Chapter 4
Network Programming

Outline

4.1. Overview of sockets

4.2. Establishing Connections

4.3. TCP Clients and Servers

4.4. UDP Clients and Servers

4.5. Secure Sockets Layer

Networking Basics

- *Computer networking is to send and receive messages among computers on the Internet.*
- To browse the Web or send email, your computer must be connected to the Internet.
- Your computer can connect to the Internet through an Internet Service Provider (ISP) using a dialup, DSL, or cable modem, or through a local area network (LAN).
- Java Networking is a concept of connecting two or more computing devices together so that we can share resources.
- Advantage of Java Networking
  - sharing resources
  - centralize software management
- When a computer needs to communicate with another computer, it needs to know an IP address.

Networking Basics

- **Internet Protocol (IP)** addresses
  - Uniquely identifies the computer on the Internet. or
  - IP address is a unique number assigned to a node of a network.
  - It is a logical address that can be changed.
  - Every host on Internet has a unique IP address
  - An IP address consists of four dotted decimal numbers ranging from **0** and **255**, such as

    `143.89.40.46, 203.184.197.198`

    `203.184.197.196, 203.184.197.197, 127.0.0.`
  - Since it is difficult to remember IP address, there is a special server called Domain Name Server(DNS), which translates hostname to IP address
  - Example: **DomainName**: www.aastu.edu.et , www.google.com, localhost

    IP Addresess: **10.1.25.16** **216.58.207.4** **127.0.0.1**
  - One domain name can correspond to multiple internet addresses:
    - www.yahoo.com:

      66.218.70.49; 66.218.70.50; 66.218.71.80; 66.218.71.84; …
  - Domain Name Server (DNS) maps names to numbers

Networking Basics

- **A protocols** is a set of rules that facilitate communications between machines or hosts.
- Examples:
  - HTTP: HyperText Transfer Protocol
  - FTP: File Transfer Protocol
  - SMTP: Simple Message Transfer Protocol
  - TCP: Transmission Control Protocol
  - UDP: User Datagram Protocol, good for, e.g., video delivery)
- TCP:
  - Connection-oriented protocol
  - enables two hosts to establish a connection and exchange streams of data.
  - Acknowledgement is send by the receiver. It is reliable but slow
  - Uses Stream-based communications
  - guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.
- UDP:
  - Enables connectionless communication
  - Acknowledgement is not sent by the receiver. So it is not reliable but fast. Uses packet-based communications.
  - Cannot guarantee lossless transmission.

Networking Basics

Port Number

- The port number is used to uniquely identify different applications.
- It acts as a communication endpoint between applications.
- The port number is associated with the IP address for communication between two applications.
- Port numbers are ranging from 0 to 65536, but port numbers 0 to 1024 are reserved for privileged services.
- Many standard port numbers are pre-assigned
  - time of day 13, ftp 21, telnet 23, smtp 25, http 80
- You can choose any port number that is not currently used by other programs.
- IP address + port number = "phone number " for service or application

MAC Address

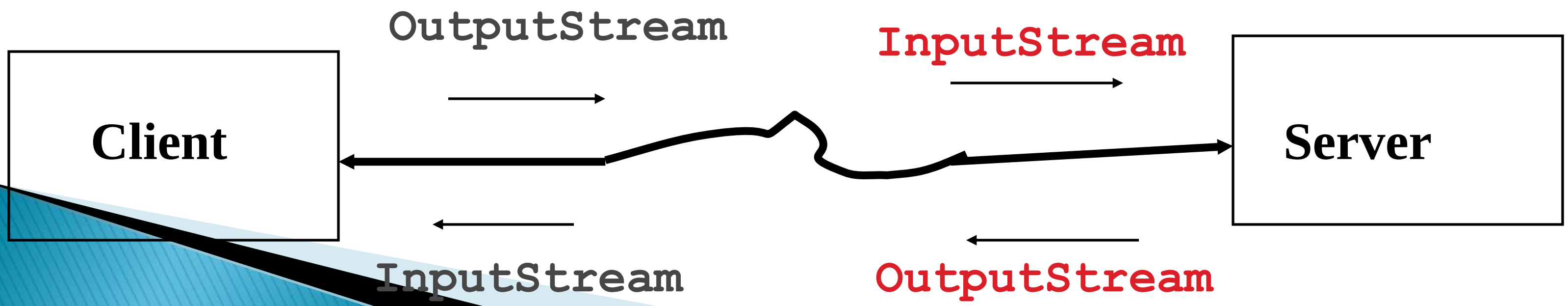
- MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller).
- A network node can have multiple NIC but each with unique MAC.

Networking Basics

- Client-Server interaction
  - Communication between hosts is two-way, but usually the two hosts take different roles.
  - **Server waits for client to make request**
    - Server registered on a known port with the host ("public IP number")
    - Listens for incoming client connections
  - **Client "calls" server to start a conversation**
    - Client making calls uses hostname/IP address and port number
    - Sends request and waits for response
  - Standard services always running
    - ftp, http, smtp, etc. server running on host using expected port
  - Server offers shared resource (information, database, files, printer, compute power) to clients

Socket-Level Programming

- Java Socket programming is used for communication between the applications running on different JRE.
- Java Socket programming can be connection-oriented or connection-less.
- **Socket** and **ServerSocket** classes are used for connection-oriented socket programming.
- **DatagramSocket** and **DatagramPacket** classes are used for connection-less socket programming.
- Java socket programming provides facility to share data between different computing devices.
- Send and receive data using streams

