chapter 2

Basic Sockets

You are now ready to learn about writing your own socket applications. We begin by demonstrating how Java applications identify network hosts using the InetAddress and SocketAddress abstractions. Then we present examples of the use of Socket and ServerSocket, through an example client and server that use TCP. Then we do the same thing for the DatagramSocket abstraction for clients and servers that use UDP. For each abstraction, we list the most significant methods, grouped according to usage, and briefly describe their behavior.[1]

2.1 Socket Addresses

Recall that a client must specify the IP address of the host running the server program when it initiates communication. The network infrastructure then uses this *destination address* to route the client's information to the proper machine. Addresses can be specified in Java using a string that contains either a numeric address—in the appropriate form for the version, e.g., 192.0.2.27 for IPv4 or fe20:12a0::0abc:1234 for IPv6—or a name (e.g., *server.example.com*). In the latter case the name must be *resolved* to a numerical address before it can be used for communication.

[1]**Note:** For each Java networking class described in this text, we include only the most important and commonly used methods, omitting those that are deprecated or beyond the use of our target audience. However, this is something of a moving target. For example, the number of methods provided by the Socket class grew from 23 to 42 between version 1.3 and version 1.6 of the language. The reader is encouraged and expected to refer to the API specification documentation from *http://java.sun.com* as the current and definitive source.

The InetAddress abstraction represents a network destination, encapsulating both names and numerical address information. The class has two subclasses, Inet4Address and Inet6Address, representing the two versions in use. Instances of InetAddress are immutable: once created, each one always refers to the same address. We'll demonstrate the use of InetAddress with an example program that first prints out all the addresses—IPv4 and IPv6, if any—associated with the local host, and then prints the names and addresses associated with each host specified on the command line.

To get the addresses of the local host, the program takes advantage of the Network Interface abstraction. Recall that IP addresses are actually assigned to the connection between a host and a network (and not to the host itself). The NetworkInterface class provides access to information about all of a host's interfaces. This is extremely useful, for example when a program needs to inform another program of its address.

InetAddressExample.java

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

public class InetAddressExample {

  public static void main(String[] args) {

    // Get the network interfaces and associated addresses for this host
    try {
      Enumeration<NetworkInterface> interfaceList = NetworkInterface.getNetworkInterfaces();
      if (interfaceList == null) {
        System.out.println("--No interfaces found--");
      } else {
        while (interfaceList.hasMoreElements()) {
          NetworkInterface iface = interfaceList.nextElement();
          System.out.println("Interface " + iface.getName() + ":");
          Enumeration<InetAddress> addrList = iface.getInetAddresses();
          if (!addrList.hasMoreElements()) {
            System.out.println("\t(No addresses for this interface)");
          }
          while (addrList.hasMoreElements()) {
            InetAddress address = addrList.nextElement();
            System.out.print("\tAddress "
                + ((address instanceof Inet4Address ? "(v4)"
                    : (address instanceof Inet6Address ? "(v6)" : "(?)"))));
            System.out.println(": " + address.getHostAddress());
          }
        }
      }
29     } catch (SocketException se) {
30       System.out.println("Error getting network interfaces:" + se.getMessage());
31     }
32
33     // Get name(s)/address(es) of hosts given on command line
34     for (String host : args) {
35       try {
36         System.out.println(host + ":");
37         InetAddress[] addressList = InetAddress.getAllByName(host);
38         for (InetAddress address : addressList) {
39           System.out.println("\t" + address.getHostName() + "/" + address.getHostAddress());
40         }
41       } catch (UnknownHostException e) {
42         System.out.println("\tUnable to find address for " + host);
43       }
44     }
45   }
46 }
```

InetAddressExample.java

1. **Get a list of this host's network interfaces:** line 9
   The static method `getNetworkInterfaces()` returns a list containing an instance of `NetworkInterface` for each of the host's interfaces.
2. **Check for empty list:** lines 10–12
   The loopback interface is generally always included, even if the host has no other network connection, so this check will succeed only if the host has no networking subsystem at all.
3. **Get and print address(es) of each interface in the list:** lines 13–27
   - **Print the interface's name:** line 15
     The `getName()` method returns a local name for the interface. This is usually a combination of letters and numbers indicating the type and particular instance of the interface—for example, "`lo0`" or "`eth0`".
   - **Get the addresses associated with the interface:** line 16
     The `getInetAddresses()` method returns another `Enumeration`, this time containing instances of `InetAddress`—one per address associated with the interface. Depending on how the host is configured, the list may contain only IPv4, only IPv6, or a mixture of both types of address.
   - **Check for empty list:** lines 17–19
   - **Iterate through the list, printing each address:** lines 20–26
     We check each instance to determine which subtype it is. (At this time the only subtypes of `InetAddress` are those listed, but conceivably there might be others someday.) The

getHostAddress() method of InetAddress returns a String representing the numerical address in the format appropriate for its specific type: dotted-quad for v4, colon-separated hex for v6. See the synopsis "String representations" below for a description of the different address formats.

4. **Catch exception:** lines 29–31
   The call to getNetworkInterfaces() can throw a SocketException.
5. **Get names and addresses for each command-line argument:** lines 34–44
   - **Get list of addresses for the given name/address:** line 37
   - **Iterate through the list, printing each:** lines 38–40
     For each host in the list, we print the name returned by getHostName() followed by the numerical address returned by getHostAddress().

To use this application to find information about the local host, the publisher's Web server (*www.mkp.com*), a fake name (*blah.blah*), and an IP address, do the following:

```
% java InetAddressExample www.mkp.com blah.blah 129.35.69.7

Interface lo:
Address (v4): 127.0.0.1
Address (v6): 0:0:0:0:0:0:0:1
Address (v6): fe80:0:0:0:0:0:0:1%1
Interface eth0:
Address (v4): 192.168.159.1
Address (v6): fe80:0:0:0:250:56ff:fec0:8%4
www.mkp.com:
www.mkp.com/129.35.69.7
blah.blah:
Unable to find address for blah.blah
129.35.69.7:
129.35.69.7/129.35.69.7
```

You may notice that some v6 addresses have a suffix of the form %*d*, where *d* is a number. Such addresses have limited scope (typically they are link-local), and the suffix identifies the particular scope with which they are associated; this ensures that each listed address string is unique. Link-local IPv6 addresses begin with fe8.

You may also have noticed a delay when resolving blah.blah. Your resolver looks in several places before giving up on resolving a name. When the name service is not available for some reason—say, the program is running on a machine that is not connected to any network—attempting to identify a host by name may fail. Moreover, it may take a significant amount of time to do so, as the system tries various ways to resolve the name to an IP address. It is, therefore, good to know that you can always refer to a host using the IP address in dotted-quad notation. In any of the examples in this book, if a remote host is specified by name, the host running the example must be configured to convert names to addresses, or the example won't work. If you can ping a host using one of its names (e.g., run the command "ping *server.example.com*"), then the examples should work with names. If your ping test fails or the example hangs, try specifying the host by IP address, which avoids the name-to-address conversion altogether. (See also the `isReachable()` method of `InetAddress`, discussed below.)

`InetAddress`: Creating and accessing

