- **Introduction:**
- Networking is the concept of connecting multiple remote or local devices together. Java program communicates over the network at **application layer.**
  All the Java networking classes and interfaces use java.net package. These classes and interfaces provide the functionality to develop system-independent network communication.

  The java.net package provides the functionality for two common protocols:

  TCP (Transmission Control Protocol)
  TCP is a connection based protocol that provides a reliable flow of data between two devices. This protocol provides the reliable connections between two applications so that they can communicate easily. It is a connection based protocol.

  UDP (User Datagram Protocol)
  UDP protocol sends independent packets of data, called datagram from one computer to another with no guarantee of arrival. It is not connection based protocol.
- Networking Terminology
- **i) Request and Response**
  When an input data is sent to an application via network, it is called **request**.
  The output data coming out from the application back to the client program is called **response**.

  ii) Protocol
  A **protocol** is basically a set of rules and guidelines which provides the instructions to send request and receive response over the network.
  For example: TCP, UDP, SMTP, FTP etc.

  iii) IP Address
  IP Address stands for Internet protocol address. It is an identification number that is assigned to a node of a computer in the network.
  For example: 192.168.2.01
  Range of the IP Address
  0.0.0.0 to 255.255.255.255

  iv) Port Number
  The **port number** is an identification number of server software. The port number is unique for different applications. It is a 32-bit positive integer number having between ranges 0 to 65535.

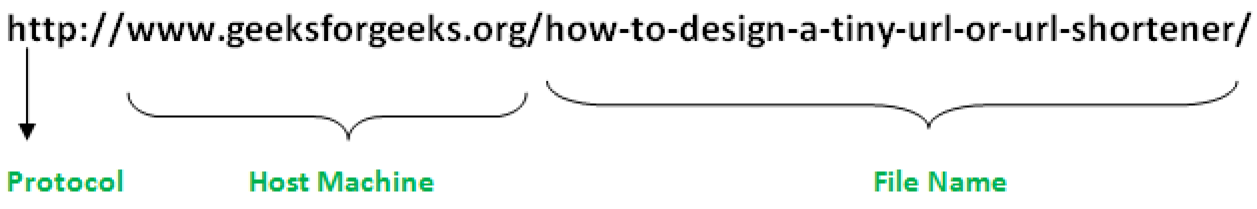
  v) Socket
  Socket is a listener through which computer can receive requests and responses. It is an endpoint of two way communication link. Every server or programs runs on the different computers that has a socket and is bound to the specific port number.

- **Manipulating URL's** :

URL class in Java with Examples

The URL class is the gateway to any of the resource available on the internet. A Class URL represents a Uniform Resource Locator, which is a pointer to a "resource" on the World Wide Web. A resource can point to a simple file or directory, or it can refer to a more complicated object, such as a query to a database or to a search engine

What is a URL?
As many of you must be knowing that Uniform Resource Locator-URL is a string of text that identifies all the resources on Internet, telling us the address of the resource, how to communicate with it and retrieve something from it.
A Simple URL looks like:


http://www.geeksforgeeks.org/how-to-design-a-tiny-url-or-url-shortener/
Protocol
Host Machine
File Name

Components of a URL:-
A URL can have many forms. The most general however follows three-components system-

1. **Protocol:** HTTP is the protocol here
2. **Hostname:** Name of the machine on which the resource lives.
3. **File Name:** The path name to the file on the machine.
4. **Port Number:** Port number to which to connect (typically optional).

Some constructors for URL class:-

1. **URL(String address) throws MalformedURLException:** It creates a URL object from the specified String.
2. **URL(String protocol, String host, String file):** Creates a URL object from the specified protcol, host, and file name.
3. **URL(String protocol, String host, int port, String file):** Creates a URL object from protocol, host, port and file name.
4. **URL(URL context, String spec):** Creates a URL object by parsing the given spec in the given context.
5. **URL(String protocol, String host, int port, String file, URLStreamHandler handler):-**
   Creates a URL object from the specified protocol, host, port number, file, and handler.
6. **URL(URL context, String spec, URLStreamHandler handler):-**
   Creates a URL by parsing the given spec with the specified handler within a specified context.

- **Reading File on a Web Server:**

Reading from an HTTP server

Given the proper permissions, you can easily read files directly from the filesystem of an HTTP server. In this section we'll show how to read a file from a web server just by specifying the URL of the desired file.

