Network Computing Lecture 4

Professor Louise E. Moser

Spring 2012

Java URL Objects and Java Sockets

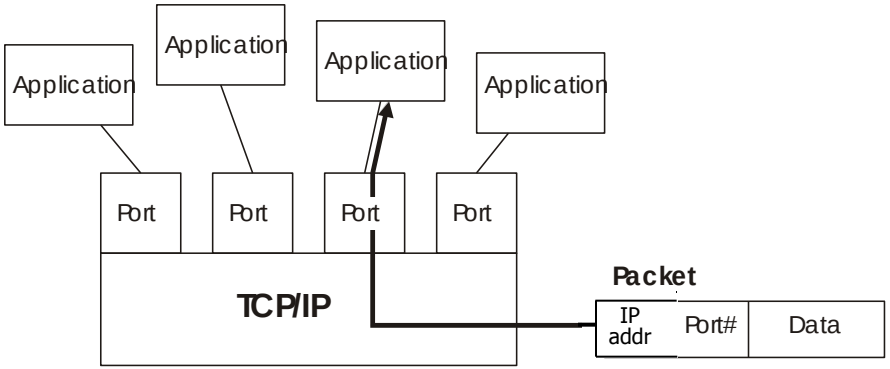
- **Java URL objects** are useful for reading a Web page
- **Java sockets** are useful when two programs must talk to one another, i.e., communicate over the network

Java Interface to Transport Layer

- Computers operating over the Internet use protocols at different layers
  - Application Layer - HTTP
  - Transport Layer - TCP, UDP
  - Network Layer - IP
- Java provides a *java.net* package that makes it easy to use TCP and UDP
- For reliable communication, use TCP
  - The *URL*, *URLConnection*, *Socket*, and *ServerSocket* classes in Java allow you to use TCP
- For unreliable communication with lower overheads, use UDP
  - The *DatagramPacket*, *DatagramSocket*, and *MulticastSocket* classes in Java allow you to use UDP
  - Note that many security firewalls do not allow you to use UDP

Ports

- A message transmitted over the Internet carries a 32-bit IP address for IPv4 (and a 128-bit IP address for IPv6) of the destination computer
- The message also carries a 16-bit port number that identifies a specific application within the computer at the destination IP address



- When you open a TCP/IP connection, a port is allocated at each end of the connection for exclusive use by that connection
- Ports 0 - 1023 are reserved for specific applications, such as email, http and ftp. They are called **well-known ports**. You should not use them.
- Ports 1024 - 65535 are free for all to use

Universal Resource Locators

- A **Universal Resource Locator** (URL) is an address of a resource on the Internet
- The Internet address is called a **URL address**
  - A Java object containing a URL address is called a **URL object**
- A URL address contains:
  - A **protocol id** such as *http* followed by *://*
  - A **resource name** that contains:
    - host name - the name of a machine on the Internet
    - file pathname - the pathname of a file on the host machine consisting of:
      - subdirectory names
      - file name
    - port number - for a specific application on the host machine (optional)
    - reference - for a specific location within a file (optional)
- Thus, *http://ece.ucsb.edu/Faculty/LouiseMoser/default.html*
  protocol id host name subdirectory names file name

Creating URL Objects in Java

- An absolute URL can be created from a String using a **URL constructor**

  URL games = new URL("http://www.games.com/");

- Once you have a URL, you can create others from it using a URL constructor with two arguments
  - Example:

    URL newgame = new URL(games,"newgame.html");

    concatenates the existing URL and the String to obtain

    http://www.games.com/newgame.html
- Many html files contain internal labeled reference points called **anchors**, indicated by \#
  - Example:

    URL newgameBottom = new URL(newgame,"\#BOTTOM");

    gives you *http://www.games.com/newgame.html\#**BOTTOM***

Creating URL Objects

- There is also a URL constructor that takes three strings: a **protocol**, a **machine name**, and a **path name**
  - Example:

    URL newgame =

    new URL("http", "www.games.com", "/newgame.html");

    yields

    http://www.games.com/newgame.html
- A fourth constructor adds a **port number** (1024 in the example)
  - Example:

    URL thegame =

    new URL("http", "www.games.com", 1024, "/newgame.html");

    yields

    *http://www.games.com**:1024**/newgame.html*
- URL objects are final (write-once). You cannot change them.

