Kapitel 20

Client/Server-Programmierung in Netzwerken

In den letzten Jahren hat die Anzahl und Nutzung von Computernetzwerken explosionsartig zugenommen. Ob nun in Bildungseinrichtungen, Verwaltungsbehörden, Unternehmen, öffentlichen Einrichtungen oder im Privatbereich – in fast allen Bereichen des täglichen Lebens haben wir mittlerweile direkt oder indirekt Kontakt mit vernetzten Rechnersystemen. Für viele Nutzerinnen und Nutzer von Rechnern sind Aktivitäten wie das Lesen von Webseiten im Internet bzw. der Zugriff auf Daten auf einem entfernten Rechner, das Versenden von E-Mails, das Diskutieren in Chatrooms oder die gemeinsame Verwendung eines Druckers zusammen mit anderen Nutzern beinahe zur Selbstverständlichkeit geworden. Alle diese Anwendungen setzen, genau wie die bereits in Kapitel 17 behandelte Ausführung von Applets in einem Browser, voraus, dass verschiedene Programme auf unterschiedlichen Rechnern miteinander kommunizieren.
Als Programmiersprache für das Internet bietet Java natürlich die Möglichkeit, Programme zu schreiben, die eine derartige Kommunikation über Netzwerke realisieren können. In diesem Kapitel wollen wir daher einige wichtige, aber natürlich bei Weitem nicht alle Aspekte der Netzwerk-Programmierung kennen lernen. Aufgrund der umfangreichen Java-Klassenbibliothek im Paket `java.net` und mit unseren Kenntnissen hinsichtlich Threads (Kapitel 18) und Streams (Kapitel 19) können wir mit wenig Aufwand Programme entwickeln, die im Internet mit anderen Programmen bzw. Rechnern kommunizieren. Dabei ist es eigentlich nicht einmal notwendig, dass wir die zugrunde liegende Netzwerktechnologie oder die Details der Kommunikationsvorgänge kennen bzw. verstehen. Dennoch erläutern wir im nachfolgenden Abschnitt zunächst einige Begriffe aus der Welt der Netzwerke und der Netzwerk-Kommunikation, um zumindest ein Grundverständnis dafür zu vermitteln, bevor wir uns dann der eigentlichen Netzwerk-Programmierung in Java zuwenden.

20.1 Wissenswertes über Netzwerk-Kommunikation

20.1.1 Protokolle

Der Daten- bzw. Nachrichtenaustausch in einem Netzwerk erfolgt immer paarweise, das heißt, ein Programm auf einem Rechner nimmt mit einem Programm auf einem anderen Rechner Kontakt auf und tauscht mit ihm Daten aus. Damit dies auch tatsächlich funktioniert, müssen sich die kommunizierenden Computer bzw. Programme zuvor auf ein so genanntes **Protokoll** geeinigt haben. Darunter versteht man alle Regeln für den Verbindungsaufbau, den eigentlichen Datenaustausch und den Verbindungsabbau. Die Kommunikation über eine Netzwerkverbindung läuft jedoch nicht direkt von Anwendungsprogramm zu Anwendungsprogramm, sondern wird über verschiedene Schichten des gesamten Kommunikationssystems abgewickelt. Daher müssen auch für jede dieser Schichten entsprechende Protokolle festgelegt sein.
Will man Programme auf unterschiedlichsten Rechnern miteinander verbinden, ist ein standardisiertes Modell für den Aufbau (die Architektur) des Kommunikationssystems unerlässlich. Ein solcher Standard, der mit sieben verschiedenen Schichten arbeitet, wurde daher in Form des OSI-Standards (OSI steht für Open System Interconnect) von der Internationalen Standardisierungs-Organisation (ISO [28]) festgelegt. In der Praxis findet man allerdings wesentlich häufiger den TCP/IP-Standard, der eine etwas vereinfachte Unterteilung in vier Schichten vornimmt:

- In der obersten Schicht, der **Anwendungsschicht**, wird mit den Protokollen gängiger Netzwerkanwendungen, wie zum Beispiel ein **File Transfer Protocol** (**FTP**, Übertragung von Dateien), ein **Hypertext Transfer Protocol** (**HTTP**, Übertragung von Hypertext-Dokumenten) und ein **Simple Mail Transfer Protocol** (**SMTP**, Versenden von Mails) oder mit Protokollen spezieller Anwendungen gearbeitet.
- In der darunter liegenden Schicht, der **Transportschicht**, wird als Transportprotokoll das **Transmission Control Protocol** (**TCP**) oder das **User Datagram Protocol** (**UDP**) eingesetzt.
- Unterhalb der Transportschicht befindet sich die **Netzwerkschicht** (auch Internetschicht genannt), in der das **Internetprotokoll** (**IP**) für die Kommunikation zuständig ist.
- Auf der untersten Schicht, der **physikalischen Schicht**, die für die tatsächliche Verbindung über das „Netz“ in Form von Leitungen zwischen den Rechnern zuständig ist, laufen typischerweise Protokolle wie zum Beispiel **Ethernet** oder **Fiber Distributed Data Interface** (**FDDI**, Übertragung auf Lichtwellenleitern).

Versendet eine Anwendung Daten an eine andere Anwendung über ein Netzwerk, so durchlaufen die Daten die verschiedenen Schichten. Dabei verändert die Protokoll-Software der jeweiligen Schicht die Daten, indem zusätzliche Informationen eingearbeitet werden, die beim Empfänger-Rechner die Protokoll-Software

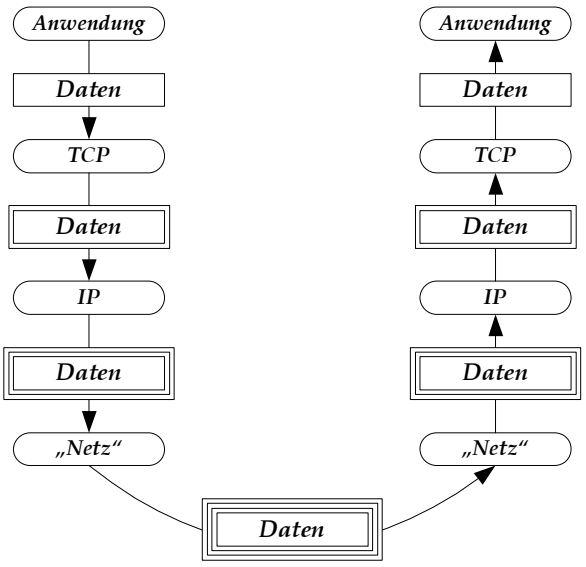

Abbildung 20.1: Datenübertragung im TCP/IP-Schichtenmodell

der entsprechenden Schicht nutzten kann, um die Daten zu verarbeiten. Abbildung 20.1 verdeutlicht diesen Vorgang.
Dabei stellt das IP in der Netzwerkschicht einen verbindungslosen und unzuverlässigen Dienst für den Transport von einzelnen Datenpaketen bereit, der diese lediglich mit der Empfängeradresse versieht und möglichst auf dem schnellsten Wege (man spricht von einem möglichst effizienten **Routing**, abhängig von der gerade vorherrschenden Netzlast) verschickt.
Das TCP in der Transportschicht stellt hingegen eine zuverlässige (virtuelle) Verbindung zwischen Sender- und Empfänger-Anwendung her. Die Daten werden in kleine Pakete eingeteilt, und es wird stets sichergestellt, dass diese fehlerfrei übertragen werden, indem vom Empfänger eine Bestätigung über deren Erhalt gefordert wird und die Pakete, falls erforderlich, mehrfach gesendet werden. Zusätzlich wird dafür gesorgt, dass auch die Reihenfolge der verschickten Datenpakete erhalten bleibt. In den weiteren Abschnitten dieses Kapitels beschäftigen wir uns noch ausführlich mit der Realisierung von TCP-Verbindungen in Java.
Das alternative Transportschicht-Protokoll UDP ist im Gegensatz zu TCP ein unzuverlässiges Protokoll, das im Prinzip lediglich die IP-Funktionalität an die Anwendungsschicht weiterreicht und weder Reihenfolge noch fehlerfreie Übermittlung garantiert. Allerdings gibt es auch für UDP sinnvolle Einsatzbereiche, wie zum Beispiel die Übertragung von Messwerten einer Wetterstation in kurzen Zeitabständen, bei der es nicht unbedingt auf die Vollständigkeit der übermittelten Daten ankommt.

