

## 14. Beispiel: I/O-Streams

14.1 Eingabe/Ausgabe

14.2 Dateien

14.3 Streams

14.4 Lesen: Inputstreams, Reader

14.5 Schreiben: Outputstreams, Writer

14.6 I/O Ausnahmen

14.7 Zusammenfassung

# Überblick

---

## 14. Beispiel: I/O-Streams

14.1 Eingabe/Ausgabe

14.2 Dateien

14.3 Streams

14.4 Lesen: Inputstreams, Reader

14.5 Schreiben: Outputstreams, Writer

14.6 I/O Ausnahmen

14.7 Zusammenfassung

- Eine wichtige Eigenschaft von Programmen ist die Abstraktion von konkreten Daten.
- Eine Möglichkeit der Abstraktion von konkreten Daten ist die Verwendung von *formalen Parametern*, die bei der Verwendung des Programmes für eine bestimmte Aufgabe mit *aktuellen Parametern* belegt werden und so das *abstrakte* Programm auf *konkrete* Daten anwenden.
- Bisher haben wir die Verwendung von Programmen mit Parameterübergabe kennengelernt (`main`-Methode mit Parameter-Array). Weitere Möglichkeiten sind:
  - interaktive Abfrage vom Benutzer während der Laufzeit
  - Einlesen von Dateien, Datenströmen, Sensormesswerten etc.

- Umgekehrt gibt ein Programm auch oft Information an den Benutzer zurück.
- Wir haben schon oft die einfache Ausgabe mit `System.out.println(String s)` benutzt.
- Wenn es sehr viel Information ist, sollte ein Programm direkt in eine oder mehrere Dateien schreiben können.
- Vielleicht soll ein Programm auch mal Information an ein anderes Programm übergeben.

- In Java sind die grundlegenden Klassen für Ein- und Ausgabe im Package `java.io` gesammelt.
- Wir werden nun einen Blick auf einige dieser Klassen und ihren Gebrauch werfen.
- Nach diesem Kapitel wissen Sie dann auch endlich, was bei `System.out.println(String s)` (eine Methode, die wir von Anfang an benutzt haben, ohne wirklich etwas darüber zu wissen) eigentlich passiert.

## 14. Beispiel: I/O-Streams

### 14.1 Eingabe/Ausgabe

### 14.2 Dateien

### 14.3 Streams

### 14.4 Lesen: Inputstreams, Reader

### 14.5 Schreiben: Outputstreams, Writer

### 14.6 I/O Ausnahmen

### 14.7 Zusammenfassung

Betrachten wir zunächst eine Abstraktion von Dateien:

- Eine Datei hat einen Namen bzw. einen (relativen oder absoluten) Pfad, der auf einem gegebenen Dateisystem zu dieser Datei führt.
- Damit ist eine Datei (relativ oder absolut) eindeutig charakterisiert.
- Eine Datei kann gelesen, geschrieben, gelöscht, neu angelegt werden.
- Eine Datei kann verborgen sein, hat einen Eintrag, wann die letzte Änderung stattfand, eine Größe.
- Eine Datei kann ein Verzeichnis sein und wiederum andere Dateien beinhalten.
- Eine Abstraktion, die Klasse `File`, bezieht sich auf diese und andere Eigenschaften, ohne vom Inhalt der Datei zu wissen.

- Ein Objekt der Klasse `File` kann durch vier Konstruktoren erzeugt werden:
  - `File(String pathname)` – ein `File`-Objekt, das eine Datei für den angegebenen Pfad repräsentiert (*kann* existieren, *muß* aber nicht).
  - `File(String parent, String child)` – ein `File`-Objekt namens `child` innerhalb des Verzeichnisses mit Namen `parent`.
  - `File(File parent, String child)` – hier wird das Elternverzeichnis nicht durch den Namen, sondern bereits durch ein `File`-Objekt angegeben.
  - `File(URI uri)` – die Datei wird durch ein `URI`-Objekt spezifiziert, also ein Objekt, das eine Uniform Resource Identifier (`URI`) Referenz repräsentiert (siehe Klasse `java.net.URI`).
- Tipp: Die Bestandteile eines Pfades werden unter Unix mit Slash (`/`) getrennt, unter Windows mit Backslash (`\`). Java ermöglicht aber, diese Eigenschaft portabel zu halten: Die statische Variable `File.separator` hält den auf dem aktuellen System gültigen Trenner.

