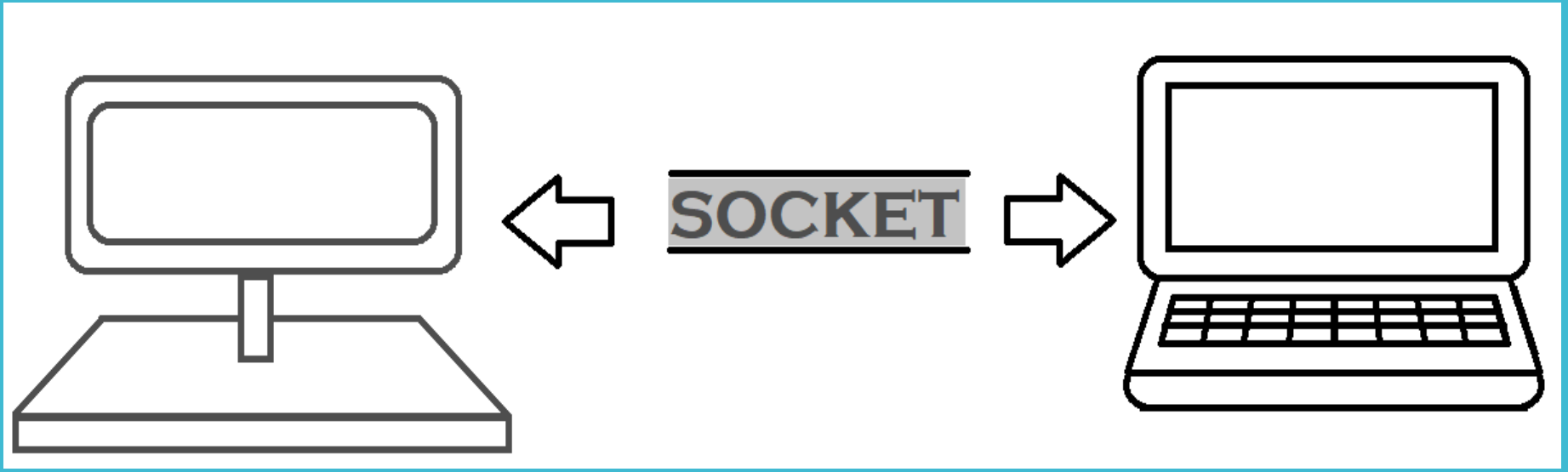
Advance Networking



Jatin Ambasana

SUBJECT:
Advanced Java Programming

TOPIC:
Advance Networking

Agenda

- Networking Basics
  - TCP, UDP, Ports, DNS, Client-Server Model
- TCP/IP in Java
- Sockets
- URL
  - The java classes: `URL, URLConnection`
- Datagrams

JAVA NETWORKING

- Java Networking is a **concept of connecting two or more computing devices** together so that we can share resources.
- Java **socket programming** provides facility to share data between different computing devices.
- Advantage of Java Networking : Sharing resources

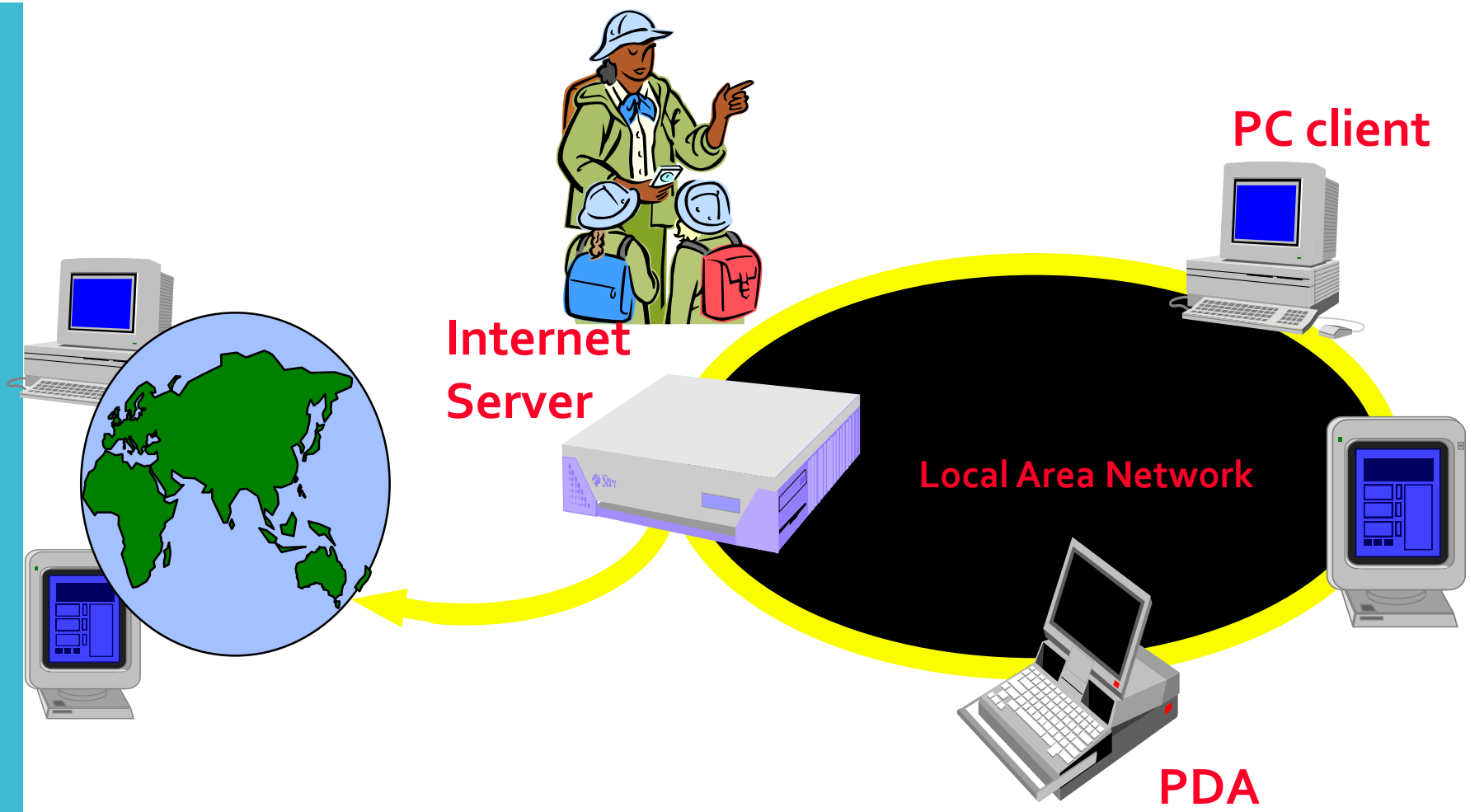
Java Networking Terminology

- The widely used java networking terminologies are given below:
  1. **IP Address :** IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255. It is a logical address that can be changed.
  2. **Protocol :** A protocol is a set of rules basically that is followed for communication. For example:
     I. TCP
     II. FTP
     III. Telnet
     IV. SMTP
     V. POP etc.

Java Networking Terminology

3. **Port Number** : The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications. The port number is associated with the IP address for communication between two applications.
4. **MAC Address :** MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.
5. **Connection-oriented And Connection-less Protocol** : In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP. But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP.
6. **Socket** : A socket is an endpoint between two way communication

The Network is Computer

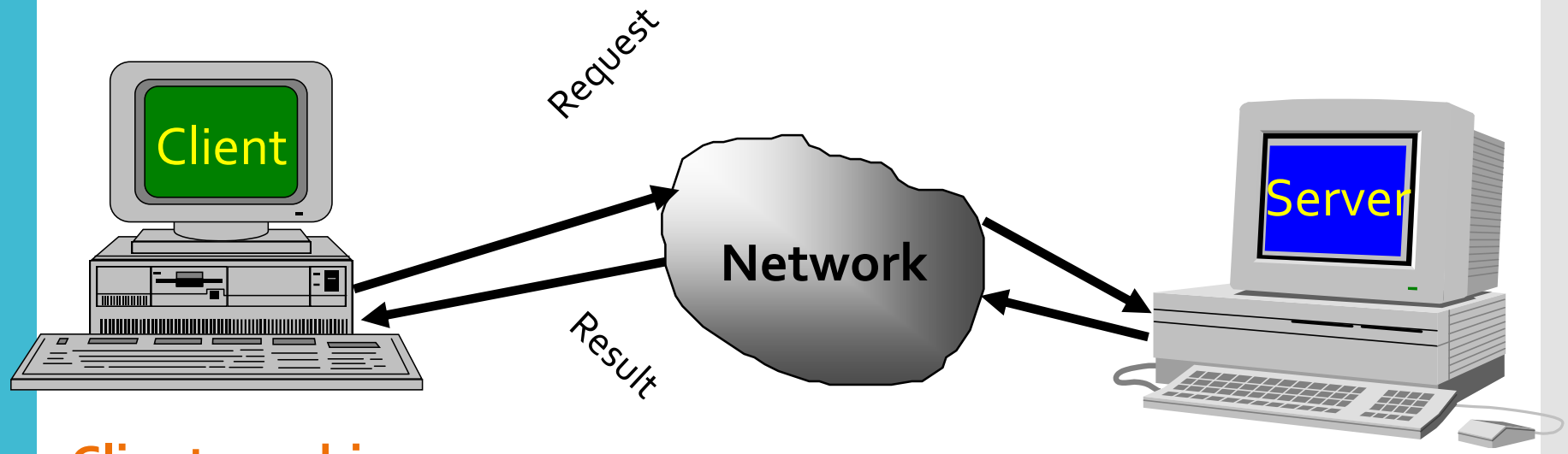


Increased demand for Internet applications

- To take advantage of opportunities presented by the Internet, businesses are continuously seeking new and innovative ways and means for offering their services via the Internet.
- This created a huge demand for software designers with skills to create new Internet-enabled applications or migrate existing/legacy applications on the Internet platform.
- Object-oriented Java technologies—Sockets, threads, RMI, clustering, Web services-- have emerged as leading solutions for creating portable, efficient, and maintainable large and complex Internet applications.