```
static InetAddress[ ] getAllByName(String host)
static InetAddress getByName(String host)
static InetAddress getLocalHost()
byte[] getAddress()
```

The static factory methods return instances that can be passed to other `Socket` methods to specify a host. The input `String` to the factory methods can be either a domain name, such as “skeezix” or “*farm.example.com*”, or a string representation of a numeric address. For numeric IPv6 addresses, the shorthand forms described in Chapter 1 may be used. A name may be associated with more than one numeric address; the `getAllByName()` method returns an instance for each address associated with a name.

The `getAddress()` method returns the binary form of the address as a byte array of appropriate length. If the instance is of `Inet4Address`, the array is four bytes in length; if of `Inet6Address`, it is 16 bytes. The first element of the returned array is the most significant byte of the address.

As we have seen, an `InetAddress` instance may be converted to a `String` representation in several ways.

`InetAddress`: String representations

```
String toString()
String getHostAddress()
String getHostName()
String getCanonicalHostName()
```

These methods return the name or numeric address of the host, or a combination thereof, as a properly formatted `String`. The `toString()` method overrides the `Object` method to return a string of the form “*hostname.example.com/192.0.2.127*” or “*never.example.net/2000::620:1a30:95b2*”. The numeric representation of the address (only) is returned by `getHostAddress()`. For an IPv6 address, the string representation always includes the full eight groups (i.e., exactly seven colons “:”) to prevent ambiguity when a port number is appended separated by another colon—a common idiom that we’ll see later. Also, an IPv6 address that has limited scope, such as a link-local address will have a *scope identifier* appended. This is a local identifier added to prevent ambiguity (since the same link-local address can be used on different links), but is not part of the address transmitted in the packet.

The last two methods return the name of the host only, their behavior differing as follows: If this instance was originally created by giving a name, `getHostName()` will return that name with no resolution step; otherwise, `getHostName()` resolves the address to the name using the system-configured resolution mechanism. The `getCanonicalName()` method, on the other hand, always tries to resolve the address to obtain a *fully qualified domain name* (like "*ns1.internat.net*" or "*bam.example.com*"). Note that that address might differ from the one with which the instance was created, if different names map to the same address. Both methods return the numerical form of the address if resolution cannot be completed. Also, both check permission with the security manager before sending any messages.

The `InetAddress` class also supports checking for properties, such as membership in a class of "special purpose" addresses as discussed in Section 1.2, and reachability, i.e., the ability to exchange packets with the host.

`InetAddress`: Testing properties

```
boolean isAnyLocalAddress()
boolean isLinkLocalAddress()
boolean isLoopbackAddress()
boolean isMulticastAddress()
boolean isMCGlobal()
boolean isMCLinkLocal()
boolean isMCNodeLocal()
boolean isMCOrgLocal()
boolean isMCSiteLocal()
boolean isReachable(int timeout)
boolean isReachable(NetworkInterface netif, int ttl, int timeout)
```

These methods check whether an address is of a particular type. They all work for both IPv4 and IPv6 addresses. The first three methods above check whether the instance is one of, respectively, the "don't care" address, an address in the link-local class, or the loopback address (matches 127.*.*.* or ::1). The fourth method checks whether it is a multicast address (see Section 4.3.2), and the `isMC...()` methods check for various *scopes* of multicast address. (The scope determines, roughly, how far packets addressed to that destination can travel from their origin.)

The last two methods check whether it is actually possible to exchange packets with the host identified by this `InetAddress`. Note that, unlike the other methods, which involve simple syntactic checks, these methods cause the networking system to take action, namely sending packets. The system attempts to send a packet until the specified number of milliseconds passes. The latter form is more specific: it determines whetherthe destination can be contacted by sending packets out over the specified NetworkInterface, with the specified *time-to-live (TTL)* value. The TTL limits the distance a packet can travel through the network. Effectiveness of these last two methods may be limited by the security manager configuration.

The NetworkInterface class provides a large number of methods, many of which are beyond the scope of this book. We describe here the most useful ones for our purposes.

NetworkInterface: Creating, getting information

```
static Enumeration⟨NetworkInterface⟩ getNetworkInterfaces()
static NetworkInterface getByInetAddress(InetAddress addr)
static NetworkInterface getByName(String name)
Enumeration⟨InetAddress⟩ getInetAddresses()
String getName()
String getDisplayName()
```

The first method above is quite useful, making it easy to learn an IP address of the host a program is running on: you get the list of interfaces with getNetworkInterfaces(), and use the getInetAddresses() instance method to get all the addresses of each. *Caveat:* the list contains *all* the interfaces of the host, including the loopback virtual interface, which cannot send or receive messages to the rest of the network. Similarly, the list of addresses may contain link-local addresses that also are not globally reachable. Since the order is unspecified, you cannot simply take the first address of the first interface and assume it can be reached from the Internet; instead, use the property-checking methods of InetAddress (see above) to find one that is not loopback, not link-local, etc.

The getName() methods return the name of the *interface* (not the host). This generally consists of an alphabetic string followed by a numeric part, for example eth0. The loopback interface is named lo0 on many systems.

2.2 TCP Sockets

Java provides two classes for TCP: Socket and ServerSocket. An instance of Socket represents one end of a TCP connection. A *TCP connection* is an abstract two-way channel whose ends are each identified by an IP address and port number. Before being used for communication, a TCP connection must go through a setup phase, which starts with the client's TCP sending a connection request to the server's TCP. An instance of ServerSocket listens for TCP connection requests and creates a new Socket instance to handle each incoming connection. Thus, servers handle both ServerSocket and Socket instances, while clients use only Socket.

We begin by examining an example of a simple client.

2.2.1 TCP Client

The client initiates communication with a server that is passively waiting to be contacted. The typical TCP client goes through three steps:

1. Construct an instance of Socket: The constructor establishes a TCP connection to the specified remote host and port.
2. Communicate using the socket's I/O streams: A connected instance of Socket contains an InputStream and OutputStream that can be used just like any other Java I/O stream (see Section 2.2.3).
3. Close the connection using the close() method of Socket.

Our first TCP application, called TCPEchoClient.java, is a client that communicates with an *echo server* using TCP. An echo server simply repeats whatever it receives back to the client. The string to be echoed is provided as a command-line argument to our client. Some systems include an echo server for debugging and testing purposes. You may be able to use a program such as **telnet** to test if the standard echo server is running on your system (e.g., at command line "telnet server.example.com 7"); or you can go ahead and run the example server introduced in the next section.)

TCPEchoClient.java

