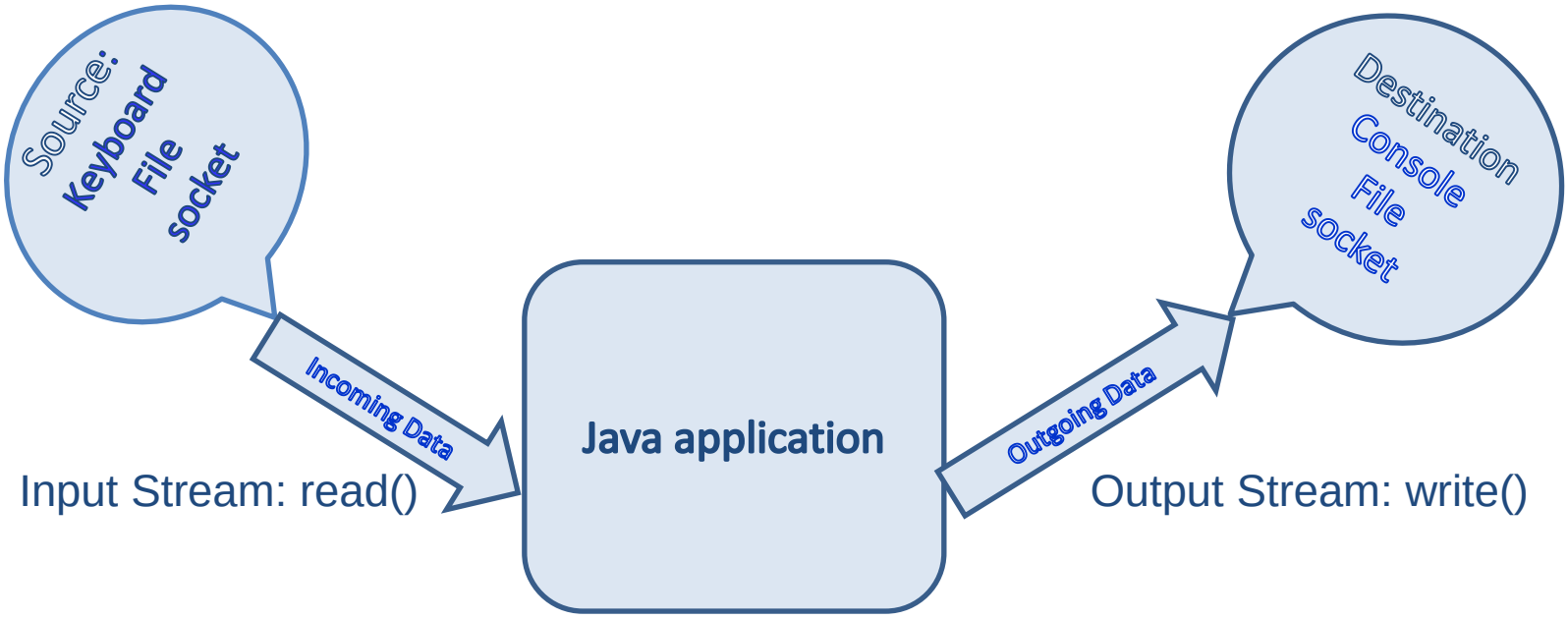
IO , Serializtion & Networking

Authored by : SangeetaJ / GaneshR ***Presented by: Sangeeta Joshi***

Agenda

- I/O Fundamentals
- Stream Classes
- Input and Output Stream
- File I/O
- File Writer
- File Stream
- Scanner
- Intro to Networking
- Client - Server
- Socket
- Port number

Basic IO operations



Flow of data is depicted as Stream
Input Stream : Flow of data from a source to java application
Output Stream: Flow of data from application to some destination