20.1.2 IP-Adressen

Für die Kommunikation zwischen zwei Programmen bzw. Rechnern und die Abwicklung des Protokolls ist es natürlich notwendig, die jeweilige Adresse des Partner-Rechners im Netzwerk, die so genannte **IP-Adresse**, zu kennen. Das Internet-Protokoll arbeitet derzeit mit numerischen Adressen, die 4 Bytes (also 32 Bits) lang sind und in der Regel durch vier durch Punkte getrennte Zahlen im Bereich 0 bis 255 dargestellt werden. Die IP-Adresse des WWW-Servers `www.hanser.de` des Hanser-Verlages lautet beispielsweise `194.59.179.52`.
Da die IP-Adresse eines Rechners im Internet weltweit eindeutig sein muss, werden die Adressen von der zentralen Organisation **ICANN** (Internet Corporation for Assigned Names and Numbers [27]) verwaltet bzw. vergeben. Aufgrund der wachsenden Zahl von Rechnern im Internet sollen IP-Adressen künftig mit 16 Bytes bzw. durch acht durch Punkte getrennte Hexadezimalzahlen im Bereich 0000 bis FFFF dargestellt werden.
Wesentlich einprägsamer können Internet-Adressen natürlich in Form so genannter **Domain-Namen** oder **Host-Namen** notiert werden, wobei sich einer IP-Adresse auch mehrere Namen (man spricht dann von **Alias-Namen**) zuordnen lassen. Um diese Art der Notation im Internet verwenden zu können, wird allerdings ein Dienst benötigt, der die Abbildung des Namens auf die tatsächliche IP-Adresse vornehmen kann. Dieser Dienst heißt **Domain Name Service** (**DNS**) und wird auch von den Java-Klassen bei Bedarf genutzt. Wir wollen seine Funktionsweise anhand der Klasse `InetAddress` aus dem Paket `java.net` kurz demonstrieren.
Objekte der Klasse `InetAddress` können nicht wie üblich per Konstruktor erzeugt, sondern müssen durch Aufruf der Klassenmethode

- **`public static`** `InetAddress getByName(String host)` **`throws`** `UnknownHostException`
  führt für den Rechner `host` eine Anfrage beim DNS durch und liefert ein Objekt, das die IP-Adresse des durch `host` angegebenen Rechners darstellt, zurück. Dabei kann `host` als Domain-Name oder als IP-Adresse angegeben werden.

konstruiert werden. Danach können wir die Instanzmethoden `getHostAddress` und `getHostName` benutzen, um den Rechner-Namen und die IP-Adresse eines `InetAddress`-Objekts als Zeichenkette zu erhalten. Wir haben die drei genannten Methoden in dem einfachen Programm

```
1 import java.net.*;
2 class DNSAnfrage {
3   public static void main(String[] args) {
4     try {
5         InetAddress ip = InetAddress.getByName(args[0]);
6         System.out.println("Angefragter Name: " + args[0]);
7         System.out.println("IP-Adresse:       " + ip.getHostAddress());
8         System.out.println("Host-Name:        " + ip.getHostName());
9     } catch (ArrayIndexOutOfBoundsException aex) {
```

```
10          System.out.println("Aufruf: java DNSAnfrage <hostname>");
11      } catch (UnknownHostException uex) {
12          System.out.println("Kein DNS-Eintrag fuer " + args[0]);
13      }
14    }
15  }
```

eingesetzt, das die IP-Adresse eines per Kommandozeilenparameter übergebenen Rechnernamens beim DNS erfragt und danach die IP-Adresse und den Rechnernamen auf das Konsolenfenster ausgibt. Falls der Parameter beim Start vergessen wurde oder der Rechner dem DNS nicht bekannt ist, werden die entsprechenden Ausnahmen behandelt, indem entsprechende Informationen ausgegeben werden. Aufrufe unseres Programms laufen daher (bei vorhandener Internetverbindung) wie folgt ab:

Konsole

```
> java DNSAnfrage www.hanser.de
Angefragter Name: www.hanser.de
IP-Adresse:       194.59.179.52
Host-Name:        www.hanser.de

> java DNSAnfrage 192.18.97.71
Angefragter Name: 192.18.97.71
IP-Adresse:       192.18.97.71
Host-Name:        flres.java.Sun.COM

> java DNSAnfrage lord.of.the.rings
Kein DNS-Eintrag fuer lord.of.the.rings
```

20.1.3 Ports und Sockets

Wie wir bereits wissen, ist die Transportschicht für die eigentliche Verbindung zwischen Sender- und Empfänger-Anwendung zuständig. Weil auf einem Rechner durchaus mehrere Anwendungen gleichzeitig Internet-Kommunikation betreiben können, der Rechner in der Regel aber nur über eine physikalische Verbindung zum Internet verfügt, lässt sich der Weg, den die übermittelten Daten nehmen sollen, nicht allein anhand der IP-Adresse festlegen. Für welche Anwendung die Daten bestimmt sind, bestimmt daher eine zusätzliche Adressierungs-Information, die das Transportschicht-Protokoll in die Daten einarbeitet – die so genannte **Port-Nummer**.[1] Jede Netzwerk-Anwendung auf einem Rechner wird über einen festgelegten **Port** abgewickelt, so dass die Daten an die richtige Stelle ausgeliefert werden können.
Port-Nummern sind ganze Zahlen im Bereich von 0 bis 65535. Während die Port-Nummern im Bereich von 0 bis 1023 für Standardanwendungen (z. B. Port 21 für

[1] Vergleicht man diese Adressierungsart mit der herkömmlichen Verteilung von Brief- oder Paketpost in einem Wohnheim, so entspricht die IP-Adresse der üblichen Adresse mit Straße und Hausnummer, während die Port-Nummer die Zimmernummer des Empfängers spezifiziert.

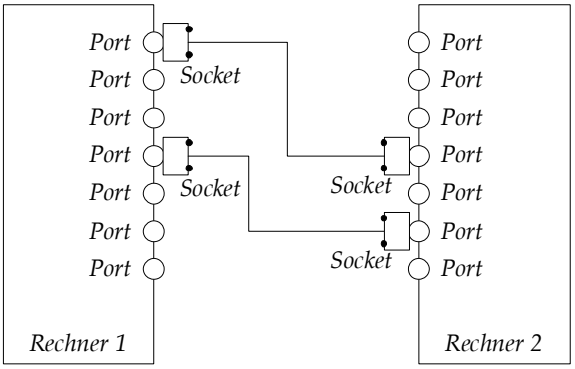

Abbildung 20.2: Netzwerkverbindungen über Ports und Sockets

einen FTP-Server, Port 25 für einen SMTP-Server oder Port 80 für einen HTTP-Server) reserviert sind, sind alle anderen Werte frei verfügbar und können für selbst geschriebene Netzwerk-Anwendungen verwendet werden.
Steht für eine Netzwerk-Kommunikation zwischen zwei Rechnern fest, welche Anwendungen bzw. Ports miteinander kommunizieren, so sind dadurch die Endpunkte der Verbindung bzw. der Datenübertragung in beide Richtungen bestimmt. Einen solchen durch IP-Adresse und Port-Nummer eindeutig festgelegten Endpunkt einer Netzwerk-Kommunikationsverbindung nennt man **Socket** (deutsch: Steckdose, Buchse). Abbildung 20.2 verdeutlicht diesen Sachverhalt anhand von zwei Netzwerkverbindungen, die beide jeweils eine Anwendung (einen Port) auf Rechner 1 mit einer Anwendung (einem Port) auf Rechner 2 verbinden. Im nächsten Abschnitt beschäftigen wir uns nun mit der Java-Realisierung von TCP-Sockets.