```
import java.net.Socket;
import java.net.SocketException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class TCPEchoClient {

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

    if ((args.length < 2) || (args.length > 3))  // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");

    String server = args[0];       // Server name or IP address
    // Convert argument String to bytes using the default character encoding
15      byte[] data = args[1] getBytes();
16
17      int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
18
19      // Create socket that is connected to server on specified port
20      Socket socket = new Socket(server, servPort);
21      System.out.println("Connected to server...sending echo string");
22
23      InputStream in = socket.getInputStream();
24      OutputStream out = socket.getOutputStream();
25
26      out.write(data);  // Send the encoded string to the server
27
28      // Receive the same string back from the server
29      int totalBytesRcvd = 0;  // Total bytes received so far
30      int bytesRcvd;           // Bytes received in last read
31      while (totalBytesRcvd < data.length) {
32        if ((bytesRcvd = in.read(data, totalBytesRcvd,
33                         data.length - totalBytesRcvd)) == -1)
34          throw new SocketException("Connection closed prematurely");
35        totalBytesRcvd += bytesRcvd;
36      }  // data array is full
37
38      System.out.println("Received: " + new String(data));
39
40      socket.close();  // Close the socket and its streams
41    }
42  }
```

TCPEchoClient.java

1. **Application setup and parameter parsing:** lines 0–17
   - **Convert the echo string:** line 15
     TCP sockets send and receive sequences of bytes. The `getBytes()` method of `String` returns a byte array representation of the string. (See Section 3.1 for a discussion of character encodings.)
   - **Determine the port of the echo server:** line 17
     The default echo port is 7. If we specify a third parameter, `Integer.parseInt()` takes the string and returns the equivalent integer value.
2. **TCP socket creation:** line 20
   The `Socket` constructor creates a socket and connects it to the specified server, identified either by name or IP address, before returning. Note that the underlying TCP

deals only with IP addresses; if a name is given, the implementation resolves it to the corresponding address. If the connection attempt fails for any reason, the constructor throws an `IOException`.

3. **Get socket input and output streams:** lines 23–24
   Associated with each connected `Socket` instance is an `InputStream` and an `OutputStream`. We send data over the socket by writing bytes to the `OutputStream` just as we would any other stream, and we receive by reading from the `InputStream`.
4. **Send the string to echo server:** line 26
   The `write()` method of `OutputStream` transmits the given byte array over the connection to the server.
5. **Receive the reply from the echo server:** lines 29–36
   Since we know the number of bytes to expect from the echo server, we can repeatedly receive bytes until we have received the same number of bytes we sent. This particular form of `read()` takes three parameters: 1) byte array to receive into, 2) byte offset into the array where the first byte received should be placed, and 3) the maximum number of bytes to be placed in the array. `read()` blocks until some data is available, reads up to the specified maximum number of bytes, and returns the number of bytes actually placed in the array (which may be less than the given maximum). The loop simply fills up *data* until we receive as many bytes as we sent. If the TCP connection is closed by the other end, `read()` returns −1. For the client, this indicates that the server prematurely closed the socket.

   Why not just a single read? TCP does not preserve `read()` and `write()` message boundaries. That is, even though we sent the echo string with a single `write()`, the echo server may receive it in multiple chunks. Even if the echo string is handled in one chunk by the echo server, the reply may still be broken into pieces by TCP. One of the most common errors for beginners is the assumption that data sent by a single `write()` will always be received in a single `read()`.
6. **Print echoed string:** line 38
   To print the server's response, we must convert the byte array to a string using the default character encoding.
7. **Close socket:** line 40
   When the client has finished receiving all of the echoed data, it closes the socket.

We can communicate with an echo server named *server.example.com* with IP address 192.0.2.1 in either of the following ways:

```
% java TCPEchoClient server.example.com "Echo this!"
Received: Echo this!
% java TCPEchoClient 192.0.2.1 "Echo this!"
Received: Echo this!
```

See `TCPEchoClientGUI.java` on the book's Web site for an implementation of the TCP echo client with a graphical interface.

Socket: Creation

```
Socket(InetAddress remoteAddr, int remotePort)
Socket(String remoteHost, int remotePort)
Socket(InetAddress remoteAddr, int remotePort, InetAddress localAddr, int localPort)
Socket(String remoteHost, int remotePort, InetAddress localAddr, int localPort)
Socket()
```

The first four constructors create a TCP socket and *connect* it to the specified remote address and port before returning. The first two do not specify the local address and port, so a default local address and some available port are chosen. Specifying the local address may be useful on a host with multiple interfaces. `String` arguments that specify destinations can be in the same formats that are accepted by the `InetAddress` creation methods. The last constructor creates an unconnected socket, which must be explicitly connected (via the `connect()` method, see below) before it can be used for communication.

Socket: Operations

```
void connect(SocketAddress destination)
void connect(SocketAddress destination, int timeout)
InputStream getInputStream()
OutputStream getOutputStream()
void close()
void shutdownInput()
void shutdownOutput()
```

The `connect()` methods cause a TCP connection to the specified endpoints to be opened. The abstract class `SocketAddress` represents a generic form of address for a socket; its subclass `InetSocketAddress` is specific to TCP/IP sockets (see description below). Communication with the remote system takes place via the associated I/O streams, which are obtained through the `get...Stream()` methods.

The `close()` method closes the socket and its associated I/O streams, preventing further operations on them. The `shutDownInput()` method closes the input side of a TCP stream. Any unread data is silently discarded, including data buffered by the socket, data in transit, and data arriving in the future. Any subsequent attempt to read from the socket will cause an exception to be thrown. The `shutDownOutput()` method has a similar effect on the output stream, but the implementation will attempt to ensure that any data already written to the socket's output stream is delivered to the other end. See Section 4.5 for further details.

Caveat: By default, Socket is implemented on top of a TCP connection; however, in Java, you can actually change the underlying implementation of Socket. This book is about TCP/IP, so for simplicity we assume that the underlying implementation for all of these networking classes is the default.

Socket: Getting/testing attributes

```
InetAddress getInetAddress()
int getPort()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getRemoteSocketAddress()
SocketAddress getLocalSocketAddress()
```

These methods return the indicated attributes of the socket, and any method in this book that returns a SocketAddress actually returns an instance of InetSocketAddress. The InetSocketAddress encapsulates an InetAddress and a port number.

The Socket class actually has a large number of other associated attributes referred to as *socket options*. Because they are not necessary for writing basic applications, we postpone introduction of them until Section 4.4.

InetSocketAddress: Creating and accessing

