

# Networking

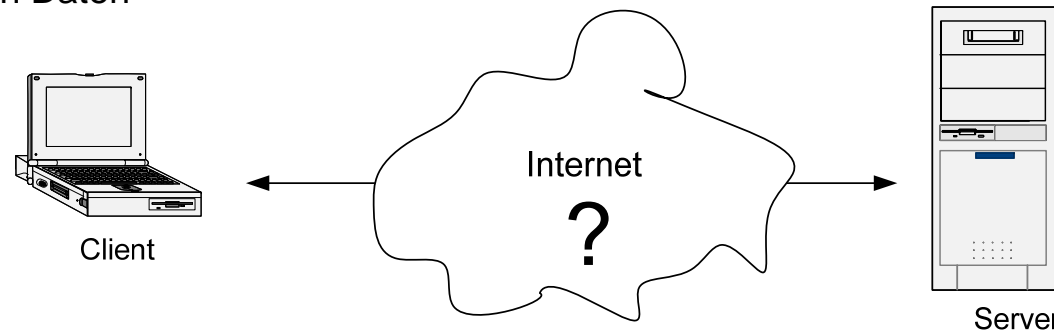


- Grundlagen
- Klasse Socket
- Klasse ServerSocket
- Klasse URL



# Netzwerkprogrammierung in Java

- Programme schreiben, wobei Teile auf unterschiedlichen Rechnern laufen und diese Programmteile miteinander kommunizieren
- Grundproblematik der Netzwerkprogrammierung
  - Zueinanderfinden der verteilten Programme
  - Aufbau einer Verbindung
  - Verständigung über die Kommunikation
  - Austausch von Nachrichten
  - Austausch von Daten

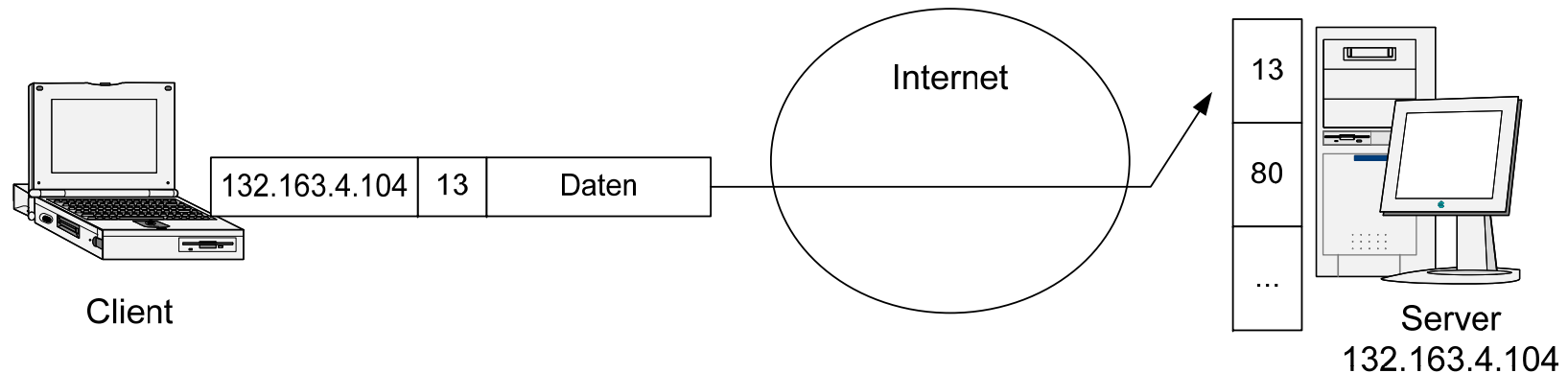


## 2 Modelle:

1. Socket-Streaming (dieses Kapitel)
2. Remote-Objekte (nächstes Kapitel)



# Prinzipien von Socket-Streaming



- Adressierung über IP-Adressen und Ports
- Client und Server
  - Client schickt Anforderung an Server
  - Server behandelt Anforderung und
  - schickt Antwort zurück
- Austausch von Nachrichten und Daten zwischen Client und Server erfolgt über Byte-Streams ähnlich dem Dateizugriff (siehe Streaming)
  - ➔ Nachrichten und Daten müssen interpretiert werden; erfolgt unter Verwendung eines entsprechenden Protokolls, z.B. HTTP, SMTP oder andere

Damit steht Socket-Streaming im Gegensatz zu Remote-Objects, die die Semantik von Java-Objekten haben!



