java.net

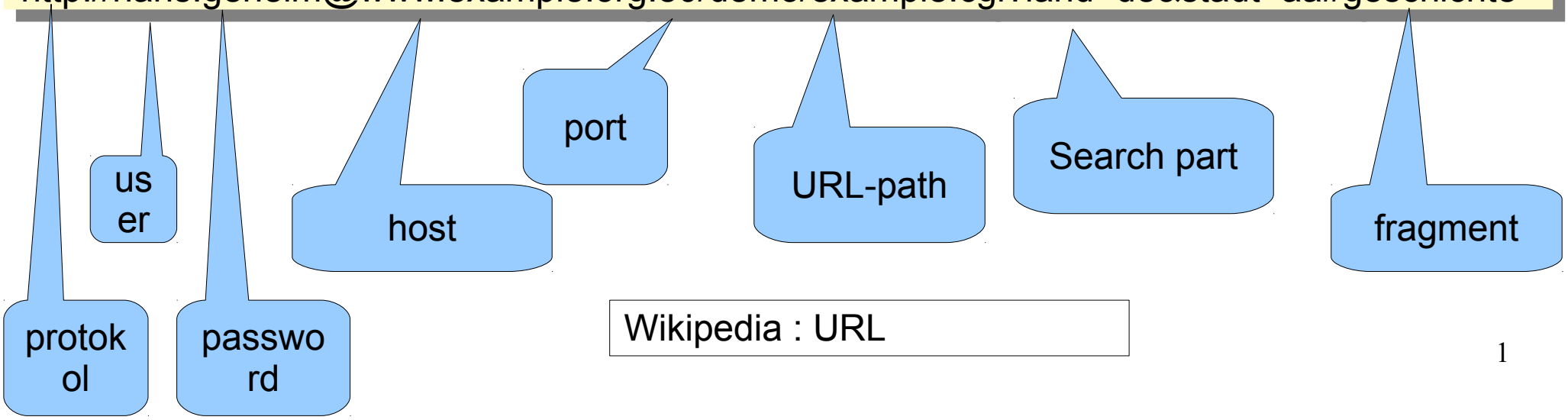
Klassenbibliothek für den Datentransport im Netz

Entfernte Ressourcen werden über URLs oder URIs identifiziert.

Es werden verbindungorientierte und verbindungslose Datentransfers unterstützt.

Die primitivste Art, Daten zu laden, arbeitet mit der Klasse URL

http://hans:geheim@www.example.org:80/demo/example.cgi?land=de&stadt=aa#geschichte



Wikipedia : URL

Übertragungsarten

Man unterscheidet zwischen verbindungsloser und verbindungsorientierter Datenübertragung

Verbindunsgorientierte Datenübertragung mit den Klassen:

URL

URLConnection

Socket, ServerSocket

Verbindungslose Datenübertragung mit den Klassen:

Datagramm, DatagrammSocket

Die Klasse URL

(verbindungsorientiert)

Methoden zur URL

- getProtocol() http, ftp, nfs, ldap, info …
- getHost()
- getPort()
- getFile()
- getQuery()

Methoden zur Datenübertragung

- **openStream()**
- getContent()
- **openConnection()**

```
import java.net.*;
import java.io.*;
public class UrlLoad
{
  public static void main(String args[])
  {
    if (args.length!=2)
      {System.err.println("Usage:..."); System.exit(1);}
    try
    {
      URL        url = new URL(args[0]);
      OutputStream o = new FileOutputStream(args[1]);
      InputStream  i = url.openStream();
      int len;
      byte buf[]=new byte[128];
      while ((len=i.read(buf))!=-1) o.write(buf,0,len);
      o.close();
      i.close();
    }
    catch (MalformedURLException e)
    { System.out.println(e); System.exit(1);}
    catch (IOException e)
    { System.out.println(e); System.exit(1);}
  }
}
```

URLConnection

(verbindungsorientiert)

