

Introduction to Java™



Module 11: Networking

Network Programming



- ⌘ Historically, network programming has been **error-prone, difficult, and complex**.
- ⌘ The programmer had to **know many details about the network** and sometimes even the hardware.
- ⌘ You usually needed to understand the various “layers” of the networking protocol.
- ⌘ There were a lot of **different functions in each different networking library** concerned with
 - ⊠ connecting, packing, and unpacking blocks of information; shipping those blocks back and forth; and handshaking.

Network Programming



- ⌘ It was a daunting task.
- ⌘ However, the **concept of networking** is not so difficult.
 - ☑ You want to get some information from that machine over there and move it to this machine here, or vice versa.
- ⌘ **Similar to reading and writing files**
 - ☑ the file exists on a remote machine
 - ☑ the remote machine can decide exactly what it wants to do about the information you're requesting or sending.

Java Network Programming



- ⌘ One of Java's great strengths is **painless networking**.
- ⌘ The programming model you use is that of a file.
 - ☑ you actually wrap the network connection (a **"socket"**) with stream objects, so you end up using the same method calls as you do with all other streams.
- ⌘ Java's built-in multithreading: **handling multiple connections at once**.

Java Network Programming

⌘ Java uses the TCP/IP protocol

☒ The programmer doesn't see the details of TCP/IP.

⌘ Identifying a machine:

```
//: WhoAmI.java
// Finds out your network address when you're
// connected to the Internet.
import java.net.*;

public class WhoAmI {
    public static void main(String[] args) throws Exception {
        if(args.length != 1) {
            System.err.println(
                "Usage: WhoAmI MachineName");
            System.exit(1);
        }
        InetAddress a = InetAddress.getByName(args[0]);
        System.out.println(a);
    }
}
```

Sockets



- ⌘ The *socket* is the software abstraction used to represent the “**terminals**” of a connection between two machines.
- ⌘ For a given connection there's a socket on each machine.
 - ☑ In Java, you create a socket to make the connection to the other machine.
 - ☑ Then you get an **InputStream** and **OutputStream** from the socket in order to be able to treat the connection as an IO stream object.

Sockets



- ⌘ There are two stream-based socket classes:
 - ☒ a **ServerSocket** that a server uses to “listen” for incoming connections
 - ☒ and a **Socket** that a client uses in order to initiate a connection.
- ⌘ Once a client makes a socket connection, the **ServerSocket** returns a corresponding server side **Socket** through which direct communications will take place.
 - ☒ You have a true **Socket to Socket** connection.
 - ☒ Use **getInputStream()** and **getOutputStream()** to produce the corresponding **InputStream** and **OutputStream** objects from each **Socket**.
 - ☒ These **must be wrapped inside buffers and formatting classes** just like any other stream object.

Sockets



- ⌘ When you create a **ServerSocket**, you give it **only a port number**.
- ⌘ When you create a **Socket** you must give both the **IP address and the port number** where you're trying to connect.

A simple server and client



- ⌘ All the server does is **wait for a connection**
 - ☒ then uses the **Socket** produced by that connection to create an **InputStream** and **OutputStream**.
- ⌘ Then it reads from the **InputStream** and it echoes to the **OutputStream** until it receives the line END.
- ⌘ The client makes the connection to the server
 - ☒ then creates an **OutputStream**. Lines of text are sent through the **OutputStream**.
- ⌘ The client creates an **InputStream** to hear what the server is saying.
- ⌘ Both the server and client use the same port number and the client uses the local loopback address to connect to the server.

The Server



```
//: JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;

public class JabberServer {
    // Choose a port outside of the range 1-1024:
    public static final int PORT = 8080;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Started: " + s);
    }
}
```

The Server (cont)



```
try {
    // Blocks until a connection occurs:
    Socket socket = s.accept();
    try {
        System.out.println(
            "Connection accepted: "+ socket);
        BufferedReader in=new BufferedReader(
            new InputStreamReader(
                socket.getInputStream()));
        // Output is automatically flushed
        // by PrintWriter:
        PrintWriter out =
            new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(
                        socket.getOutputStream())),true);
```

The Server (cont)



```
while (true) {
    String str = in.readLine();
    if (str.equals("END")) break;
    System.out.println("Echoing: " + str);
    out.println(str);
}
// Always close the two sockets...
} finally {
    System.out.println("closing...");
    socket.close();
}
} finally {
    s.close();
}
}
```

The Server (explanation)

- ⌘ See that the **ServerSocket** just needs a port number, not an IP address.
- ⌘ When you call **accept()**, the method *blocks* until some client tries to connect to it.
 - ☑ When a connection is made, **accept()** returns with a **Socket** object representing that connection.
- ⌘ The next part of the program looks just like opening files for reading and writing.
 - ☑ Every time you write to **out**, its buffer must be flushed so the information goes out over the network.
- ⌘ The infinite **while** loop reads lines from the **BufferedReader in** and writes information to **System.out** and to the **PrintWriter out**.

The Client

```
//: JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;

public class JabberClient {
    public static void main(String[] args)
        throws IOException {
        // Passing null to getByName() produces the
        // special "Local Loopback" IP address, for
        // testing on one machine w/o a network:
        InetAddress addr = InetAddress.getByName(null);
        // Alternatively, you can use
        // the address or name:
        // InetAddress addr = InetAddress.getByName("127.0.0.1");
        // InetAddress addr = InetAddress.getByName("localhost");
    }
}
```

The Client (cont)

```
System.out.println("addr = " + addr);
Socket socket =
    new Socket(addr, JabberServer.PORT);
// Guard everything in a try-finally to make
// sure that the socket is closed:
try {
    System.out.println("socket = " + socket);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(
            socket.getInputStream()));
    // Output is automatically flushed
    // by PrintWriter:
    PrintWriter out = new PrintWriter(
        new BufferedWriter( new OutputStreamWriter(
            socket.getOutputStream())), true);
```

The Client (cont)



```
    for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
    }
    out.println("END");
} finally {
    System.out.println("closing...");
    socket.close();
}
}
```


The Client (explanation)


⌘ In **main()** you can see all three ways to produce the **InetAddress** of the local loopback IP address:

☑ using **null**, **localhost**, or the explicit reserved address **127.0.0.1**.

☑ To connect to a machine across a network you substitute that machine's IP address.

⌘ Note that the **Socket** called **socket** is created with both the **InetAddress** and the port number.

The Client (explanation)



- ⌘ Once the **Socket** object has been created, the process of turning it into a **BufferedReader** and **PrintWriter** is the same as in the server.
- ⌘ Note that the buffer must again be flushed
 - ☒ (which happens automatically via the second argument to the **PrintWriter** constructor).
 - ☒ If the buffer isn't flushed, the whole conversation will hang because the initial "howdy" will never get sent.
- ⌘ Each line that is sent back from the server is written to **System.out** to verify that everything is working correctly.

Summary



- ⌘ There's actually a lot more to networking than can be covered in this introductory treatment.
- ⌘ Java networking also provides fairly extensive support for URLs, including protocol handlers for different types of content that can be discovered at an Internet site.
- ⌘ You'll also get the portability benefits of Java so you won't have to worry about the particular platform the server is hosted on.
These and other features are fully and carefully described in *Java Network Programming* by Elliotte Rusty Harold (O'Reilly, 1997).