "CE31" and "ELX100" that are derived from the name of the vendor and model of hardware on that particular network interface. For example, this code fragment attempts to find the primary Ethernet interface on a Unix system:

```
try {
  NetworkInterface ni = NetworkInterface.getByName("eth0");
  if (ni == null) {
    System.err.println("No such interface:  eth0" );
  }
}
catch (SocketException ex) {
  System.err.println("Could not list sockets." );
}
```

6.3.1.2. public static NetworkInterface getByInetAddress(InetAddress address) throws SocketException

The getByInetAddress() method returns a NetworkInterface object representing the network interface bound to the specified IP address. If no network interface is bound to that IP address on the local host, then it returns null. If anything goes wrong, it throws a SocketException. For example, this code fragment finds the network interface for the local loopback address:

```
try {
  InetAddress local = InetAddress.getByName("127.0.0.1");
```

```
      NetworkInterface ni = NetworkInterface.getByName(local);
      if (ni == null) {
        System.err.println("That's weird. No local loopback address.");
      }
    }
    catch (SocketException ex) {
      System.err.println("Could not list sockets." );
    }
    catch (UnknownHostException ex) {
      System.err.println("That's weird. No local loopback address.");
    }
```

6.3.1.3. public static Enumeration getNetworkInterfaces( ) throws SocketException

The `getNetworkInterfaces()` method returns a `java.util.Enumeration` listing all the network interfaces on the local host. Example 6-10 is a simple program to list all network interfaces on the local host:

Example 6-10. A program that lists all the network interfaces

```
import java.net.*;
import java.util.*;

public class InterfaceLister {

    public static void main(String[] args) throws Exception {

      Enumeration interfaces = NetworkInterface.getNetworkInterfaces( );
      while (interfaces.hasMoreElements( )) {
        NetworkInterface ni = (NetworkInterface) interfaces.nextElement( );
        System.out.println(ni);
      }

    }

}
```

Here's the result of running this on the IBiblio login server:

```
% java InterfaceLister
name:eth1 (eth1) index: 3 addresses:
/192.168.210.122;

name:eth0 (eth0) index: 2 addresses:
/152.2.210.122;

name:lo (lo) index: 1 addresses:
/127.0.0.1;
```

You can see that this host has two separate Ethernet cards plus the local loopback address. Ignore the number of addresses (3, 2, and 1). It's a meaningless number, not the actual number of IP addresses bound to each interface.

6.3.2. Getter Methods

Once you have a `NetworkInterface` object, you can inquire about its IP address and name. This is pretty much the only thing you can do with these objects.

6.3.2.1. public Enumeration getInetAddresses( )

A single network interface may be bound to more than one IP address. This situation isn't common these days, but it does happen. The `getInetAddresses( )` method returns a `java.util.Enumeration` containing an `InetAddress` object for each IP address the interface is bound to. For example, this code fragment lists all the IP addresses for the eth0 interface:

```
NetworkInterface eth0 = NetworkInterrface.getByName("eth0");
Enumeration addresses = eth0.getInetAddresses( );
while (addresses.hasMoreElements( )) {
  System.out.println(addresses.nextElement( ));
}
```

6.3.2.2. public String getName( )

The `getName( )` method returns the name of a particular `NetworkInterface` object, such as eth0 or lo.

6.3.2.3. public String getDisplayName( )

The `getDisplayName( )` method allegedly returns a more human-friendly name for the particular `NetworkInterface`—something like "Ethernet Card 0". However, in my tests on Unix, it always returned the same string as `getName( )`. On Windows, you may see slightly friendlier names such as "Local Area Connection" or "Local Area Connection 2".

6.3.3. Object Methods

The `NetworkInterface` class defines the `equals()`, `hashCode( )`, and `toString ( )` methods with the usual semantics:

```
public boolean equals( )
public int hashCode( )
public String toString( )
```

Two `NetworkInterface` objects are equal if they represent the same physical network interface (e.g., both point to the same Ethernet port, modem, or wireless card) and they have the same IP address. Otherwise, they are not equal.

`NetworkInterface` does not implement `Cloneable`, `Serializable`, or `Comparable`. `NetworkInterface` objects cannot be cloned, compared, or serialized.