```
InetSocketAddress(InetAddress addr, int port)
InetSocketAddress(int port)
InetSocketAddress(String hostname, int port)
static InetSocketAddress createUnresolved(String host, int port)
boolean isUnresolved()
InetAddress getAddress()
int getPort()
String getHostName()
String toString()
```

The InetSocketAddress class provides an immutable combination of host address and port. The port-only constructor uses the special "any" address, and is useful for servers. The constructor that takes a string hostname attempts to resolve the name to an IP address; the static `createUnresolved()` method allows an instance to be created without attempting this resolution step. The `isUnresolved()` method returns TRUE if the instance was created this way, or if the resolution attempt in the constructor failed. The `get...()` methods provide access to the indicated components, with `getHostName()` providing the name associated with the contained `InetAddress`. The `toString()` method overrides that of `Object` and returns a string consisting of the name associated with the contained address (if known), a '/' (slash), the address in numeric form, a ':' (colon), and the port number. If the `InetSocketAddress` is unresolved, only the `String` with which it was created precedes the colon.

2.2.2 TCP Server

We now turn our attention to constructing a server. The server's job is to set up a communication endpoint and passively wait for connections from clients. The typical TCP server goes through two steps:

1. Construct a `ServerSocket` instance, specifying the local port. This socket listens for incoming connections to the specified port.
2. Repeatedly:
   a. Call the `accept()` method of `ServerSocket` to get the next incoming client connection. Upon establishment of a new client connection, an instance of `Socket` for the new connection is created and returned by `accept()`.
   b. Communicate with the client using the returned `Socket`'s `InputStream` and `OutputStream`.
   c. When finished, close the new client socket connection using the `close()` method of `Socket`.

Our next example, `TCPEchoServer.java`, implements the echo service used by our client program. The server is very simple. It runs forever, repeatedly accepting a connection, receiving and echoing bytes until the connection is closed by the client, and then closing the client socket.

TCPEchoServer.java

```
0  import java.net.*;  // for Socket, ServerSocket, and InetAddress
1  import java.io.*;   // for IOException and Input/OutputStream
2
3  public class TCPEchoServer {
4
5    private static final int BUFSIZE = 32;   // Size of receive buffer
6
7    public static void main(String[] args) throws IOException {
 8
 9     if (args.length != 1)  // Test for correct # of args
10       throw new IllegalArgumentException("Parameter(s): <Port>");
11
12     int servPort = Integer.parseInt(args[0]);
13
14     // Create a server socket to accept client connection requests
15     ServerSocket servSock = new ServerSocket(servPort);
16
17     int recvMsgSize;   // Size of received message
18     byte[] receiveBuf = new byte[BUFSIZE];  // Receive buffer
19
20     while (true) { // Run forever, accepting and servicing connections
21       Socket clntSock = servSock.accept();     // Get client connection
22
23       SocketAddress clientAddress = clntSock.getRemoteSocketAddress();
24       System.out.println("Handling client at " + clientAddress);
25
26       InputStream in = clntSock.getInputStream();
27       OutputStream out = clntSock.getOutputStream();
28
29       // Receive until client closes connection, indicated by -1 return
30       while ((recvMsgSize = in.read(receiveBuf)) != -1) {
31         out.write(receiveBuf, 0, recvMsgSize);
32       }
33       clntSock.close();  // Close the socket.  We are done with this client!
34    }
35    /* NOT REACHED */
36    }
37  }
```

TCPEchoServer.java

1. **Application setup and parameter parsing:** lines 0–12
2. **Server socket creation:** line 15
   servSock listens for client connection requests on the port specified in the constructor.
3. **Loop forever, iteratively handling incoming connections:** lines 20–34
   - **Accept an incoming connection:** line 21
     The sole purpose of a ServerSocket instance is to supply a new, connected Socket instance for each new incoming TCP connection. When the server is ready to handle a client, it calls accept(), which blocks until an incoming connection is made to the ServerSocket's port. (If a connection arrives between the time the server socket is constructed and the call to accept(), the new connection is queued, and in that case accept() returns immediately. See Section 6.4.1 for details of connection establishment.) The accept() method of ServerSocket returns an instance of Socket

that is already connected to the client's remote socket and ready for reading and writing.

- **Report connected client:** lines 23–24
  We can query the newly created `Socket` instance for the address and port of the connecting client. The `getRemoteSocketAddress()` method of `Socket` returns an instance of `InetSocketAddress` that contains the address and port of the client. The `toString()` method of `InetSocketAddress` prints the information in the form "/⟨address⟩:⟨port⟩". (The name part is empty because the instance was created from the address information only.)
- **Get socket input and output streams:** lines 26–27
  Bytes written to this socket's `OutputStream` will be read from the client's socket's `InputStream`, and bytes written to the client's `OutputStream` will be read from this socket's `InputStream`.
- **Receive and repeat data until the client closes:** lines 30–32
  The while loop repeatedly reads bytes (when available) from the input stream and immediately writes the same bytes back to the output stream until the client closes the connection. The `read()` method of `InputStream` fetches up to the maximum number of bytes the array can hold (in this case, BUFSIZE bytes) into the byte array (*receiveBuf*) and returns the number of bytes read. `read()` blocks until data is available and returns −1 if there is no more data available, indicating that the client closed its socket. In the echo protocol, the client closes the connection when it has received the number of bytes that it sent, so in the server we expect to eventually receive a −1 from `read()`. (Recall that in the client, receiving a −1 from `read()` indicates a protocol error, because it can only happen if the server prematurely closed the connection.)

  As previously mentioned, `read()` does not have to fill the entire byte array to return. In fact, it can return after having read only a single byte. This `write()` method of `OutputStream` writes *recvMsgSize* bytes from *receiveBuf* to the socket. The second parameter indicates the offset into the byte array of the first byte to send. In this case, 0 indicates to take bytes starting from the front of *data*. If we had used the form of `write()` that takes only the buffer argument, *all* the bytes in the buffer array would have been transmitted, possibly including bytes that were not received from the client.
- **Close client socket:** line 33
  Closing the socket releases system resources associated with the connection, and is required for servers, because there is a system-specific limit on the number of open `Socket` instances a program can have.

`ServerSocket`: Creation

```
ServerSocket(int localPort)
ServerSocket(int localPort, int queueLimit)
ServerSocket(int localPort, int queueLimit, InetAddress localAddr)
ServerSocket()
```

A TCP endpoint must be associated with a specific port in order for clients to direct their connections to it. The first three constructors create a TCP endpoint that is associated with the specified local port and ready to *accept* incoming connections. Valid port numbers are in the range 0–65,535. (If the port specified is zero, an arbitrary unused port will be picked.) Optionally, the size of the connection queue and the local address can also be set. Note that the maximum queue size may not be a hard limit, and cannot be used to control client population. The local address, if specified, must be an address of one of this host's network interfaces. If the address is not specified, the socket will accept connections to any of the host's IP addresses. This may be useful for hosts with multiple interfaces where the server wants to accept connections on only one of its interfaces.

The fourth constructor creates a ServerSocket that is not associated with any local port; it must be *bound* to a port (see bind() below) before it can be used.

ServerSocket: Operation

