

---

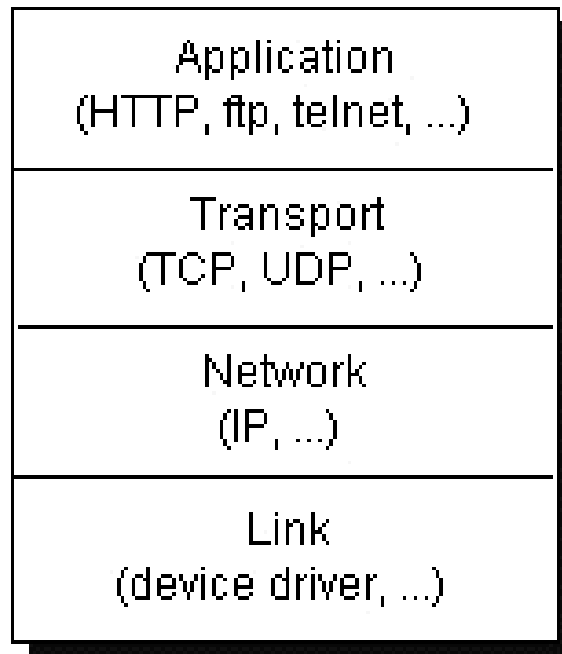
# JAVA NETWORKING

API



# Networking Basics

---



When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the `java.net` package. These classes provide system-independent network communication.



# TCP

---

**Definition:** *TCP* (*Transmission Control Protocol*) is a connection-based protocol that provides a reliable flow of data between two computers.



# TCP

---

TCP 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.



# TCP

---

The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

---



# UDP

---

**Definition:** *UDP (User Datagram Protocol)* is 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.



# Understanding Ports

---

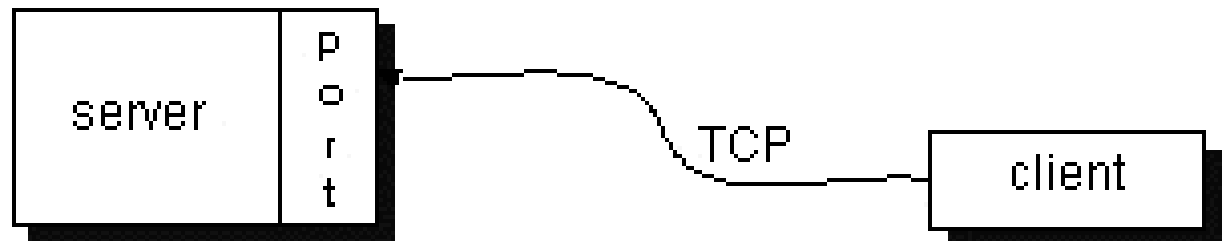
Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number,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



# Understanding Ports

---

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port, as illustrated here:

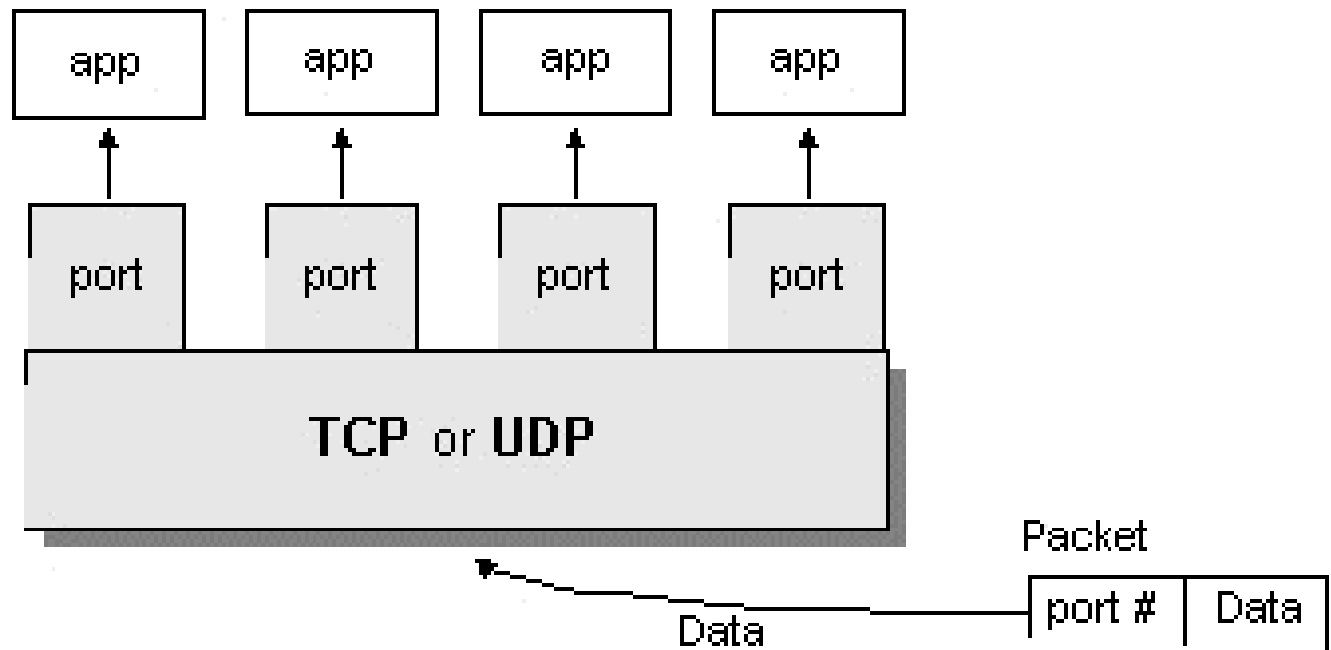




# Understanding Ports

---

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



# The InetAddress Class

---

The `java.net.InetAddress` class is Java's high-level representation of an IP address, both IPv4 and IPv6. It is used by most of the other networking classes, including `Socket`, `ServerSocket`, `URL`, `DatagramSocket`, `DatagramPacket`, and more. Usually, it includes both a hostname and an IP address.

---



# The InetAddress Class

---

If you know a numeric address, you can create an `InetAddress` object from that address without talking to DNS using `InetAddress.getByAddress\(\)`. This method can create addresses for hosts that do not exist or cannot be resolved:



# The InetAddress Class

---

```
public static InetAddress  
getByAddress(byte[] addr) throws  
UnknownHostException
```

```
public static InetAddress  
getByAddress(String hostname, byte[]  
addr) throws UnknownHostException
```



# What Is a Network Interface?

---

A *network interface* is the point of interconnection between a computer and a private or public network. A network interface is generally a network interface card (NIC), but does not have to have a physical form. Instead, the network interface can be implemented in software.



# What Is a Network Interface?

---

For example, the loopback interface (127.0.0.1 for IPv4 and ::1 for IPv6) is not a physical device but a piece of software simulating a network interface. The loopback interface is commonly used in test environments.



# The NetworkInterface Class

---

The NetworkInterface class represents a local IP address. This can either be a physical interface such as an additional Ethernet card (common on firewalls and routers) or it can be a virtual interface bound to the same physical hardware as the machine's other IP addresses.

---



# The NetworkInterface Class

---

The `NetworkInterface` class provides methods to enumerate all the local addresses, regardless of interface, and to create `InetAddress` objects from them. These `InetAddress` objects can then be used to create sockets, server sockets, and so forth.

---





# URL

---

URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.

A URL has two main components:

- Protocol identifier: For the URL `http://example.com`, the protocol identifier is `http`.
  - Resource name: For the URL `http://example.com`, the resource name is `example.com`.
- 



# Creating URL

---

*URL DMCS = new*

*URL("http://www.dmcs.p.lodz.pl/");*

Relative URL:

- *URL indexDMCS = new URL(DMCS, "/index.html");*

Other constructors:

- *URL("http", "www.dmcs.pl", "/index.html");*

- *URL("http", 80, "www.dmcs.pl", "/index.html");*



# MalformedURLException

---

Each of the four URL constructors throws a **MalformedURLException** if the arguments to the constructor refer to a null or unknown protocol.

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



# Reading from URL vs. URLConnection

---

- You can use either way to read from a URL.
- Reading from a URLConnection instead of reading directly from a URL might be more useful. This is because you can use the URLConnection object for other tasks (like writing to the URL) at the same time.