6.4. Some Useful Programs

You now know everything there is to know about the `java.net.InetAddress` class. The tools in this class alone let you write some genuinely useful programs. Here we'll look at two examples: one that queries your domain name server interactively and another that can improve the performance of your web server by processing log files offline.

6.4.1. HostLookup

nslookup is an old Unix utility that converts hostnames to IP addresses and IP addresses to hostnames. It has two modes: interactive and command-line. If you enter a hostname on the command line, *nslookup* prints the IP address of that host. If you enter an IP address on the command line, *nslookup* prints the hostname. If no hostname or IP address is entered on the command line, *nslookup* enters interactive mode, in which it reads hostnames and IP addresses from standard input and echoes back the corresponding IP addresses and hostnames until you type "exit". Example 6-11 is a simple character mode application called `HostLookup`, which emulates *nslookup*. It doesn't implement any of *nslookup*'s more complex features, but it does enough to be useful.

Example 6-11. An nslookup clone

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

public class HostLookup {

  public static void main (String[] args) {

    if (args.length > 0) { // use command line
      for (int i = 0; i < args.length; i++) {
        System.out.println(lookup(args[i]));
      }
    }
    else {
      BufferedReader in = new BufferedReader(new InputStreamReader
                                                 (System.in));
      System.out.println("Enter names and IP addresses.
                                   Enter \"exit\" to quit.");
      try {
        while (true) {
          String host = in.readLine( );
          if (host.equalsIgnoreCase("exit") ||
                                host.equalsIgnoreCase("quit")) {
            break;
          }
          System.out.println(lookup(host));
        }
      }
      catch (IOException ex) {
        System.err.println(ex);
      }

   }

  } /* end main */


  private static String lookup(String host) {

    InetAddress node;

    // get the bytes of the IP address
    try {
      node = InetAddress.getByName(host);
    }
    catch (UnknownHostException ex) {
      return "Cannot find host " + host;
    }

    if (isHostname(host)) {
      return node.getHostAddress( );
    }
    else {  // this is an IP address
      return node.getHostName( );
    }
  }  // end lookup

  private static boolean isHostname(String host) {

    // Is this an IPv6 address?
    if (host.indexOf(':') != -1) return false;

    char[] ca = host.toCharArray( );
    // if we see a character that is neither a digit nor a period
    // then host is probably a hostname
    for (int i = 0; i < ca.length; i++) {
      if (!Character.isDigit(ca[i])) {
        if (ca[i] != '.') return true;
      }
    }

    // Everything was either a digit or a period
    // so host looks like an IPv4 address in dotted quad format
    return false;

   }  // end isHostName

 } // end HostLookup
```

Here's some sample output; the input typed by the user is in bold:

```
$ java HostLookup utopia.poly.edu
128.238.3.21
$ java HostLookup 128.238.3.21
utopia.poly.edu
$ java HostLookup
Enter names and IP addresses. Enter "exit" to quit.
cs.nyu.edu
128.122.80.78
199.1.32.90
star.blackstar.com
localhost
127.0.0.1
stallio.elharo.com
Cannot find host stallio.elharo.com
stallion.elharo.com
127.0.0.1
127.0.0.1
stallion.elharo.com
java.oreilly.com
208.201.239.37
208.201.239.37
www.oreillynet.com
exit
$
```

There are three methods in the `HostLookup` program: `main( )`, `lookup( )`, and `isHostName( )`. The `main( )` method determines whether there are command-line arguments. If there are command-line arguments, `main()` calls `lookup( )` to process each one. If there are no command-line arguments, `main( )` chains a `BufferedReader` to an

`InputStreamReader` chained to `System.in` and reads input from the user with the `readLine( )` method. (The warning about this method in Chapter 4 doesn't apply here because the program is reading from the console, not a network connection.) If the line is "exit", then the program exits. Otherwise, the line is assumed to be a hostname or IP address and is passed to the `lookup()` method.

The `lookup( )` method uses `InetAddress.getByName( )` to find the requested host, regardless of the input's format; remember that `getByName( )` doesn't care if its argument is a name or a dotted quad address. If `getByName( )` fails, `lookup( )` returns a failure message. Otherwise, it gets the address of the requested system. Then `lookup( )` calls `isHostName( )` to determine whether the input string `host` is a hostname such as cs.nyu.edu, a dotted quad IPv4 address such as 128.122.153.70, or a hexadecimal IPv6 address such as *FEDC::DC:0:7076:10*. `isHostName()` first looks for colons, which any IPv6 hexadecimal address will have and no hostname will have. If it finds any, it returns false. Checking for IPv4 addresses is a little trickier because dotted quad addresses don't contain any character that can't appear in a hostname. Instead, `isHostName( )` looks at each character of the string; if all the characters are digits or periods, `isHostName( )` guesses that the string is a numeric IP address and returns false. Otherwise, `isHostName( )` guesses that the string is a hostname and returns true. What if the string is neither? Such an eventuality is very unlikely: if the string is neither a hostname nor an address, `getByName( )` won't be able to do a lookup and will throw an exception. However, it would not be difficult to add a test making sure that the string looks valid; this is left as an exercise for the reader. If the user types a hostname, `lookup( )` returns the corresponding dotted quad or hexadecimal address using `getHostAddress( )`. If the user types an IP address, then we use the `getHostName()` method to look up the hostname corresponding to the address, and return it.

6.4.2. Processing Web Server Log Files

Web server logs track the hosts that access a web site. By default, the log reports the IP addresses of the sites that connect to the server. However, you can often get more information from the names of those sites than from their IP addresses. Most web servers have an option to store hostnames instead of IP addresses, but this can hurt performance because the server needs to make a DNS request for each hit. It is much more efficient to log the IP addresses and convert them to hostnames at a later time, when the server isn't busy or even on another machine completely. Example 6-12 is a program called `Weblog` that reads a web server log file and prints each line with IP addresses converted to hostnames.

Most web servers have standardized on the common log file format, although there are exceptions; if your web server is one of those exceptions, you'll have to modify this program. A typical line in the common log file format looks like this:

```
205.160.186.76 unknown - [17/Jun/2003:22:53:58 -0500]
                              "GET /bgs/greenbg.gif HTTP 1. 0" 200 50
```

This line indicates that a web browser at IP address 205.160.186.76 requested the file */bgs/greenbg.gif* from this web server at 11:53 p.m. (and 58 seconds) on June 17, 2003. The file was found (response code 200) and 50 bytes of data were successfully transferred to the browser.

The first field is the IP address or, if DNS resolution is turned on, the hostname from which the connection was made. This is followed by a space. Therefore, for our purposes, parsing the log file is easy: everything before the first space is the IP address, and everything after it does not need to be changed.

The Common Log File Format

If you want to expand `Weblog` into a more general web server log processor, you need a little more information about the common log file format. A line in the file has the format:

```
remotehost rfc931 authuser [date] "request" status bytes
```

remotehost
: `remotehost` is either the hostname or IP address from which the browser connected.

rfc931
: rfc931 is the username of the user on the remote system, as specified by Internet protocol RFC 931. Very few browsers send this information, so it's almost always either unknown or a dash. This is followed by a space.

authuser
: `authuser` is the authenticated username as specified by RFC 931. Once again, most popular browsers or client systems do not support this; this field usually is filled in with a dash, followed by a space.

[date]
: The date and time of the request are given in brackets. This is the local system time when the request was made. Days are a two-digit number ranging from 01 to 31. The month is Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec. The year is indicated by four digits. The year is followed by a colon, the hour (from 00 to 23), another colon, two digits signifying the minute (00 to 59), a colon, and two digits signifying the seconds (00 to 59). Then comes the closing bracket and another space.

"request"
: The request line exactly as it came from the client. It is enclosed in quotation marks because it may contain embedded spaces. It is not guaranteed to be a valid HTTP request since client software may misbehave.

status
: A numeric HTTP status code returned to the client. A list of HTTP 1.0 status codes is given in Chapter 3. The most common response is 200, which means the request was successfully processed.

bytes
The number of bytes of data that was sent to the client as a result of this request.

The dotted quad format IP address is converted into a hostname using the usual methods of `java.net.InetAddress`. Example 6-12 shows the code.

Example 6-12. Process web server log files

```
import java.net.*;
import java.io.*;
import java.util.*;
import com.macfaq.io.SafeBufferedReader;

public class Weblog {

  public static void main(String[] args) {

    Date start = new Date( );
    try {
      FileInputStream fin =  new FileInputStream(args[0]);
      Reader in = new InputStreamReader(fin);
      SafeBufferedReader bin = new SafeBufferedReader(in);

