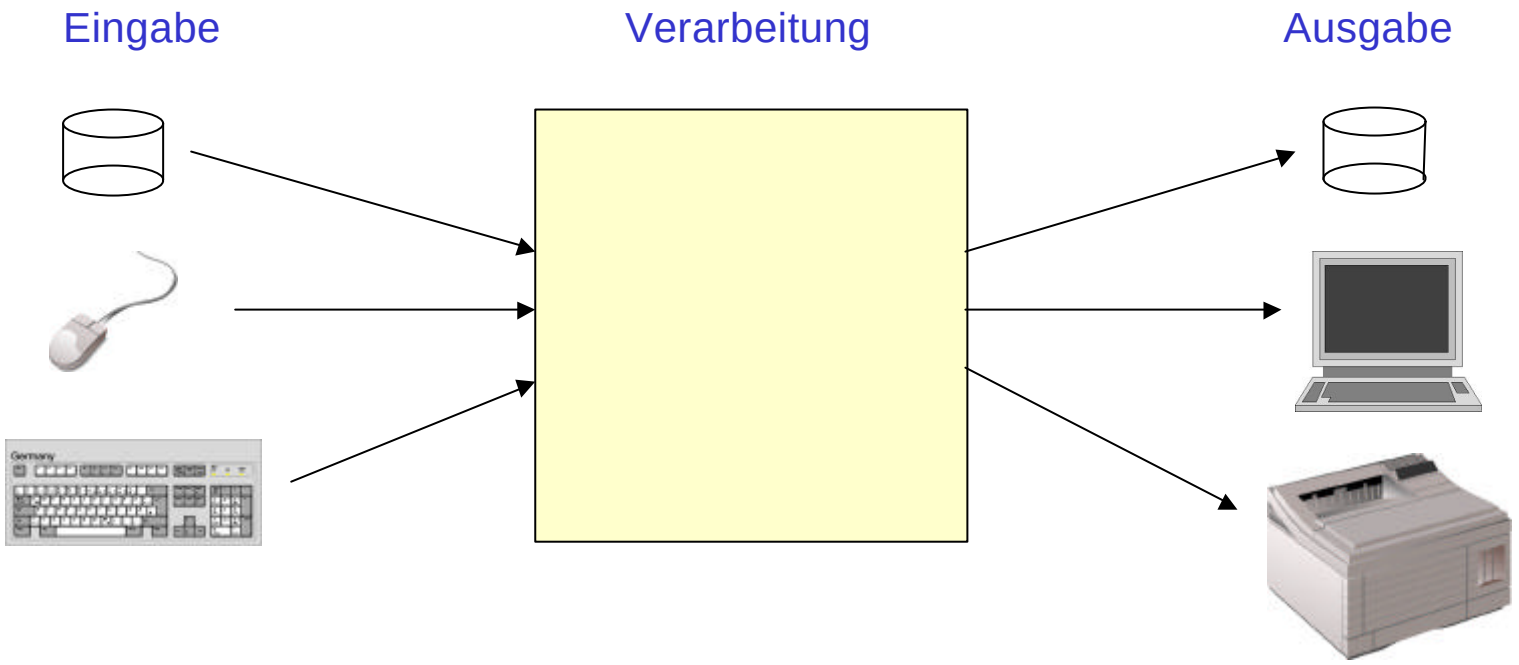
Wo sind wir?

- Java-Umgebung
- Lexikale Konventionen
- Datentypen
- Kontrollstrukturen
- Ausdrücke
- Klassen, Pakete, Schnittstellen
- JVM
- Exceptions
- Java Klassenbibliotheken
- **Ein-/Ausgabe**
- Collections
- Threads
- Applets, Sicherheit
- Grafik
- Beans
- Integrierte Entwicklungsumgebungen

Ein-/Ausgabe

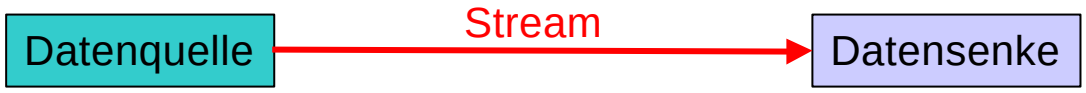


Die zu übertragenen Daten können Bytes, Zeichen, Fließkommazahlen, Felder, komplexe Objekte, ... sein.

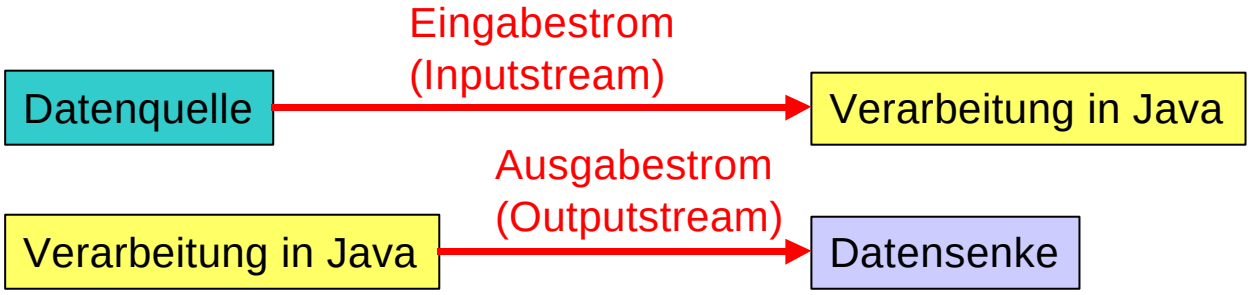
Folgerung: Ein-/Ausgabe kann sehr heterogen sein → großer Aufwand, wenig Portabilität → Abstraktion dringend erforderlich

Streams

Abstraktion der Kommunikation: Stream
Kommunikationsweg zwischen Datenquelle und Datensenke, über den seriell Daten übertragen werden. Operationen auf Streams (z.B. Lesen, Schreiben) sind unabhängig vom konkreten Ein-/Ausgabegerät.