I've created a method named getToDoList() to perform the applet's read function. This method is called once when the applet is first started, and again each time the user clicks on the Refresh button.

The code for the getToDoList() method is shown in **Listing 2**. First we create an object named ***url*** using the URL class. In this example the URL is hard-wired into the constructor method for the URL class. (Note that for more robust applet's that you want to share with others, you'll want to use the getCodeBase() method and a relative file path instead of the hard-coded path I'm using.)

Next, we create a URLConnection object named ***urlConnection*** by invoking url.openConnection(). Then, this connection is defined to be an input connection by invoking setDoInput() method of the URLConnection class. To make sure that we get a real copy of the data file and not a cached file, we also invoke setUsesCaches(false).

```
void getToDoList ()

{

 try

  {

   URL              url;

   URLConnection      urlConn;

   DataInputStream    dis;



   url = new URL("http://webserver.our-intranet.com/ToDoList/ToDoList.txt");
   // Note:  a more portable URL:

   //url = new URL(getCodeBase().toString() + "/ToDoList/ToDoList.txt");



   urlConn = url.openConnection();

   urlConn.setDoInput(true);

   urlConn.setUseCaches(false);



   dis = new DataInputStream(urlConn.getInputStream());

   String s;



   toDoList.clear();



   while ((s = dis.readLine()) != null)

   {

     toDoList.addItem(s);

   }

     dis.close();

   }
```

```
   catch (MalformedURLException mue) {}

   catch (IOException ioe) {}

  }

}
```

Listing 2: The getToDoList() method retrieves the ToDoList.txt file from the HTTP web server.

Once the initial communication parameters are configured, the data file is read from the web server using the URLConnection's getInputStream() method. The stream is converted to a DataInputStream as it's read in.

As each line of data is read, it's added to the on-screen List component using the addItem() method of the *List* component. Also note that because the getToDoList() method can be called any number of times when the user clicks on the Refresh button, it's necessary to call the clear() method of the ***toDoList*** object before adding any new text to it. Finally, you should close the stream when you're finished reading from it.

Assuming that you have read permission for the URL you're trying to read from, that's all you need to do to read from the file, and add it's contents to the List component.

Java Socket Programming

Socket provides an endpoint of two way communication link using TCP protocol. Java socket can be connection oriented or connection less. TCP provides two way communication, it means data can be sent across both the sides at same time.

Socket Class

The **java.net.Socket** class is used to create a socket so that both the client and the server can communicate with each other easily. A **socket** is an endpoint for communication between two computers. The **Socket** class inherits the **Object** class and implements the **Closeable** interface.

Socket Class Constructors

Constructor	Description
Socket()	Creates an unconnected socket, with the system-default type of SocketImpl.
public Socket(InetAddress address, int port)	Creates a stream socket with specified IP address to the specified port number.
public Socket(InetAddress host, int port, boolean stream)	Uses the DatagramSocket.
public Socket(InetAddress address, int port, InetAddress localAddr, int local port)	Creates a connection with specified remote address and remote port.
public Socket(Proxy, proxy)	Creates a connectionless socket specifying the type of proxy.
protected Socket(SocketImpl impl)	Creates a connectionless Socket with a user-specified SocketImpl.

ServerSocket Class

Socket class is used to create socket and send the request to the server. Java ServerSocket class waits for request to come over the network. It works on the basis of request and then returns a result to the request. It implements the Closeable interface.

ServerSocket Class Constructors

Constructor	Description
ServerSocket()	Creates an unbound server socket.
ServerSocket(int port)	Creates a server socket, bound to the specified port.
ServerSocket(int port, int backlog)	Creates a server socket, bound to the specified port, with specified local port.
ServerSocket(int port, int backlog, inetAddress bindAddrs)	Creates a server socket, bound to specified port, listen backlog, and IP address.

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols −

- **TCP** − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP** − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This chapter gives a good understanding on the following two subjects −

- **Socket Programming** − This is the most widely used concept in Networking and it has been explained in very detail.
- **URL Processing** − This would be covered separately. Click here to learn about URL Processing in Java language.

Socket Programming

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets −

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.

- **Establishing a simple server using Stream Sockets:**

Server Programming

Establish a Socket Connection

To write a server application two sockets are needed.

- A ServerSocket which waits for the client requests (when a client makes a new Socket())
- A plain old Socket socket to use for communication with the client.