20.2 Client/Server-Programmierung

Unter einem **Server** (deutsch: Diener) versteht man ein Programm, das auf einem Rechner läuft und einen bestimmten **Dienst** anbietet, der über das Netzwerk von anderen Programmen bzw. Rechnern genutzt werden kann. Der Rechner, auf dem das Programm läuft, heißt dann **Server-Rechner** oder auch **Server-Host**.[2] Ein Programm, das über das Netzwerk den Dienst eines Servers anfordert, wird **Client** (deutsch: Klient, Kunde) genannt, der entsprechende Rechner, auf dem der Client läuft, heißt dann **Client-Rechner** oder **Client-Host**.
Wollen Server und Client eine Kommunikationsverbindung aufbauen, müssen Server-seitig folgende Vorgänge ablaufen:

[2] Sehr häufig wird der Begriff Server auch (fälschlicherweise) für den Rechner verwendet, auf dem ein oder mehrere Server laufen.

1. Der Server erzeugt einen speziellen Server-Socket, der an einen Port gebunden ist, dessen Nummer den potentiellen Clients bekannt sein muss.
2. Der Server wartet darauf, dass sich ein Client anmeldet, der eine Verbindung aufbauen möchte.
3. Hat der Server die Anfrage eines Clients akzeptiert, erzeugt er an einem freien Port einen weiteren Socket, über den die Kommunikation abgewickelt werden kann.
4. Danach werden über diesen Socket die benötigten Ein- und Ausgabeströme zum Client geöffnet.
5. Über die Ströme wird der Datenaustausch gemäß dem festgelegten Protokoll abgewickelt.
6. Die Ströme und der Socket werden geschlossen.
7. Der Server wird beendet, oder es beginnt ab Schritt 2 eine weitere Client-Kommunikation.

Client-seitig sieht der Ablauf wie folgt aus:

1. Der Client nimmt über die IP-Adresse und Port-Nummer Kontakt mit dem Server auf und erzeugt einen Socket, über den die Kommunikation mit dem Server abgewickelt werden kann.
2. Hat der Server die Anfrage akzeptiert, werden über den Socket die benötigten Ein- und Ausgabeströme zum Server geöffnet.
3. Über die Ströme wird der Datenaustausch gemäß dem festgelegten Protokoll abgewickelt.
4. Die Ströme und der Socket werden geschlossen.
5. Der Server wird beendet, oder es wird, beginnend bei Schritt 2, eine weitere Client-Kommunikation abgewickelt.

In den nachfolgenden Abschnitten werden wir nun sehen, wie diese Vorgänge mit relativ wenig Aufwand in Form von Java-Programmen realisierbar sind.

20.2.1 Die Klassen `ServerSocket` und `Socket`

Java stellt im Paket `java.net` zwei verschiedene Klassen für die Erzeugung von TCP-Sockets zur Verfügung. Die Klasse `ServerSocket` dient der Konstruktion spezieller Server-Sockets, während die Klasse `Socket` sowohl auf Server- als auch auf Client-Seite eingesetzt wird. Ein Server-Socket wird mit dem Konstruktor

- **`public`** `ServerSocket(`**`int`** `port)`
  erzeugt einen Server-Socket am angegebenen Port.

erzeugt und ist mit Hilfe der Instanzmethode

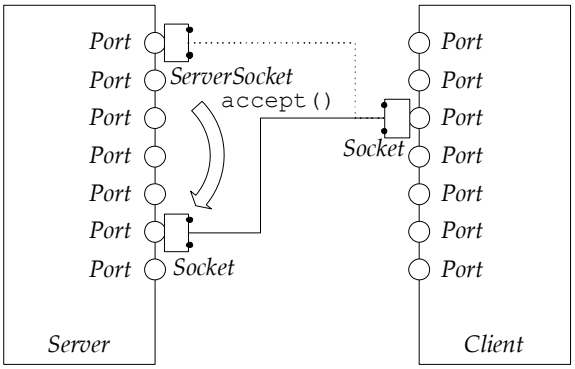

Abbildung 20.3: Sockets auf Server- und Client-Seite

- **public** `Socket accept()`
  wartet auf eine Anfrage eines Clients und erzeugt dann ein neues `Socket`-Objekt und liefert es als Ergebnis zurück.

in der Lage, die Anfrage eines Clients zu akzeptieren und ein `Socket`-Objekt zu erzeugen, über das man die Kommunikation abwickeln kann (siehe auch Abbildung 20.3). Auf Client-Seite werden `Socket`-Objekte direkt mit den Konstruktoren

- **public** `Socket(InetAddress address,` **int** `port)`
  erzeugt einen Socket und verbindet ihn mit der Anwendung, die auf dem Rechner mit der durch `address` festgelegten Adresse am Port `port` läuft.
- **public** `Socket(String host,` **int** `port)`
  erzeugt einen Socket und verbindet ihn mit der Anwendung, die auf dem Rechner mit dem Host-Namen bzw. der IP-Adresse `host` am Port `port` läuft. Für den String `host` wird zuvor eine DNS-Anfrage zur Bestimmung des `InetAddress`-Objekts durchgeführt.

generiert. Sowohl auf Server- als auch auf Client-Seite kann man über die `Socket`-Objekte auf die entsprechenden Ein- und Ausgabeströme zugreifen, indem die Methoden

- **public** `InputStream getInputStream()`
  liefert einen Byte-Eingabestrom über den Socket.
- **public** `OutputStream getOutputStream()`
  liefert einen Byte-Ausgabestrom über den Socket.

eingesetzt werden (vgl. auch Abbildung 20.4). Schließen lässt sich ein Sockets mit der Methode

- **public void** `close()`
  schließt den Socket und die zugehörigen Ströme.

