Networking

Instructor

Mehrnaz Zhian

Mehrnaz.zhian@senecacollege.ca

Introduction

- To browse the Web or send an email, your computer must be connected to the Internet.
- The *Internet* is the global **network** of millions of computers.
- 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).

IP Address

- When a computer needs to communicate with another computer, it needs to know the other computer's address.
- An *Internet Protocol* (IP) address uniquely identifies the computer on the Internet.
- An IP address consists of four dotted decimal numbers between **0** and **255**, such as **205.207.147.230**

Domain Name & Domain Name Server

- Since it is not easy to remember so many numbers, they are often mapped to meaningful names called ***domain names***, such as `senecacollege.ca`.

- Special servers called *Domain Name Servers* (DNS) on the Internet translate host names into IP addresses.

- When a computer contacts `senecacollege.ca`, it first asks the DNS to translate this domain name into a numeric IP address and then sends the request using the IP address.

TCP

- The Internet Protocol is a low-level protocol for delivering data from one computer to another across the Internet in packets.
- The higher-level protocol used in conjunction with the IP is the *Transmission Control Protocol* (TCP).
- TCP enables two hosts to establish a connection and exchange streams of data.
- TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.

Stream-based/Packet-based Communications

- Java supports stream-based communications.
- *Stream-based communications* use TCP for data transmission.
- Since TCP can detect lost transmissions and resubmit them, transmissions are reliable
- Stream-based communications are used in most areas of Java programming and are the focus of this course

Client/Server Computing

- Two programs on the Internet communicate through a server socket and a client socket using I/O streams.
- Java treats socket communications much as it treats I/O operations
- Java provides the **ServerSocket** class for creating a server socket and the **Socket** class for creating a client socket.

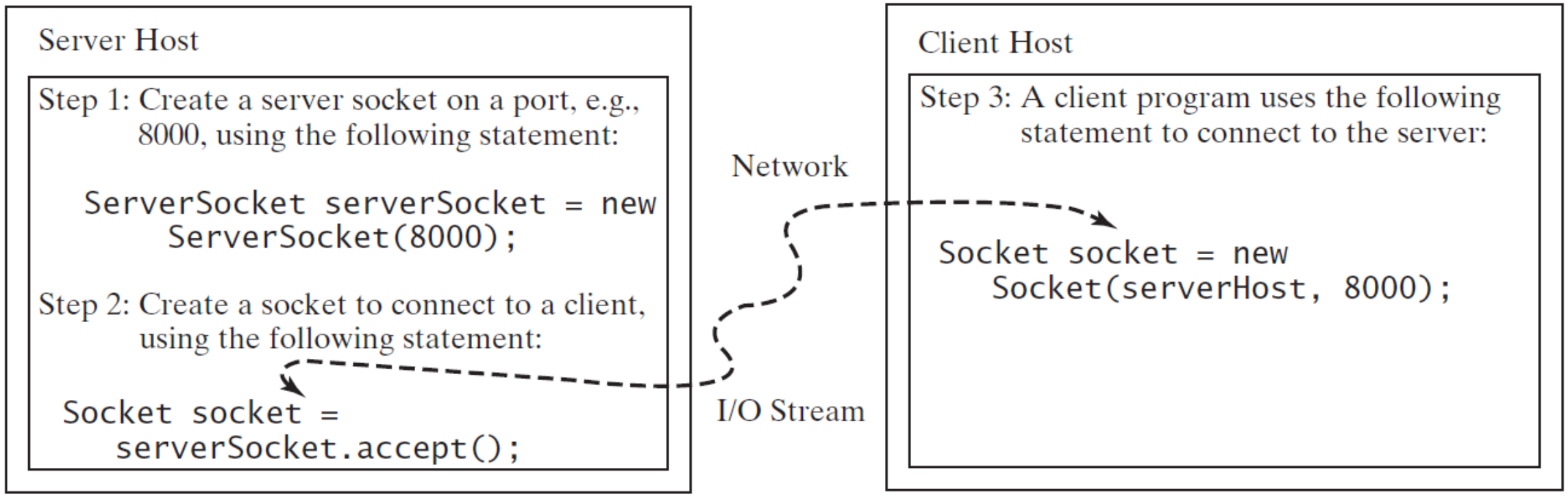
Client/Server Computing

- The Java API provides the classes for creating sockets to facilitate program communications over the Internet.
- *Sockets* are the endpoints of logical connections between two hosts and can be used to send and receive data.
- Java treats socket communications much as it treats I/O operations; thus, programs can read from or write to sockets as easily as they can read from or write to files.