Java:

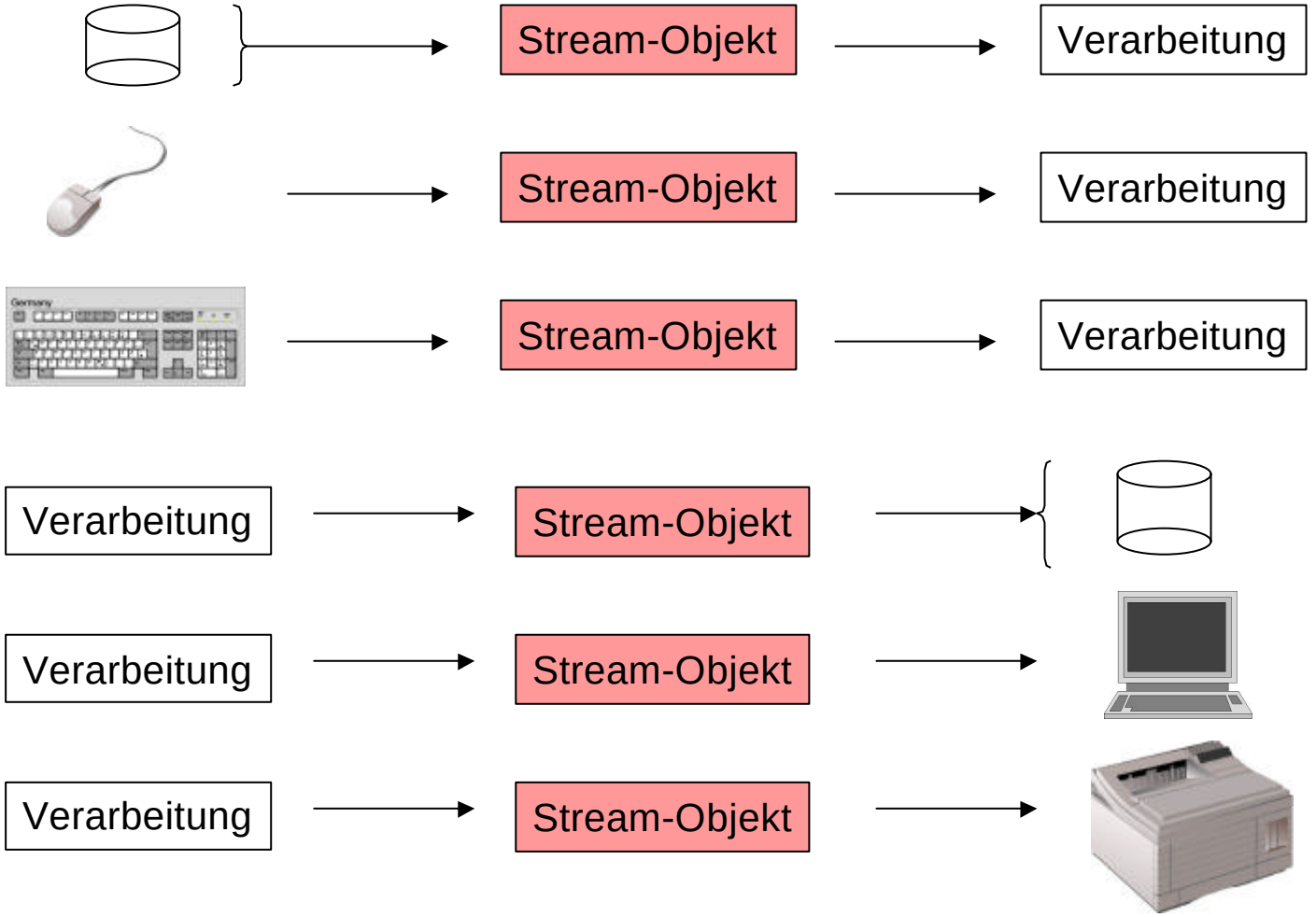


In Java können Datenquellen bzw. Datensenken sein:

- Dateien, Netzverbindungen (externe Datenquellen, -senken)
- byte-Feld, char-Feld, String, pipe (interne Datenquelle, -senke)