      String entry = null;
      while ((entry = bin.readLine( )) != null) {

        // separate out the IP address
        int index = entry.indexOf(' ', 0);
        String ip = entry.substring(0, index);
        String theRest = entry.substring(index, entry.length( ));

        // find the hostname and print it out
        try {
          InetAddress address = InetAddress.getByName(ip);
          System.out.println(address.getHostName( ) + theRest);
        }
        catch (UnknownHostException ex) {
          System.out.println(entry);
        }

      } // end while
    }
    catch (IOException ex) {
      System.out.println("Exception: " + ex);
    }

    Date end = new Date( );
    long elapsedTime = (end.getTime( )-start.getTime( ))/1000;
    System.out.println("Elapsed time: " + elapsedTime + " seconds");

  }  // end main

}
```

The name of the file to be processed is passed to `Weblog` as the first argument on the command line. A `FileInputStream fin` is opened from this file and an `InputStreamReader` is chained to `fin`. This `InputStreamReader` is buffered by chaining it to an instance of the `SafeBufferedReader` class developed in Chapter 4. The file is processed line by line in a `while` loop.

Each pass through the loop places one line in the `String` variable `entry`. `entry` is then split into two substrings: `ip`, which contains everything before the first space, and `theRest`, which is everything after the first space. The position of the first space is determined by `entry.indexOf(" ", 0)`. `ip` is converted to an `InetAddress` object using `getByName()`. `getHostName( )` then looks up the hostname. Finally, the hostname, a space, and everything else on the line (`theRest`) are printed on `System.out`. Output can be sent to a new file through the standard means for redirecting output.

`Weblog` is more efficient than you might expect. Most web browsers generate multiple log file entries per page served, since there's an entry in the log not just for the page itself but for each graphic on the page. And many visitors request multiple pages while visiting a site. DNS lookups are expensive and it simply doesn't make sense to look up each site every time it appears in the log file. The `InetAddress` class caches requested addresses. If the same address is requested again, it can be retrieved from the cache much more quickly than from DNS.

Nonetheless, this program could certainly be faster. In my initial tests, it took more than a second per log entry. (Exact numbers depend on the speed of your network connection, the speed of the local and remote DNS servers, and network congestion when the program is run.) The program spends a huge amount of time sitting and waiting for DNS requests to return. Of course, this is exactly the problem multithreading is designed to solve. One main thread can read the log file and pass off individual entries to other threads for processing.

A thread pool is absolutely necessary here. Over the space of a few days, even low-volume web servers can easily generate a log file with hundreds of thousands of lines. Trying to process such a log file by spawning a new thread for each entry would rapidly bring even the strongest virtual machine to its knees, especially since the main thread can read log file entries much faster than individual threads can resolve domain names and die. Consequently, reusing threads is essential. The number of threads is stored in a tunable parameter, `numberOfThreads`, so that it can be adjusted to fit the VM and network stack. (Launching too many simultaneous DNS requests can also cause problems.)

This program is now divided into two classes. The first class, `PooledWeblog`, shown in Example 6-13, contains the `main( )` method and the `processLogFile( )` method. It also holds the resources that need to be shared among the threads. These are the pool, implemented as a synchronized `LinkedList` from the Java Collections API, and the output log, implemented as a `BufferedWriter` named `out`. Individual threads have direct access to the pool but have to pass through `PooledWeblog`'s `log( )` method to write output.

The key method is `processLogFile()`. As before, this method reads from the underlying log file. However, each entry is placed in the `entries` pool rather than being immediately processed. Because this method is likely to run much more quickly than the threads that have to access DNS, it yields after reading each entry. Furthermore, it goes to sleep if there are more entries in the pool than threads available to process them. The amount of time it sleeps depends on the number of threads. This setup avoids using excessive amounts of memory for very large log files. When the last entry is read, the `finished` flag is set to `true` to tell the threads that they can die once they've completed their work.

Example 6-13. PooledWebLog

