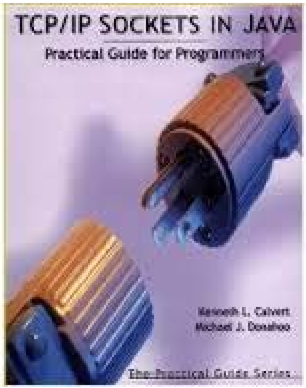
Socket Programming



Most slides are from web site of the textbook “Computer Networking: A Top Down Approach,” written by Jim Kurose and Keith Ross and “**TCP/IP Sockets in Java: Practical Guide for Programmers**”, written by Kenneth L. Calvert and Michael J. Donahoo

Outline

- What is a socket? (Revisited)
- Using sockets
  - Types (Protocols)
  - Associated functions
  - Styles

  - We will look at using sockets in JAVA
    - Note: C/C++ sockets are conceptually quite similar

What is a socket?

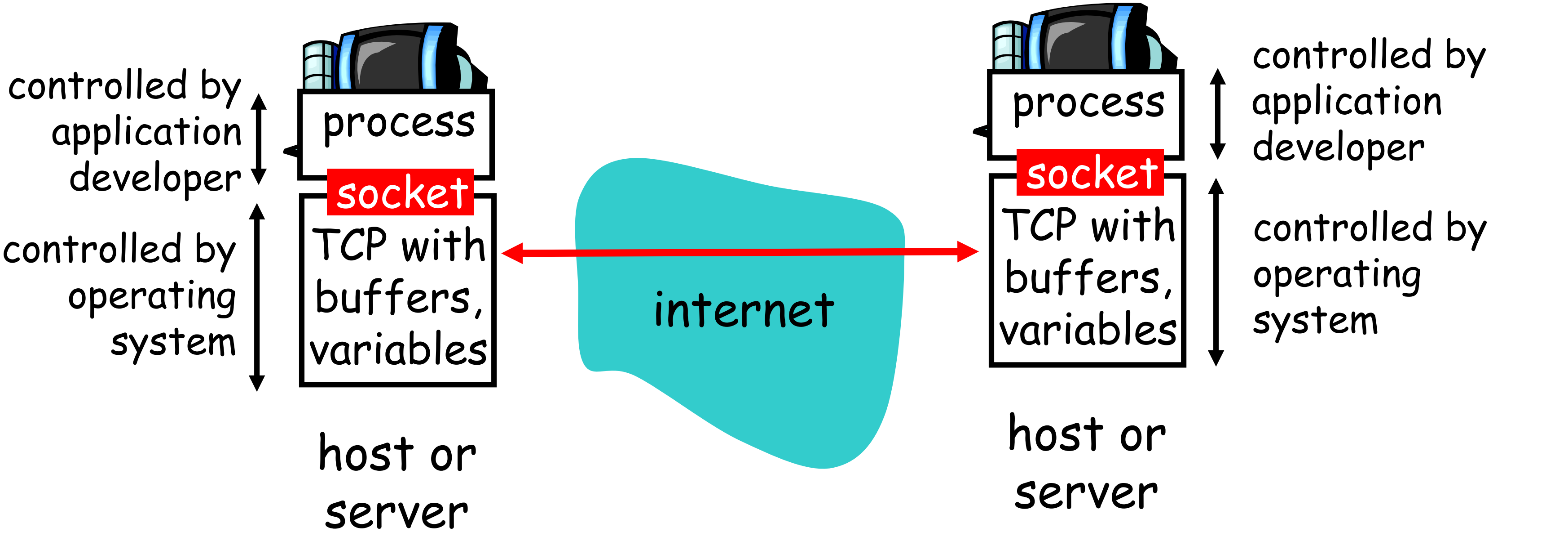
인터페이스

Socket: a door between application process and end-end-transport protocol (UDP or TCP)

socket

a *host-local*, *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

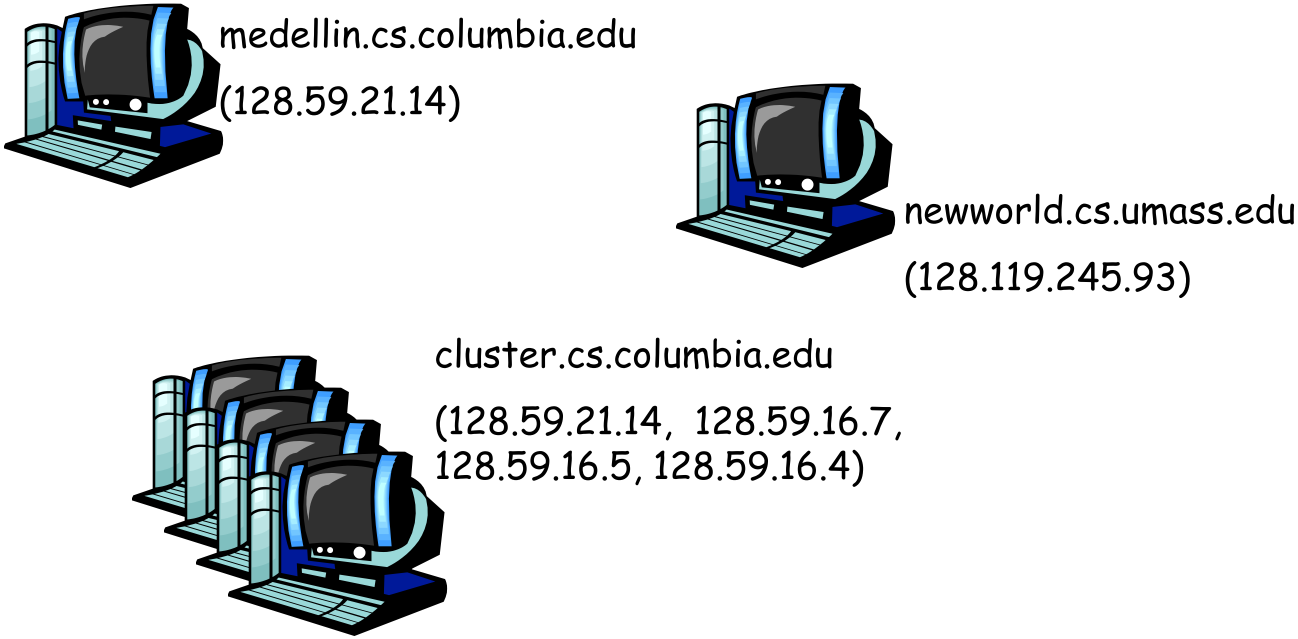
application에 있는 process


controlled by application developer
controlled by operating system
process
socket
TCP with buffers, variables
host or server
internet
process
socket
TCP with buffers, variables
controlled by application developer
controlled by operating system
host or server

What is a socket? (cont.)

- An **interface** between application and network
  - The application creates a socket
  - Once configured, the application can
    - pass data to the socket for network transmission
    - receive data from the socket (transmitted through the network by some other host)

A Socket-eye view of the Internet



medellin.cs.columbia.edu

(128.59.21.14)

newworld.cs.umass.edu

(128.119.245.93)

cluster.cs.columbia.edu

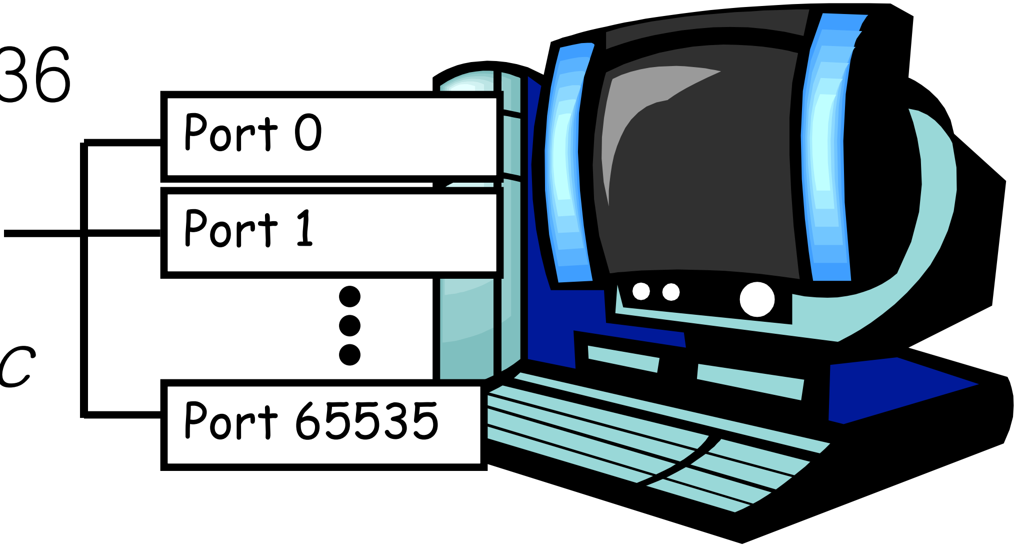
(128.59.21.14, 128.59.16.7,
128.59.16.5, 128.59.16.4)

+ port 번호 필요

- Each host machine has an IP address
- When a packet arrives at a host, how to associate with a process

Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



❒ A socket provides an interface to send data to/from the network through a port

지정 안된건 application 개발자들이 맘대로 쓸 수 있다

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

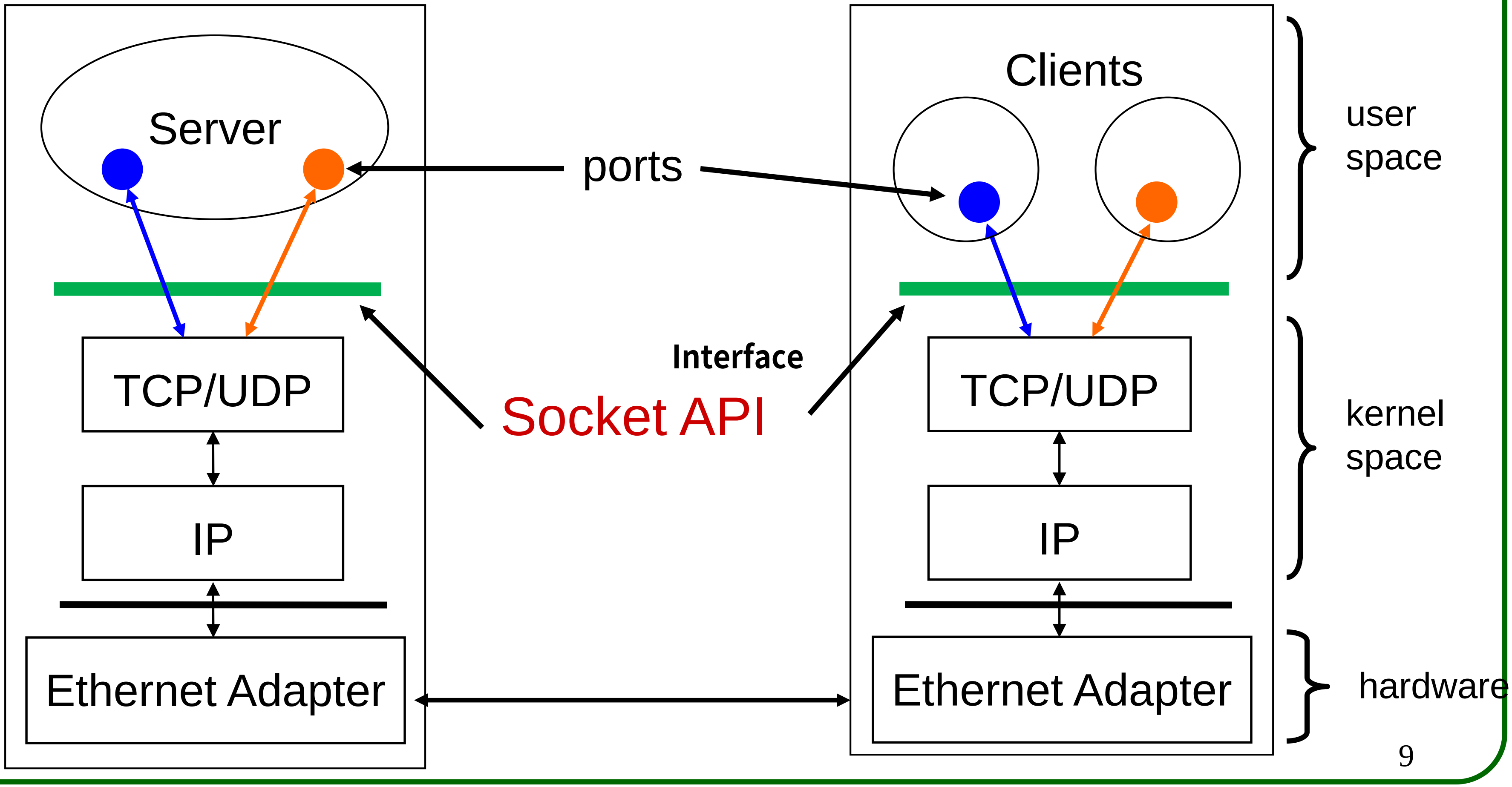
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- Several APIs
  - JAVA socket API
  - Windows Socket API (WINSOCK)
- two types of transport service via socket API:
  - unreliable datagram **UDP가 사용**
  - reliable, byte stream-oriented **TCP가 사용**

Example API functions

- JAVA
  - Socket class, ServerSocket class
  - socket()
  - accept()
  - getInputStream()
  - getOutputStream()
  - getLocalHost()
  - close()
  - …
- C
  - socket()
  - bind()
  - accept()
  - send()
  - recv()
  - select()
  - close()
  - …

Server and Client

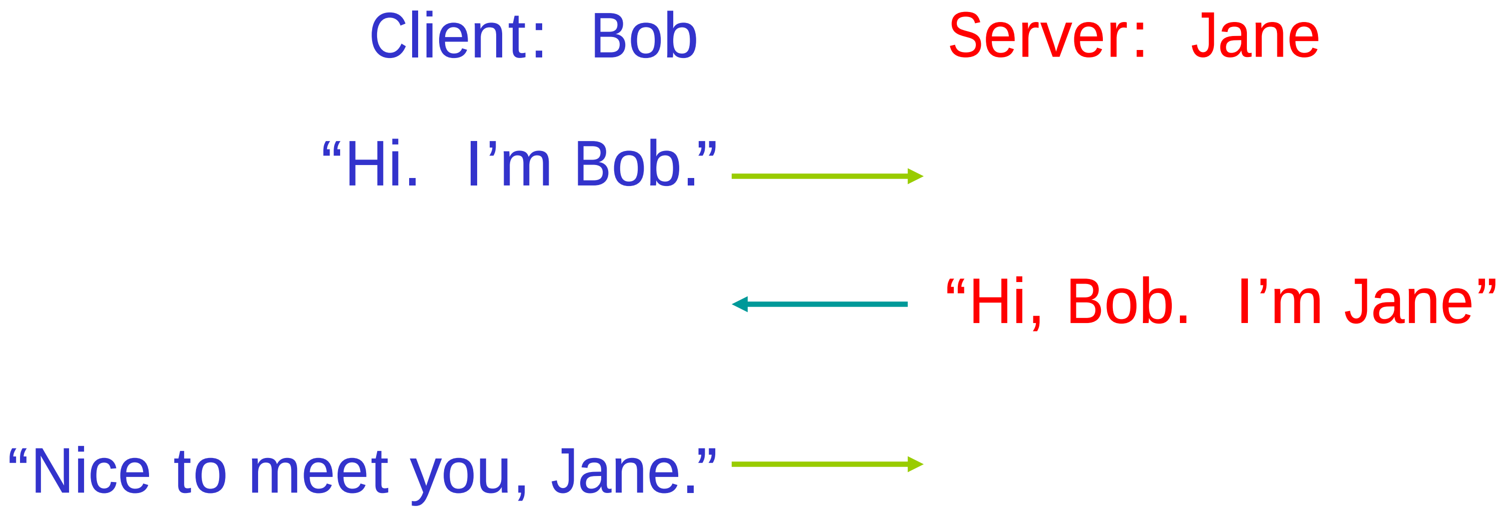
- Server and Client exchange messages over the network through a common Socket API



Server and Client

- Client: Initiates the connection

client가 항상 먼저 contact



Client: Bob

Server: Jane

"Hi. I'm Bob."

"Hi, Bob. I'm Jane"

"Nice to meet you, Jane."

- Server: Passively waits to respond

TCP Client/Server Interaction

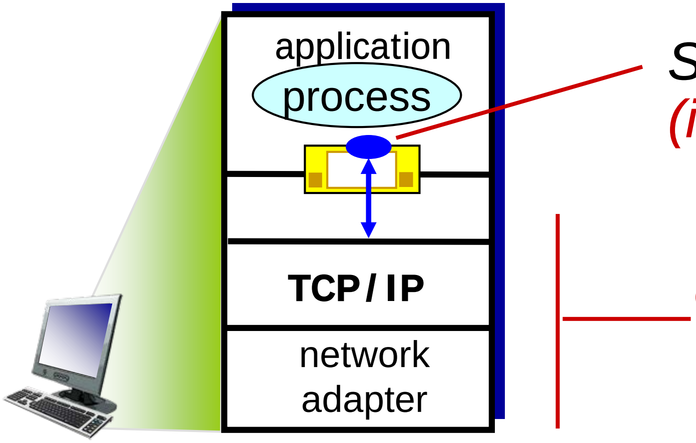
Server starts by getting ready to receive client connections…

Server

1. Create a TCP socket
2. Repeatedly:
   a. Listen (wait) & Accept new connection
   b. Communicate
   c. Close the connection

Client

1. Create a TCP socket & Connect server
2. Communicate
3. Close the connection



Server socket
(identified by port#)

controlled
by OS

Create
welcoming socket
at port, e.g. 6789

포트 번호를 지정해야됨

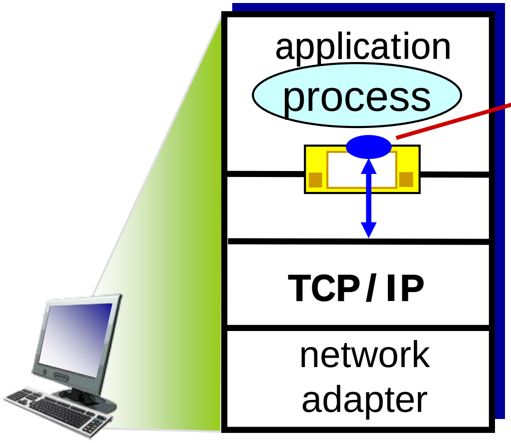
```
ServerSocket servSock = new ServerSocket(servPort);
```

Server

1. **Create a TCP socket**
2. Repeatedly:
   a. Listen (wait) & Accept new connection
   b. Communicate
   c. Close the connection

Client

1. Create a TCP socket & Connect server
2. Communicate
3. Close the connection



Server socket
(identified by port#)

listening 하는 welcoming socket 이 request를 반복문을 돌며 지속적으로 듣다가 request message가 오는 순간 accept method가 호출되면서 client socket을 return 한다.

Wait, on welcoming socket for contact by client

```
for (;;) {
    Socket clntSock =   servSock.accept();
```

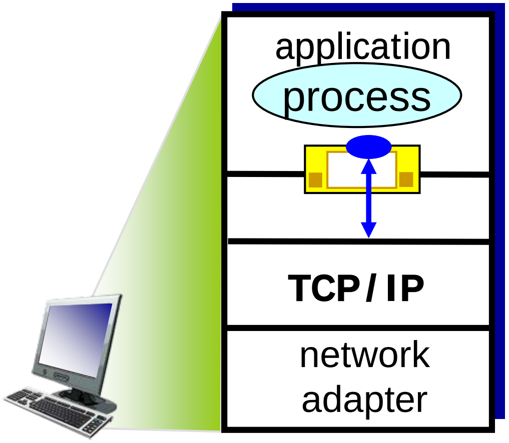
계속 받다가 accept method가 오게 되면 실제로 주고받는 소켓을 return 하게 됨

Server

1. Create a TCP socket
2. Repeatedly:
   a. **Listen (wait) & Accept new connection**
   b. Communicate
   c. Close the connection

Client

1. Create a TCP socket & Connect server
2. Communicate
3. Close the connection



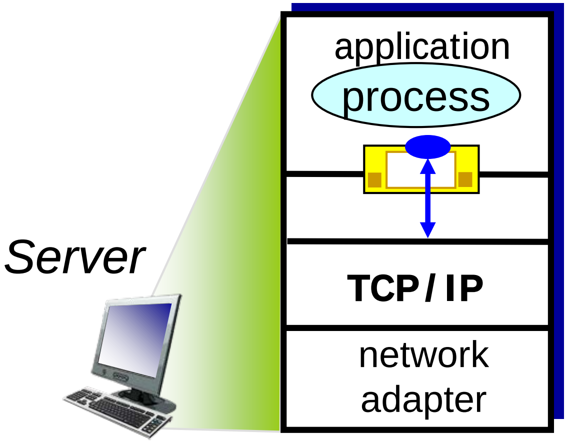
Server is now blocked waiting for connection from a client