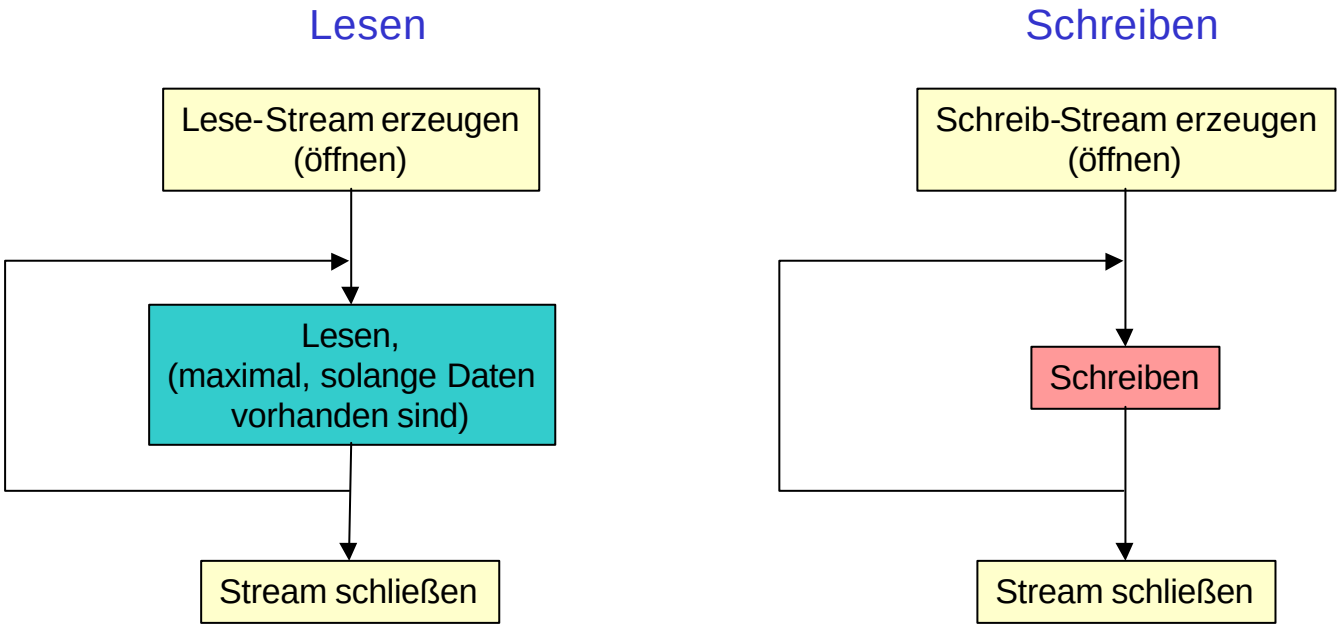
Das konkrete Ein-/Ausgabegerät ist durch die Abstraktion über Streams austauschbar, ohne dass das Programm geändert werden muss!

Streams


Stream-Objekt
Verarbeitung
Stream-Objekt
Verarbeitung
Stream-Objekt
Verarbeitung
Verarbeitung
Stream-Objekt
Verarbeitung
Stream-Objekt
Verarbeitung
Stream-Objekt

Operationen auf Streams

Vor der Nutzung eines Streams muss man ihn erst zum Lesen oder Schreiben öffnen, was implizit durch die Erzeugung eines Stream-Objektes geschieht. Anschließend kann man sequentiell Daten lesen (solange Daten vorhanden sind; EOF=End Of File) bzw. schreiben. Nach dem Schließen eines Streams sind keine weiteren Operationen mehr erlaubt, bis man den Stream wieder öffnet.



Character- und Byte-Streams

Zeichen in Java werden in Unicode (16 Bit) abgespeichert. Viele Daten liegen aber als Bytes (8 Bit) vor (Zeichen im ASCII-Code, Daten). Deshalb gibt es für die insgesamt 4 Möglichlichkeiten (Ein-/Ausgabe kombiniert mit 8/16 Bit) jeweils eine (abstrakte) Klasse in Java (Paket `java.io.*`).

	Bytes	Zeichen
Eingabe	`Inputstream`	`Reader`
Ausgabe	`Outputstream`	`Writer`

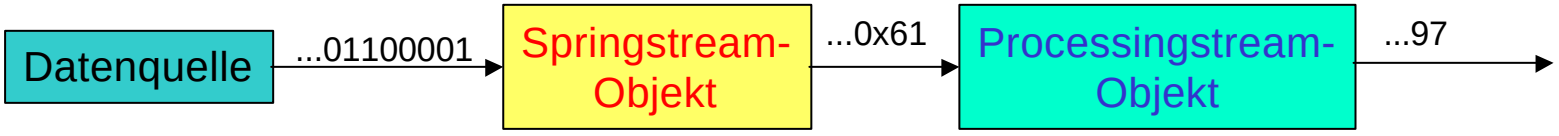
Die 4 abstrakten Basisklassen definieren nur ein Basisprotokoll, um Streams zu erzeugen usw. Die Klasse InputStream definiert aber z.B. nur eine abstrakte Methode `int read(byte[] b)`, die von konkreten Klassen implementiert werden muss (→ Springstream-/Sinkstream-Klassen).

In den Basisklassen ist z.B. auch nicht definiert, wie man ein `float` oder ein `String` lesen kann. Dies wird Aufgabe weiterer Klassen sein, die diese Basisklassen nutzen (→ Processingstream-Klassen).