Sehr viel komfortabler

Über get methods lassen sich viele Eigenschaften der entfernten Ressource erfragen

getLength()

getHeaderFields()

Nach connect() ist der Datentransfer möglich

InputStream getInputStream()

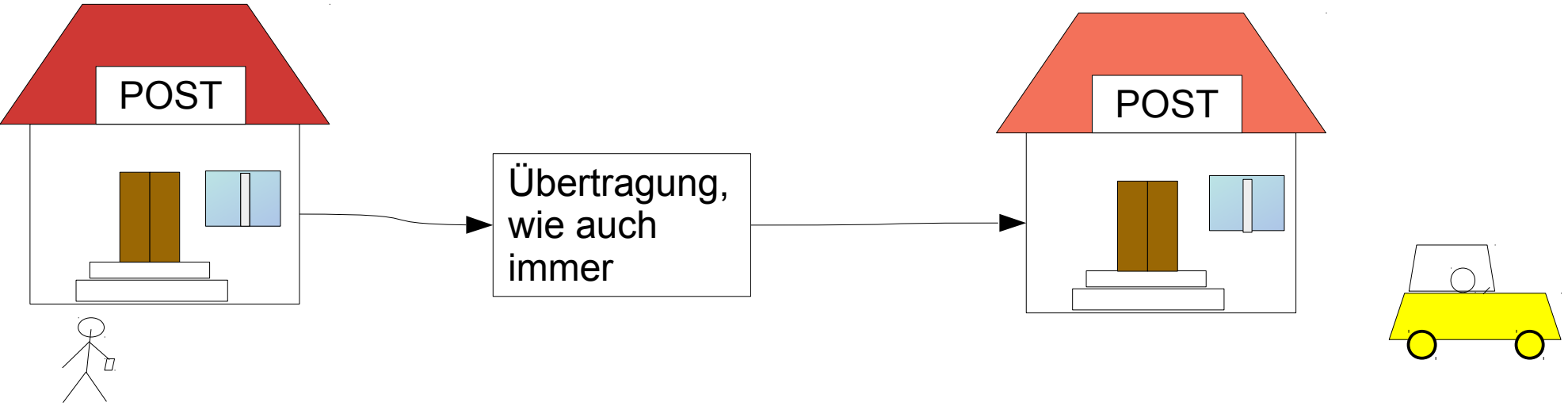
OutputStream getOutputStream()