```
import java.io.*;
import java.util.*;
import com.macfaq.io.SafeBufferedReader;

public class PooledWeblog {

  private BufferedReader in;
  private BufferedWriter out;
  private int numberOfThreads;
  private List entries = Collections.synchronizedList(new LinkedList( ));
  private boolean finished = false;
  private int test = 0;


  public PooledWeblog(InputStream in, OutputStream out,
   int numberOfThreads) {
    this.in = new BufferedReader(new InputStreamReader(in));
    this.out = new BufferedWriter(new OutputStreamWriter(out));
    this.numberOfThreads = numberOfThreads;
  }

  public boolean isFinished( ) {
    return this.finished;
  }

  public int getNumberOfThreads( ) {
    return numberOfThreads;
  }

  public void processLogFile( ) {

    for (int i = 0; i < numberOfThreads; i++) {
      Thread t = new LookupThread(entries, this);
      t.start( );
    }
```

```
          try {

            String entry = in.readLine( );
            while (entry != null) {

              if (entries.size( ) > numberOfThreads) {
                try {
                  Thread.sleep((long) (1000.0/numberOfThreads));
                }
                catch (InterruptedException ex) {}
                continue;
              }

              synchronized (entries) {
                entries.add(0, entry);
                entries.notifyAll( );
              }

              entry = in.readLine( );
              Thread.yield( );

            } // end while
        }

        public void log(String entry) throws IOException {
          out.write(entry + System.getProperty("line.separator", "\r\n"));
          out.flush( );
        }

        public static void main(String[] args) {

          try {
            PooledWeblog tw = new PooledWeblog(new FileInputStream(args[0]),
             System.out, 100);
            tw.processLogFile( );
          }
          catch (FileNotFoundException ex) {
            System.err.println("Usage: java PooledWeblog logfile_name");
          }
          catch (ArrayIndexOutOfBoundsException ex) {
            System.err.println("Usage: java PooledWeblog logfile_name");
          }
          catch (Exception ex) {
            System.err.println(ex);
            e.printStackTrace( );
          }

        }  // end main

      }
```

The `LookupThread` class, shown in Example 6-14, handles the detailed work of converting IP addresses to hostnames in the log entries. The constructor provides each thread with a reference to the `entries` pool it will retrieve work from and a reference to the `PooledWeblog` object it's working for. The latter reference allows callbacks to the `PooledWeblog` so that the thread can log converted entries and check to see when the last entry has been processed. It does so by calling the `isFinished( )` method in `PooledWeblog` when the `entries` pool is empty (i.e., has size 0). Neither an empty pool nor `isFinished( )` returning true is sufficient by itself. `isFinished( )` returns true after the last entry is placed in the pool, which occurs, at least for a small amount of time, before the last entry is removed from the pool. And `entries` may be empty while there are still many entries remaining to be read if the lookup threads outrun the main thread reading the log file.

Example 6-14. LookupThread

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

public class LookupThread extends Thread {

  private List entries;
  PooledWeblog log;   // used for callbacks

  public LookupThread(List entries, PooledWeblog log) {
    this.entries = entries;
    this.log = log;
  }

  public void run( ) {

    String entry;

    while (true) {

      synchronized (entries) {
        while (entries.size( ) == 0) {
          if (log.isFinished( )) return;
          try {
            entries.wait( );
          }
          catch (InterruptedException ex) {
          }
        }
        entry = (String) entries.remove(entries.size( )-1);
      }

      int index = entry.indexOf(' ', 0);
      String remoteHost = entry.substring(0, index);
      String theRest = entry.substring(index, entry.length( ));

      try {
        remoteHost = InetAddress.getByName(remoteHost).getHostName( );
      }
      catch (Exception ex) {
        // remoteHost remains in dotted quad format
      }
```

```
          try {
            log.log(remoteHost + theRest);
          }
          catch (IOException ex) {
          }
          this.yield( );

        }

      }

    }
```

Using threads like this lets the same log files be processed in parallel—a huge time-savings. In my unscientific tests, the threaded version is 10 to 50 times faster than the sequential version.

The biggest disadvantage to the multithreaded approach is that it reorders the log file. The output statistics aren't necessarily in the same order as the input statistics. For simple hit counting, this doesn't matter. However, there are some log analysis tools that can mine a log file to determine paths users followed through a site. These tools could get confused if the log is out of sequence. If the log sequence is an issue, attach a sequence number to each log entry. As the individual threads return log entries to the main program, the `log( )` method in the main program stores any that arrive out of order until their predecessors appear. This is in some ways reminiscent of how network software reorders TCP packets that arrive out of order.