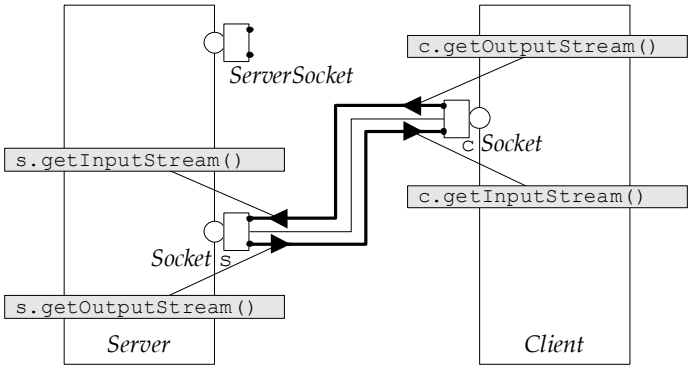

Abbildung 20.4: Datenströme über Sockets auf Server- und Client-Seite

20.2.2 Ein einfacher Server

Wir wollen uns nun mit einem einfachen Beispiel beschäftigen, das in der Konstellation aus den Abbildungen 20.3 und 20.4 die linke Seite, also den Server realisiert. Unser Server soll für eine Client-Anfrage nach der aktuellen Uhrzeit bzw. nach dem aktuellen Datum zu Verfügung stehen. Dabei soll der Client, der sich anmeldet, zunächst gefragt werden, ob er die Uhrzeit oder das Datum wissen möchte. Je nachdem, was er antwortet, wird ihm die entsprechende Information übermittelt. Danach soll unser Server bereits beendet sein.
Entsprechend den zu Beginn von Abschnitt 20.2 aufgeführten Schritten verwenden wir in unserer Server-Applikation

```
1  import java.io.*;
2  import java.net.*;
3  class DateTimeServer {
4    public static void main(String[] args) {
5      try {
6          int port = Integer.parseInt(args[0]);          // Port-Nummer
7          ServerSocket server = new ServerSocket(port);  // Server-Socket
8          System.out.println("DateTimeServer laeuft");   // Statusmeldung
9          Socket s = server.accept();    // Client-Verbindung akzeptieren
10         new DateTimeProtokoll(s).transact();     // Protokoll abwickeln
11     } catch (ArrayIndexOutOfBoundsException ae) {
12         System.out.println("Aufruf: java DateTimeServer <Port-Nr>");
13     } catch (IOException e) {
14         e.printStackTrace();
15     }
16   }
17 }
```

zunächst die als Kommandozeilenparameter geforderte Port-Nummer, um einen Server-Socket zu erzeugen. Wir bestätigen durch eine Konsolenausgabe, dass der Server läuft, und rufen danach die Methode `accept` auf, die einen Kontaktversuch durch einen Client akzeptiert und einen entsprechenden Socket `s` für

die Kommunikation erzeugt. Die Erzeugung der benötigten Datenströme und die Abwicklung des Protokolls erledigen wir, indem wir ein Objekt der Klasse `DateTimeProtokoll` erzeugen und dessen Methode `transact` aufrufen. Unsere Prokoll-Klasse haben wir wie folgt gestaltet:

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.*;
4  import java.text.*;
5  class DateTimeProtokoll {
6    static SimpleDateFormat   // Formate fuer den Zeitpunkt
7      time = new SimpleDateFormat("'Es ist gerade 'H'.'mm' Uhr.'"),
8      date = new SimpleDateFormat("'Heute ist 'EEEE', der 'dd.MM.yy");
9
10   Socket s;                   // Socket in Verbindung mit dem Client
11   BufferedReader vomClient;   // Eingabe-Strom vom Client
12   PrintWriter zumClient;      // Ausgabe-Strom zum Client
13
14   public DateTimeProtokoll (Socket s) {  // Konstruktor
15     try {
16         this.s = s;
17         vomClient = new BufferedReader(
18                           new InputStreamReader(
19                                 s.getInputStream()));
20         zumClient = new PrintWriter(
21                           s.getOutputStream(),true);
22     } catch (IOException e) {
23         System.out.println("IO-Error");
24         e.printStackTrace();
25     }
26   }
27   public void transact() {     // Methode, die das Protokoll abwickelt
28     System.out.println("Protokoll gestartet");
29     try {
30         zumClient.println("Geben Sie DATE oder TIME ein");
31         String wunsch = vomClient.readLine();   // v. Client empfangen
32         Date jetzt = new Date();                // Zeitpunkt bestimmen
33                        // vom Client empfangenes Kommando ausfuehren
34         if (wunsch.equalsIgnoreCase("date"))
35           zumClient.println(date.format(jetzt));
36         else if (wunsch.equalsIgnoreCase("time"))
37           zumClient.println(time.format(jetzt));
38         else
39           zumClient.println(wunsch +" ist als Kommando unzulaessig!");
40         s.close();       // Socket (und damit auch Stroeme) schliessen
41     } catch (IOException e) {
42         System.out.println("IO-Error");
43     }
44     System.out.println("Protokoll beendet");
45   }
46 }
```

Der Konstruktor, dem jeweils der Socket übergeben wird, ist dafür zuständig, die benötigten Ein- und Ausgabeströme zu erzeugen. Dabei greifen wir auf die beiden Methoden `getInputStream` und `getOutputStream` des `Socket`-Objekts

zurück. Zum Lesen der vom Client geschickten Informationen verwenden wir einen gepufferten Zeichenstrom, während wir die Mitteilungen des Servers an den Client über ein `PrintWriter`-Objekt verschicken, bei dem wir das automatische Flushing für `println`-Aufrufe aktivieren.
In der Methode `transact` wird zunächst auf der Konsole des Servers gemeldet, dass das Protokoll gestartet wurde, danach werden Informationen über die möglichen Kommandos an den Client geschickt. Nachdem das gewünschte Kommando vom Client empfangen wurde, wird ein `Date`-Objekt erzeugt, abhängig vom gewählten Kommando mit Hilfe der `SimpleDateFormat`-Objekte formatiert und schließlich an den Client geschickt. Nach einer weiteren Konsolenmeldung über das Ende des Protokolls (und damit in unserem Fall auch des Servers) wird lediglich noch der Socket (und damit gleichzeitig dessen Ein- und Ausgabeströme) geschlossen.
Starten wir unseren Server unter Verwendung des Ports 2222, so erhalten wir auf dem Konsolenfenster zunächst folgenden Ablauf:

Konsole

```
> java DateTimeServer 2222
DateTimeServer laeuft
```

Unser Server ist also bereit, eine Client-Anfrage zu akzeptieren. Da wir bisher über kein eigenes Client-Programm verfügen, könnten wir unseren Server beispielsweise mit einem üblichen **Telnet-Programm**, wie es auf den meisten Rechnerplattformen zur Verfügung steht, testen. Dazu müssen wir ein weiteres Konsolenfenster öffnen und dort das Kommando `telnet`, gefolgt von Rechnername und Port des Servers, als notwendige Parameter eingeben. Den Rechnernamen unseres eigenen Rechners, auf dem ja unser Server läuft, können wir dabei auch entweder als `localhost` (ein standardmäßig festgelegter Alias-Name) oder als `127.0.0.1` (eine standardmäßig festgelegte IP-Adresse) angeben. Wir erhalten dann in unserem zweiten Konsolenfester zunächst den Ablauf

Konsole

```
> telnet localhost 2222
Geben Sie DATE oder TIME ein
```

während in unserem ersten Konsolenfenster mittlerweile eine Zeile hinzugekommen ist und somit

Konsole

```
> java DateTimeServer 2222
DateTimeServer laeuft
Protokoll gestartet
```

zu lesen steht. Geben wir auf der Client-Seite nun das Kommando `time` ein, kann der Rest unseres Protokolls abgearbeitet werden, und auf der Konsole des Servers steht schließlich

Konsole

```
> java DateTimeServer 2222
DateTimeServer laeuft
Protokoll gestartet
Protokoll beendet
```

während im Telnet-Fenster nunmehr

Konsole

```
> telnet localhost 2222
Geben Sie DATE oder TIME ein
time
Es ist gerade 13.01 Uhr.