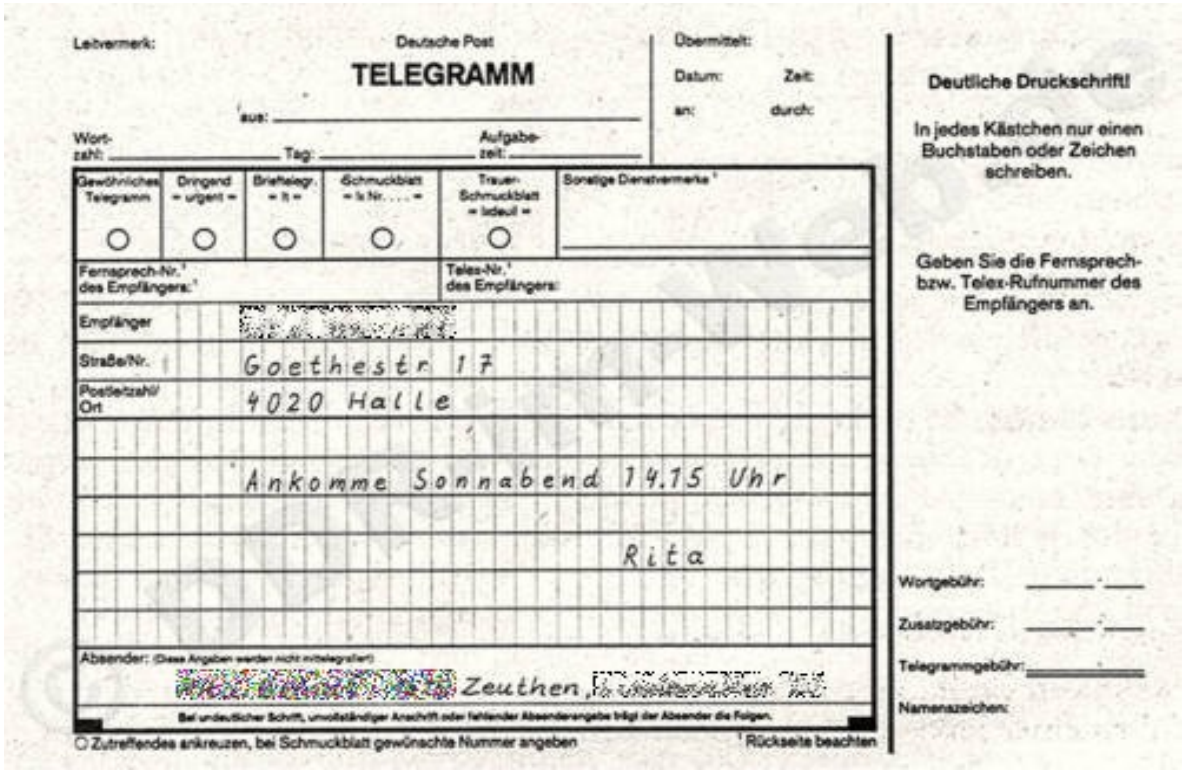
```
import java.net.*;
import java.io.*;

public class UrlCLoad
{
 public static void main(String args[])
 {
  if (args.length!=2) {System.err.println("Usage:...");
                       System.exit(1);}
  try
  {
   URL          url = new URL(args[0]);
   URLConnection c = url.openConnection();
   System.out.println("Length   :"+c.getContentLength());
   System.out.println("Type     :"+c.getContentType());
   System.out.println("Encoding :"+c.getContentEncoding());
   System.out.println("header   :"+c.getHeaderFields());
```

```
    c.connect();
    OutputStream  O = new FileOutputStream(args[1]);
    InputStream   I = c.getInputStream();
    int len;
    byte buf[]=new byte[128];
    while ((len=I.read(buf))!=-1) O.write(buf,0,len);
    O.close();
    I.close();
   }
   catch (Exception e)
   { System.out.println(e); System.exit(1);}
  }
}
```

Datagramme

(verbindungslos)





Leitvermerk:

Deutsche Post

TELEGRAMM

Übermittelt:

Datum: Zeit:

an: durch:

aus:

Wort-zahl: Tag: Aufgabe-zeit:

Gewöhnliches Telegramm	Dringend = urgent =	Brieftelegr. = lt =	Schmuckblatt = lt Nr. ... =	Trauer-Schmuckblatt = lideuil =	Sonstige Dienstvermerke
○	○	○	○	○	

Fernsprech-Nr. des Empfängers:

Telex-Nr. des Empfängers:

Empfänger

Straße/Nr. Goethestr 17

Postleitzahl/ Ort 4020 Halle

Ankomme Sonnabend 14.15 Uhr

Rita

Absender: (Diese Angaben werden nicht mittelegrafiert)

Zeuthen

Bei undeutlicher Schrift, unvollständiger Anschrift oder fehlender Absenderangabe trägt der Absender die Folgen.

○ Zutreffendes ankreuzen, bei Schmuckblatt gewünschte Nummer angeben

¹ Rückseite beachten

Deutliche Druckschrift!

In jedes Kästchen nur einen Buchstaben oder Zeichen schreiben.

Geben Sie die Fernsprech- bzw. Telex-Rufnummer des Empfängers an.

Wortgebühr:

Zusatzgebühr:

Telegrammgebühr:

Namenszeichen:

An:.....Zeuten .....