Query Methods on URL Objects

- *getProtocol()*
- *getHost()*
- *getPort()* (returns -1 if there is no port)
- *getFile()*
- *getRef()*

- You can also get a URL address back again as a String by using the *toString()* method
  - Example:

    s = toSring(myURL);

    yields

    s = "http://ece.ucsb.edu/Faculty/Moser/default.html"

Problems with URL Objects

- It is quite likely that you will generate a malformed or incorrect URL occasionally, and your program must be able to handle it
- Trying to create a malformed URL object or trying to connect to an incorrect URL will cause an exception to be raised
- Here is an example of exception handling with a try/catch pair

```
try {
    URL myURL = new URL(. . .)
}
catch (MalformedURLException e) {
    . . .
    // exception handler code here
    . . .
}
```

Problems with URL Objects

- This next example extends the previous one to handle the failure of an attempt to connect to a URL, which is quite common

```
try {
   URL yahoo = new URL(" ... ");
   URLConnection yc = yahoo.openConnection();
}
catch (MalformedURLException e) {
   . . .  // handle malformed URL
}
catch (IOException e) {
   . . .  // handle no connection
}
```

Reading from a URL Connection

- Once you have a URL connection, you can read from it and write to it using, for example, the *BufferedReader* class

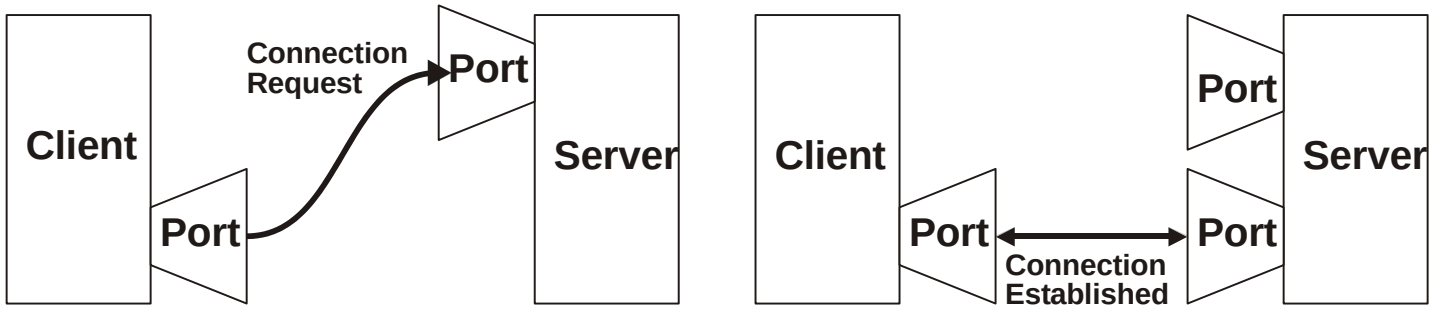
```
import java.net.*;
import java.io.*;

public class URLConnectionReader {
  public static void main(String[] args) throws Exception {
    URL yahoo = new URL("http://www.yahoo.com/");
    URLConnection yc = yahoo.openConnection();
    BufferedReader in = new BufferedReader( // gives textlines
                                new InputStreamReader( // gives chars
                                    yc.getInputStream()));

    String inputLine;
    while ((inputLine = in.readLine()) != null)
      System.out.println(inputLine);
    yc.close();
  }
}
```

Sockets in Java

- A **socket** is one endpoint of a two-way communication link between two programs running on the network
- A socket is bound to a specific **port number**
  - Incoming messages with that port number are delivered to that socket
- The *Socket* class, in the *java.net* package, implements the **client** side of a two-way connection between your Java program and another program on the network
- The *ServerSocket* class, also in the *java.net* package, implements the **server** side of a connection on which a server can listen for connection requests from clients, and accept connections to clients

Establishing a Connection

- The client initiates the connection and the server accepts the connection
  - There might be no difference between them subsequently
- The server has a port on which it listens for connection requests
- The client creates a port and sends a request to the server's port
- The server creates a new port and allocates it to a connection for this particular client
- The server then establishes the connection between this new port and the client's port
- This procedure allows the server to listen for, and to serve, multiple clients concurrently



The Client Side of a Socket Connection

- All client socket connections contain the same six steps:
  1. Open a socket to the server
  2. Open an output stream to the socket
  3. Open an input stream to the socket
  4. Write to the output stream and read from the input stream, according to the application-layer protocol agreed with the server
  5. Close the streams
  6. Close the socket
- Only step 4 differs from program to program, depending on the application-layer protocol agreed with the server

A Simple Example

- The client program opens a socket connection to another computer using port number 4444 to connect to an echo server program
- The client program opens an output stream to, and an input stream from, the port
- The client program reads text from the user's keyboard and sends it to the echo server program using the output stream
- The echo server program on the other computer sends the same text back
- The client program reads the text from the input stream and displays it
- Finally, the client program closes its streams and its socket

Example: The Client Side

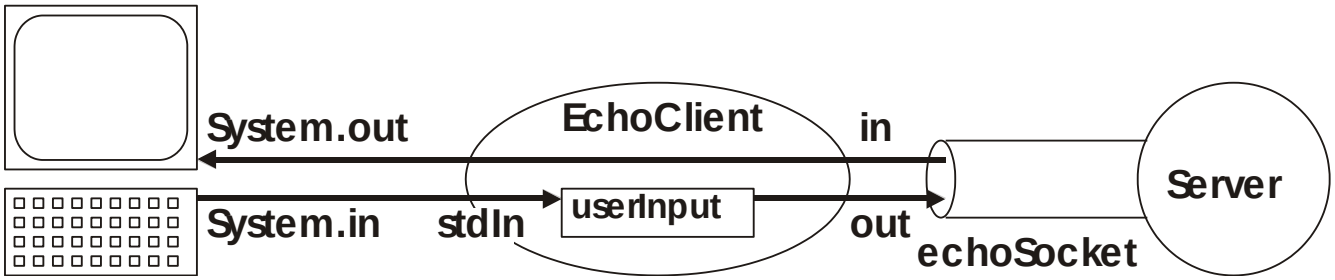
```
import java.io.*;
import java.net.*;

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

// First we declare and open the Socket, echosocket,
// and the Streams, out and in
Socket echoSocket = null;
PrintWriter out = null;
BufferedReader in = null;
```