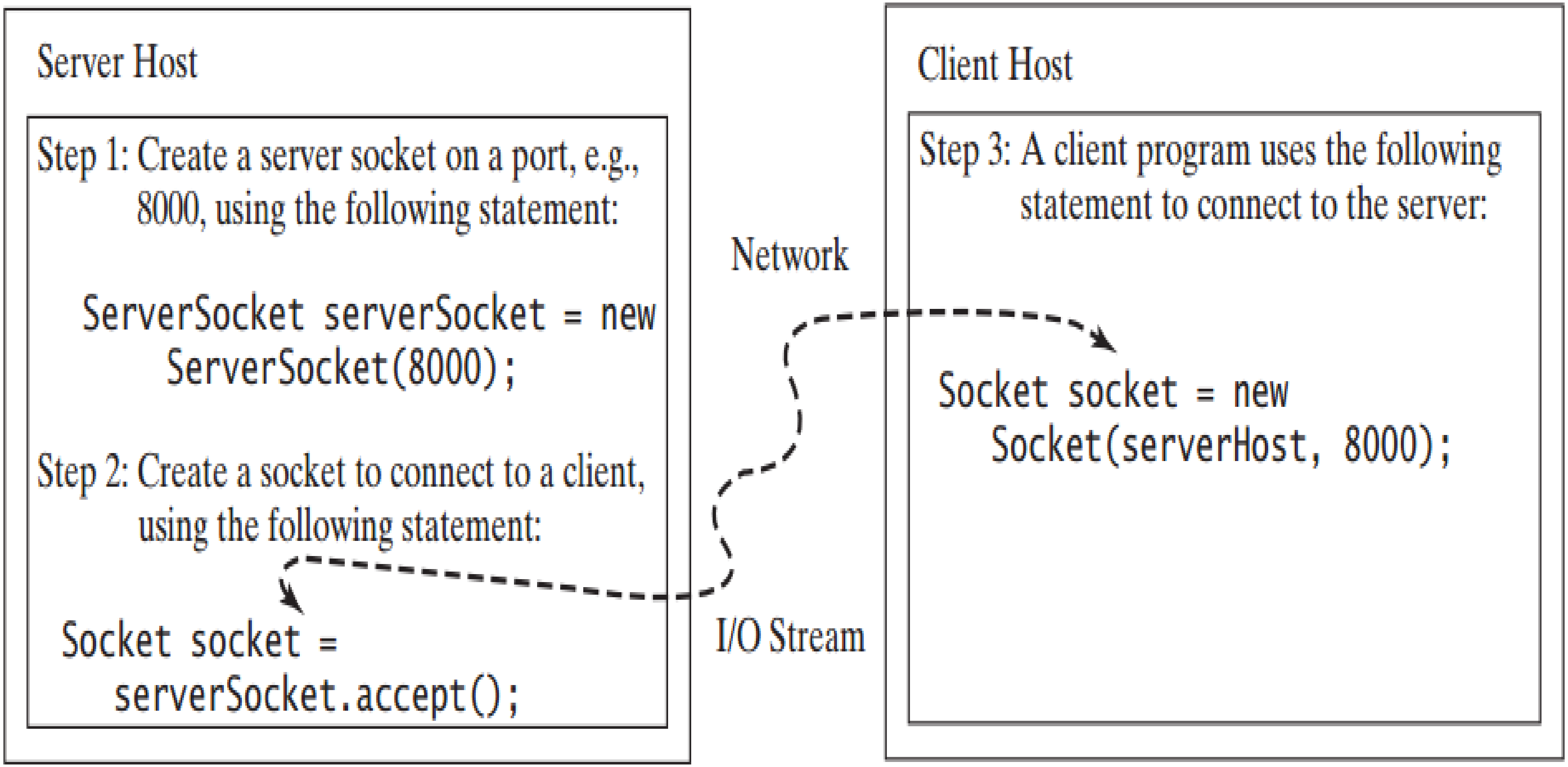


Client/Server Communications

- *Java provides the* **ServerSocket** *class for creating a server socket and the* **Socket** *class for creating a client socket.*
- *Two programs on the Internet communicate through a server socket and a client socket using I/O streams.*
- ***Sockets*** are the endpoints of logical connections between two hosts and can be used to send and receive data.
- Network programming usually involves a server and one or more clients.
- The client sends requests to the server, and the server responds.
- The client begins by attempting to establish a connection to the server.
- The server can accept or deny the connection.
- Once a connection is established, the client and the server communicate through sockets.
- The server must be running when a client attempts to connect to the server.
- The server waits for a connection request from a client.

Client/Server Communications

The statements needed to create sockets on a server and a client are shown below.



Server Host

Step 1: Create a server socket on a port, e.g., 8000, using the following statement:

```
ServerSocket serverSocket = new
    ServerSocket(8000);
```

Step 2: Create a socket to connect to a client, using the following statement:

```
Socket socket =
  serverSocket.accept();
```

Network

I/O Stream

Client Host

Step 3: A client program uses the following statement to connect to the server:

```
Socket socket = new
  Socket(serverHost, 8000);
```

Server Sockets

- To establish a server, you need to create a *server socket* and attach it to a *port*, which is where the server listens for connections.
- The port identifies the TCP service on the socket.
- The following statement creates a server socket **serverSocket**:

```
ServerSocket serverSocket = new ServerSocket(port);
```

- Attempting to create a server socket on a port already in use would cause the **java.net.BindException**.

Client Sockets

- After a server socket is created, the server can use the following statement to listen for connections:

```
    Socket socket = serverSocket.accept();
```

- This statement waits until a client connects to the server socket.
- The **client** issues the following statement to request a connection to a server:

```
Socket socket = new Socket(serverName, port);
```

- This statement opens a socket so that the client program can communicate with the server.