Ein-/Ausgabeklassen

Ausgehend von den 4 abstrakten Basisklassen sind eine Vielzahl von konkreten Klassen definiert. All diese Klassen klassifiziert man wie folgt:

- Ein Objekt einer Sinkstream-Klasse kann Daten direkt in eine Senke schreiben.
- Ein Objekt einer Springstream-Klasse kann Daten direkt aus einer Quelle lesen (und implementiert bei Byte-Streams u.a. die Methode `int read(byte[] b)`).
- Ein Objekt einer Processingstream-Klasse erweitert die Funktionalität eines Sinkstream- bzw. Springstream-Objektes (Beispiel: Pufferung von Daten, Interpretation von 4 Bytes als ein int usw.)



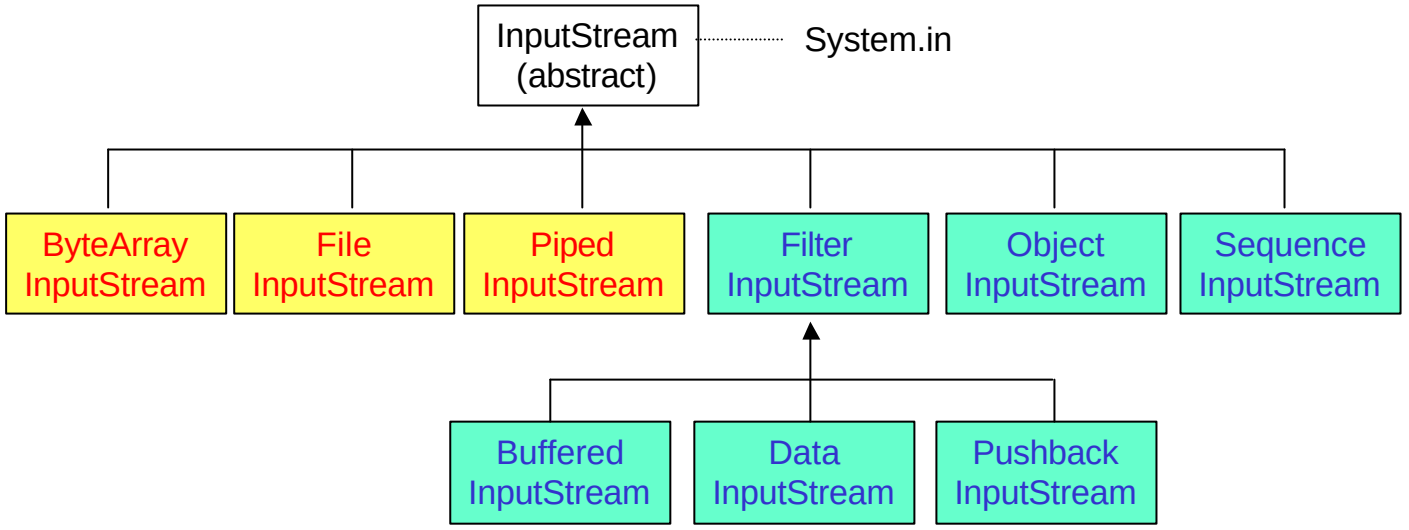
Übliches Vorgehen

- Erzeuge ein Objekt einer Springstream- bzw. Sinkstream-Klasse, um die geeignete Datenquelle bzw. -senke anzusprechen.
  Beispiel:
```
// Erzeuge Stream aus Datei
FileInputStream fis = new FileInputStream("Datei.dat");
```

- Erzeuge ein Objekt einer Processingstream-Klasse, die komfortable Methoden bereitstellt. Der Konstruktor dafür erwartet ein InputStream/OutputStream- bzw. Reader/Writer-Objekt. Gebe an dieser Stelle das Springstream- bzw. Sinkstream-Objekt aus Schritt 1 an (wg. Upcast möglich).
  Beispiel:
```
DataInputStream dis = new DataInputStream(fis);
```

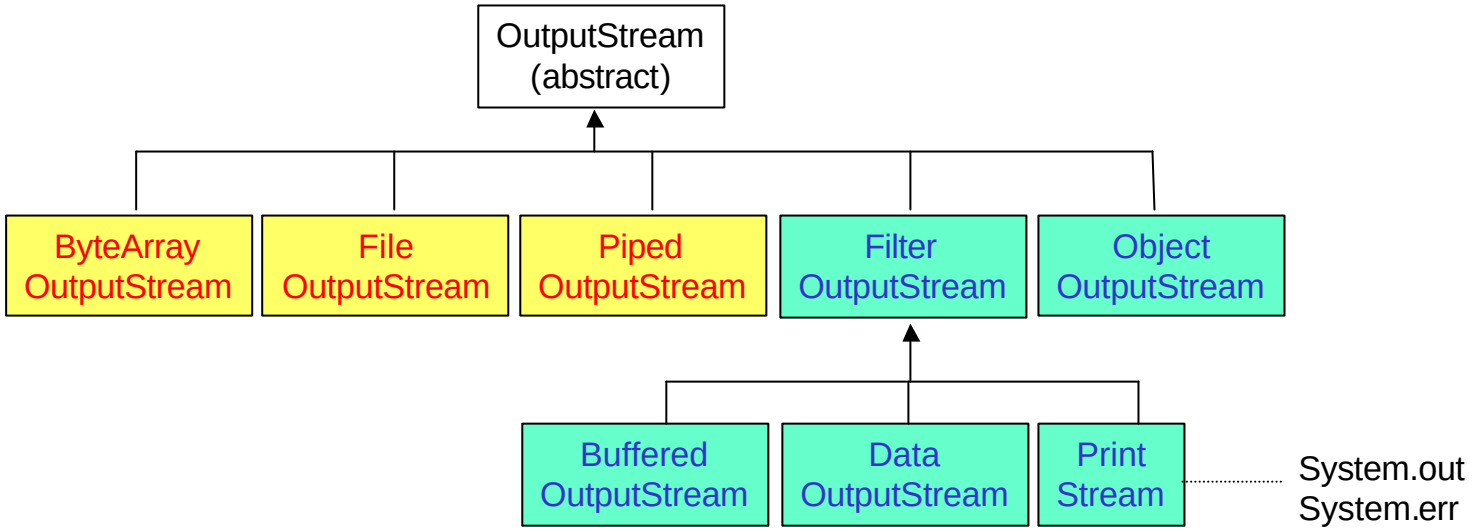
- Arbeite mit den komfortablen Methoden des Processingstream-Objektes.
  Beispiel:
```
// Lese einen double-Wert aus Datei
double d = dis.readDouble();
```