Communication

getOutputStream() method is used to send the output through the socket.

Close the Connection

After finishing, it is important to close the connection by closing the socket as well as input/output streams.

```
// A Java program for a Server
import java.net.*;
import java.io.*;

public class Server
{
   //initialize socket and input stream
   private Socket          socket   = null;
   private ServerSocket    server   = null;
   private DataInputStream in       =  null;

   // constructor with port
   public Server(int port)
   {
      // starts server and waits for a connection
      try
      {
         server = new ServerSocket(port);
         System.out.println("Server started");

         System.out.println("Waiting for a client ...");

         socket = server.accept();
```

```
            System.out.println("Client accepted");

            // takes input from the client socket
            in = new DataInputStream(
                new BufferedInputStream(socket.getInputStream()));

            String line = "";

            // reads message from client until "Over" is sent
            while (!line.equals("Over"))
            {
                try
                {
                    line = in.readUTF();
                    System.out.println(line);

                }
                catch(IOException i)
                {
                    System.out.println(i);
                }
            }
            System.out.println("Closing connection");

            // close connection
            socket.close();
            in.close();
        }
        catch(IOException i)
        {
            System.out.println(i);
        }
    }

    public static void main(String args[])
    {
        Server server = new Server(5000);
    }
}
```

Important Points

- Server application makes a ServerSocket on a specific port which is 5000. This starts our Server listening for client requests coming in for port 5000.
- Then Server makes a new Socket to communicate with the client.

```
socket = server.accept()
```

- The accept() method blocks(just sits there) until a client connects to the server.
- Then we take input from the socket using getInputStream() method. Our Server keeps receiving messages until the Client sends “Over”.

- After we're done we close the connection by closing the socket and the input stream.
- To run the Client and Server application on your machine, compile both of them. Then first run the server application and then run the Client application.

To run on Terminal or Command Prompt
Open two windows one for Server and another for Client

1. First run the Server application as ,

```
$ java Server
```

Server started
Waiting for a client …

2. Then run the Client application on another terminal as,

```
$ java Client
```

It will show – Connected and the server accepts the client and shows,

Client accepted

3. Then you can start typing messages in the Client window. Here is a sample input to the Client

```
Hello

I made my first socket connection
```

Over

Which the Server simultaneously receives and shows,

```
Hello

I made my first socket connection

Over
```

Closing connection

Notice that sending "Over" closes the connection between the Client and the Server just like said before.

If you're using Eclipse, NetBeans or likes of such-

1. Compile both of them on two different terminals or tabs
2. Run the Server program first
3. Then run the Client program
4. Type messages in the Client Window which will be received and showed by the Server Window simultaneously.
5. Type Over to end.

- <u>**Establishing a simple Client using Stream Sockets:**</u>

Client Side Programming

Establish a Socket Connection

To connect to other machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port.The java.net.Socket class represents a Socket. To open a socket:

```
Socket socket = new Socket("127.0.0.1", 5000)
```

- First argument – **IP address of Server**. (127.0.0.1 is the IP address of localhost, where code will run on single stand-alone machine).
- Second argument – **TCP Port**. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

Communication
To communicate over a socket connection, streams are used to both input and output the data.

Closing the connection
The socket connection is closed explicitly once the message to server is sent.

In the program, Client keeps reading input from user and sends to the server until "Over" is typed.

Java Implementation

```
// A Java program for a Client
import java.net.*;
import java.io.*;

public class Client
{
   // initialize socket and input output streams
   private Socket socket            = null;
   private DataInputStream  input   = null;
   private DataOutputStream out     = null;

   // constructor to put ip address and port
   public Client(String address, int port)
   {
      // establish a connection
      try
      {
         socket = new Socket(address, port);
         System.out.println("Connected");

         // takes input from terminal
         input  = new DataInputStream(System.in);

         // sends output to the socket
         out    = new DataOutputStream(socket.getOutputStream());
      }
      catch(UnknownHostException u)
      {
         System.out.println(u);
      }
      catch(IOException i)
      {
            System.out.println(i);
        }

        // string to read message from input
        String line = "";

        // keep reading until "Over" is input
        while (!line.equals("Over"))
        {
            try
            {
                line = input.readLine();
                out.writeUTF(line);
            }
            catch(IOException i)
            {
                System.out.println(i);
            }
        }

        // close the connection
        try
        {
            input.close();
            out.close();
            socket.close();
        }
        catch(IOException i)
        {
            System.out.println(i);
        }
    }

    public static void main(String args[])
    {
        Client client = new Client("127.0.0.1", 5000);
    }
}
```