I/O Fundamentals

I/O Fundamentals

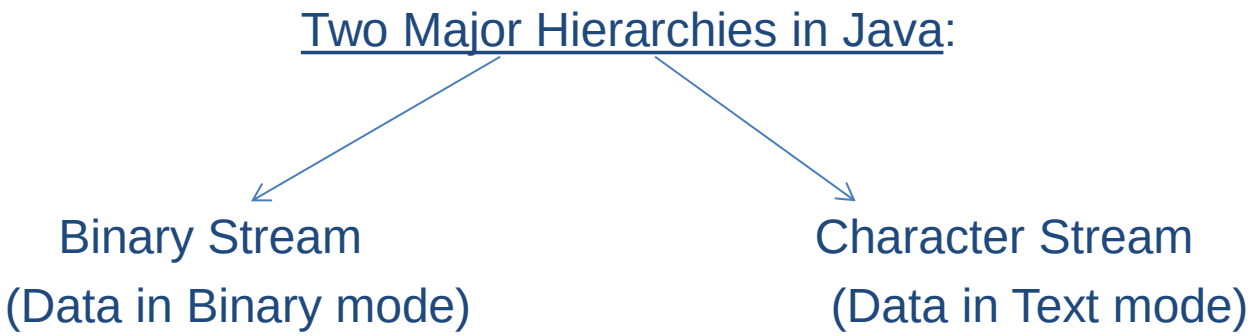
- A stream can be thought of as a flow of data from a source or to a sink.

- A source stream initiates the flow of data, also called an input stream.

- A sink stream terminates the flow of data, also called an output stream.

- Sources and sinks are both node streams.

IO



- Java technology supports two types of streams: *character and byte*.
- Input and output of *character* data is handled by *readers and writers*.
- Input and output of byte data is handled by *Input streams and output streams*:
- Normally, the term stream refers to a byte stream.
- The terms reader and writer refer to character streams.

Stream Classes

- Fundamental Stream Classes:

InputStreamReader
&
OutputStreamWriter

The InputStream Methods

The InputStream Methods:

- The three basic read methods are:

```
int read()

int read(byte[] buffer)

int read(byte[] buffer, int offset, int length)
```

The OutputStream Methods

The OutputStream Methods:

- The three basic write methods are:

  void write(int c)

  void write(byte[] buffer)

  void write(byte[] buffer, int offset, int length)

- Other methods include:

  void close()

  void flush()

The Reader Methods

The Reader Methods

- The three basic read methods are:

```
int read()

int read(char[] cbuf)

int read(char[] cbuf, int offset, int length)
```

- The Reader class is an abstract class for reading character streams which is available in the java.io package.
- An InputStreamReader is a bridge from byte streams to character streams i.e. it reads bytes and decodes them into Unicode characters according to a particular platform
- The BufferedReader class is the subclass of the Reader class. It reads character-input stream data from a memory area.