Client Sockets

- **serverName** is the server’s Internet host name or IP address.
- The following statement creates a socket on the client machine to connect to the host 130.254.204.33 at port 8000:

```
Socket socket = new Socket("130.254.204.33", 8000)
```

- Alternatively, you can use the domain name to create a socket, as follows:

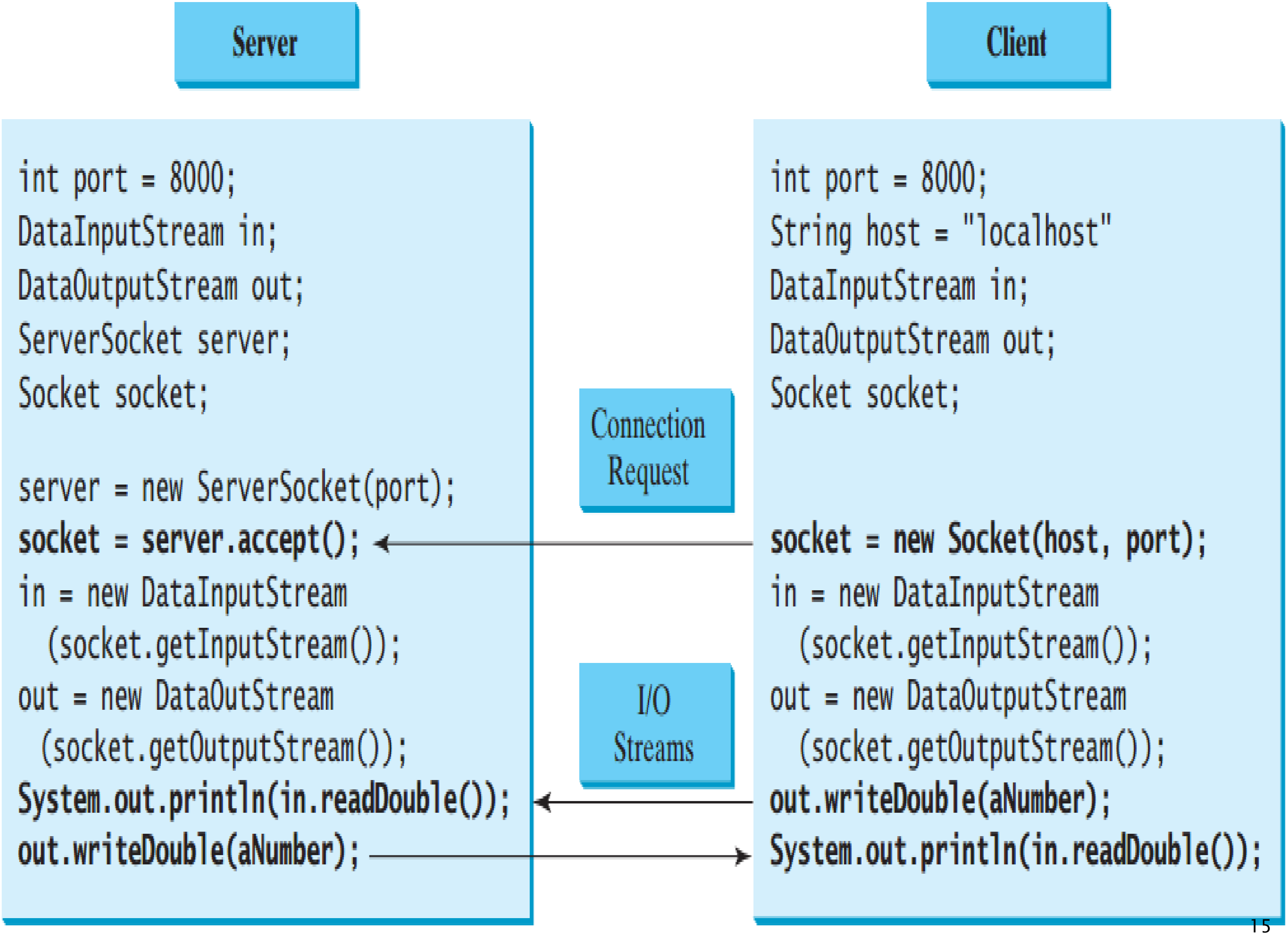
```
Socket socket = new Socket(“www.google.com", 8000);
```

- When you create a socket with a host name, the JVM asks the DNS to translate the host name into the IP address.

Data Transmission through Sockets

- After the server accepts the connection, communication between the server and client is conducted the same as for I/O streams.

- The statements needed to create the streams and to exchange data between them are shown in the Figure below.



Server

```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server = new ServerSocket(port);
socket = server.accept();
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutStream
  (socket.getOutputStream());
System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

Client

```
int port = 8000;
String host = "localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;


socket = new Socket(host, port);
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

Data Transmission through Sockets

- To get an input stream and an output stream, use the **getInputStream()** and **getOutputStream()** methods on a socket object.

- For example, the following statements create an **InputStream** stream called **input** and an **OutputStream** stream called **output** from a socket:

```
InputStream   input   =   socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

```
DataInputStream in = new DataInputStream(input);
DataOutputStream out = new DataOutputStream(output));
```

Data Transmission through Sockets

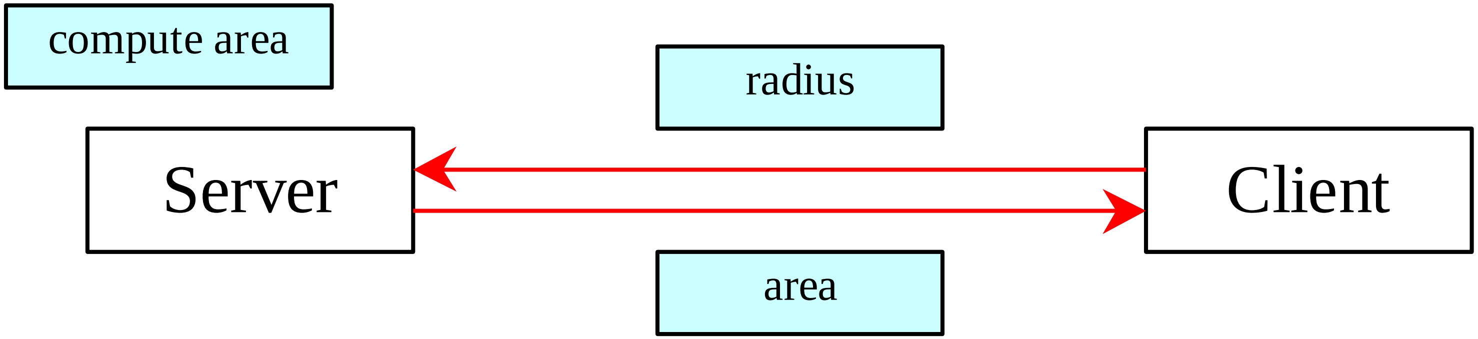
- The **InputStream** and **OutputStream** streams are used to read or write bytes.
- You can use **DataInputStream**, **DataOutputStream**, **BufferedReader**, and **PrintWriter** to wrap on the **InputStream** and **OutputStream** to read or write data, such as **int**, **double**, or **String**.
- The following statements, for instance, create the **DataInputStream** stream **input** and the **DataOutputStream** stream **output** to read and write primitive data values:

```
DataInputStream input = new DataInputStream (socket.getInputStream());
DataOutputStream output = new DataOutputStream (socket.getOutputStream());
```

- The server can use **input.readDouble()** to receive a **double** value from the client, and **output.writeDouble(d)** to send the **double** value **d** to the client.
- Binary I/O is more efficient than text I/O because text I/O requires encoding and decoding.
- Therefore, it is better to use binary I/O for transmitting data between a server and a client to improve performance

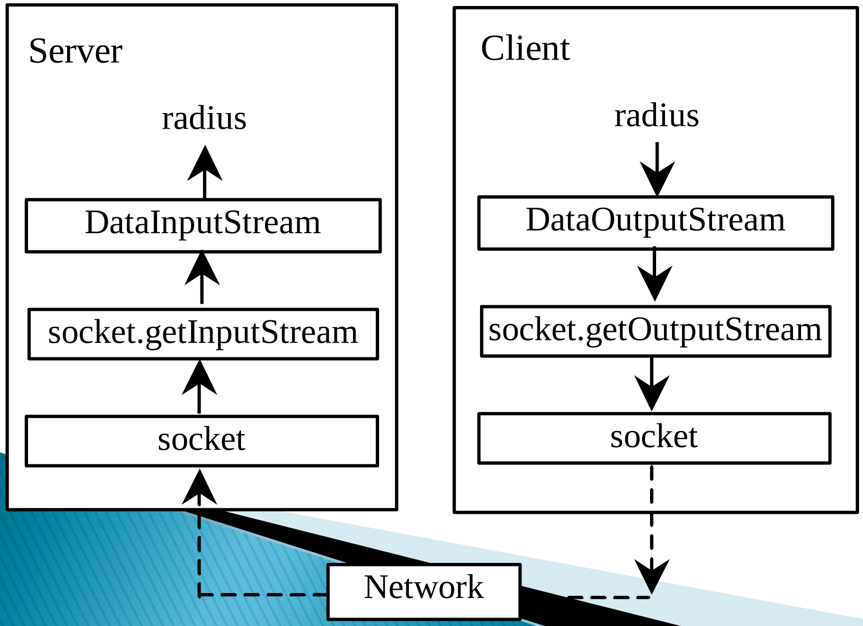
A Client/Server Example

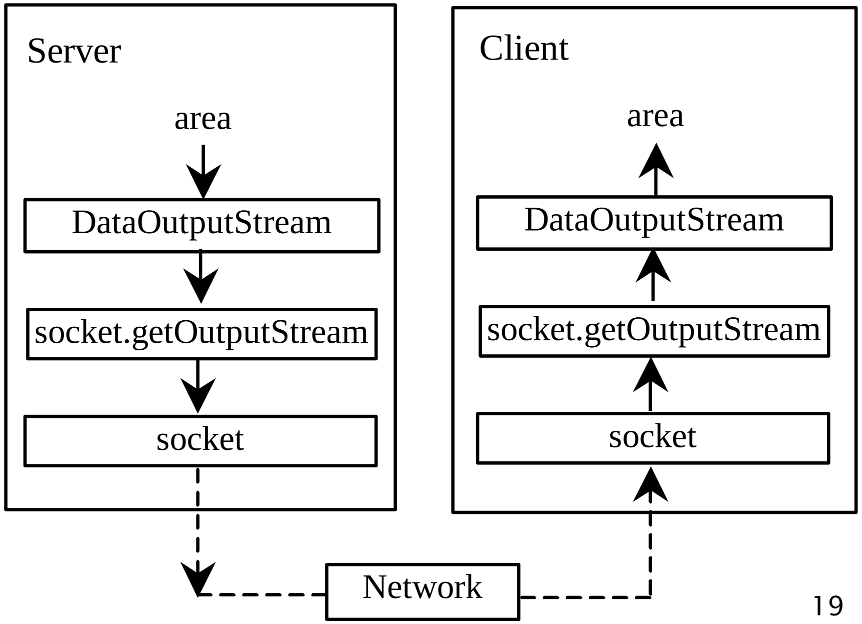
- **Problem**: Write a client and a server program that the client sends data to a server. The server receives the data, uses it to produce a result, and then sends the result back to the client. The client displays the result on the console. In this example, the data sent from the client is the radius of a circle, and the result produced by the server is the area of the circle. The client sends the radius to the server; the server computes the area and sends it to the client.



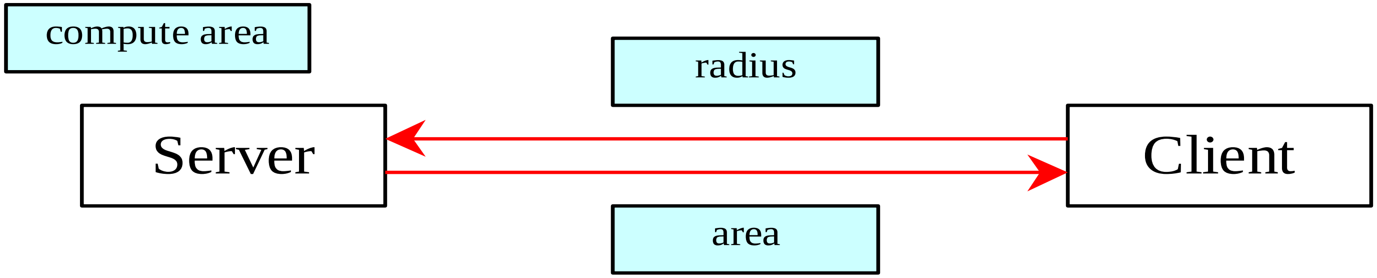
A Client/Server Example

- The client sends the radius through a **DataOutputStream** on the output stream socket, and the server receives the radius through the **DataInputStream** on the input stream socket, as shown in Figure (A) below.
- The server computes the area and sends it to the client through a **DataOutputStream** on the output stream socket, and the client receives the area through a **DataInputStream** on the input stream socket, as shown in Figure (B) below.





A Client/Server Example



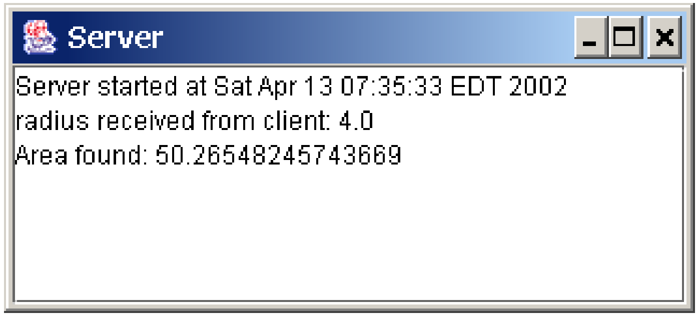
compute area

radius

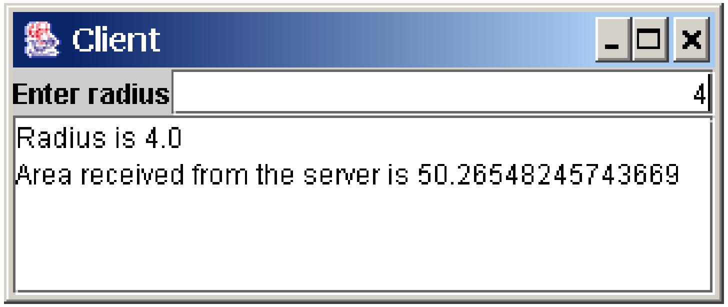
Server

Client

area



