

Ein- und Ausgabe mit Java

Michael Wiedau ¹
math.- techn. Assistent/Informatik (IHK)

14. Juni 2004

¹Betreuung durch Frau Dr. Sonja Niessen, Rechenzentrum der RWTH Aachen

Inhaltsverzeichnis

1 Streamkonzept in Java	2
1.1 Die Klassenhierarchie	2
2 Eingabe	4
2.1 Das Fundament: Die Basisklasse Reader	4
2.2 Ab in die Datei: InputStreamReader und FileReader	5
2.3 Sehr nützlich: StringReader und CharArrayReader	5
2.4 Geschachtelt: Andere Reader	6
2.5 Schaubild	8
3 Ausgabe	9
3.1 Die abstrakte Basisklasse Writer	9
3.2 Ab in die Datei: Die Klasse FileWriter	9
3.3 Hier geblieben: Die Klassen StringWriter und CharArrayWriter .	10
3.4 Java kann alles: Weitere nützliche Ausgabeklassen	11
3.5 Selbstgemacht: FileWriter zum Erweitern	11
3.6 Schaubild	14
4 Fazit	15

Kapitel 1

Streamkonzept in Java

Viele Programmiersprachen bieten die Möglichkeit, Daten in Form von sogenannten Streams zu behandeln, d.h. Streams zu lesen und zu erzeugen. Ein Stream ist dabei nichts anderes, als eine Menge von Daten "unterwegs" von einer Quelle zu einem Ziel. Der entscheidende Vorteil dieses Konzepts besteht darin, dass es ermöglicht, von der Art der Daten zu abstrahieren. So kann es dem Zielobjekt gleich sein, woher die Daten stammen. Umgekehrt gilt auch, dass sich ein Quellobjekt nicht darum kümmern muss, was das Zielobjekt mit diesen Daten anstellt (einer solchen Quelle kann es z.B. egal sein, ob sie Daten an den Druckertreiber oder eine Datei verschickt, solange der richtige Stream gewählt wird).

Alle Ein- und Ausgaben werden in Java mit Streams realisiert. Streams dienen dazu Daten von einem Ort zum anderen zu leiten. Alle für die Ein- und Ausgabe benötigten Klassen und Methoden können mit `import java.io.*` eingebunden werden.

1.1 Die Klassenhierarchie

Die Streams stammen jeweils von einer gemeinsamen abstrakten Basisklasse ab (InputStream, Reader, OutputStream, Writer), so dass man von einer gewissen Grundfunktionalität ausgehen kann. Zur Grundfunktionalität der Klassen InputStream und Reader zählt das Schliessen, Einlesen der Daten (evtl. auch nur von Teilen), das Bestimmen der Länge dieser Daten und die Unterstützung eines Synchronmarken- Mechanismus zum Wiederauffinden vorher gesetzter Marken. Die Klassen OutputStream und Writer deklarieren Memberfunktionen für das Schreiben (evtl. nur von Teilen), Schliessen des Stroms und das Leeren eines vorhandenen Puffers. Es gibt zwei unterschiedliche Typen von Streams:

1. Byte-Streams
2. Charakter-Streams

Bis zu Version 1.0.x gab es in Java nur **Byte-Streams**. In diesen wurden die Daten in 8 Bit großen Einheiten transportiert. Das reichte zum Lesen von herkömmlichen ASCII Zeichen aus. Allerdings gab es bei der Einführung von Unicode Zeichen Probleme. Daher erfanden die Java-Entwickler Charakter-Streams. Damit werden die Daten in 16 Bit großen Einheiten transportiert. Dadurch kann auf lokale Besonderheiten eingegangen werden. Um zu den 8-Bit-Zeichensätzen in externen Dateien kompatibel zu bleiben, wurden explizite Brückenklassen eingeführt, die Character-Streams in Byte-Streams überführen und umgekehrt. Byte-Streams können dazu verwendet werden um ganze Objekte serialisiert direkt abspeichern zu können. In diesem Skript werden hauptsächlich Charakter-Streams zu Ein- und Ausgabe verwendet. Alle Klassen des Pakets erzeugen unter Umständen Ausnahmen vom Typ `IOException`, die durch den Programmierer abgefangen werden müssen.