Hierarchie der wichtigsten InputStream-Klassen



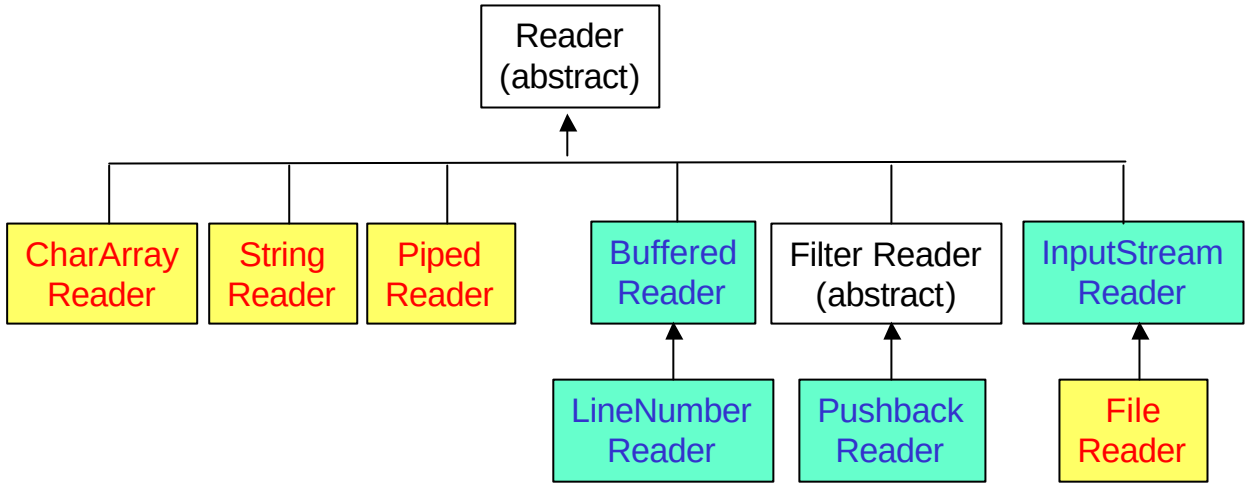
- `ByteArrayInputStream`: Auslesen von Bytes aus einem Objekt vom Typ "Feld von Bytes"
- `FileInputStream`: Auslesen von Bytes aus einer externen Datenquelle (z.B. Datei)
- `PipedInputStream`: Kommunikation von Threads über Pipes
- `FilterInputStream`: im Wesentlichen Oberklasse für 3 Unterklassen
- `BufferedInputStream`: gepufferte Eingabe (evtl. effizienter)
- `DataInputStream`: primitive Datentypen (int, float,...) binär einlesen (big endian)
- `PushBackInputStream`: 1 Byte lesen und wieder zurückstellen möglich
- `ObjectInputStream`: (komplexe) Objekte binär einlesen
- `SequenceInputStream`: Hintereinander Einlesen von mehreren Eingabe-Streams wie von einem logischen Stream