- Stellt ein `File`-Objekt `d` ein Verzeichnis dar (`d.isDirectory()`), kann das Objekt ein Array der Dateien in diesem Verzeichnis zu Verfügung stellen, z.B. `d.listFiles()` – oder `d.listFiles(FileFilter filter)`, um nur bestimmte Dateien (abhängig vom angegebenen Filter) in das Array aufzunehmen (z.B. nur Verzeichnisse).
- Soll für ein `File`-Objekt `d`, das auf dem Filesystem noch keinem realen File entspricht, ein Verzeichnis angelegt werden, gibt es die Methoden `d.mkdir()` bzw. `d.mkdirs()`.
- Die Methode **public** `File getParentFile()` gibt ein Objekt zurück, das das Verzeichnis repräsentiert, in dem dieses `File`-Objekt liegt.
- Lese- und Schreibzugriffe auf das Filesystem können zu Ausnahmen führen (am häufigsten `SecurityException` und `IOException`).

## FileFilter für Verzeichnisse

```
import java.io.File;
import java.io.FileFilter;

/**
 * FileFilter, der nur Verzeichnisse akzeptiert.
 */
public class DirectoryFileFilter implements FileFilter
{
    /**
     * Akzeptiert nur Verzeichnisse.
     *
     * @param pathname File-Objekt, das dem Filter uebergeben wird
     * @return true, wenn pathname ein Verzeichnis repraesentiert, false sonst
     * @see java.io.FileFilter#accept(java.io.File)
     */
    public boolean accept(File pathname)
    {
        return pathname.isDirectory();
    }
}
```

## Verzeichnisbaum ausgeben

```
import java.io.File;

/**
 * Klasse zur Ausgabe eines Verzeichnisbaumes.
 */
public class Verzeichnisbaum
{
    /**
     * Ein FileFilter, der nur Verzeichnisse akzeptiert.
     */
    public static final DirectoryFileFilter DIRECTORY_FILE_FILTER =
        new DirectoryFileFilter();

    /**
     * Die Einrueckung, um die eine tiefere Ebene
     * im Verzeichnisbaum eingerueckt wird.
     */
    public static final String EINRUECKUNG = "  ";
}
```

## Verzeichnisbaum ausgeben (cont.)

```
/**
 * Gibt den Verzeichnisbaum ausgehend
 * vom angegebenen Verzeichnis rekursiv aus.
 *
 * @param verzeichnis das Verzeichnis,
 * von dem ausgehend der Verzeichnisbaum ausgegeben werden soll
 * @param einrueckung die Einrueckung fuer die aktuelle Tiefe im Baum
 * (wird fuer eine tiefere Ebene jeweils um
 * {@link #EINRUECKUNG EINRUECKUNG} erweitert).
 */
public static void gibVerzeichnisbaumAus(File verzeichnis, String einrueckung)
{
    File[] verzeichnisse = verzeichnis.listFiles(DIRECTORY_FILE_FILTER);
    for(int i = 0; i < verzeichnisse.length; i++)
    {
        System.out.println(einrueckung+verzeichnisse[i].getName());
        gibVerzeichnisbaumAus(verzeichnisse[i], einrueckung+EINRUECKUNG);
    }
}
```

## Verzeichnisbaum ausgeben (cont.)

```
/**
 * Gibt den Verzeichnisbaum eines Verzeichnisses aus.
 *
 * Wenn ein Verzeichnis angegeben wird, wird der Baum
 * von diesem Verzeichnis aus durchwandert,
 * andernfalls vom aktuellen Arbeitsverzeichnis ausgehend.
 *
 * @param args wenn das Array leer ist, geht die Ausgabe des
 * Verzeichnisbaumes vom aktuellen Arbeitsverzeichnis aus, andernfalls
 * vom angegebenen Verzeichnis
 */
public static void main(String[] args)
{
    if(args.length > 0)
    {
        gibVerzeichnisbaumAus(new File(args[0]), "");
    }
    else
    {
        gibVerzeichnisbaumAus(new File(System.getProperty("user.dir")), "");
    }
}
}
```

## zum Ausprobieren...

Diese Beispielklassen finden Sie unter

<http://www.dbs.ifi.lmu.de/Lehre/Info2/SS07/skript/programmbeispiele/iostreams/DirectoryFileFilter.java>

bzw.