Elements of C-S Computing

a client, a server, and network

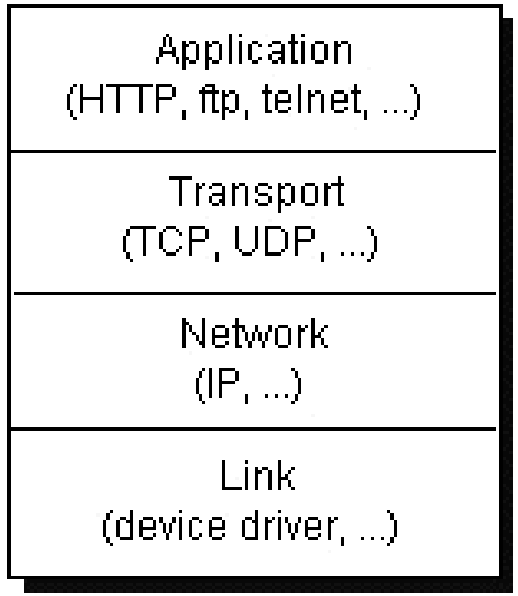


Client machine

Server machine

Networking Basics

Computers running on the Internet communicate with each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP)



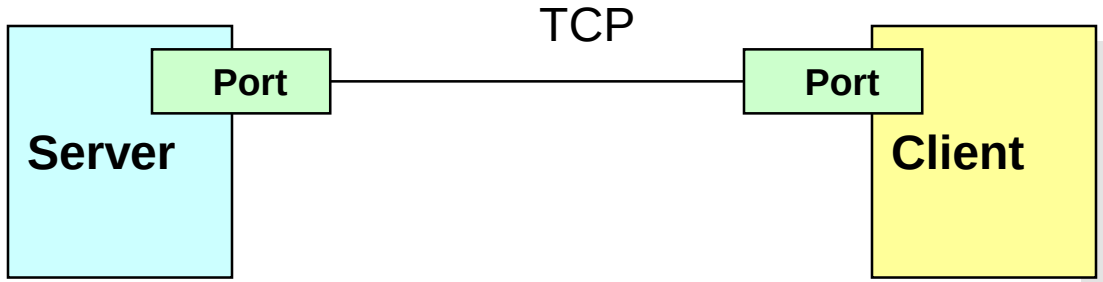
Java Socket Programming

- Java Socket programming is used for **communication between the applications** running on different JRE.
- Java Socket programming **can be connection-oriented or connection-less**.
- **Socket and ServerSocket classes** are used for connection-oriented socket programming and **DatagramSocket and DatagramPacket** classes are used for connection-less socket programming.
- The client in socket programming
- must know two information:
  1. IP Address of Server, and
  2. Port number

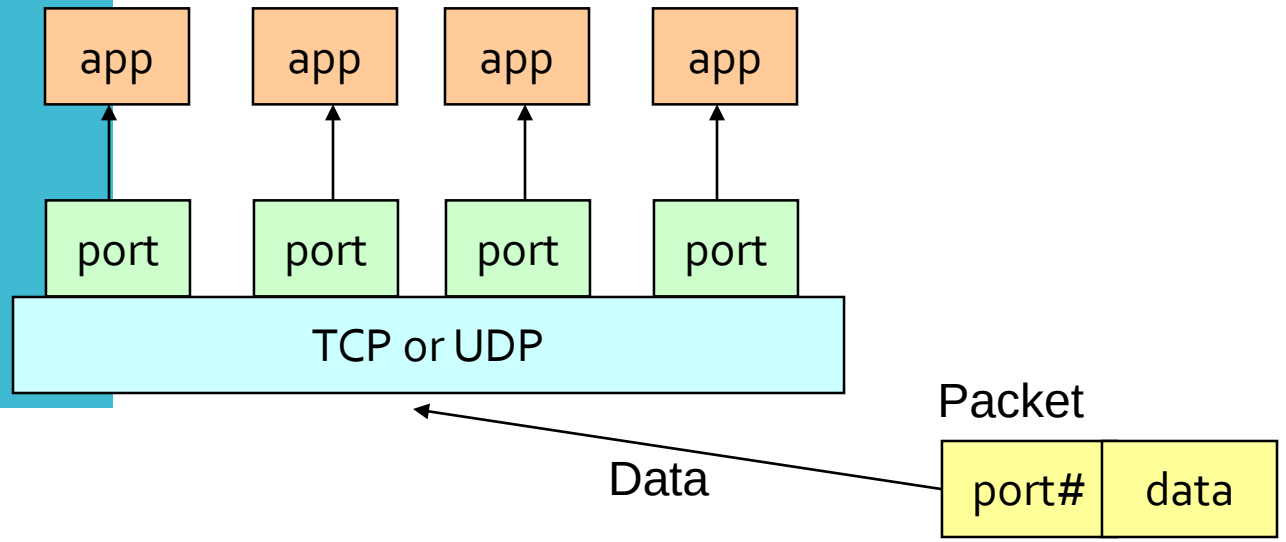
DNS - Domain name system

- The **Domain Name system** (DNS) associates various sorts of information with so-called domain names.
- Most importantly, it serves as the "phone book" for the Internet by translating human-readable computer hostnames, e.g. *www.example.com*, into the IP addresses, e.g. *208.77.188.166*, that networking equipment needs to deliver information.
- It also stores other information such as the list of mail exchange servers that accept email for a given domain.

Understanding Ports



- The TCP and UDP protocols use *ports* to map incoming data to a particular *process* running on a computer.



Understanding Ports

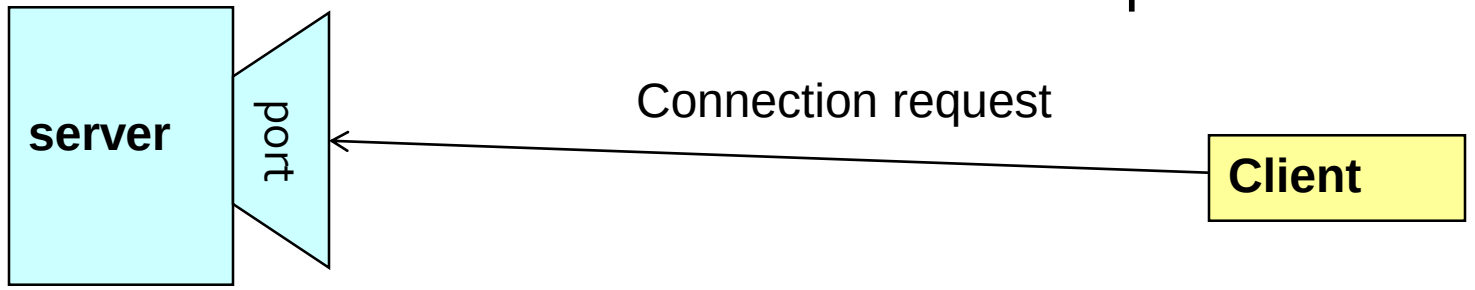
- Port is represented by a positive (16-bit) integer value
- Some ports have been reserved to support common/well known services:
  - ftp 21/tcp
  - telnet 23/tcp
  - smtp 25/tcp
  - login 513/tcp
- User level process/services generally use port number value >= 1024

Sockets

- Sockets provide an interface for programming networks at the transport layer.
- Network communication using Sockets is very much similar to performing file I/O
  - In fact, socket handle is treated like file handle.
  - The streams used in file I/O operation are also applicable to socket-based I/O
- Socket-based communication is programming language independent.
  - That means, a socket program written in Java language can also communicate to a program written in Java or non-Java socket program.

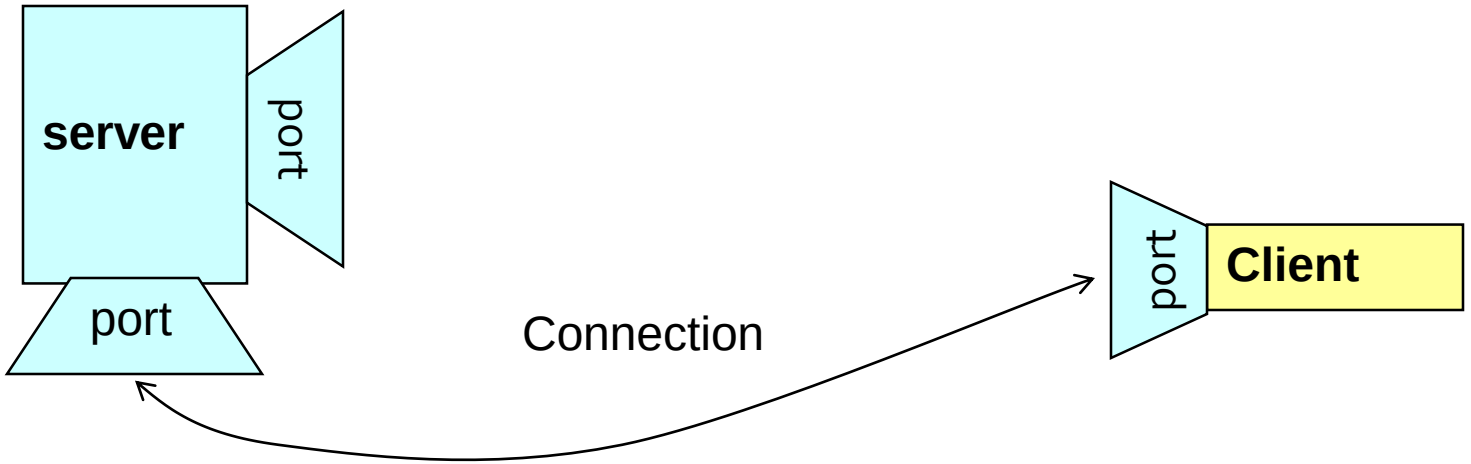
Socket Communication

- A server (program) runs on a specific computer and has a socket that is bound to a specific port. The server waits and listens to the socket for a client to make a connection request.

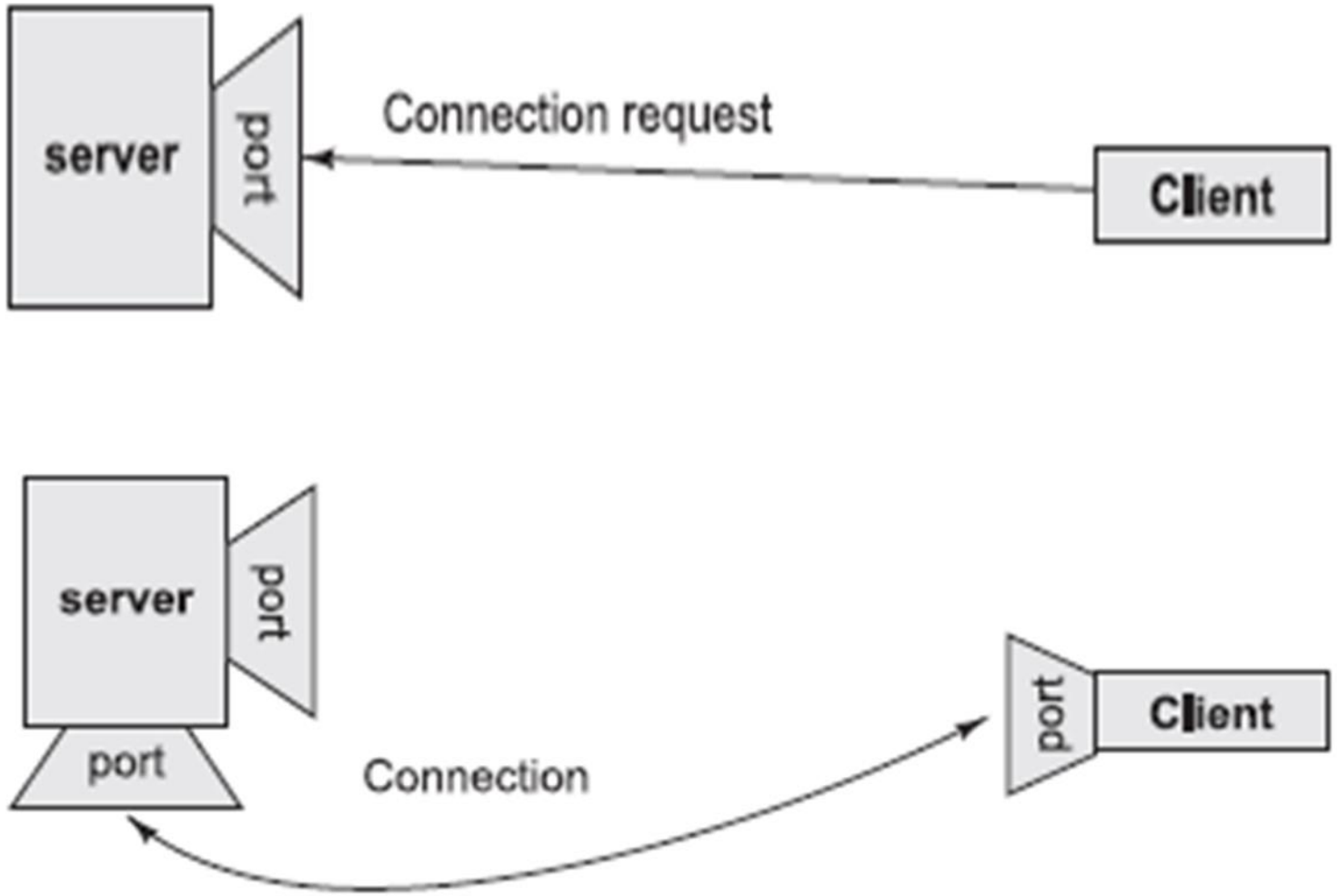


Socket Communication

- If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bounds to a different port. It needs a new socket (consequently a different port number) so that it can continue to listen to the original socket for connection requests while serving the connected client.