Hierarchie der wichtigsten OutputStream-Klassen



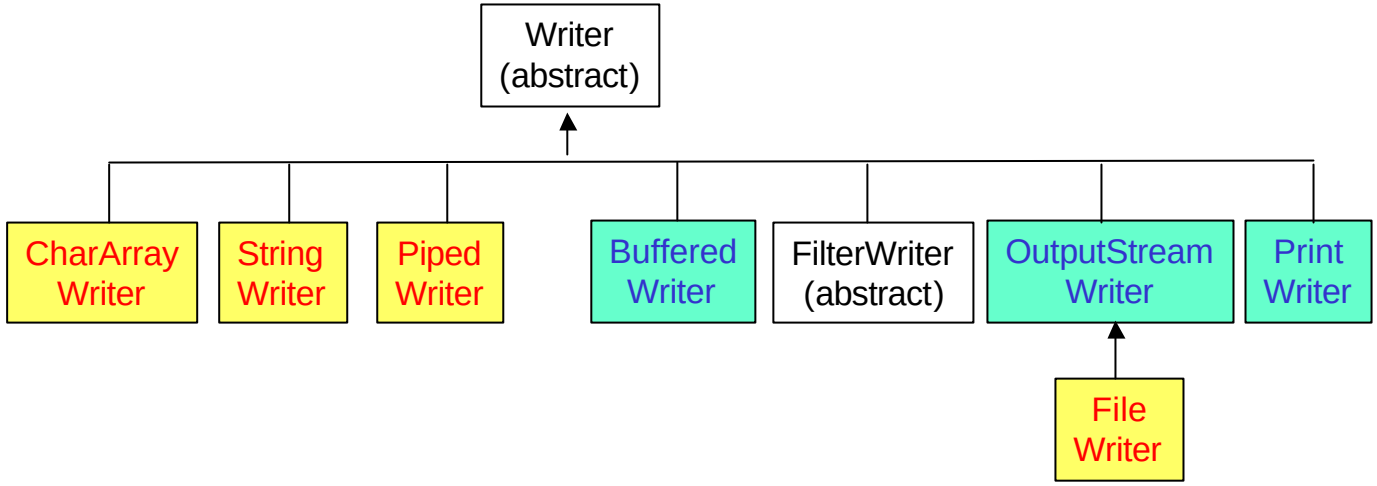
- `ByteArrayOutputStream`: Schreiben von Bytes in ein Objekt vom Typ "Feld von Bytes"
- `FileOutputStream`: Schreiben von Bytes in eine externe Datenquelle (z.B. Datei)
- `PipedOutputStream`: Kommunikation von Threads über Pipes
- `FilterOutputStream`: im Wesentlichen Oberklasse für 3 Unterklassen
- `BufferedOutputStream`: gepufferte Ausgabe
- `DataOutputStream`: primitive Datentypen (int, float,...) binär ausgeben (big endian)
- `PrintStream`: komfortable Ausgabe über Stringumwandlung (Beispiel: `System.out.println(...)`)
- `ObjectOutputStream`: (komplexe) Objekte binär ausgeben

Hierarchie der wichtigsten Reader-Klassen



- `CharArrayReader`: Lesen von Zeichen aus einem Objekt vom Typ "Feld von char"
- `StringReader`: Lesen von Zeichen aus einem String
- `PipedReader`: Kommunikation von Threads über Pipes
- `BufferedReader`: gepuffertes Lesen von Zeichen
- `LineNumberReader`: Einlesen von Zeichen mit Abfragemöglichkeit für Zeilennummer
- `FilterReader`: Basis für eigene Filterklassen
- `PushbackReader`: Zurückstellen von Zeichen möglich
- `InputStreamReader`: Brückenklasse (`byte` nach `char`)
- `FileReader`: "Bequeme" Brückenklasse

Hierarchie der wichtigsten Writer-Klassen



- `CharArrayWriter`: Schreiben von Zeichen in ein Objekt vom Typ "Feld von char"
- `StringWriter`: Schreiben von Zeichen in einen String
- `PipedWriter`: Kommunikation von Threads über Pipes
- `BufferedWriter`: gepufferte Zeichenausgabe
- `FilterWriter`: Basis für eigene Filterklassen
- `OutputStreamWriter`: Brückenklasse (`char` nach `byte`)
- `FileWriter`: "Bequeme" Brückenklasse
- `PrintWriter`: komfortable Ausgabe über Stringumwandlung

Anmerkung: `Reader/Writer` sind nicht direkt mit externen Datenquellen möglich, sondern nur über Brückenklassen (später).

Übersicht Springstream-/Sinkstream-Klassen

Quelle/Senke	byte-orientiert	zeichenorientiert
byte/char-Feld (intern)	`ByteArrayInputStream` `ByteArrayOutputStream`	`CharArrayReader` `CharArrayWriter`
String (intern)	(deprecated) (deprecated)	`StringReader` `StringWriter`
Pipe (intern)	`PipedInputStream` `PipedOutputStream`	`PipedReader` `PipedWriter`
Datei (extern)	`FileInputStream` `FileOutputStream`	`FileReader` `FileWriter`

Details zu allen Ein-/Ausgabeklassen sind der API-Dokumentation zu entnehmen.

Übersicht Processingstream-Klassen

Funktion	byte-orientiert	zeichenorientiert
Pufferung	`BufferedInputStream` `BufferedOutputStream`	`BufferedReader` `BufferedWriter`
Zurückschreiben	`PushbackInputStream`	`PushbackReader`
Textausgabe	`PrintStream`	`PrintWriter`
Filter	`FilterInputStream` `FilterOutputStream`	`FilterReader` (abstract) `FilterWriter` (abstract)
Zeilen zählen		`LineNumberReader`
Ein-/Ausgabe von Daten primitiven Typs	`DataInputStream` `DataOutputStream`	
Objektserialisierung	`ObjectInputStream` `ObjectOutputStream`	
Stream-Verkettung	`SequenceInputStream`	
Brückenklassen	`InputStreamReader` `OutputStreamWriter`	