# Writing to URLConnection

---

To write to an URL you have to do following steps:

1. Create a URL.
2. Open a connection to the URL.
3. Set output capability on the URLConnection.
4. Get an output stream from the connection.
5. Write to the output stream.
6. Close the output stream.



# Reading from URL

---

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

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

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

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

```
    BufferedReader in = new BufferedReader( new InputStreamReader(  
yahoo.openStream()));
```

```
    String inputLine;
```

```
    while ((inputLine = in.readLine()) != null)
```

```
        System.out.println(inputLine); //HTML from yahoo
```

```
    in.close();
```

```
}}
```

---



# Connecting to a URL

---

When you connect to a URL, you are initializing a communication link between your Java program and the URL over the network.

```
try {  
    URL yahoo = new URL("http://www.yahoo.com/");  
    URLConnection yahooConnection = yahoo.openConnection();  
} catch (MalformedURLException e) {  
    // new URL() failed . . .  
}  
} catch (IOException e) {  
    // openConnection() failed . . .  
}  
}
```



# Reading from URLConnection

---

```
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();

}}


```

---





# Writing to a URLConnection

---

...

```
String stringToReverse = URLEncoder.encode(args[0]);  
URL url = new URL("http://java.sun.com/cgi-bin/backwards");  
URLConnection connection = url.openConnection();  
connection.setDoOutput(true);  
PrintWriter out = new PrintWriter( connection.getOutputStream());  
out.println("string=" + stringToReverse);  
//backward script waits for data in the above form  
out.close();
```

... 

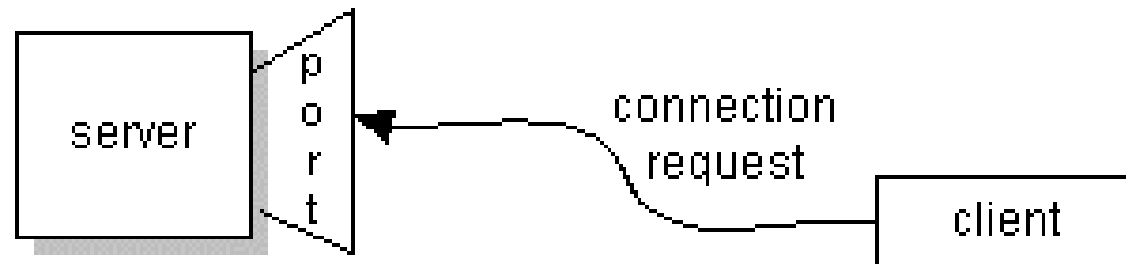
---

# Sockets

A **socket** is one endpoint of a two-way communication link between two programs running on the network.

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.

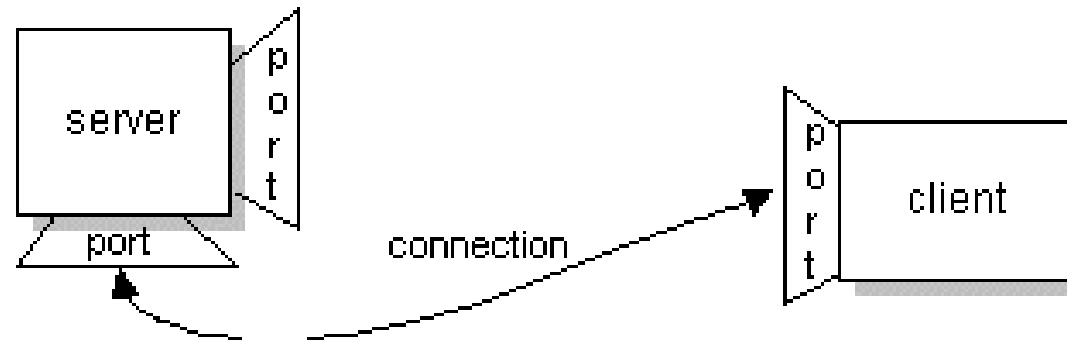


# Sockets(2)

If everything goes well, the server accepts the connection.

Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (and consequently a different port number) so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.



# Reading from and writing to a Socket

---

**To do is you should perform five basic steps:**

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.