```
Server started at Sat Apr 13 07:35:33 EDT 2002
radius received from client: 4.0
Area found: 50.26548245743669
```



Enter radius 4

```
Radius is 4.0
Area received from the server is 50.26548245743669
```

Note: Start the server, then the client.

A Client/Server Example

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;

public class Server extends JFrame {
// Text area for displaying contents
private JTextArea jta = new JTextArea();

public static void main(String[] args) {
new Server();
}

public Server() {
// Place text area on the frame
setLayout(new BorderLayout());
add(new JScrollPane(jta), BorderLayout.CENTER);

setTitle("Server");
setSize(500, 300);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true); // It is necessary to show the frame here!
```

A Client/Server Example

```
try {
 // Create a server socket
ServerSocket serverSocket = new ServerSocket(8000);
 jta.append("Server started at " + new Date() + '\n' );
 // Listen for a connection request
Socket socket = serverSocket.accept();
 // Create data input and output streams
DataInputStream inputFromClient = new
DataInputStream(socket.getInputStream());
DataOutputStream outputToClient = new
DataOutputStream(socket.getOutputStream());
 while (true) {
 // Receive radius from the client
double radius = inputFromClient.readDouble();
  double area = radius * radius * Math.PI; // Compute
area
 // Send area back to the client
outputToClient.writeDouble(area);
 jta.append("Radius received from client: " + radius
+ '\n' );
 jta.append("Area found: " + area + '\n' );
 }
}
```

A Client/Server Example

```
  import java.io.*;
   import java.net.*;
   import java.awt.*;
   import java.awt.event.*;
   import javax.swing.*;
  public class Client extends JFrame {
   // Text field for receiving radius
   private JTextField jtf = new JTextField();
   // Text area to display contents
   private JTextArea jta = new JTextArea();
   // IO streams
   private DataOutputStream toServer;
   private DataInputStream fromServer;
   public static void main(String[] args) {
   new Client();
   }
   public Client() {
   // Panel p to hold the label and text field
   JPanel p = new JPanel();
   p.setLayout(new BorderLayout());
   p.add(new JLabel("Enter radius"), BorderLayout.WEST);
   p.add(jtf, BorderLayout.CENTER);
   jtf.setHorizontalAlignment(JTextField.RIGHT);
   setLayout(new BorderLayout());
   add(p, BorderLayout.NORTH);
   add(new JScrollPane(jta), BorderLayout.CENTER);
   jtf.addActionListener(new ActionListener(){
//Copy paste here the code on from the TextActionListener Slide
));
   setTitle("Client");
   setSize(500, 300);
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setVisible(true); // It is necessary to show the frame here!
```

A Client/Server Example

```
try {
   // Create a socket to connect to the server
  Socket socket = new Socket("localhost", 8000);


   // Create an input stream to receive data from
  the server
  fromServer = new
  DataInputStream(socket.getInputStream());

   // Create an output stream to send data to the
  server
  toServer = new
  DataOutputStream(socket.getOutputStream());

   }
   catch (IOException ex) {
   jta.append(ex.toString() + '\n' );
   }
}
```

TextField ActionListener

```
jtf.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
      try {
        // Get the radius from the text field
        double radius = Double.parseDouble(jtf.getText().trim());
        // Send the radius to the server
        toServer.writeDouble(radius);
        toServer.flush();
        // Get area from the server
        double area = fromServer.readDouble();
        // Display to the text area
        jta.append("Radius is " + radius + "\n");
        jta.append("Area received from the server is" + area + '\n');
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  });
```

The InetAddress Class

- *The server program can use the* **InetAddress** *class to obtain the information about the IP address and host name for the client.*
- You can use the **InetAddress** class to find the client's host name and IP address.
- You can use the statement shown below to create an instance of **InetAddress** for the client on a socket.

```
InetAddress inet = socket.getInetAddress();
```

- Next, you can display the client's host name and IP address, as follows:

System.out.println(**"Client's host name is "** + inet.getHostName());

System.out.println(**"Client's IP Address is "** + inet.getHostAddress());

- You can also create an instance of **InetAddress** from a host name or IP address using the static **getByName()** method.
- For example, the following statement creates an **InetAddress** for the host **www.aastu.edu.et**.