The Writer Methods

The Writer Methods

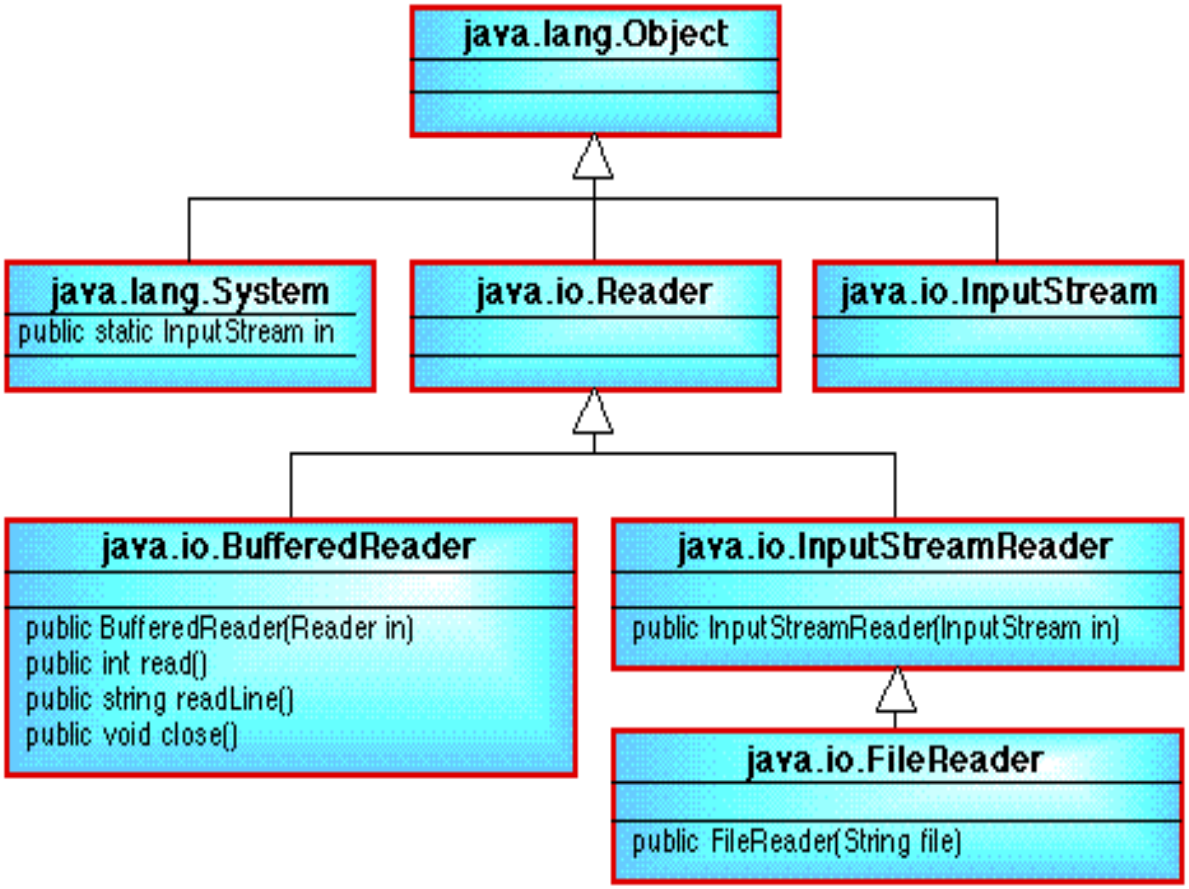
- The basic write methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