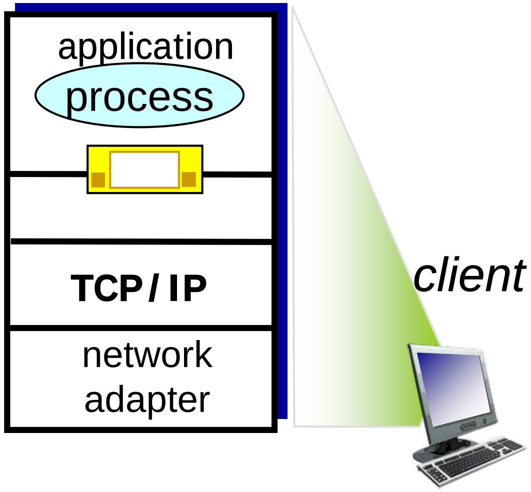
Server

1. Create a TCP socket
2. Repeatedly:
   a. Listen (wait) & Accept new connection
   b. Communicate
   c. Close the connection

Client

1. Create a TCP socket & Connect server
2. Communicate
3. Close the connection





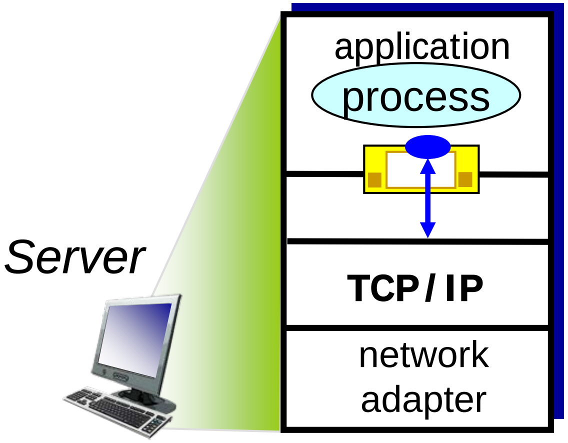
Later, a client decides to talk to the server…

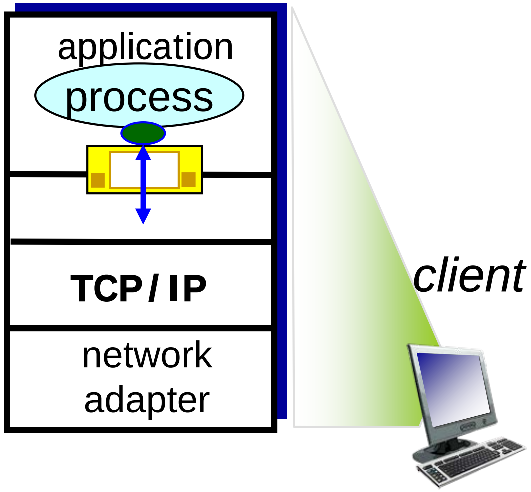
Server

1. Create a TCP socket
2. Repeatedly:
   a. Listen (wait) & Accept new connection
   b. Communicate
   c. Close the connection

Client

1. Create a TCP socket & Connect server
2. Communicate
3. Close the connection





Create
client socket,
connect to server

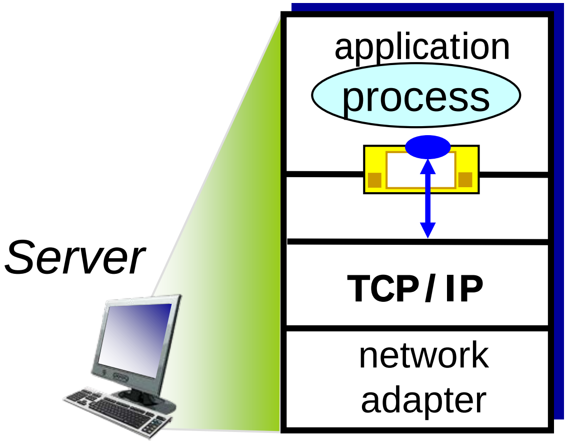
```
Socket socket = new Socket(serverIP, servPort);
```

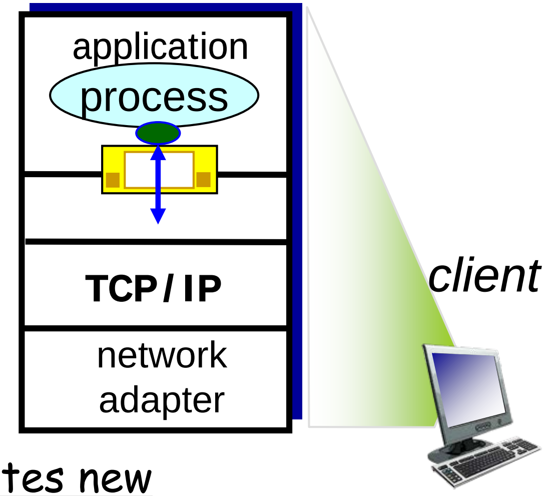
Server

1. Create a TCP socket
2. Repeatedly:
   a. Listen (wait) & Accept new connection
   b. Communicate
   c. Close the connection

Client

1. Create a TCP socket & Connect server
2. Communicate
3. Close the connection





```
Socket clntSock =   servSock.accept();
```

← Accept! &
server TCP creates new socket for the client

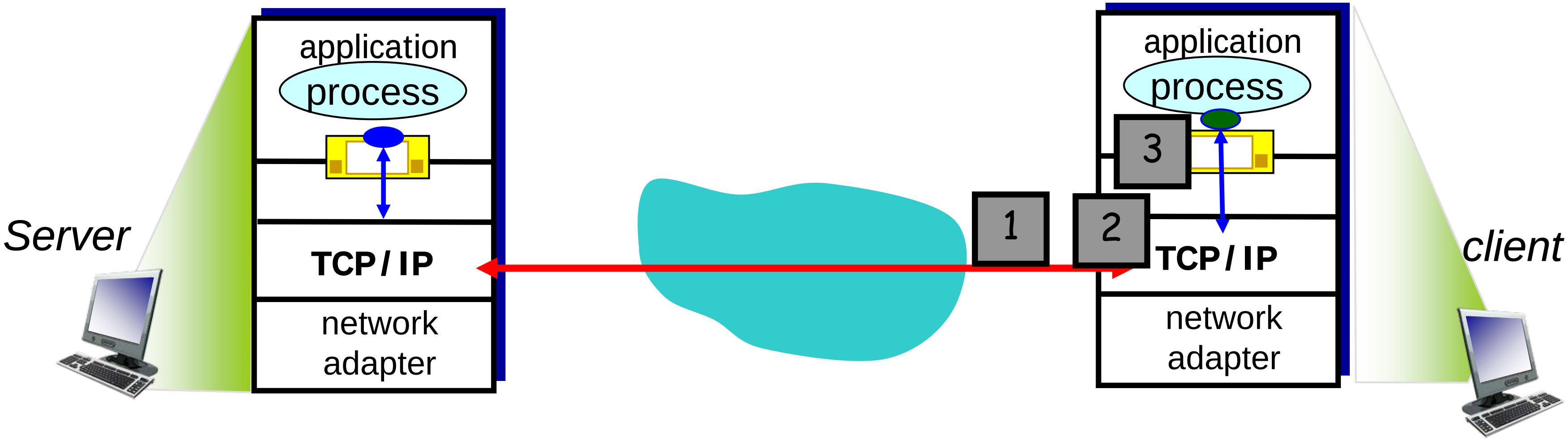
```
InputStream in = clntSock.getInputStream();
recvMsgSize = in.read(byteBuffer);
```

Server

1. Create a TCP socket
2. Repeatedly:
   a. Listen (wait) & Accept new connection
   b. **Communicate**
   c. Close the connection

Client

1. Create a TCP socket & Connect server
2. Communicate
3. Close the connection



InputStream in = clntSock.getInputStream();
recvMsgSize = in.read(byteBuffer);

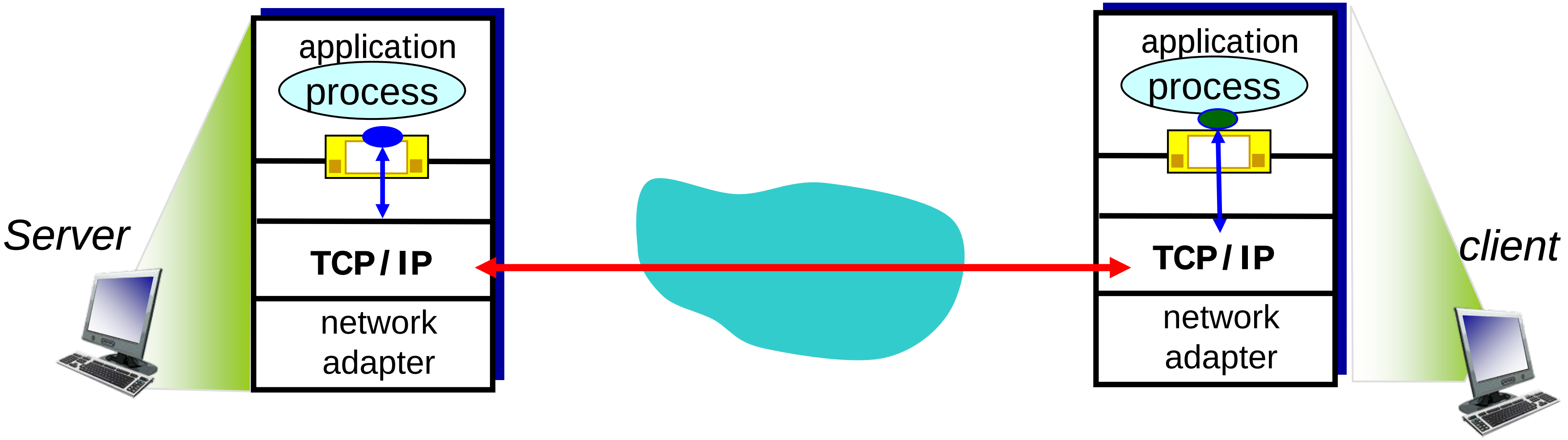
OutputStream out = socket.getOutputStream();
out.write(byteBuffer);

Server

1. Create a TCP socket
2. Repeatedly:
   a. Listen (wait) & Accept new connection
   b. **Communicate**
   c. Close the connection

Client

1. Create a TCP socket & Connect server
2. Communicate
3. Close the connection



close(clntSocket)

close(sock);

