Grundlagen der Programmierung

Dr. Christian Herzog
Technische Universität München

Wintersemester 2017/2018

Kapitel 10: Ein- und Ausgabe

Ziele dieses Kapitels

- Sie verstehen das Konzept der Ströme
  - Sie können Klassen wie **`Reader`**, **`Writer`**, **`InputStream`** und **`OutputStream`** benutzen.
- Sie können in Java Dateien lesen und schreiben

Das Problem

- In den Informatik-Systemen, die wir bisher implementiert haben, haben wir *Klassen deklariert*, *instanziiert* und die *Attribute mit Werten gefüllt*. Die Werte waren allerdings nur während der Laufzeit des Systems verfügbar.
- Häufig sollen Werte extern gespeichert werden, so dass sie bei einem erneuten Lauf wieder zur Verfügung stehen.
- In vielen Fällen werden auch Werte von einem Informatik-System erzeugt, die dann von einem anderen System benötigt werden.
- Bei interaktiven Systemen wollen wir bereits während der Ausführung Daten mit der Umgebung (Benutzer) austauschen:
  - Wir wollen Werte über eine Tastatur eingeben lassen, andere Werte sollen auf dem Bildschirm erscheinen.
- Zur Speicherung von Werten und zur Interaktion mit Benutzern führen wir jetzt die Konzepte *Strom* und *Datei* ein.

Datei

- Wir haben bereits die Kommunikation mit der Umgebung des Systems zugelassen, allerdings sehr spärlich.
  - **`System.out.println()`**: Ausdrucken von Daten auf dem Bildschirm
- Unser Ziel ist jetzt die Modellierung der Interaktion von Informatik-Systemen mit ihrer Umgebung.
- **Definition Datei:** Eine Verwaltungseinheit zur Repräsentation von externen Daten nach gewissen Organisationsformen, die den Zugriff innerhalb des Informatik-Systems auf die Daten festlegen.

Speicherung von Dateien

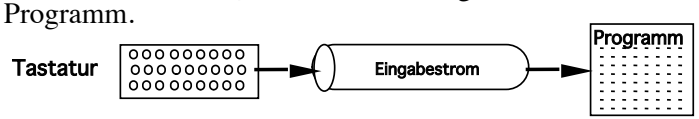
- Eine Datei ist mit einem ***Ein-Ausgabegerät,*** kurz E/A-Gerät, (engl. I/O device) verbunden, das die Daten einer Datei permanent speichern kann.
  - Beispiele von E/A-Geräten: Platte, CD, DVD, Magnetband, Stick
- Abhängig vom Typ der Daten in der Datei unterscheiden wir ebenfalls verschiedene Arten von Dateien: ***Text-Datei, Binär-Datei, Personal-Datei, Studenten-Datei***,...

Strom

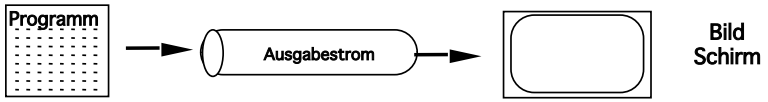
- Um auf Dateien innerhalb eines Informatik-Systems zugreifen zu können, führen wir den Begriff des Stroms ein.
- **Definition Strom:** Die interne Repräsentation einer (externen) Datei oder eines E/A-Gerätes in einem Informatik-System.
- **Definition Eingabe:** Das Lesen der Daten von einer Datei oder einem Eingabegerät in einen Strom. Der Strom heißt dann Eingabestrom.
- **Definition Ausgabe:** Das Schreiben von Daten eines Stroms auf eine Datei oder ein Ausgabegerät. Der Strom heißt dann Ausgabestrom.

Beispiele von Eingabe- und Ausgabeströmen

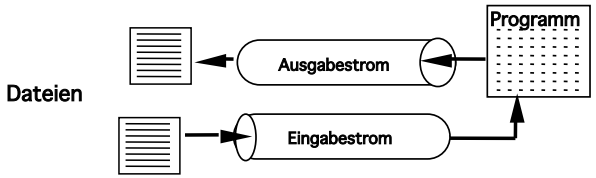
- Eine *Tastatur* ist eine Quelle für einen Eingabestrom von Zeichen an ein Programm.



- Ein *Bildschirm* ist ein Empfänger für einen Ausgabestrom von Zeichen von einem Programm.