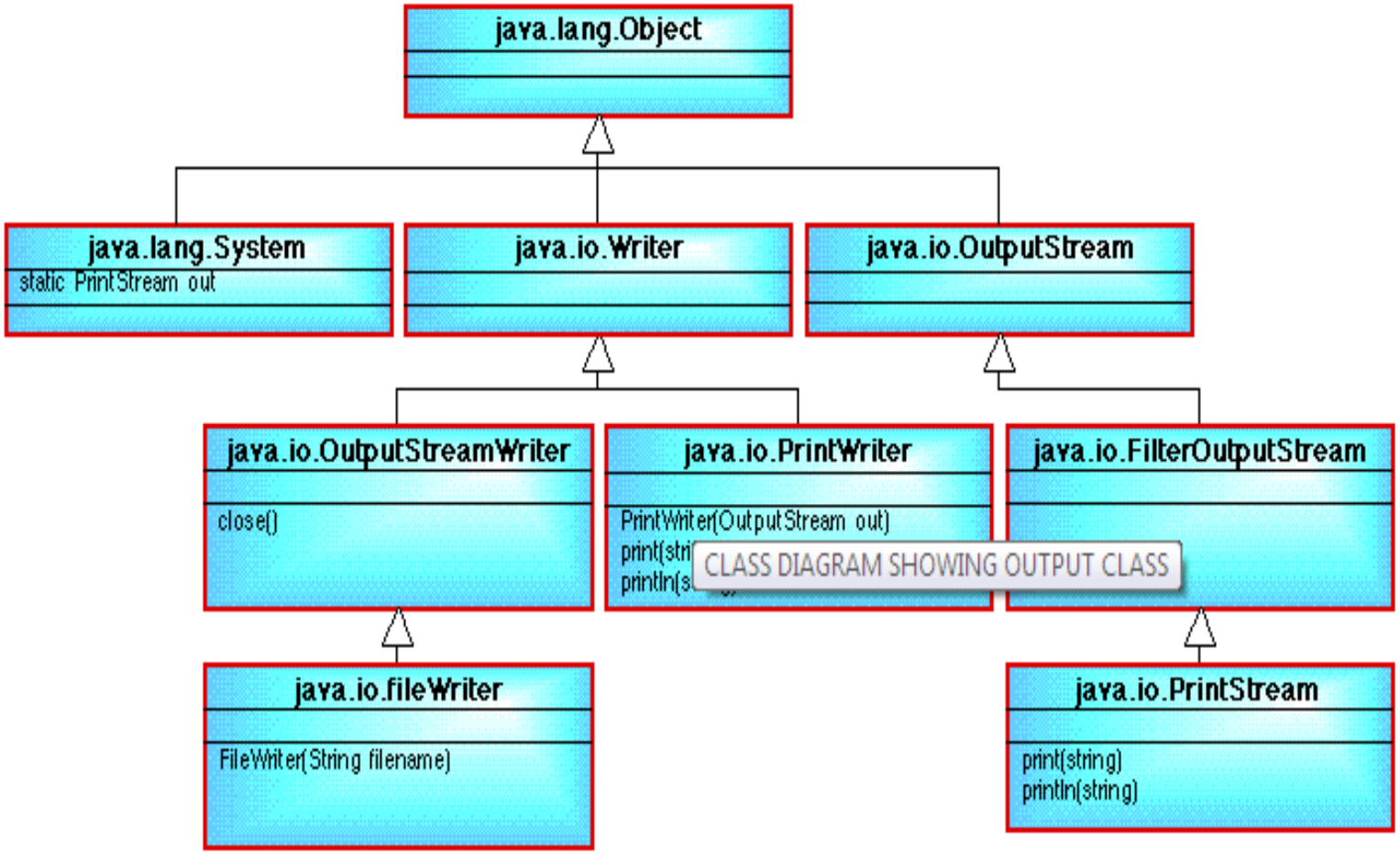
- Other methods include:

```
void close()
void flush()
```

Reader Classes



Writer Classes



File Stream I/O

File Stream I/O

- For file input:

  Use the FileReader class to read characters.
  Use the BufferedReader class to use the readLine method.

- For file output:

  Use the FileWriter class to write characters.
  Use the PrintWriter class to use the print and println methods.

Files and File I/O

The java.io package enables you to do the following:

- Create File objects
- Manipulate File objects
- Read and write to file streams

Files and File I/O

Write to file

At this point we have a writer object and we can send real content to the file. You can do this using the write() method, which has more variant but the most commonly used requires a string as input parameter.

Calling the write() method doesn't mean that it immediately writes the data into the file. The output is maybe cached so if you want to send your data immediately to the file you need to call the flush() method.

As last step you should close the file with the close() method and you are done.

Standard Streams

Standard Streams : feature provided by many operating systems.

- By default, they read input from the *keyboard*
- write output to display.
- They also support I/O operations on files.

- *System.in* : used to read input from the keyboard.
- *System. Out* : used to write output to display.
- *System. Err* : used to write error output

System.in is a byte stream that has no character stream features.

To use Standard Input as a character stream, wrap System.in within the InputStreamReader as an argument.