# IP-Adressierung

- 4 Byte IP-Adresse



- geteilt in *Netzwerk-ID* und eine *Host-ID*
- Adresstypen: Klassen A, B, C

Klasse	Netzwerk-ID	Host-ID	Beschreibung
A	7	24	Für sehr große Netzbetreiber vorgesehen.
B	14	16	Ein Klasse-B-Netz erlaubt immerhin noch die eindeutige Adressierung von 65534 unterschiedlichen Rechnern innerhalb des Netzwerks.
C	21	8	Klasse-C-Netze sind für kleinere Unternehmen vorgesehen, die nicht mehr als 254 unterschiedliche Rechner adressieren müssen.

- Domain-Namen: symbolische Namen für IP-Adressen



# Ports

- Ports dienen dazu, Serverdienste auf einem Rechner zu identifizieren
- Ganzzahl im Bereich 0 .. 65535
- Viele Portnummern standardisiert (*well known ports*: 0-1024):

Name	Port	Transport	Beschreibung
<nil>	0	-	Reserviert
echo	7	tcp/udp	Gibt jede Zeile zurück, die der Client sendet
discard	9	tcp/udp	Ignoriert jede Zeile, die der Client sendet
daytime	13	tcp/udp	Liefert ASCII-String mit Datum und Uhrzeit
ftp	21	tcp	Versenden und Empfangen von Dateien
telnet	23	tcp	Interaktive Session mit entferntem Host
smtp	25	tcp	Versenden von E-Mails
whois	43	tcp	Einfacher Namensservice
finger	79	tcp	Liefert Benutzerinformationen
www	80	tcp	Der Web-Server
pop3	110	tcp	Übertragen von Mails
rmi	1099	tcp	Remote Method Invocation



# Sockets

- Sockets sind eine Programmierschnittstelle für streambasierte Kommunikation
- Übertragen von Daten ähnlich dem Schreiben einer Datei
  - Aufbau einer Verbindung über IP-Adresse und Port
  - Lesen oder Schreiben von Daten
  - Schließen der Verbindung
- In Java unterscheidet man zwischen Client- und Server-Sockets
  - Client-Sockets (Klasse Socket) für das eigentliche Lesen und Schreiben (sowohl am Client als auch am Server)
  - Server-Sockets (Klasse ServerSocket) für das Annehmen von Client-Requests
- Lesen und Schreiben erfolgt über `InputStreams` und `OutputStreams`



# Wichtige Klassen von java.net

- **Socket**  
Socket für Data-Streaming
- **ServerSocket**  
Socket für die Annahme von Requests am Server
- **InetAddress**  
Repräsentiert eine IP-Adresse
- **URL**  
Repräsentiert eine URL



# Klasse InetAddress

- Klasse InetAddress repräsentiert eine IP-Adresse

```
public class InetAddress extends Object implements Serializable
```

- statische Methoden zur Erzeugung von InetAddress-Objekten

```
static InetAddress[] getAllByName(String host)
static InetAddress getByAddress(byte[] addr)
static InetAddress getByAddress(String host, byte[] addr)
static InetAddress getByName(String host)
static InetAddress getLocalHost()
```

- Zugriff auf 4 Byte IP-Adresse und symbolischer Name

```
String getCanonicalHostName()
String getHostName()
String.getHostAddress()
byte[] getAddress()
```

```
Canonical: oberon.ssw.uni-linz.ac.at
Name: oberon.ssw.uni-linz.ac.at
Host address: 140.78.145.1
Address: {140, 78, 145, 1}
```



- Grundlagen
- Klasse Socket
- Klasse ServerSocket
- Klasse URL



# Klasse Socket

- Die Klasse Socket repräsentiert einen Socket für Data-Streaming mit

```
public class Socket
```

- Konstruktor zum Aufbau einer Verbindung mit Adresse und Port

```
public Socket(String address, int port)
```

```
public Socket(InetAddress address, int port)
```

- Zugriffsmethoden auf InputStream und OutputStream für dieses Socket-Objekt

```
public InputStream getInputStream()
```

```
public OutputStream getOutputStream()
```