Server

1. Create a TCP socket
2. Repeatedly:
   a. Listen (wait) & Accept new connection
   b. Communicate
   c. **Close the connection**

Client

1. Create a TCP socket & Connect server
2. Communicate
3. **Close the connection**

send – recv quickview

- No correlation between `send()` and `recv()`

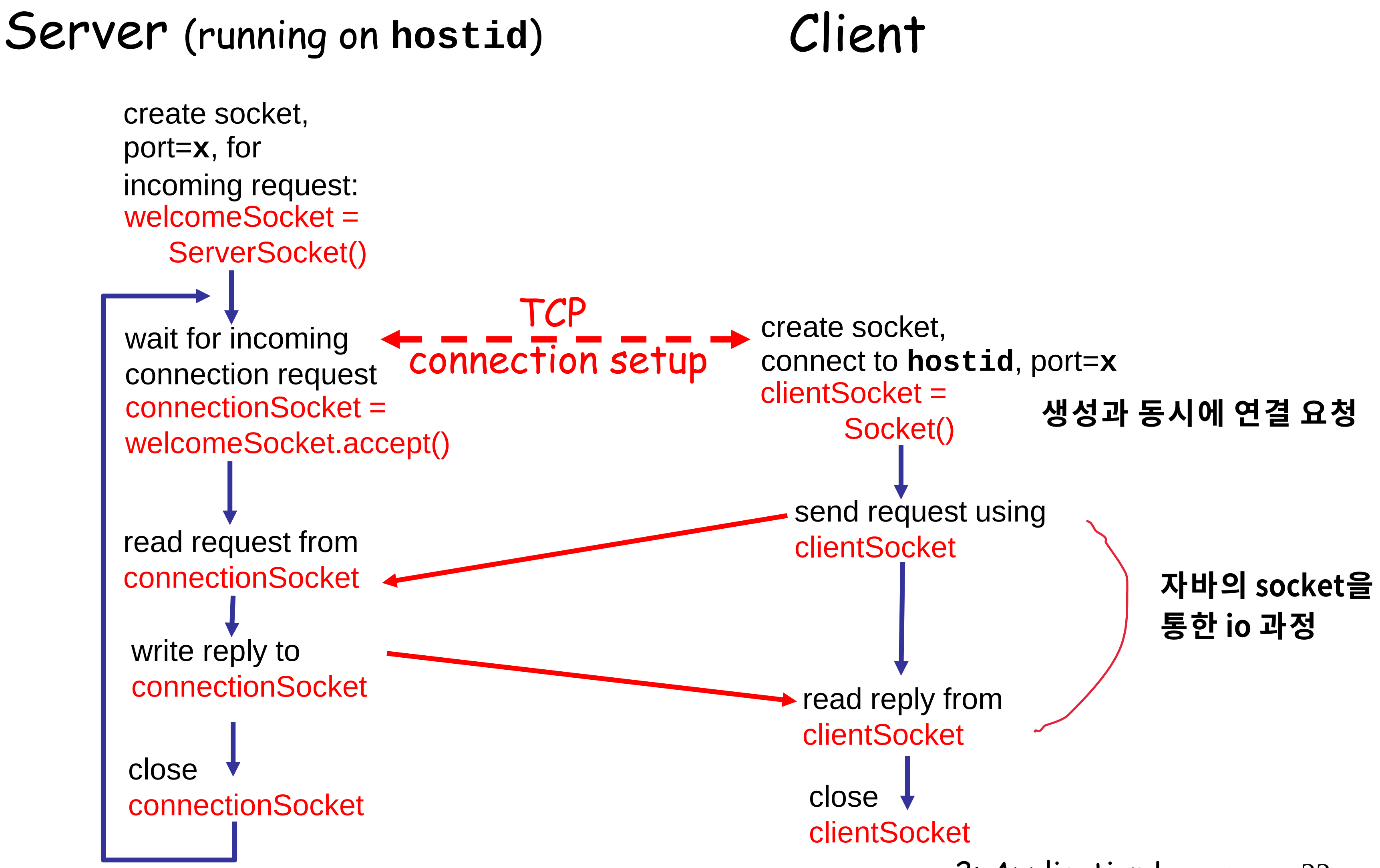
Client	Server
out.write(“Hello Bob”)	
	in.read() -> “Hello ”
	in.read() -> “Bob”
	out.write(“Hi ”)
	out.write(“Jane”)
in.read() -> “Hi Jane”	

Client/server socket interaction: Summary

- **Server**: program/computer **providing** a service
  - Creates a local socket
  - Binds local socket to a specific port (in Java, this step is included in the creation step) **그래서 port 번호가 필요**
  - Listens for incoming connections
  - Accepts a connection, assigning a new socket for the connection (in Java, listen & accept are done together)
  - Sends/receives data

- **Client**: program/computer **requesting** a service
  - Client knows server address and port
  - Creates a local socket
  - Connects to remote socket
  - Sends/receives data

Client/server socket interaction: Summary



Server (running on **hostid**)

Client

create socket, port=**x**, for incoming request:
welcomeSocket = ServerSocket()

wait for incoming connection request
connectionSocket = welcomeSocket.accept()

TCP connection setup

create socket, connect to **hostid**, port=**x**
clientSocket = Socket()

생성과 동시에 연결 요청

send request using clientSocket

read request from connectionSocket

write reply to connectionSocket

read reply from clientSocket

자바의 socket을 통한 io 과정

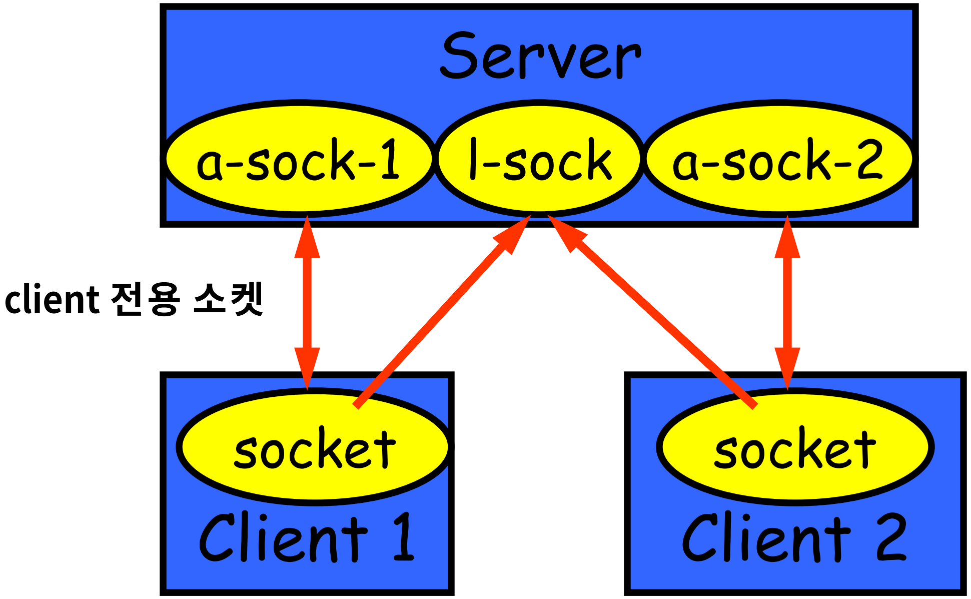
close connectionSocket

close clientSocket

Connection setup @ server side

connectionSocket = welcomeSocket.accept()

- When contacted by a client, server TCP creates a new socket for server process to communicate with the client
- The accepted connection is on a new socket
- The listen-socket continues to listen for other active participants
- Why?
  - allows server to talk with multiple clients !!



Java Socket Programming

Java Sockets Programming

- The package java.net provides support for sockets programming (and more).
- Typically you import everything defined in this package with:

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

Socket class

- Corresponds to active TCP sockets only!
  - client sockets
  - socket returned by accept();
- Server-side listen (passive) sockets are supported by a different class:
  - ServerSocket

TCP 기준

- UDP sockets are supported by
  - DatagramSocket

UDP 기준

JAVA TCP Sockets

- java.net.Socket
  - Implements client sockets (also called just “sockets”).
  - An endpoint for communication between two machines.
  - Constructor and Methods
    - **Socket(String host, int port):** Creates a stream socket and connects it to the specified port number on the named host.
    - InputStream getInputStream()
    - OutputStream getOutputStream()
    - close()

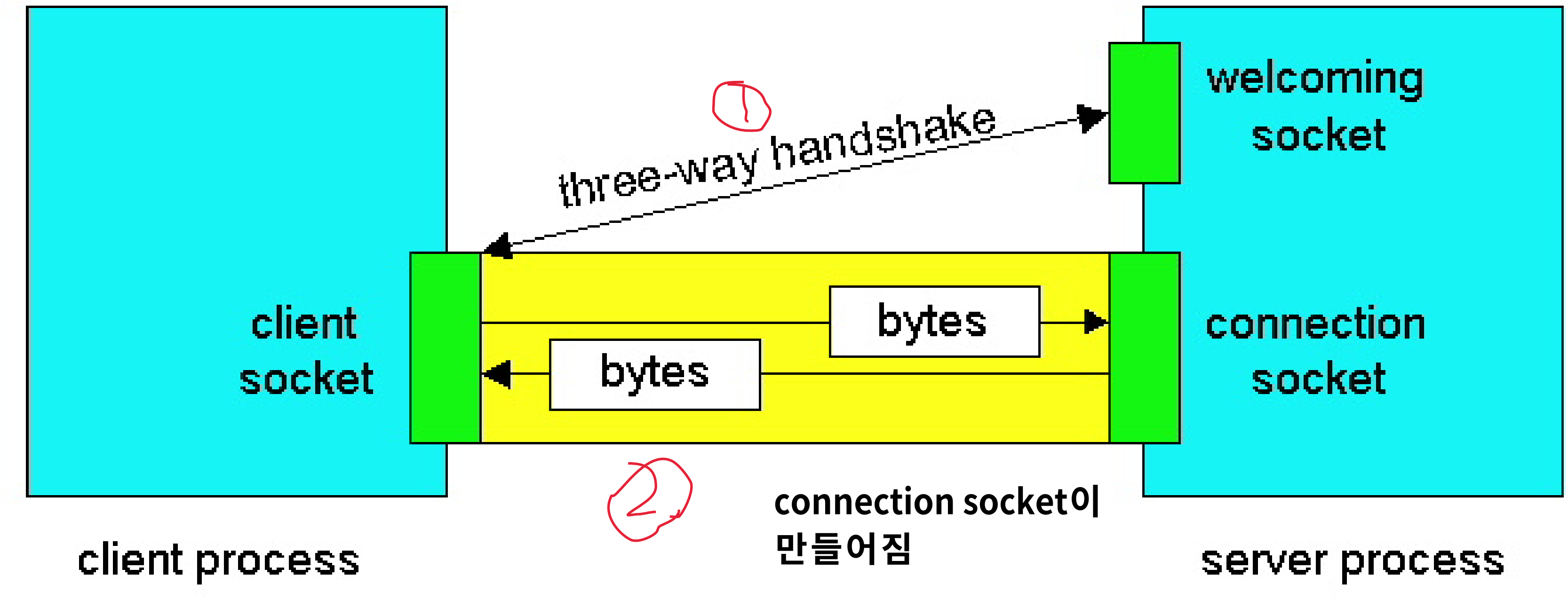
string 형태의 ip 주소(url도 가능), port 번호

- java.net.ServerSocket
  - Implements server sockets.
  - Waits for requests to come in over the network.
  - Performs some operation based on the request.
  - Constructor and Methods

지정된 pocket으로 연결

    - **ServerSocket(int port) :** Creates a server socket and binds it to the specified local port number
    - Socket **Accept():** Listens for a connection to be made to this socket and accepts it. This method blocks until a connection is made.

