Helwan University

Faculty of Computer Science

Programming Language 2

Object oriented design using JAVA

Dr. Ayman Ezzat

Email: ayman@fcih.net

Web: www.fcih.net/ayman

Unit 9: Network Programming

Outline

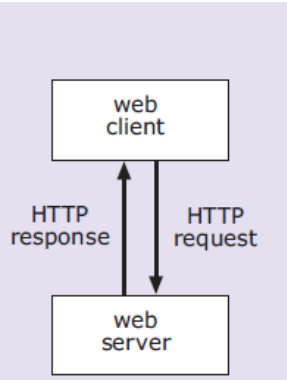
1. Background
2. Accessing the web using Java
3. Programming with sockets

1. Background

A distributed system is a collection of computers at different locations, connected by communications links. The functions and data of the system are distributed across these computers, which are known as **hosts**.

2-tier architecture

- **In 2-tier architecture (known as Client-server architecture)**, each computer in the system may be defined as **either a client or a server**.
  - **A server** is a computer that manages data, printers or network traffic.
  - **A client** is normally a PC or workstation on which users run application programs or user interface code. Clients request services from server.



A client–server system (2-tier system)

1. Background: Addressing on the internet

Identifying hosts on the web

- There are two ways of identifying a host on the internet:

  (1) Using a symbolic address, for example: www.google.com .
  - This address can be read as: The computer **www** belonging to The **Open** University, an **ac**ademic institution associated with the **UK** (United Kingdom).

  (2) Using a numeric address (IP addressing), for example: 193.22.33.201
  - This form of address uniquely identifies a computer by these four numbers.
  - Each number fits into 8 bits (so that each number must be in the range 0 to 255).
  - It is often known as dotted **quad notation** or **IP Version 4 (IPv4)** addressing.

- **DNS**: The part of the internet that keeps track of which computers are associated with which symbolic addresses is known as the Internet Domain Name Service, usually abbreviated to DNS.

1. Background: Addressing on the internet

Addressing resources on the web

- Resources on the web are identified using unique addresses. This unique address is known as a **Uniform Resource Locator (URL)**.
  - **Example: http://www.fcih.net/Pl2.php**
    - **http** → specifies the protocol to use is HTTP.
    - **www.fcih.net**→ the name of the host computer
    - The remainder of the URL is known as **the path**. In this case, it can be found in the folder **Staff**, a subfolder of **Computing**. The name of the file is **I_Newton.htm**.

In Java, the `URL` class may be used to create URLs

- **Here are some examples of creating URLs using different constructors:**

Example1:
```
URL OU = new URL("http://www.google.com");
```

Example2:
```
URL OU = new URL("http", "www.google.com", "");
```

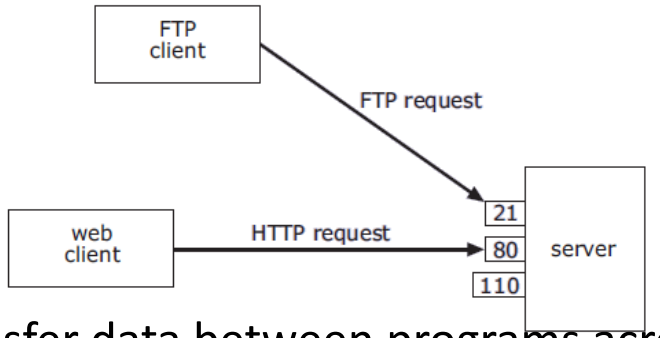
Example3:
```
URL OU = new URL("ftp","ftp.google.com",25, "/file.txt");
```

- In example3, the port number to be used is 25.

