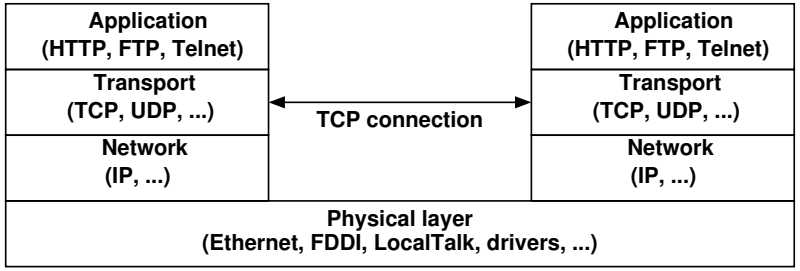
# Fundamentals of Internet Connections
# Objectives

- To understand programming of clients that connect to servers via TCP
- To understand the basics of programming of servers that accept TCP connections
- To practice programming of application-level internet connections (HTTP)
- Knowledge from this lecture will be needed at a lab but not necessarily at the project
  - But it is important also for practicing Java code writing and Java documentation lookup





# TCP connections




- TCP achieves a circuit-based connection (like a phone call) over the packet-based IP network
- A client connects to a server on another machine. That server ”listens to” a specific port on that machine.





# `java.net.Socket`

- A socket is an ”endpoint for communication”.
  - It represents a TCP connection
  - but one can build sockets over any transport protocol
- To create a Socket you need to know what machine you want to connect to, and to what port
  `Socket(String machineName, int port)`
  - throws `java.net.UnknownHostException` if host not found
  - throws `java.io.IOException` if you can't connect (e.g. when there's no server listening on that port!)
- Once you built your socket, you may access two streams
  - one stream to talk to the server: `OutputStream getOutputStream()`
  - one stream to listen to the server's response: `InputStream getInputStream()`
- When you are done with the socket you call `close()` on the streams and the socket (in that order) to free network resources on the machine





# Connect to an Echo port

This program sends some text to an echo port (7) and prints the response on the screen.
Start the program with `java TextToEcho www.nada.kth.se "some message"`

```
import java.io.*;
import java.net.*;
public class TextToEcho {
  public static void main(String argv[])
  throws IOException, UnknownHostException {
    Socket conn = new Socket(argv[0], 7);
    PrintWriter talk =
       new PrintWriter(new OutputStreamWriter(conn.getOutputStream()));
    Reader listen = new InputStreamReader(conn.getInputStream());
    talk.print(argv[1]);
    talk.flush(); /* we make sure that the chars reach the server */
    char buffer[] = new char[80];
    int n = listen.read(buffer, 0, 80);
    System.out.println("The server said: " + new String(buffer, 0, n));
    talk.close(); listen.close(); conn.close();
  }
}
```

# Some comments on socket programs

- Once we master streams, creating socket programs is easy.
  - A socket is just a pair of streams
- If you start reading from the server before you are sure that the server will respond, your program might wait forever!
  - Forgetting to flush will sometimes cause the server not to receive anything (so there's no way it is going to respond)
  - To see where your program is stuck: put println statements just before every place where you suspect that the program hangs.
  - To kill the program:
    - Use Ctrl-Break on Windows
    - On Unix(-like): `ctrl-Z, bg, ps, kill QUIT javaProcess`
- Often reading from the server and writing to the server takes place in two separate threads so reading can wait without blocking
- What makes you know when the server will respond? The protocol!
- The echo server sends back immediately whatever it got
- In other protocols (e.g. HTTP) the server waits until it has at least one line to send before it starts responding





# HTTP request

Here is a program that makes a HTTP request and prints the result.

```
java HttpAccess www.nada.kth.se index.html | more
```

```
import java.io.*; import java.net.*;
public class HttpAccess {
  public static void main(String argv[])
  throws IOException, UnknownHostException {
    Socket conn = new Socket(argv[0], 80);
    PrintWriter talk =
      new PrintWriter(new OutputStreamWriter(conn.getOutputStream()));
    talk.println("GET /" + argv[1] + " HTTP/1.0"); /* HTTP command */
    talk.println("Host: " + argv[0]); /* HTTP header(s) */
    talk.println(); /* empty line, no content after */
    talk.flush(); /* PrintWriter println() is not autoflush! */
    BufferedReader listen =
      new BufferedReader(new InputStreamReader(conn.getInputStream()));
    String line;
    while ( (line = listen.readLine()) != null)
       System.out.println(line);
    talk.close(); listen.close(); conn.close();
  }
}
```