Kapitel 2

Eingabe

2.1 Das Fundament: Die Basisklasse Reader

Alle für die Ausgabe von Daten benutzten Klassen basieren auf der abstrakten Basisklasse `Reader`. Von ihr werden dann verschiedene Unterklassen abgeleitet die sich schachteln lassen. Die Klasse `Reader` besitzt folgende zu implementierende Schnittstellen:

```
public Reader()
public void close()
public void mark(int readAheadLimit)
public boolean markSupported()
public int read()
public int read(char[] cbuf)
public int read(char[] cbuf, int off, int len)
public long skip(long n)
public boolean ready()
public void reset()
```

Mit Hilfe der `read()` Methoden ist es möglich Zeichen vom Eingabestrom zu lesen. Die Methode `ready()` liefert als Rückgabe `true`, wenn ein `read()` ohne Blockierung der Eingabe möglich ist. Mit der Methode `mark()` lässt sich eine bestimmte Position innerhalb des Eingabestroms markieren. Die Methode sichert dabei die Position. Mit beliebigen `reset()`-Aufrufen lässt sich diese konkrete Stelle zu einem späteren Zeitpunkt wieder anspringen. `mark()` besitzt einen Ganzzahl-Parameter, der angibt, wie viele Zeichen gelesen werden dürfen, bevor die Markierung nicht mehr gültig ist. Die Zahl ist wichtig, da sie die interne Größe des Puffers bezeichnet, der für den Strom angelegt werden muss. Nicht jeder Datenstrom unterstützt dieses Hin- und Herspringen. Die Klasse `StringReader` unterstützt etwa die Markierung einer Position, die Klasse `FileReader` dagegen nicht. Daher sollte vorher mit `markSupported()` überprüft

werden, ob das Markieren auch unterstützt wird. Wenn der Datenstrom es nicht unterstützt und wir diese Warnung ignorieren, werden wir eine `IOException` bekommen. Denn `Reader` implementiert `mark()` und `read()` ganz einfach und muss im Bedarfsfall überschrieben werden. Mit `skip()` können einzelne Zeichen auf dem Eingabeband überlesen werden.

2.2 Ab in die Datei: `InputStreamReader` und `FileReader`

Die Klasse `InputStreamReader` dient dazu die Daten von `Byte` in `Charakter-Streams` umzuwandeln. Davon abgeleitet wird dann die Klasse `FileReader`, welche es ermöglicht Daten aus Dateien zu lesen. Dazu stellt die Klasse `FileReader` verschiedene Konstruktoren zu Verfügung:

```
public FileReader(String fileName) throws FileNotFoundException
public FileReader(File file) throws FileNotFoundException
public FileReader(FileDescriptor fd)
```

Wird dem Konstrukt der `fileName` übergeben, dann wird die Datei zum Lesen geöffnet. Klappt das nicht, dann wird eine `FileNotFoundException` geschmissen. Ein Code-Teil könnte dann so aussehen:

```
FileReader f;
int c;
try
{
    f = new FileReader("c:\\config.sys");
    while ((c = f.read()) != -1) {
        System.out.print((char)c);
    }
    f.close();
}
catch (IOException e)
{
    System.out.println("Fehler beim Lesen der Datei");
}
```

Das obige Beispiel zeigt, daß die Methode `read()` den Wert `-1` liefert, wenn das Ende des Eingabestroms erreicht worden ist.