1. Background: Addressing on the internet

Ports and sockets

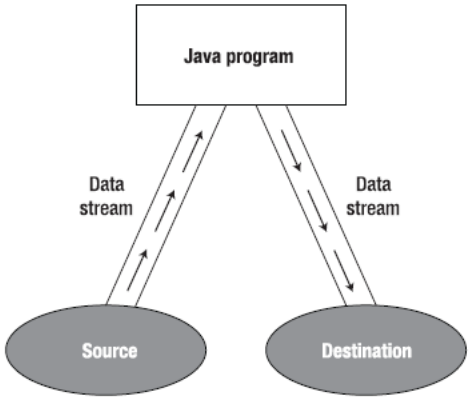
- **A port** is a conduit into a host on the internet.
  - TCP communication into and out of a computer is via numbered ports.
    - Ports numbered from 0 to 1023 are reserved for dedicated services.
      - For example, port 80 is used for web server communication and port 21 is used for FTP requests.
    - This allows a client to request a particular kind of service and allows servers to listen for particular kinds of requests on specific port numbers



- **A socket** is the software mechanism that allows programs to transfer data between programs across the internet.
  - A socket is a **logical idea** of a connection to a host on the internet
  - **A socket is unique** since it is **associated with:**
    - **a port number** and **IP address** of a host.
  - For client–server communication, we need a **matching pair of sockets**, one on the client and the other on the server.

1. Background: Remember from Unit4

- **A stream** is essentially a **sequence of bytes**, representing a **flow of data** from a **source** to a **destination**.
  - Sources and destinations include:
    - keyboard,
    - screen
    - various sorts of data files,
    - and between networked computers.



- **Any read or write is performed in three simple steps:**

  Step 1. Open the stream
    - **you need to define some objects here**

  Step 2. Until there is more data, keep reading in a read, or writing in a write.
    - **You need to use the methods of the objects defined in step1 to read the stream.**

  Step 3. Close the stream.

1. Background: Remember from Unit4

Writing text to a file using `PrintWriter` class

Step1a: create a File object that points to your text file.

Step1b: create PrintWriter object that points to your text file.

Step2: use the PrintWriter object to write text

Step3: close the PrintWriter when finished using it!

```
import java.io.*;
. . .
File myFile = new File("C:/aaa.txt");
PrintWriter pr = new PrintWriter(myFile);
pr.println("aaaaaa");
pr.println("bbbbbb");
pr.close();
```

1. Background: Remember from Unit4

Getting input using `BufferedReader` and another stream reader

- The **BufferedReader** **wraps another Reader** and **improves performance**.
  - Examples of readers are **InputStreamReader** to get input from keyboard and **FileReader** to get input from files**.**
- The **BufferedReader** provides a **readLine** method to read a line of text.

Example1: reading input from keyboard using **InputStreamReader**

Step1: open an input stream

Step2: get data

Step3: close the input stream

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
String s = stdin.readLine();
System.out.println(s);
stdin.close();
```

Example2: reading input from a File using **FileReader**

Step1: open an input stream

Step2: get data

Step3: close the input stream

```
BufferedReader in = new BufferedReader(new FileReader("C:/test.txt"));
String s;
while ((s = in.readLine()) != null)
    System.out.println(s + "\n");
in.close();
```

Outline

1. Background
2. Accessing the web using Java
3. Programming with sockets

2. Accessing the web using Java

URL class and `openStream` method

- The URL class has a method, `openStream()`, opens a stream to a web resource, e.g. web page, and allows the programmer to access the contents of the resource.
- **Example**: The following code is for a class called WebReader2 that accesses a website and displays the contents of the home page of the site.

```
import java.net.*;
import java.io.*;
public class WebReader2 {
    private BufferedReader fromWebSite;
    public WebReader2(String address) throws MalformedURLException, IOException {
        URL selectedURL = new URL(address);
        fromWebSite = new BufferedReader(new InputStreamReader(selectedURL.openStream()));
    }
    public void print() throws IOException {
        String in = fromWebSite.readLine();
        while (in != null) {
            System.out.println(in);
            in = fromWebSite.readLine();
            fromWebSite.close();
        }
    }
}
```

Notice the stream reader!