Ankomme Sonnabend 14.15 Uhr Rita

Datagramme

- Verbindungslose Datenübertragung
- Zwei Klassen werden benötigt
  - DatagramSocket (entspricht dem Postamt)
  - DatagramPacket (entspricht dem Formular)
- Das Datagram enthält die Adressinformationen des Empfängers, des Absenders und Daten in Form eines ByteArrays
- Werden mehrere Datagramme an einen Empfänger gesandt, so ist die Reihenfolge des Empfangs undefiniert, da jedes Datagramm seinen eigenen Weg durch das (weltweite) Netzwerk finden muss.

- Es ist nicht sichergestellt, dass ein Datagramm seinen Empfänger wirklich erreicht.
- Reihenfolge des Sendens entspricht nicht unbedingt der Reihenfolge des Empfangens
- Datagramme übertragen eine kurze Informationen von maximal 65507 Bytes.

Datagramm versenden

```
String myHost="MeinHost.informatik,htw-dresden.de"
byte[] mess=args[0].getBytes();

// Liefert Internetadresse
InetAddress addr=InetAddress.getByName(myHost);

// Das Datenpaket
DatagramPacket packs= new DatagramPacket
                    (mess,mess.length, addr, myPort);

// Der SendeSocket
DatagramSocket ds = new DatagramSocket();

// Paket absenden
ds.send(packs);
```

Datagram empfangen

```
// leeres Datagramm bereitstellen

DatagramPacket packr=
                  new DatagramPacket(new byte[1024],1024);
DatagramSocket ds=new DatagramSocket(12345);

// Warten auf Empfang, Socket von voriger Folie weiter
// verwendet oder neuen DatagramSocket anlegen
ds.receive(packr);

// Paket auspacken
messr=new String(packr.getData(),0,packr.getLength());

System.out.println(messr);
ds.close();
```

Portnummer!

ByteArray von oben,
Nur in der Länge
getLength()
auswerten!!

Client/Server

Server:
- DatagramSocket mit Portnummer
- Leeres Datagram zum Empfangen
- Auf Empfang warten
- Antwort in empfangenes Datagram einsetzen
- Senden

Client:
- DatagramSocket mit/ohne Portnummer
- Datagram mit Empfängerinformation
  - Host (InetAddress)
  - Port
  - Message
  - Länge
- Senden
- Mit dem selben DatagramSocket auf Antwort warten

Sockets

(verbindungsorientiert)

- Grundsätzliches Konzept zur Datenübertragung
- Bedeutet soviel wie Sockel/Steckdose
- Universelle Schnittstelle zur Kommunikation zwischen Hosts
- Adressinformationen enthalten den Host und eine Schnittstellennummer, die Portnummer.
- Socket(String host, int port);
- serverseitiges „lauschen“ wird mit der Klasse ServerSocket realisiert, für den eigentlichen Datentransport gibt es die Klasse Socket.

Sockets

- Eine aus Sockets basierende Anwendung besteht aus
  - einem Server, der Daten bereitstellt und versendet
  - Einem Client, der Daten anfordert, auf die Daten wartet und diese anzeigt, verarbeitet, ...
- Als Client nutzen wir für unsere Experimente zunächst einen WebBrowser
- Um einen Server zu bauen, werden die Klassen Socket und ServerSocket benötigt.

Verwendung von ServerSocket

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

public class TestServ1
{
  public static void main(String argv[])throws IOException
  {
    ServerSocket ss =
            new ServerSocket(Integer.parseInt(argv[0]));

    while (true) new TinyServConn(ss.accept());

  }
}
```

ServerSocket

- Der ServerSocket „lauscht“ auf dem angegebenen Port auf eingehende Requests.
- Er macht das mit der Methode accept. accept blockiert so lange, bis ein Request eingeht.
- Ist das der Fall, liefert sie ein Socketobjekt, über das der weitere Datenverkehr bidirektional läuft.
- In unserem Beispiel wird das über ein Objekt der Klasse TinyServConn abgewickelt.
- Es arbeitet als Thread, somit kehrt ServerSocket sofort zurück zu accept und kann weitere Requests entgegen nehmen.