2.3 Sehr nützlich: `StringReader` und `CharArrayReader`

Mit Hilfe der Klassen `StringReader` und `CharArrayReader` kann der Eingabestrom aus einem `String` bzw. einem `Char-Array` bezogen werden. Das ist bei

der Verarbeitung oft sehr nützlich, da Strings programm-intern leicht verändert werden können. Es lassen sich damit allgemeine Eingabeklassen schreiben, die entweder eine Datei oder aber einen internen String zu Eingabe benutzen. Ein Beispiel für `StringReader`:

```
Reader f;
int c;
String s;
s = "Das folgende Programm zeigt die Verwendung\r\n";
s += "der Klasse StringReader am Beispiel eines\r\n";
s += "Programms, das einen Reader konstruiert, der\r\n";
s += "den Satz liest, der hier an dieser Stelle steht:\r\n";
try {
    f = new StringReader(s);
    while ((c = f.read()) != -1) {
        System.out.print((char)c);
    }
    f.close();
} catch (IOException e) {
    System.out.println("Fehler beim Lesen des Strings");
}
```

2.4 Geschachtelt: Andere Reader

Im folgenden werden kurz einige weitere sehr nützliche Reader vorgestellt, die ineinander verschachtelt werden können:

BufferedReader Der Reader puffert die Eingaben beim Lesen und erhöht somit die Performance beim Lesen von Datenträgern. Er besitzt die Methode `readLine()` welche eine komplette Zeile aus einer Datei liest und diese zurückgibt.

Ein Beispiel für `BufferedReader`:

```
import java.io.*;
import java.util.*;
public class DateiIOChar{
    public static void main(String s[]) {
        Vector cont = new Vector(50,10);

        try{
            BufferedReader br = new
                BufferedReader(new
                    FileReader("DateiIOChar.java"));
            String zeile;
            while ((zeile=br.readLine())!=null) cont.add(zeile);
            br.close();
        }
```

```
    }
    catch(FileNotFoundException e) { }
    catch(IOException e) { }
    for(int i=(cont.size()-1); i>=0; i--){
        System.out.println((String)cont.elementAt(i));
    }
}
}
```

LineNumberReader Arbeitet im Prinzip so wie **BufferedReader**, speichert aber zusätzlich noch in einer internen Variable die Anzahl der bisher gelesenen Zeilen ab.

FilterReader Abstrakte Basisklasse für die Konstruktion von Eingabefiltern.

PushBackReader Diese Klasse besitzt zusätzliche Methoden um eingelesene Zeichen wieder in den Eingabestrom zurück zu schieben. Dieses ist z.B. im Compilerbau nützlich.

InputStreamReader Basisklasse für alle Reader, die einen Byte-Stream in einen Character-Stream umwandeln.