```
void bind(int port)
void bind(int port, int queuelimit)
Socket accept()
void close()
```

The bind() methods associate this socket with a local port. A ServerSocket can only be associated with one port. If this instance is already associated with another port, or if the specified port is already in use, an IOException is thrown.

accept() returns a connected Socket instance for the next new incoming connection to the server socket. If no established connection is waiting, accept() blocks until one is established or a timeout occurs.

The close() method closes the socket. After invoking this method, incoming client connection requests for this socket are rejected.

ServerSocket: Getting attributes

```
InetAddress getInetAddress()
SocketAddress getLocalSocketAddress()
int getLocalPort()
```

These return the local address/port of the server socket. Note that, unlike a Socket, a ServerSocket has no associated I/O Streams. It does, however, have other attributes called options, which can be controlled via various methods, as described in Section 4.4.

2.2.3 Input and Output Streams

As illustrated by the examples above, the basic I/O paradigm for TCP sockets in Java is the *stream* abstraction. (The NIO facilities, added in Java 1.4, provide an alternative abstraction, which we will see in Chapter 5.) A stream is simply an ordered sequence of bytes. Java *input streams* support reading bytes, and *output streams* support writing bytes. In our TCP client and server, each `Socket` instance holds an `InputStream` and an `OutputStream` instance. When we write to the output stream of a `Socket`, the bytes can (eventually) be read from the input stream of the `Socket` at the other end of the connection.

`OutputStream` is the abstract superclass of all output streams in Java. Using an `OutputStream`, we can write bytes to, flush, and close the output stream.

`OutputStream`: Operation

```
abstract void write(int data)
void write(byte[ ] data)
void write(byte[ ] data, int offset, int length)
void flush()
void close()
```

The `write()` methods transfer to the output stream a single byte, an entire array of bytes, and the bytes in an array beginning at `offset` and continuing for `length` bytes, respectively. The single-byte method writes the low-order eight bits of the integer argument. These operations, if called on a stream associated with a TCP socket, may block if a lot of data has been sent, but the other end of the connection has not called `read()` on the associated input stream recently. This can have undesirable consequences if some care is not used (see Section 6.2).

The `flush()` method pushes any buffered data out to the output stream. The `close()` method terminates the stream, after which further calls to `write()` will throw an exception.

`InputStream` is the abstract superclass of all input streams. Using an `InputStream`, we can read bytes from and close the input stream.

`InputStream`: Operation

```
abstract int read()
int read(byte[ ] data)
int read(byte[ ] data, int offset, int length)
int available()
void close()
```

The first three methods get transfer data from the stream. The first form places a single byte in the low-order eight bits of the returned `int`. The second form transfers up to *data.length* bytes from the input stream into *data* and returns the number of bytes transferred. The third form does the same, but places data in the array beginning at `offset`, and transfers only up to `length` bytes. If no data is available, but the end-of-stream has not been detected, all the `read()` methods block until at least one byte can be read. All methods return −1 if called when no data is available and end-of-stream has been detected.

The `available()` method returns the number of bytes available for reading at the time it was called. `close()` shuts down the stream, causing further attempts to read to throw an `IOException`.

2.3 UDP Sockets

UDP provides an end-to-end service different from that of TCP. In fact, UDP performs only two functions: 1) it adds another layer of addressing (ports) to that of IP, and 2) it detects some forms of data corruption that may occur in transit and discards any corrupted messages. Because of this simplicity, UDP sockets have some different characteristics from the TCP sockets we saw earlier. For example, UDP sockets do not have to be connected before being used. Where TCP is analogous to telephone communication, UDP is analogous to communicating by mail: you do not have to "connect" before you send a package or letter, but you do have to specify the destination address for each one. Similarly, each message—called a *datagram*—carries its own address information and is independent of all others. In receiving, a UDP socket is like a mailbox into which letters or packages from many different sources can be placed. As soon as it is created, a UDP socket can be used to send/receive messages to/from any address and to/from many different addresses in succession.

Another difference between UDP sockets and TCP sockets is the way that they deal with message boundaries: *UDP sockets preserve them.* This makes receiving an application message simpler, in some ways, than it is with TCP sockets. (This is discussed further in Section 2.3.4.) A final difference is that the end-to-end transport service UDP provides is best-effort: there is no guarantee that a message sent via a UDP socket will arrive at its destination, and messages can be delivered in a different order than they were sent (just like letters sent through the mail). A program using UDP sockets must therefore be prepared to deal with loss and reordering. (We'll provide an example of this later.)

Given this additional burden, why would an application use UDP instead of TCP? One reason is efficiency: if the application exchanges only a small amount of data—say, a single request message from client to server and a single response message in the other direction—TCP's connection establishment phase at least doubles the number of messages (and the number of round-trip delays) required for the communication. Another reason is flexibility: when something other than a reliable byte-stream service is required, UDP provides a minimal-overhead platform on which to implement whatever is needed.

Java programmers use UDP sockets via the classes DatagramPacket and DatagramSocket. Both clients and servers use DatagramSockets to send and receive DatagramPackets.

2.3.1 DatagramPacket

Instead of sending and receiving streams of bytes as with TCP, UDP endpoints exchange self-contained messages, called datagrams, which are represented in Java as instances of DatagramPacket. To send, a Java program constructs a DatagramPacket instance containing the data to be sent and passes it as an argument to the send() method of a DatagramSocket. To receive, a Java program constructs a DatagramPacket instance with preallocated space (a byte[]), into which the contents of a received message can be copied (if/when one arrives), and then passes the instance to the receive() method of a DatagramSocket.

In addition to the data, each instance of DatagramPacket also contains address and port information, the semantics of which depend on whether the datagram is being sent or received. When a DatagramPacket is sent, the address and port identify the destination; for a received DatagramPacket, they identify the source of the received message. Thus, a server can receive into a DatagramPacket instance, modify its buffer contents, then send the same instance, and the modified message will go back to its origin. Internally, a DatagramPacket also has *length* and *offset* fields, which describe the location and number of bytes of message data inside the associated buffer. See the following reference and Section 2.3.4 for some pitfalls to avoid when using DatagramPackets.

DatagramPacket: Creation