# Beispiel Client-Socket: Zugriff auf WebServer (1)

- Das folgende Beispiel zeigt eine Client-Anfrage an einen Web-Server
  - Verbindungsaufbau erfolgt mit der in der Kommandozeile angegebenen Adresse des Web-Servers
  - Es wird dabei ein Socket-Objekt erzeugt, wobei der Port 80 als Standardport für WWW verwendet wird

```
C: \> java GetPage <server> <page>
```

```
public class GetPage {  
    public static void main(String[] args) {  
        Socket sock = null;  
        try {  
            InetAddress adr = InetAddress.getBy_name(args[0]);  
            sock = new Socket(adr, 80);
```

- Danach werden vom Socket-Objekt sowohl ein InputStream als auch OutputStream geholt
- Es erfolgt die Konstruktion und die Ausgabe eines HTTP GET-Kommandos an den Web-Server wobei das zweite Kommandozeilenargument den Dateinamen bestimmt

```
        OutputStream out = sock.getOutputStream();  
        BufferedReader in = new BufferedReader(  
            new InputStreamReader(sock.getInputStream()));  
        // send GET command  
        String s = "GET " + args[1] + " HTTP/1.0" + "\r\n\r\n";  
        out.write(s.getBytes());
```



## Beispiel Client-Socket: Zugriff auf WebServer (2)

- Die Antwort vom Server wird zeilenweise gelesen und auf der Konsole ausgegeben

```
String line = in.readLine();
while (line != null) {
    System.out.println(line);
    line = in.readLine();
}
```

- Es folgt noch die catch-Anweisung, um alle auftretenden IOExceptions zu fangen
- Am Ende werden InputStream, OutputStream und Socket geschlossen

```
} catch (IOException e) {
    System.err.println(e.toString());
} finally {
    // in.close();
    // out.close();
    if (sock != null) {
        try { sock.close(); } catch (IOException ign) { ... log ... }
    }
}
}
```



- Grundlagen
- Klasse Socket
- Klasse ServerSocket
- Klasse URL



# Klasse ServerSocket

- Die Klasse ServerSocket repräsentiert den Einstieg in einen Server und dient dazu Client-Requests anzunehmen

```
public class ServerSocket
```

- Konstruktor zum Anlegen eines ServerSockets mit entsprechender Portnummer. Der ServerSocket horcht damit auf den Port auf dem lokalen Rechner.

```
public ServerSocket(int port) throws IOException
```

- Die wichtigste Methode ist die Methode accept. Mit dieser horcht der ServerSocket auf hereinkommende Requests. Sie blockiert, bis ein Request am Port hereinkommt und liefert dann einen normalen Socket für die Datenkommunikation mit dem Client.

```
public Socket accept()
```



# Beispiel ServerSocket: SimpleEchoServer (1)

- Das folgende Programm zeigt die Verwendung von ServerSocket
  - Der ServerSocket wird mit Portnummer 7 erzeugt und horcht dann auf hereinkommende Client-Requests

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

public class SimpleEchoServer {
    public static void main(String[] args) {
        ServerSocket echod = null;
        Socket socket = null;
        try {
            System.out.println("Warte auf Verbindung auf Port 7...");
            echod = new ServerSocket(7);
            socket = echod.accept();
```

- sobald ein Request eingeht, liefert accept einen Socket für Ein- und Ausgabe.

```
System.out.println("Verbindung hergestellt");
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```



## Beispiel ServerSocket: SimpleEchoServer (2)

- Über `InputStream in` und `OutputStream out` erfolgt das Lesen der Daten vom Client und das Zurücksenden der selben Daten zum Client

```
int c;
while ((c = in.read()) != -1) {
    out.write((char) c);
    System.out.print((char) c);
}
```

- Danach werden alle Sockets geschlossen die Verbindung abgebaut.

```
System.out.println("Verbindung beenden");
} catch (IOException e) {
    System.err.println(e.toString());
} finally {
    if (socket != null) {
        try { socket.close(); } catch (IOException ioex) { ... }
    }
    if (echod != null) {
        try { echod.close(); } catch (IOException ioex) { ... }
    }
}
}
```



# Beispiel Mehrere Client bedienen: EchoServer (1)

- Normalerweise will ein Server viele einkommende Clients bedienen
- Dies wird erreicht, indem der accept-Aufruf in eine Schleife verpackt und für jeden einkommenden Client-Request ein eigener Thread EchoClientThread erzeugt wird.