Socket Communication



Cont.

The Classes

- ContentHandler, DatagramPacket, DatagramSocket, DatagramSocketImpl, HttpURLConnection, InetAddress, MulticastSocket, ServerSocket, Socket, SocketImpl, URL, URLConnection

The Interfaces

- ContentHandlerFactory, FileNameMap, SocketImplFactory , URLStreamHandlerFactory

Exceptions

- BindException, ConnectException, MalformedURLException , NoRouteToHostException, ProtocolException, SocketException , UnknownHostException, UnknownServiceException

Transmission Control Protocol

- A connection-based protocol that provides a reliable flow of data between two computers.
- Provides a point-to-point channel for applications that require reliable communications.
  - The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel
- Guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.

User Datagram Protocol

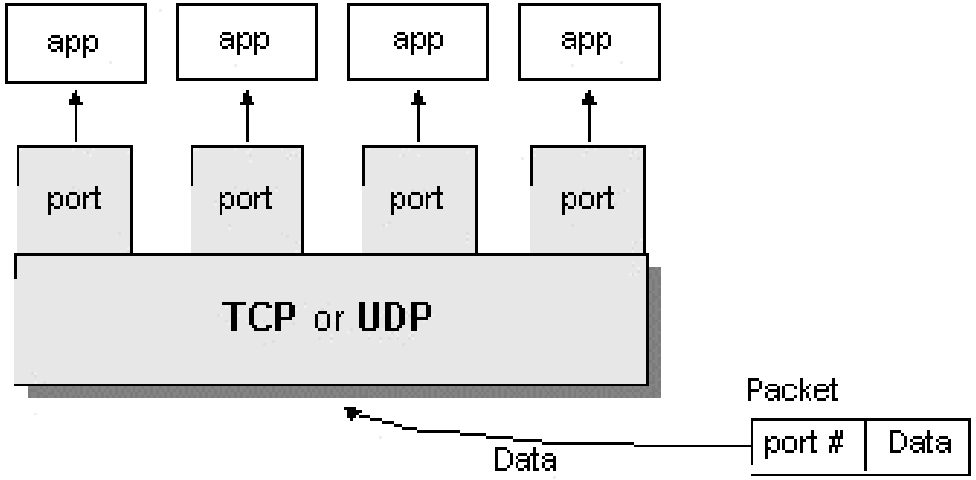
- A protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival. UDP is not connection-based like TCP and is not reliable:
  - Sender does not wait for acknowledgements
  - Arrival order is not guaranteed
  - Arrival is not guaranteed
- Used when speed is essential, even in cost of reliability
  - e.g. streaming media, games, Internet telephony, etc.

Ports

- Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined.
  - The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.

Ports – Cont.

- Port numbers range from 0 to 65,535 (16-bit)
  - Ports 0 - 1023 are called *well-known ports.* They are reserved for use by well-known services:
    - 20, 21: FTP
    - 23: TELNET
    - 25: SMTP
    - 110: POP3
    - 80: HTTP



Networking Classes in the JDK

- Through the classes in java.net, Java programs can use TCP or UDP to communicate over the Internet.
  - The `URL`, `URLConnection`, `Socket`, and `ServerSocket` classes all use TCP to communicate over the network.
  - The `DatagramPacket`, `DatagramSocket`, and `MulticastSocket` classes are for use with UDP.

Networking Classes in the JDK

- InetAddress
- Socket and ServerSocket Class
- URL class

TCP/IP in Java

- Accessing TCP/IP from Java is straightforward. The main functionality is in the following classes:
  - `java.net.InetAddress` : Represents an IP address (either IPv4 or IPv6) and has methods for performing DNS lookup (next slide).
  - `java.net.Socket` : Represents a TCP socket.
  - `java.net.ServerSocket` : Represents a server socket which is capable of waiting for requests from clients.

InetAddress

- The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address.
- We interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address.
- The InetAddress class hides the number inside.

Factory Methods

- static InetAddress getLocalHost( )

  throws UnknownHostException

- static InetAddress getByName(String hostName)

  throws UnknownHostException

- static InetAddress[ ] getAllByName(String hostName)

  throws UnknownHostException

Example:

```
class InetAddressTest1
{
 public static void main(String args[])
      throws UnknownHostException
 {
   InetAddress Address = InetAddress.getLocalHost();
   System.out.println(Address);
   Address =
            InetAddress.getByName("www.google.com");
   System.out.println(Address);
   InetAddress SW[] =
            InetAddress.getAllByName("www.yahoo.com");
   for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
 }
}
```

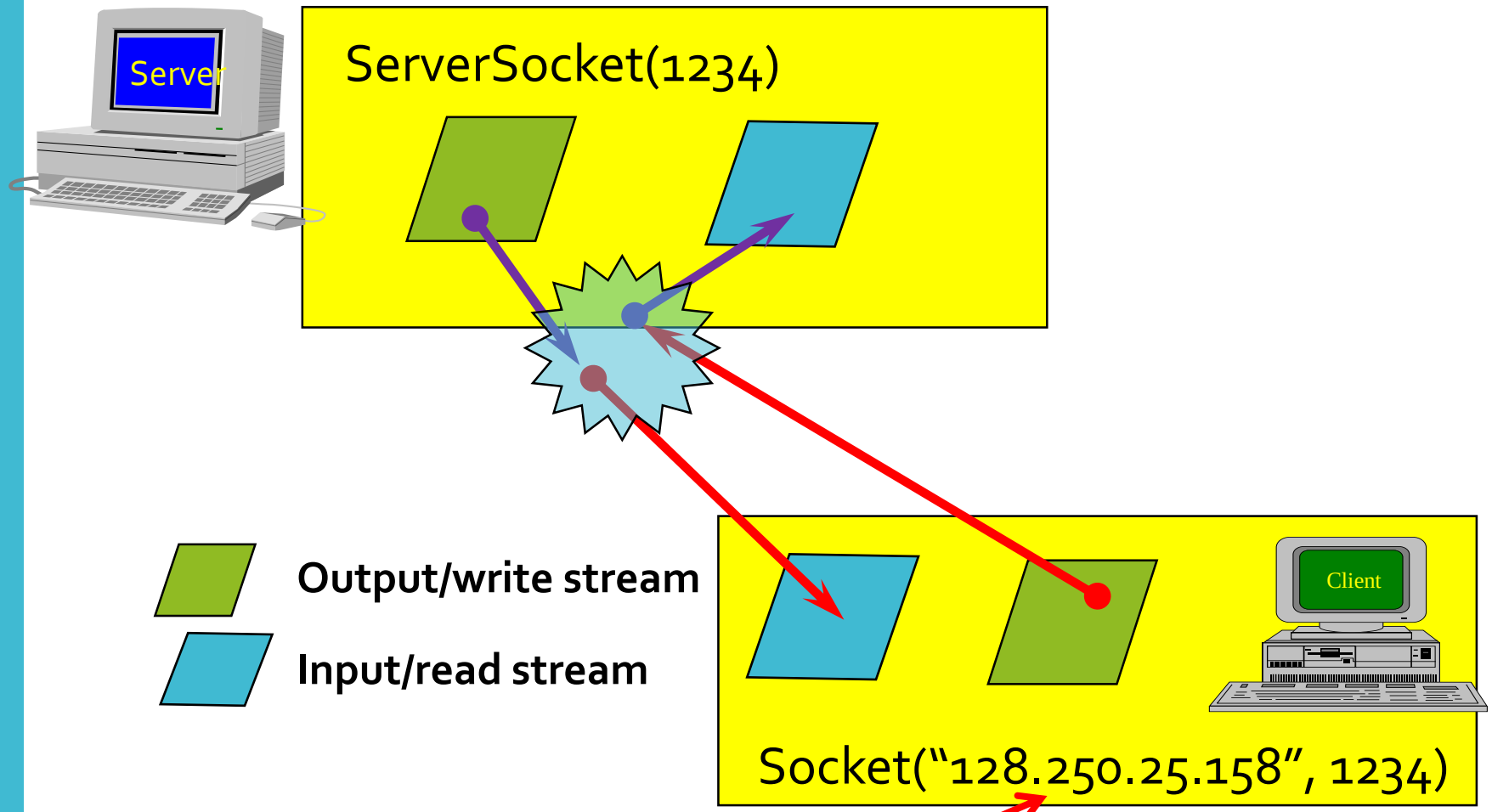
Instance Methods

```
class InetAddressTest2
{
  public static void main(String args[])
   throws UnknownHostException
  {
    InetAddress Address =
    InetAddress.getByName("www.google.com");
    System.out.println(Address.getHostAddress());
    System.out.println(Address.getHostName());
    if(Address.isMulticastAddress())
      System.out.println("It is multicast address");
  }
}
```

Sockets and Java Socket Classes

- A socket is an 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 destined to be sent.
- Java's .net package provides two classes:
  - Socket – for implementing a client
  - ServerSocket – for implementing a server

Java Sockets



It can be host_name like
"books.google.com" or "localhost"

Client Sockets

- Java wraps OS sockets (over TCP) by the objects of class java.net.Socket

`Socket(String remoteHost,int remotePort)`

- Creates a TCP socket and connects it to the remote host on the remote port (hand shake)
- Write and read using streams:
  - InputStream getInputStream()
  - OutputStream getOutputStream()

Constructors

- Socket(String *remoteHost*, int *remotePort*)
- Socket(InetAddress ip, int *remotePort*)

Instance Methods

- InetAddress getInetAddress( )
- int getPort( )
- int getLocalPort( )
- InputStream getInputStream( )
- OutputStream getOutputStream( )
- void close( )

Socket class

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

A **socket is simply an endpoint for communications** between the machines. The Socket class can be used to create a socket.

ServerSocket class

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

The **ServerSocket class can be used to create a server socket**.
This object is used to establish communication with the clients.

