Socket and Serialization

Comp-303 : Programming Techniques

Lecture 12

Alexandre Denault
Computer Science
McGill University

Winter 2004

Announcements

- Midterm is in one week.
- I'll try to post some information on the midterm this weekend.
- Don't forget that the midterm is in Leacock 26.
- The last class for midterm material is today.

Last lecture . . .

- Java provides many tools to implement threading behavior.
- When implementing threads, you have the choice between extending Thread and implementing Interface.
- Methods such as *yield()*, *sleep()*, *wait()*, *notify()* and *notifyAll()* allow you to control the behavior of your threads.
- We have barely scratched the surface: timers, thread groups, priorities, etc.
- There are many more issues you have to deal with when programming concurrent behavior: race condition, atomicity, sharing, etc.
- If you're interested in learning more about concurrency, check out Comp-409.

References and additional material

- This course is *heavily* inspired from the Sun's Socket Tutorial and Sun's IO Tutorial (i.e. a lot of material was taken directly from the tutorial):

  `http://java.sun.com/docs/books/tutorial/networking/sockets/`

  `http://java.sun.com/docs/books/tutorial/essential/io/serialization.html`

- A good (but advanced and expensive) book on sockets would be:

  `Unix Network Programming`

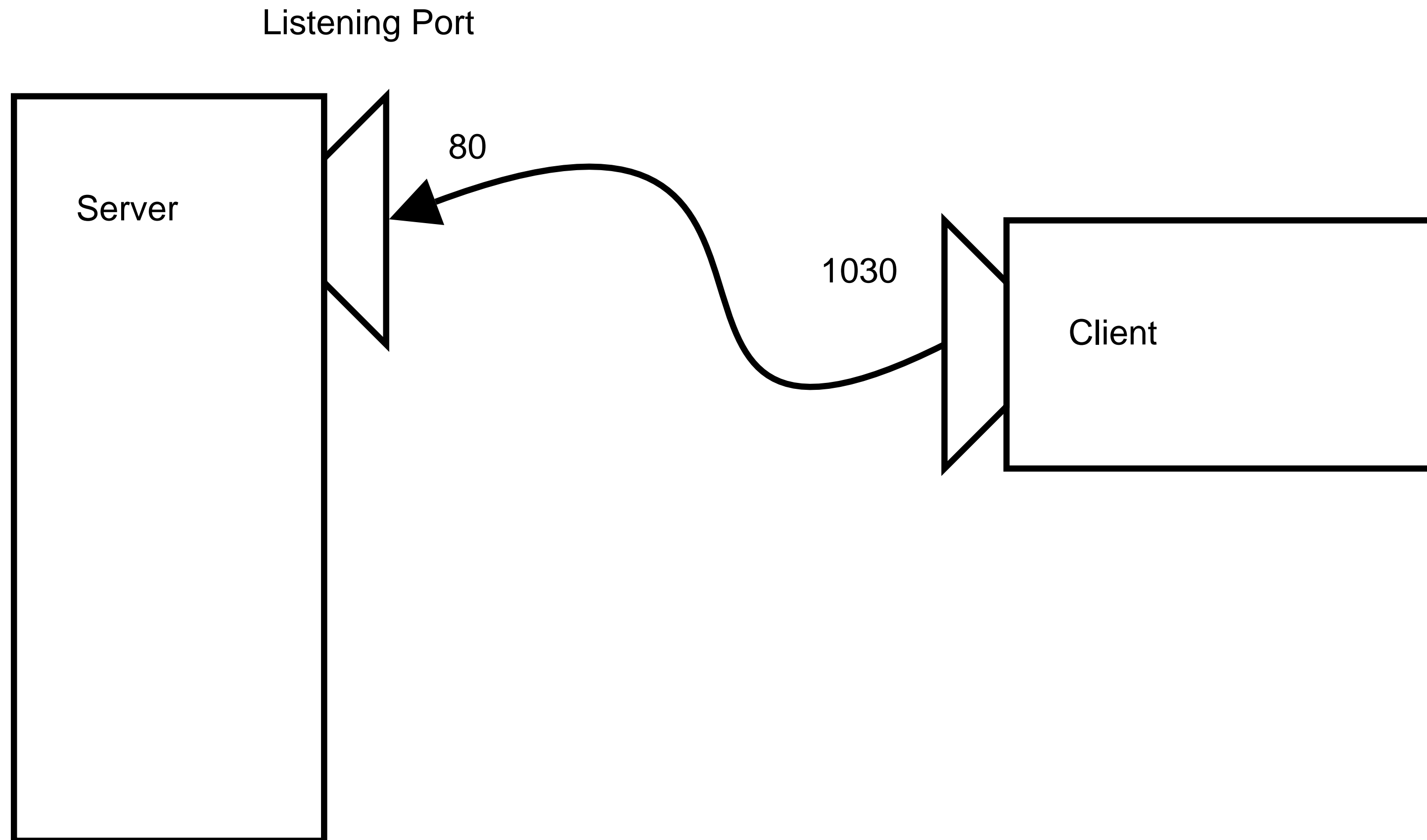
  `W. Richard Stevens`

  `Prentice Hall PTR`

Network Sockets

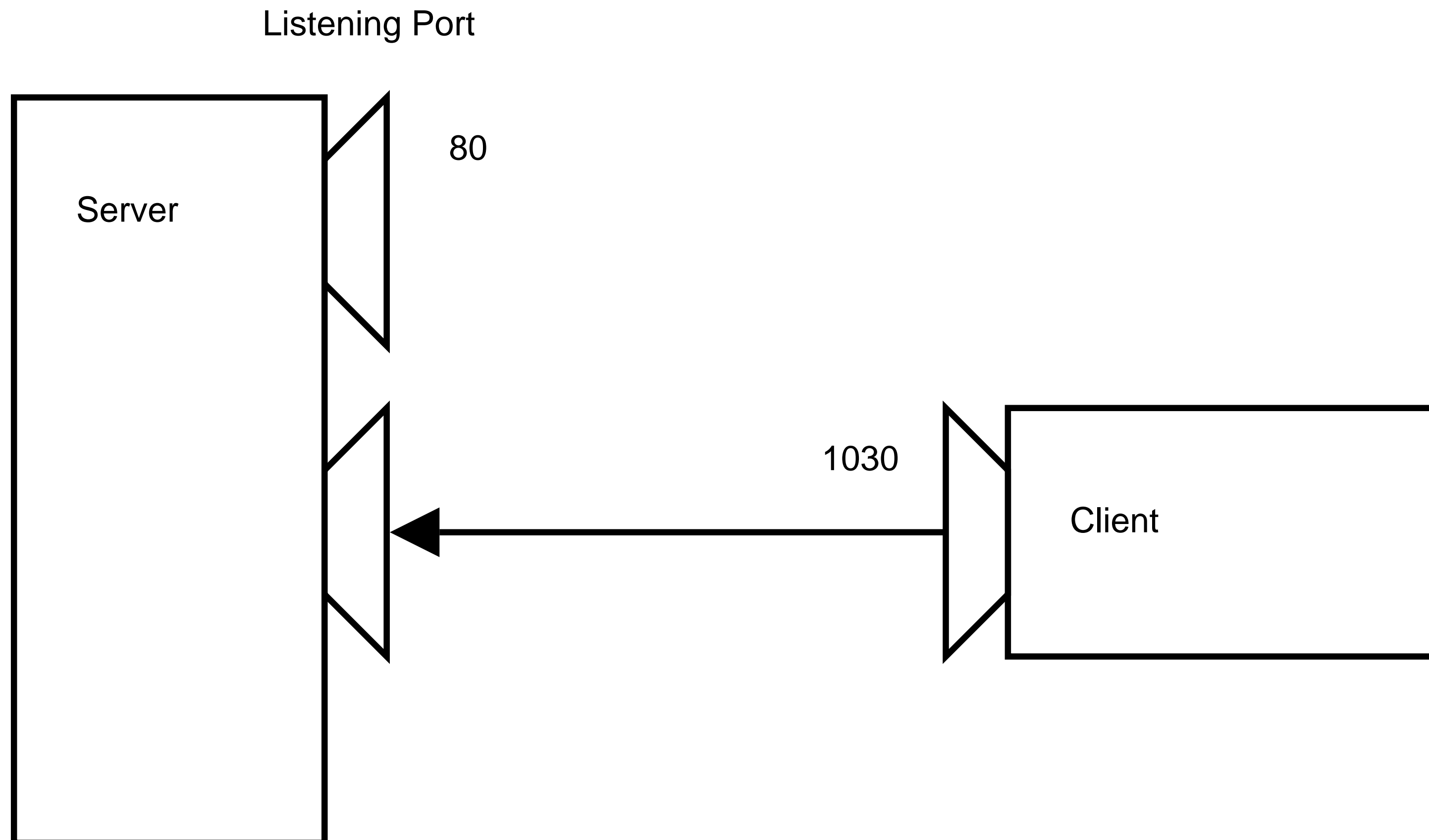
- To understand Java sockets, we must first understand TCP/IP sockets.
- Every unique machine has a unique address called an IP address. (ex: 132.206.51.234 is the CS mail server)
- IP address are hard to remember, so we also have the domain name system (ex: mail.cs.mcgill.ca)
- Every machine has a fixed number of ports (65536).
- Ports allows us to recognize IP data from different applications.
- The port range is divided as follows
  - 0-1023: The Well Known Ports
  - 1024-49151: The Registered Ports
  - 49152-65535: The Dynamic and/or Private Ports