Beispiel 1: Gepufferte Ein-/Ausgabe einzelner Bytes

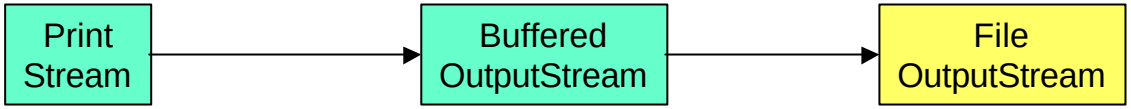
```
import java.io.*;
class Test1 {
    public static void main(String[] args) throws IOException {
        // Stream-Objekte erzeugen (Sinkstream und Processingstream)
        FileOutputStream fos = new FileOutputStream("Datei1.dat");
        BufferedOutputStream bos = new BufferedOutputStream(fos, 5);

        // Binäres Schreiben und anschließend Datei schließen
        for(int i=0; i<10; i++)
            bos.write(i);
        bos.flush();
        bos.close();

        // Stream-Objekte erzeugen (Springstream und Processingstream)
        FileInputStream fis = new FileInputStream("Datei1.dat");
        BufferedInputStream bis = new BufferedInputStream(fis, 2);

        // Binäres Lesen und anschließend Datei schließen
        int b;
        for(int i=0; i<10; i++)
            if((b = bis.read ()) == -1)
                System.out.println("Fehler: Ende der Datei erreicht");
            else
                System.out.println(b);
        bis.close();
    }
}
```

Beispiel 2: Binäre Ein-/Ausgabe primitiver Datentypen

```
import java.io.*;
class Test2 {
    public static void main(String[] args) throws IOException {
        // Stream-Objekte erzeugen (Sinkstream und Processingstream)
        FileOutputStream fos = new FileOutputStream("Datei2.dat");
        DataOutputStream dos = new DataOutputStream(fos);

        // Binäres Schreiben und anschließend Stream schließen
        dos.writeInt(4711);
        dos.writeDouble(3.1415);
        dos.writeChar('a');
        dos.close();

        // Stream-Objekte erzeugen (Springstream und Processingstream)
        FileInputStream fis = new FileInputStream("Datei2.dat");
        DataInputStream dis = new DataInputStream(fis);

        // Binäres Lesen und anschließend Stream schließen
        int i = dis.readInt();
        double d = dis.readDouble();
        char c = dis.readChar();
        dis.close();
    }
}
```

Z.B. binäre Ausgabe von 3: `00000000 00000000 00000000 00000011` (32 Bits)

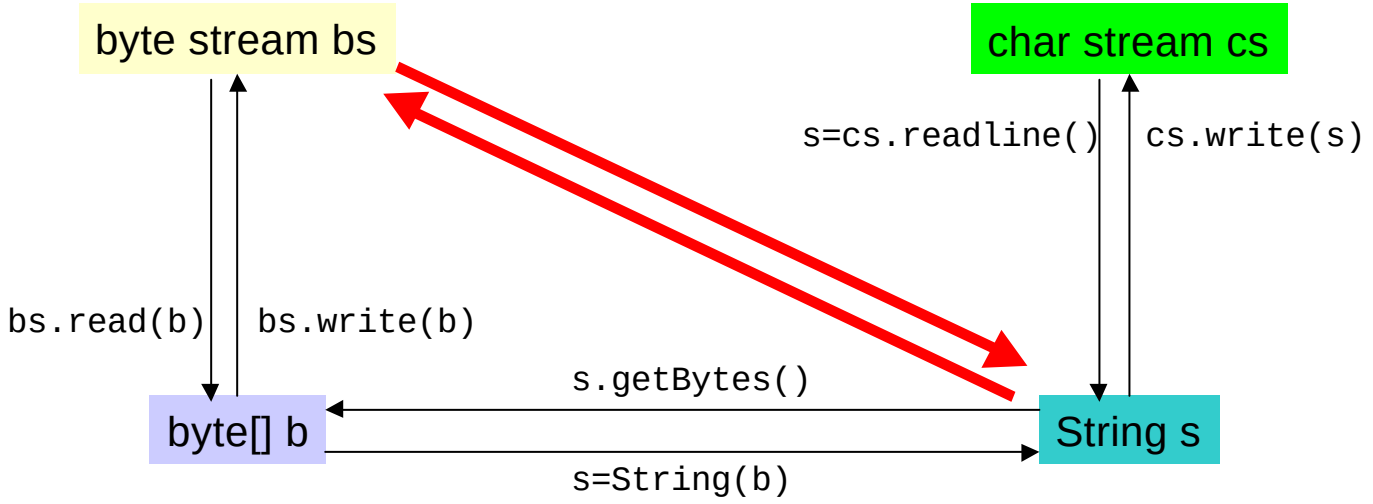
Beispiel 3: Schachtelung von Processingstreams




```
import java.io.*;
class Test3 {
    public static void main(String[] args) throws IOException {

        // Stream-Objekt erzeugen
        FileOutputStream fos = new FileOutputStream("Datei3.dat");
        BufferedOutputStream bos = new BufferedOutputStream(fos, 10);

        // Aufgrund der Vererbung kann sich bos wie ein OutputStream verhalten
        PrintStream pos = new PrintStream(bos);

        // Schreiben (gepuffert und komfortabel)
        for(int i=0; i<1000; i++)
            pos.println("i hat den Wert " + i);

        // Puffer entleeren und Stream schließen
        pos.flush();
        pos.close();
    }
}
```

Brückenklassen

Durch Nutzung der Brückenklasse `InputStreamReader` und Unterklasse `FileReader` kann man von einem byte-orientierten Eingabestrom (Springstream-Objekt) direkt Zeichen lesen.
Durch Nutzung der Brückenklasse `OutputStreamWriter` und Unterklasse `FileWriter` kann man Zeichen direkt in einem Byte-Stream (Sinkstream-Objekt) schreiben.
Die Umsetzung zwischen Zeichen und Bytes wird über ein Encoding (z.B. ASCII, ISO-8859-1, UTF-8) gesteuert.