Implementing a Client

1. Create a Socket Object:

```
client = new Socket( server, port_id );
```

2. Create I/O streams for communicating with the server.

```
is = new DataInputStream(client.getInputStream() );
os = new DataOutputStream( client.getOutputStream() );
```

3. Perform I/O or communication with the server:
   - Receive data from the server:

```
String line = is.readLine();
```

   - Send data to the server:

```
os.writeBytes("Hello\n");
```

4. Close the socket when done:

```
client.close();
```

Example: Whois server

```
class Whois {
   public static void main(String args[  ]) throws
   Exception {
      int c;
      Socket s = new Socket("internic.net", 43);
      InputStream in = s.getInputStream();
      OutputStream out = s.getOutputStream();
      String str="www.google.com";
      byte buf[] = str.getBytes();
      out.write(buf);
      while ((c = in.read()) != -1)
      System.out.print((char) c);
      s.close();
   }
}
```

Example: Time server

```
public class DayTime{

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

   Socket theSocket = new Socket("time.nist.gov", 13);

   InputStream timeStream =theSocket.getInputStream();

   StringBuffer time = new StringBuffer( );

   int c;

   while ((c = timeStream.read( )) != -1)
       time.append((char) c);

   String timeString = time.toString( ).trim( );

   System.out.println("It is " + timeString + " at " +
   "localhost");

   }

}
```

ServerSocket

- This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.
- A server socket is technically not a socket: when a client connects to a server socket, a TCP connection is made, and a (normal) socket is created for each end point.

Constructors

- `ServerSocket (int port)`
  `throws BindException, IOException`
- `ServerSocket (int port, int maxQueue)`
  `throws BindException, IOException`
- `ServerSocket (int port, int maxQ, InetAddress ip)` *`throws IOException`*

Implementing a Server

- Open the Server Socket:

```
ServerSocket server;
DataOutputStream os;
DataInputStream is;
server = new ServerSocket( PORT );
```

- Wait for the Client Request:

```
Socket client = server.accept();
```

- Create I/O streams for communicating to the client

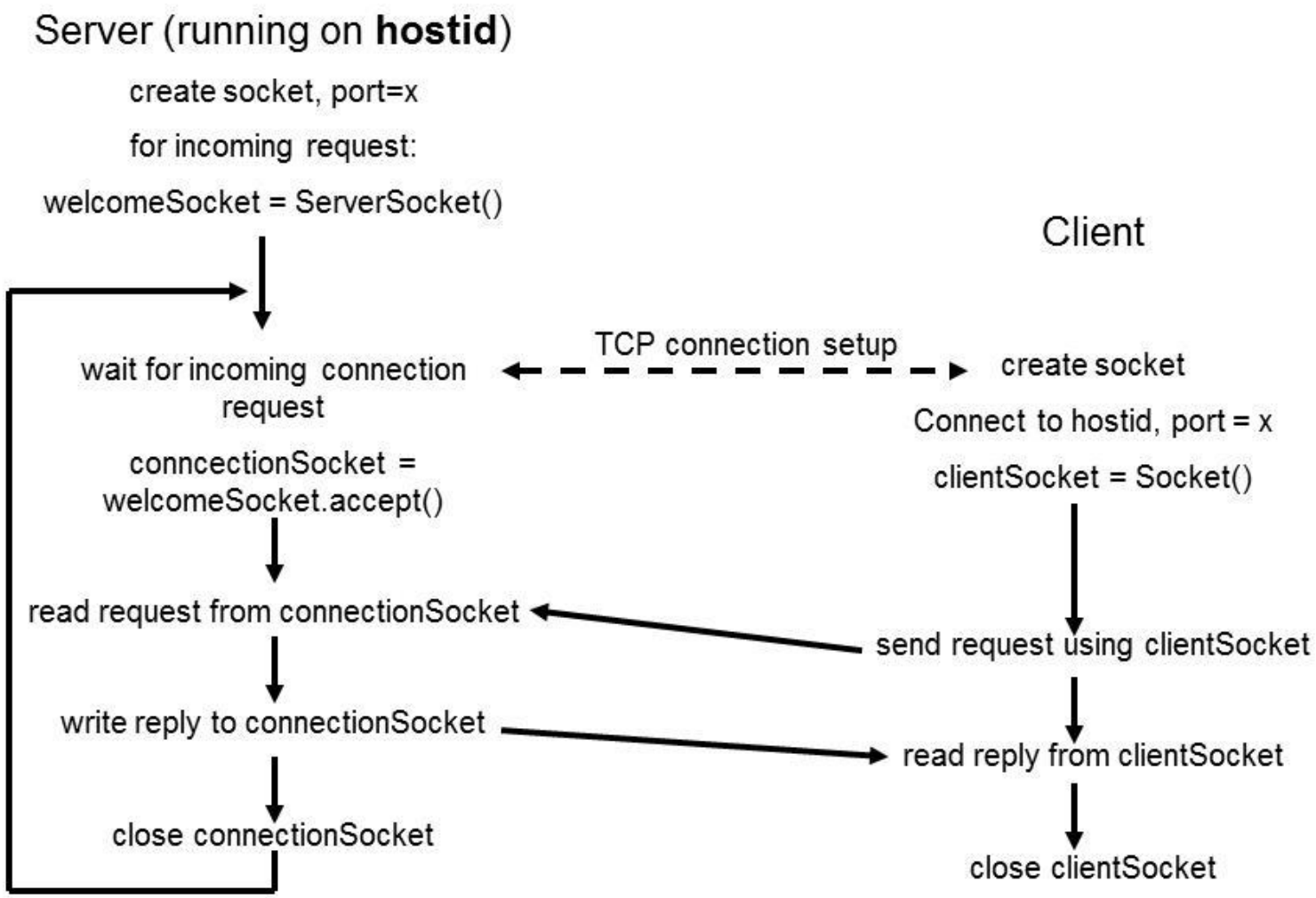
```
is = new DataInputStream(client.getInputStream() );
os = new DataOutputStream(client.getOutputStream());
```

- Perform communication with client
  - Receive from client: `String line = is.readLine();`
  - Send to client: `os.writeBytes("Hello\n");`
- Close sockets: `client.close();`

Accepting Connections

- Usually, the `accept()` method is executed within an infinite loop
  - i.e., `while(true){...}`
- The accept method returns a new socket (with a new port) for the new channel. It blocks until connection is made.
- Syntax:
  - Socket accept() throws IOException

Client-Server Interaction via TCP



Examples

- Server1
- Client1

Program 1: Sending data from a client to a server (Using Connection Oriented Programming)

```
import java.io.*;
import java.net.*;
public class Client1 {
 public static void main(String[] args)
 {
  try{
  Socket s=new Socket("localhost",6666);
  DataOutputStream dout =
  new DataOutputStream(s.getOutputStream());
  dout.writeUTF("Hello Server");
  dout.flush();
  dout.close();
  s.close();
  }

catch(Exception e){System.out.println(e);}
 }
}
```

```
import java.io.*;
import java.net.*;
public class Server1 {
public static void main(String[] args){
 try{
  ServerSocket ss=new ServerSocket(6666);
  Socket s=ss.accept();
  DataInputStream din =
  new DataInputStream(s.getInputStream());
  String  str=(String)din.readUTF();
  System.out.println("Client: "+str);
  din.close();
  s.close();
  ss.close();
 }
 catch(Exception e)
 {System.out.println(e);}
}
}
```

Server1.java

Program 1: Sending data from client to server

```
import java.io.*;
import java.net.*;
public class Server1 {
public static void main(String[] args){
 try{
  ServerSocket ss=new ServerSocket(6666);
  Socket s=ss.accept();
  DataInputStream dis =
  new DataInputStream(s.getInputStream());
  String  str=(String)dis.readUTF();
  System.out.println("Client says= "+str);
  ss.close();
 }
 catch(Exception e)
 {System.out.println(e);}
}
}
```

Client1.java

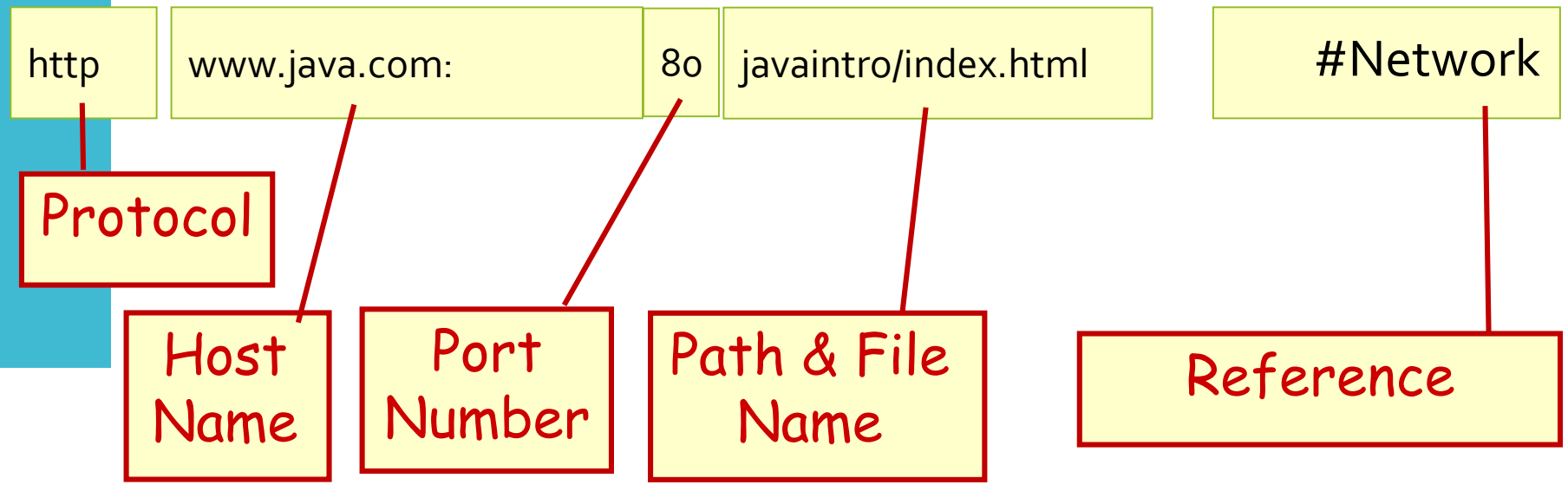
Program 1: Sending data from client to server

```
import java.io.*;
import java.net.*;
public class Client1 {
 public static void main(String[] args)
 {
  try{
  Socket s=new Socket("localhost",6666);
  DataOutputStream dout =
  new DataOutputStream(s.getOutputStream());
  dout.writeUTF("Hello Server");
  dout.flush();
  dout.close();
  s.close();
  }
  catch(Exception e){System.out.println(e);}
 }
}
```

URL - *Uniform Resource Locator*

- URL is a reference (an address) to a resource on the Internet.
  - A resource can be a file, a database query and more.
- URLs are just a subset of the more general concept of Uniform Resource Identifiers (URIs) which are meant to describe all points in the information space

  http://www.java.com:80/javaintro/index.html#Network



Class `URL`

- Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.
- We distinguish between:
  - Absolute URL - contains all of the information necessary to reach the resource.
  - Relative URL - contains only enough information to reach the resource relative to (or in the context of) another URL.

Constructors

- URL(String urlSpecifier)
- URL(URL urlObj, String urlSpecifier)
- URL(String protName, String hostName, int port, String path)
- URL(String protName, String hostName, String path)

Commonly used methods of Java URL class

Method	Description
public String getProtocol()	Returns the protocol of the URL.
public String getHost()	Returns the host name of the URL.
public String getPort()	Returns returns the Port Number of the URL.
public String getFile()	Returns the file name of the URL.
public URLConnection openConnection()	Returns the instance of URLConnection i.e. associated with this URL.
public String getQuery()	Returns the query string of the URL.
public String toString()	Returns the string representation of the URL.
public String getDefaultPort()	it returns the default port of the URL.