# Reading from and writing to a Socket - example

---

```
import java.io.*;  
import java.net.*; //Socket  
public class EchoClient {  
public static void main(String[] args) throws IOException {  
    Socket echoSocket = null;  
    PrintWriter out = null;  
    BufferedReader in = null;
```


**Verte→**



# Reading ... (2)

```
try {  
    echoSocket = new Socket("comp", 7); //Echo Server port  
    out = new PrintWriter(echoSocket.getOutputStream(), true);  
    in = new BufferedReader(new InputStreamReader(  
        echoSocket.getInputStream()));  
} catch (UnknownHostException e) {  
    System.err.println("Don't know about host: comp."); System.exit(1);  
} catch (IOException e) {  
    System.err.println("Couldn't get I/O for " + "the connection to: comp.");  
    System.exit(1);    } Verte→
```

---



# Reading ... (3)

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

---

```
String userInput;
```

```
while ((userInput = stdIn.readLine()) != null) {           out.println(userInput);  
    System.out.println("echo: " + in.readLine());  
}
```

```
out.close(); in.close();
```

```
stdIn.close();      echoSocket.close();
```

```
//The precedence is important. You should first close all streams
```

```
// connected to the socket and than socket itself.
```

```
}}
```



# Reading from URL - socket version

---

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

```
String hostname = "http://www.yahoo.com";
```

```
int port = 80;
```

```
String filename = "/index.html";
```

```
Socket s = new Socket(hostname, port);
```

```
InputStream sin = s.getInputStream();
```

```
BufferedReader fromServer = new BufferedReader(new  
InputStreamReader(sin));
```

```
OutputStream sout = s.getOutputStream();
```

```
PrintWriter toServer = new PrintWriter(new OutputStreamWriter(sout));
```

```
verte-->
```

---





# Reading from URL - socket version(2)

---

```
toServer.print("GET " + filename + " HTTP/1.0\n\n");  
toServer.flush();
```

```
for(String l = null; (l = fromServer.readLine()) != null; )
```

```
System.out.println(l);
```

```
toServer.close();
```

```
fromServer.close();
```

```
s.close();
```



# Creating a server socket

---

```
try {  
    serverSocket = new ServerSocket(4444);  
} catch (IOException e) {  
    System.out.println("Could not listen on port: 4444");  
    System.exit(-1);  
}
```



# Sever side of sockets

---

If the server successfully connects to its port, then the `ServerSocket` object is successfully created and the server continues to the next step - accepting a connection from a client:

```
Socket clientSocket = null;  
  
try {  
    clientSocket = serverSocket.accept();  
  
} catch (IOException e) {  
    // accept failed  
  
}
```



# Sever side of sockets

---

- The accept method waits until a client starts up and requests a connection on the host and port of this server. When a connection is requested and successfully established, the accept method returns a new Socket object which is bound to a new port.



# Sever side of sockets

---

After the server successfully establishes a connection with a client, it communicates with the client:

```
PrintWriter out = new PrintWriter(  
clientSocket.getOutputStream(), true);
```

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



# Supporting multiple clients

---

Server can service clients simultaneously through the use of threads - one thread per each client connection. The basic flow of logic in such a server is this:

```
while (true) {  
  
    accept a connection ;  
  
    create a thread to deal with the client ;  
  
}
```

---



# 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 classes to help you write Java programs that use datagrams to send and receive packets over the network:

- **DatagramSocket**
  - **DatagramPacket**
- 



# Sending Datagrams

---

...

```
byte[] buf = new byte[256];
```

```
InetAddress address = InetAddress.getByName(„http://...”);
```

```
DatagramPacket packet = new DatagramPacket(buf, buf.length,  
    address, 4445);
```

```
socket.send(packet);
```

...

---





# Receiving and sending Datagram

---

...

```
Datagram Socket socket = new DatagramSocket(4445);  
byte[] buf = new byte[256];  
DatagramPacket packet = new DatagramPacket(buf, buf.length);  
socket.receive(packet);  
InetAddress address = packet.getAddress();  
int port = packet.getPort();  
packet = new DatagramPacket(buf, buf.length, address, port);  
socket.send(packet);
```

---

... 