Programmieren I

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2009/2010

Dateien: Allgemeines

Kaum eine Anwendung im „echten Leben“ kommt ohne Dateien aus, sei es zum Lesen oder Schreiben (oder beides).

Hierbei wird im Wesentlichen zwischen zwei Typen von Dateien unterschieden:

Textdateien enthalten lesbaren Text. Diese kann man in einem normalen Editor (notepad, emacs, . . .) öffnen, lesen und bearbeiten.

Binärdateien enthalten Daten in binärer Form, die nicht zum direkten Lesen durch den Menschen gedacht sind (und zum Teil auch nicht druckbare Zeichen enthalten), sondern zur Verarbeitung durch ein Programm gedacht sind. Beispiele hierfür sind Word-, Excel-Dateien, Grafiken (z. B. GIF, JPG), Musik (z. B. MP3).

Was ist eine Datei?

In erster Linie ist eine Datei eine Menge von Bytes bzw. Zeichen auf einem Speichermedium, in die man Bytes bzw. Zeichen sequenziell schreiben oder daraus lesen kann.

Zweitens sind mit einer Datei (je nach Fähigkeiten des Betriebssystems) Eigenschaften wie Dateiname, Zugriffsrechte (teilweise je nach User unterschiedlich) oder Zugriffs-, Modifikationsdatum verbunden.

In dieser Vorlesung geht es um den ersten Punkt, also das Schreiben und Lesen von Daten.

Eingabeströme

Java stellt zwei einfache, abstrakte Klassen für Eingabeströme zur Verfügung:

`InputStream` stellt rudimentäre Funktionalität zum Lesen von Bytes zur Verfügung.

`Reader` stellt rudimentäre Funktionalität zum Lesen von Zeichen zur Verfügung.

(NB: Dass Lesen von Bytes und Zeichen ein Unterschied ist, liegt daran, dass es in Zeiten von Unicode Zeichen aus mehr als einem Byte bestehen können – *multi byte characters*)

Diese Sicht auf einen Datenstrom zur Eingabe ist allgemein und nicht nur auf Dateien bezogen. Genauso könnte sich also ein `InputStream` auf einen String oder eine Socket-Verbindung im Netzwerk beziehen.

Methoden von `Reader`

`close()` Schließen der Datei, implementiert die abstrakte Methode aus dem Interface `Closeable`.

`read(char cbuf[], int offset, int length)` liest in einen Character-Array `cbuf` ab dem Offset `offset` eine Menge von `length` Zeichen.
Das Lesen ist blockierend, d. h., wenn weniger Zeichen im Eingabestrom zur Verfügung stehen, wird gewartet (es sei denn, es tritt EOF – *end of file* auf).
Rückgabewert: Zahl der gelesenen Zeichen, oder -1 nach EOF
Diese Methode ist abstrakt und muss in eigenen Reader-Klassen implementiert werden!

`read(char cbuf[])` Äquivalent zu `read(cbuf,0,cbuf.length)`.

`read()` liest ein Zeichen ein, wobei dieses als `int` zurückgegeben wird, bzw. -1, falls das Dateiende erreicht wurde.

Weitere Readerklassen

Die Klasse `Reader` als abstrakte Klasse stellt die allgemeine Grundfunktionalität zur Verfügung.

Konkrete Implementierungen von `Reader` sind z. B.:

InputStreamReader Dieser Reader macht aus einem Byte-Stream `InputStream`, der dem Konstruktor übergeben wird, einen Character-Stream.
Beispiel:
```
new InputStreamReader(System.in)
```

FileReader Dieser Reader liest aus einer Datei auf einem Speichermedium (z. B. Festplatte):
```
new FileReader("daten.txt")
```


Weitere Readerklassen (Forts.)

Im Prinzip reicht die Klasse `FileReader` völlig aus, um aus einer Datei lesen zu können ... es ist nur sehr unbequem, dies zeichenweise tun zu müssen.

Beim Lesen von Text-Dateien will man meist zeilenweise lesen. Dies kann mit mit der Klasse `BufferedReader` geleistet werden.

`BufferedReader` ist ein High-Level-Reader, der einen anderen Reader um die Möglichkeit bereichern kann, eine Zeile als String einzulesen.
Hierfür dient dann die Methode `String readLine()`.

