CHAPTER 4

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 4-byte-long IPv4 addresses. However, a small but growing number of IP addresses are 16-byte-long 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 *login.ibiblio.org* is 152.19.134.132. 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.hamiltonweath-er.tk* is *2400:cb00:2048:0001:0000:0000:6ca2:c665*. Leading zeros do not need to be writ-ten. Therefore, the address of *www.hamiltonweather.tk* can be written as *2400:cb00:2048:1:0:0:6ca2:c665*. A double colon, at most one of which may appear in any address, indicates multiple zero blocks. For example, the address *2001:4860:4860:0000:0000:0000:0000:8888* can be written more compactly as *2001:4860:4860::8888*. 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 instead be written as *FEDC:BA98:7654:3210:FEDC:BA98:118.84.50.16*.

IP addresses are great for computers, but they are a problem for humans, who have a hard time remembering long numbers. In the 1950s, G. A. Miller 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. For more information on this, see "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," 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 *login.ibiblio.org*) with IP addresses that computers can remember (such as *152.19.134.132*). Servers usually have at least one hostname. Clients often have a hostname, but often don't, especially if their IP address is dynamically assigned at startup.

Colloquially, people often use "Internet address" to mean a hostname (or even an email address, or full URL). 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.beand.com* and *xom.nu* are really the same Linux box. The name *www.beand.com* really refers to a website rather than a particular machine. In the past, when this website 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. Because 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 websites, where it splits the load across multiple systems. For instance, *www.oreilly.com* is actually two machines, one at 208.201.239.100 and one at 208.201.239.101.

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.

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. Usually, it includes both a hostname and an IP address.

Creating New InetAddress Objects

There are no public constructors in the `InetAddress` class. Instead, `InetAddress` has static factory methods that connect to a DNS server to resolve a hostname. The most common is `InetAddress.getByName()`. For example, this is how you look up *www.oreilly.com*:

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

This method does not merely set a private `String` field in the `InetAddress` class. It actually makes a connection to the local DNS server to look up the name and the numeric address. (If you've looked up this host previously, the information may be cached locally, in which case a network connection is not required.) If the DNS server can't find the address, this method throws an `UnknownHostException`, a subclass of `IOException`.

Example 4-1 shows a complete program that creates an `InetAddress` object for *www.oreilly.com* including all the necessary imports and exception handling.

Example 4-1. 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
```

You can also do a reverse lookup by IP address. For example, if you want the hostname for the address 208.201.239.100, pass the dotted quad address to `InetAddress.getByName()`:

```
InetAddress address = InetAddress.getByName("208.201.239.100");
System.out.println(address.getHostName());
```

If the address you look up does not have a hostname, `getHostName()` simply returns the dotted quad address you supplied.

I mentioned earlier that *www.oreilly.com* actually has two addresses. Which one `getHostName()` returns is indeterminate. If, for some reason, you need all the addresses of a host, call `getAllByName()` instead, which returns an array:

```
try {
  InetAddress[] addresses = InetAddress.getAllByName("www.oreilly.com");
  for (InetAddress address : addresses) {
    System.out.println(address);
  }
} catch (UnknownHostException ex) {
  System.out.println("Could not find www.oreilly.com");
}
```

Finally, the `getLocalHost()` method returns an `InetAddress` object for the host on which your code is running:

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

This method tries to connect to DNS to get a real hostname and IP address such as "elharo.laptop.corp.com" and "192.1.254.68"; but if that fails it may return the *loopback* address instead. This is the hostname "localhost" and the dotted quad address "127.0.0.1".

Example 4-2 prints the address of the machine it's run on.

Example 4-2. 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.

If you know a numeric address, you can create an `InetAddress` object from that address without talking to DNS using `InetAddress.getByAddress()`. This method can create addresses for hosts that do not exist or cannot be resolved:

```
public static InetAddress getByAddress(byte[] addr) throws UnknownHostException
public static InetAddress getByAddress(String hostname, byte[] addr)
    throws UnknownHostException
```

The first `InetAddress.getByAddress()` factory method creates an `InetAddress` object with an IP address and no hostname. The second `InetAddress.getByAddress()` method creates an `InetAddress` object with an IP address and a hostname. For example, this code fragment makes an `InetAddress` for 107.23.216.196:

```
byte[] address = {107, 23, (byte) 216, (byte) 196};
InetAddress lessWrong = InetAddress.getByAddress(address);
InetAddress lessWrongWithname = InetAddress.getByAddress(
   "lesswrong.com", address);
```

Note that it had to cast the two large values to bytes.

Unlike the other 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. This could be 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. Because 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.)

Caching

Because 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. 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.

These times can be controlled by the system properties `networkaddress.cache.ttl` and `networkaddress.cache.negative.ttl`. The first of those, `networkad dress.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. Negative 1 is interpreted as "never expire."

Besides local caching inside the `InetAddress` class, the local host, the local domain name server, and other DNS servers elsewhere on the Internet 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 `UnknownHos tException`, `NoRouteToHostException`, and `ConnectException`, depending on the changes made to the DNS.

Lookups by IP address

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 only performed when the hostname is requested, either explicitly via a `getHostName()`. 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, the hostname remains the original dotted quad string. However, no `UnknownHostException` is thrown.

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 such as *www.oreilly.com* or an IP address such as 208.201.239.37, always choose the hostname. Use an IP address only when a hostname is not available.

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, in such an environment, `getLocalHost()` always returns a hostname of *localhost/127.0.0.1*. 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.

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 such as *www.oreilly.com*, a dotted quad IP address such as *208.201.239.37*, or a hexadecimal IPv6 address such as *FEDC::DC:0:7076:10*.

Getter Methods

The `InetAddress` class contains four 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 String getCanonicalHostName()
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. This makes `InetAddress` immutable and thus thread safe.

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();
```

The `getCanonicalHostName()` method is similar, but it's a bit more aggressive about contacting DNS. `getHostName()` will only call DNS if it doesn't think it already knows the hostname. `getCanonicalHostName()` calls DNS if it can, and may replace the existing cached hostname. For example:

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

The `getCanonicalHostName()` method is particularly useful when you're starting with a dotted quad IP address rather than the hostname. Example 4-3 converts the dotted quad address *208.201.239.37* into a hostname by using `InetAddress.getByName()` and then applying `getCanonicalHostName()` on the resulting object.

Example 4-3. Given the address, find the hostname

```
import java.net.*;

public class ReverseTest {

  public static void main (String[] args) throws UnknownHostException {
    InetAddress ia = InetAddress.getByName("208.201.239.100");
    System.out.println(ia.getCanonicalHostName());
  }
}
```

Here's the result:

```
% java ReverseTest
oreilly.com
```

The `getHostAddress()` method returns a string containing the dotted quad format of the IP address. Example 4-4 uses this method to print the IP address of the local machine in the customary format.

Example 4-4. 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.

If you want to know the IP address of a machine (and you rarely do), then use the `getAddress()` method, which 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. 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 4-5 demonstrates.

Example 4-5. Determining whether an IP address is v4 or v6

```
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;
  }
}
```

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 includes 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()
```

The `isAnyLocalAddress()` 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, as might be the case on a system with multiple Ethernet cards or an Ethernet card and an 802.11 WiFi interface. 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. ::).

The `isLoopbackAddress()` 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).

The `isLinkLocalAddress()` 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 packets addressed to a link-local address 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.

The `isSiteLocalAddress()` 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.

The `isMulticastAddress()` 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 13.

The `isMCGlobal()` 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 13, IPv4 uses time-to-live (TTL) values to control scope rather than addressing.

The `isMCOrgLocal()` 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.

The `isMCSiteLocal()` 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. Site-wide multicast addresses begin with FF05 or FF15, depending on whether the multicast address is a well known permanently assigned address or a transient address.

The `isMCLinkLocal()` 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.

The `isMCNodeLocal()` 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 this method was added to the JDK, but Sun didn't get the memo in time.

Example 4-6 is a simple program to test the nature of an address entered from the command line using these 10 methods.

Example 4-6. Testing the characteristics of an IP address

```
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.
```

Testing Reachability

The `InetAddress` class has two `isReachable()` methods that test whether a particular node is reachable from the current host (i.e., 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.

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

These methods attempt to use traceroute (more specifically, ICMP echo requests) to find out if the specified address is 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).

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()
```

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* because both names refer to the same IP address. Example 4-7 creates `InetAddress` objects for *www.ibiblio.org* and *helios.ibiblio.org* and then tells you whether they're the same machine.

Example 4-7. Are www.ibiblio.org and helios.ibiblio.org 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.ibiblio.org");
      if (ibiblio.equals(helios)) {
        System.out.println
            ("www.ibiblio.org is the same as helios.ibiblio.org");
      } else {
        System.out.println
            ("www.ibiblio.org is not the same as helios.ibiblio.org");
      }
    } 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.ibiblio.org
```

The `hashCode()` method is 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.

Like all good classes, `java.net.InetAddress` has a `toString()` method that returns a short text representation of the object. Example 4-1 and Example 4-2 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 and later, the hostname is set to the empty string.

Inet4Address and Inet6Address

Java uses two 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
```

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). `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 nonzero. 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 `Inet4Ad dress` instead. However, you rarely need to do this.