InetAddress address = InetAddress.getByName(“**www.aastu.edu.et"**);

Example: The InetAddress Class

```
import java.net.*;
public class IdentifyHostNameIP {
 public static void main(String[] args) {
    try {
    InetAddress address = InetAddress.getByName("www.bdu.edu.et");
    System.out.print("Host name:"+ address.getHostName());
   System.out.println("IP address:"+ address.getHostAddress());
   }
   catch (UnknownHostException ex) {
  System.err.println("Unknown host or IP address www.aastu.edu.et
");
   }
 }
 }
```

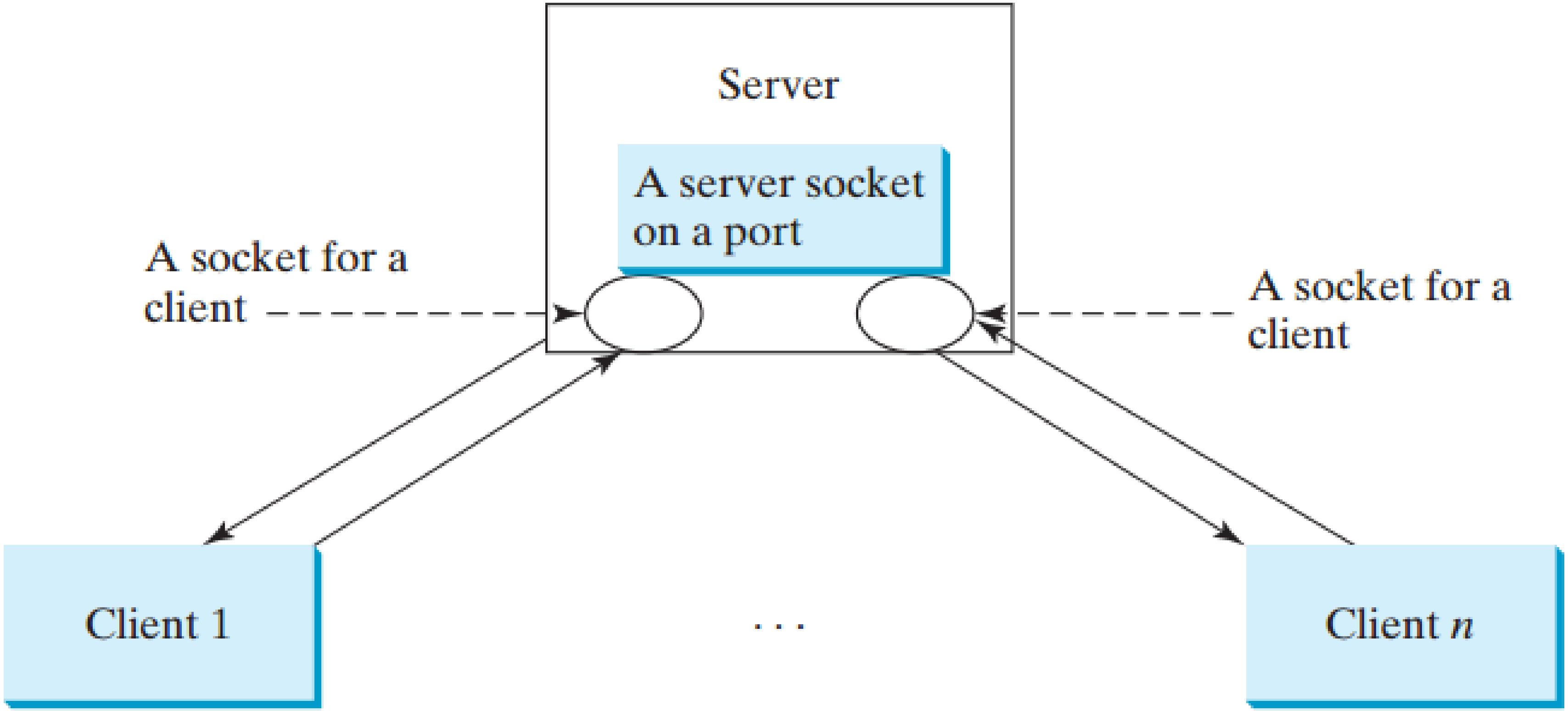
Serving Multiple Clients

- *A server can serve multiple clients.*
- *The connection to each client is handled by one thread.*
- Multiple clients are quite often connected to a single server at the same time.
- You can use threads to handle the server's multiple clients simultaneously.
- Simply create a thread for each connection.
- Here is how the server handles the establishment of a connection:

```
while (true) {
  Socket socket = serverSocket.accept();
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

- The server socket can have many connections.
- Each iteration of the while loop creates a new connection.
- Whenever a connection is established, a new thread is created to handle communication between the server and the new client; and this allows multiple connections to run at the same time.

Example: Serving Multiple Clients



Note: Start the server first, then start multiple clients.

Example: Serving Multiple Clients

```
import java.io.*;
 import java.net.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;
public class MultiThreadServer extends JFrame {
// Text area for displaying contents
private JTextArea jta = new JTextArea();

 public static void main(String[] args) {
new MultiThreadServer();
}

public MultiThreadServer() {
// Place text area on the frame
setLayout(new BorderLayout());
add(new JScrollPane(jta), BorderLayout.CENTER);

setTitle("MultiThreadServer");
setSize(500, 300);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true); // It is necessary to show the frame here!

 try {
 // Create a server socket
ServerSocket serverSocket = new ServerSocket(8000);
 jta.append("MultiThreadServer started at " + new Date() + '\n' );
 // Number a client
 int clientNo = 1;
```

Example: Serving Multiple Clients

```
 while (true) {
 // Listen for a new connection request
Socket socket = serverSocket.accept();

 // Display the client number
 jta.append("Starting thread for client " + clientNo +
 " at " + new Date() + '\n' );