Beispiel: Eingabe von der Tastatur

```
import java.io.*;

public class App {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));

        try {
            String zeile = in.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Analyse

Am vorigen Beispiel wird das Konzept der Stream- und Reader-Klassen in Java mit einer Trennung in Low-Level- und High-Level-Klassen deutlich:

- `System.in` ist ein allgemeiner `InputStream`, aus dem Bytes gelesen werden können,
- `InputStreamReader` macht aus diesem Stream einen Low-Level-Reader, aus dem eine Menge von Zeichen gelesen werden kann,
- `BufferedReader` dekoriert diesen Low-Level-Reader mit einem High-Level-Reader, der dann zeilenweises Einlesen erlaubt.

Hierdurch wird eine Trennung zwischen Datenquelle und der Art des Lesens erreicht. Die „Physik“ (Lesen von der Platte, von einem Socket, . . .) steckt im Low-Level-Reader, der Komfort dann im High-Level-Reader.

Beispiel: Eingabe aus einer Datei

Das Lesen aus einer Datei ist nun fast identisch (außer, dass man die `FileNotFoundException` abfangen muss):

```
import java.io.*;
public class App {
    public static void main(String[] args) {
        BufferedReader in;
        try {
            in = new BufferedReader(
                    new FileReader("daten.txt"));
            String zeile = in.readLine();
            System.out.println(zeile);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```


Beispiel: Lesen von CSV-Daten

Für ein komplexeres Beispiel zum Lesen aus Dateien betrachten wir eine Klasse Adresse, die als Attribute Vor- und Nachname, Straße, PLZ und Ort besitzt:

```
public class Adresse {
    private String vorname;
    private String nachname;
    private String strasse;
    private String plz;
    private String ort;

    public Adresse(String vorname, String nachname,
            String strasse, String plz, String ort) {
        this.vorname = vorname;
        this.nachname = nachname;
        this.strasse = strasse;
        this.plz = plz;
        this.ort = ort;
    }
```

Beispiel: Lesen von CSV-Daten (Forts.)

```
    public String getNachname() {
        return nachname;
    }
    public String getOrt() {
        return ort;
    }
    public String getPlz() {
        return plz;
    }
    public String getStrasse() {
        return strasse;
    }
    public String getVorname() {
        return vorname;
    }
}
```


Problemstellung

Tabellen-artige Daten wie Adressen werden häufig in Form von *comma separated values* (CSV) in Text-Dateien gespeichert. Die können dann z. B. einfach von einer Tabellenkalkulation wie Excel verarbeitet werden.

Beispiel für eine CSV-Datei:

```
Karl;Mueller;Rosenstr. 7;12345;Rosendorf
Erich;Mustermann;Musterweg 1;90000;Bad Homburg
Angelika;Schmidt;Annastr. 23;22457;Hamburg
Thomas;von Sokolowski;Rheinstr. 5;64283;Darmstadt
```

Typisch ist, dass die Felder durch ein Zeichen separiert werden, das im Feldinhalt typischer Weise nicht vorkommt (z. B „;“ oder „|“).

Problemstellung (Forts.)

Zum Einlesen von Adressdaten im CSV-Format soll der Klasse `Adresse` nun eine neue statische Methode

```
Adresse[] parse(String dateiname)
```

hinzu gefügt werden.

Diese Methode muss also:

- eine Datei zum Lesen öffnen und mit einem `BufferedReader` dekorieren,
- jede Zeile der Datei in einzelne Felder aufsplitten,
- den Konstruktor von `Adresse` mit diesen Daten aufrufen,
- die Adressen in einen Array packen und zurück geben.

Implementierung von parse

```
public static Adresse[] parse(String dateiname) {
   ArrayList<Adresse> temp = new ArrayList<Adresse>();
   try {
      BufferedReader in = new BufferedReader(
            new FileReader(dateiname));
      String zeile;
      while ( (zeile=in.readLine()) != null) {
         String[] items = zeile.split(";");
         temp.add(new Adresse(items[0],items[1],items[2],
                                    items[3],items[4]));
      }
   } catch (FileNotFoundException e) {
      System.err.println("Kann Datei "+dateiname+" nicht öffnen");
   } catch (IOException e) {
      System.err.println("Lesefehler");
   }
   System.out.println(temp.size()+" Adressen eingelesen");
   return temp.toArray(new Adresse[0]);
}
```


Zwischenfilter

Ein weiterer Vorteil des Konzepts mit Low-Level-Streams und diese dekorierenden High-Level-Streams liegt darin, dass man nach Wunsch Zwischen-Filter einbauen kann.

Diese können transparent Ein- und Ausgabeströme manipulieren. Beispielsweise könnte man so erreichen, dass Daten beim Schreiben verschlüsselt und beim Lesen entschlüsselt werden, ohne dass die Anwendung dies „merkt“, weil dieser Prozess als Filter zwischen Low- und High-Level-Stream bzw. -Reader geschaltet ist.

Implementierung eines Filters

Als kleines Beispiel soll ein Inputfilter implementiert werden, der alle eingelesenen Zeichen in Großbuchstaben umwandelt.