<http://www.dbs.ifi.lmu.de/Lehre/Info2/SS07/skript/programmbeispiele/iostreams/Verzeichnisbaum.java>

## 14. Beispiel: I/O-Streams

14.1 Eingabe/Ausgabe

14.2 Dateien

**14.3 Streams**

14.4 Lesen: Inputstreams, Reader

14.5 Schreiben: Outputstreams, Writer

14.6 I/O Ausnahmen

14.7 Zusammenfassung

- Wie wir gesehen haben, stellt ein `File`-Objekt eine Abstraktion einer Datei dar, die vom Inhalt der Datei absieht.
- Datei-Inhalte (und andere Informationen, die gelesen oder geschrieben werden) werden durch das Konzept der *Streams* modelliert.
- Ein *Stream* ist zunächst auch ein abstraktes Konstrukt, das grundsätzlich für die Fähigkeit steht, Zeichen auf ein (imaginäres) Ausgabegerät zu schreiben (`OutputStream`) oder von einem (imaginären) Eingabegerät zu lesen (`InputStream`).

- Eigentliche Grundlage von Lese- und Schreiboperationen sind Bytes.
- Die ursprünglichen Klassen für Lese- und Schreiboperationen sind `InputStream` und `OutputStream`, die *Byte-Streams* behandeln.
- Dabei ist jede Transporteinheit 8 Bit lang, das gewährleistet Kompatibilität zu Textdateien, die mit konventionellen Programmiersprachen erstellt wurden oder gelesen werden sollen.
- Probleme gibt es allerdings mit den 16 Bit langen Unicode-Zeichen, die innerhalb von Java zur Zeichendarstellung verwendet werden.

- Zur Verarbeitung von Text gibt es daher die *Character-Streams*, `Reader` bzw. `Writer`.
- Brückenklassen ermöglichen die Verbindung von *Byte-Streams* und *Character-Streams*: So erhält ein Konstruktor der Klasse `InputStreamReader` als Parameter einen `InputStream`.
- `Reader` bzw. `Writer` sind der gewöhnliche Weg, um *Textdateien* zu lesen bzw. zu schreiben. *Byte-Streams* sind eher von Bedeutung, um mit nicht-textuellen Daten umzugehen.

## 14. Beispiel: I/O-Streams

14.1 Eingabe/Ausgabe

14.2 Dateien

14.3 Streams

14.4 Lesen: Inputstreams, Reader

14.5 Schreiben: Outputstreams, Writer

14.6 I/O Ausnahmen

14.7 Zusammenfassung

- Die (abstrakte) Klasse `InputStream`, von der alle Eingabe-Byte-Streams erben, stellt Methoden zum Lesen von der jeweiligen Quelle (die im Konstruktor übergeben wird) zu Verfügung.
- Im Gegensatz zur Klasse `File` sind bei einem `InputStream` die meisten Methoden darauf angewiesen, dass die angegebene Eingabequelle tatsächlich existiert und lesbar ist. Deshalb sind hier für die meisten Methoden `IOExceptions` (aus unterschiedlichen Gründen) möglich.
- Unterschiedliche `read`-Methoden der Klasse geben ein gelesenes Byte als `int` zurück oder schreiben es in ein übergebenes `byte`-Array.
- Wird der Eingabe-Stream nicht mehr benötigt, dann schließt man ihn mittels `close()`.

- `FileInputStream`: Byte-Stream zum Lesen aus einer Datei.
- `SequenceInputStream`: Kann zwei oder mehr Eingabe-Streams so verbinden, dass die Daten nacheinander aus den angegebenen Quellen gelesen werden.
- `FilterInputStream`: Eine Klasse, die einen `InputStream` hält und Methoden-Aufrufe an diesen `InputStream` weitergibt. Ableitende Klassen können die Methoden dann einfach überschreiben, um besondere Aktionen beim Lesen (z.B. Filter) durchzuführen.
- Wichtige Unterklasse von `FilterInputStream`: Der `BufferedInputStream`, der den Lesevorgang puffert und damit für viele Fälle die Performanz deutlich erhöht.

- Die abstrakte Basis-Klasse aller Character-Input-Streams ist der `Reader`.
- Der `Reader` stellt (analog zum `InputStream`) Methoden zu Verfügung, um das nächste Zeichen zu lesen und zurückzugeben (als `int`) oder in ein bereitgestelltes `char`-Array zu schreiben.
- Auch hier gilt: Wird der Character-Eingabe-Strom nicht mehr benötigt, dann schließt man ihn mittels `close()`.