 // Find the client's host name and IP address
InetAddress inetAddress = socket.getInetAddress();
 jta.append("Client " + clientNo + "'s host name is “
 + inetAddress.getHostName() + "\n");
 jta.append("Client " + clientNo + "'s IP Address is “
 + inetAddress.getHostAddress() + "\n");

 // Create a new thread for the connection
 HandleAClient task = new HandleAClient(socket);

 // Start the new thread
 new Thread(task).start();

 // Increment clientNo
 clientNo++;
 }
 }
 catch(IOException ex) {
 System.err.println(ex);
 }
 }
```

Example: Serving Multiple Clients

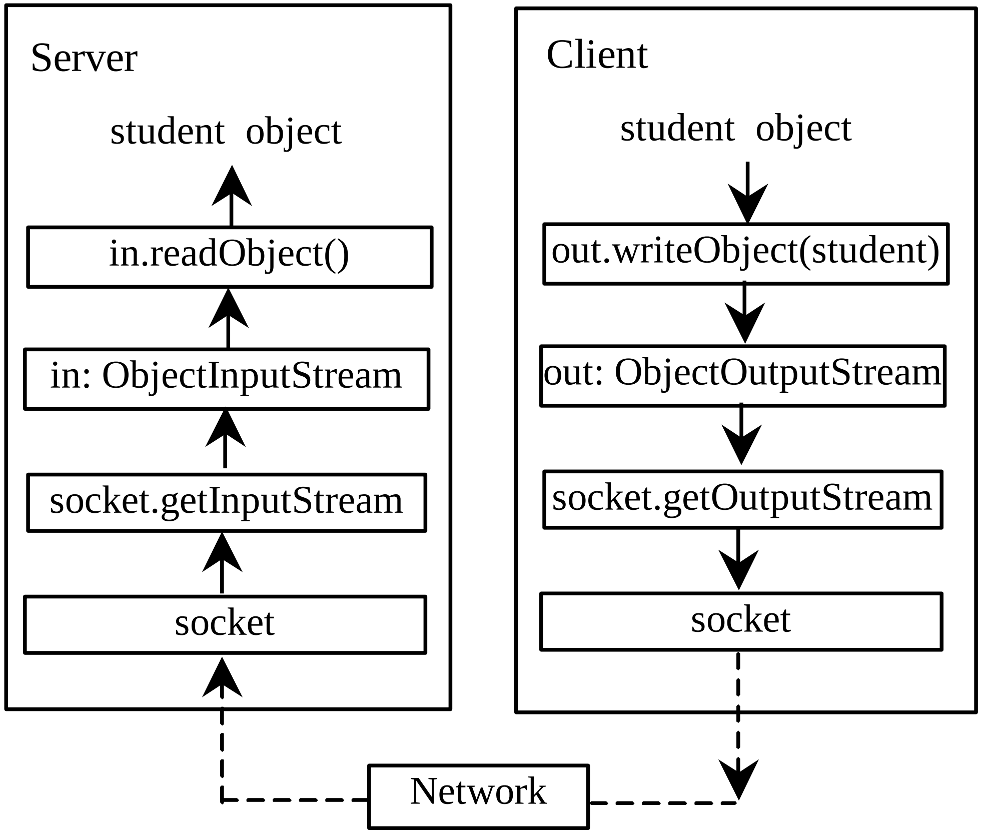
```
   // Inner class
    // Define the thread class for handling new connection
   class HandleAClient implements Runnable {
   private Socket socket; // A connected socket

    /** Construct a thread */
    public HandleAClient(Socket socket) {
    this.socket = socket;
    }

    @Override /** Run a thread */
    public void run(){
    try {
    // Create data input and output streams
   DataInputStream inputFromClient = new DataInputStream(socket.getInputStream());
   DataOutputStream outputToClient = new DataOutputStream(socket.getOutputStream());

    // Continuously serve the client
    while (true) {
    // Receive radius from the client
    double radius = inputFromClient.readDouble();
    // Compute area
    double area = radius * radius * Math.PI;
    // Send area back to the client
   outputToClient.writeDouble(area);
    jta.append("radius received from client: " +
    radius + '\n' );
    jta.append("Area found: " + area + '\n' );
    }
    }
    catch(IOException e) {
    System.err.println(e);
    }
    }
    }
}
```

Sending and Receiving Objects

- *A program can send and receive objects from another program.*
- In the preceding examples, you learned how to send and receive data of primitive types.
- You can also send and receive objects using **ObjectOutputStream** and **ObjectInputStream** on socket streams.
- To enable passing, the objects must be serializable.
- The following example demonstrates how to send and receive objects.

Example: Passing Objects in Network Programs

Write a program that collects student information from a client and send them to a server. Passing student information in an object.



Example: Passing Objects in Network Programs

```
  public class StudentAddress implements java.io.Serializable {
   private String name;
   private String street;
   private String city;
   private String state;
   private String zip;
  public StudentAddress(String name, String street, String city,
   String state, String zip) {
   this.name = name;
   this.street = street;
   this.city = city;
   this.state = state;
   this.zip = zip;
   }
   public String getName() {
   return name;
   }
   public String getStreet() {
   return street;
   }
   public String getCity() {
   return city;
   }

   public String getState() {
   return state;
   }

   public String getZip() {
   return zip;
   }
}
```

Example: Passing Objects in Network Programs

```
import java.io.*;
 import java.net.*;
 import java.awt.*;
 import java.awt.event.*;
 import javax.swing.*;
 import javax.swing.border.*;
public class StudentClient extends JApplet {
 private JTextField jtfName = new JTextField(32);
 private JTextField jtfStreet = new JTextField(32);
 private JTextField jtfCity = new JTextField(20);
 private JTextField jtfState = new JTextField(2);
 private JTextField jtfZip = new JTextField(5);
 // Button for sending a student's address to the server
 private JButton jbtRegister = new JButton("Register to the
Server");
 // Indicate if it runs as application
private boolean isStandAlone = false;
 // Host name or IP address
 String host = "localhost";
 public void init() {
 // Panel p1 for holding labels Name, Street, and City
 JPanel p1 = new JPanel();
 p1.setLayout(new GridLayout(3, 1));
 p1.add(new JLabel("Name"));
 p1.add(new JLabel("Street"));
 p1.add(new JLabel("City"));
```

Example: Passing Objects in Network Programs