Example: The Client Side (cont)

```
try {
  // Create a new Socket echoSocket and a new PrintWriter out
  // true tells PrintWriter to send out the line,
  // rather than to wait for the buffer to become full
  echoSocket = new Socket("alpha.ece.ucsb.edu", 4444);
  out = new PrintWriter(echoSocket.getOutputStream(), true);
  // Create a new BufferedReader in to read from the Socket echoSocket
  // via the intermediate InputStreamReader
  in = new BufferedReader(new InputStreamReader(
                                    echoSocket.getInputStream()));
}
catch (UnknownHostException e) {
  System.err.println("Don't know about alpha.ece.ucsb.edu.");
  System.exit(1);
}
catch (IOException e) {
  System.err.println("Can't connect to alpha.ece.ucsb.edu.");
  System.exit(1);
}
```

Example: The Client Side (cont)

```
// Sending and receiving using echoSocket

BufferedReader stdIn = new BufferedReader(
                                new InputStreamReader(System.in));
String userInput;
// Reads in lines of text from the keyboard, i.e., System.in
while ((userInput = stdIn.readLine()) != null) {
     // Sends the text out on the socket
     out.println(userInput);
    // Reads the text back in and then prints it
    System.out.println("echo: " + in.readLine());
}
```



Example: The Client Side (cont)

```
        // Closes the streams and the socket
            stdIn.close();
            out.close();
            in.close();
            echoSocket.close();
    }
}
```

- Well-behaved programs clean up after themselves
- However, note that you do not need to close *System.in* or *System.out*

Example: The Server Side

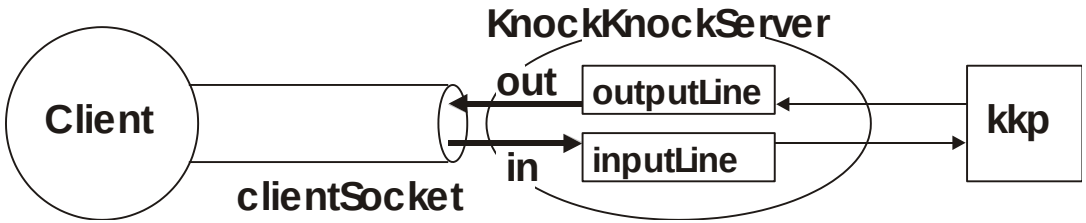
```
import java.io.*;
import java.net.*;

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

    ServerSocket serverSocket = null;
    try {
        // Server first opens a socket on port 4444
        serverSocket = new ServerSocket(4444);
    }
    catch (IOException e) {
        System.err.println("Could not listen on port: 4444.");
        System.exit(1);
    }
```

Example: The Server Side (cont)

```
Socket clientSocket = null;
  try {
      // Server invokes serverSocket.accept to wait for a client
      // When a client connects, serverSocket.accept returns
      // a new socket that the server can use to communicate with the client
     clientSocket = serverSocket.accept();
 }
catch (IOException e) {
      System.err.println("Accept failed.");
      System.exit(1);
}
```

Example: The Server Side (cont)

```
// To use the socket, first create a PrinterWriter and a BufferedReader
//  for the new clientSocket
PrintWriter out = new PrintWriter(
                              clientSocket.getOutputStream(), true);
BufferedReader in = new BufferedReader(
                                  new InputStreamReader(
                                       clientSocket.getInputStream()));

// Also create a new instance kkp of the protocol
String inputLine, outputLine;
KnockKnockProtocol kkp = new KnockKnockProtocol();
```



Example: The Server Side (cont)

```
// Communicating with the client

// Get the first message from the protocol kkp and
// send it to the client
   outputLine = kkp.processInput(null);
   out.println(outputLine);

// Until the input is a null or the output is "Bye"
// Get a message from the client via the socket
// in.readLine connects to clientSocket
while ((inputLine = in.readLine()) != null) {
    // Give message to the protocol kkp and get back a response
    outputLine = kkp.processInput(inputLine);
    // Send response to the client via the socket
    // outputLine connects to clientSocket
    out.println(outputLine);
    if (outputLine.equals("Bye"))
        break;
}
```

Example: The Server Side (cont)

```
// Close the streams and the sockets
        out.close();
        in.close();
        clientSocket.close();
        serverSocket.close();
 }
}
```