```
class TinyServConn extends Thread
{
   Socket sock;
   TinyServConn(Socket s)
   {
     sock=s;
     start();
   }
   public void run()
   {
    try
    {
     String Command=null;
     String req,buf;
     System.out.println("Socket.localPort:"+sock.getLocalPort()
                        +" Socket.port:"+sock.getPort());
     BufferedReader BR = new BufferedReader
              (new InputStreamReader(sock.getInputStream()));
     while((buf=BR.readLine())!=null)
              System.out.println(buf);
     sock.close();
    }catch(IOException e){System.out.println("I/O Error "+e);}
   }
}
```

- Bei jedem eingehenden Request wird ein neuer Thread erzeugt, der diesen Request bedient.
- Es wird ein Kommunikationskanal zw. einem lokalen, vom System vergebenen Port und dem entfernten Port, der die Verbindung angefordert hat, aufgebaut.
- Ein Browser soll zunächst als Client dienen.
- Der Server gibt bei eingehendem Request alle Informationen, die der Request enthält, auf die Konsole aus.
- Der Client erhält keine Antwort, und reagiert ggf. irgendwann mit timeout

Erzielte Ausgabe:

```
> java TestServ1 12345
Socket.localPort:12345 Socket.port:45141
GET /spass HTTP/1.1
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.5; Linux) KHTML/3.5.9 (like Gecko) SUSE
Accept: text/html, image/jpeg, image/png, text/*, image/*, */*
Accept-Encoding: x-gzip, x-deflate, gzip, deflate
Accept-Charset: utf-8, utf-8;q=0.5, *;q=0.5
Accept-Language: de, en
Host: localhost:12345
Connection: Keep-Alive
```

```
Socket.localPort:12345 Socket.port:34241
GET /ScheespassAufDerStrasse HTTP/1.0
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.5; Linux) KHTML/3.5.10 (like Gecko) SUSE
Accept: text/html, image/jpeg, image/png, text/*, image/*, */*
Accept-Encoding: x-gzip, x-deflate, gzip, deflate
Accept-Charset: utf-8, utf-8;q=0.5, *;q=0.5
Accept-Language: de, en
Host: 141.56.132.145:12345
Via: 1.1 rlux24.rz.htw-dresden.de:3128 (squid/2.7.STABLE5)
X-Forwarded-For: 141.56.132.76
Cache-Control: max-age=259200
Connection: keep-alive
```

Daten sollen fließen

- Nachfolgen wird TinyServConn so mdifiziert, dass der Name der angefoderten Ressource (Datei) aus der Zeile

  GET <name der datei> HTTP/1.1

  extrahiert wird.
- Sodann wird die Datei geöffnet.
- Mit Hilfe des Sockets wird ein OutputStream erzeugt (`OutputStream out=sock.getOutputStream();`) über den die aus der göffneten Datei gelesenen Daten zum Client gesandt werden(`while ((len=fis.read(buf))!=-1) out.write(buf,0,len);`).

Daten sollen fließen

```
public void run()
{
    try
    {
     String Command=null;
     String req="",S;
     byte buf[]=new byte[128];
     BufferedReader BR=
      new BufferedReader(new InputStreamReader(sock.getInputStream()));

     // Finde die Zeile, die mit GET beginnt !
     while(!(S=BR.readLine()).startsWith("GET"));req=S;

     // gefunden!!
     System.out.println("Req:"+req);

      // Zeile mit GET zerlegen
     StringTokenizer st=new StringTokenizer(req);
```

```
    if ((st.countTokens()>=2)
    &&  (Command=st.nextToken()).equals("GET"))
    {
     OutputStream out=sock.getOutputStream();
     if ((req =st.nextToken()).startsWith("/")) req=req.substring(1);