# Client-Server paradigm

- Most of the Internet today is built around the client-server approach
- A server is a program that offers a service
  (echoing characters, serving web pages, tell the time, . . . )
- A client is a program that asks a server for a certain service
- The server waits for some client to connect
- The client initiates a conversation that respects a set of rules called *protocol*
- Alternatives to client-server: remote procedure call (RPC), Web Services





# `java.net.ServerSocket`

A `java.net.ServerSocket`

- binds a port of the local machine
  - `ServerSocket(int port)`
  - throws `java.io.IOException` if the socket can't be open (e.g. when the port is not free!)
  - throws `java.lang.SecurityException` if our code is not allowed to bind a TCP port.
    - e.g. if a Java applet downloaded from the Internet
    - No need to catch this as it's a RuntimeException
- accepts connections from clients
  - Socket `accept()` returns the other endpoint of the connection
  - Communication with the client takes place on the input stream and output stream of the returned Socket
  - `accept()` blocks until a request from a client comes!
- `close()` your server socket when you are done.
  - Rarely happens. Servers usually call `accept()` in an endless loop

# A simple echo server

This program starts an echo server. Start with: `java SimpleEchoServer port`

```
import java.io.*; import java.net.*;
public class SimpleEchoServer {
  public static void main(String argv[])
  throws IOException {
    ServerSocket server =
      new ServerSocket (Integer.valueOf(argv[0]).intValue());
    while (true) {
      Socket conn = server.accept();
      System.out.println(
         new java.util.Date() + " " + conn.getInetAddress());
      InputStream in = conn.getInputStream();
      OutputStream out= conn.getOutputStream();
      byte[] buffer= new byte[8192];
      while((int n= in.read(buffer, 0, 8192))!=-1)
        out.write(buffer, 0, n);
      in.close(); out.close(); conn.close();
    }
  }
}
```





# A simple echo server . . .

On Windows:
Start server: `start java SimpleEchoServer 7`
Send message: `java TextEcho localhost "blah blah blah"`

On Unix you need to add a non-default port as an echo port, e.g. 7777, then
Start server: `java SimpleEchoServer 7777 &`
Send message: `java TextEcho localhost 7777 "blah blah blah"`





# Multithreaded servers

If a client connects to our echo server and sends crap for 20 minutes, all other clients will have to wait 20 minutes before they can get a response.
To avoid that, most servers process requests concurrently in separate processes, or lightweight processes (threads)
Creating a thread for each request is very expensive, But we'll do that for the sake of exemplification.

Advanced servers use thread pools.
Instead of being destroyed at the end of the client request, the thread is put "on hold" in a pool, and reused when another request comes





# Multithreaded Echo Server

```
import java.io.*;
import java.net.*;
public class EchoServer implements Runnable {
  Socket conn; /* member variable, for run() */
  public EchoServer(Socket s){ conn = s; }
  public static void main(String argv[]) throws IOException {
    /* run is a member method, must create object to call it */
    ServerSocket server =
      new ServerSocket(Integer.valueOf(argv[0]).intValue());
    while(true) new Thread(new EchoServer(server.accept())).start();
  }
  public void run() {
    try { /* run does not throw any exceptions */
       InputStream in = conn.getInputStream();
       OutputStream out = conn.getOutputStream();
       byte[] buffer = new byte[8192];
       int n;
       while((n = in.read(buffer, 0, 8192))!=-1)
       out.write(buffer, 0, n);
       in.close(); out.close(); conn.close();
    }
    catch(IOException e) { e.printStackTrace(); } } }
```