Client/Server Computing (Cont.)

- 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 the client.

The server creates a server socket and, once a connection to a client is established, connects to the client with a client socket



Server Sockets

- To establish a connection, 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.
- Port numbers range from 0 to 65535 (2^{16}-1), but port numbers 0 to 1024 (*System Ports*) are reserved for privileged services
- For instance, the email server runs on port 25, and the Web server usually runs on port 80.
- You can choose any port number that is not currently used by other programs.

Client Sockets

- 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 a **java.net.BindException**
- 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

Client Sockets

- 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.

- ***serverName*** is the server's Internet host name or IP address

Client Sockets

- The following statement creates a socket on the client machine to connect to the host 205.207.147.230 at port 8000:

```
Socket socket = new Socket("205.207.147.230", 8000)
```

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

```
Socket socket = new Socket("senecacollege.ca", 8000);
```

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

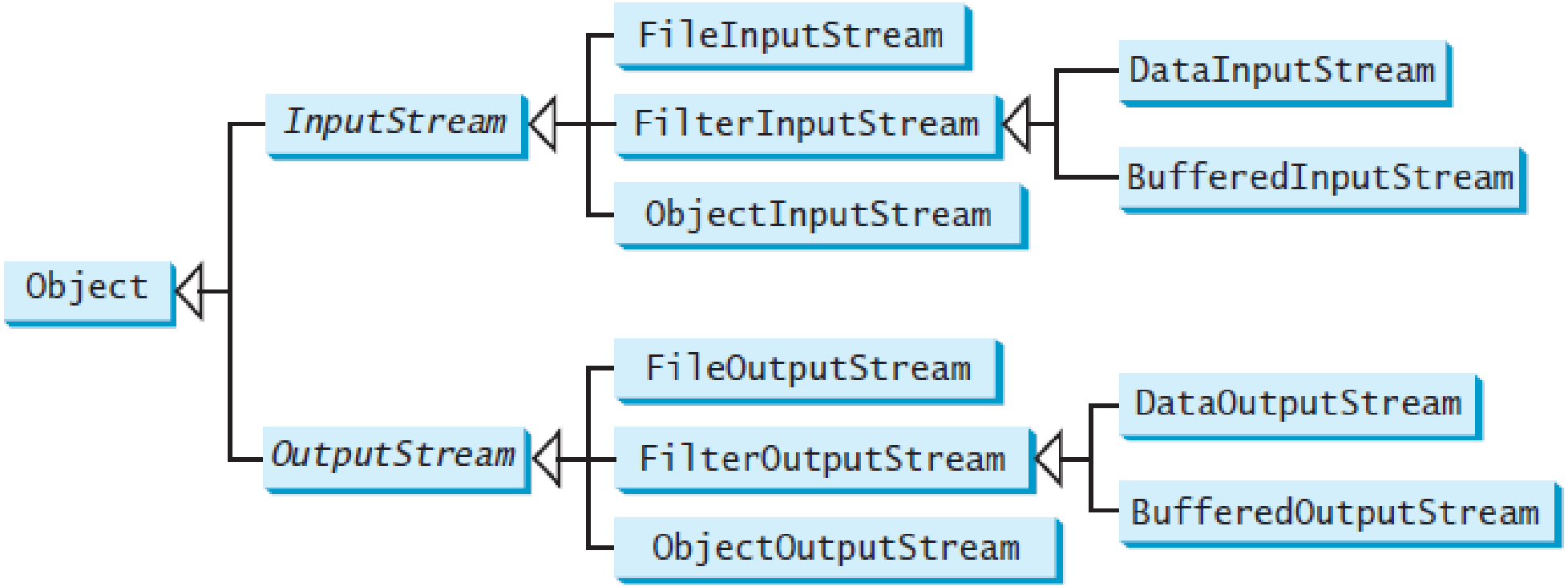
Socket Programming – Notes

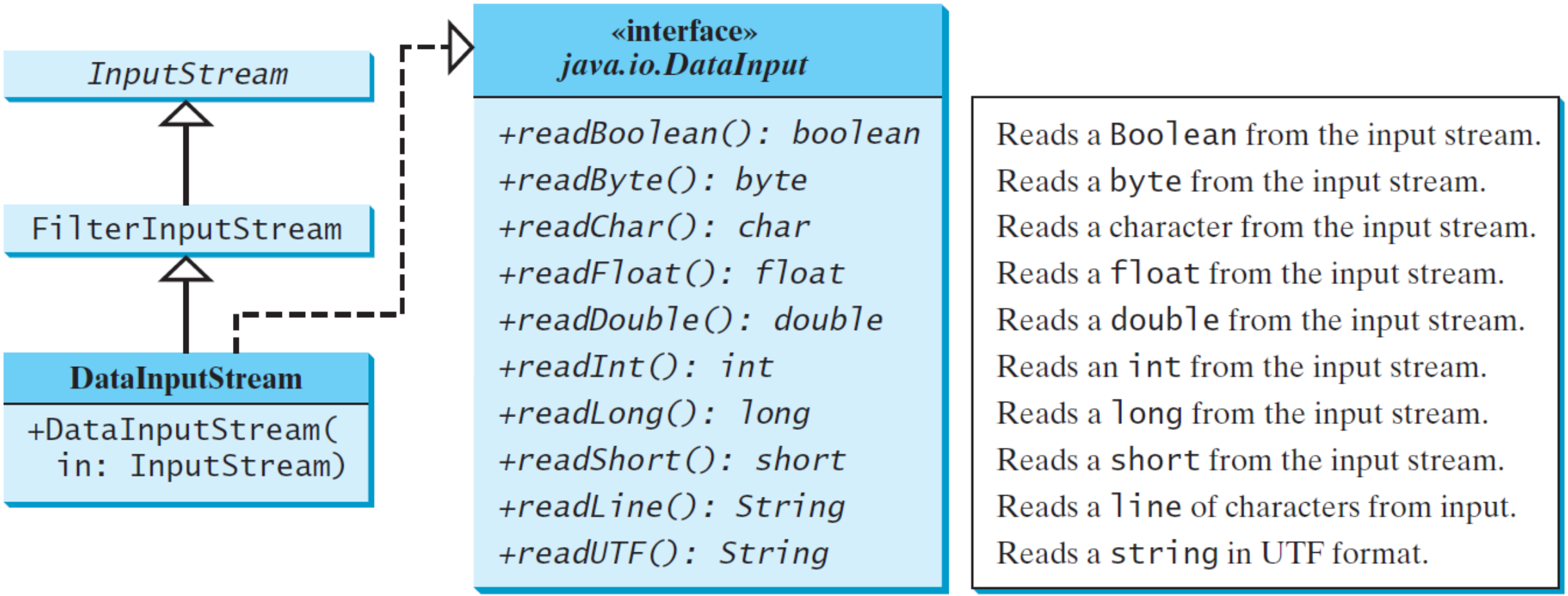
- A program can use the host name **localhost** or the IP address **127.0.0.1** to refer to the machine on which a client is running.