2.5 Schaubild

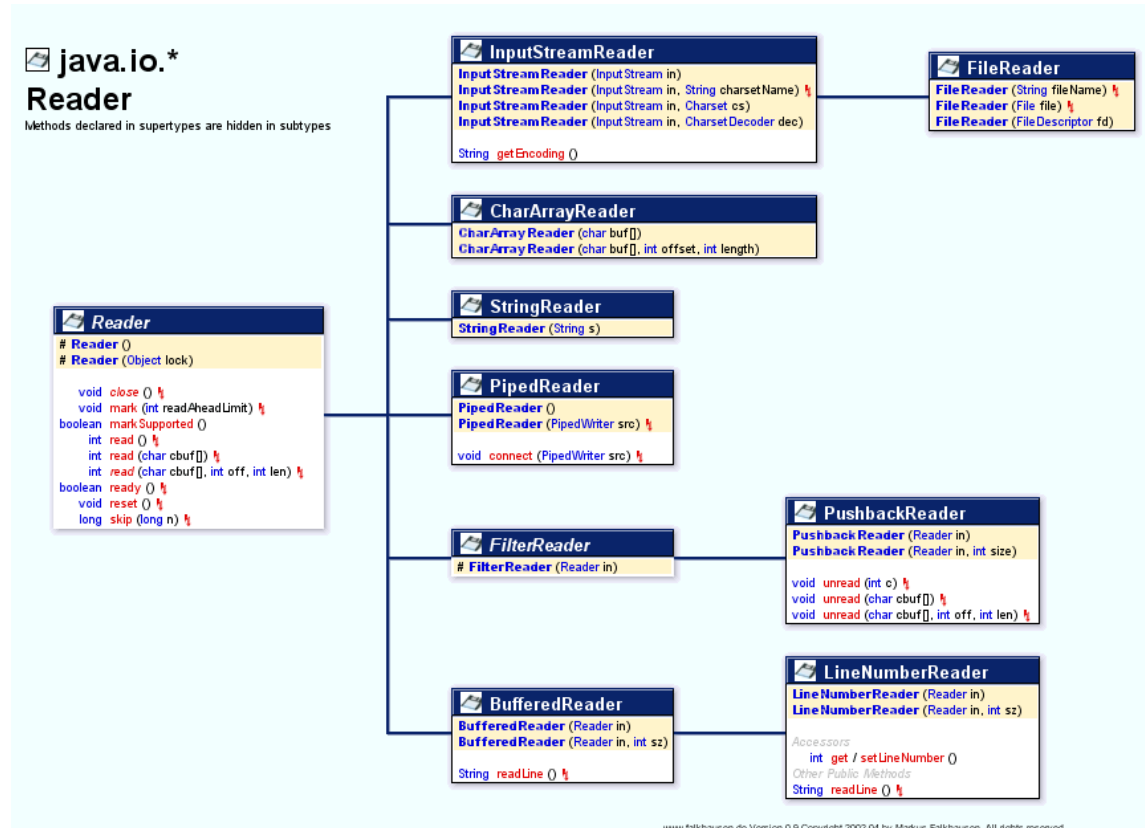


Schaubild entnommen von [4].

Kapitel 3

Ausgabe

3.1 Die abstrakte Basisklasse `Writer`

Was die Klasse `Reader` für die Ausgabe ist, das ist die Klasse `Writer` für die Eingabe. Sie besitzt die Methoden `close()`, `flush()` und mehrere Überladungen der Methode `write()`. Alle für die Ausgabe von Daten benutzten Klassen basieren auf der abstrakten Basisklasse `Writer`. Sie besitzt die Methoden `close()`, `flush()` und mehrere Überladungen der Methode `write()`. Beim Aufruf des Konstruktors wird der Ausgabestrom automatisch geöffnet. Danach kann mit einer der `write()` Methoden etwas in den Datenstrom geschrieben werden. Die möglichen Aufrufe der `write` Funktion sind:

```
public void write(int c)
public void write(char[] cbuf)
abstract public void write(char[] cbuf, int off, int len)
public void write(String str)
public void write(String str, int off, int len)
```

Von der sehr allgemeinen abstrakten Basisklasse `Writer`, welche einen reinen Byte-Stream vorsieht wird durch die Ableitung der Klasse `OutputStreamWriter` ein allgemeiner, abstrakter Ausgabestrom abgeleitet. Dieser Ausgabestrom konvertiert die Daten von Byte nach Charakter. Diese Klasse ist für den Programmierer aber zunächst einmal uninteressant. Wesentlich interessanter und häufiger benutzt ist die Klasse `FileWriter`.

3.2 Ab in die Datei: Die Klasse `FileWriter`

Die Klasse `FileWriter` implementiert die abstrakten Eigenschaften der Basisklasse `Writer` und besitzt zusätzlich noch Methoden um Dateien explizit zu öffnen:

```
public FileWriter(String fileName) throws IOException
```

```
public FileWriter(String fileName, boolean append) throws IOException
public FileWriter(File file) throws IOException
public FileWriter(FileDescriptor fd)
```

Am einfachsten kann eine Datei geöffnet werden, indem der Dateiname als String-Parameter `fileName` übergeben wird. Falls `fileName` eine bereits vorhandene Datei bezeichnet, wird sie geöffnet und ihr bisheriger Inhalt gelöscht, andernfalls wird eine neue Datei mit diesem Namen angelegt. Zusätzlich kann dem Aufruf der Wert `append` übergeben werden. Wenn dieser `true` ist, dann werden die Daten an die Datei angehängt.

```
001 /* Dateiausgabe mit Java */
002
003 import java.io.*;
004
005 public class DateiAusgabe
006 {
007     public static void main(String[] args)
008     {
009         String hello = "Hallo MATAs\r\n";
010         FileWriter f1;
011
012         try {
013             f1 = new FileWriter("hallo.txt");
014             f1.write(hello);
015             f1.close();
016         } catch (IOException e) {
017             System.out.println("Fehler beim Erstellen der Datei");
018         }
019     }
020 }
```