- Eine *Datei* ist Empfänger oder Quelle für Ströme von Zeichen

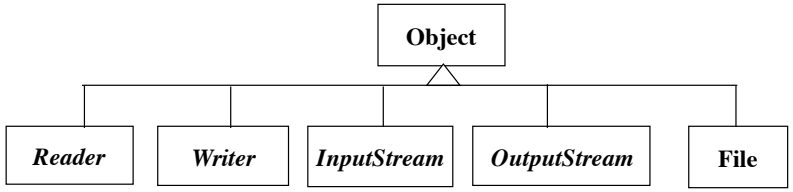


Modellierung von Strömen

- Wir modellieren Ströme als Klassen. Allgemein:
- Ein Strom hat mehrere Attribute:
  - **`Datei`**: Name der mit dem Strom assoziierten Datei
  - **`Marke`**: Zeiger auf das derzeitige Element (current element).
- Ein Strom stellt gewöhnlich folgende Dienste bereit:
  - **`Open()`**: Öffnen der Verbindung mit einer Datei / einem Gerät
  - **`Read()`**: Lesen eines Elementes
  - **`Write()`**: Schreiben eines Elementes
  - **`Close()`**: Schließen der Verbindung mit der Datei / dem Gerät.
- Die genaue Funktionalität der Dienste und die Implementierung von Strömen ist abhängig von der Programmiersprache, die Implementierung von Dateien ist außerdem oft noch abhängig vom Betriebssystem.
- Java unterstützt Ströme und Dateien.
  - Die Methoden lösen i.a. Ausnahmen der Klasse **`IOException`** bzw. von deren Unterklassen aus.

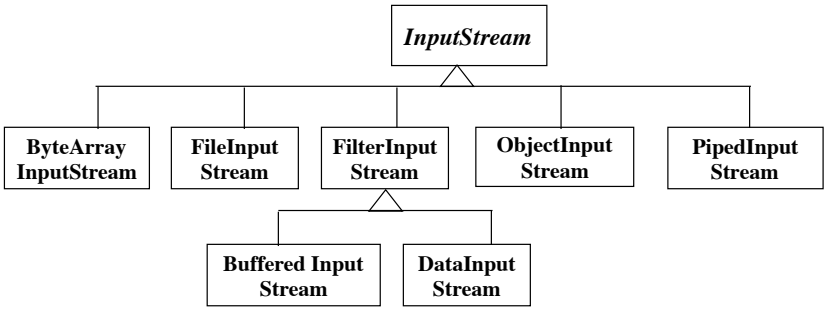
Ströme und Dateien in Java

- Java stellt eine große Anzahl von unterschiedlichen Strömen für Ein-Ausgabe (**`Reader`**, **`Writer`**, **`InputStream`**, **`OutputStream`**) und eine betriebssystem-unabhängige Beschreibung für Dateien (**`File`**) bereit, die alle im Paket **`java.io`** definiert sind.



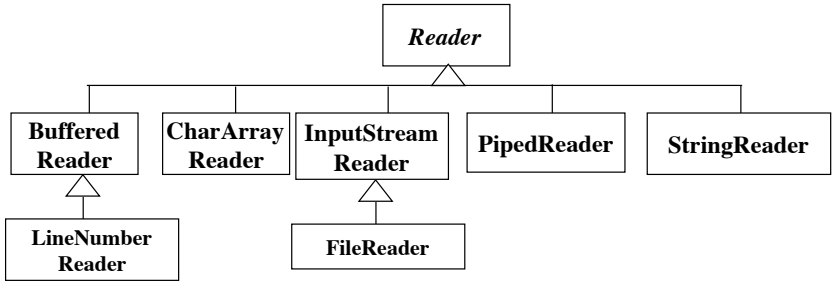
Klasse	Beschreibung
`Reader`	Abstrakte Klasse für textuelle Eingabeströme
`Writer`	Abstrakte Klasse für textuelle Ausgabeströme
`InputStream`	Abstrakte Klasse für binäre Eingabeströme
`OutputStream`	Abstrakte Klasse für binären Ausgabeströme
`File`	Plattform-unabhängige Beschreibung von Dateien