Client requests connection



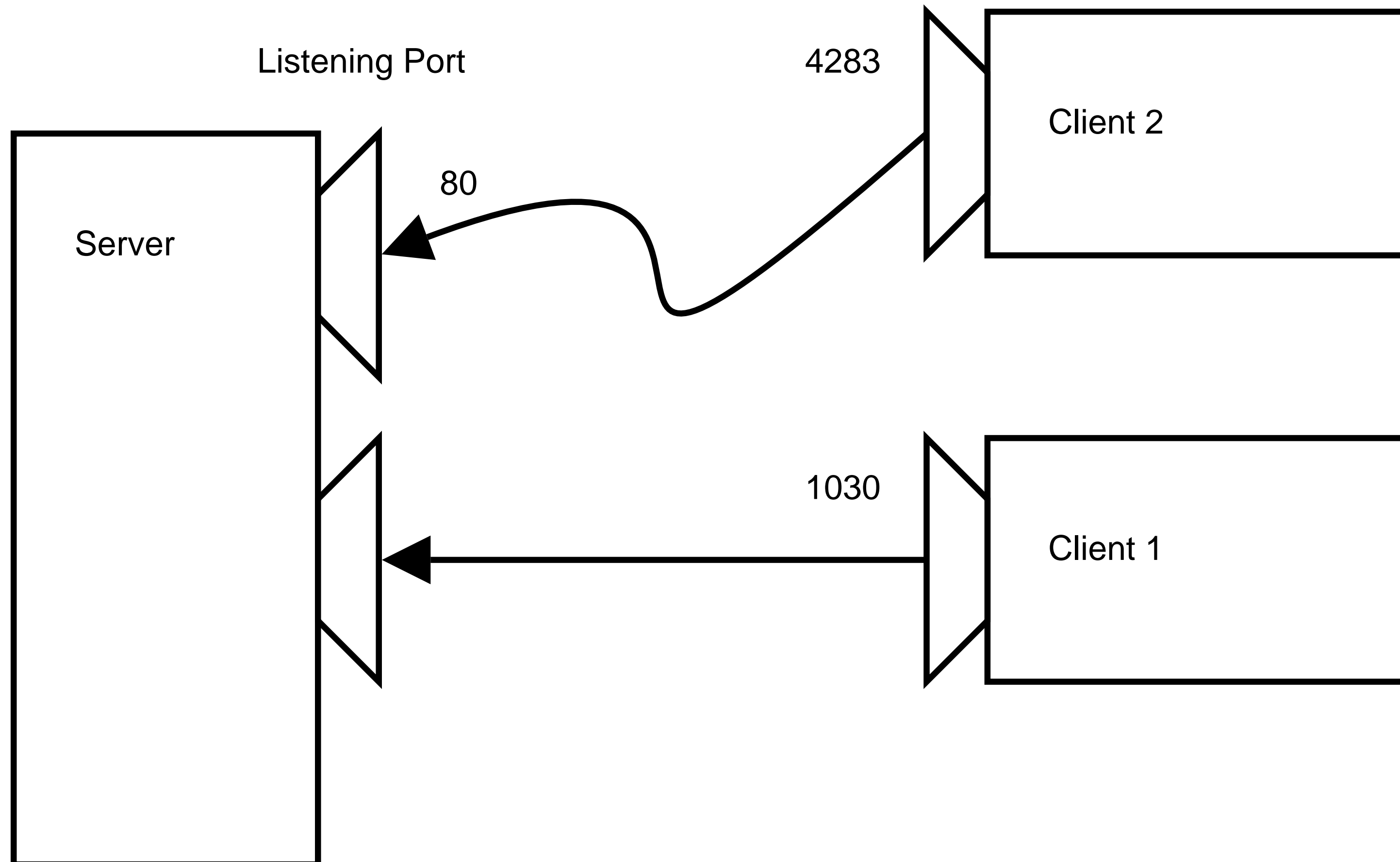
To make a connection request, the client tries to rendezvous with the server on the server's machine and port.