Die Ausnahme `IOException` kann unterschiedliche Bedeutungen haben. Im Konstruktor zeigt sie einen Fehler beim Öffnen der Datei. In der Methode `write` kann sie z.B. einen Schreibfehler definieren. Anders als im obigen Beispiel sollten (bei absolut korrekter Programmierung) die Fehler bei jedem Methoden-Aufruf abgefangen werden.

3.3 Hier geblieben: Die Klassen `StringWriter` und `CharArrayWriter`

In Java ist es durch die Streams möglich Ausgaben nicht nur in Dateien sondern auch in Strings oder Char-Arrays umzuleiten. Die Klasse `StringWriter` repräsentiert dabei eine Klasse mit deren Hilfe Eingabedaten in einen sogenannten `StringBuffer` geschrieben werden können. Der `StringBuffer` wächst dabei dynamisch. Konstruktoren:

```
public StringWriter()
public StringWriter(int initialSize)
```

Mit Hilfe der Methode `getBuffer()` kann man aus dem `StringBuffer` Objekt den String-Puffer extrahieren. Die bekannte Methode `toString()` gibt den Inhalt des Puffers als String zurück:

```
public StringBuffer getBuffer()
public String toString()
```

Die Klasse `CharArrayWriter` arbeitet im Prinzip genauso wie `StringWriter`. Allerdings besitzt sie nicht die Methode `getBuffer()` sondern eine Methode `toCharArray` welche ein Char-Array mit dem Text zurück liefert.

```
public String toString()
public char[] toCharArray()
```

Zusätzlich besitzen beide Klassen noch Methoden die den Puffer kontrollieren und die Weiterleitung des Datenstroms an andere Writer ermöglichen:

```
public void reset()
public int size()
public void writeTo(Writer out) throws IOException
```

Die Funktion `reset()` leert den internen Puffer. `size()` liefert die Grösse des Puffers.

3.4 Java kann alles: Weitere nützliche Ausgabeklassen

Java stellt weitere sehr nützliche Ausgabeklassen bereit. Hier ein kleiner Überblick:

`BufferedWriter` Klasse zum Zwischenspeichern der im Daten im Stream. Sehr nützlich z.B. zur Ausgabe in Dateien, da dabei nicht jede `print()` oder `write()` Anweisung sofort ausgeführt wird.

`PrintWriter` besitzt neben den üblichen `write()` Methoden noch verschiedene Überladungen der Methode `print()`. Damit können die meisten in Java vorkommenden Datentypen direkt ausgegeben werden. (Analog besitzt sie auch `println()` Methoden die am Ende noch ein Zeilenendezeichen dranhängen).