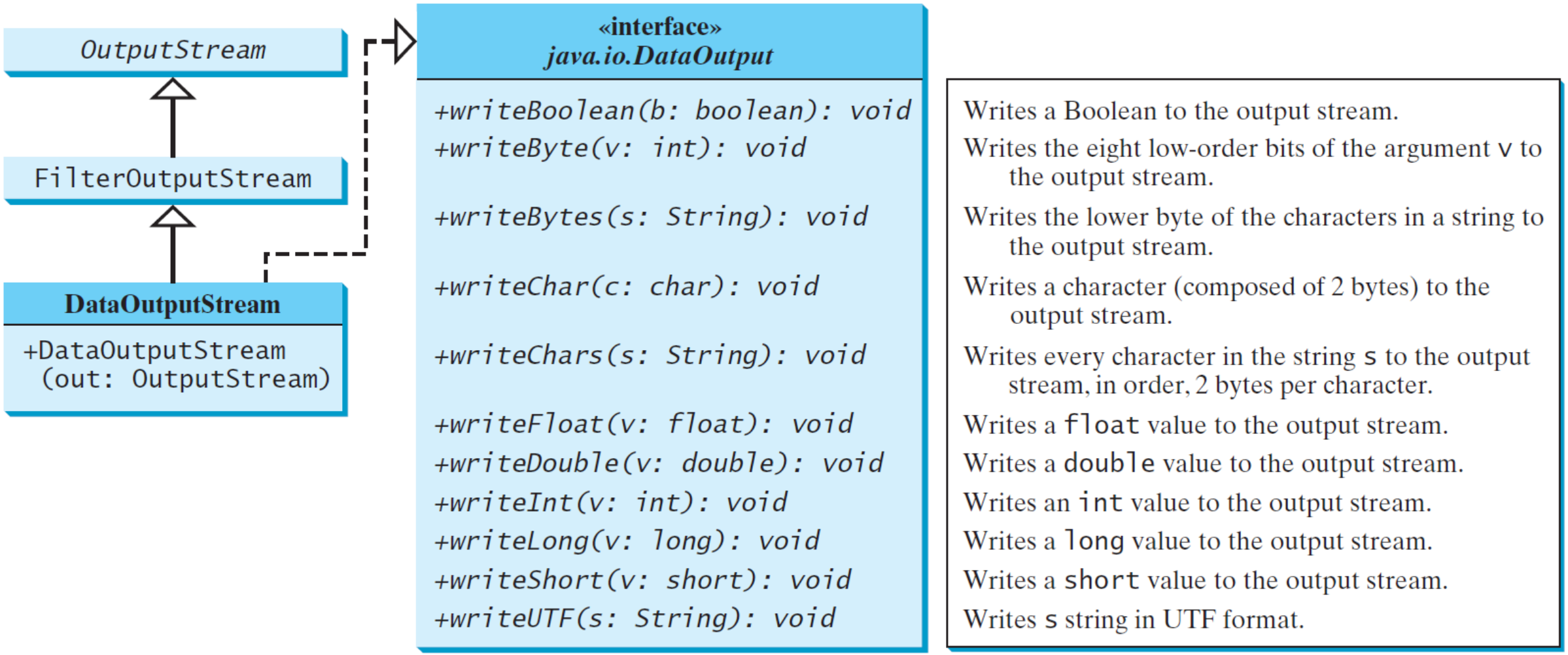
- The **Socket** constructor throws a **java.net.UnknownHostException** if the host cannot be found.

Data Transmission through Sockets

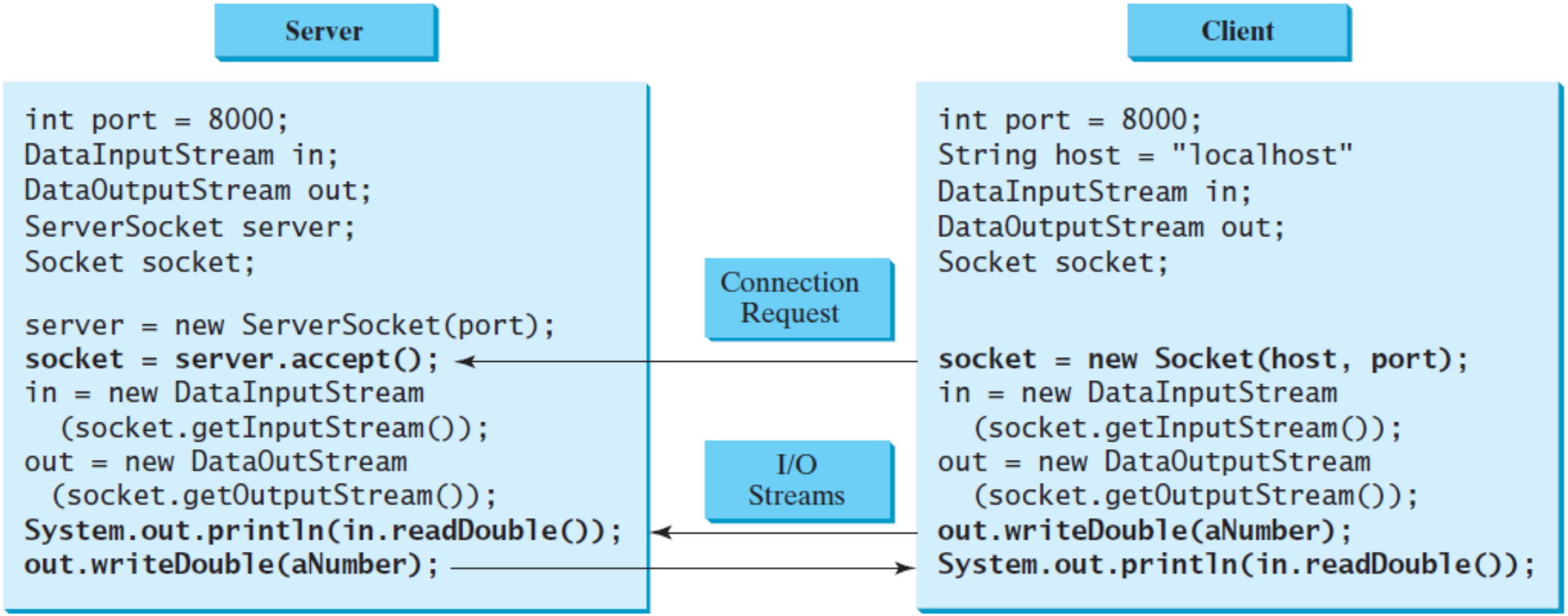
- After the server accepts the connection, communication between the server and the client is conducted in the same way as for I/O streams.