Server accepts connection



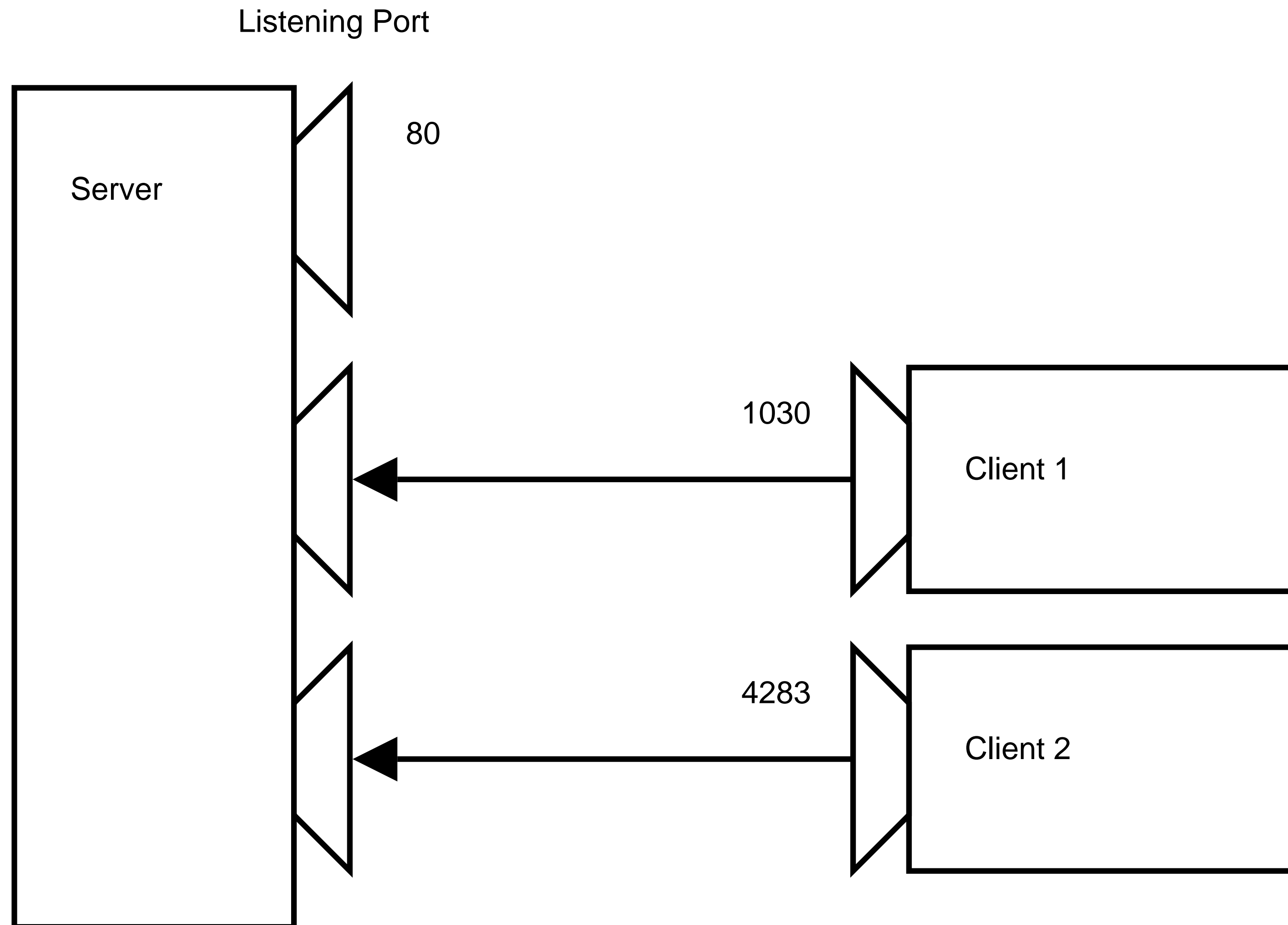
Upon acceptance, the server gets a new socket.