Sockets



Client socket, welcoming socket (passive) and connection socket (active)

Socket Constructors

- Constructor creates a TCP connection to a named TCP server.
  - There are a number of constructors:

```
Socket(InetAddress server, int port);
```

ip 주소

```
Socket(String hostname, int port);

Socket(InetAddress server, int port,
          InetAddress local, int localport);
```

Socket Methods

```
void close();
InputStream getInputStream();
OutputStream getOutputStream();
```

- Lots more (setting/getting socket options, partial close, etc.)

Socket I/O

- Socket I/O is based on the Java I/O support
  - in the package **`java.io`**
    - import java.io.*;
- InputStream and OutputStream are abstract classes
  - common operations defined for all kinds of InputStreams, OutputStreams…
- example

```
DataOutputStream outToServer = new
DataOutputStream(clientSocket.getOutputStream());
```

InputStream Basics

```
// reads some number of bytes and
// puts in buffer array b
int read(byte[] b);
// reads up to len bytes
int read(byte[] b, int off, int len);
```

Both methods can throw **`IOException`**.

Both return −1 on EOF.

OutputStream Basics

```
// writes b.length bytes
void write(byte[] b);
// writes len bytes starting
// at offset off
void write(byte[] b, int off, int len);
```

Both methods can throw **`IOException`**.

InetAddress class

ip 주소를 얻는 방법

- static methods you can use to create new InetAddress objects.
  - static InetAddress getByName(String host)
  - static InetAddress[] getAllByName(String host)
    - e.g. daum.net, naver.com, …
  - static InetAddress getLocalHost()

```
InetAddress x  = InetAddress.getByName(
                            “sw.gachon.ac.kr”);
InetAddress local = InetAddress.getLocalHost();
```

❖ Throws **UnknownHostException**

InetAddress class (Example)

```
try {

  InetAddress ad = InetAddress.getByName(hostname);
  System.out.println(hostname + ":" +
                ad.getHostAddress());

} catch (UnknownHostException e) {

  System.out.println("No address found for " +
                       hostname);

}
```

TCPClient.java

USE: 127.0.0.1

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

class TCPClient {
    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;
```

Create client socket, connect to server

```
        Socket clientSocket = new Socket("hostname", 6789);
```

Create input stream

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

Create output stream attached to socket

```
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```

client socket을 통해서 outputstream 객체를 얻어 DataOutput Stream 객체를 생성

TCPClient.java

server으로부터 온 데이터를 읽기 위한 객체

Create input stream attached to socket

```
  BufferedReader inFromServer =
          new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

  sentence = inFromUser.readLine();
```

Send line to server

```
  outToServer.writeBytes(sentence + '\ n');
```

Read line from server

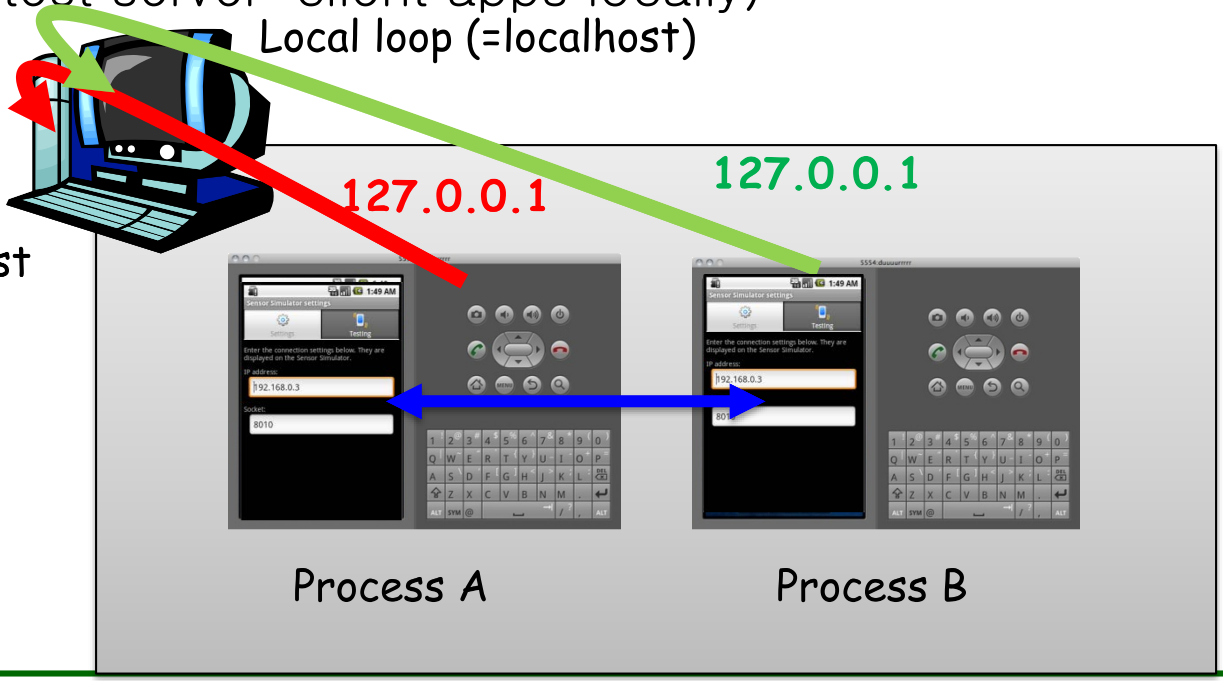
```
  modifiedSentence = inFromServer.readLine();

  System.out.println("FROM SERVER: " + modifiedSentence);

  clientSocket.close();

  }
}
```

NOTE: 127.0.0.1

IP address: 127.0.0.1 ?

a special purpose address reserved for use on each computer. → localhost or local loop

Used to access a local computer's TCP/IP network resources
(or to test server-client apps locally)



Sensor Simulator settings

Settings | Testing

Enter the connection settings below. They are displayed on the Sensor Simulator.

IP address:
192.168.0.3

Socket:
8010

TCPServer.java

socket을 통해 io 객체를 얻어낸다

```
import java.io.*;
import java.net.*;
class TCPServer {
  public static void main(String argv[]) throws Exception
     {
        String clientSentence;
        String capitalizedSentence;
```

Create welcoming socket at port 6789

```
        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {
```

Wait, on welcoming socket for contact by client

```
           Socket connectionSocket = welcomeSocket.accept();
```

Create input stream, attached to socket

```
           BufferedReader inFromClient = new BufferedReader(new
               InputStreamReader(connectionSocket.getInputStream()));
```

connectionSocket을 통해 input stream을 읽음

TCPServer.java

Create output stream, attached to socket

```
DataOutputStream  outToClient =
   new DataOutputStream( connectionSocket.getOutputStream());
```

Read in line from socket

```
    clientSentence = inFromClient.readLine();
```

```
    capitalizedSentence = clientSentence.toUpperCase() + '\  n';
```

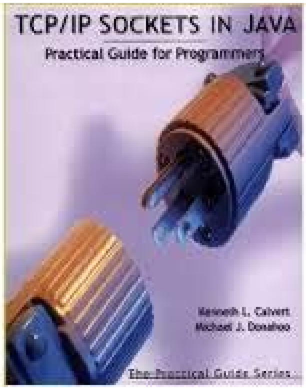
Write out line to socket

```
    outToClient.writeBytes(capitalizedSentence);
```

```
  }
 }
}
```

End of while loop, loop back and wait for another client connection for Socket Programming

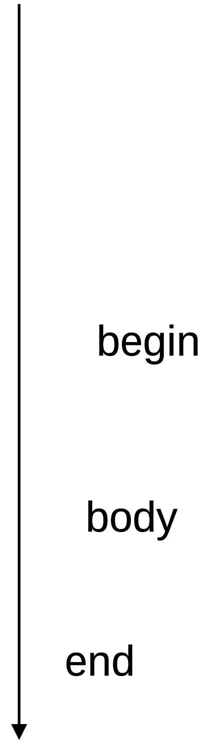
Java Thread Basics



Most slides are from *Prof. Rajkumar Buyya*
Cloud Computing and Distributed Systems (CLOUDS) Laboratory
Dept. of Computer Science and Software Engineering
University of Melbourne, Australia http://www.cloudbus.org/~raj or http://www.buyya.com

A single threaded program

```
class ABC
{
....
     public void main(..)
     {
     ...

     ..
     }
}
```