InputStream
FilterInputStream
DataInputStream
+DataInputStream(
in: InputStream)
«interface»
java.io.DataInput
+readBoolean(): boolean
+readByte(): byte
+readChar(): char
+readFloat(): float
+readDouble(): double
+readInt(): int
+readLong(): long
+readShort(): short
+readLine(): String
+readUTF(): String
Reads a Boolean from the input stream.
Reads a byte from the input stream.
Reads a character from the input stream.
Reads a float from the input stream.
Reads a double from the input stream.
Reads an int from the input stream.
Reads a long from the input stream.
Reads a short from the input stream.
Reads a line of characters from input.
Reads a string in UTF format.


OutputStream
FilterOutputStream
DataOutputStream
+DataOutputStream (out: OutputStream)
«interface»
java.io.DataOutput
+writeBoolean(b: boolean): void
+writeByte(v: int): void
+writeBytes(s: String): void
+writeChar(c: char): void
+writeChars(s: String): void
+writeFloat(v: float): void
+writeDouble(v: double): void
+writeInt(v: int): void
+writeLong(v: long): void
+writeShort(v: short): void
+writeUTF(s: String): void
Writes a Boolean to the output stream.
Writes the eight low-order bits of the argument v to the output stream.
Writes the lower byte of the characters in a string to the output stream.
Writes a character (composed of 2 bytes) to the output stream.
Writes every character in the string s to the output stream, in order, 2 bytes per character.
Writes a float value to the output stream.
Writes a double value to the output stream.
Writes an int value to the output stream.
Writes a long value to the output stream.
Writes a short value to the output stream.
Writes s string in UTF format.

- The statements needed to create the streams and to exchange data between them.
- The server and client exchange data through I/O streams on top of the socket.



Data Transmission through Sockets (Cont.)