- Als abgeleitete Klasse kennen wir schon den `InputStreamReader`. Hier ist die Datenquelle ein `ByteStream`.
- Andere Unterklassen sind z.B.:
  - `FileReader` – zum Einlesen aus einer Datei.
  - `BufferedReader` – puffert die Eingabe und ermöglicht so das Einlesen von ganzen Zeilen. Dieser Reader ist für viele Probleme der Reader der Wahl, wenn man Text zeilenweise einlesen will.
  - `LineNumberReader` – Ableitung aus dem `BufferedReader`, der zusätzlich die Zeilen zählen kann.
  - `FilterReader` – abstrakte Klasse als Basis zur Konstruktion von Eingabefiltern (benutzt als Quelle einen beliebigen weiteren Reader).

```
public static void main(String[] args) throws IOException
{
    BufferedReader reader = new BufferedReader(
        new FileReader(
            new File(args[0])));
    for(String line; (line=reader.readLine())!=null;)
    {
        System.out.println(line);
    }
}
```

- Die Klasse `System` hält einen `InputStream`, `System.in`, als Attribut, der standardmäßig auf den Konsoleninput verweist.
- Über diesen `InputStream`, der mit Beginn des Programms zunächst offen ist, können also Tastatureingaben abgefragt werden.
- Auf `System.in` kommen auch Daten an, die in ein Programm gepiped werden. Beispiel auf einer Unix-Konsole:  
`programm1 | programm2`  
Der Output (`System.out`) von `programm1` wird der Input (`System.in`) von `programm2`.

```
public static void main(String[] args) throws IOException
{
    BufferedReader input = new BufferedReader(
        new InputStreamReader(
            System.in));
    System.out.println("Wie heisst Du?");
    String benutzerName = input.readLine();
    System.out.println("Hallo, " + benutzerName);
}
```

```
public static void main(String[] args) throws IOException
{
    BufferedReader input = new BufferedReader(
        new InputStreamReader(
            System.in));
    for(String line; (line=input.readLine())!=null;)
    {
        System.out.println(line);
    }
}
```

### 14. Beispiel: I/O-Streams

#### 14.1 Eingabe/Ausgabe

#### 14.2 Dateien

#### 14.3 Streams

#### 14.4 Lesen: Inputstreams, Reader

#### 14.5 Schreiben: Outputstreams, Writer

#### 14.6 I/O Ausnahmen

#### 14.7 Zusammenfassung

- `OutputStream` ist die Basis-Klasse für Byte-OutputStreams.
- Wie beim `InputStream` gibt es eine `close`-Methode, um einen `OutputStream` nach Gebrauch zu schließen.
- Meistens muß man außerdem explizit die `flush`-Methode aufrufen, um Daten, die gepuffert sind, physikalisch auf das Zielgerät zu schreiben.
- Analog zu den `read`-Methoden des `InputStreams` gibt es `write`-Methoden, um Bytes (als `int` übergeben) oder `byte`-Arrays zu schreiben.
- Auch hier muß man mit `IOExceptions` rechnen, da ja das Zielgerät vorhanden und beschreibbar sein muß, damit die Methoden ohne Fehler arbeiten können.

- `FileOutputStream` – um in eine Datei zu schreiben.
- `ByteArrayOutputStream` – um in ein `byte`-Array zu schreiben.
- `FilterOutputStream` – ermöglicht einen Filterschritt vor dem Schreiben.
- `BufferedOutputStream` – erbt von `FilterOutputStream`, schaltet einen Puffer dazwischen, was die Performanz verbessern kann.
- `PrintStream` – erweitert `FilterOutputStream` um die Fähigkeit, bequem alle möglichen Datentypen (primitive Typen, Strings und allgemein Objekte) zu schreiben (mit den `print` und `println`-Methoden). Die Besonderheit eines `PrintStreams` ist, dass keine `IOExceptions` geworfen werden. Stattdessen werden die `IOExceptions`, die der zugrundeliegende Stream wirft, als Fehler gemerkt. Der Fehler-Status des `PrintStreams` ist dann über die `checkError`-Methode abrufbar.

- Das Pendant zum `Reader` als `Character-OutputStream` ist der `Writer`.
- Wie beim `Byte-OutputStream` gibt es `close`, `flush` und `write`-Methoden, letztere erhalten aber Parameter vom Typ `char` bzw. `int` statt `byte`.
- Außerdem gibt es `write`-Methoden, die einen `String` erhalten können.