The NetworkInterface Class

The `NetworkInterface` class 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.

Factory Methods

Because `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.

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.");
}
```

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, 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.getByInetAddress(local);
  if (ni == null) {
    System.err.println("That's weird. No local loopback address.");
  }
} catch (SocketException ex) {
  System.err.println("Could not list network interfaces." );
} catch (UnknownHostException ex) {
  System.err.println("That's weird. Could not lookup 127.0.0.1.");
}
```

public static Enumeration getNetworkInterfaces() throws SocketException

The `getNetworkInterfaces()` method returns a `java.util.Enumeration` listing all the network interfaces on the local host. Example 4-8 is a simple program to list all network interfaces on the local host:

Example 4-8. A program that lists all the network interfaces

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

public class InterfaceLister {

  public static void main(String[] args) throws SocketException {
    Enumeration<NetworkInterface> interfaces = NetworkInterface.getNetwork
    Interfaces();
    while (interfaces.hasMoreElements()) {
      NetworkInterface ni = 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. The Ethernet card with index 2 has the IP address 152.2.210.122. The Ethernet card with index 3 has the IP address 192.168.210.122. The loopback interface has address 127.0.0.1, as always.

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.

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());
}
```

public String getName()

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

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."

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 you'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 logfiles offline.

SpamCheck

A number of services monitor spammers, and inform clients whether a host attempting to connect to them is a known spammer or not. These *real-time blackhole lists* need to respond to queries extremely quickly, and process a very high load. Thousands, maybe millions, of hosts query them repeatedly to find out whether an IP address attempting a connection is or is not a known spammer.

The nature of the problem requires that the response be fast, and ideally it should be cacheable. Furthermore, the load should be distributed across many servers, ideally ones located around the world. Although this could conceivably be done using a web server, SOAP, UDP, a custom protocol, or some other mechanism, this service is in fact cleverly implemented using DNS and DNS alone.

To find out if a certain IP address is a known spammer, reverse the bytes of the address, add the domain of the blackhole service, and look it up. If the address is found, it's a spammer. If it isn't, it's not. For instance, if you want to ask *sbl.spamhaus.org* if 207.87.34.17 is a spammer, you would look up the hostname *17.34.87.207.sbl.spamhaus.org*. (Note that despite the numeric component, this is a hostname ASCII string, not a dotted quad IP address.)

If the DNS query succeeds (and, more specifically, if it returns the address 127.0.0.2), then the host is known to be a spammer. If the lookup fails—that is, it throws an `UnknownHostException`—it isn't. Example 4-9 implements this check.

Example 4-9. SpamCheck

```java
import java.net.*;

public class SpamCheck {

  public static final String BLACKHOLE = "sbl.spamhaus.org";

  public static void main(String[] args) throws UnknownHostException {
    for (String arg: args) {
      if (isSpammer(arg)) {
        System.out.println(arg + " is a known spammer.");
      } else {
        System.out.println(arg + " appears legitimate.");
      }
    }
  }

  private static boolean isSpammer(String arg) {
    try {
      InetAddress address = InetAddress.getByName(arg);
      byte[] quad = address.getAddress();
      String query = BLACKHOLE;
      for (byte octet : quad) {
        int unsignedByte = octet < 0 ? octet + 256 : octet;
        query = unsignedByte + "." + query;
      }
      InetAddress.getByName(query);
      return true;
    } catch (UnknownHostException e) {
      return false;
    }
  }
}
```

Here's some sample output:

```
$ java SpamCheck 207.34.56.23 125.12.32.4 130.130.130.130
207.34.56.23 appears legitimate.
125.12.32.4 appears legitimate.
130.130.130.130 appears legitimate.
```

If you use this technique, be careful to stay on top of changes to blackhole list policies and addresses. For obvious reasons, blackhole servers are frequent targets of DDOS and other attacks, so you want to be careful that if the blackhole server changes its address or simply stops responding to any queries, you don't begin blocking all traffic.

Further note that different blackhole lists can follow slightly different protocols. For example, a few lists return *127.0.0.1* for spamming IPs instead of *127.0.0.2*.

Processing Web Server Logfiles

Web server logs track the hosts that access a website. 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 4-10 is a program called `Weblog` that reads a web server logfile and prints each line with IP addresses converted to hostnames.

Most web servers have standardized on the common logfile format. A typical line in the common logfile format looks like this:

```
205.160.186.76 unknown - [17/Jun/2013:22:53:58 -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, 2013. 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 logfile is easy: everything before the first space is the IP address, and everything after it does not need to be changed.