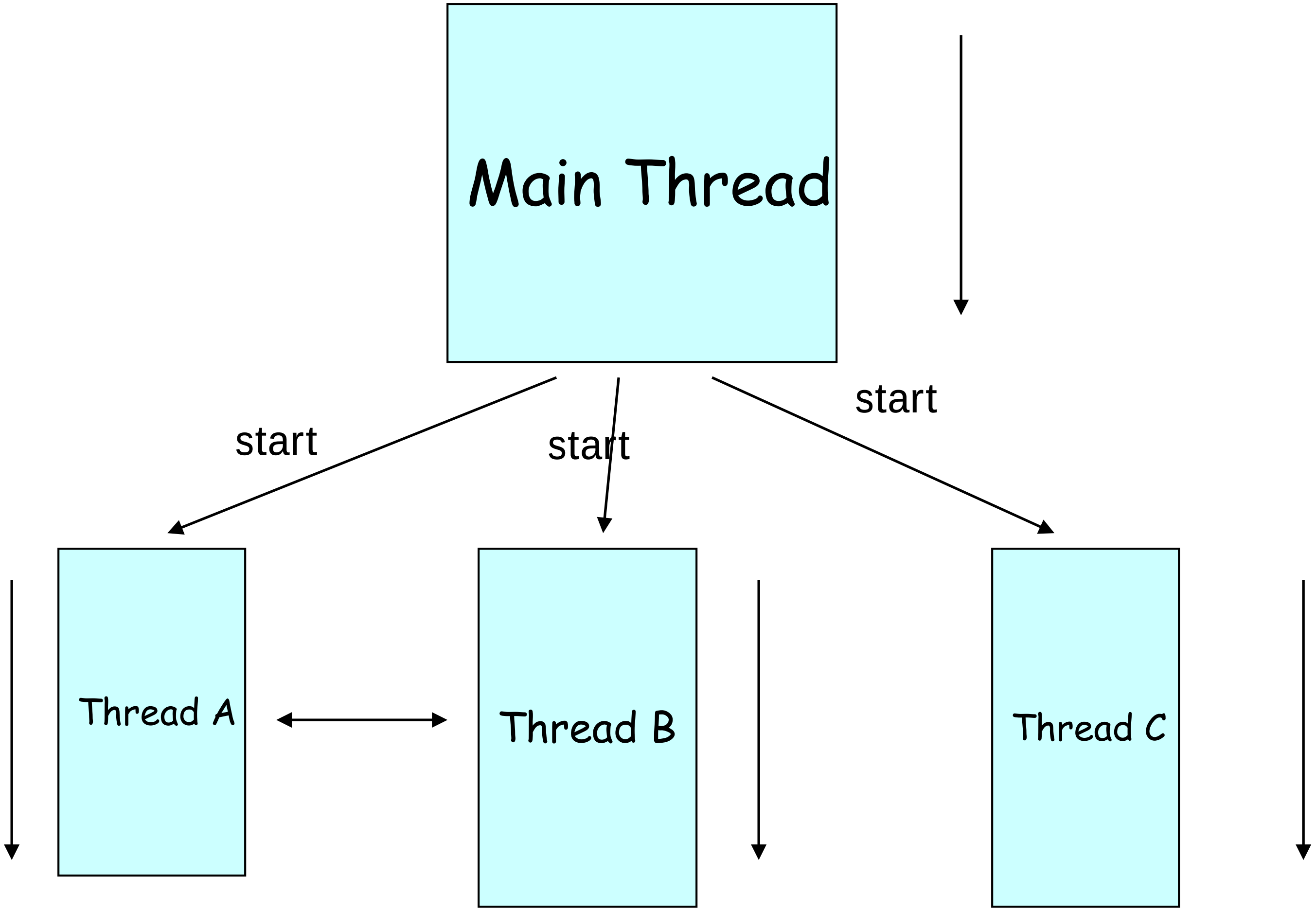


begin

body

end

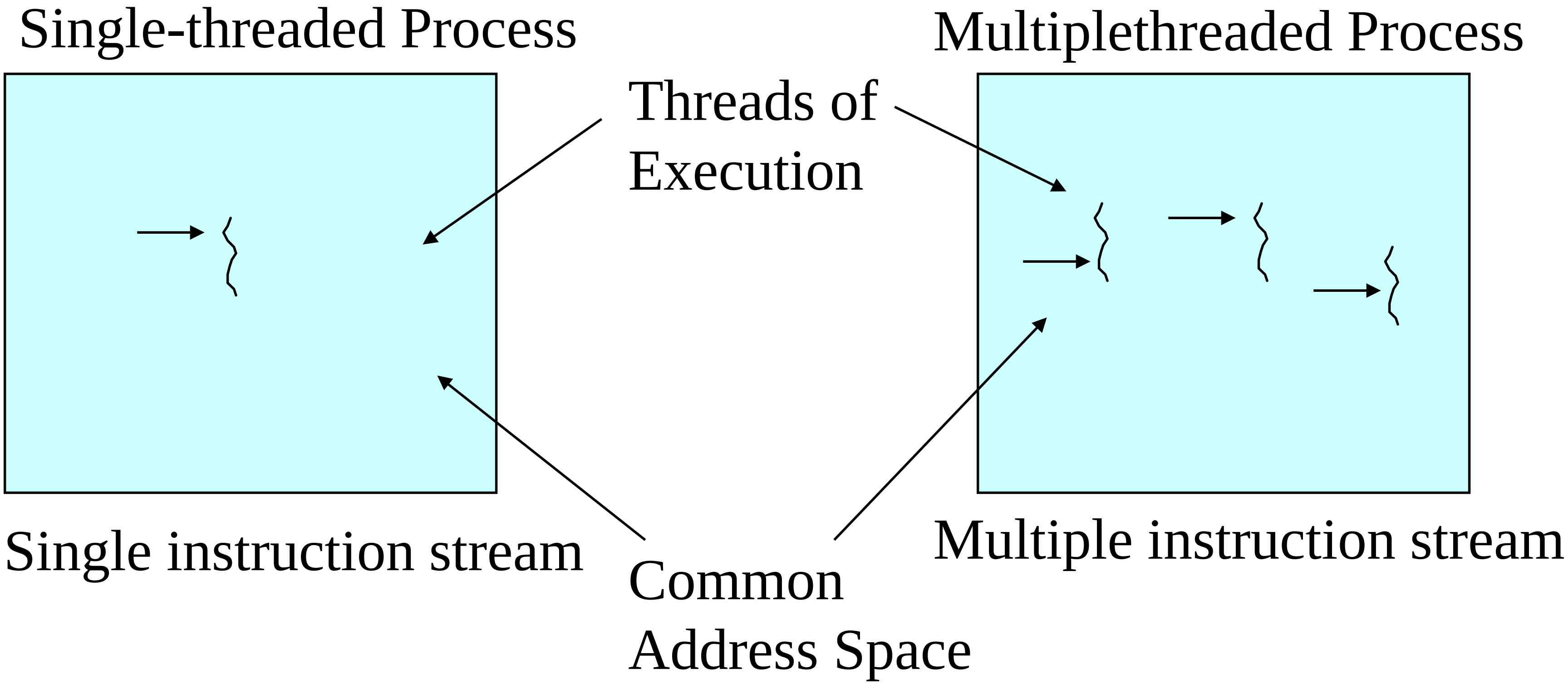
A Multithreaded Program



Threads may switch or exchange data/results

Single and Multithreaded Processes

threads are light-weight processes within a process



Single-threaded Process

Multiplethreaded Process

Threads of Execution

Single instruction stream

Multiple instruction stream

Common Address Space

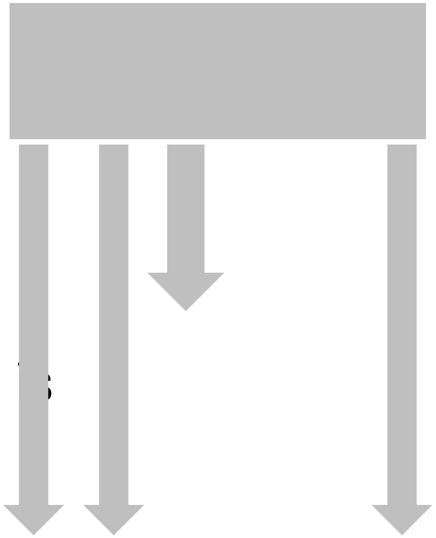
What are Threads?

1. A Thread is
   - a flow of control in a process
   - A piece of code that run in concurrent with other threads.

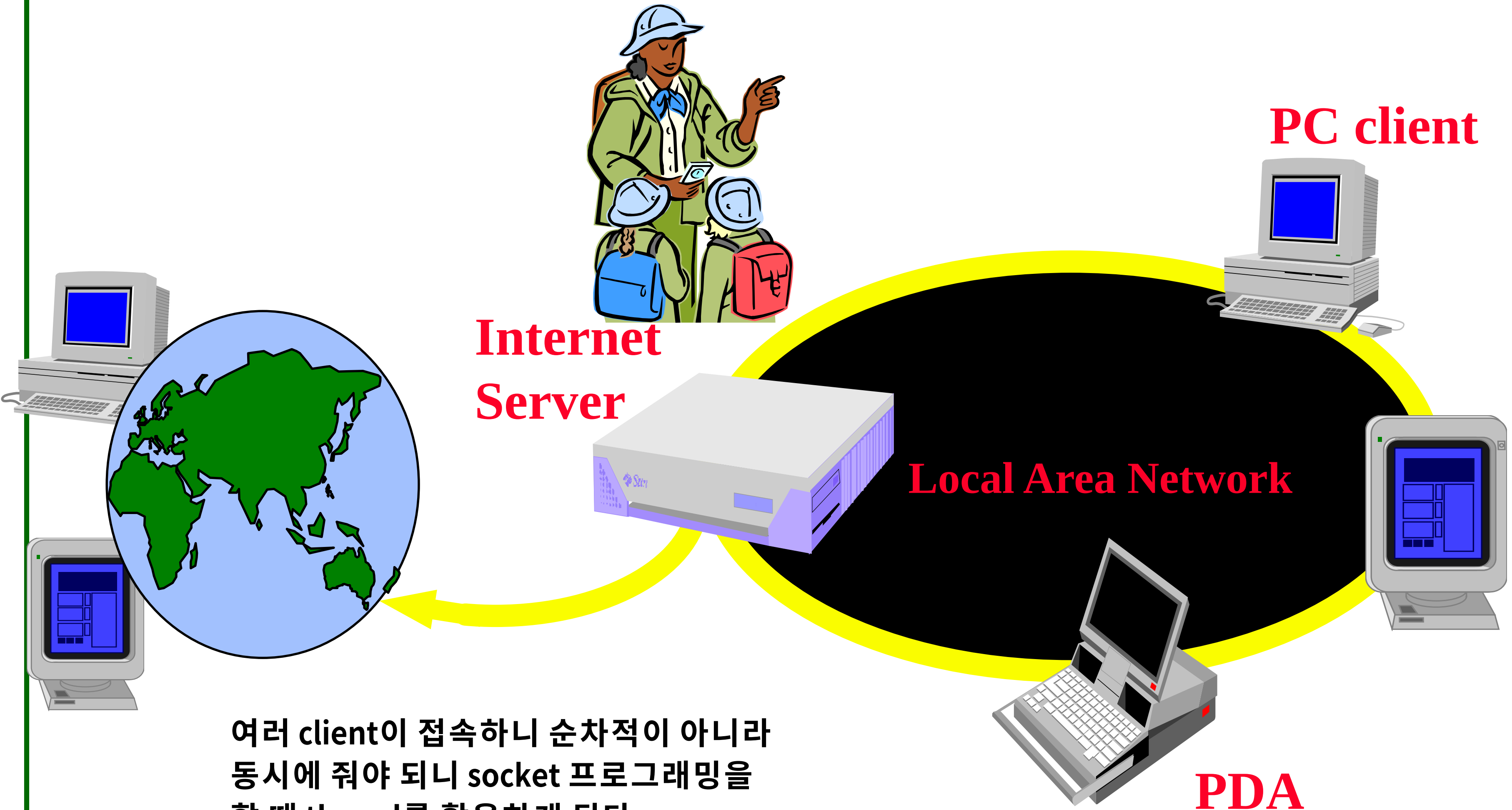
2. Each thread has its own call **stack**. The call stack is used on method calling, parameter passing, and storage for the called method's local variables.

