

Ein- und Ausgabe in Java

- Das Package `java.io` enthält eine Sammlung von Klassen, die die Kommunikation mit dem Computer steuern
 - Einlesen und Ausgeben von Dateien
 - Ausgabe auf dem Bildschirm
 - Einlesen von der Tastatur
 - etc.
- IO steht für Input / Output
 - also für Eingabe und Ausgabe
- IO Exception
 - Beinahe alle IO-Methoden können eine Exception werfen
 - z.B. wenn die Verbindung zum Ein- oder Ausgabegerät abreißt
 - Die meisten Exceptions sind vom Typ `java.io.IOException`

Datenströme

- Ein- und Ausgabe ist in Java über sogenannte Datenströme (data streams) organisiert
- **Input Stream:**
 - Ein Daten-Strom, der von einer *Daten-Quelle* zum Computer führt
 - Tastatur
 - File System
 - etc.
- **Output Stream:**
 - Ein Daten-Strom, der vom Computer zu einer *Daten-Senke* führt
 - Bildschirm
 - Drucker
 - File System
 - etc.

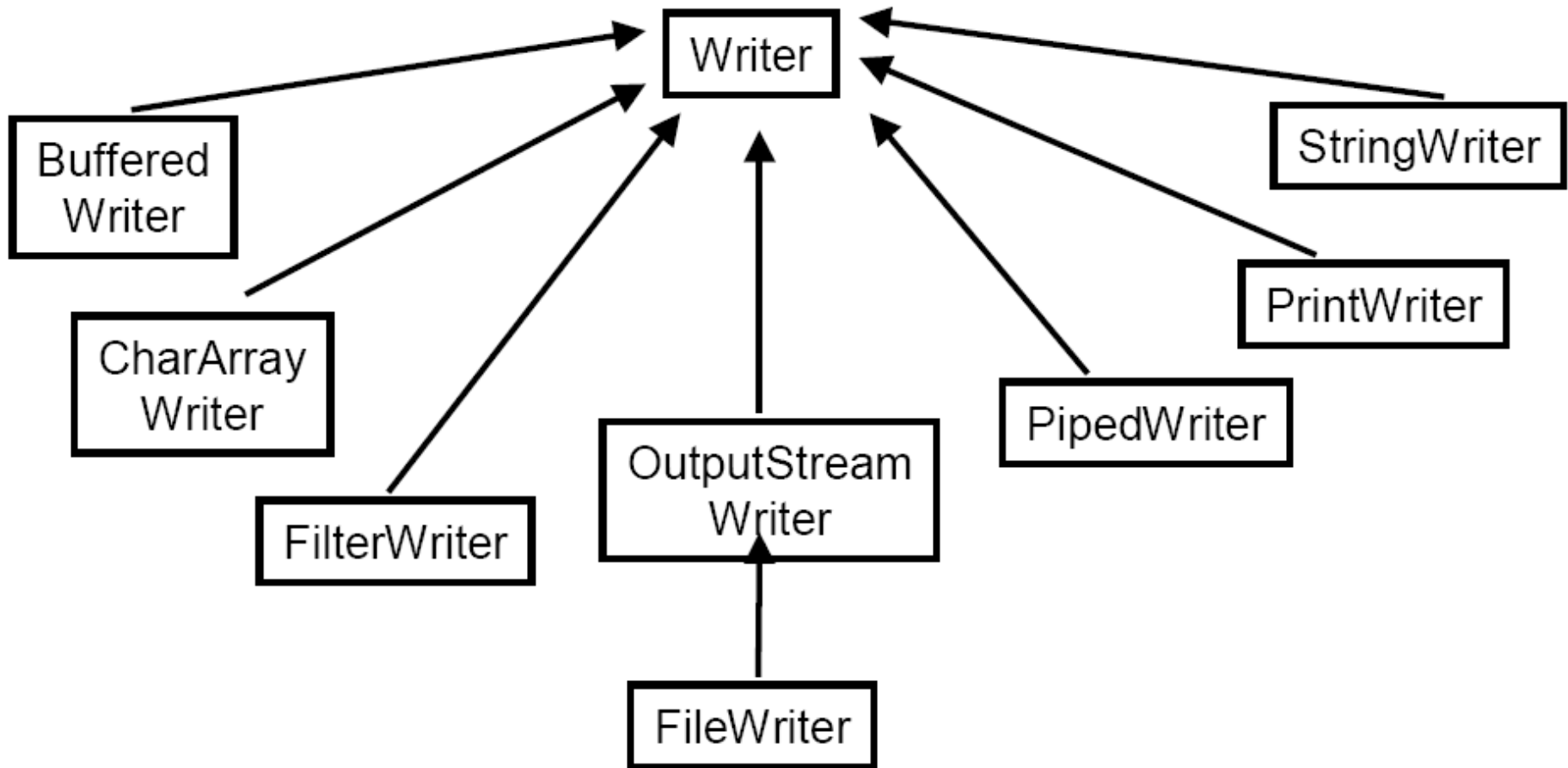
Datenströme

- Datenströme können beliebig miteinander kombiniert werden
 - Aneinanderhängen von Streams
 - also die Ausgabe eines Streams ist die Eingabe eines anderen Streams
 - Schachteln von Streams
 - am Eingabeteil wird ein Vorverarbeitungsschritt vorgeschaltet
 - am Ausgabeteil wird eine Nachverarbeitung durchgeführt
 - das erlaubt das Konstruieren von abstrakteren Streams auf der Basis von einfachen Streams
- Zwei grundlegende Typen von Streams:
 - Byte-Streams
 - Übertragen wird nur ein einzelnes Byte (8 bit)
 - Character-Streams
 - Übertragen wird ein ganzes Zeichen (in Java 16 bit, Unicode)
 - Wir betrachten nur Character-Streams
 - Byte Streams funktionieren analog

Klasse `java.io.Writer`

- Abstrakte Basisklasse für alle Character Output-Streams
- **Methoden:**
 - Konstruktor (für abstrakte Klasse nicht vorhanden)
 - stellt die Verbindung zur Datensenke her
 - muß in abgeleiteten Klassen konkret definiert werden
 - `public void close()`
 - Schließen der Verbindung
 - `public void write(int b) throws IOException`
 - Schreibt ein Character
 - das Argument ist integer, aber nur die ersten 16 Bits werden geschrieben (Konvertierung von `char` auf `int` ist automatisch)
 - `public void write(String s) throws IOException`
 - Schreiben eines Strings