```
DatagramPacket(byte[ ] data, int length)
DatagramPacket(byte[ ] data, int offset, int length)
DatagramPacket(byte[ ] data, int length, InetAddress remoteAddr, int remotePort)
DatagramPacket(byte[ ] data, int offset, int length, InetAddress remoteAddr, int remotePort)
DatagramPacket(byte[ ] data, int length, SocketAddress sockAddr)
DatagramPacket(byte[ ] data, int offset, int length, SocketAddress sockAddr)
```

These constructors create a datagram whose data portion is contained in the given byte array. The first two forms are typically used to construct DatagramPackets for receiving because the destination address is not specified (although it could be specified later with setAddress() and setPort(), or setSocketAddress()). The last four forms are typically used to construct DatagramPackets for sending.

Where offset is specified, the data portion of the datagram will be transferred to/from the byte array beginning at the specified position in the array. The length parameter specifies the number of bytes that will be transferred from the byte array when sending, or the maximum number to be transferred when receiving; it may be smaller, but not larger than data.length.

The destination address and port may be specified separately, or together in a SocketAddress.

DatagramPacket: Addressing

```
InetAddress getAddress()
void setAddress(InetAddress address)
int getPort()
void setPort(int port)
SocketAddress getSocketAddress()
void setSocketAddress(SocketAddress sockAddr)
```

In addition to constructors, these methods supply an alternative way to access and modify the address of a DatagramPacket. Note that in addition, the receive() method of DatagramSocket sets the address and port to the datagram sender's address and port.

DatagramPacket: Handling data

```
int getLength()
void setLength(int length)
int getOffset()
byte[ ] getData()
void setData(byte[ ] data)
void setData(byte[ ] buffer, int offset, int length)
```

The first two methods return/set the internal length of the data portion of the datagram. The internal datagram length can be set explicitly either by the constructor or by the setLength() method. Attempting to make it larger than the length of the associated buffer results in an IllegalArgumentException. The receive() method of DatagramSocket uses the internal length in two ways: on input, it specifies the maximum number of bytes of a received message that will be copied into the buffer and on return, it indicates the number of bytes actually placed in the buffer.

getOffset() returns the location in the buffer of the first byte of data to be sent/received. There is no setOffset() method; however, it can be set with setData().

The getData() method returns the byte array associated with the datagram. The returned object is a reference to the byte array that was most recently associated with this DatagramPacket, either by the constructor or by setData(). The length of the returned buffer may be greater than the internal datagram length, so the internal length and offset values should be used to determine the actual received data.

The setData() methods make the given byte array the data portion of the datagram. The first form makes the entire byte array the buffer; the second form makes bytes offset through offset + length − 1 the buffer. The second form always updates the internal offset and length.

2.3.2 UDP Client

A UDP client begins by sending a datagram to a server that is passively waiting to be contacted. The typical UDP client goes through three steps:

1. Construct an instance of DatagramSocket, optionally specifying the local address and port.
2. Communicate by sending and receiving instances of DatagramPacket using the send() and receive() methods of DatagramSocket.
3. When finished, deallocate the socket using the close() method of DatagramSocket.

Unlike a Socket, a DatagramSocket is not constructed with a specific destination address. This illustrates one of the major differences between TCP and UDP. A TCP socket is required to establish a connection with another TCP socket on a specific host and port before any data can be exchanged, and, thereafter, it *only* communicates with that socket until it is closed. A UDP socket, on the other hand, is not required to establish a connection before communication, and each datagram can be sent to or received from a different destination. (The connect() method of DatagramSocket does allow the specification of the remote address and port, but its use is optional.)

Our UDP echo client, UDPEchoClientTimeout.java, sends a datagram containing the string to be echoed and prints whatever it receives back from the server. A UDP echo server simply sends each datagram that it receives back to the client. Of course, a UDP client only communicates with a UDP server. Many systems include a UDP echo server for debugging and testing purposes.

One consequence of using UDP is that datagrams can be lost. In the case of our echo protocol, either the echo request from the client or the echo reply from the server may be lost in the network. Recall that our TCP echo client sends an echo string and then blocks on read() waiting for a reply. If we try the same strategy with our UDP echo client and the echo request datagram is lost, our client will block forever on receive(). To avoid this problem, our client uses the setSoTimeout() method of DatagramSocket to specify a maximum amount of time to block on receive(), so it can try again by resending the echo request datagram. Our echo client performs the following steps:

1. Send the echo string to the server.
2. Block on receive() for up to three seconds, starting over (up to five times) if the reply is not received before the timeout.
3. Terminate the client.

UDPEchoClientTimeout.java

```
import java.net.DatagramSocket;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.io.IOException;
import java.io.InterruptedIOException;

public class UDPEchoClientTimeout {

  private static final int TIMEOUT = 3000;   // Resend timeout (milliseconds)
  private static final int MAXTRIES = 5;     // Maximum retransmissions

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

    if ((args.length < 2) || (args.length > 3)) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");
    }
    InetAddress serverAddress = InetAddress.getByName(args[0]);  // Server address
    // Convert the argument String to bytes using the default encoding
    byte[] bytesToSend = args[1].getBytes();

    int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;

    DatagramSocket socket = new DatagramSocket();

    socket.setSoTimeout(TIMEOUT);  // Maximum receive blocking time (milliseconds)

    DatagramPacket sendPacket = new DatagramPacket(bytesToSend,  // Sending packet
        bytesToSend.length, serverAddress, servPort);

    DatagramPacket receivePacket =                              // Receiving packet
        new DatagramPacket(new byte[bytesToSend.length], bytesToSend.length);

    int tries = 0;      // Packets may be lost, so we have to keep trying
33      boolean receivedResponse = false;
34      do {
35        socket.send(sendPacket);          // Send the echo string
36        try {
37          socket.receive(receivePacket);  // Attempt echo reply reception
38
39          if (!receivePacket.getAddress().equals(serverAddress)) {// Check source
40            throw new IOException("Received packet from an unknown source");
41          }
42          receivedResponse = true;
43        } catch (InterruptedIOException e) {  // We did not get anything
44          tries += 1;
45          System.out.println("Timed out, " + (MAXTRIES - tries) + " more tries...");
46        }
47      } while ((!receivedResponse) && (tries < MAXTRIES));
48
49      if (receivedResponse) {
50        System.out.println("Received: " + new String(receivePacket.getData()));
51      } else {
52        System.out.println("No response -- giving up.");
53      }
54      socket.close();
55    }
56  }
```

UDPEchoClientTimeout.java

1. **Application setup and parameter processing:** lines 0–20
2. **UDP socket creation:** line 22
   This instance of DatagramSocket can send datagrams to any UDP socket. We do not specify a local address or port so some local address and available port will be selected. We could explicitly set them with the setLocalAddress() and setLocalPort() methods or in the constructor, if desired.
3. **Set the socket timeout:** line 24
   The timeout for a datagram socket controls the maximum amount of time (milliseconds) a call to receive() will block. Here we set the timeout to three seconds. Note that timeouts are not precise: the call may block for more than the specified time (but not less).
4. **Create datagram to send:** lines 26–27
   To create a datagram for sending, we need to specify three things: data, destination address, and destination port. For the destination address, we may identify the echo server either by name or IP address. If we specify a name, it is converted to the actual IP address in the constructor.

5. **Create datagram to receive:** lines 29–30
   To create a datagram for receiving, we only need to specify a byte array to hold the datagram data. The address and port of the datagram source will be filled in by `receive()`.
6. **Send the datagram:** lines 32–47
   Since datagrams may be lost, we must be prepared to retransmit the datagram. We loop sending and attempting a receive of the echo reply up to five times.
   - **Send the datagram:** line 35
     `send()` transmits the datagram to the address and port specified in the datagram.
   - **Handle datagram reception:** lines 36–46
     `receive()` blocks until it either receives a datagram or the timer expires. Timer expiration is indicated by an `InterruptedIOException`. If the timer expires, we increment the send attempt count (*tries*) and start over. After the maximum number of tries, the while loop exits without receiving a datagram. If `receive()` succeeds, we set the loop flag *receivedResponse* to TRUE , causing the loop to exit. Since packets may come from anywhere, we check the source address of the recieved datagram to verify that it matches the address of the specified echo server.
7. **Print reception results:** lines 49–53
   If we received a datagram, *receivedResponse* is `true`, and we can print the datagram data.
8. **Close the socket:** line 54

Before looking at the code for the server, let's take a look at the main methods of the `DatagramSocket` class.

`DatagramSocket`: Creation