InputStreamReader inp = new InputStreamReader(system.in);

Scanner

- Scanner : a final class introduced in Java 5.0
- The scanner API provides basic input functionality for reading data from the system console or any data stream
- java.util.Scanner class can be used to accomplish the same thing but with less code
- This is an enhanced input functionality, as it makes easy to read primitive data types from the keyboard
- A simple text scanner which can parse primitive types and strings using regular expressions
- A Scanner breaks its input into tokens using a delimiter pattern.
  (default: whitespace)

Stream Enlayering & handling primitive data

Apart from Binary or Text mode, data can also be read or written in ***primitive data type format*** like : int ,float ..
DataInput & DataOutput Streams are provided for this purpose.

If you want to read or write ***primitive data*** to file then, choose appropriate File Stream object & Enwrap it into DataStream object.
This is known as ***enlayering***.

Methods in DataInputStream: readInt() ,readFloat() ,readUTF()….

Methods in DataOutputStream: writeInt() ,writeFloat() , writeUTF()….

Demo :Layering

```
public static void main(String[] args) throws IOException
 {
  DataOutputStream dos=new DataOutputStream(new
    FileOutputStream("MyData"));

    dos.writeInt(1);
    dos.writeFloat(2.2f);
    dos.writeUTF("hi");

  DataInputStream dis=new DataInputStream(new
    FileInputStream("MyData"));

   System.out.println(dis.readInt());
  System.out.println(dis.readFloat());
  System.out.println(dis.readUTF());
   }
```

Serialization & Deserialization

Java supports Persistence

The state of Java Objects can be stored permanently (to some file….)
Advantage: Future Reuse.

Serialization & Deserialization :

It is the process of saving an object's state to a sequence of bytes
&
the process of rebuilding those bytes into a live object at some future time

Streams & methods provided:
ObjectOutputStream : writeObject() ……. Serialization

ObjectInputStream : readObject() ……. Deserialization

Serializable : A tagging interface

- Object Serialization is not possible if class does not implement ***Serializable*** interface.
- In such case, ***NotSerializableException*** will be thrown.

Networking Introduction

- The Internet is all about connecting machines together.
- One of the most exciting aspects of Java is that it incorporates an easy-to-use, cross-platform model for network communications.

What are sockets ? What is Port no ?

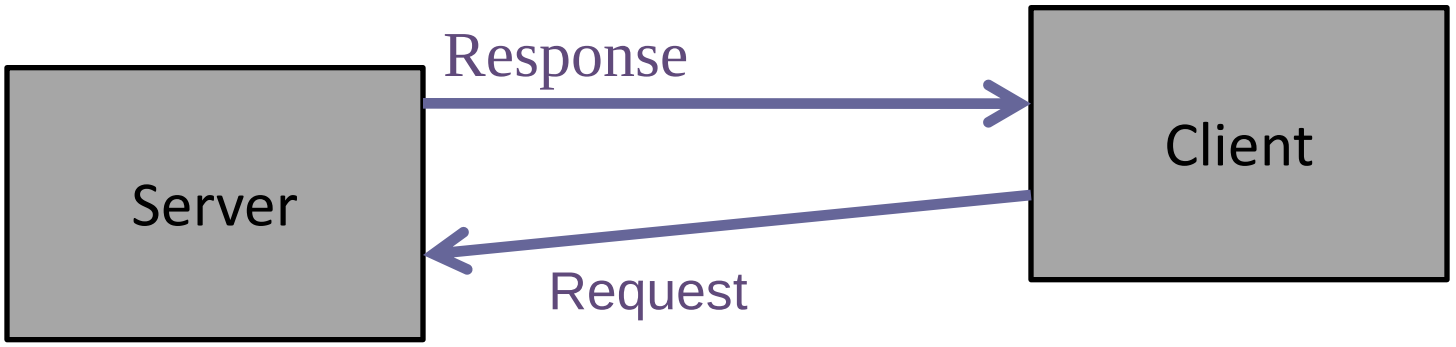
- The Internet Protocol breaks all communications into packets, finite-sized chunks of data which are separately and individually routed from source to destination.
- Socket is an abstraction that is provided to an application programmer to send or receive data to another process.
- Data can be sent to or received from another process running on the same machine or a different machine.
- Socket is a combination of IP Address & Port number
- A port number is a logical no. assigned to a process to identify it on a given system
- Port number up to 1024 are reserved port numbers. Also known as well-known ports.