- **<u>Client server interaction with stream socket connections:</u>**

Socket Client Example

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

Example

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;

public class GreetingClient {

  public static void main(String [] args) {
    String serverName = args[0];
    int port = Integer.parseInt(args[1]);
    try {
      System.out.println("Connecting to " + serverName + " on port " + port);
      Socket client = new Socket(serverName, port);

      System.out.println("Just connected to " + client.getRemoteSocketAddress());
      OutputStream outToServer = client.getOutputStream();
      DataOutputStream out = new DataOutputStream(outToServer);

      out.writeUTF("Hello from " + client.getLocalSocketAddress());
      InputStream inFromServer = client.getInputStream();
      DataInputStream in = new DataInputStream(inFromServer);

      System.out.println("Server says " + in.readUTF());
      client.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Socket Server Example

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument −

Example

```
// File Name GreetingServer.java
import java.net.*;
import java.io.*;

public class GreetingServer extends Thread {
  private ServerSocket serverSocket;

  public GreetingServer(int port) throws IOException {
    serverSocket = new ServerSocket(port);
    serverSocket.setSoTimeout(10000);
  }

  public void run() {
      while(true) {
         try {
            System.out.println("Waiting for client on port " +
               serverSocket.getLocalPort() + "...");
            Socket server = serverSocket.accept();

            System.out.println("Just connected to " + server.getRemoteSocketAddress());
            DataInputStream in = new DataInputStream(server.getInputStream());

            System.out.println(in.readUTF());
            DataOutputStream out = new DataOutputStream(server.getOutputStream());
            out.writeUTF("Thank you for connecting to " + server.getLocalSocketAddress()
               + "\nGoodbye!");
            server.close();

         } catch (SocketTimeoutException s) {
            System.out.println("Socket timed out!");
            break;
         } catch (IOException e) {
            e.printStackTrace();
            break;
         }
      }
   }

   public static void main(String [] args) {
      int port = Integer.parseInt(args[0]);
      try {
         Thread t = new GreetingServer(port);
         t.start();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Compile the client and the server and then start the server as follows −

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

Check the client program as follows −

Output

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

- **Connectionless Client-Server interaction with Datagrams:**

- Connection-oriented transmission is like the telephone system in which you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for the duration of your phone call, even when you are not talking.
- Connectionless transmission with datagrams is more like the way mail is carried via the postal service. If a large message will not fit in one envelope, you break it into separate message pieces that you place in separate, sequentially numbered envelopes. Each of the letters is then mailed at the same time. The letters could arrive in order, out of order or not at all (the last case is rare, but it does happen). The person at the receiving end reassembles the message pieces into sequential order before attempting to make sense of the message. If your message is small enough to fit in one envelope, you need not worry about the "out-of-sequence" problem, but it is still possible that your message might not arrive. One difference between datagrams and postal mail is that duplicates of datagrams can arrive at the receiving computer.
- Figure 24.9Fig. 24.12 use datagrams to send packets of information via the User Datagram Protocol (UDP) between a client application and a server application. In the Client application (Fig. 24.11), the user types a message into a textfield and presses **Enter**. The program converts the message into a byte array and places it in a datagram packet that is sent to the server. The Server (Fig. 24.9) receives the packet and displays the information in it, then **echoes the packet back to the client**. Upon receiving the packet, the client displays the information it contains.

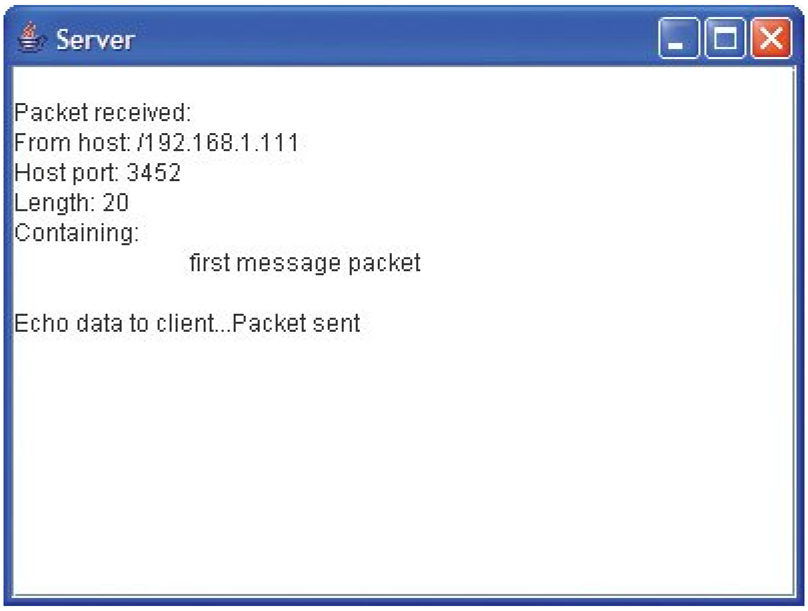
Figure 24.9. Server side of connectionless client/server computing with datagrams.