```
DatagramSocket()
DatagramSocket(int localPort)
DatagramSocket(int localPort, InetAddress localAddr)
```

These constructors create a UDP socket. Either or both of the local port and address may be specified. If the local port is not specified, or is specified as 0, the socket is bound to any available local port. If the local address is not specified, the packet can receive datagrams addressed to any of the local addresses.

`DatagramSocket`: Connection and Closing

```
void connect(InetAddress remoteAddr, int remotePort)
void connect(SocketAddress remoteSockAddr)
void disconnect()
void close()
```

The connect() methods set the remote address and port of the socket. Once connected, the socket can only communicate with the specified address and port; attempting to send a datagram with a different address and port will throw an exception. The socket will only receive datagrams that originated from the specified port and address; datagrams arriving from any other port or address are ignored. *Caveat:* A socket connected to a multicast or broadcast address can only *send* datagrams because a datagram source address is always a unicast address (see Section 4.3). Note that connecting is strictly a local operation because (unlike TCP) there is no end-to-end packet exchange involved. disconnect() unsets the remote address and port, if any. The close() method indicates that the socket is no longer in use; further attempts to send or receive throw an exception.

DatagramSocket: Addressing

```
InetAddress getInetAddress()
int getPort()
SocketAddress getRemoteSocketAddress()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getLocalSocketAddress()
```

The first method returns an InetAddress instance representing the address of the remote socket to which this socket is connected, or null if it is not connected. Similarly, getPort() returns the port number to which the socket is connected, or −1 if it is not connected. The third method returns both address and port conveniently encapsulated in an instance of SocketAddress, or null if unconnected.

The last three methods provide the same service for the *local* address and port. If the socket has not been bound to a local address, getLocalAddress() returns the wildcard ("any local address") address. getLocalPort() always returns a local port number; if the socket was not been bound before the call, the call causes the socket to be bound to any available local port. The getLocalSocketAddress() returns null if the socket is not bound.

DatagramSocket: Sending and receiving

```
void send(DatagramPacket packet)
void receive(DatagramPacket packet)
```

The `send()` method sends the `DatagramPacket`. If connected, the packet is sent to the address to which the socket is connected, unless the `DatagramPacket` specifies a different destination, in which case an exception is thrown. Otherwise, the packet is sent to the destination indicated by the `DatagramPacket`. This method does not block.

The `receive()` method blocks until a datagram is received, and then copies its data into the given `DatagramPacket`. If the socket is connected, the method blocks until a datagram is received from the remote socket to which it is connected.

`DatagramSocket`: Options

```
int getSoTimeout()
void setSoTimeout(int timeoutMillis)
```

These methods return and set, respectively, the maximum amount of time that a `receive()` call will block for this socket. If the timer expires before data is available, an `InterruptedIOException` is thrown. The timeout value is given in milliseconds.

Like `Socket` and `ServerSocket`, the `DatagramSocket` class has many other options. They are described more fully in Section 4.4.

2.3.3 UDP Server

Like a TCP server, a UDP server's job is to set up a communication endpoint and passively wait for clients to initiate communication; however, since UDP is connectionless, UDP communication is initiated by a datagram from the client, without going through a connection setup as in TCP. The typical UDP server goes through three steps:

1. Construct an instance of `DatagramSocket`, specifying the local port and, optionally, the local address. The server is now ready to receive datagrams from any client.
2. Receive an instance of `DatagramPacket` using the `receive()` method of `DatagramSocket`. When `receive()` returns, the datagram contains the client's address so we know where to send the reply.
3. Communicate by sending and receiving `DatagramPackets` using the `send()` and `receive()` methods of `DatagramSocket`.

Our next program example, `UDPEchoServer.java`, implements the UDP version of the echo server. The server is very simple: it loops forever, receiving datagrams and then sending the same datagrams back to the client. Actually, our server only receives and sends back the first 255 (*ECHOMAX*) characters of the datagram; any excess is silently discarded by the socket implementation (see Section 2.3.4).

UDPEchoServer.java