3. Each virtual machine instance has at least one main thread .

- The application might decide to launch additional Threads for specific purposes.



Web/Internet Applications: Serving Many Users Simultaneously



여러 client이 접속하니 순차적이 아니라
동시에 줘야 되니 socket 프로그래밍을
할 때 thread를 활용하게 된다

We have observed..

TCPServer.java

```
import java.io.*;
import java.net.*;
class TCPServer {
  public static void main(String argv[]) throws Exception
    {
      String clientSentence;
      String capitalizedSentence;

      ServerSocket welcomeSocket = new ServerSocket(6789);

      while(true) {

       Socket connectionSocket = welcomeSocket.accept();

       BufferedReader inFromClient = new BufferedReader(new
             InputStreamReader(connectionSocket.getInputStream()));

   DataOutputStream  outToClient =
      new DataOutputStream( connectionSocket.getOutputStream());

      clientSentence = inFromClient.readLine();

      capitalizedSentence = clientSentence.toUpperCase() + '₩n';

      outToClient.writeBytes(capitalizedSentence);

    }
  }
}
```

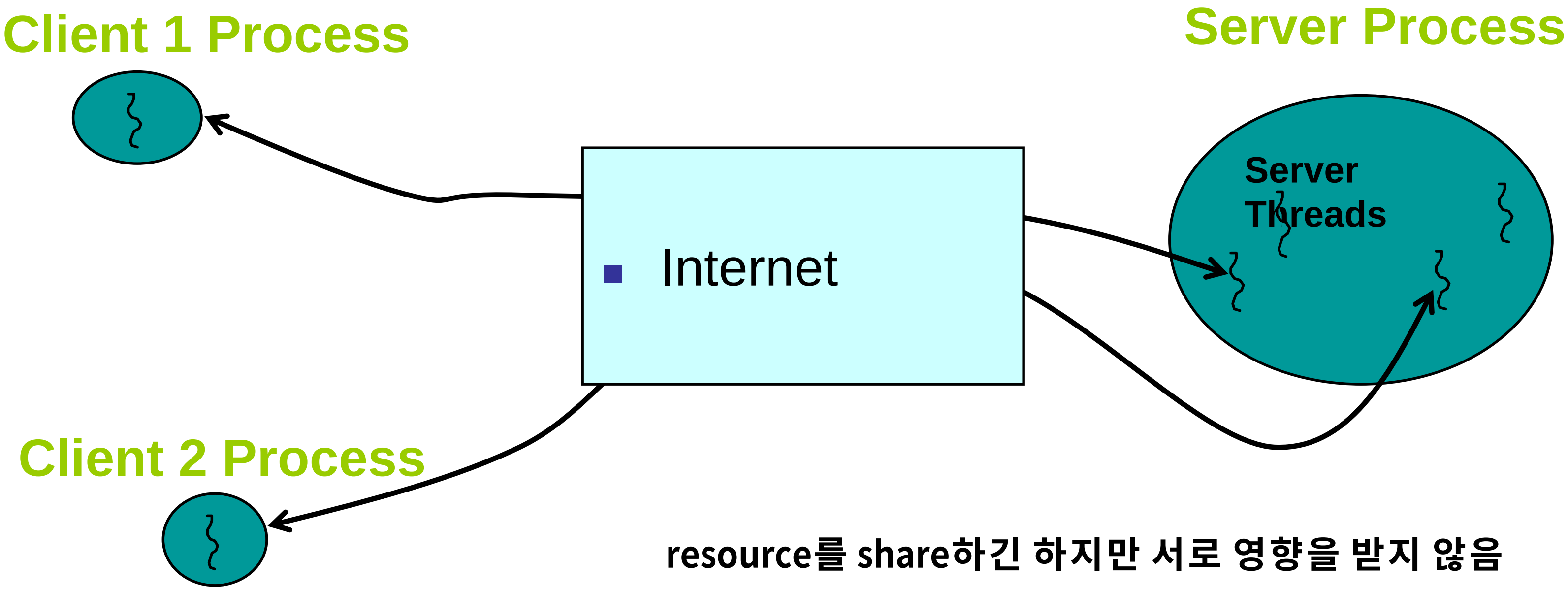
기다린다

The server is **blocked** until the client will send any message

So, all the other clients can't get any responses from the server

순차적으로 실행되기 때문에 다른 클라이언트들은 처리가 되지 않는다

server should be able to serve Multiple Clients Concurrently: Multithreaded Server

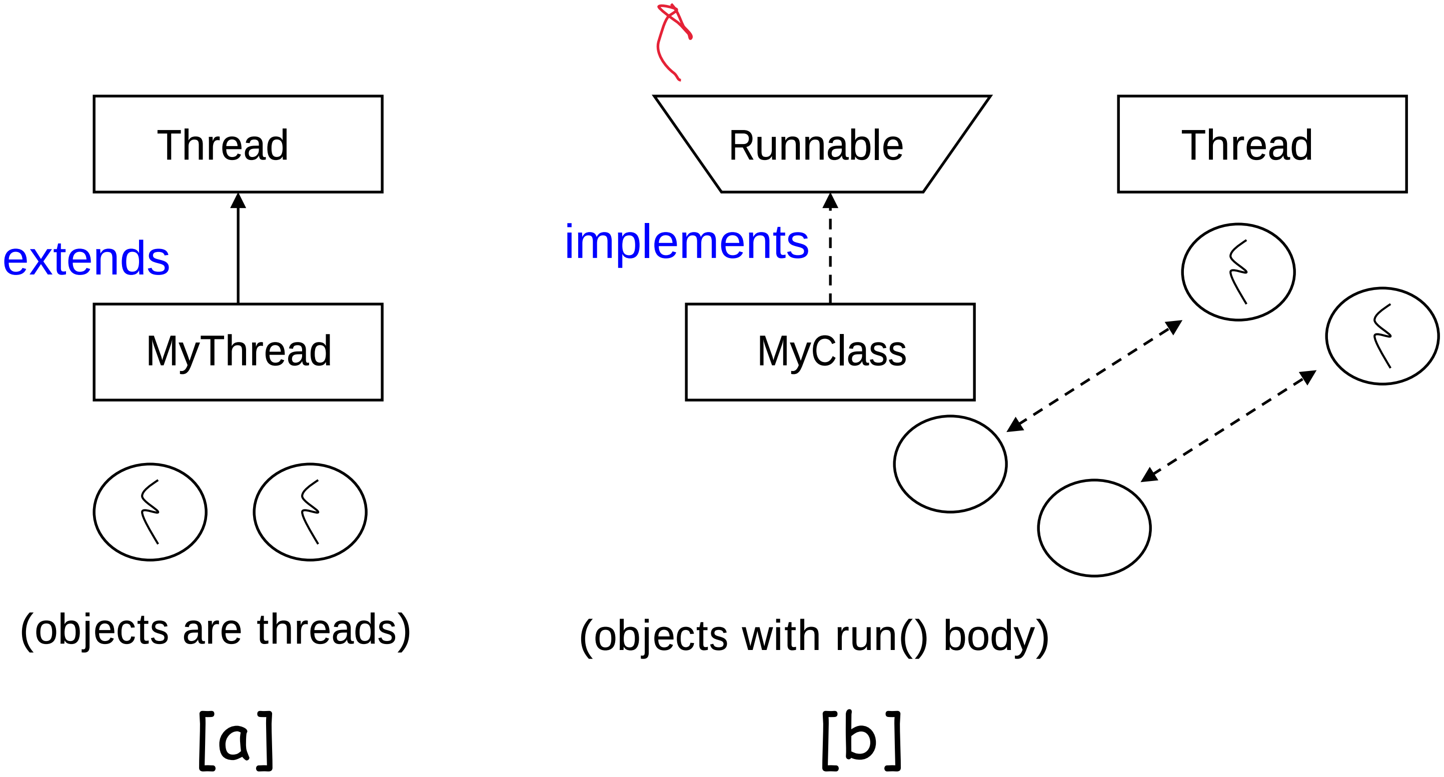


resource를 share하긴 하지만 서로 영향을 받지 않음

Two ways to use Thread

- [a] Create a class that extends the **Thread class**
- [b] Create a class that implements the **Runnable** interface

thread에서 수행해야될 작업을 정의하는 객체



1st method: Extending Thread class

- Create a class by extending Thread class and override run() method:

```
class MyThread extends Thread
{
     public void run()
    {
       // thread body of execution
    }
}
```

- **Create a thread:**

```
MyThread thr1 = new MyThread();
```

- **Start Execution of threads:**

```
thr1.start();
```

- or Create and Execute together:

```
new MyThread().start();
```

An example

```
class MyThread extends Thread {
        public void run() {
              System.out.println(" this thread is running ... ");
        }
}

class ThreadEx1 {
        public static void main(String [] args ) {
           MyThread t = new MyThread();
           t.start();
       }
}
```

Example 2

```
class MyThreadA extends Thread {
   public void run() { // entry point for thread
         for (;;) {
                 System.out.println("hello world1");
         }
   }
}

class MyThreadB extends Thread {
   public void run() { // entry point for thread
         for (;;) {
                 System.out.println("hello world2");
         }
   }
}

public class Main1 {
   public static void main(String [] args) {
         MyThreadA t1 = new MyThreadA();
         MyThreadB t2 = new MyThreadB();
         t1.start();
         t2.start();
         // main terminates, but in Java the other threads keep running
         // and hence Java program continues running
   }
}
```

2nd method: Threads by implementing Runnable interface

- Create a class that implements the interface Runnable and override run() method:

```
class MyThread implements Runnable
{
   .....
   public void run()
   {
      // thread body of execution
   }
}
```