3.5 Selbstgemacht: `FilterWriter` zum Erweitern

Als spezielle Klasse bietet sich des weiteren noch die Klasse `FilterWriter` an. Diese Klasse kann als Basisklasse genutzt werden, um eigene Ausgabeklassen zu schreiben. Damit lassen sich die Ausgaben an die eigenen Bedürfnisse anpassen (z.B. Zeilennummer mit ausgeben o.ä.). Hier mal ein Beispiel [1]:

```
001 /* Listing1805.java */
002
003 import java.io.*;
004
005 class UpCaseWriter
006 extends FilterWriter
007 {
008     public UpCaseWriter(Writer out)
009     {
010         super(out);
011     }
012
013     public void write(int c)
014     throws IOException
015     {
016         super.write(Character.toUpperCase((char)c));
017     }
018
019     public void write(char[] cbuf, int off, int len)
020     throws IOException
021     {
022         for (int i = 0; i < len; ++i) {
023             write(cbuf[off + i]);
024         }
025     }
026
027     public void write(String str, int off, int len)
028     throws IOException
029     {
030         write(str.toCharArray(), off, len);
031     }
032 }
033
034 public class EigeneAusgabe
035 {
036     public static void main(String[] args)
037     {
038         PrintWriter f;
039         String s = "und dieser String auch";
040
041         try {
042             f = new PrintWriter(
043                 new UpCaseWriter(
044                     new FileWriter("upcase.txt")));
045             //Aufruf von aussen
046             f.println("Diese Zeile wird schoen gross geschrieben");
```

```

047         //Test von write(int)
048         f.write('a');
049         f.println();
050         //Test von write(String)
051         f.write(s);
052         f.println();
053         //Test von write(String, int, int)
054         f.write(s,0,17);
055         f.println();
056         //Test von write(char[], int, int)
057         f.write(s.toCharArray(),0,10);
058         f.println();
059         //---
060         f.close();
061     } catch (IOException e) {
062         System.out.println("Fehler beim Erstellen der Datei");
063     }
064 }
065 }

```

Die Klasse `FilterWriter` besitzt selbst keine Methode um ein Zeichen vom typ `char` auszugeben. Daher wird im obigen Beispiel (Zeile 013) die Methode `write()` überschrieben. In den Zeilen 042-044 ist zu sehen, wie die Klassen ineinander verschachtelt werden können. Jede `Writer`-Klasse besitzt jeweils einen Konstruktor, welchem wieder ein `Writer-Stream` übergeben werden kann. Die Aufrufe der verschiedenen Arten der Methode `write()` werden in der `main()`-Methode sichtbar. Diese Aufrufe ließen sich mit `println()` nicht realisieren. Im Konstruktor wird der Konstruktor der Mutterklasse (`super`) aufgerufen. Es gibt drei `write()` Methoden von denen zwei die Möglichkeit besitzen einen Offset und eine Länge zu übergeben um den String zu beschneiden. Die Ausgabe des Programms lautet wie folgt:

```

DIESE ZEILE WIRD SCHOEN GROSS GESCHRIEBEN
A
UND DIESER STRING AUCH
UND DIESER STRING
UND DIESER

```

Alle hier vorgestellten Streaming-Klassen bieten die Möglichkeit, dem Konstruktor einen anderen Stream zu übergeben aus dem die neue Klasse dann ihre Daten bezieht. Im obigen Beispiel steht z.B. in den Zeilen 042-044:

```

f = new PrintWriter(
new UpCaseWriter(
new FileWriter("upcase.txt")));

```

Dabei wird mit `FileWriter("upcase.txt")` ein neuer Dateizugriff ermöglicht. Dieser übergibt seine Daten dann direkt an die neue Klasse `UpCaseWriter`.

Dieser wiederum gibt die Daten an `PrintWriter` weiter, der dann die nötigen Ausgabefunktionen `write(...)`, `println(...)`, etc. besitzt. So können die unterschiedlichen Reader und Writer miteinander verschachtelt werden.