```
// Fig. 24.9: Server.java
// Server that receives and sends packets from/to a client.
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;

public class Server extends JFrame
{
   private JTextArea displayArea; // displays packets received
   private DatagramSocket socket; // socket to connect to client

   // set up GUI and DatagramSocket
   public Server()
   {
      super( "Server" );

      displayArea = new JTextArea(); // create displayArea
      add( new JScrollPane( displayArea ), BorderLayout.CENTER );
      setSize( 400, 300 ); // set size of window
      setVisible( true ); // show window

      try // create DatagramSocket for sending and receiving packets
      {
        socket = new DatagramSocket( 5000 );
      } // end try
      catch ( SocketException socketException )
      {
        socketException.printStackTrace();
        System.exit( 1 );
      } // end catch
    } // end Server constructor

    // wait for packets to arrive, display data and echo packet to client
    public void waitForPackets()
    {
      while ( true )
      {
        try // receive packet, display contents, return copy to client
        {
          byte data[] = new byte[ 100 ]; // set up packet
          DatagramPacket receivePacket =
            new DatagramPacket( data, data.length );

          socket.receive( receivePacket ); // wait to receive packet

          // display information from received packet
          displayMessage( "\nPacket received:" +
            "\nFrom host: " + receivePacket.getAddress() +
            "\nHost port: " + receivePacket.getPort() +
            "\nLength: " + receivePacket.getLength() +
            "\nContaining:\n\t" + new String( receivePacket.getData(),
              0, receivePacket.getLength() ) );

          sendPacketToClient( receivePacket ); // send packet to client
        } // end try
        catch ( IOException ioException )
        {
          displayMessage( ioException.toString() + "\n" );
          ioException.printStackTrace();
        } // end catch
      } // end while
    } // end method waitForPackets

    // echo packet to client
    private void sendPacketToClient( DatagramPacket receivePacket )
      throws IOException
    {
      displayMessage( "\n\nEcho data to client..." );

      // create packet to send
      DatagramPacket sendPacket = new DatagramPacket(
         receivePacket.getData(), receivePacket.getLength(),
         receivePacket.getAddress(), receivePacket.getPort() );

      socket.send( sendPacket ); // send packet to client
      displayMessage( "Packet sent\n" );
   } // end method sendPacketToClient

   // manipulates displayArea in the event-dispatch thread
   private void displayMessage( final String messageToDisplay )
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // updates displayArea
            {
               displayArea.append( messageToDisplay ); // display message
            } // end method run
         } // end anonymous inner class
      ); // end call to SwingUtilities.invokeLater
   } // end method displayMessage
} // end class Server
```

Figure 24.10. Class that tests the Server:

```
// Fig. 24.10: ServerTest.java
// Tests the Server class.
import javax.swing.JFrame;

public class ServerTest
{
   public static void main( String args[] )
   {
      Server application = new Server(); // create server
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.waitForPackets(); // run server application
   } // end main
} // end class ServerTest
```

Server window after packet of data is received from Client



```
Packet received:
From host: /192.168.1.111
Host port: 3452
Length: 20
Containing:
            first message packet