 - `public void write(String s, int start, int n) throws IOException`
 - Schreibt `n` Charaters eines Strings, beginnend bei `start`
 - Es gibt auch Konstruktoren für `char[]` statt `String`

Überblick über Writer-Klassen



Buffering

- In vielen Fällen wird nach einem `write` nicht sofort geschrieben
- sondern es wird gewartet, bis sich eine gewisse Menge von Daten angesammelt haben
 - in einem sogenannten Puffer (engl. Buffer)
- die werden dann in regelmäßigen Abständen automatisch geschrieben

- Mit Hilfe der `flush`-Methode kann man das Schreiben erzwingen
 - `public void flush()`
 - Schreiben aller noch ausstehenden Daten
 - also alles was sich durch Aufrufe von `write` angesammelt hat, aber noch nicht geschrieben worden ist.

Klasse `java.io.PrintWriter`

- Konkrete Klasse zur Ausgabe auf eine Datei
- Konstruktoren:
 - `public PrintWriter(String name)`
throws `IOException`
 - Öffnet das File mit dem Namen `name` zum Schreiben
 - Falls das Öffnen des Files schiefgeht, wirft die Methode eine `IOException`
 - `public PrintWriter(String n, boolean app)`
throws `IOException`
 - öffnet das File mit Namen `n` zum Schreiben.
 - Falls die bool-sche Variable `app` auf `true` gesetzt ist, wird an das File angehängt
 - d.h. der ursprüngliche Inhalt des Files geht nicht verloren
 - Falls etwas schiefgeht → `IOException`
 - Es gibt auch Konstruktoren, die statt des Strings, der den File-Namen enthält, ein Objekt vom Typ `File` erwarten.

Beispiel

```
import java.io.*;

public class WriteToFile
{
    public static void main(String[] args)
    {
        FileWriter out;

        try {
            out = new FileWriter("hallo.txt");
            out.write("Hallo JAVA\r\n");
            out.close();
        }
        catch (Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }
}
```

Hier wird eine Output Stream auf das File "hallo.txt" geöffnet.

Ein Stream-Objekt mit Namen `out` wird deklariert

Ein String wird auf `out` geschrieben

Dann wird `out` geschlossen

Falls es eine Exception gab, wird die auf den Output-Stream `err`, der für Fehlermeldungen reserviert ist, geschrieben.

Und das Programm mit dem Rückgabewert 1 beendet.