- `OutputStreamWriter` – übergibt den `Character Output` an einen `Byte-OutputStream`.
- `FileWriter` – erweitert den `OutputStreamWriter`, um in eine Datei zu schreiben.
- `PrintWriter` – ermöglicht wie der `PrintStream` die bequeme Ausgabe aller möglichen Datentypen (primitive Typen, Strings und allgemein Objekte) in einen `Text Outputstream` zu schreiben (mit den `print` und `println`-Methoden).  
Wie beim `PrintStream` werden keine `IOExceptions` geworfen, sondern als Fehler gemerkt. Der Fehler-Status des `PrintWriters` ist über die `checkError`-Methode abrufbar.
- `BufferedWriter` – puffert die Ausgabe zur Verbesserung der Performanz.

```
public static void schreibe(String text,
                           File zieldatei)
    throws IOException
{
    PrintWriter out = new PrintWriter(
        new FileWriter(zieldatei));

    out.print(text);
    out.flush();
    out.close();
}
```

## Standard- und Fehler-Ausgabestrom: `System.out`, `System.err`

- Mit `System.out` und `System.err` stehen zwei `PrintStreams` zu Verfügung, die standardmäßig offen sind (und keine `IOExceptions` werfen).
- Daten, die durch `System.out.println` in den Standard-`PrintStream` geschrieben werden, erscheinen (sofern `System.out` nicht umgeleitet wurde) auf der Konsole (Standard-Ausgabe).
- Daten, die durch `System.err.println` in den Error-`PrintStream` geschrieben werden, erscheinen (sofern `System.err` nicht umgeleitet wurde) auf der Konsole (Fehler-Ausgabe).

- Die Methode `printStackTrace()` eines Objekts der Klasse `Throwable` (also z.B. eine `IOException`) schreibt in die Fehler-Ausgabe `System.err`.
- Beim Aufruf eines Programmes über die Konsole erscheinen beide Ausgabeströme, `System.out` und `System.err`, gemischt. Man kann sie aber unterschiedlich behandeln, z.B. durch Pipen (hier unter `bash`):  

```
java programm 2> error.log
```

Dieser Aufruf führt dazu, dass der Error-Stream in die Datei `error.log` umgeleitet wird (die dabei neu angelegt bzw. überschrieben wird), während der Standard-Stream auf der Konsole ausgegeben wird.

### 14. Beispiel: I/O-Streams

#### 14.1 Eingabe/Ausgabe

#### 14.2 Dateien

#### 14.3 Streams

#### 14.4 Lesen: Inputstreams, Reader

#### 14.5 Schreiben: Outputstreams, Writer

#### 14.6 I/O Ausnahmen

#### 14.7 Zusammenfassung

- Neben den Klassen zur Ein- und Ausgabeverarbeitung finden sich im Package `java.io` auch diverse Ausnahmen, die verschiedene mögliche Fehler signalisieren.
- `IOException` ist die Oberklasse für alle Ausnahmen, die etwas mit dem I/O-Prozess zu tun haben.
- Eine häufig speziell behandelte Unterklasse dieser Exception ist `FileNotFoundException`, die auftritt, wenn von einer nicht-existierenden (oder aus sonstigen Gründen nicht auffindbaren/lesbaren/schreibbaren/erstellbaren) Datei gelesen oder in sie geschrieben werden soll (also z.B. beim Konstruieren eines `FileInputStreams`).

```
try
{
    BufferedReader reader = new BufferedReader(
        new FileReader(
            new File(args[0])));

    try
    {
        for(String line; (line=reader.readLine())!=null;)
        {
            System.out.println(line);
        }
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
catch(FileNotFoundException e)
{
    e.printStackTrace();
}
```

## 14. Beispiel: I/O-Streams

14.1 Eingabe/Ausgabe

14.2 Dateien

14.3 Streams

14.4 Lesen: Inputstreams, Reader

14.5 Schreiben: Outputstreams, Writer

14.6 I/O Ausnahmen

14.7 Zusammenfassung

Sie kennen jetzt das Package `java.io` als Sammlung von Klassen, die mit Ein- und Ausgabe zu tun haben:

- die Klasse `File` als abstrakte Darstellung von Dateien und Verzeichnissen,
- das Konzept der Streams, um Eingabe und Ausgabe zu behandeln,
- Input- und Outputstreams für Bytes,
- Input- und Outputstreams für Text-Daten (Character-Streams: `Reader` und `Writer`),
- typische I/O-Ausnahmen.