Example of Java URL class

```
import java.io.*;
import java.net.*;
public class URLDemo{
public static void main(String[ ] args){
try{
  URL url=new URL("http://srpec.org.in/contacts");
  System.out.println("Protocol: "+url.getProtocol());
  System.out.println("Host Name: "+url.getHost());
  System.out.println("Port Number: "+url.getPort());
  System.out.println("File Name: "+url.getFile());
}
catch(Exception e){System.out.println(e);}
}
}
```

Example

```
class URLDemo{
public static void main(String args[]) throws
MalformedURLException
{
 URL hp = new URL("http://content-
ind.cricinfo.com/ci/content/current/story/news.html");
 System.out.println("Protocol: " + hp.getProtocol());
 System.out.println("Port: " + hp.getPort());
 System.out.println("Host: " + hp.getHost());
 System.out.println("File: " + hp.getFile());
 System.out.println("Ext:" + hp.toExternalForm());
}
}
```

Output

Protocol: http

Port: -1

Host: content-ind.cricinfo.com

File: /ci/content/current/story/news.html

Ext:http://content-ind.cricinfo.com/ci/content/current/story/news.html

URLConnection class

- The **Java URLConnection** class represents a communication link between the URL and the application.
- This class can be used to read and write data to the specified resource referred by the URL.
- The **openConnection()** method of URL class returns the object of URLConnection class.

URLConnection class

- URLConnection is an abstract class that represents an active connection to a resource specified by a URL.
- The URLConnection class has two different but related purposes.
- First, it provides more control over the interaction with a server (especially an HTTP server) than the URL class.
- We can inspect the header sent by the server and respond accordingly. We can set the header fields used in the client request. We can use a URLConnection to download binary files.
- URLConnection lets us send data back to a web server with POST or PUT and use other HTTP request methods.

Process

1. Construct a URL object.
2. Invoke the URL object's openConnection( ) method to retrieve a URLConnection object for that URL.
3. Configure the URLConnection.
4. Read the header fields.
5. Get an input stream and read data.
6. Get an output stream and write data.
7. Close the connection.

Reading Data from Server

1. Construct a URL object.
2. Invoke the URL object's openConnection( ) method to retrieve a URLConnection object for that URL.
3. Invoke the URLConnection's getInputStream( ) method.
4. Read from the input stream using the usual stream API.
5. The getInputStream() method returns a generic InputStream that lets you read and parse the data that the server sends.
6. public InputStream getInputStream( )

URLConnection Example

```
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(
                                new InputStreamReader(
                                yc.getInputStream()));
        String inputLine;

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

URLConnection Example2

```
public class SourceViewer2 {
  public static void main (String[] args) {
    if  (args.length > 0) {
      try {
        //Open the URLConnection for reading
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection( );
        InputStream raw = uc.getInputStream( );
        InputStream buffer = new BufferedInputStream(raw);
        // chain the InputStream to a Reader
        Reader r = new InputStreamReader(buffer);
        int c;
        while ((c = r.read( )) != -1)
        {  System.out.print((char) c);     }
      }
      catch (MalformedURLException ex) {
       System.err.println(args[0]+" is not a parseable URL");
      }
      catch (IOException ex) {
       System.err.println(ex);
      }
    } //  end if
  } // end main
}  // end SourceViewer2
```

Difference between URL and URLConnection

- URLConnection provides access to the HTTP header.
- URLConnection can configure the request parameters sent to the server.
- URLConnection can write data to the server as well as read data from the server.

Header Information

- HTTP/1.1 200 OK
- Date: Mon, 18 Oct 1999 20:06:48 GMT
- Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
- Last-Modified: Mon, 18 Oct 1999 12:58:21 GMT
- Accept-Ranges: bytes
- Content-Length: 35259
- Connection: close
- Content-Type: text/html

Methods

- public String getContentType( )
- public int getContentLength( )
- public long getDate( )
- public long getExpiration( )
- public long getLastModified( )

Example

```
public class HeaderViewer {

public static void main(String args[])    {
try       {

URL u = new
URL("http://www.rediffmail.com/index.html")
;

URLConnection uc = u.openConnection( );

System.out.println("Content-type: " +

uc.getContentType( ));

System.out.println("Content-encoding: "

+ uc.getContentEncoding( ));

System.out.println("Date: " + new
Date(uc.getDate( )));

System.out.println("Last modified: "

+ new Date(uc.getLastModified( )));

System.out.println("Expiration date: "

+ new Date(uc.getExpiration( )));

System.out.println("Content-length: " +

uc.getContentLength( ));

}  // end try

catch (MalformedURLException ex) {
  System.out.println("I can't
understand this URL...");
}
catch (IOException ex) {
  System.err.println(ex);
}
 System.out.println( );
 }  // end main
}  // end HeaderViewer
```

Sample Output

- **Sample output:**

  Content-type: text/htmlContent-encoding: nullDate: Mon Oct 18 13:54:52 PDT 1999Last modified: Sat Oct 16 07:54:02 PDT 1999Expiration date: Wed Dec 31 16:00:00 PST 1969 Content-length: -1

- **Sample output for: http://www.oreilly.com/graphics/space.gif**

  Content-type: image/gifContent-encoding: nullDate: Mon Oct 18 14:00:07 PDT 1999Last modified: Thu Jan 09 12:05:11 PST 1997Expiration date: Wed Dec 31 16:00:00 PST 1969 Content-length: 57

Retrieving Header field

- public String getHeaderField(String name)
- Example:
  - `String contentType = uc.getHeaderField("content-type");`
  - `String contentEncoding = uc.getHeaderField("content-encoding"));`
  - `String data = uc.getHeaderField("date");`
  - `String expires = uc.getHeaderField("expires");`
  - `String contentLength = uc.getHeaderField("Content-length");`

Java HttpURLConnection class

- The **Java HttpURLConnection** class is http specific URLConnection. It works for HTTP protocol only.
- By the help of HttpURLConnection class, you can information of any HTTP URL such as header information, status code, response code etc.
- The java.net.HttpURLConnection is subclass of URLConnection class.

How to get the object of HttpURLConnection class

The openConnection() method of URL class returns the object of URLConnection class.
Syntax:

- **public** URLConnection openConnection()**throws** IOException{}

You can typecast it to HttpURLConnection type as given below.

- URL url=**new** URL("http://www.javatpoint.com/java-tutorial");
- HttpURLConnection huc=(HttpURLConnection)url.openConnection();

Datagrams

UDP / Connection-less Programming

TCP vs. UDP

No.	TCP	UDP
1	This Connection oriented protocol	This is connection-less protocol
2	The TCP connection is byte stream	The UDP connection is a message stream
3	It does not support multicasting and broadcasting	It supports broadcasting
4	It provides error control and flow control	The error control and flow control is not provided
5	TCP supports full duplex transmission	UDP does not support full duplex transmission
6	It is reliable service of data transmission	This is an unreliable service of data transmission
7	The TCP packet is called as segment	The UDP packet is called as user datagram.

UDP in Java

- DatagramPacket
- DatagramSocket

Datagrams

- A *datagram* is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.
- The java.net package contains three classes to help you write Java programs that use datagrams to send and receive packets over the network: DatagramSocket and DatagramPacket

Receiving DatagramPacket

DatagramPacket Constructor

- `public DatagramPacket(byte[] buffer, int length)`

- `public DatagramPacket(byte[] buffer, int offset, int length)`

  - Example:

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new
DatagramPacket(buffer, buffer.length);
```

Sending Datagrams

- **DatagramPacket Constructor**
- `public DatagramPacket`

```
    (
        byte[] barr, int length,
        InetAddress address, int port
    )
```

  - Creates a datagram packet. This constructor is used to send the packets.

DatagramSocket

DatagramSocket Constructors

- `public DatagramSocket( ) throws SocketException`
- `public DatagramSocket(int port) throws SocketException`
- `public DatagramSocket(int port, InetAddress interface) throws SocketException`

Sending and Receiving Packets

DatagramSocket Methods

- `public void send(DatagramPacket dp) throws IOException`
- `public void receive(DatagramPacket dp) throws IOException`

Program 1: Sending data from a Sender to Receiver (Using Connection-Less Programming)

```
import java.io.*;
import java.net.*;
public class DatagramSender
{
public static void main(String[] args)
throws Exception
{
 DatagramSocket ds = new DatagramSocket();
 String str = "Hello Datagrams";
 InetAddress ip =
           InetAddress.getByName("127.0.0.1");
 DatagramPacket dp =
           new DatagramPacket(str.getBytes(),
                 str.length(), ip, 3000);
 ds.send(dp);
 ds.close();
}
}
```

```
import java.io.*;
import java.net.*;
public class DatagramReceiver
{
public static void main(String[] args)
throws Exception
{
 DatagramSocket ds = new DatagramSocket(3000);
 byte[] buf = new byte[1024];
 DatagramPacket dp =
                new DatagramPacket(buf, 1024);
 ds.receive(dp);
 String str =
    new String(dp.getData(), 0, dp.getLength());
 System.out.println(str);
 ds.close();
}
}
```

DatagramSocket Class

Methods

Method	Description
void bind(SocketAddress addr)	Binds the DatagramSocket to a specific address and port.
void connect(InetAddress address, int port)	Connects the socket to a remote address for the socket.
void send(DatagramPacket p)	Sends the datagram packet from the socket.
void receive(DatagramPacket p)	Receives the datagram packet from the socket.
void disconnect()	Disconnects the socket.
void close()	Closes the datagram socket.
DatagramChannel getChannel()	Returns a unique DatagramChannel object associated with the datagram socket.
InetAddress getInetAddress()	Returns the address to where the socket is connected.
InetAddress getLocalAddress()	Gets the local address to which the socket is connected.
int getLocalPort()	Returns the port number on the local host to which the socket is bound.
int getPort()	Returns the port number to which the socket is connected.
boolean isClosed()	Returns the status of socket i.e. closed or not.
boolean isConnected()	Returns the connection state of the socket.

DatagramPacket Class

Methods

Method	Description
InetAddress getAddress()	It returns the IP address of the machine to which the datagram is being sent or from which the datagram was received.
byte[] getData()	It returns the data buffer.
int getLength()	It returns the length of the data to be sent or the length of the data received.
int getOffset()	It returns the offset of the data to be sent or the offset of the data received.
int getPort()	It returns the port number on the remote host to which the datagram is being sent or from which the datagram was received.
SocketAddress getSocketAddress()	It gets the SocketAddress (IP address + port number) of the remote host that the packet is being sent to or is coming from.
void setAddress (InetAddress iaddr)	It sets the IP address of the machine to which the datagram is being sent.
void setData(byte[] buff)	It sets the data buffer for the packet.
void setLength(int length)	It sets the length of the packet.
void setPort(int iport)	It sets the port number on the remote host to which the datagram is being sent.
void setSocketAddress (SocketAddress addr)	It sets the SocketAddress (IP address + port number) of the remote host to which the datagram is being sent.

Cont.

- The format of datagram packet is:

  | Msg | length | Host | serverPort |

- Java supports datagram communication through the following classes:
- DatagramPacket
- DatagramSocket
- The class DatagramPacket contains several constructors that can be used for creating packet object.
- One of them is:
- DatagramPacket(byte[] buf, int length, InetAddress address, int port);
- This constructor is used for creating a datagram packet for sending packets of length length to the specified port number on the specified host.

Cont.

- The message to be transmitted is indicated in the first argument.

- The key methods of DatagramPacket class are:
- byte[] getData()
- Returns the data size.

- int getLength()
- Returns the length of the data to be sent or the length of the data received.

Cont.