Another client requests connection



It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

Server accepts connection



Important listening ports

- 20/21 : File transfer protocol (FTP)
- 22 : Secure Shell (SSH)
- 23 : Telnet
- 25 : Simple Mail Transfer Protocol (SMTP)
- 80 : World Wide Web (HTTP)
- 137/138/139 : NetBIOS (Microsoft File Sharing)
- 143 : Internet Mail Protocol (IMAP)
- 443 : HTTP protocol over TLS/SSL
- 2049 : NFS
- 2346-2349 : Redstorm Game Servers

Socket Communication

- The client and server can now communicate by writing to or reading from their sockets.
- So, what is a socket?

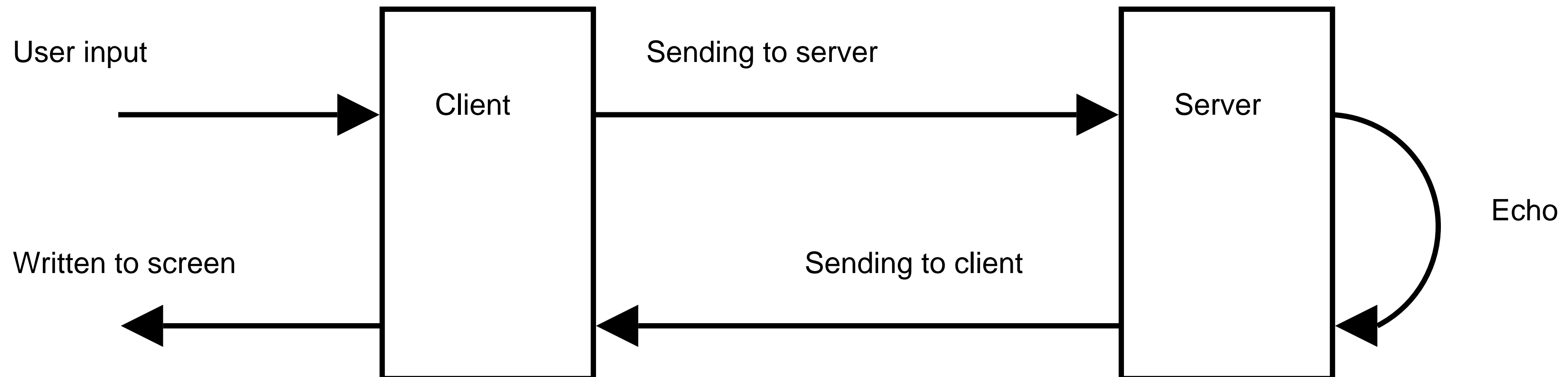
  > A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

- The *java.net* package in the Java platform provides the *Socket* and *ServerSocket* classes.
- Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program.