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

public class EchoServer {
    public static void main(String[] args) {
        int cnt = 0;
        try {
            System.out.println("Warte auf Verbindungen auf Port 7...");
            ServerSocket echod = new ServerSocket(7);
            while (true) {
                Socket socket = echod.accept();
                new EchoClientThread(++cnt, socket).start();
            }
        } catch (IOException e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }
}
```



## Beispiel Mehrere Client bedienen: EchoServer (2)

- In der run-Methode des EchoClientThreads werden die Daten wie gehabt gelesen und zurückgeschickt.

```
class EchoClientThread extends Thread {
    private int    name;
    private Socket socket;

    public EchoClientThread(int name, Socket socket) {
        this.name = name; this.socket = socket;
    }

    public void run() {
        try {
            InputStream in = socket.getInputStream();
            OutputStream out = socket.getOutputStream();
            int c;
            while ((c = in.read()) != -1) {
                out.write((char) c);
                System.out.print((char) c);
            }
        } catch (IOException e) {
            // Log exception ...
        } finally {
            if (socket != null) { ... socket.close(); ... }
        }
    }
}
```



# Beispiel E-Mail senden

- E-Mail senden erfolgt nach dem SMTP-Protokoll auf Port 25
- Das SMTP-Protokoll hat folgendes Format (RFC 821, RFC 2821)

```
HELO Sender Host
MAIL FROM: <E-Mail Adresse des Absenders>
RCPT TO: <E-Mail Adresse des Empfängers>
DATA
From: <E-Mail Adresse des Absenders>
Subject: <Betreff>
To: <E-Mail Adresse des Empfängers>
Date: <Datum>

E-Mail Nachricht in mehreren Zeilen
.
QUIT
```



# Beispiel E-Mail senden

- Verbindungsaufbau erfolgt
  - durch Öffnen eines Sockets zum E-Mail-Host
  - und Erzeugen eines `PrintWriters` und eines `Readers` für den Socket

```
Socket mailSocket = new Socket("email.uni-linz.ac.at", 25);
PrintWriter out = new PrintWriter(mailSocket.getOutputStream());
BufferedReader in = new BufferedReader(
    new InputStreamReader(mailSocket.getInputStream()));
```

- Senden von Nachrichten erfolgt durch zeilenweises Schicken der Protokollanweisungen und Daten, wobei
  - jede Zeile mit `"\r\n"` abgeschlossen werden muss
  - Nach jedem Kommando und nach dem gesamten Text jeweils die Antwortzeilen gelesen werden müssen
- Die DATA Section
  - Sollten jeweils Zeilen enthalten mit Angabe von
    - Absender (From: ...)
    - Empfänger (To: ...)
    - Datum (Date: ...)
    - Betreff (Subject: ...)
  - mehrere Zeilen mit Nachricht
    - Text mit beliebigen ASCII-Zeichen (0 bis 127)
  - Am Ende einen einzelnen Punkt `.` in einer Zeile (`"\r\n.\r\n"`)



# Beispiel E-Mail senden

```
public static void main(String[] args) {
    Socket mailSocket;
    BufferedReader in;
    PrintWriter out;

    try {
        mailSocket = new Socket("...Adresse des Mail servers ...", 25);
        out = new PrintWriter(mailSocket.getOutputStream());
        in = new BufferedReader(new InputStreamReader(mailSocket.getInputStream()));

        receive(in);
        String hostName = InetAddress.getLocalHost().getHostName();
        send(out, "HELO " + hostName);
        receive(in);
        send(out, "MAIL FROM: <max.mustermann@jku.at>");
        receive(in);
        send(out, "RCPT TO: <herbert.praehofer@jku.at>");
        receive(in);
        send(out, "DATA");
        receive(in);
        send(out, "From: max.mustermann@jku.at");
        send(out, "To: franz.mayr@jku.at");
        send(out, "Date: " + DateFormat.getDateInstance().format(new Date()));
        send(out, "Subject: 2 zeilige Mail");
        send(out, "");
        send(out, "Dies ist eine Mail");
        send(out, "mit zwei Zeilen.");
        send(out, ".");
        receive(in);
        send(out, "Quit");
        receive(in);
    }
    catch (UnknownHostException e) { ... }
    catch (IOException e) { ... }
    finally { if (mailSocket != null) { try { mailSocket.close(); } catch ... } }
}
```



# Beispiel E-Mail senden (Forts.)

