UNIT 2
INTERNET ADDRESSES

LH - 4HRS

PREPARED BY:

ER. SHARAT MAHARJAN

NETWORK PROGRAMMING

PRIME COLLEGE

CONTENTS (LH - 4HRS)

2.1 The InetAddress Class: Creating New InetAddress Objects, Getter Methods

2.2 Address Types, Testing Reachability and Object Methods

2.3 Inet4Address and Inet6Address

2.4 The Network Interface Class: Factory Method & Getter Method

2.5 Some Useful Programs: SpamCheck, Processing Web Server Logfiles

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 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.
- An IPv4 address is normally written as four unsigned bytes, each ranging from 0 to 255. Bytes are separated by periods for the convenience of human eyes. For example, the address for javatpoint is 104.21.79.8. 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, the address of www.javatpoint.com is 2606:4700:3037::6815:4f08. Leading zeros do not need to be written.

- 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 as 2001:4860:4860::8888.
- In mixed networks of IPv4 and IPv6, 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 instead written as FEDC:BA98:7654:3210:FEDC:BA98:118.84.50.16.
- The internet's designer invented the Domain Name System (DNS).

2.1 The InetAddress Class

- The java.net.InetAddress class is 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 we look up www.javatpoint.com:

- InetAddress address = InetAddress.getByName("www.javatpoint.com");

  It actually makes connection in the local DNS server to look up the name and the numeric address and if the DNS server can't find the address, this method throws an UnknownHostException, a subclass of IOException.

Example 1: A program that prints the address of www.javatpoint.com.

```
package examples;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class JavaInternetAddressByName {
	public static void main(String[] args) {
		try {
			InetAddress address = InetAddress.getByName("www.javatpoint.com");
			System.out.println(address);
		} catch(UnknownHostException e) {
			System.out.println("Couldn't find the host.");
		}
	}
}
```

OUTPUT:

```
www.javatpoint.com/104.21.79.8
```

- We can also do a reverse lookup by IP address. For example, if we want the hostname for the address 104.21.79.8, passing the dotted quad address to InetAddress.getByName():

```
InetAddress ia = InetAddress.getByName("104.21.79.8");
System.out.println(ia.getHostName());
```

- If the address we look up doesn't have a hostname, getHostName() simply returns the dotted quad address we supplied.

Getter Methods

Methods	Description
public static InetAddress getByName(String host) throws UnknownHostException	It returns the instance of InetAddress containing Host IP and name.
public static InetAddress getLocalHost() throws UnknownHostException	It returns the instance of InetAddress containing local host and address.
public String getHostName()	It returns the host name of the IP address.
public String getHostAddress()	It returns the IP address in string format.

- 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();
System.out.println(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.

2. Given the address, find the hostname.

```
import java.net.*;
public class ReverseTest {
     public static void main(String[] args) {
          try {
          InetAddress machine =
               InetAddress.getByName("104.21.79.8");
          System.out.println(machine.getCanonicalHostName());
          } catch (UnknownHostException e) {
               System.out.println("No hostname found.");
          }
     }
}
```

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

3. Find the IP address of the local machine.

```
import java.net.*;
public class Unit2 {
	public static void main(String[] args) {
		try {
		InetAddress machine = InetAddress.getLocalHost();
		System.out.println(machine.getHostAddress());
		} catch (UnknownHostException e) {
		System.out.println("No hostname found.");
		}
	}
}
```

- If you want to know the IP address of a machine, then use the **getAddress()** method, which returns an IP address as an array of bytes in network byte order. 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 ints 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.
- **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.**

4. Determining whether an IP address is v4 or v6.

```
import java.net.*;
public class AddressTest{
      public static void main(String[] args) {
            try {
            InetAddress machine =
            InetAddress.getByName("www.prime.edu.np");
            byte[] address = machine.getAddress();
                  if(address.length == 4)
                        System.out.println("IPv4 is being used.");
                  else
                        System.out.println("IPv6 is being used.");
            } catch (UnknownHostException e) {
            System.out.println("No hostname found.");
      }}}
```

2.2 Address Types, Testing Reachability and Object Methods

Address Types

Some IP addresses and some patterns of addresses have special meanings. For instance, 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:

1. public boolean isAnyLocalAddress()
2. public boolean isLoopbackAddress()
3. public boolean isLinkLocalAddress()
4. public boolean isSiteLocalAddress()
5. public boolean isMulticastAddress()
6. public boolean isMCGlobal()
7. public boolean isMCNodeLocal()
8. public boolean isMCLinkLocal()
9. public boolean isMCSiteLocal()
10. public boolean isMCOrgLocal()

1. The **isAnyLocalAddress()** method returns true if the address is a wildcard address, false otherwise. 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. ::).
2. The **isLoopbackAddress()** method returns true if the address is the loopback address, false otherwise. 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).
3. 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.
4. 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.
5. 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**.

6. 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**.
7. 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.
8. 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.
9. 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.
10. 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.

5. 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()) {  //begin if
 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."); } }   //end if
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.** (only check ip whether sites are on same server-shared web hosting)

6. 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:

Cmd: java IBiblioAliases

Output: 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 1 **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.

2.3 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

- **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 Inet4Address instead. However, you rarely need to do this.

2.4 The Network Interface Class: Factory Method & Getter Method

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 below is a simple program to list all network interfaces on the local host:

7. 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.
                                                    getNetworkInterfaces();
    while (interfaces.hasMoreElements()) {
      NetworkInterface ni = interfaces.nextElement();
      System.out.println(ni);
    }}}
```

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

Cmd: java InterfaceLister

Output:

- 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 = NetworkInterface.getByName(“wlan2");

Enumeration<InetAddress> 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.”

2.5 Some Useful Programs: SpamCheck, Processing Web Server Logfiles

- 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

- 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**(service monitoring spammers and inform clients), 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.
- If the DNS query succeeds then the host is known to be a spammer. If the lookup fails—that is, it throws an UnknownHostException—it isn't.

8. SpamCheck

```
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(); //bytes not string
     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;
    }
  }
}
```

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 9** 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.

9. Processing Web Server Logfiles

```
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);
    }}}
```

THANK YOU FOR YOUR ATTENTION