Example : Echo Client and Server

- The Echo server simply receives data from its client and echoes it back.
- *EchoClient* creates a socket thereby getting a connection to the Echo server.
- It reads input from the user on the standard input stream, and then forwards that text to the Echo server by writing the text to the socket.
- The server echoes the input back through the socket to the client.
- The client program reads and displays the data passed back to it from the server:

Echo Process



Echo Client

```
import java.io.*;
import java.net.*;

public class EchoClient {

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

      Socket echoSocket = null;
      PrintWriter out = null;
      BufferedReader in = null;
```

Echo Client

```
try {
   echoSocket = new Socket("taranis", 7);
   out = new PrintWriter(echoSocket.getOutputStream(), true);
   in = new BufferedReader(new InputStreamReader(
                                echoSocket.getInputStream()));
} catch (UnknownHostException e) {
   System.err.println("Don't know about host: taranis.");
   System.exit(1);
} catch (IOException e) {
   System.err.println("Couldn't get I/O for "
                                + "the connection to: taranis.");
   System.exit(1);
}
```

Echo Client

```
        String userInput;

        BufferedReader stdIn = new BufferedReader(
                                         new InputStreamReader(System.in));

        while ((userInput = stdIn.readLine()) != null) {
           out.println(userInput);
           System.out.println("echo: " + in.readLine());
        }

        out.close();
        in.close();
        stdIn.close();
        echoSocket.close();
    }
}
```

Step one: Connect to server

- The first step is to establish a connection with the server.

```
echoSocket = new Socket("taranis", 7);
```

- If the server is unreachable, an *UnknownHostException* is thrown.
- Next, we need to set up I/O.

```
out = new PrintWriter(echoSocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
```

- To send data to the server, we use a *PrintWriter* (which allows us to write to an output stream).
- To receive data, we use a *BufferedReader* (like the one we use to read from STDIN).
- If we can't set up our I/O, an *IOException* is thrown.

Step Two: Read from STDIN

- We can read from STDIN using a *BufferedReader* object.

```
BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));

String userInput;

while ((userInput = stdIn.readLine()) != null) { ...
```

- Data read from STDIN is sent directly to the server.

Step Three: Sending and receiving

- Data is send to the server by writing to the output stream using the *PrintWriter* object.

```
out.println(userInput);
```

- Data is received by reading the input stream with the *BufferedReader*object.

```
System.out.println("echo: " + in.readLine());
```

Step Four: Closing the connection

- Once we are finished communicating with the server, we can close our socket.

```
out.close();
in.close();
stdIn.close();
echoSocket.close();
```

- First statement closes our output stream.
- Second statement closes our input stream.
- Third statement closes our link on the STDIN stream.
- Fourth statement closes the socket to the server (and the connection).
- Reading/Writing to a closed stream/socket causes an exception.

Echo Server

- Opening and closing a socket on a server is very similar to opening and closing a socket on a client.
- However, the server uses two types of socket
  - A *ServerSocket* to listen for new connections.
  - A regular *Socket* to communicate with the client.

Step One: Wait from incoming connection

```
try {
   serverSocket = new ServerSocket(4444);
catch (IOException e) {
   System.out.println("Could not listen on port: 4444");
   System.exit(-1)
}
```

- The following code sets up a server socket and waits for incoming connections on port 4444.

Step Two: Accepting a new connection

```
Socket clientSocket = null;

try {
   clientSocket = serverSocket.accept();
} catch (IOException e) {
   System.out.println("Accept failed: 4444");
   System.exit(-1);
}
```

- To accept a connection, the *accept()* method must be called.
- The *accept()* method is a blocking I/O call, it will not return until a new connection is established.
- Once the connection is established, the server can use the *Socket* object like we saw in the client example (I/O streams).

Step Three: Closing the server socket

- Once the server socket is closed, the server will not accept any incoming communication.

```
serverSocket.close();
```

- This call does not affect sockets that are already established.
- To disconnect clients from the server, each socket must be individually closed.

Supporting Multiple Clients

- The echo server we described can listen for and handle a single connection request.
- However, multiple client requests can come into the same port.
- Client connection requests are queued at the port, so the server must accept the connections sequentially.
- The server can service them simultaneously through the use of threads - one thread per each client connection.

```
while (true) {
   accept a connection ;
   create a thread to deal with the client ;
end while
```

UDP Sockets

- UDP sockets are outside the scope of this class.
- They work in a connectionless mode.
- UDP are much faster than typical TCP connections.
- UDP provides no error handling (detection, recovery, etc).