```
 0  import java.io.IOException;
 1  import java.net.DatagramPacket;
 2  import java.net.DatagramSocket;
 3
 4  public class UDPEchoServer {
 5
 6    private static final int ECHOMAX = 255; // Maximum size of echo datagram
 7
 8    public static void main(String[] args) throws IOException {
 9
10      if (args.length != 1) { // Test for correct argument list
11        throw new IllegalArgumentException("Parameter(s): <Port>");
12      }
13
14      int servPort = Integer.parseInt(args[0]);
15
16      DatagramSocket socket = new DatagramSocket(servPort);
17      DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX], ECHOMAX);
18
19      while (true) { // Run forever, receiving and echoing datagrams
20        socket.receive(packet); // Receive packet from client
21        System.out.println("Handling client at " + packet.getAddress().getHostAddress()
22                                            + " on port " + packet.getPort());
23        socket.send(packet); // Send the same packet back to client
24        packet.setLength(ECHOMAX); // Reset length to avoid shrinking buffer
25      }
26      /* NOT REACHED */
27    }
28  }
```

UDPEchoServer.java

1. **Application setup and parameter parsing:** lines 0–14
   UDPEchoServer takes a single parameter, the local port of the echo server socket.
2. **Create and set up datagram socket:** line 16
   Unlike our UDP client, a UDP server must explicitly set its local port to a number known by the client; otherwise, the client will not know the destination port for its echo request datagram. When the server receives the echo datagram from the client, it can find out the client's address and port from the datagram.
3. **Create datagram:** line 17
   UDP messages are contained in datagrams. We construct an instance of DatagramPacket with a buffer of *ECHOMAX* 255 bytes. This datagram will be used both to receive the echo request and to send the echo reply.

4. **Iteratively handle incoming echo requests:** lines 19–25
   The UDP server uses a single socket for all communication, unlike the TCP server, which creates a new socket with every successful `accept()`.
   - **Receive echo request datagram, print source:** lines 20–22
     The `receive()` method of `DatagramSocket` blocks until a datagram is received from a client (unless a timeout is set). There is no connection, so each datagram may come from a different sender. The datagram itself contains the sender's (client's) source address and port.
   - **Send echo reply:** line 23
     packet already contains the echo string and echo reply destination address and port, so the `send()` method of `DatagramSocket` can simply transmit the datagram previously received. Note that when we receive the datagram, we interpret the datagram address and port as the *source* address and port, and when we send a datagram, we interpret the datagram's address and port as the *destination* address and port.
   - **Reset buffer size:** line 24
     The internal length of *packet* was set to the length of the message just processed, which may have been smaller than the original buffer size. If we do not reset the internal length before receiving again, the next message will be truncated if it is longer than the one just received.

2.3.4 Sending and Receiving with UDP Sockets

In this section we consider some of the differences between communicating with UDP sockets compared to TCP. A subtle but important difference is that UDP preserves message boundaries. Each call to `receive()` on a `DatagramSocket` returns data from at most one call to `send()`. Moreover, different calls to `receive()` will never return data from the same call to `send()`.

When a call to `write()` on a TCP socket's output stream returns, all the caller knows is that the data has been copied into a buffer for transmission; the data may or may not have actually been transmitted yet. (This is covered in more detail in Chapter 6.) UDP, however, does not provide recovery from network errors and, therefore, does not buffer data for possible retransmission. This means that by the time a call to `send()` returns, the message has been passed to the underlying channel for transmission and is (or soon will be) on its way out the door.

Between the time a message arrives from the network and the time its data is returned via `read()` or `receive()`, the data is stored in a *first-in, first-out (FIFO)* queue of received data. With a connected TCP socket, all received-but-not-yet-delivered bytes are treated as one continuous sequence of bytes (see Chapter 6). For a UDP socket, however, the received data may have come from different senders. A UDP socket's received data is kept in a queue of messages, each with associated information identifying its source. A call to `receive()` will never return more than one message. However, if `receive()` is called with a `DatagramPacket` containing a buffer of size *n*, and the size of the first message in the receive queue exceeds *n*, only the first *n* bytes of the message are returned. The remaining bytes are quietly discarded, with no indication to the receiving program that information has been lost!

For this reason, a receiver should always supply a `DatagramPacket` that has enough space to hold the largest message allowed by the application protocol at the time it calls to `receive()`. This technique will guarantee that no data will be lost. The maximum amount of data that can be transmitted in a `DatagramPacket` is 65,507 bytes—the largest payload that can be carried in a UDP datagram. Thus it's always safe to use a packet that has an array of size 65,600 or so.

It is also important to remember here that each instance of `DatagramPacket` has an internal notion of message length that may be changed whenever a message is received into that instance (to reflect the number of bytes in the received message). Applications that call `receive()` more than once with the same instance of `DatagramPacket` should explicitly reset the internal length to the actual buffer length before each subsequent call to `receive()`.

Another potential source of problems for beginners is the `getData()` method of `DatagramPacket`, which always returns the entire original buffer, ignoring the internal offset and length values. Receiving a message into the `DatagramPacket` only modifies those locations of the buffer into which message data was placed. For example, suppose *buf* is a byte array of size 20, which has been initialized so that each byte contains its index in the array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Suppose also that *dg* is a `DatagramPacket`, and that we set *dg*'s buffer to be the middle 10 bytes of *buf*:

```
dg.setData(buf,5,10);
```

Now suppose that *dgsocket* is a `DatagramSocket`, and that somebody sends an 8-byte message containing

41	42	43	44	45	46	47	48

to *dgsocket*. The message is received into *dg*:

```
dgsocket.receive(dg);
```

Now, calling *dg*.`getData()` returns a reference to the original byte array *buf*, whose contents are now

0	1	2	3	4	41	42	43	44	45	46	47	48	13	14	15	16	17	18	19

Note that only bytes 5–12 of *buf* have been modified and that, in general, the application needs to use `getOffset()` and `getData()` to access just the received data. One possibility is to copy the received data into a separate byte array, like this:

```
byte[] destBuf = new byte[dg.getLength()];
System.arraycopy(dg.getData(), dg.getOffset(), destBuf, 0, destBuf.length);
```

As of Java 1.6, we can do it in one step using the convenience method `Arrays.copyOfRange()`:

```
byte[] destBuf = Arrays.copyOfRange(dg.getData(),dg.getOffset(),
                     dg.getOffset()+dg.getLength());
```

We didn't have to do this copying in `UDPEchoServer.java` because the server did not read the data from the `DatagramPacket` at all.

2.4 Exercises

1. For `TCPEchoServer.java`, we explicitly specify the port to the socket in the constructor. We said that a socket must have a port for communication, yet we do not specify a port in `TCPEchoClient.java`. How is the echo client's socket assigned a port?
2. When you make a phone call, it is usually the callee that answers with "Hello." What changes to our client and server examples would be needed to implement this?
3. What happens if a TCP server never calls `accept()`? What happens if a TCP client sends data on a socket that has not yet been `accept()`ed at the server?
4. Servers are supposed to run for a long time without stopping—therefore, they must be designed to provide good service no matter what their clients do. Examine the server examples (`TCPEchoServer.java` and `UDPEchoServer.java`) and list anything you can think of that a client might do to cause it to give poor service to other clients. Suggest improvements to fix the problems that you find.
5. Modify `TCPEchoServer.java` to read and write only a single byte at a time, sleeping one second between each byte. Verify that `TCPEchoClient.java` requires multiple reads to successfully receive the entire echo string, even though it sent the echo string with one `write()`.
6. Modify `TCPEchoServer.java` to read and write a single byte and then close the socket. What happens when the `TCPEchoClient` sends a multibyte string to this server? What is happening? (Note that the response could vary by OS.)
7. Modify `UDPEchoServer.java` so that it only echoes every other datagram it receives. Verify that `UDPEchoClientTimeout.java` retransmits datagrams until it either receives a reply or exceeds the number of retries.
8. Modify `UDPEchoServer.java` so that *ECHOMAX* is much shorter (say, 5 bytes). Then use `UDPEchoClientTimeout.java` to send an echo string that is too long. What happens?
9. Verify experimentally the size of the largest message you can send and receive using a `DatagramPacket`.
10. While `UDPEchoServer.java` explicitly specifies its local port in the constructor, we do not specify the local port in `UDPEchoClientTimeout.java`. How is the UDP echo client's socket given a port number? *Hint:* The answer is different for TCP.