Ausgabe auf Strings

- Ein String kann ebenso als Ausgabe-Einheit betrachtet werden wie ein File
 - im Prinzip ist ja ein File nichts anderes als ein langer String
- Dafür gibt es die Klasse `StringWriter`
 - schreibt nicht auf ein File, sondern auf ein `StringBuffer`-Objekt
- Methoden:
 - implementiert alle Methoden von `Writer`, zusätzlich noch:
 - `toString()`
 - gibt den momentanen Inhalt des `StringBuffer`s als String zurück
 - `getBuffer()`
 - retourniert das `StringBuffer`-Objekt
- Analog dazu gibt es die Klasse `CharArrayWriter`
 - der auf einen Array von characters schreibt.

Schachteln von Streams

- Manche Methoden verwenden einen bereits definierten Stream
 - führen aber am Anfang oder am Ende noch verschiedene Operationen durch
 - der Konstruktor dieser Streams muß einen anderen bereits definierten Stream erhalten.

Beispiele:

- `FilterWriter`
 - Abstrakte Basisklasse für die Konstruktion von Ausgabefiltern
- `PrintWriter`
 - Ausgabe aller Basistypen im Textformat
- `BufferedWriter`
 - Writer zur Ausgabepufferung
 - neue Methode `newline()` für einen Zeilenumbruch

Klasse `java.io.PrintWriter`

- Dient zur Ausgabe von Texten
- Neue Methoden sind
 - `print`
 - Es gibt eine `print`-Methode für jeden Standard-Typ
 - z.B. `print(int i)`, `println(boolean b)`, etc.
 - `println`
 - analog definiert, aber macht ein newline nach jeder Ausgabe
- `System.out` ist eine Klassen-Konstante vom Typ `PrintStream`
 - `PrintStream` funktioniert genauso wie `PrintWriter`
 - nur für Byte-Streams statt Character-Streams

Beispiel

```
public static void main(String[] args)
{
    PrintWriter pw;
    double sum = 0.0;
    int nenner;

    try {
        FileWriter fw = new FileWriter("zwei.txt");
        BufferedWriter bw = new BufferedWriter(fw);
        pw = new PrintWriter(bw);

        for (nenner = 1; nenner <= 1024; nenner++ ) {
            sum += 1.0 / nenner;
            pw.print("Summand: 1/");
            pw.print(nenner);
            pw.print("    Summe: ");
            pw.println(sum);
        }
        pw.close();
    }
    catch (IOException e) {
        System.out.println("Fehler beim Erstellen der Datei");
    }
}
```

Ein Stream-Objekt mit Namen `fw` wird deklariert, das auf eine File "zwei.txt" schreiben soll

Auf `fw` soll nicht direkt, sondern über einen Puffer `bw` geschrieben werden

Auf `bw` wird auch noch nicht direkt geschrieben, sondern ein `PrintWriter` definiert

Beispiel

```
public static void main(String[] args)
{
    PrintWriter pw;
    double sum = 0.0;
    int nenner;

    try {
        pw = new PrintWriter(
            new BufferedWriter(
                new FileWriter("zwei.txt") ) ) ;

        for (nenner = 1; nenner <= 1024; nenner++ ) {
            sum += 1.0 / nenner;
            pw.print("Summand: 1/");
            pw.print(nenner);
            pw.print("    Summe: ");
            pw.println(sum);
        }
        pw.close();
    }
    catch (IOException e) {
        System.out.println("Fehler beim Erstellen der Datei");
    }
}
```

Man kann diese verschachtelten Datenströme natürlich auch durch verschachtelte Aufrufe der Konstrukturen erzeugen

Berechnet wird die Summe aller Zahlen $1/n$ für $n=1$ bis 1024.

Dann wird `pw` geschlossen (und schließt die darunterliegenden Streams)

Klasse `java.io.FilterWriter`

- Abstrakte Klasse zur Definition eines Output-Filters
 - modifiziert die Ausgabe
 - bevor sie an den zugrundeliegenden Output-Stream weitergegeben wird
- Konstruktor benötigt daher wiederum einen existierenden Output-Filter
 - Ein Konstruktor, der den existierenden Output-Filter übernimmt und intern als Datenkomponente abspeichert ist vordefiniert
 - kann also von Unterklassen mittels `super` aufgerufen werden.
- Es gibt keine vorgeschriebenen zusätzlichen Methoden

Beispiel: Filter für Großschreibung

```
public class UpCaseWriter extends FilterWriter {
```

```
    public UpCaseWriter(Writer out) {  
        super(out);  
    }
```

Der Konstruktor hat einen Stream als Argument, aufgerufen wird der Konstruktor der Überklasse.

```
    public void write(int c) throws IOException {  
        super.write(Character.toUpperCase((char) c));  
    }
```