- 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();
```

Data Transmission through Sockets (Cont.)

- 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:

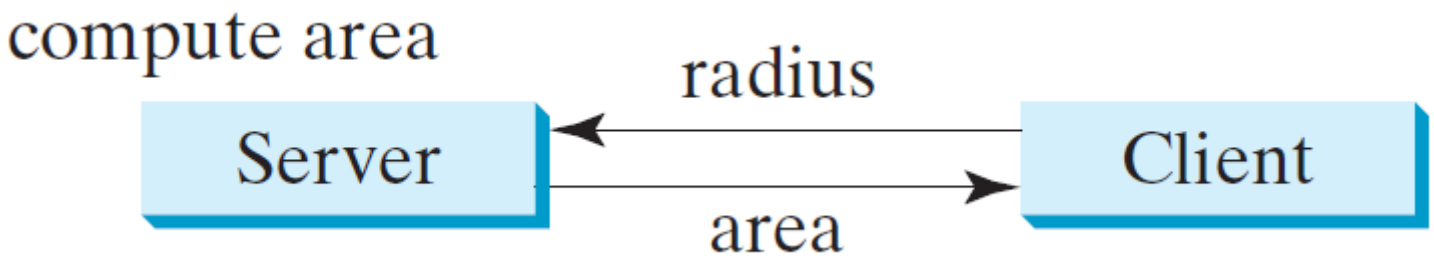
```
DataInputStream input = new DataInputStream
        (socket.getInputStream());
DataOutputStream output = new DataOutputStream
        (socket.getOutputStream());
```

Data Transmission through Sockets (Cont.)

- 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.
- It is better to use binary I/O for transmitting data between a server and a client to improve performance.

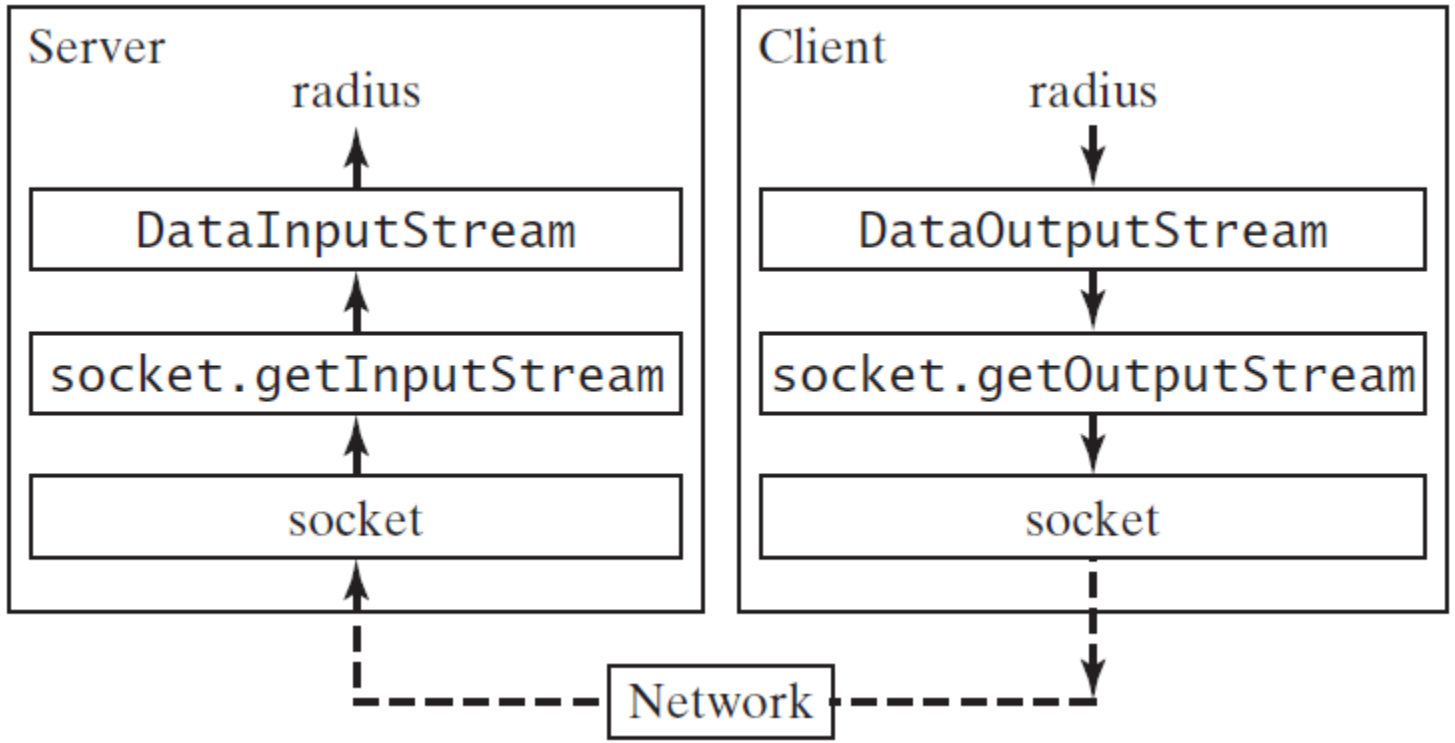
A Client/Server Example

- This example presents a client program and a server program.
- The client sends data to a server and 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 comprise the radius of a circle, and the result produced by the server is the area of the circle



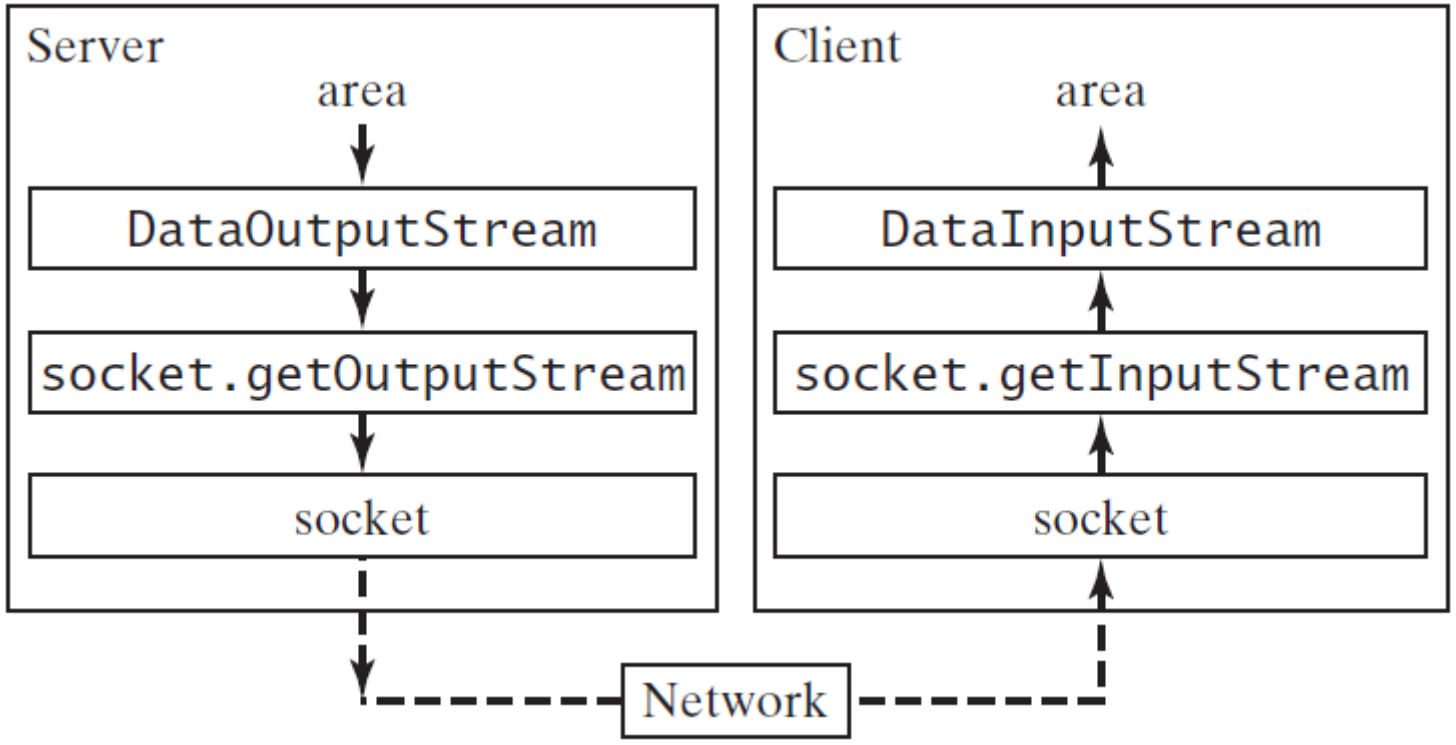
A Client/Server Example (Cont.)

- 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:



A Client/Server Example (Cont.)

- 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



A Client/Server Example (Cont.)

- Let's see server and client programs and a sample run of them
- You start the server program first and then start the client program
- In the client program, enter a radius in the text field and press ***Enter*** to send the radius to the server
- The server computes the area and sends it back to the client
- This process is repeated until one of the two programs terminates
- The networking classes are in the package **java.net**
- You should import this package when writing Java network programs

Serving Multiple Clients

- A server can serve multiple clients
- The connection to each client is handled by one thread

```
while (true) {
        // Connect to a client
        Socket socket = serverSocket.accept();
        Thread thread = new ThreadClass(socket);
        thread.start();
}
```

Multithreading enables a server to handle multiple independent clients