Verbindung zu Host verloren.
```

zu lesen ist, womit angezeigt wird, dass die Verbindung zum Telnet-Client unterbrochen wurde, da der Server nach Übermittlung der Zeitangabe beendet war.

20.2.3 Ein einfacher Client

Anstelle des Telnet-Clients könnten wir natürlich auch einen eigenen, spezialisierten Client verwenden, der genau den Bedürfnissen einer Kommunikation mit unserem `DateTimeServer`-Programm angepasst ist. Eine entsprechende Klasse haben wir als

```
1  import java.net.*;
2  import java.io.*;
3
4  class DateTimeClient {
5    public static void main(String[] args) {
6      String hostName = "";  // Rechner-Name bzw. -Adresse
7      int port;              // Port-Nummer
8      Socket c = null;       // Socket fuer die Verbindung zum Server
9
10     try {
11         hostName = args[0];
12         port = Integer.parseInt(args[1]);
13         c = new Socket(hostName, port);
14
15         BufferedReader vomServer = new BufferedReader(
16                                        new InputStreamReader(
17                                            c.getInputStream()));
18         PrintWriter zumServer = new PrintWriter(
19                                      c.getOutputStream(),true);
20
21         BufferedReader vonTastatur = new BufferedReader(
22                                          new InputStreamReader(
23                                                System.in));
24
25         // Protokoll abwickeln
```

```
26          System.out.println("Server " + hostName +":"+ port + " sagt:");
27          String text = vomServer.readLine(); // vom Server empfangen
28          System.out.println(text);          // auf die Konsole schreiben
29          text = vonTastatur.readLine();     // von Tastatur lesen
30          zumServer.println(text);           // zum Server schicken
31          text = vomServer.readLine();       // vom Server empfangen
32          System.out.println(text);          // auf die Konsole schreiben
33
34          // Socket (und damit auch Stroeme) schliessen
35          c.close();
36       } catch (ArrayIndexOutOfBoundsException ae) {
37          System.out.println("Aufruf:");
38          System.out.println("java DateTimeClient <HostName> <PortNr>");
39       } catch (UnknownHostException ue) {
40          System.out.println("Kein DNS-Eintrag fuer " + hostName);
41       } catch (IOException e) {
42          System.out.println("IO-Error");
43       }
44    }
45 }
```

implementiert, in deren `main`-Methode wir die zu Beginn von Abschnitt 20.2 aufgeführten Schritte für eine Client-Applikation realisiert haben. Wir erzeugen darin zunächst einen Socket unter Verwendung des Rechnernamens und der Port-Nummer, die als Kommandozeilenparameter übergeben werden. Danach erzeugen wir die benötigten Ein- und Ausgabeströme in Verbindung mit dem Server (dabei greifen wir wieder auf die beiden Methoden `getInputStream` und `getOutputStream` des `Socket`-Objekts zurück) und einen gepufferten Eingabestrom für Tastatureingaben. Danach wickeln wir das Protokoll ab, indem wir die vom Server gelesenen Informationen auf die Konsole schreiben, ein Kommando von der Tastatur einlesen, dieses zum Server schicken, schließlich die Antwort des Servers lesen und ebenfalls auf die Konsole ausgeben.
Starten wir unseren Server erneut an Port 2222 und rufen dann diesen einfachen Client auf, so ergibt sich der Ablauf

Konsole

```
> java DateTimeClient localhost 2222
Server localhost:2222 sagt:
Geben Sie DATE oder TIME ein
date
Heute ist Samstag, der 21.06.03
```

auf der Konsole des Clients, der unmittelbar danach auch beendet ist.

20.2.4 Ein Server für mehrere Clients

Unser einfacher Server aus Abschnitt 20.2.2 ist so gestaltet, dass er nach seinem Start seinen Dienst lediglich einem einzigen Client zur Verfügung stellt und danach beendet ist. Wollen wir diesen Dienst mehreren Clients zur Verfügung stel-

len, so könnten wir in unserer Klasse `DateTimeServer` ganz einfach die Anweisungen in Zeile 9 und 10 in eine Schleife packen, die mehrmals oder sogar uendlich oft durchlaufen wird. Allerdings müsste dann jeweils das komplette Protokoll für einen Client abgewickelt sein, bevor der nächste Client sich an den Server wenden kann. Mit dem in Kapitel 18 Erlernten können wir allerdings auch dieses kleine Problem recht einfach lösen, indem wir unsere Protokoll-Klasse zu einem Thread machen.
Als Server, der von mehreren Clients genutzt werden kann, verwenden wir daher die Klasse

```
1  import java.io.*;
2  import java.net.*;
3  class DateTimeMultiServer {
4    public static void main(String[] args) {
5      try {
6          int port = Integer.parseInt(args[0]);          // Port-Nummer
7          ServerSocket server = new ServerSocket(port); // Server-Socket
8          System.out.println("DateTimeServer laeuft");  // Statusmeldung
9          while (true) {
10           Socket s = server.accept(); // Client-Verbindung akzeptieren
11           new DateTimeDienst(s).start();              // Dienst starten
12         }
13     } catch (ArrayIndexOutOfBoundsException ae) {
14         System.out.println("Aufruf: java DateTimeServer <Port>");
15     } catch (IOException e) {
16         e.printStackTrace();
17     }
18   }
19 }
```

in der wir nach Erzeugung des Server-Sockets in einer Endlosschleife jeweils die nächste Client-Verbindung akzeptieren und für den zugehörigen Socket einen Thread erzeugen und starten, der das eigentliche Protokoll abwickelt. Von der Methode `start` eines Thread-Objekts wissen wir ja, dass sie dafür sorgt, dass dessen `run`-Methode nebenläufig ausgeführt wird und dass sie danach sofort beendet ist. Daher kann das Server-Programm unverzüglich zum nächsten Schleifendurchlauf übergehen und eine weitere Client-Anfrage bearbeiten, noch bevor das Protokoll für den ersten Client komplett abgewickelt ist. Abbildung 20.5 stellt diese Situation grafisch dar. Während Client 1 mit dem Server kommuniziert, nimmt Client n mit dem Server gerade Kontakt auf (gestrichelte Linie) und erhält von der `accept`-Methode einen Socket für die Kommunikation zugewiesen.
Zur Vervollständigung unserer mehrfädigen Server-Implementierung müssen wir nun noch unsere ursprüngliche Protokoll-Klasse `DateTimeProtokoll` in eine Thread-Klasse verwandeln. Dazu lassen wir die Klasse

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.*;
4  import java.text.*;
5  class DateTimeDienst extends Thread {
6    static SimpleDateFormat     // Formate fuer den Zeitpunkt
```