- void setData(byte[] buf)
- Sets the data buffer size for this packet.

- void setLength(int length)
- Sets the length for this packet.

Cont.

- The class DatagramSocket supports various methods that can be used for transmitting or receiving data a datagram over the network. The two key methods are:

- void send(DatagramPacket p)
- Sends a datagram packet from this socket.

- void receive(DatagramPacket p)
- Receives a datagram packet from this socket.

Example:

- UDPServer
- UDPClient

Server Program

```
import java.io.*;
import java.net.*;
public class udp_server
{
   public static void main(String args[])
   {
      DatagramSocket sock = null;
      try
      {
         //1. creating a server socket, parameter is
local port number
         sock = new DatagramSocket(7777);
```

Cont.

```
        //buffer to receive incoming data
        byte[] buffer = new byte[65536];
        DatagramPacket incoming = new
DatagramPacket(buffer, buffer.length);
        //2. Wait for an incoming data
        packet("Server socket created. Waiting for
incoming data...");
        //communication loop
        while(true)
        {
           sock.receive(incoming);
           byte[] data = incoming.getData();
           String s = new String(data, 0,
incoming.getLength());
```

Cont.

```
//echo the details of incoming data - client ip :
client port - client message

packet(incoming.getAddress().getHostAddress()
+ " : " + incoming.getPort() + " - " + s);

            s = "OK : " + s;

            DatagramPacket dp = new
DatagramPacket(s.getBytes() ,
s.getBytes().length , incoming.getAddress() ,
incoming.getPort());

            sock.send(dp);     }        }

      catch(IOException e)

      {    System.err.println("IOException " + e);    }

   } }
```

Client Program

```
import java.io.*;
import java.net.*;
public class udp_client
{
   public static void main(String args[])
   {
      DatagramSocket sock = null;
      int port = 7777;
      String s;
      BufferedReader cin = new BufferedReader(new
InputStreamReader(System.in));
```

Cont.

```
try
    {
        sock = new DatagramSocket();
        InetAddress host =
InetAddress.getByName("localhost");
while(true)
        {
            //take input and send the packet
            packet ("Enter message to send : ");
            s = (String)cin.readLine();
            byte[] b = s.getBytes();
```

Cont.

```
DatagramPacket  dp = new DatagramPacket(b , b.length , host , port);
sock.send(dp);
//now receive reply
//buffer to receive incoming data
byte[] buffer = new byte[65536];
DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
sock.receive(reply);
```

Cont.

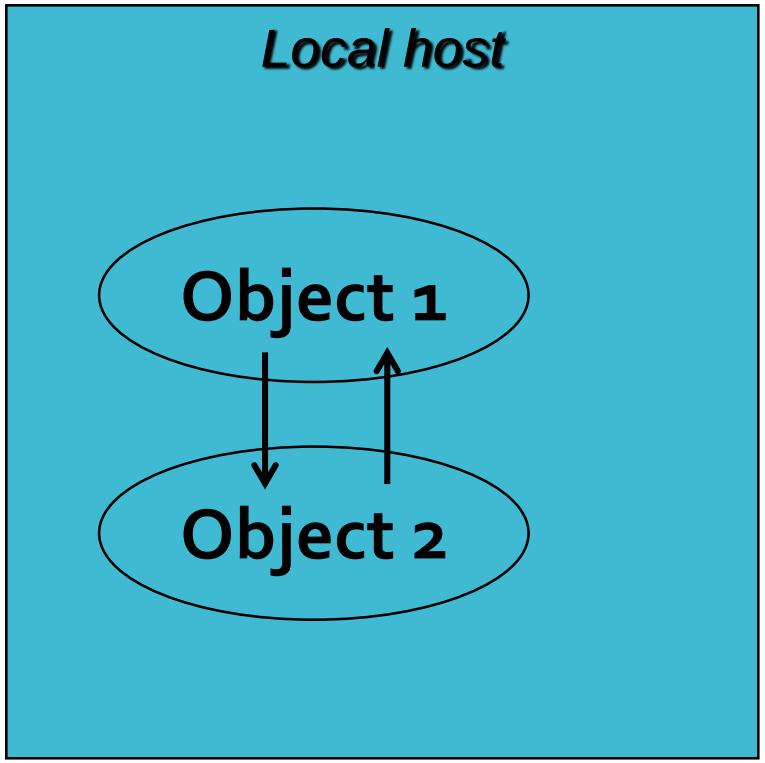
```
            byte[] data = reply.getData();
            s = new String(data, o, reply.getLength());

//echo the details of incoming data - client ip : client
port - client                message
            packet(reply.getAddress().getHostAddress()  +
" : " + reply.getPort() + " - "      + s); } }
      catch(IOException e)
      {
         System.err.println("IOException " + e);      }
   }
}
```

Remote Method Invocation

(RMI)