Echo data to client...Packet sent
```

Figure 24.11. Client side of connectionless client/server computing with datagrams.

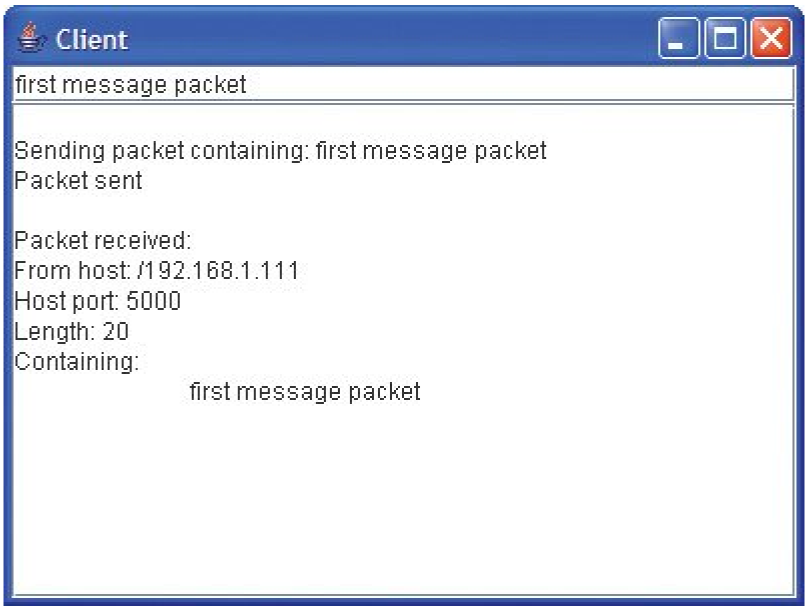
```
// Fig. 24.11: Client.java
// Client that sends and receives packets to/from a server.
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class Client extends JFrame
{
  private JTextField enterField; // for entering messages
  private JTextArea displayArea; // for displaying messages
  private DatagramSocket socket; // socket to connect to server

  // set up GUI and DatagramSocket
  public Client()
  {
    super( "Client" );

    enterField = new JTextField( "Type message here" );
    enterField.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent event )
            {
               try // create and send packet
               {
                  // get message from textfield
                  String message = event.getActionCommand();
                  displayArea.append( "\nSending packet containing: " +
                     message + "\n" );

                  byte data[] = message.getBytes(); // convert to bytes

                  // create sendPacket
                  DatagramPacket sendPacket = new DatagramPacket( data,
                     data.length, InetAddress.getLocalHost(), 5000 );

                  socket.send( sendPacket ); // send packet
                  displayArea.append( "Packet sent\n" );
                  displayArea.setCaretPosition(
                     displayArea.getText().length() );
               } // end try
               catch ( IOException ioException )
               {
                  displayMessage( ioException.toString() + "\n" );
                  ioException.printStackTrace();
               } // end catch
            } // end actionPerformed
         } // end inner class
      ); // end call to addActionListener

      add( enterField, BorderLayout.NORTH );

      displayArea = new JTextArea();
      add( new JScrollPane( displayArea ), BorderLayout.CENTER );

      setSize( 400, 300 ); // set window size
      setVisible( true ); // show window

      try // create DatagramSocket for sending and receiving packets
      {
         socket = new DatagramSocket();
      } // end try
      catch ( SocketException socketException )
      {
         socketException.printStackTrace();
         System.exit( 1 );
      } // end catch
   } // end Client constructor

```

```
// wait for packets to arrive from Server, display packet contents
public void waitForPackets()
{
  while ( true )
  {
    try // receive packet and display contents
    {
      byte data[] = new byte[ 100 ]; // set up packet
      DatagramPacket receivePacket = new DatagramPacket(
        data, data.length );

      socket.receive( receivePacket ); // wait for packet

      // display packet contents
      displayMessage( "\nPacket received:" +
        "\nFrom host: " + receivePacket.getAddress() +
        "\nHost port: " + receivePacket.getPort() +
        "\nLength: " + receivePacket.getLength() +
        "\nContaining:\n\t" + new String( receivePacket.getData(),
          0, receivePacket.getLength() ) );
    } // end try
    catch ( IOException exception )
    {
      displayMessage( exception.toString() + "\n" );
      exception.printStackTrace();
    } // end catch
  } // end while
} // end method waitForPackets

// manipulates displayArea in the event-dispatch thread
private void displayMessage( final String messageToDisplay )
{
  SwingUtilities.invokeLater(
    new Runnable()
    {
      public void run() // updates displayArea
      {
        displayArea.append( messageToDisplay );
      } // end method run
    } // end inner class
  ); // end call to SwingUtilities.invokeLater
} // end method displayMessage
} // end class Client
```

Figure 24.12. Class that tests the Client.