# A simple HTTP server

This program starts a HTTP server which sends back whatever is sent to it.
Start with: `java SimpleHttpServer port`.
Access with a browser: `http://localhost:port/someFile?someParam=someValue`

```
import java.io.*; import java.net.*;
public class SimpleHttpServer {
  public static void main(String argv[]) throws IOException {
    ServerSocket server =
      new ServerSocket(Integer.valueOf(argv[0]).intValue());
    while(true) {
      Socket con = server.accept();
      BufferedReader listen =
       new BufferedReader(new InputStreamReader(con.getInputStream()));
      PrintWriter talk =
       new PrintWriter(new OutputStreamWriter(con.getOutputStream()));
      talk.println("HTTP/1.1 200"); /* HTTP command */
      talk.println("Content-type: text/plain"); /* HTTP header(s) */
      talk.println(); /* empty line, as requested */
      String line; /* HTTP response content starts: */
      while((line=listen.readLine()).length()>0)
      talk.println(line);
      talk.close(); listen.close(); con.close(); } } }
```

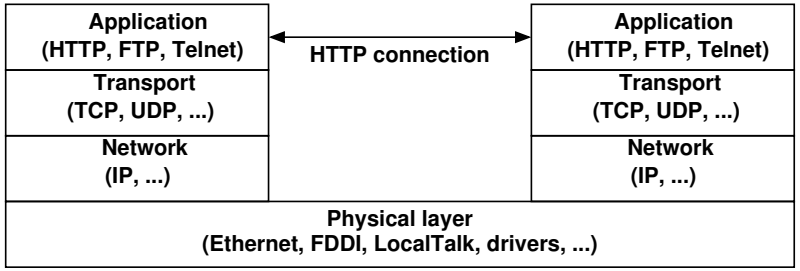




# Application-level connections

We've made a HTTP connection by hand, making a TCP socket and "speaking" the HTTP protocol over the socket.

Instead, we can use a ready-made application level class that does the dirty protocol job for us.




At the server side, instead of speaking the HTTP protocol, we can use a ready-made and optimized server (like Apache) and simply program how to treat a request (e.g. via CGI, next time)





# `java.net.URLConnection`

- Communicates to a URL, over a number of protocols
- You cannot construct a `URLConnection` directly.
  - You need a `java.net.URL` first, then call `openConnection()`
- Before you connect, you can configure how the URLConnection will work
  - `setDoInput(boolean)`, `setDoOutput(boolean)` `output` is `false` by default
  - `setUseCaches(boolean)` can force a "reload" if `false`
  - `setRequestProperty(String name, String value)` can set e.g. a HTTP header
- `connect()` does the actual opening of a TCP connection (socket) and information exchange
- If there is more information to send and you have called `setDoOutput(true)`, you can call `getOutputStream()` to send it
  - This can be used for e.g. a HTTP POST request
- After that, you can get various details about the response (headers)
  - `getContentType()`, `getContentLength()`, `getContentEncoding()`, `getDate()`
  - `getHeaderField(String name)` provides info on any response header
  - `getContent()` or `getInputStream()` give you the response itself (i.e. the file you requested in the case of a HTTP connection)
    Remember to close any stream that you request from the connection





# HTTP access with URLConnection

This program makes a HTTP request and prints data about the result, and the result itself.
`java URLAccess http://www.nada.kth.se/index.html`

```
import java.io.*; import java.net.*;
public class URLAccess {
  public static void main(String argv[]) throws IOException {
    URLConnection con = new URL(argv[0]).openConnection();
    con.connect();
    System.out.println("type: " + con.getContentType());
    System.out.println("length: " + con.getContentLength());
    if(con instanceof HttpURLConnection)
      System.out.println("method: "
          +((HttpURLConnection)con).getRequestMethod());
    BufferedReader content =
      new BufferedReader(new InputStreamReader(con.getInputStream()));
    String line;
    while((line = content.readLine()) != null)
      System.out.println(line);
    content.close();
  }
}
```

# Some more on `URLConnection`

Other protocols than HTTP are supported

Actually `URL.openConnection()` never returns a `URLConnection` object

It allways return an object from one of it's subclasses

Most notably `HttpURLConnection`