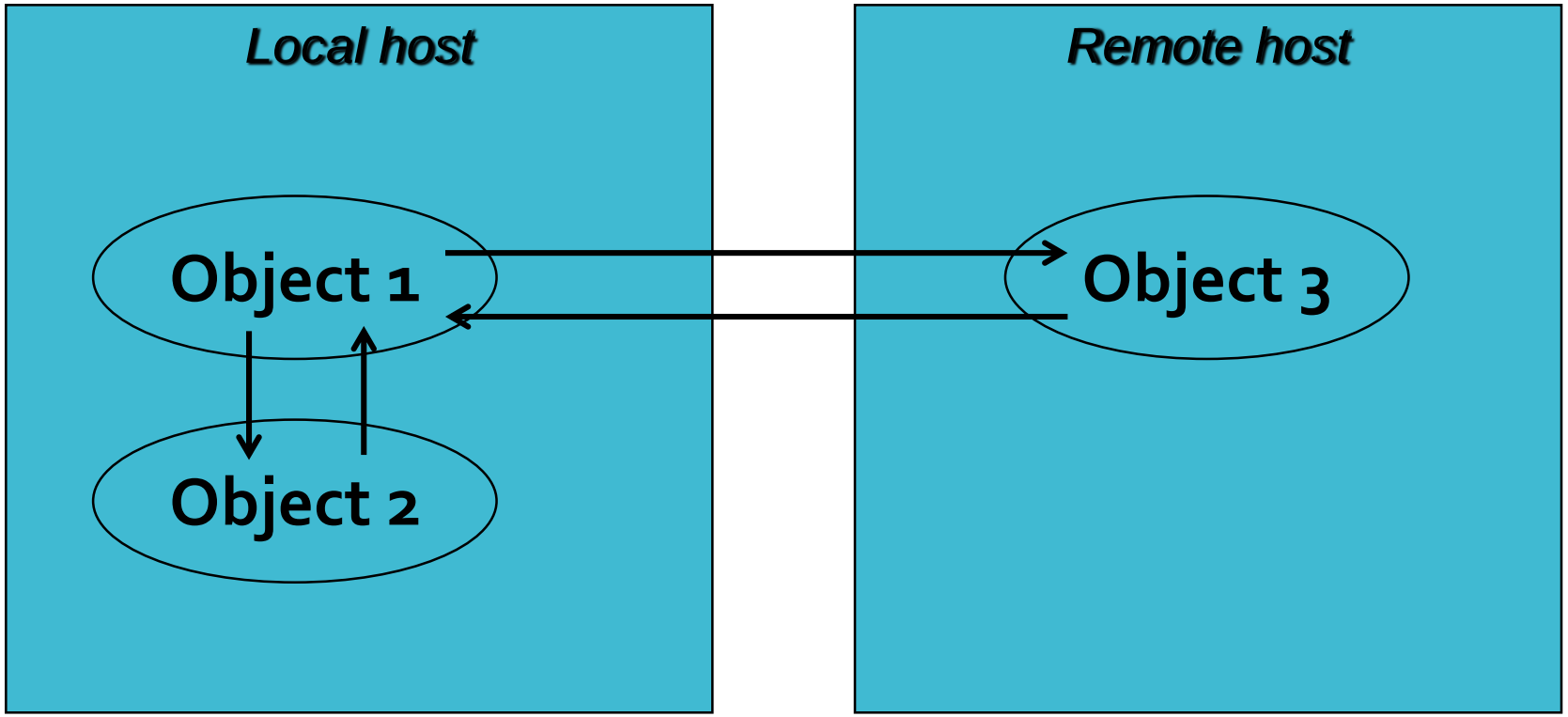
Till now

- We worked with only local objects



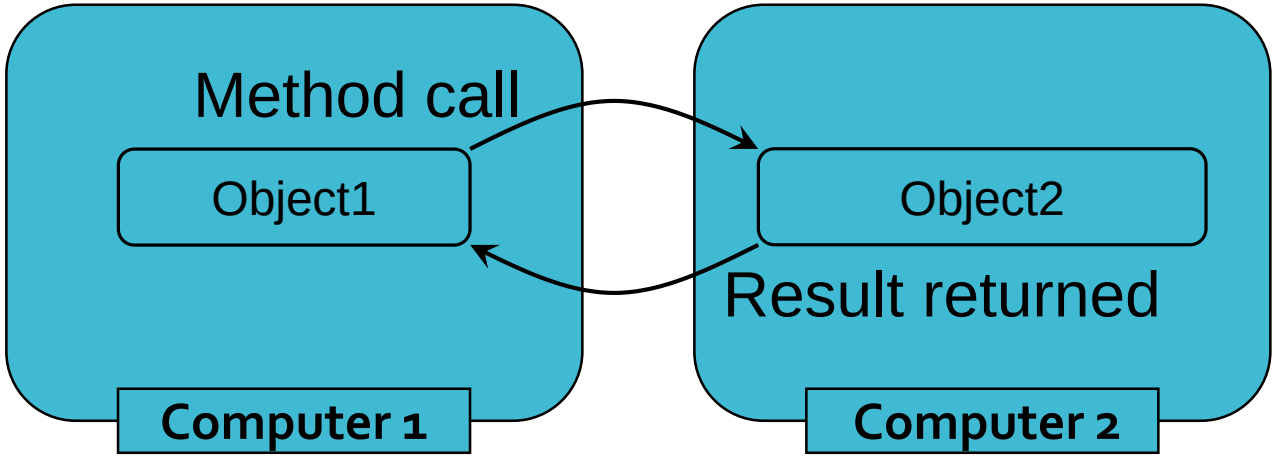
Now…

- We will work with remote objects
  - Network and Distributed Objects



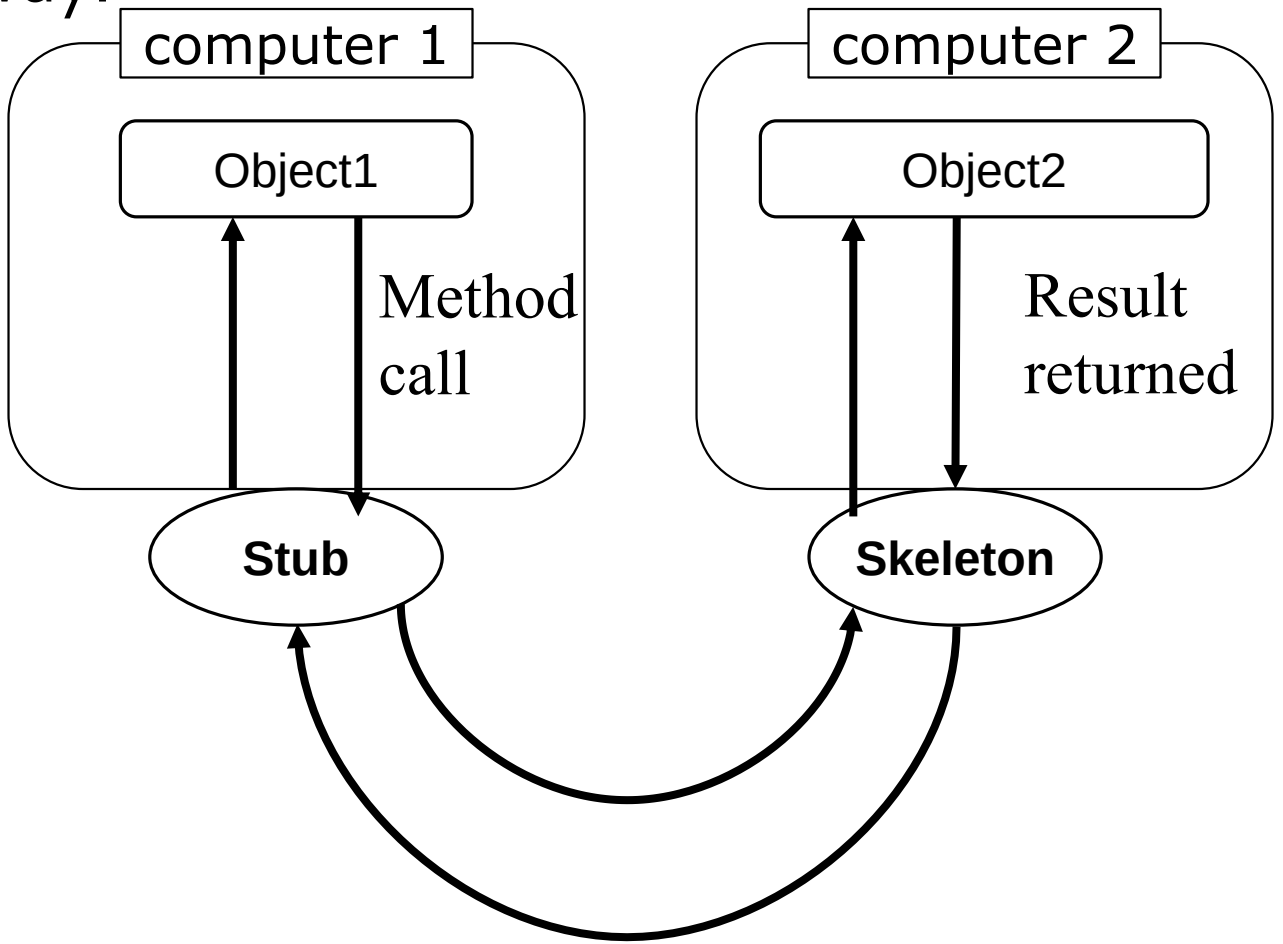
The general idea of RMI

- Instantiate an object on another machine
- Invoke methods on the remote object



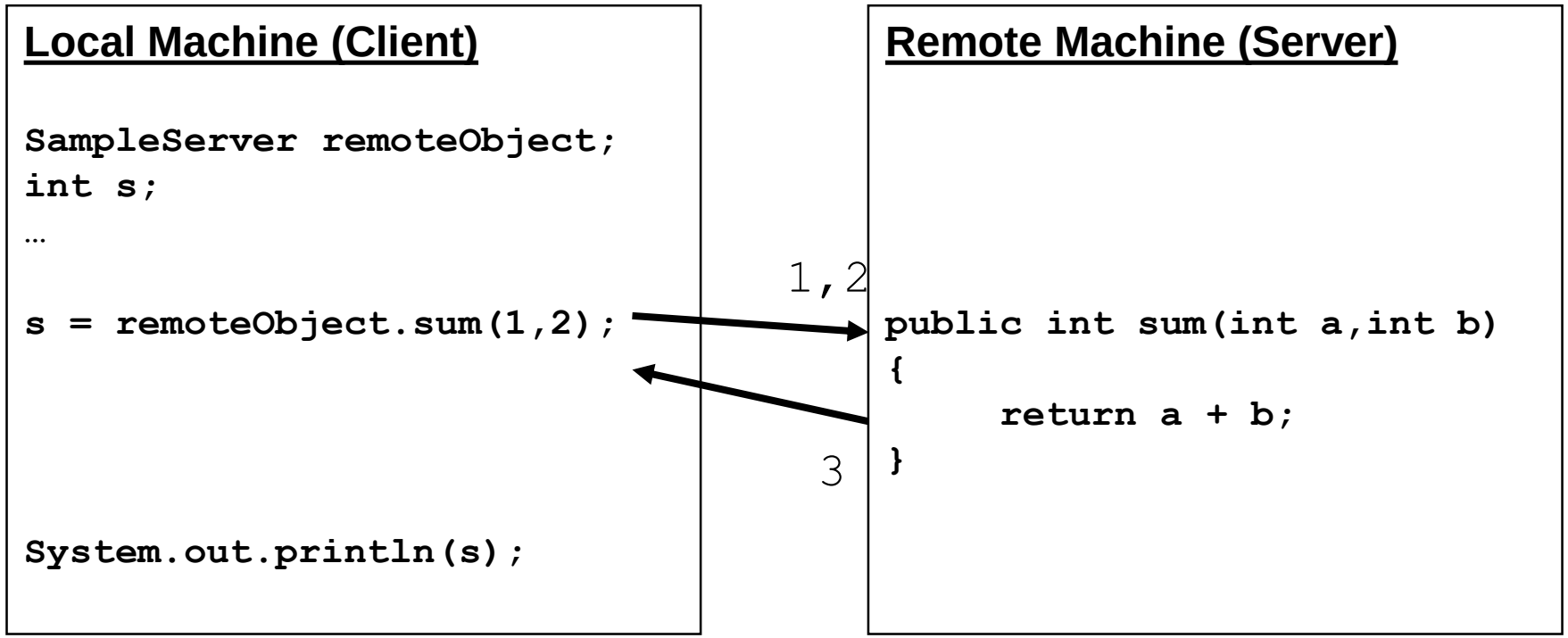
The general idea of RMI with Stub & Skeleton

- Same idea represented by many in following way:

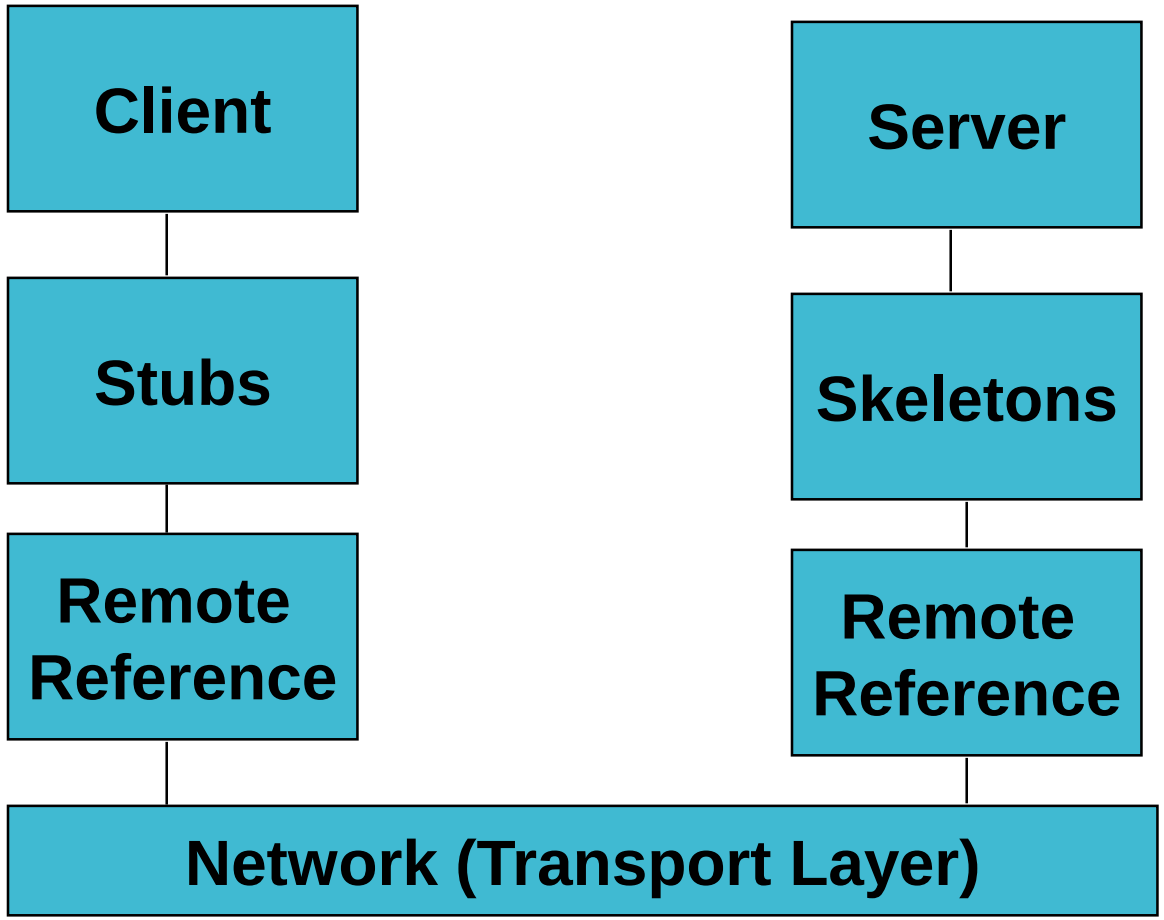


General idea of RMI w.r.t. Java Code

- Java RMI allowed programmer to execute remote function class using the same semantics as local functions calls.



RMI Architecture



How RMI works?

- Java makes RMI (Remote Method Invocation) *fairly* easy
- RMI is purely Java-specific
  - Java to Java communications only
- To send a message to a remote "server object,"
  - The "client object" has to *find* the object
    - Do this by looking it up in a registry
  - The client object then has to marshal the parameters (prepare them for transmission)
    - Java requires Serializable parameters
    - The server object has to unmarshal its parameters, do its computation, and marshal its response
  - The client object has to unmarshal the response

RMI Terminology

- Remote object ➔ An object on another computer
- Client object ➔ Object making the request (sending a message to the other object by a method call)
- Server object ➔ Object receiving the request
- As usual, "client" and "server" can easily trade roles (each can make requests of the other)
- The rmiregistry is a special server that looks up objects by name (which are unique)
  - Hopefully, the name is unique!
- rmic is a special compiler for creating stub (client) and skeleton (server) classes

RMI Terminology (cont.)