```
public class TestWebReader {
    public static void main(String[] args) throws MalformedURLException, IOException {
        WebReader2 wr = new WebReader2("http://www.google.com");
        wr.print();
    }
}
```

2. Accessing the web using Java

URL class and openStream method (Cont'd)

- The WebReader class could be written as follows with better exception handling.

```
import java.net.*; import java.io.*;
public class WebReader {
    private BufferedReader fromWebSite;
    public WebReader(String address) {
        try {
            URL selectedURL = new URL(address);
            fromWebSite = new BufferedReader(new InputStreamReader(selectedURL.openStream()));
        } catch (MalformedURLException me){System.out.println("Malformed URL found " + me);
        } catch (IOException io){System.out.println("Problems connecting " + io);}
    }
    public void print() {
        String in;
        try {
            in = fromWebSite.readLine();
            while (in != null) {
                System.out.println(in);
                in = fromWebSite.readLine();
            }
            fromWebSite.close();
        } catch (IOException io){System.out.println("Problems reading " + io);}
    }
}
```

- Why do the catch clauses in the constructor of the WebReader class need to be in that specific order? What would happen if they were swapped around?

Outline

1. Background
2. Accessing the web using Java
3. Programming with sockets

3. Programming with sockets

Each socket is unique since it consists of an **IP address** and a **port number**.

Sockets on the client

- The class `Socket` allows us to create sockets on the client system. The most common Socket constructor has the form:

```
Socket(String, int)
```

  - You have to put the address in the String part, and the port number in the int part. For ex.:

```
Socket s = new Socket("www.yahoo.com", 4000);
Socket s = new Socket("192.168.1.1", 4000);
```

- Communication between clients and a server is achieved by input/output streams.
  - Each socket has an associated input stream and output stream – the methods **`getInputStream`** and **`getOutputStream`** give access to these streams. For ex.:

```
InputStream in = s.getInputStream();
OutputStream out = s.getOutputStream();
```

  - These streams can then be used to send and receive data, using a similar approach to writing and reading files.

3. Programming with sockets

Sockets on the server

- The creation of sockets is **approached differently**. This is because the server does not normally initiate a connection – rather, it waits for a client to request a connection.
  - Hence it cannot create a socket until it knows the address of the client that wants to establish a connection
- The class `ServerSocket` is used within a server for setting up sockets. The class has two important constructors

  `ServerSocket(int)` **&** `ServerSocket(int, int)`

  - The port number should be given to the single argument constructor, while the (port_number, max_number_of_clients) should be given to the 2 arg. constructor. For ex.:

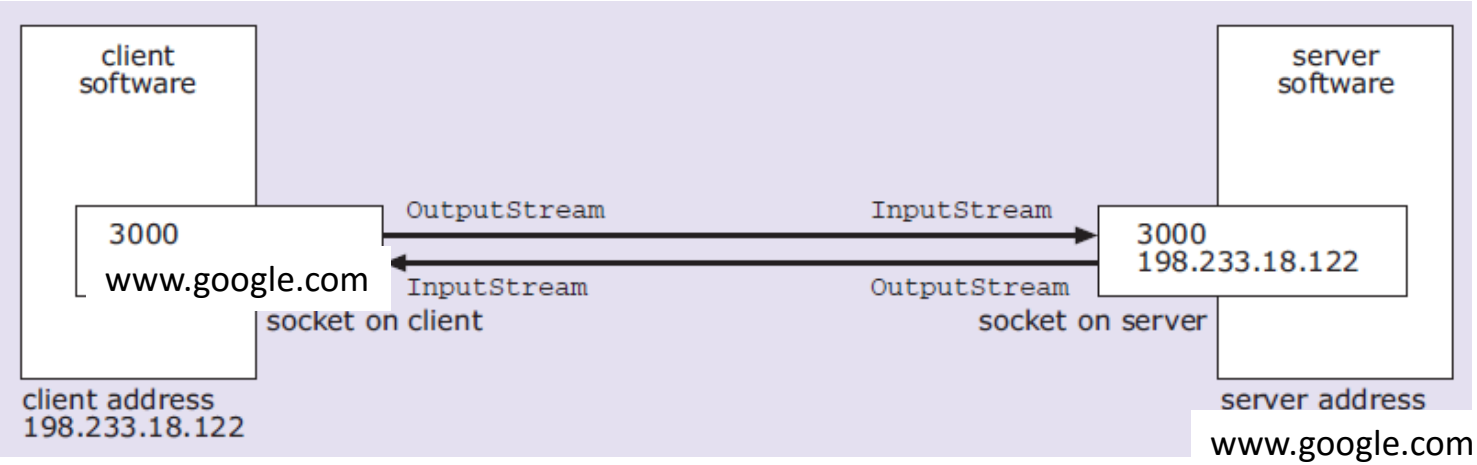
```
ServerSocket ss = new ServerSocket(80); //port 80
ServerSocket ss = new ServerSocket(80, 30); //port 80, max clients is 30
```

- Once a ServerSocket object is created, the server will wait for clients to request a connection. When a connection is made, a socket linked to the client is created; this is achieved via the `accept` method. For example:

```
// server code
ServerSocket ss = new ServerSocket(3000, 30);
...
// wait until connection request made by client
Socket sock = ss.accept();
```

3. Programming with sockets

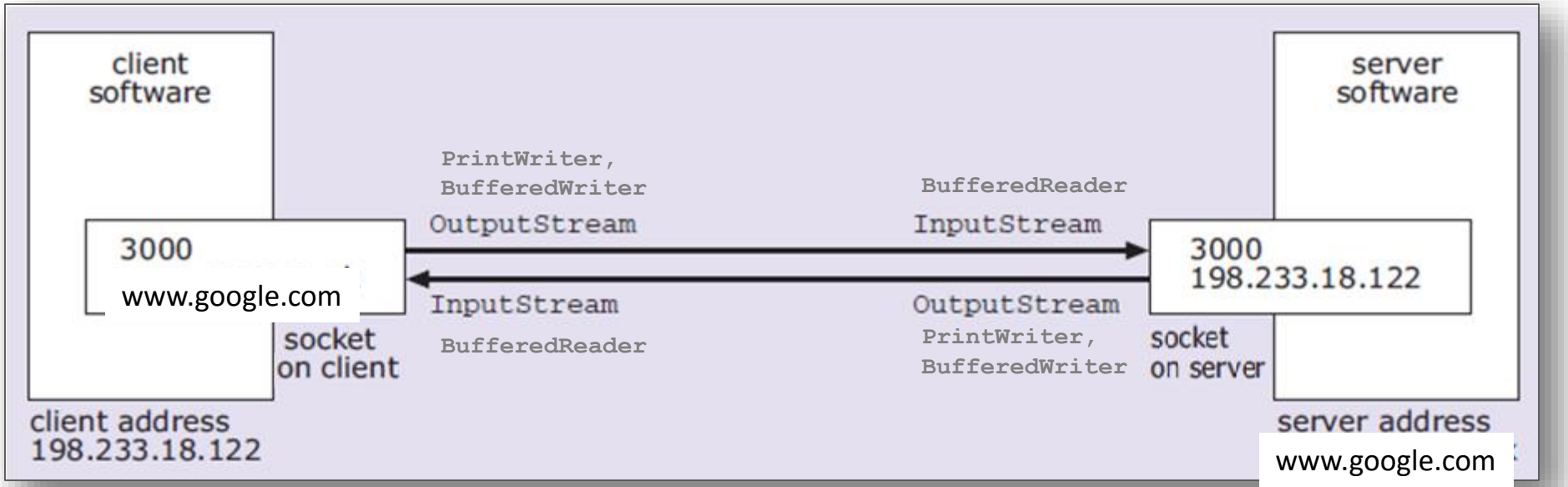
- The previous example could be illustrated as follows:



- In programming servers and clients you should bear in mind that:
  - A connection is a linked pair of sockets:
    - **at the server end**, the socket has the **address of the client** and a suitable **port** number for the service required by the client;
    - **at the client end**, the socket has the **address of the server** and the same **port** number as the server for the particular service required;
  - The InputStream entering the client receives data from the OutputStream leaving the server and the InputStream entering the server receives data from the OutputStream leaving the client.

3. Programming with sockets

HOW to send/receive data between a server and a client:



CLIENT program:

1 - Create socket (Server IP &Server port#)
2 - Open in/out streams
PrintWriter, BufferedReader, BufferedWriter + socket's getOutputStream(), getInputStream()
- Send/receive data
- Close in/out streams
3 - Close socket

SERVER program

1- Create ServerSokcet object with a port number
2- Open socket→ use accept() of ServerSocket object
3 - Open in/out streams
PrintWriter, BufferedReader, BufferedWriter + socket's getOutputStream(), getInputStream()
- Send/receive data
- Close in/out streams
4- Close socket

3. Programming with sockets

A simple client–server example

The client code!