Transmission Control Protocol (TCP)

- Transmission Control Protocol, TCP, establishes a connection between the two endpoints.
- The socket remains open throughout the duration of the communications.
- TCP is responsible for
  - breaking the data into packets,
  - buffering the data,
  - resending lost or out of order packets,
  - acknowledging receipt,
  - controlling rate of data flow.

Client/Server Computing

- You can use the Java language to communicate with remote file systems using a client/server model.
- A server listens for connection requests from clients across the network or even from the same machine.
- Clients know how to connect to the server via an IP address and port number.



Client/Server Computing(contd)

Creating TCP Clients

To create a TCP client, do the following:

1. Create a Socket object attached to a remote host, port.
   Socket client = new Socket(host, port);
   When the constructor returns, you have a connection.

2. Get input and output streams associated with the socket.

Client/Server Computing(contd)

Creating TCP Servers

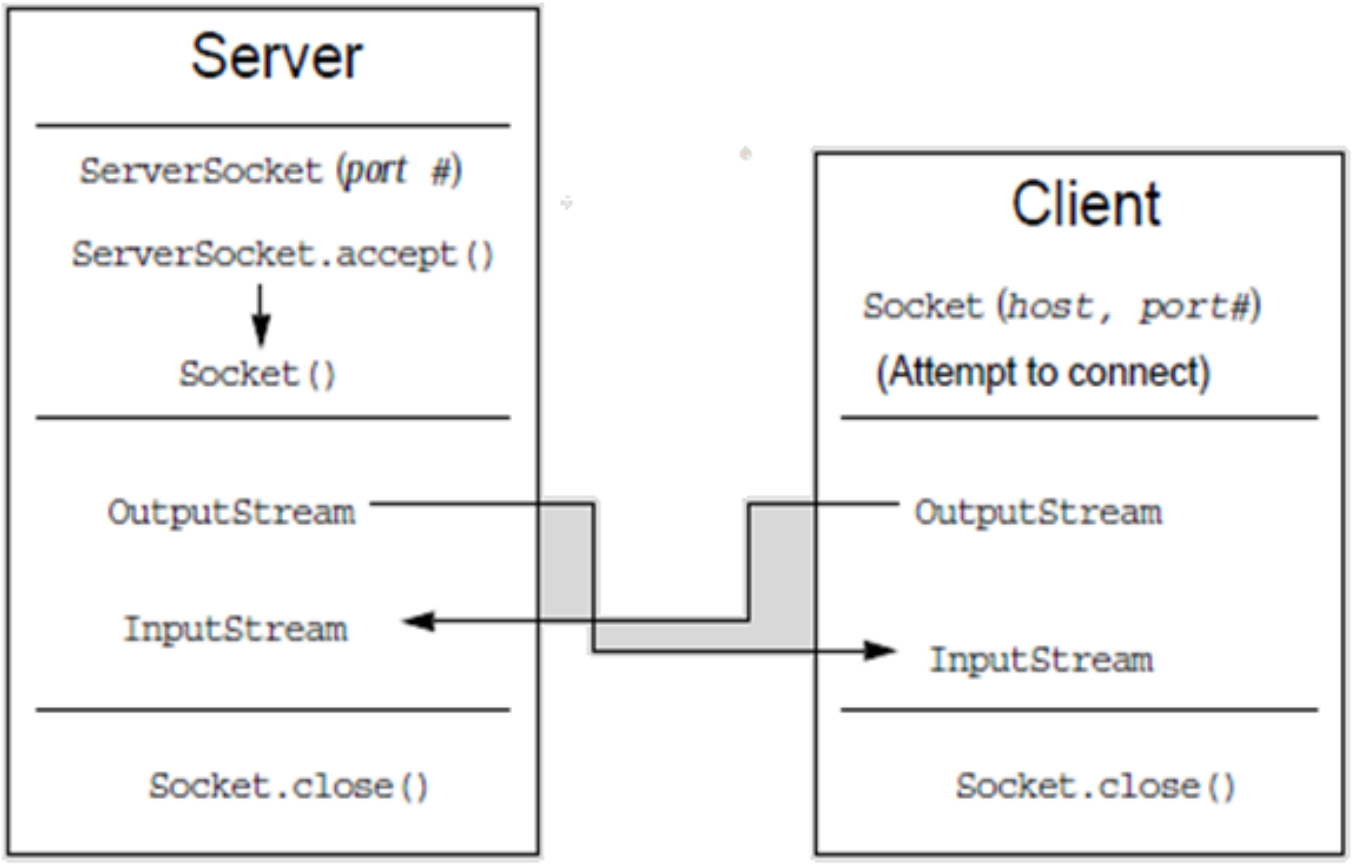
To create a TCP server, do the following:

1. Create a ServerSocket attached to a port number.
   ServerSocket server = new ServerSocket(port);

2. Wait for connections from clients requesting connections to that port.
   Socket channel = server.accept();
   You'll get a Socket object as a result of the connection.

3. Get input and output streams associated with the socket.

Client/Server Computing(contd)



Client/Server Computing(contd)

Client must contact server:

- server process must first be running.
- server must have created socket (door) that welcomes client's contact.

Client contacts server by:

- creating client-local TCP socket.
- specifying IP address, port number of server process.
- When client creates socket: client TCP establishes connection to server TCP.

When contacted by client, server TCP creates new socket for server process to communicate with client and allows server to talk with multiple clients on the basis of source port numbers.

Java TCP Socket Programming

Part of the java.net package

- import java.net.*;
- Provides two classes of sockets for TCP
- Socket – client side of socket
- ServerSocket – server side of socket

Java TCP Socket Programming

ServerSocket performs functions bind and listen

- Bind – fix to a certain port number
- Listen – wait for incoming requests on the port

Socket performs function connect

- Connect – begin TCP session

A simple Server Example

```
import java.net.*;
import java.io.*;
public class SimpleServer {
public static void main(String args[]) {
    ServerSocket  s = null;
    // Register your service on port 5131
    try    {
                s = new ServerSocket(5131);
                } catch (IOException e) {
                //ignore
                }
```

A simple Server Example(contd)

```
// Run the listen/accept loop forever
while (true) {
   try {
            // Wait here and listen for a connection
            Socket s1 = s.accept();
            // Get output stream associated with the socket
            OutputStream s1out = s1.getOutputStream();
            BufferedWriter bw = new BufferedWriter(
            new OutputStreamWriter(s1out));
            // Send your string!
            bw.write("Hello Net World!\n");
```

A simple Server Example(contd)

```
                // Close the connection, but not the server socket
                bw.close();
                s1.close();
                } catch (IOException e) {
                // ignore
                } // try-catch end
} // while end
} // main method end
} // SimpleServer program end
```

A simple Client Example

```
import java.net.*;
import java.io.*;
public class SimpleClient {
public static void main(String args[]) {
try {
// Open your connection to a server, at port 5131 localhost used here
Socket s1 = new Socket("127.0.0.1", 5432);
// Get an input stream from the socket
InputStream is = s1.getInputStream();
// Decorate it with a "data" input stream
DataInputStream dis = new DataInputStream(is);
// Read the input and print it to the screen
System.out.println(dis.readUTF());
```

A simple Client Example(contd)

```
// When done, just close the steam and connection
dis.close();
s1.close();
} catch (ConnectException connExc) {
System.err.println("Could not connect.");
} catch (IOException e) {
// ignore
} // try-catch end
} // main method end
} // SimpleClient program end
```

Assignments

Any Questions?