Hierfür wird:

- eine neue Kindklasse von `Reader` definiert,
- diese dekoriert einen anderen Reader, der dem Konstruktor als Referenz übergeben wird,
- die abstrakten Methoden `read(char[], int, int)` und `close()` aus Reader implementiert,
- wobei die Implementierung jeweils auf die entsprechenden Methoden aus dekorierten Reader zurück greift,
- aber beim `read(...)` die eingelesenen Zeichen manipulieren kann.

Beispiel

```
public class UpperInputReader extends Reader {
    private Reader in;
    public UpperInputReader(Reader in) {
        this.in = in;
    }
    @Override
    public int read(char[] cbuf, int offset, int length) throws IO
        int count = in.read(cbuf, offset, length);
        if (count != -1) {
            for (int i=offset; i<offset+length; i++) {
                cbuf[i] = Character.toUpperCase(cbuf[i]);
            }
        }
        return count;
    }
    @Override
    public void close() throws IOException {
        in.close();
    }
}
```


Anwendung

Mit diesem Filter kann man nun einfach benutzen, um ganz transparent alle von der Tastatur gelesenen Zeichen als Großbuchstaben zu lesen:

```
public class App {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(
            new UpperInputReader(
                new InputStreamReader(System.in)));

        String zeile = in.readLine();

        System.out.println(zeile);
    }
}
```


Die Klasse `FilterReader`

Zur Vereinfachung des Schreibens von Filtern gibt es die Klasse `FilterReader`, die einen dem Konstruktor als Referenz übergebenen Reader dekoriert. Hierbei sind alle Methoden von Reader so implementiert, dass sie für den dekorierten Reader ausgeführt werden.

Diese Klasse `FilterReader` ist dazu gedacht, dass eigene Filterklassen von dieser erben und dort einzelne Methoden überladen, um eine Manipulation des Eingabestroms zu erreichen.

Neue Version von UpperInputReader

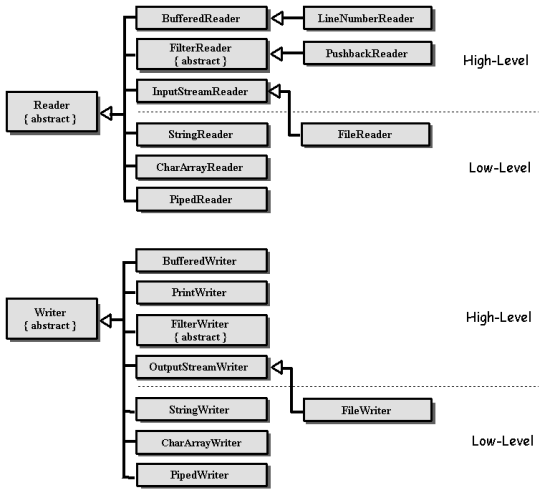
```
public class UpperInputReader extends FilterReader {
    public UpperInputReader(Reader arg0) {
        super(arg0);
    }

    @Override
    public int read(char[] cbuf, int offset, int length)
                        throws IOException {
        int count = in.read(cbuf, offset, length);
        if (count != -1) {
            for (int i=offset; i<offset+length; i++) {
                cbuf[i] = Character.toUpperCase(cbuf[i]);
            }
        }
        return count;
    }

}
```


Ausgabe in Dateien – Allgemeines

Das Konzept der Ausgabe in Java ist analog zur Eingabe:

- Es gibt abstrakte Klassen `OutputStream` und `Writer` mit der Grundfunktionalität zum Schreiben von Bytes bzw. Zeichen.
- Low-Level-Klassen sorgen dafür, dass die abstrakten Methoden implementiert werden. Bei Dateien sind dies `FileOutputStream` und `FileWriter`,
- High-Level-Klassen erweitern die Grundfunktionalität,
- `OutputStreamWriter` dient als Brücke, um aus einem Byte-Ausgabestrom einen Character-Ausgabestrom in Form eines Writers zu machen.

Schaubild: Reader- und Writer-Klassen



(aus: Friedrich Esser, Java 2 – Designmuster und Zertifizierungswissen, Galileo Computing)

BufferedWriter und PrintWriter

Im Allgemeinen wird man einen Writer mit einer der Klassen `BufferedWriter` oder `PrintWriter` dekorieren.

Diese bieten insbesondere die Möglichkeit, mit der Methode `println(...)` zeilenweise Daten auszugeben.

`PrintWriter` ermöglicht zusätzlich mit der Methode `format(...)` eine formatierte Ausgabe.

Beispiel

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

public class App {
    public static void main(String[] args) {
        try {
            PrintWriter out = new PrintWriter(
                        new FileWriter("daten.txt"));

            for (double x=0; x<1; x+=.01) {
                out.format("%8.2f %8.2f\n", x, Math.sin(x));
            }

            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```