```
import java.io.*; import java.net.*;
public class MyClient {
    private Socket socket;  // socket to server
    private InputStream is; // for communication to server
    private BufferedReader fromServer; // for communication to server
    static final int SERVER_PORT_NUMBER = 3000;
    public void run() {     // set up connection to the server
        try {
            //create socket with the server IP (localhost) & port#
            socket = new Socket("127.0.0.1", SERVER_PORT_NUMBER);
            //read from server using BufferedReader
            fromServer = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            System.out.println("Server said: " + fromServer.readLine());
            fromServer.close();
            //close socket
            socket.close();
        } catch (IOException e) {System.out.println("Trouble contacting the server " + e);}
    }
}
```

Summary of Client program

1- Create socket with IP/port#

2 - Open in/out streams

- Send/receive data

- Close in/out streams

3- Close socket

Don't forget to:

- Import appropriate libs

- handle errors!

Note:

you should run the server before running the client! (Why??)

```
public class TestMyClient {
    public static void main(String[] args) {
        MyClient client1 = new MyClient();
        client1.run();
    }
}
```

3. Programming with sockets

A simple client–server example

The server code!

USING

PrintWriter

Summary of Server program

1- Create ServerSokcet object

2- Open socket→ use accept()

3 - Open in/out streams

- Send/receive data

- Close in/out streams

4- Close socket

Don't forget to:

- Import appropriate libs

- handle errors!

```
import java.io.*;import java.net.*;
public class MyServer {
    private ServerSocket serverSocket;
    private Socket socket;          // socket to link to the client
    private OutputStream os;        // for communication to client
    private PrintWriter pwToClient;   // for communication to client
    static final int PORT_NUMBER = 3000; // must be higher than 1023
    public MyServer() {  // constructor
        try {
            serverSocket = new ServerSocket(PORT_NUMBER);
        } catch (IOException e) {System.out.println("Trouble on port "+ PORT_NUMBER + ": " + e);}
    }
    public void run() {     //in this method, the core work happens
        try {
            // wait for a connection request
            socket = serverSocket.accept();
            //use printwriter to send data to client
            pwToClient = new PrintWriter(socket.getOutputStream(), true);
            pwToClient.println("This is a message to client");
            pwToClient.close();
            //close socket
            socket.close(); //close the socket
        } catch (IOException e) {System.out.println("Trouble with a connection " + e);}
    }
}
```

```
public class TestMyServer {
    // create a server and have it greet the client
    public static void main(String[] args) {
        MyServer server1 = new MyServer();
        server1.run();
    }
}
```

3. Programming with sockets

A simple client–server example

The server code:

USING

BufferedWriter &

OutputStreamWriter

Summary of Server program

1- Create ServerSokcet object

2- Open socket→ use accept()

3 - Open in/out streams

- Send/receive data

- Close in/out streams

4- Close socket

Don't forget to:

- Import appropriate libs

- handle errors!