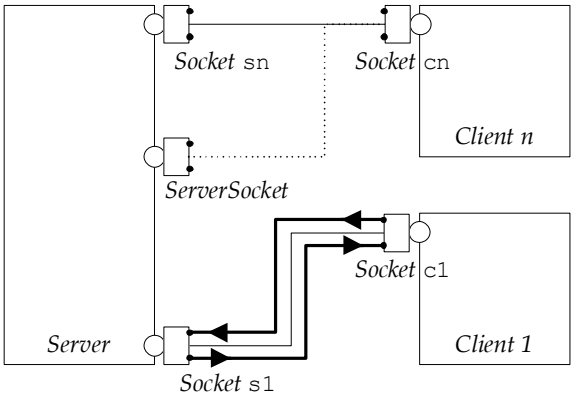

Abbildung 20.5: Ein Server behandelt mehrere Clients gleichzeitig

```
7       time = new SimpleDateFormat("'Es ist gerade 'H'.'mm' Uhr.'"),
8       date = new SimpleDateFormat("'Heute ist 'EEEE', der 'dd.MM.yy");
9     static int anzahl = 0;      // Anzahl der Clients insgesamt
10    int nr = 0;                 // Nummer des Clients
11    Socket s;                   // Socket in Verbindung mit dem Client
12    BufferedReader vomClient;   // Eingabe-Strom vom Client
13    PrintWriter zumClient;      // Ausgabe-Strom zum Client
14
15    public DateTimeDienst (Socket s) {  // Konstruktor
16      try {
17          this.s = s;
18          nr = ++anzahl;
19          vomClient = new BufferedReader(
20                            new InputStreamReader(
21                                  s.getInputStream()));
22          zumClient = new PrintWriter(
23                            s.getOutputStream(),true);
24      } catch (IOException e) {
25          System.out.println("IO-Error bei Client " + nr);
26          e.printStackTrace();
27      }
28    }
29    public void run() {  // Methode, die das Protokoll abwickelt
30      System.out.println("Protokoll fuer Client " + nr + " gestartet");
31      try {
32        while (true) {
33          zumClient.println("Geben Sie DATE, TIME oder QUIT ein");
34          String wunsch = vomClient.readLine(); // vom Client empfangen
35          if (wunsch == null || wunsch.equalsIgnoreCase("quit"))
36            break;                              // Schleife abbrechen
37          Date jetzt = new Date();              // Zeitpunkt bestimmen
38                        // vom Client empfangenes Kommando ausfuehren
39          if (wunsch.equalsIgnoreCase("date"))
40            zumClient.println(date.format(jetzt));
41          else if (wunsch.equalsIgnoreCase("time"))
```

```
42            zumClient.println(time.format(jetzt));
43          else
44            zumClient.println(wunsch+ "ist als Kommando unzulaessig!");
45        }
46        s.close();        // Socket (und damit auch Stroeme) schliessen
47      } catch (IOException e) {
48        System.out.println("IO-Error bei Client " + nr);
49      }
50      System.out.println("Protokoll fuer Client " + nr + " beendet");
51    }
52  }
```

einfach von der Klasse `Thread` erben und in ihrem Konstruktor Nummern für die erzeugten Objekte vergeben. Außerdem wandern die Anweisungen für die Abwicklung des Protokolls aus der Methode `transact` nun in die `run`-Methode, so dass sie nebenläufig abgearbeitet werden können. Diese Anweisungen haben wir zusätzlich in eine Endlosschleife verpackt, so dass ein Client die Kommandos DATE und TIME auch mehrfach senden kann, bevor er die Verbindung mit QUIT wieder abbricht.
Starten wir nun unseren neuen Server `DateTimeMultiServer` an Port 3333, so können mehrere Clients auf ihn zugreifen, um Zeit- oder Datumsabfragen durchzuführen, was im Konsolenfenster z. B. wie folgt protokolliert werden könnte:

Konsole

```
> java DateTimeMultiServer 3333
DateTimeServer laeuft
Protokoll fuer Client 1 gestartet
Protokoll fuer Client 2 gestartet
Protokoll fuer Client 2 beendet
Protokoll fuer Client 3 gestartet
Protokoll fuer Client 3 beendet
Protokoll fuer Client 1 beendet
Protokoll fuer Client 4 gestartet
IO-Error bei Client 4
Protokoll fuer Client 4 beendet
```

Dabei haben wir die ersten drei Client-Anfragen mit dem Telnet-Client und Anfrage 4 mit unserer Klasse `DateTimeClient` durchgeführt. Wie wir sehen, tritt bei Client 4 ein Fehler auf, was dadurch zustande kommt, dass das relativ spezielle Protokoll des Clients nicht so richtig mit dem des Servers zusammenspielt. Es wird nämlich nur eine einzige Zeit- oder Datumsangabe angefordert und danach abgebrochen. Im nächsten Abschnitt wollen wir uns daher noch mit einem alternativen Client-Programm beschäftigen.

20.2.5 Ein Mehrzweck-Client

Um auch bei anderen Servern Anfragen durchführen zu können, wollen wir nun noch ein etwas allgemeineres Client-Programm entwerfen, in dem wir einfach je-

weils im Wechsel alle Daten, die der Server übermittelt, empfangen und anschließend ein Kommando (bzw. eine Zeile) von Tastatur einlesen und an den Server schicken. Diese Vorgänge wiederholen wir, bis das Kommando QUIT eingegeben wird. In unserer Klasse

```
1  import java.net.*;
2  import java.io.*;
3  public class MyClient {
4    // liest alle vom Server geschickten Daten
5    static void zeigeWasKommt(BufferedReader sin) throws IOException {
6      String str = null;
7      try {
8        while ((str = sin.readLine()) != null)
9          System.out.println(str);
10     }
11     catch (SocketTimeoutException sto) {
12     }
13   }
14   static void zeigePrompt() {
15     System.out.print("> ");
16     System.out.flush();
17   }
18   public static void main(String[] args) {
19     try {
20       System.out.println("Client laeuft. Beenden mit QUIT");
21       Socket c = new Socket(args[0], Integer.parseInt(args[1]));
22       c.setSoTimeout(500); // setze Timeout auf eine halbe Sekunde
23       BufferedReader vomServer = new BufferedReader(
24                                        new InputStreamReader(
25                                              c.getInputStream()));
26       PrintWriter zumServer = new PrintWriter(
27                                     c.getOutputStream(), true);
28       BufferedReader vonTastatur = new BufferedReader(
29                                          new InputStreamReader(
30                                                System.in));
31       String zeile;
32
33       do {
34         zeigeWasKommt(vomServer);
35         zeigePrompt();
36         zeile = vonTastatur.readLine();
37         zumServer.println(zeile);
38       } while(!zeile.equalsIgnoreCase("quit"));
39
40       c.close();         // Socket (und damit auch Stroeme) schliessen
41     } catch (ArrayIndexOutOfBoundsException ae) {
42         System.out.println("Aufruf: java MyClient <Port-Nummer>");
43     } catch (UnknownHostException ux) {
44         System.out.println("Kein DNS-Eintrag fuer " + args[0]);
45     } catch (IOException e) {
46         e.printStackTrace();
47     }
48   }
49 }
```

haben wir daher für das Lesen der vom Server empfangenen Daten eine Methode `zeigeWasKommt` definiert, die in einer Schleife aus dem Eingabestrom vom Server liest und die gelesenen Zeilen auf dem Konsolenfenster ausgibt. Allerdings wird diese Schleife erst abgebrochen, wenn das Stromende erreicht ist, also genau genommen dann, wenn die Verbindung zum Server beendet wird. Wenn der Server gerade keine Daten schickt, muss die Methode `readLine` jeweils warten. Dies hat natürlich zur Folge, dass wir nach einem Aufruf der Methode `zeigeWasKommt` keine Möglichkeit haben, zwischen den einzelnen Lesevorgängen auch einmal etwas zum Server zu schicken.
Dieses Problem kann man aber leicht dadurch beheben, dass man den Socket, über den die Kommunikation läuft, so einstellt, dass er nicht „ewig" auf Server-Daten wartet, sondern den Lesevorgang nach einer festgelegten Zeit ohne jegliche Datenübermittlung abbricht. Dazu stellt die Klasse `Socket` die Methode

- **`public void`** `setSoTimeout(`**`int`** `timeout)`
  aktiviert (für `timeout` > 0) bzw. deaktiviert (für `timeout` = 0) den Socket-Timeout, so dass bei einer Leseoperation über den Eingabestrom des Sockets maximal `timeout` Millisekunden auf Daten gewartet wird. Sollte diese Zeit überschritten werden, wird eine Ausnahme vom Typ `SocketTimeoutException` geworfen. Ist `timeout` = 0, so wird unendlich lange gewartet.