InputStream: Modelliert binäre Eingabeströme



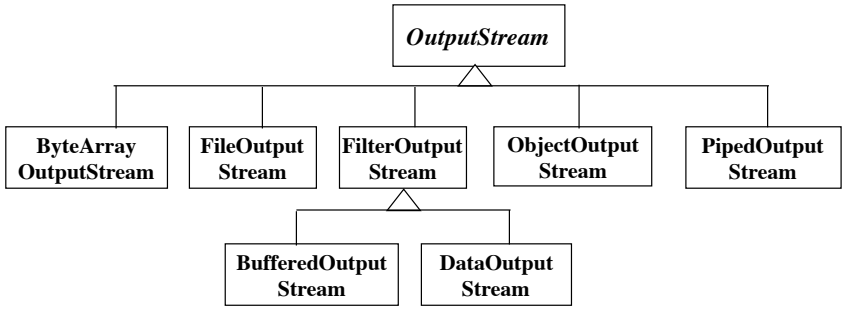
Klasse	Beschreibung
ByteArrayInputSteam	Erlaubt das Lesen von Reihungen, als ob sie Ströme wären
FileInputStream	Ermöglicht das Lesen von Bytes aus Binär-Dateien
FilterInputStream	Ermöglicht das Filtern von Daten auf verschiedene Arten
Buffered InputStream	Ermöglicht das Puffern von Eingabedaten
DataInputStream	Lesen von vordefinierten elementaren Java-Typen
ObjectInputStream	Zum Deserialisieren von Objekten
PipedInputStream	Lesen von Daten aus einem anderen Thread

Reader: Modelliert textuelle Eingabeströme



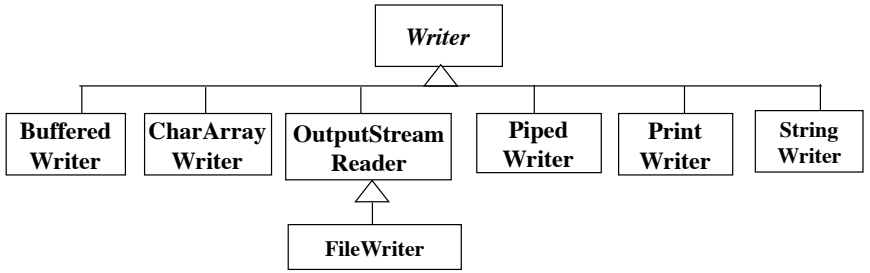
Klasse	Beschreibung
BufferedReader	Gepufferte Eingabe bei textuellen Eingabeströmen
CharArrayReader	Eingabe-Operationen auf Reihungen vom Typ Char
FileReader	Zeicheneingabe bei Dateien
PipedReader	Methoden zum Filtern von Zeicheneingaben
StringReader	Eingabe-Operationen für Zeichenketten (String)
LineNumberReader	Zählt Anzahl der Text-Zeilen, die gelesen wurden.

OutputStream: Modelliert binäre Ausgabeströme



Klasse	Beschreibung
ByteArrayOutputSteam	Erlaubt das Schreiben von Reihungen als ob sie Ströme wären
FileOutputStream	Ermöglicht das Schreiben von Bytes in Binär-Dateien
FilterOutputStream	Ermöglicht das Filtern von Daten auf verschiedene Arten
Buffered OutputStream	Ermöglicht das Puffern von Ausgabedaten
DataOutputStream	Schreiben von vordefinierten elementaren Java-Typen
ObjectOutputStream	Zum Serialisieren von Objekten
PipedOutputStream	Schreiben von Daten auf anderen Thread

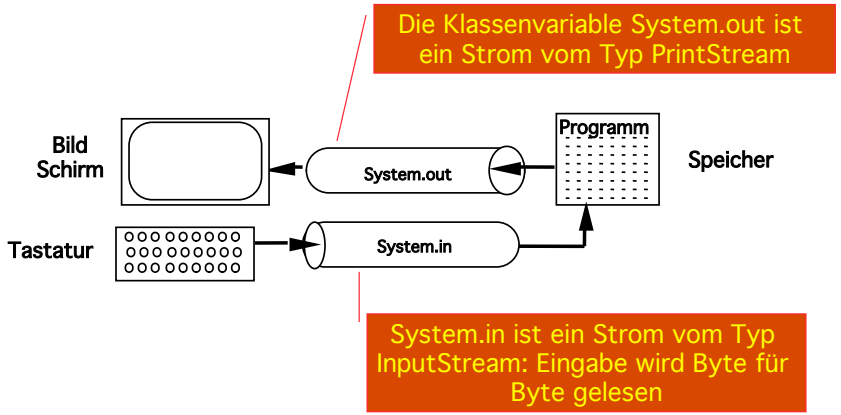
Writer: Modelliert textuelle Ausgabeströme