Warning about Sockets

- The Java *Socket* class sends data in plain text.
- If you want some improved security, you might want to look at *SSLSocket* which is more secure, but much more complicated to use.

Java Sockets

Last chance for questions about network sockets ...

What is Serialization

Serialization is the process of taking the memory data structure of an object and encoding it into a serial (hence the term) sequence of bytes. This encoded version can then be saved to disk, sent across a network connection, or otherwise communicated to a recipient.

(from Wikipedia.org)

Why do I need Serialization?

In Java, serialization can be used for 2 things:

- Remote Method Invocation (RMI)–communication between objects via sockets
- Lightweight persistence–the archival of an object for use in a later invocation of the same program

How do I use Serialization?

Java provides to objects in *java.io*

- ObjectInputStream
- ObjectOutputStream

Writing to an ObjectOutputStream

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

- *ObjectOutputStream* must be constructed on another stream.
- The writeObject method serializes the specified object, traverses its references to other objects recursively, and writes them all.
- The writeObject method throws a NotSerializableException if it's given an object that is not serializable.

Reading from an ObjectInputStream

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

- *ObjectInputStream* must be constructed on another stream.
- The objects must be read from the stream in the same order in which they were written.
- The readObject method deserializes the next object in the stream and traverses its references to other objects recursively to deserialize all objects that are reachable from it.

Data types in ObjectStreams

- *ObjectOutputStream* implements many methods for writing primitive data types, such as the *writeInt* method.
- *ObjectInputStream* also implements methods for reading primitive data types.
- The return value from readObject is an object that is cast to and assigned to a specific type.

Serialization over sockets

- I can build my *ObjectOutputStream* or *ObjectInputStream* over Socket stream.

```
socClient = new Socket(serverIp, serverPort);
socClient.setSoTimeout(10000);

socketOut = new ObjectOutputStream(socClient.getOutputStream());
socketIn = new ObjectInputStream(socClient.getInputStream());
socketOut.flush();
```

- Object serialization over sockets is identical to object serialization over files.

Providing Object Serialization for Your Classes

- An object is serializable only if its class implements the Serializable interface.

```
package java.io;
public interface Serializable {

};
```

- Making instances of your classes serializable is trivial. You just add the implements *Serializable* clause to your class declaration.

```
public class MySerializableClass implements Serializable { ...
```

- You don't need to add any methods. *ObjectOutputStream* and *ObjectInputStream* have default method for serialization.

Customizing Serialization

- Sometimes, default serialization can be slow, and a class might want more explicit control over the serialization.
- You can customize serialization for your classes by providing two methods for it: *writeObject* and *readObject*.
- Custom serialization is outside the scope of this class, but you can find more detail in the Sun's tutorial.

Protecting Sensitive Information

- When developing a class that provides controlled access to resources, you must take care to protect sensitive information and functions.
- Several techniques are available to protect sensitive data in classes.
- The easiest is to mark fields that contain sensitive data as private *transient*.
- *transient* and *static* fields are not serialized or deserialized.
- Sun's tutorial has additional information on protecting sensitive data.

Warning about Serialization

- The Java specifications did not provide serialization compatibility between JVMs (or even versions of JVM).
- However, Sun seems to have remove this warning from its documentation.
- Serialization *should* be compatible between JVM.

Summary

- Java Sockets work a lot like any other TCP/IP sockets.
- Java provides two socket objects : *Socket* and *ServerSocket*.
- Communicating over sockets is identical to reading/writing to files.
- Java provides serialization as a mean to save object and transmit them.
- *ObjectOutputStream* and *ObjectInputStream* can be used file any kind of streams (files, socket, etc).
- By implementing *Serializable*, your objects will be serializable.

Tool of the day: Google NewsGroups

- Google Groups contains the entire archive of Usenet discussion groups dating back to 1981.
- The database containing more than 800 million posts (few terabytes of data).
- This may be your most important resource when debugging.

  http://groups.google.ca/
- The Google 20 Year Usenet Timeline is worth reading.