```
/** Receives a line from BufferedReader in
 * @param in
 * @return
 * @throws IOException
 */
private static String receive(BufferedReader in) throws IOException {
    String str = in.readLine();
    if (str == null) {
        str = "";
    }
    System.out.println("smtp (receiving): " + str);
    return str;
}

/** Sends a line to PrintWriter out
 * @param out
 * @param str
 * @throws IOException
 */
private static void send(PrintWriter out, String str) throws IOException {
    out.print(str);
    System.out.println("smtp (sending): " + str);
    out.print("\r\n");
    out.flush();
}
```



- Grundlagen
- Klasse Socket
- Klasse ServerSocket
- Klasse URL



# URLs

- URL vereinfacht das Abholen von Informationen auf Remote-Sites
  - URL wird mit dem String, der die Remote-Ressource adressiert, angelegt, z.B. File <http://java.sun.com/index.html>

```
public final class URL implements java.io.Serializable
```

```
public URL(String spec) throws MalformedURLException
```

```
public URL(String protocol, String host, int port,  
String file) throws MalformedURLException
```

- Erlaubt einen InputStream für ein URL zu holen; mit diesem kann die Ressource gelesen werden

```
public final InputStream openStream() throws IOException
```

- Mit getContent kann der gesamte Inhalt der URL-Ressource geholt werden

```
public final Object getContent() throws IOException
```

- Mit openConnection wird eine URLConnection geöffnet

```
public URLConnection openConnection() throws IOException
```



# Beispiel URL: Sichern eine Remote-Resource (1)

- Das folgende Beispiel liest eine Ressource und sichert den Inhalt in einer Datei
  - Die URL wird mit der in der Kommandozeile gegebenen Adresse der Ressource erzeugt
  - Es wird eine Datei und ein OutputStream auf die Datei erzeugt
  - Es wird auf den InputStream der URL zugegriffen
  - Die Daten werden gelesen und auf den File ausgegeben

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

public class SaveURL {

    public static void main(String[] args) {
        OutputStream out = null;
        InputStream in = null;
        try {
            URL url = new URL(args[0]);
            out = new FileOutputStream(args[1]);
            in = url.openStream();
            int len;
            byte[] b = new byte[100];
            while ((len = in.read(b)) != -1) {
                out.write(b, 0, len);
            }
        }
    }
}
```

```
C:\> java SaveURL <url> <output-filename>
```



## Beispiel URL: Sichern eine Remote-Resource (2)

- Wichtig ist das Abfangen der entsprechenden Exceptions

```
    } catch (MalformedURLException e) {  
        ...  
    } catch (IOException e) {  
        ...  
    } finally {  
        if (out != null) {  
            try { out.close(); }  
            catch (IOException e) {  
                ...  
            }  
        }  
        if (in != null) {  
            try { in.close(); }  
            catch (IOException e) {  
                ...  
            }  
        }  
    }  
}
```



# Exceptions

- Netzwerkkommunikation ist grundsätzlich eine unsichere Sache
- Es gibt daher eine Vielzahl von Exceptions, die die möglichen Pannen gut verdeutlichen

## Exception-Hierarchie:

IOException :	Basisklasse für alle Exceptions mit IO
MalformedURLException :	ungültige URL
SocketException :	Basisklasse für Exceptions bei Sockets
BindException :	Fehler bei Versuch Socket an eine <u>lokale</u> Adresse oder Port zu binden
ConnectException :	Fehler bei Versuch Socket an eine <u>remote</u> Adresse oder Port zu verbinden
NoRouteToHostException :	Remote Host kann nicht erreicht werden
PortUnreachableException :	Port konnte nicht erreicht werden
UnknownHostException :	Host mit IP-Adresse nicht bekannt
UnknownServiceException :	Service bei URL nicht unterstützt
ProtocolException :	z.B. TCP Fehler
InterruptedIOException :	
SocketTimeoutException :	Timeout für Socket-Operation



# Literatur

- Horstmann, Cornell, Core Java 2, Band2 Expertenwissen, Markt und Technik, 2002: Kapitel 3
- Krüger, Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003, <http://www.javabuch.de>: Kapitel 45