Klasse	Beschreibung
BufferedWriter	Gepufferte Ausgabe bei textuellen Ausgabeströmen
CharArrayWriter	Ausgabe-Operationen auf Reihungen (Array of Char)
FileWriter	Für Ausgabe auf Text-Dateien
PipedWriter	Methoden zum Filtern bei Ausgabe von Zeichen (Char)
PrintWriter	Textuelle Ausgabe von Java's Basistypen
StringWriter	Ausgabe von Zeichenketten (String)

Die Standard Ein/Ausgabe in Java basiert auf Strömen

- Informelles Modell mit Strömen



Wichtiges Konzept: Puffern von Daten

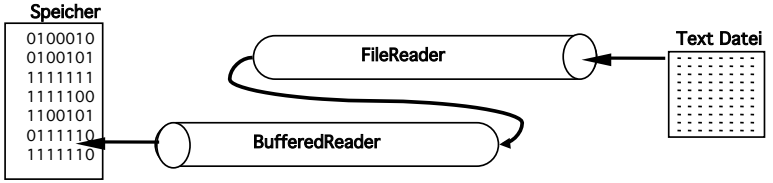
- **Definition Eingabepuffer (input buffer):** Ein Bereich im Speicher für die temporäre Speicherung von bereits gelesenen aber noch nicht verarbeiteten Daten:
  - Anstatt ein Byte nach dem anderen von dem Eingabegerät zu lesen, werden zunächst eine große Anzahl von Bytes gleichzeitig in den Puffer gelesen,
  - Diese werden dann stückweise bei jeder Lese-Operation ins Programm transferiert.
- **Definition Ausgabepuffer (output buffer):** Ein Bereich im Speicher für die temporäre Speicherung von zu schreibenden Daten.
  - Daten werden erst auf das Ausgabegerät geschrieben, wenn der Puffer voll ist (oder bei einer sogenannten **`flush()`**-Operation).
- Puffer helfen, den Geschwindigkeitsunterschied zwischen langsamen Zugriffen auf Geräte/Dateien und schnellen Prozessoren auszugleichen.

Wichtiges Konzept: Konkatenation von Strömen

- Die Deklarationen

```
FileReader fReader = new FileReader(fileName);
BufferedReader bReader = new BufferedReader(fReader);
```

  konkatenieren zwei Ströme vom Typ **`BufferedReader`** und **`FileReader`**.
- Das Programm kann dann **`bReader.readLine()`** benutzen, um Zeile für Zeile - und nicht Zeichen für Zeichen - von der Text-Datei mit Namen **`fileName`** zu lesen.
- Interpretation: Die Daten fließen von der Text-Datei erst durch **`FileReader`**, dann durch **`BufferedReader`** zum Speicher.

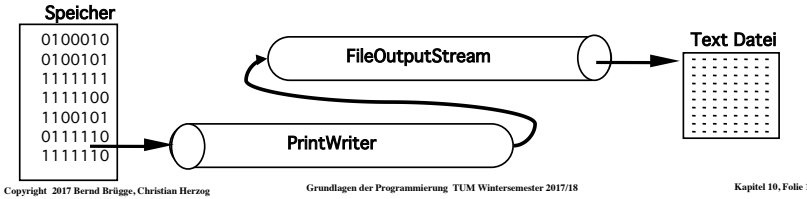


Konkatenation von Ausgabeströmen

- Die Deklarationen

```
FileOutputStream fOStream = new FileOutputStream (fileName);
PrintWriter pWriter = new PrintWriter (fOStream);
```

  konkatenieren zwei Ströme **`FileOutputStream`** und **`PrintWriter`**.
- Das Programm kann dann die „bequemen“ Methoden **`pWriter.print()`** und **`pWriter.println()`** benutzen.
- Interpretation: Die Daten fließen vom Speicher erst durch **`PrintWriter`**, dann durch **`FileOutputStream`** zur Text-Datei.
- **Bequem**: Die Methoden von **`PrintWriter`** behandeln alle Ausnahmen, die bei **`FileOutputStream`** auftreten können, und deklarieren selbst keine.



Wann benutzen wir welchen Strom?