```
import java.io.*;import java.net.*;
public class NewServer {
    private ServerSocket serverSocket;
    private Socket socket;
    private PrintWriter pw;
    private BufferedWriter toClient;
    public NewServer() throws IOException {
        serverSocket = new ServerSocket(4000);
    }
    public void run() throws IOException {
        socket = serverSocket.accept();
        toClient = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
        toClient.write("hellllooooooo");
        toClient.close();
        socket.close();
    }
}
```

```
public class TestMyClient {
    public static void main(String[] args) {
        MyClient client1 = new MyClient();
        client1.run();
    }
}
```

3. Programming with sockets

A simple client–server example

The client code:

same as before but improved!

Summary of Client program

1- Create socket with IP/port#

2 - Open in/out streams

- Send/receive data

- Close in/out streams

3- Close socket

Don't forget to:

- Import appropriate libs

- handle errors!

Note:

you should run the server before running the client! (Why??)

```
import java.io.*; import java.net.*;
public class myClient {
    private Socket socket;  // socket to server
    private InputStream is; // for communication to server
    private BufferedReader fromServer; // for communication to server
    static final int SERVER_PORT_NUMBER = 3000;
    public void run() {     // set up connection to the server
        try {
            socket = new Socket("127.0.0.1", SERVER_PORT_NUMBER);
            openStreams();
            String messageFromServer = fromServer.readLine();
            System.out.println("Server said: " + messageFromServer);
            closeStreams();
            socket.close();
        } catch (IOException e) {System.out.println("Trouble contacting the server " + e);}
    }
    private void openStreams() throws IOException {
        is = socket.getInputStream();
        fromServer = new BufferedReader(new InputStreamReader(is));
    }
    private void closeStreams() throws IOException {
        fromServer.close();
        is.close();
    }
}
```

```
public class TestMyServer {
    // create a server and have it greet the client
    public static void main(String[] args) {
        MyServer server1 = new MyServer();
        server1.run();
    }
}
```

3. Programming with sockets

A simple client–server example

The server code:

same as before but improved!

Summary of Server program

1- Create ServerSokcet object

2- Open socket→ use accept()

3 - Open in/out streams

- Send/receive data

- Close in/out streams

4- Close socket

Don't forget to:

- Import appropriate libs

- handle errors!

```
import java.io.*;import java.net.*;
public class myServer {
    private ServerSocket serverSocket;
    private Socket socket;          // socket to link to the client
    private OutputStream os;        // for communication to client
    private PrintWriter pwClient;   // for communication to client
    static final int PORT_NUMBER = 3000; // must be higher than 1023
    public myServer() {  // constructor
        try {
            serverSocket = new ServerSocket(PORT_NUMBER);
        } catch (IOException e) {System.out.println("Trouble on port "+ PORT_NUMBER + ": " + e);}
    }
    public void run() {     //in this method, the core work happens
        try {
            socket = serverSocket.accept(); // wait for a connection request
            openStreams();  //a private method to open a stream to client
            pwClient.println("This is a message to client");
            closeStreams(); //close the stream
            socket.close(); //close the socket
        } catch (IOException e) {System.out.println("Trouble with a connection " + e);}
    }
    private void openStreams() throws IOException {
        os = socket.getOutputStream();
        pwClient = new PrintWriter(os, true);
    }
    private void closeStreams() throws IOException {
        pwClient.close();
        os.close();
    }
}
```

```
public class TestMyClient {
    public static void main(String[] args) {
        MyClient client1 = new MyClient();
        client1.run();
    }
}
```