```
// Fig. 24.12: ClientTest.java
// Tests the Client class.
import javax.swing.JFrame;

public class ClientTest
{
   public static void main( String args[] )
   {
      Client application = new Client(); // create client
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.waitForPackets(); // run client application
   } // end main
}  // end class ClientTest
```

Client window after sending packet to Server and receiving packet back from Server



```
first message packet

Sending packet containing: first message packet
Packet sent

Packet received:
From host: /192.168.1.111
Host port: 5000
Length: 20
Containing:
            first message packet
```

Server Class

Class Server (Fig. 24.9) declares two **DatagramPackets** that the server uses to send and receive information and one **DatagramSocket** that sends and receives the packets.
The Server constructor (lines 1937) creates the graphical user interface in which the packets of information will be displayed. Line 30 creates the DatagramSocket in a TRy block. Line 30 uses the DatagramSocket constructor that takes an integer port number argument (5000 in this example) to bind the server to a port where it can receive packets from clients. Clients sending packets to this Server specify the same port number in the packets they send.
A **SocketException** is thrown if the DatagramSocket constructor fails to bind the DatagramSocket to the specified port.

Common Programming Error 24.2

Specifying a port that is already in use or specifying an invalid port number when creating a DatagramSocket results in a SocketException.

Server method waitForPackets (lines 4068) uses an infinite loop to wait for packets to arrive at the Server. Lines 4748 create a DatagramPacket in which a received packet of information can be stored. The DatagramPacket constructor for this purpose receives two argumentsa byte array in which the data will be stored and the length of the array. Line 50 uses DatagramSocket method **receive** to wait for a packet to arrive at the Server. Method receive blocks until a packet arrives, then stores the packet in its DatagramPacket argument. The method throws an IOException if an error occurs while receiving a packet.

When a packet arrives, lines 5358 call method displayMessage (declared at lines 8697) to append the packet's contents to the textarea. DatagramPacket method **getAddress** (line 54) returns an InetAddress object containing the host name of the computer from which the packet was sent. Method **getPort** (line 55) returns an integer specifying the port number through which the host computer sent the packet. Method **getLength** (line 56) returns an integer representing the number of bytes of data sent. Method **getData** (line 57) returns a byte array containing the data. Lines 5758 initialize a String object using a three-argument constructor that takes a byte array, the offset and the length. This String is then appended to the text to display.

After displaying a packet, line 60 calls method sendPacketToClient (declared at lines 7183) to create a new packet and send it to the client. Lines 7779 create a DatagramPacket and pass four arguments to its constructor. The first argument specifies the byte array to send. The second argument specifies the number of bytes to send. The third argument specifies the client computer's Internet address, to which the packet will be sent. The fourth argument specifies the port where the client is waiting to receive packets. Line 81 sends the packet over the network. Method **send** of DatagramSocket throws an IOException if an error occurs while sending a packet.

Client Class

Class Client (Fig. 24.11) works similarly to class Server, except that the Client sends packets only when the user types a message in a textfield and presses the **Enter** key. When this occurs, the program calls method actionPerformed (lines 3257), which converts the string the user entered into a byte array (line 41). Lines 4445 create a DatagramPacket and initialize it with the byte array, the length of the string that was entered by the user, the IP address to which the packet is to be sent (InetAddress.getLocalHost() in this example) and the port number at which the Server is waiting for packets (5000 in this example). Line 47 sends the packet. Note that the client in this example must know that the server is receiving packets at port 5000otherwise, the server will not receive the packets.

Note that the DatagramSocket constructor call (line 71) in this application does not specify any arguments. This no-argument constructor allows the computer to select the next available port number for the DatagramSocket. The client does not need a specific port number, because the server receives the client's port number as part of each DatagramPacket sent by the client. Thus, the server can send packets back to the same computer and port number from which it receives a packet of information.

Client method waitForPackets (lines 81107) uses an infinite loop to wait for packets from the server. Line 91 blocks until a packet arrives. This does not prevent the user from sending a packet, because the GUI events are handled in the event-dispatch thread. It only prevents the while loop from continuing until a packet arrives at the Client. When a packet arrives, line 91 stores it in receivePacket, and lines 9499 call method displayMessage (declared at lines 110121) to display the packet's contents in the textarea.