     // req enthaelt Dateiname der zu sendenden Daten
     System.out.println("File to send: "+req); // zur Demo
     File F=new File(req);
     FileInputStream fis=new FileInputStream(req);

     int len;
     while ((len=fis.read(buf))!=-1) out.write(buf,0,len);

     sock.shutdownOutput(); // wichtig!!
    }
    System.out.println("ready");
    sock.close();
   }catch(IOException e) {System.out.println("I/O Error "+e);}
  }
}
```

Security

- Mit diesem Server kann jeder von überall auf alles zugreifen, wofür der Server Leserechte hat.
- Wirkungsvolle Abhilfe schafft der SecurityManager
- Es kann in einem JavaProzess immer nur ein SecurityManager laufen.
- Das Aktiviren eines SecurityManagers ist im Leben eines JavaProzesses nur einmalig möglich.
- Die Klasse SecurityManager stellt check...()-methoden zur Verfügung, die eine SecurityException werfen.
- Durch Überschreiben einzelner check-methoden werden gezielt Aktivitäten erlaubt.

Mit Security manager

```
public static void main(String argv[]) throws IOException
{
  Properties P=System.getProperties();
  final String LibPath=P.getProperty("sun.boot.library.path");
  System.setSecurityManager(new SecurityManager()
   {
     public void checkAccept (String host, int port)  {}
     public void checkPermission      (Permission p){}
     public void checkRead(String file)
     {
           . . .
     }
    }
  );
  ServerSocket ss = new ServerSocket(Integer.parseInt(argv[0]));
  while (true) new TinyServConn(ss.accept());
}
```

Die Methode checkRead()

```
public void checkRead(String file)
{

        if(file.startsWith("/") | file.contains(".."))
    throw (new SecurityException("Fobidden Access: "+ file));
}
```

Die Methode checkRead()

```
public void checkRead(String file)
{
        // für den Zugriff auf Klassenbibliothek
         if (file.startsWith(LibPath)) return;


        // für die eigene Klasse, startsWith geht nicht!!
        // offenbar wird ein absoluter Pfad gebildet
        if (file.endsWith("TinyServConn.class")) return;


        if(file.startsWith("/") | file.contains(".."))
    throw (new SecurityException("Fobidden Access: "+ file));
}
```

Ein Client

- Der Client muss in etwas das senden, was sonst der Browser sendet.
- Ganz wichtig:
  - Alle Zeiölen, die der Client zum Server sendet, müssen mit \n\r

Ein Client

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

// java TinyClient <host> <port> <file>

public class TinyClient
{
  public static void main(String args[])
  {
    try
    {
      Socket S;
      InetAddress ihost=InetAddress.getByName(args[0]);
      S=new Socket(ihost,Integer.parseInt(args[1]));
      PrintStream    O=new PrintStream(S.getOutputStream());
      O.print("GET "+args[2]+"\r\n" );
      O.print("Host: "+args[0]+":"+Integer.parseInt(args[1])+"\r\n" );
      BufferedReader I=new BufferedReader(new InputStreamReader(S.getInputStream()));
      String X;
      while ((X=I.readLine())!=null) System.out.println(X);
      S.close();
    }catch (Exception e){System.out.println(e); e.printStackTrace();}
  }
}
```

Ein paar kleine Verbesserungen

Man könnte über das Fileobjekt prüfen, ob die Datei vorhanden und lesbar ist und ggf. eine Message an den Client ausgeben.

```
File F=new File(req);
if (F.exists() && F.canRead() && F.isFile())
{
  FileInputStream fis=new FileInputStream(req);
  int len;
  while ((len=fis.read(buf))!=-1) out.write(buf,0,len);
}
else
{
   pout.println
      ("Message form Server:"
      +req
      +" not found or not readable or no File");
}
```

Exception an den Client weiterleiten

- In der Variante unten wird eine Exception an den Client ausgegeben und der Stream danach ordentlich geschlossen. Als Besonderheit haben wir ein try in einem catch Block.Auch Security Exceptions vom SecurityManager werden hier weitergeleitet.

```
}catch(Exception e)
{
  System.out.println("I/O Error "+e);
  pout.println("Message form Server:"+e);
  try
  {
    sock.shutdownOutput();sock.close();
  }catch (Exception xyz){}
}
```