```
 // Panel jpState for holding state
 JPanel jpState = new JPanel();
 jpState.setLayout(new BorderLayout());
 jpState.add(new JLabel("State"),
    BorderLayout.WEST);
 jpState.add(jtfState, BorderLayout.CENTER);
// Panel jpZip for holding zip
 JPanel jpZip = new JPanel();
 jpZip.setLayout(new BorderLayout());
 jpZip.add(new JLabel("Zip"), BorderLayout.WEST);
 jpZip.add(jtfZip, BorderLayout.CENTER);

 // Panel p2 for holding jpState and jpZip
 JPanel p2 = new JPanel();
 p2.setLayout(new BorderLayout());
 p2.add(jpState, BorderLayout.WEST);
 p2.add(jpZip, BorderLayout.CENTER);

 // Panel p3 for holding jtfCity and p2
 JPanel p3 = new JPanel();
 p3.setLayout(new BorderLayout());
 p3.add(jtfCity, BorderLayout.CENTER);
```

Example: Passing Objects in Network Programs

```
p3.add(p2, BorderLayout.EAST);
 // Panel p4 for holding jtfName, jtfStreet, and p3
 JPanel p4 = new JPanel();
 p4.setLayout(new GridLayout(3, 1));
 p4.add(jtfName);
 p4.add(jtfStreet);
 p4.add(p3);
 // Place p1 and p4 into StudentPanel
 JPanel studentPanel = new JPanel(new BorderLayout());
 studentPanel.setBorder(new BevelBorder(BevelBorder.RAISED));
 studentPanel.add(p1, BorderLayout.WEST);
 studentPanel.add(p4, BorderLayout.CENTER);
 // Add the student panel and button to the applet
 add(studentPanel, BorderLayout.CENTER);
 add(jbtRegister, BorderLayout.SOUTH);
 // Register listener
jbtRegister.addActionListener(new ButtonListener());

 // Find the IP address of the Web server
 if (!isStandAlone)
host = getCodeBase().getHost();
 }

 /** Handle button action */
 private class ButtonListener implements ActionListener {
 @Override
 public void actionPerformed(ActionEvent e) {
 try {
 // Establish connection with the server
Socket socket = new Socket(host, 8000);
```

Example: Passing Objects in Network Programs

```
 // Create an output stream to the server
ObjectOutputStream toServer = new
ObjectOutputStream(socket.getOutputStream());
 // Get text field
 String name = jtfName.getText().trim();
 String street = jtfStreet.getText().trim();
 String city = jtfCity.getText().trim();
 String state = jtfState.getText().trim();
 String zip = jtfZip.getText().trim();

 // Create a StudentAddress object and send to the
server
 StudentAddress s =
 new StudentAddress(name, street, city, state,
zip);
toServer.writeObject(s);
 }
 catch (IOException ex) {
System.err.println(ex);
 }
 }
 }
```

Example: Passing Objects in Network Programs

```
/** Run the applet as an application */
 public static void main(String[] args) {
 // Create a frame
JFrame frame = new JFrame("Register Student Client");

 // Create an instance of the applet
 StudentClient applet = new StudentClient();
 applet.isStandAlone = true;

 // Get host
if (args.length == 1) applet.host = args[0];

 // Add the applet instance to the frame
 frame.add(applet, BorderLayout.CENTER);

 // Invoke init() and start()
 applet.init();
 applet.start();

 // Display the frame
 frame.pack();
 frame.setVisible(true);
 }
 }
```

Example: Passing Objects in Network Programs

```
import java.io.*;
import java.net.*;
public class StudentServer {
 private ObjectOutputStream outputToFile;
 private ObjectInputStream inputFromClient;
public static void main(String[] args) {
 new StudentServer();
 }

 public StudentServer() {
 try {
 // Create a server socket
ServerSocket serverSocket = new ServerSocket(8000);
 System.out.println("Server started ");

 // Create an object output stream
 outputToFile = new ObjectOutputStream(
 new FileOutputStream("student.dat", true));

 while (true) {
 // Listen for a new connection request
Socket socket = serverSocket.accept();

 // Create an input stream from the socket
inputFromClient = new
ObjectInputStream(socket.getInputStream());
```

Example: Passing Objects in Network Programs

```
// Read from input
 Object object =  inputFromClient.readObject();

 // Write to the file
 outputToFile.writeObject(object);
 System.out.println("A new student object is stored");
 }
}
 catch(ClassNotFoundException ex) {
 ex.printStackTrace();
 }
 catch(IOException ex) {
 ex.printStackTrace();
 }
 finally {
 try {
 inputFromClient.close();
 outputToFile.close();
 }
 catch (Exception ex) {
 ex.printStackTrace();
 }
 }
 }
 }
```

The URL Class

- Audio and images are stored in files.
- The java.net.URL class can be used to identify the files on the Internet.
- In general, a URL (Uniform Resource Locator) is a pointer to a "resource" on the World Wide Web.
- A resource can be something as simple as a file or a directory.
- You can create a URL object using the following constructor:

```
public URL(String spec) throws MalformedURLException
```

- For example, the following statement creates a URL object for http://www.sun.com:

```
try {
  URL url = new URL("http://www.sun.com");
}
catch(MalformedURLException ex) {
}
```

Creating a URL Instance

- To retrieve the file, first create a URL object for the file.
- For example, the following statement creates a URL object for http://www.cs.armstrong.edu/liang/index.html.

```
URL url = new URL("http://www.cs.armstrong.edu/liang/index.html");
```

- You can then use the **openStream()** method defined in the URL class to open an input stream to the file's URL.

```
InputStream inputStream = url.openStream();
```

The End!!