zur Verfügung, die jeweils vor der ersten Leseoperation aufgerufen werden muss. In der `main`-Methode unserer Klasse `MyClient` haben wir unmittelbar nach der Erzeugung des Sockets den Timeout auf eine halbe Sekunde eingestellt. Aus diesem Grund arbeiten wir in der Methode `zeigeWasKommt` mit einem Catch-Block, der die Socket-Timeout-Ausnahme abfängt, ohne etwas zu tun.
In der **`do`**-Schleife unseres Client-Programms rufen wir somit jeweils die Methode `zeigeWasKommt` auf, um alles angezeigt zu bekommen, was der Server bis zum Timeout geschickt hat, und geben danach mit der Methode `zeigePrompt` ein >-Zeichen aus, um anzuzeigen, dass jetzt eine Eingabe erfolgen kann, die anschließend an den Server geschickt wird. Wenn wir unseren Client aufrufen, bemerken wir auch die kurze Verzögerung, mit der das Prompt-Zeichen auf dem Konsolenfenster erscheint, das z. B. beim Zugriff auf unseren Server `DateTimeMultiServer` auf unserem lokalen Rechner an Port 3333 wie folgt aussehen könnte:

Konsole

```
> java MyClient localhost 3333
Client laeuft. Beenden mit QUIT
Geben Sie DATE, TIME oder QUIT ein
> time
Es ist gerade 11.44 Uhr.
Geben Sie DATE, TIME oder QUIT ein
> date
Heute ist Sonntag, der 22.06.03
Geben Sie DATE, TIME oder QUIT ein
```

```
> year
year ist als Kommando unzulaessig!
Geben Sie DATE, TIME oder QUIT ein
> quit
```

Im Rahmen der Übungsaufgaben, werden Sie sehen, dass dieses einfache, aber doch recht universelle Client-Programm `MyClient` problemlos auch zum Test anderer Server-Typen eingesetzt werden kann.

20.3 Wissenswertes über URLs

Bereits in Kapitel 17 haben wir uns im Zusammenhang mit Applets, die den Browser dazu veranlassen sollen, eine bestimmte Webseite anzuzeigen, mit der Klasse `URL` beschäftigt. Wir wissen daher bereits, dass ein Objekt dieser Klasse jeweils die Adresse eines Dokuments im Internet (die URL des Dokuments) darstellt. Solche URLs sind in zweifacher Hinsicht im Rahmen der Netzwerk-Programmierung von Bedeutung. Zum einen erlaubt Java auch den direkten Zugriff auf WWW-Dokumente über ihre URL (d. h. wir müssen nicht unbedingt die Kommunikation über Sockets explizit programmieren), zum anderen können wir die URL aber auch benutzen, um innerhalb eines Applets eine explizite Netzwerkverbindung zu programmieren (d. h. wir können aus der URL die Rechneradresse bestimmen, zu der wir aufgrund der Applet-Sicherheitsrestriktionen überhaupt eine Verbindung aufbauen dürfen). Mit diesen beiden Aspekten wollen wir uns zum Schluss des Kapitels über Netzwerk-Programmierung noch kurz auseinandersetzen.

20.3.1 Client/Server-Kommunikation über URLs

Zur eindeutigen Adressierung von Dokumenten im World Wide Web hat man sich auf folgendes Format

PROTOKOLL://RECHNERNAME:PORT/DOKUMENTNAME

für eine URL geeinigt, wobei der Doppelpunkt und die Angabe des Ports (:PORT) optional sind, da in der Regel die Portnummer bereits durch das angegebene Protokoll festgelegt ist. Die URL

```
http://www.hanser.de/computer/index.htm
```

bezeichnet somit das Protokoll `http`, den Rechner `www.hanser.de` und die Datei `computer/index.htm` (also die Datei `index.htm` im Unterverzeichnis `computer`). Weitere typischerweise in URLs genannte Protokolle sind zum Beispiel `ftp`, wenn Daten auf einem FTP-Server angesprochen werden sollen, oder `file`, wenn ein Dokument auf dem lokalen Rechner adressiert werden soll.

Die Klasse `URL` aus dem Paket `java.net` stellt für ihre Objekte unter anderem die Methode

- **`public final`** `InputStream openStream()`
  öffnet eine Verbindung zur URL und liefert einen Eingabestrom über diese Verbindung als Ergebnis zurück.

zur Verfügung. Damit lässt sich beispielsweise sehr leicht (ohne explizite Programmierung von Sockets) ein Programm schreiben, das den Inhalt eines Webdokuments als reinen Text auf dem Konsolenfenster ausgeben kann:

```
1  import java.net.*;
2  import java.io.*;
3  public class LiesURL {
4    public static void main(String[] args) {
5      try {
6          URL u = new URL(args[0]);
7          BufferedReader in = new BufferedReader(
8                                      new InputStreamReader(
9                                             u.openStream()));
10         String zeile;
11         while ((zeile = in.readLine()) != null)
12           System.out.println(zeile);
13         in.close();
14     } catch (ArrayIndexOutOfBoundsException ae) {
15         System.out.println("Aufruf: java LiesURL <URL>");
16     } catch (MalformedURLException me) {
17         System.out.println(args[0] + " ist keine zulaessige URL");
18     } catch (IOException e) {
19         e.printStackTrace();
20     }
21   }
22 }
```

Neben der Methode `openStream` steht `URL`-Objekten auch eine Methode `openConnection` zur Verfügung, die ebenfalls eine Verbindung herstellt und als Objekt der Klasse `URLConnection` zurückliefert. Mit den Instanzmethoden `getInputStream` und `getOutputStream` kann man über diese Verbindung auf Ein- und Ausgabeströme zugreifen, um sowohl lesend als auch schreibend (zum Beispiel bei interaktiven Webseiten) mit der URL zu kommunizieren.

20.3.2 Netzwerkverbindungen in Applets

Wenn wir uns an die in Abschnitt 17.5 behandelten Sicherheitseinschränkungen bei Applets erinnern, so wissen wir, dass Java-Applets keine Verbindungen zu einem anderen Rechner (mit Ausnahme des Rechners, von dem das Applet geladen wurde) aufnehmen dürfen. Wir können somit in einem Applet einen Netzwerk-Socket nur zu einem anderen Server-Programm auf dem Rechner, der das Applet ausgeliefert hat, erzeugen. Zur Bestimmung dieses Rechners können wir zunächst mit der Methode `getCodeBase` oder `getDocumentBase` unserer Applet-Klasse

die URL des Verzeichnisses, in der das Applet oder die HTML-Seite liegt, bestimmen (siehe auch Abschnitt 17.5). Danach können wir die Instanzmethode

- **public** `String getHost()`
  liefert den Rechnernamen aus der URL.

der Klasse `URL` anwenden, um den Rechnernamen zu erhalten. Wir benötigen nun nur noch die Information über den Port, mit dem wir in Verbindung treten wollen, um einen Socket zu erzeugen. In unserer Applet-Klasse

```
1  import javax.swing.*;
2  import java.net.*;
3  import java.io.*;
4  import java.applet.*;
5  public class DateTimeApplet extends JApplet {
6    public void init() {
7      try {
8          Socket socket = new Socket(this.getCodeBase().getHost(), 7777);
9          BufferedReader in = new BufferedReader(
10                                 new InputStreamReader(
11                                       socket.getInputStream()));
12         PrintWriter out = new PrintWriter(
13                                socket.getOutputStream(), true);
14         in.readLine();
15         out.println("date");
16         String s = in.readLine();
17         getContentPane().add(new JLabel(s,JLabel.CENTER));
18     } catch (IOException e) {
19         String s = "Verbindung zum DateTimeServer fehlgeschlagen!";
20         getContentPane().add(new JLabel(s,JLabel.CENTER));
21     }
22   }
23 }
```

haben wir ein Applet realisiert, das beim Start den auf dem gleichen Rechner an Port 7777 laufenden Server `DateTimeMultiServer` kontaktiert und die übermittelte Datumsangabe in einem Label anzeigt.