3.6 Schaubild

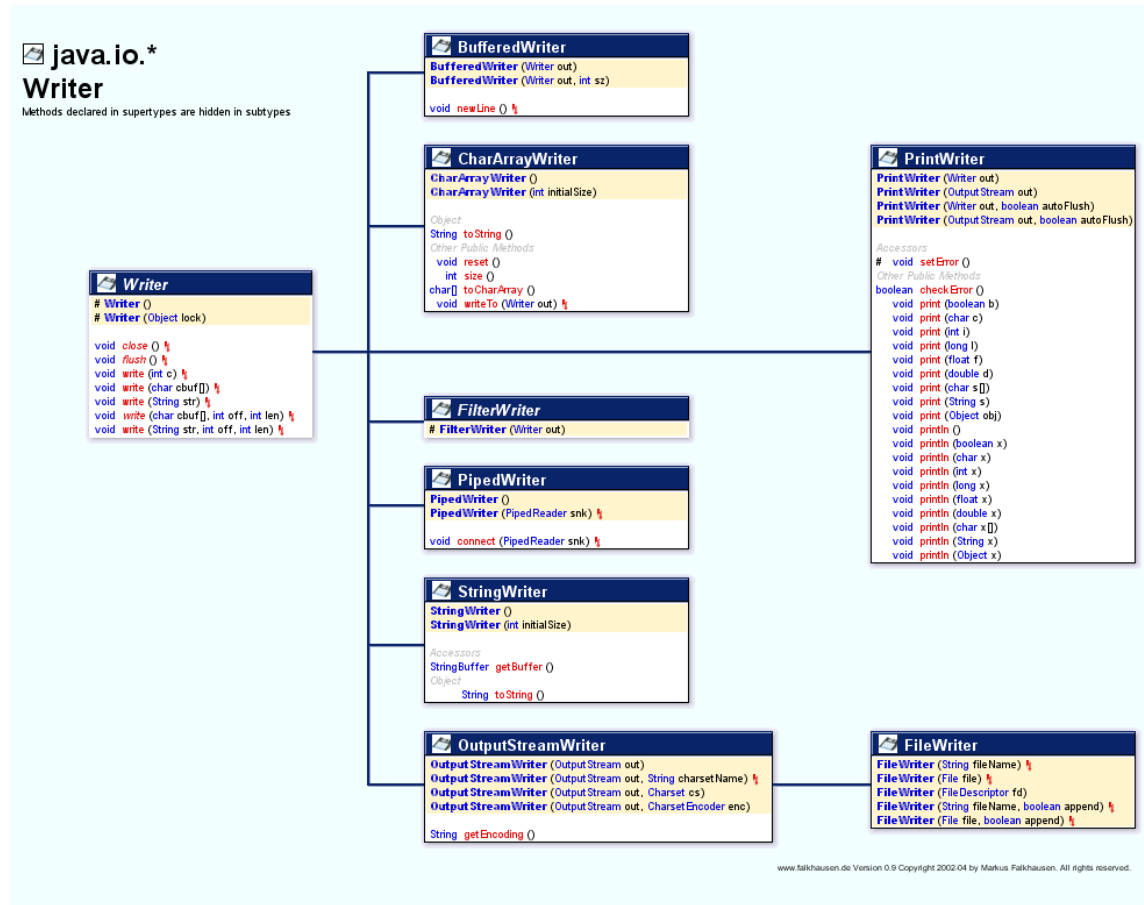


Schaubild entnommen von [5].

Kapitel 4

Fazit

Wir haben gesehen, dass es unterschiedliche Ein- und Ausgabestreams gibt. Hier nochmal eine kurze Zusammenfassung der behandelten Character-Stream Arten:

Typ	CharacterStreams
Puffern	BufferedReader BufferedWriter
Filtern	FilterReader
Konvertierung von Byte nach Character	InputStreamReader OutputStreamReader
Zählen	LineNumberReader
Schreiben	PrintWriter

Um einen Ein- oder Ausgabestrom zu Schließen benutzt man für gewöhnlich die Methode `close()`. Alle nach `close()` aufgerufenen Lese- oder Schreiboperationen werden mit einer Exception verhindert. Sollte man die Funktion `close()` vergessen, so versucht zunächst der Garbage-Collector die Datei zu schliessen. Gelingt ihm das nicht, kann es zu Deadlocks kommen.

Literaturverzeichnis

- [1] Martin Krüger. *Handbuch der Java-Programmierung*. 3. Auflage, Addison-Wesley Verlag, 2002, www.javabuch.de
- [2] RRZN. *Java 2 - Grundlagen und Einführung*. 1. Auflage, Regionales Rechenzentrum für Niedersachsen, Universität Hannover, 2001, www.rrzn.uni-hannover.de
- [3] <http://java.sun.com/j2se/1.4.2/docs/api/>
- [4] <http://www.falkhausen.de/en/diagram/html/java.io.Reader.html>
- [5] <http://www.falkhausen.de/en/diagram/html/java.io.Writer.html>