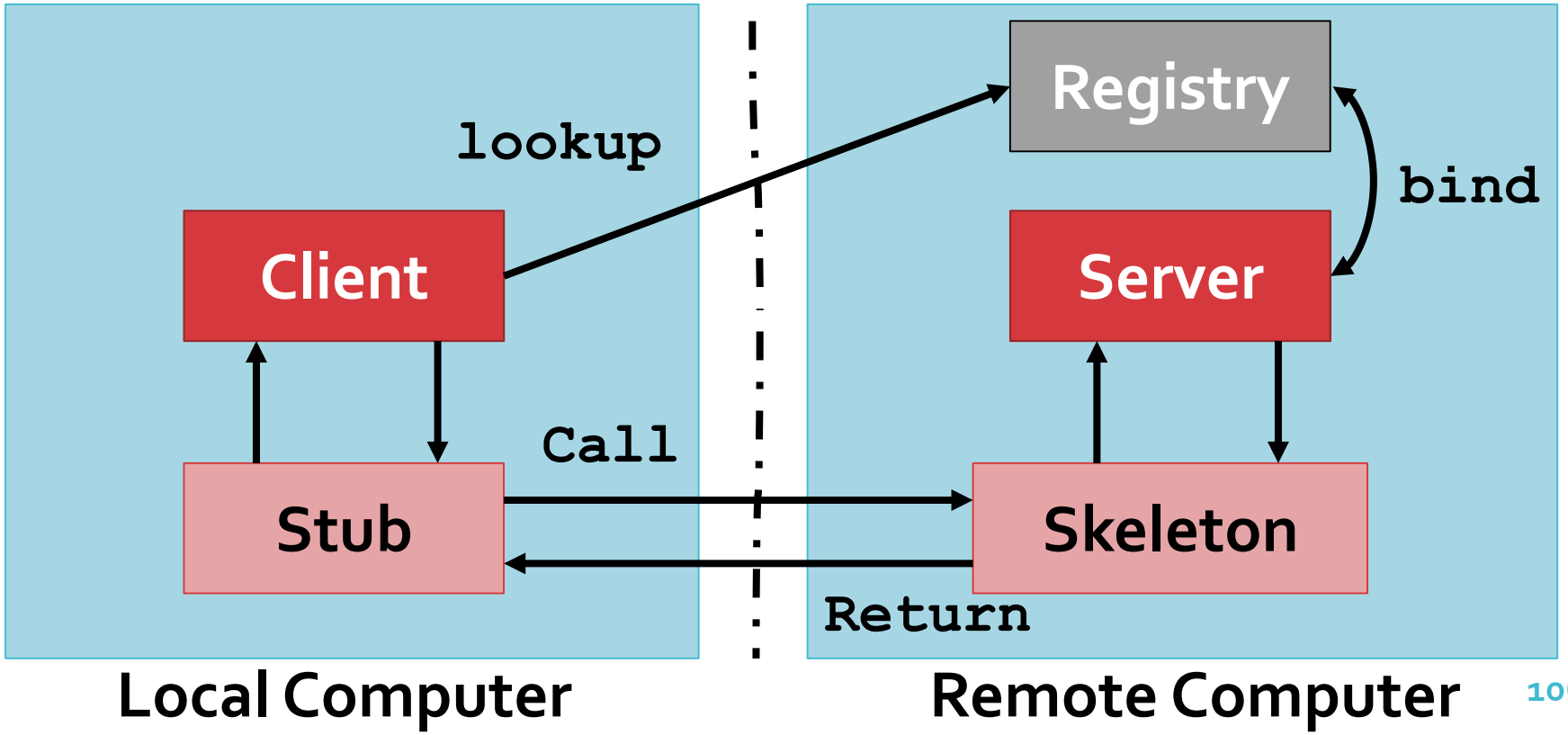
- Client - user interface
- Server - data source
- Stubs
  - marshals argument data (serialization)
  - unmarshals results data (deserialization)
- Skeletons (not reqd w/Java 2)
  - unmarshals argument data
  - marshals results data

RMI Processes

- For RMI, you need to be running *three* processes
  - The Client
  - The Server
  - The Object Registry, rmiregistry, which is like a DNS service for objects

Overall mechanism

- The server must first bind its name to the registry
- The client lookup the server name in the registry to establish remote references.
- The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.



Overall mechanism (cont.)

- A client invokes a remote method, the call is first forwarded to skeleton.
- The stub is responsible for sending the remote call over to the server-side skeleton
- The stub opening a socket to the remote server, marshaling the object parameters and forwarding the data stream to the skeleton.
- A skeleton contains a method that receives the remote calls, unmarshals the parameters, and invokes the actual remote object implementation.

The Steps

- Define the remote Interface to the server
- Create the Server (the remote object) by implementing the remote interface.
- Create the Client
- Compile the source files (javac) →
  - Interface, Server, Client
- Generate client Stubs and server Skeletons (rmic)
- Start the RMI registry
- Start the remote server objects
- Run the client

Interfaces

- Interfaces define behavior
- Classes define implementation

- So,
  - In order to use a remote object, the client must know its behavior (interface), but does not need to know its implementation (class)
  - In order to provide an object, the server must know both its interface (behavior) and its class (implementation)
- In short,
  - The interface must be available to both client and server
  - The class of any transmitted object must be on both client and server
  - The class whose method is being used should only be on the server

Classes

- A Remote class is one whose instances can be accessed remotely
  - On the computer where it is defined, instances of this class can be accessed just like any other object
  - On other computers, the remote object can be accessed via object handles
- A Serializable class is one whose instances can be marshaled (turned into a linear sequence of bits)
  - Serializable objects can be transmitted from one computer to another
- It probably isn't a good idea for an object to be both remote and serializable

Conditions to serialize an Object

- If an object is to be serialized:
  - The class must be declared as public
  - The class must implement Serializable
    - However, Serializable does not declare any methods
  - The class must have a no-argument constructor
  - All fields of the class must be serializable: either primitive types or Serializable objects
    - Exception: Fields marked transient will be ignored during serialization

Remote interfaces and classes

- A Remote class has two parts:
  - The interface (used by both client and server):
    - Must be public
    - Must extend the interface java.rmi.Remote
    - Every method in the interface must declare that it throws java.rmi.RemoteException (other exceptions may also be thrown)
  - The class itself (used only by the server):
    - Must implement the Remote interface
    - Should extend java.rmi.server.UnicastRemoteObject
    - May have locally accessible methods that are not in its Remote interface

Remote & Serializable

- A Remote object lives on another computer (such as the Server)
  - You can send messages to a Remote object and get responses back from the object
  - All you need to know about the Remote object is its interface
  - Remote objects don't pose much of a security issue
- You can transmit a *copy* of a Serializable object between computers
  - The receiving object needs to know how the object is implemented; it needs the class as well as the interface
  - There is a way to transmit the class definition
  - Accepting classes *does* pose a security issue

Server

- The class that defines the server object should extend UnicastRemoteObject
  - This makes a connection with exactly one other computer
  - If you must extend some other class, you can use exportObject() instead
  - Sun does *not* provide a MulticastRemoteObject class
- The server class needs to register its server object:
  - String url = "rmi://" + ***host*** + ":" + ***port*** + "/" + ***objectName***;
    - The default port is 1099
  - Naming.rebind(url, ***object***);
- Every remotely available method must throw a RemoteException (because connections can fail)
- Every remotely available method should be synchronized

RMI Example

Hello, World!

Hello world server: interface

```
import java.rmi.*;

public interface HelloInterface extends Remote

{
  public String say() throws RemoteException;
}
```

Hello world server: class

```
import java.rmi.*;
import java.rmi.server.*;

public class Hello extends UnicastRemoteObject
implements HelloInterface
{
  private String message;
  // Strings are serializable

  public Hello (String msg) throws
RemoteException
  {
     message = msg;
  }

  public String say() throws RemoteException
  {
     return message;
  }
}
```

Registering the hello world server

```
class HelloServer
{
public static void main (String[] argv) {
try
{
 Naming.rebind("rmi://localhost/HelloServer",
                    new Hello("Hello, world!"));
 System.out.println("Hello Server is ready.");
}
catch (Exception e)
{
 System.out.println("Hello Server failed:"+e);
}
}
}
```

Running the hello world client program

```
class HelloClient
{
public static void main (String[] args)
{
 HelloInterface hello;
 String name = "rmi://localhost/HelloServer";
 try
 {
  hello =
(HelloInterface)Naming.lookup(name);
  System.out.println(hello.say());
 }
 catch (Exception e) {
  System.out.println("Exception: "+e);
 }
}
}
```

The Steps

- Create the Interface to the server
- Create the Server
- Create the Client
- Compile the Interface (javac)
- Compile the Server (javac)
- Compile the Client (javac)
- Generate Stubs and Skeletons (rmic)

rmic command

- The class that implements the remote object should be compiled as usual
- Then, it should be compiled with rmic:
  - rmic Hello
- This will generate files Hello_Stub.class and Hello_Skel.class
- These classes do the actual communication
  - The "Stub" class must be *copied* to the client area
  - The "Skel" was needed in SDK 1.1 but is no longer necessary

Running RMI

- Run following command in three different terminal windows:
  1. Run the registry program:
     - rmiregistry
  2. Run the server program:
     - java HelloServer
  3. Run the client program:
     - java HelloClient

- If all goes well, it should o/p as the following message:
  - `"Hello, world!"`

RMI Example 2

Remote Server for calculating Sum

Step 1: Defining the Remote Interface

```
/* SampleServer.java */
import java.rmi.*;

public interface SumServer extends Remote
{
  public int sum(int a,int b)
          throws RemoteException;
}
```

Step 2: Develop the remote object and its interface

```
/* SampleServerImpl.java */
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;


public class SampleServerImpl extends
 UnicastRemoteObject implements SampleServer
{
SampleServerImpl() throws RemoteException
{      super();                }
//Implement the remote methods
public int sum(int a,int b) throws RemoteException
{      return a + b;           }
}
```

Step 2: Develop the remote object and its interface

```
/* SampleServerImpl.java */
public static void main(String args[]) {
 try  {
 //set the security manager
  System.setSecurityManager(new RMISecurityManager());

  //create a local instance of the object
  SampleServerImpl Server = new SampleServerImpl();

  //put the local instance in the registry
  Naming.rebind("SAMPLE-SERVER" , Server);

  System.out.println("Server waiting.....");
 }
 catch (java.net.MalformedURLException me) {
   System.out.println("Malformed URL: "+ me.toString());
 }
 catch (RemoteException re)  {
   System.out.println("Remote exception: "
                          + re.toString());  }
}
```

Step 3: Develop the client program

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
public class SampleClient
{
public static void main(String[]  args)
{
 // set the security manager for the client
 System.setSecurityManager(new RMISecurityManager());
 //get the remote object from the registry
 try {
  System.out.println("Security Manager loaded");
  String url = "//localhost/SAMPLE-SERVER";
  SampleServer remoteObject =
            (SampleServer)Naming.lookup(url);
  System.out.println("Got remote object");
  System.out.println("1 + 2 = "+remoteObject.sum(1,2) );
 }
```

Step 3: Develop the client program

```
catch (RemoteException e)
    {
     System.out.println("Lookup Error: "+ e.toString());
    }
catch (MalformedURLException e)
    {
     System.out.println("Malformed URL: "+e.toString());
    }
catch (NotBoundException e)
    {
     System.out.println("NotBound: " + e.toString());
    }
}
}
```

Step 4 & 5: Compile the Java source files & Generate the client stubs and server skeletons

```
C:\rmi> set CLASSPATH="."

C:\rmi> javac SampleServer.java

C:\rmi> javac SampleServerImpl.java

C:\rmi> rmic SampleServerImpl

C:\rmi> javac SampleClient.java
```

Step 6: Start the RMI registry

```
C:\rmi> start rmiregistry
```

Steps 7 & 8: Start the remote server objects & Run the client

```
C:\rmi> java SampleServerImpl

C:\rmi> java SampleClient
```

Steps' summary

1. Start the registry server, rmiregistry
2. Start the object server
    1. The object server registers an object, with a name, with the registry server
3. Start the client
    1. The client looks up the object in the registry server
4. The client makes a request
    1. The request actually goes to the Stub class
    2. The Stub classes on client and server talk to each other
    3. The client's Stub class returns the result