- Bei binärer Ein-Ausgabe:
  - Unterklassen von **`InputStream`** und **`OutputStream`**.
- Bei Textueller Ein-Ausgabe:
  - Unterklassen von **`Reader`** und **`Writer`**.
- Beispiel: **`PrintWriter`** ist eine Unterklasse von **`Writer`**. Sie stellt Methoden zur textuellen Ausgabe von Objekten vom Typ **`int`**, **`long`**, **`float`**, **`double`**, **`String`** und **`Object`** bereit:

```
public void print(int i);          public void println(int i);
public void print(long l);         public void println(long l);
public void print(float f);        public void println(float f);
public void print(double d);       public void println(double d);
public void print(String s);       public void println(String s);
public void print(Object o);       public void println(Object o);
```


Verbindung von Dateien bzw. Geräten und Strömen

- Jede Programmiersprache muss ein Konzept bereitstellen, um Ströme mit Dateien zu verbinden.
  - In Java geschieht die Verbindung im Konstruktor der Strom-Klasse, die mit dem Dateinamen als Argument aufgerufen wird.
- Beispiel:

```
String fileName = "/home/bob/src/trivial.java";

FileWriter fWriter = new FileWriter(fileName);
```

  verbindet den Ausgabestrom **`fWriter`** mit einer Datei namens **`/home/bob/src/trivial.java`** .

Standard-Ein-/Ausgabe in Java

- Die Klasse **`System`** stellt in Java über Klassenvariablen Ströme bereit, die mit der Tastatur (**`System.in`**) und dem Bildschirm (**`System.out`** bzw. **`System.err`**) verbunden sind:

Field Summary	
static PrintStream	**err** The "standard" error output stream.
static InputStream	**in** The "standard" input stream.
static PrintStream	**out** The "standard" output stream.

Ein Beispiel: Kopieren vom Dateien

- Wie wollen in einer Klasse **`FileUtils`** eine Methode **`copyFile()`** zur Verfügung stellen, die zwei Dateinamen als Parameter erhält und die eine Datei in die andere kopiert.

Importieren der Klassen-Bezeichner aus dem Paket **`java.io`** (Siehe Kapitel 11)

```
import java.io.*;
class FileUtils {

  public static void copyFile (String inputFile, String outputFile) {

    // Es wird versucht, die Eingabedatei zum Lesen zu oeffnen:
    FileReader fReader = null;
    try {
      fReader = new FileReader (inputFile);
    }
    catch (FileNotFoundException e) {
      System.out.println("Fehler: Die Datei " + inputFile +
                  " konnte nicht zum Lesen geoeffnet werden.");
      System.exit(97);
    }
    // Konkatenation von Eingabestroemen:
    BufferedReader bReader = new BufferedReader(fReader);
```


Kopieren vom Dateien (cont'd)

```
    // Versuch, die Ausgabedatei zum Schreiben zu oeffnen:
    FileOutputStream fOStream = null;
    try {
       fOStream = new FileOutputStream (outputFile);
    }
    catch (IOException e) {
       System.out.println("Fehler: Die Datei " + outputFile +
                   " konnte nicht zum Schreiben geoeffnet werden.");
       System.exit(98);
    }

    // Konkatenation von Ausgabestroemen:
    PrintWriter pWriter = new PrintWriter (fOStream);
```


Kopieren vom Dateien (cont'd)

```
      // Der Inhalt der Eingabedatei wird zeilenweise in die
      // Ausgabedatei kopiert:
   try {
      String inputLine;
      do {
                // Eine Zeile lesen:
         inputLine = bReader.readLine();
                // Beim Dateiende wird der null-Pointer geliefert:
         if (inputLine != null)
                // Die Zeile schreiben:
            pWriter.println(inputLine);
      } while (inputLine != null); // Solange kein Dateiende
       // Schliessen der Stroeme:
      bReader.close();
      pWriter.close();
   }
   catch (IOException e) {
      System.out.println("Fehler:  " + e);
      System.exit(99);
   }
  } // copyFile()
} // class FileUtils
```


Zusammenfassung

- Eine ***Datei*** ist eine Sammlung von Daten, die extern auf einem Sekundärspeicher (Platte, CD, Band) gespeichert sind.
- Ein ***Strom*** ist ein Objekt, das Daten von anderen Objekten holt oder zu anderen Objekten liefert.
  - Ein ***Eingabestrom*** liefert Daten von einer externen Quelle zu einem Programm.
  - Ein ***Ausgabestrom*** liefert Daten vom Programm zu einem externen Gerät oder einer Datei.
- ***Puffer*** ist ein temporärer Bereich im Hauptspeicher, um Daten während der Ein- oder Ausgabe zu speichern.
- Java stellt eine Vielzahl von Klassen zur Implementierung von Strömen bereit.
- Durch ***Konkatenation*** von Strömen erhält man einen Gesamtstrom mit jeweils optimaler Schnittstelle zum Programm auf der einen Seite und zur Datei bzw. zum Gerät auf der anderen Seite.