`write(int)` konvertiert jedes Zeichen auf Großbuchstaben und ruft dann `super.write` zur Ausgabe auf.

```
    public void write(char[] cbuf, int off, int len)  
        throws IOException {  
        for (int i = 0; i < len; ++i) {  
            write(cbuf[off + i]);  
        }  
    }
```

`write(char[],int,int)` ruft `write` für jedes gewünschte Zeichen des character Arrays auf

```
    public void write(String str, int off, int len)  
        throws IOException {  
        write(str.toCharArray(), off, len);  
    }
```

`write(String,int,int)` konvertiert auf `char[]`

Beispiel: Filter für Großschreibung

```
public class UpCaseWriter extends FilterWriter {
```

Die Methoden `write(char[])` und `write(String)` müssen nicht definiert werden, da sie in der übergeordneten Klasse über einen Aufruf von `write(char[], int, int)` bzw. `write(String, int, int)` implementiert sind. Polymorphie sorgt dann für das richtige Resultat

Der Konstruktor hat einen Stream als Argument, aufgerufen wird der Konstruktor der Überklasse.

```
IOException {
```

```
super.write(Character.toUpperCase((char) c));
```

```
}
```

`write(int)` konvertiert jedes Zeichen auf Großbuchstaben und ruft dann `super.write` zur Ausgabe auf.

```
public void write(char[] cbuf, int off, int len)
    throws IOException {
    for (int i = 0; i < len; ++i) {
        write(cbuf[off + i]);
    }
}
```

`write(char[], int, int)` ruft `write` für jedes gewünschte Zeichen des character Arrays auf

```
public void write(String str, int off, int len)
    throws IOException {
    write(str.toCharArray(), off, len);
}
```

`write(String, int, int)` konvertiert auf `char[]`

Beispiel für Verwendung

```
public static void main(String[] args)
{
    PrintWriter f;

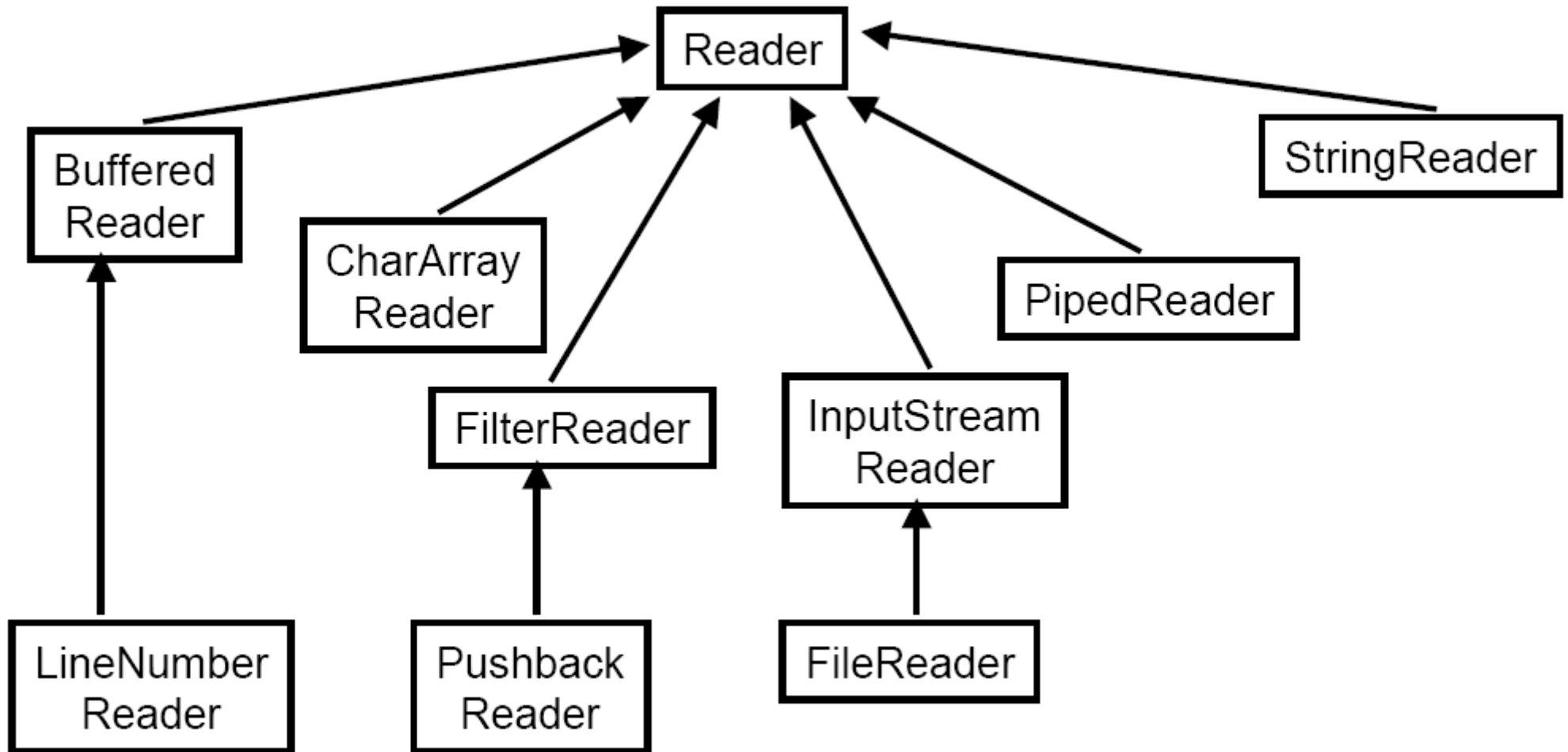
    try {
        f = new PrintWriter(
            new UpCaseWriter(
                new FileWriter("upcase.txt")));
        //Aufruf von außen
        f.println("Diese Zeile wird schön groß geschrieben");
    }
    catch (IOException e) {
        System.out.println(
            "Fehler beim Erstellen der Datei");
    }
}
```