Runnable 객체를 생성

  - Creating Object:

```
MyThread myObject = new MyThread();
```

  - Creating Thread Object:

```
Thread thr1 = new Thread( myObject );
```

  - Start Execution:

```
thr1.start();
```

An example

```
class MyThread implements Runnable  {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
}

class ThreadEx2 {
        public static void main(String [] args  ) {
                Thread t = new Thread(new MyThread());
                 t.start();
        }
}
```

Thread Priority

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running. The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
  - Java allows users to change priority:
    - ThreadName.setPriority(intNumber)
      - MIN_PRIORITY = 1
      - NORM_PRIORITY=5
      - MAX_PRIORITY=10

Thread Priority Example

```
class A extends Thread
{
        public void run()
          {
                System.out.println("Thread A started");
                for(int i=1;i<=4;i++)
                  {
                         System.out.println("₩t From ThreadA: i= "+i);
                  }
                  System.out.println("Exit from A");
          }
}
class B extends Thread
{
        public void run()
          {
                System.out.println("Thread B started");
                for(int j=1;j<=4;j++)
                  {
                         System.out.println("₩t From ThreadB: j= "+j);
                  }
                  System.out.println("Exit from B");
          }
}
```

Thread Priority Example

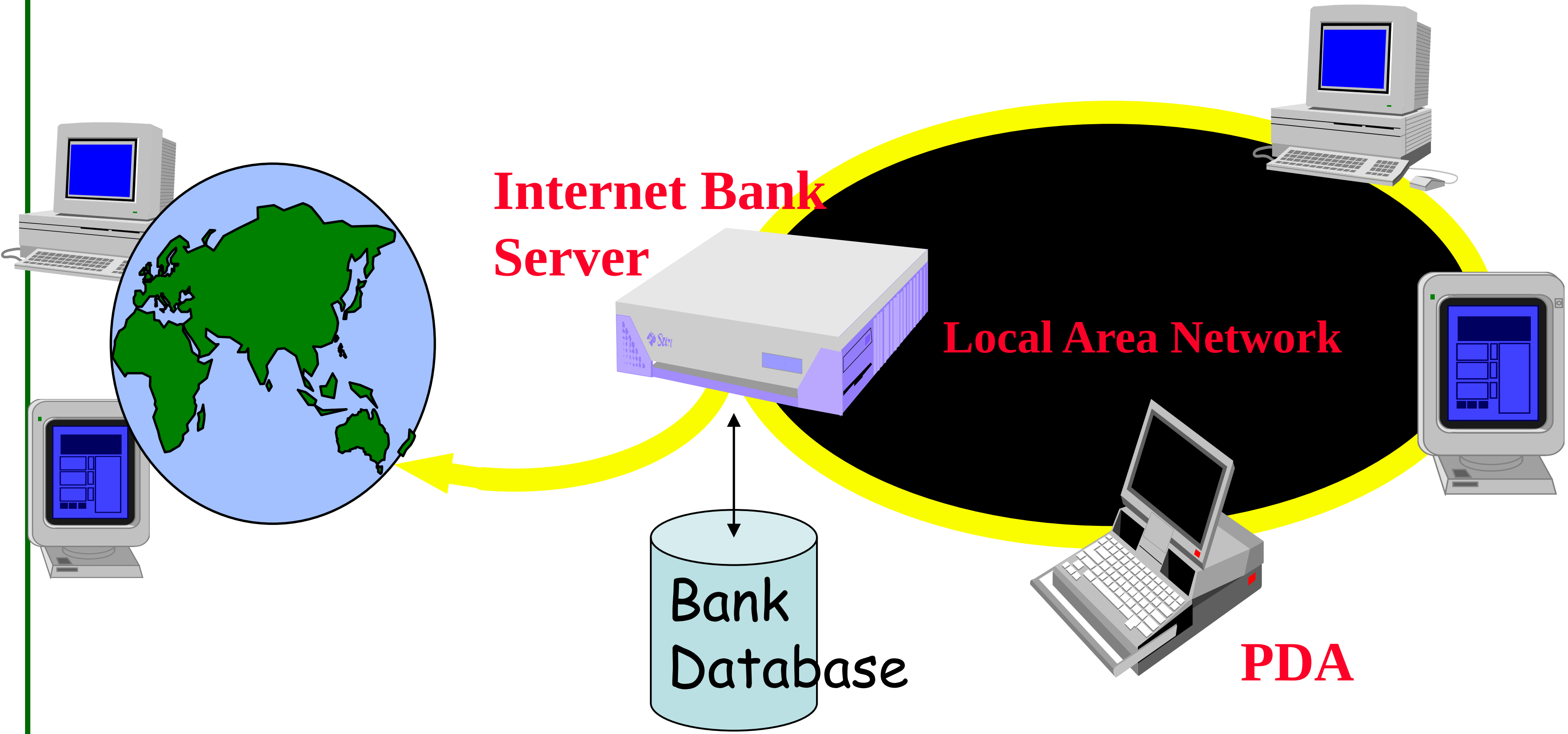
```
class C extends Thread
{
       public void run()
         {
              System.out.println("Thread C started");
              for(int k=1;k<=4;k++)
                {
                      System.out.println("₩t From ThreadC: k= "+k);
                }
                 System.out.println("Exit from C");
         }
}
class ThreadPriority
{
          public static void main(String args[])
           {
                      A threadA=new A();
                      B threadB=new B();
                      C threadC=new C();
                      threadC.setPriority(Thread.MAX_PRIORITY);
                      threadB.setPriority(threadA.getPriority()+1);
                      threadA.setPriority(Thread.MIN_PRIORITY);
                    System.out.println("Started Thread A");
                     threadA.start();
                    System.out.println("Started Thread B");
                     threadB.start();
                    System.out.println("Started Thread C");
                     threadC.start();
                     System.out.println("End of main thread");
           }
}
```

자원을 할당받는게 달라지니
for문 끝나는 시간이 달라진다

Accessing Shared Resources

- Applications Access to Shared Resources need to be coordinated.
  - Printer (two person jobs cannot be printed at the same time)
  - Simultaneous operations on your bank account.
  - Can the following operations be done at the same time on the same account?
    - Deposit()
    - Withdraw()
    - Enquire()

Online Bank: Serving Many Customers and Operations



PC client

Internet Bank Server

Local Area Network

Bank Database

PDA

Shared Resources

- If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- Use “Synchronized” method:
  - public synchronized void update()
  - {
    - …
  - }

작업을 못하게 막고 다 끝났을때 사용할
수 있도록 함 = synchronization

the driver: 3rd Threads sharing the same object

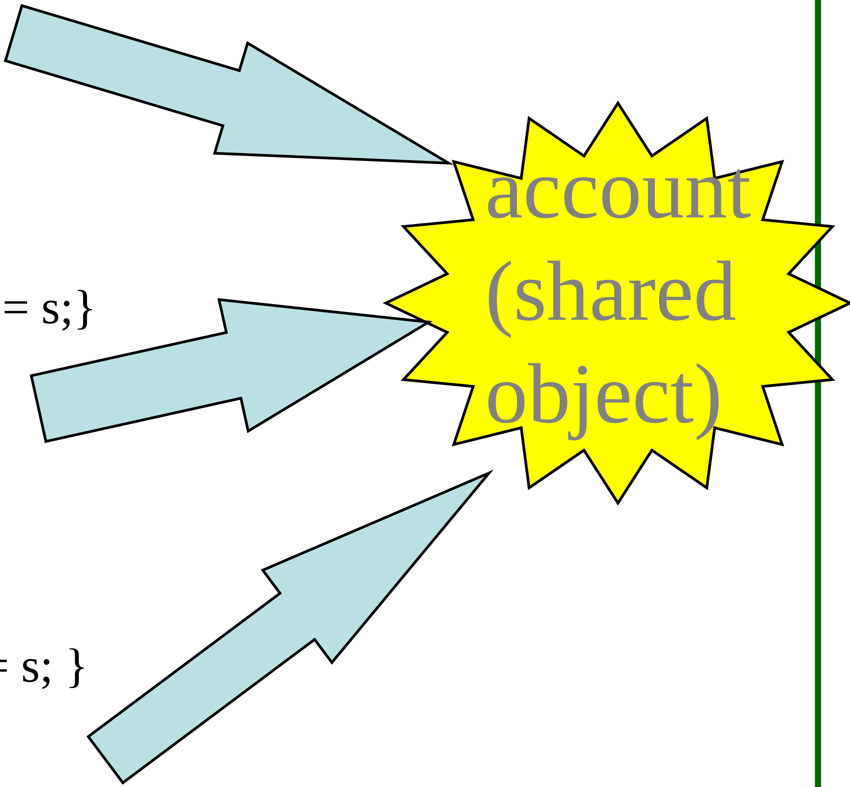
```
class InternetBankingSystem {
        public static void main(String [] args  ) {
            Account accountObject = new Account ();
            Thread t1 = new Thread(new MyThread(accountObject));
            Thread t2 = new Thread(new YourThread(accountObject));
            Thread t3 = new Thread(new HerThread(accountObject));
            t1.start();
            t2.start();
            t3.start();
          // DO some other operation
        } // end main()
}
```

Shared account object between 3 threads

```
class MyThread implements Runnable  {
 Account account;
        public MyThread (Account s) {  account = s;}
        public void run() { account.deposit(); }
} // end class MyThread
```

runnable 객체

```
class YourThread implements Runnable  {
 Account account;
        public YourThread (Account s) { account = s;}
        public void run() { account.withdraw(); }
} // end class YourThread

class HerThread implements Runnable  {
 Account account;
        public HerThread (Account s) { account = s; }
        public void run() {account.enquire(); }
} // end class HerThread
```



account (shared object)

Monitor (shared object access): serializes operation on shared object

```
class Account {   // the 'monitor'
   int balance;

      // if 'synchronized' is removed, the outcome is unpredictable
       public synchronized void deposit( ) {
          // METHOD BODY : balance += deposit_amount;
        }

        public synchronized void withdraw( ) {
          // METHOD BODY: balance -= deposit_amount;
        }
        public synchronized void enquire( ) {
          // METHOD BODY: display balance.
        }
}
```

References

- Rajkumar Buyya, Thamarai Selvi, Xingchen Chu, **Mastering OOP with Java**, McGraw Hill (I) Press, New Delhi, India, 2009.
- Sun Java Tutorial – Concurrency:
  - http://java.sun.com/docs/books/tutorial/essential/concurrency/

End.