20.4 Übungsaufgaben

Aufgabe 20.1

Schreiben Sie ein Java-Programm, das ein einfaches Online-CD-Archiv als Server realisiert. Das Archiv ist dabei einfach eine Ansammlung von Textdateien (gespeichert im Verzeichnis `cdArchiv`), wobei jede Datei eine Aufzählung der Stücke (Tracks) auf der entsprechenden CD enthält. Die Klasse `CDServer`, deren `main`-Methode die Port-Nummer als Kommandozeilenargument übergeben bekommt und einen `ServerSocket` mit diesem Port verbindet, soll in einer Endlosschleife für jeden Client, der eine Verbindung aufbaut, einen `CDVerbindung`-Thread erzeugen und starten. Dieser Thread soll bei seiner Erzeugung die Ströme zum

Client öffnen und alle hergestellten Verbindungen und die darüber abgewickelten Aktionen auf dem Bildschirm protokollieren.
Die vom Client geschickten Kommandos sollen wie folgt bearbeitet werden:

- Sendet der Client das Kommando `list`, so ist unter Verwendung der Methode `list()` der Klasse `File` nur der Inhalt des Verzeichnisses `cdArchiv` an den Client zu schicken.
- Sendet der Client das Kommando `tracks`, gefolgt von einem CD-Titel, so ist die entsprechende Datei im Verzeichnis `cdArchiv` zu öffnen und deren Inhalt zu lesen und an den Client zu schicken.

Nachfolgend beispielhafte Konsolen-Ausgaben auf Server-Seite und Konsolen-Dialog auf Client-Seite:

Konsole

```
CDServer wartet auf Port 8888
[localhost/127.0.0.1:1587: neue Verbindung]
[localhost/127.0.0.1:1587: sende Verzeichnis der CDs]
[localhost/127.0.0.1:1587: sende Tracks der CD Yes-Magnification]
[localhost/127.0.0.1:1587: Verbindung unterbrochen]
```

Konsole

```
Client gebunden an lokalen Port: 1587
> list
Aha-HowCanISleepWithYourVoiceInMyHead
Evanescence-Fallen
MikeOldfield-TubularBells2003
Reamonn-BeautifulSky
Yes-Magnification
> tracks Yes-Magnification
1. Magnification            2. Spirit of survival
3. Don't go                 4. Give love each day
5. Can you imagine          6. We agree
7. Soft as a dove           8. Dreamtime
9. In the presence of      10. Time is time
> quit
```

Aufgabe 20.2

Schreiben Sie einen Server, der jedem Client, der mit ihm eine Verbindung aufbaut, die Möglichkeit gibt, Geldbeträge von DM in EUR bzw. von EUR in DM umrechnen zu lassen. Auf der Server-Konsole könnte z. B. Folgendes ablaufen:

Konsole

```
Der Server laeuft.
Server beenden durch Eingabe von SHUTDOWN.
Neuer Client wird bearbeitet.
```

```
Neuer Client wird bearbeitet.
SHUTDOWN
Der Server wird nun nach Abarbeitung des
naechsten Clients automatisch beendet.
Neuer Client wird bearbeitet.
Der Server ist beendet.
```

Auf Client-Seite könnte ein Dialog mit dem Server wie folgt aussehen:

Konsole

```
Der Client laueft und kann mit 'quit' beendet werden
Welche Waehrung wollen Sie eingeben (DM oder EUR)?
> DM
Welchen Wert wollen Sie umrechnen?
> 100
Wert in EUR: 51.12918811962185
Darf's noch eine Umrechnung sein?
> ja
Welche Waehrung wollen Sie eingeben (DM oder EUR)?
> EUR
Welchen Wert wollen Sie umrechnen?
> 100
Wert in DM: 195.583
Darf's noch eine Umrechnung sein?
> nein
> quit
```

Ihre Implementierung sollen Sie in die drei Klassen (`EuroServer`, `SteuerDienst`, `EuroThread`) aufteilen. Die `main`-Methode der Klasse `EuroServer` soll beim Aufruf die zu verwendende Portnummer für die Erzeugung des Server-Socket-Objekts übergeben bekommen. Danach soll ein Dienst für die Server-Steuerung (genau genommen für das Beenden des Servers nach Ende des nächsten Client-Dialogs) in Form eines `SteuerDienst`-Objekts aktiviert werden. Im Anschluss daran soll in einer Schleife für jeden Client, der eine Verbindung aufbaut, ein `EuroThread`-Objekt erzeugt und gestartet werden.
Die `run`-Methode der Thread-Klasse `SteuerDienst` meldet, dass der Server läuft, und fordert so lange Benutzer-Eingaben an, bis das Kommando `SHUTDOWN` eingelesen wird. Danach sorgt sie dafür, dass die Schleife in der `main`-Methode des Euro-Servers beendet wird.
In der `run`-Methode der Klasse `EuroThread` sollen Sie das Protokoll mit dem Client implementieren. Für die Währungsumrechnung können Sie auf die bekannte Klasse `EuroConverter` aus Aufgabe 15.6 zurückgreifen.

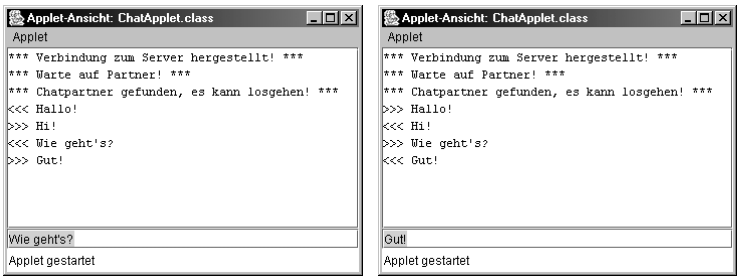

Abbildung 20.6: Zwei Applet-Chat-Clients aus Aufgabe 20.3

Aufgabe 20.3

Entwickeln Sie ein einfaches Chat-System, bestehend aus den drei Klassen `TalkServer` (der Server), `TalkDienst` (die Thread-Klasse, die den Datenaustausch zwischen Clients regelt) und `ChatApplet` (ein Applet, das einen Client mit grafischer Oberfläche realisiert, siehe Abbildung 20.6).
Der Server soll nach seinem Start in einer Endlosschleife auf jeweils zwei Clients warten und zwei Instanzen des `TalkDienst`-Threads erzeugen und starten. Als Argumente für den Konstruktor sollen die beiden Client-Sockets in jeweils vertauschter Reihenfolge übergeben werden. Der `TalkDienst`-Thread soll eine unidirektionale Kommunikation zwischen den beiden Clients ermöglichen, indem er alle Daten von einem Client liest und diese direkt zum anderen Client sendet.
Die Klasse `ChatApplet` soll von `JApplet` erben und das `Runnable`-Interface implementieren. In der init-Methode soll die grafische Oberfläche mit einem Eingabebereich (ein Textfeld) und einem Ausgabebereich (eine nicht editierbare `JTextArea` auf einer `JScrollPane`) aufgebaut, die Verbindung zum TalkServer hergestellt und der mit dem Applet verbundene Thread gestartet werden. Die notwendige Server-Adresse muss dabei mit den Methoden `getCodeBase` und `getHost` ermittelt werden. Der Port kann fest codiert werden.
In der `run`-Methode soll in einer Schleife Zeile für Zeile vom Server gelesen und in den Ausgabebereich ausgegeben werden. In der Methode `destroy` soll dafür gesorgt werden, dass beim Schließen des Applets auch der Thread korrekt beendet wird. In der Ereignisbehandlung für das Eingabe-Textfeld soll der eingegebene Text zum Server geschickt und außerdem zur Kontrolle auch im eigenen Ausgabebereich angezeigt werden.