Der neue FilterWriter UpCaseWriter kann natürlich in einen PrintWriter gesteckt werden. Das bewirkt, daß alle print-Statements groß ausgegeben werden!

Klasse `java.io.Reader`

- Abstrakte Basisklasse für alle Character Input-Streams
- **Die wichtigsten Methoden:**
 - Konstruktor (für abstrakte Klasse nicht vorhanden)
 - stellt die Verbindung zur Datenquelle her
 - muß in abgeleiteten Klassen konkret definiert werden
 - `public void close()`
 - Schließen der Verbindung
 - `public int read() throws IOException`
 - Liest ein Character (0..65535) oder -1 für Ende des Datenstroms
 - `public int read(char[] c) throws IOException`
 - Liest einen Array von Characters in den Array `c`
 - retourniert die Anzahl der gelesenen Zeichen oder -1 falls während des Lesens der Datenstrom zu Ende war
 - `public int read(char[] c, int start, int n) throws IOException`
 - Liest `n` Charaters nach `c` beginnend bei `c[start]`

Überblick über Reader-Klassen



Beispiel

```
public class BeispielReader
{
    public static void main(String[] args) {
        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");
        }
    }
}
```

Für den String `s` wird ein `StringReader` Objekt erzeugt

Zeichen für Zeichen wird (als `int`) gelesen, bis das Ende erreicht ist (-1)

Schließen nicht vergessen!

Exceptions fangen nicht vergessen!

Reader-Klassen

- Input-Streams können genauso geschachtelt und aneinandergesetzt werden wie Output-Streams
- die Funktionsweise von Klassen wie `BufferedReader`, `StringReader`, oder `FileReader` ist analog zu den entsprechenden `Writer` Klassen.
 - Das Beispiel auf der vorigen Folie würde genauso funktionieren, wenn der Satz in einem File stehen würde, und ein `FileReader`-Objekt verwendet würde
 - Dem Konstruktor wird dann der Name des Files statt des Strings `s` übergeben.

Interaktive Benutzer-Eingaben

- `System.in`
 - ist der Standard Input-Stream (ein Byte-Stream)
 - hat eine `read()` Methode, die ein Zeichen von der Tastatur lesen kann
 - üblicherweise möchte man jedoch Zeichenketten (Strings) lesen
- `InputStreamReader`
 - verwandelt den Byte-Stream in einen Character-Stream
- `BufferedReader`
 - hat eine zusätzliche Methode:

```
public String readLine() throws IOException
```

 - liest eine ganze Zeile auf einmal ein
 - Unterklasse `LineNumberReader` zählt die Zeilen mit

Beispiel

```
import java.io.*;

public class testInputStream {

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

        double zahl = 0;
        try {
            // Zahl einlesen
            System.out.print("Zahl eingeben: ");
            zahl = Double.parseDouble(input.readLine());
        }
        catch (IOException e) {
            System.err.println("Fehler beim Einlesen der Zahl!");
        }

        double quadriert = zahl * zahl;
        System.out.println("Quadrat = " + quadriert);
    }
}
```