The dotted quad format IP address is converted into a hostname using the usual methods of `java.net.InetAddress`. Example 4-10 shows the code.

Example 4-10. Process web server logfiles

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

public class Weblog {

  public static void main(String[] args) {
    try (FileInputStream fin =  new FileInputStream(args[0]);
      Reader in = new InputStreamReader(fin);
      BufferedReader bin = new BufferedReader(in);) {

      for (String entry = bin.readLine();
        entry != null;
        entry = bin.readLine()) {
        // separate out the IP address
        int index = entry.indexOf(' ');
        String ip = entry.substring(0, index);
        String theRest = entry.substring(index);

        // Ask DNS for the hostname and print it out
        try {
          InetAddress address = InetAddress.getByName(ip);
          System.out.println(address.getHostName() + theRest);
        } catch (UnknownHostException ex) {
          System.err.println(entry);
        }
      }
    } catch (IOException ex) {
      System.out.println("Exception: " + ex);
    }
  }
}
```

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 InputStream Reader is chained to fin. This InputStreamReader is buffered by chaining it to an instance of the BufferedReader class. The file is processed line by line in a for 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 from the first space to the end of the string. The position of the first space is determined by entry.indexOf(" "). The substring ip is converted to an InetAddress object using getByName(). getHostName() then looks up the hostname. Finally, the hostname 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 logfile entries per page served, because 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 logfile. 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 logfile 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 generate a logfile with hundreds of thousands of lines. Trying to process such a logfile by spawning a new thread for each entry would rapidly bring even the strongest virtual machine to its knees, especially because the main thread can read logfile 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, LookupTask, shown in Example 4-11, is a Callable that parses a logfile entry, looks up a single address, and replaces that address with the corresponding hostname. This doesn't seem like a lot of work and CPU-wise, it isn't. However, because it involves a network connection, and possibly a hierarchical series of network connections between many different DNS servers, it has a lot of downtime that can be put to better use by other threads.

Example 4-11. LookupTask

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

public class LookupTask implements Callable<String> {

  private String line;

  public LookupTask(String line) {
    this.line = line;
  }

  @Override
  public String call() {
    try {
      // separate out the IP address
      int index = line.indexOf(' ');
      String address = line.substring(0, index);
      String theRest = line.substring(index);
      String hostname = InetAddress.getByName(address).getHostName();
      return hostname + " " + theRest;
    } catch (Exception ex) {
      return line;
    }
  }
}
```

The second class, `PooledWeblog`, shown in Example 4-12, contains the `main()` method that reads the file and creates one `LookupTask` per line. Each task is submitted to an executor that can run multiple (though not all) tasks in parallel and in sequence.

The `Future` that is returned from the `submit()` method is stored in a queue, along with the original line (in case something goes wrong in the asynchronous thread). A loop reads values out of the queue and prints them. This maintains the original order of the logfile.

Example 4-12. PooledWebLog

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

// Requires Java 7 for try-with-resources and multi-catch
public class PooledWeblog {

  private final static int NUM_THREADS = 4;

  public static void main(String[] args) throws IOException {
    ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
    Queue<LogEntry> results = new LinkedList<LogEntry>();

    try (BufferedReader in = new BufferedReader(
      new InputStreamReader(new FileInputStream(args[0]), "UTF-8"));) {
      for (String entry = in.readLine(); entry != null; entry = in.readLine()) {
        LookupTask task = new LookupTask(entry);
        Future<String> future = executor.submit(task);
        LogEntry result = new LogEntry(entry, future);
        results.add(result);
      }
    }

    // Start printing the results. This blocks each time a result isn't ready.
    for (LogEntry result : results) {
      try {
        System.out.println(result.future.get());
      } catch (InterruptedException | ExecutionException ex) {
        System.out.println(result.original);
      }
    }

    executor.shutdown();
  }

  private static class LogEntry {
    String original;
    Future<String> future;

    LogEntry(String original, Future<String> future) {
     this.original = original;
     this.future = future;
    }
  }
}
```

Using threads like this lets the same logfiles 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 tech editor ran the same test on a different system and only saw a factor of four improvement, but either way it's still a significant gain.

There's still one downside to this design. Although the queue of `Callable` tasks is much more efficient than spawning a thread for each logfile entry, logfiles can be huge and this program can still burn a lot of memory. To avoid this, you could put the output into a separate thread that shared the queue with the input thread. Because early entries could be processed and output while the input was still being parsed, the queue would not grow so large. This does, however, introduce another problem. You'd need a separate signal to tell you when the output was complete because an empty queue is no longer sufficient to prove the job is complete. The easiest way is simply to count the number of input lines and make sure it matches up to the number of output lines.