Beispiel 4: Brückenklassen

```
import java.io.*;
class Test4 {
    public static void main(String[] args) throws IOException {

        // Sinkstream-Objekt erzeugen
        FileOutputStream fos1 = new FileOutputStream("Datei4a.dat");
        FileOutputStream fos2 = new FileOutputStream("Datei4b.dat");

        // Brückenklassenobjekte erzeugen mit Default-Encoding bzw. ASCII-Encoding
        OutputStreamWriter osw1 = new OutputStreamWriter(fos1);
        OutputStreamWriter osw2 = new OutputStreamWriter(fos2, "US-ASCII");

        // Schreiben eines Unicode-Strings auf die Byte-Ströme
        osw1.write("Äh, grützi miteinand");
        osw2.write("Äh, grützi miteinand");

        // Streams schließen
        osw1.close();
        osw2.close();
    }
}
```

Ein-/Ausgabe beliebiger Objekte

Bis jetzt haben wir nur Objekte primitiver Typen bearbeitet. Kann man auch komplexe Objekte ausgeben/einlesen und wenn ja, wie?

Antwort: Man kann!

Lösung: Objektserialisierung

- Die Objektklasse muss die Marker-Schnittstelle `Serializable` implementieren (Marker: Schnittstelle ist leer und dient nur zur Markierung)
- Das Schreiben eines Objektes in einen Stream vom Typ `ObjectOutputStream` geschieht über die Methode `writeObject()`
- Das Lesen eines Objektes aus einem Stream vom Typ `ObjectInputStream` geschieht über die Methode `readObject()`
- Beim Einlesen muss man die Referenz auf das eingelesene Objekt explizit casten. Wieso?

Beispiel: Klasse unterstützt Serialisierung

```
import java.io.*;
public class Person implements Serializable {

    private String name;
    private String vorname;
    private int alter;

    public Person (String n, String vn, int a) {
        this.name = n;
        this.vorname = vn;
        this.alter = a;
    }

    public void ausgeben() {
        System.out.println(vorname + " " + name + ": " + alter);
    }
}
```

Beispiel: Testprogramm

```
public class SeriTest {
    public static void main(String[] args)
      throws IOException, ClassNotFoundException {

        String name = "Datei5.dat";
        FileOutputStream fos = new FileOutputStream(name);
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        Person p1 = new Person("Adams", "Bryan", 67);
        Person p2 = new Person("Mouse", "Mickey", 82);

        // Objekte serialisiert ausgeben
        oos.writeObject(p1);
        oos.writeObject(p2);

        oos.close();

        FileInputStream fis = new FileInputStream(name);
        ObjectInputStream ois = new ObjectInputStream(fis);

        // Objekte serialisiert einlesen (Cast notwendig)
        Person q1 = (Person) ois.readObject();
        Person q2 = (Person) ois.readObject();
        q1.ausgeben();
        q2.ausgeben();
    }}
```

Funktionsweise

Die Ausgabe-Serialisierung geschieht in der Methode `writeObject()` nach folgendem Schema:

- Prüfung, ob das Objekt die Schnittstelle `Serializable` implementiert
- Eindeutiger Identifikator der Klasse wird in den Strom geschrieben
- Schreiben aller Instanzenvariablen der Reihe nach in den Ausgabestrom. Ist eine Instanzenvariable eine Referenzvariable, so wird **rekursiv** der Vorgang beginnend mit Schritt 1 auf diese Referenz angewandt. Hat eine Instanzenvariable den Modifier `transient`, so wird diese ausgelassen (z.B. bei Passworten sinnvoll).

Die Eingabe-Serialisierung geschieht in der Methode `readObject()` nach folgendem Schema:

- Rekonstruktion der Klasse des Objektes und Laden der Klasse durch die JVM (falls noch nicht erfolgt)
- Anlegen eines Objektes der Klasse. Initialisierung der Instanzenvariablen mit Default-Werten
- Einlesen der Werte der Reihe nach für die Instanzenvariablen (ebenfalls wieder Rekursion bei Referenzvariablen)

Verzeichnis auflisten

Die Klasse `File` aus dem Paket java.io ist eine systemunabhängige Abstraktion von Pfadnamen in einem Verzeichnisbaum. Über diese Klasse lassen sich z.B.die Namen aller Dateien in einem Verzeichnis auflisten.

```
import java.io.*;
public class Test {
    public static void main(String[] args) {

        File verzeichnis = new File(args[0]);
        String dateiNamen[] = verzeichnis.list();

        for(int i=0; i < dateiNamen.length; i++)
            System.out.println("Datei " + i + ": " + dateiNamen[i]);
    }
}
```

Aufruf (hier: Auflisten des aktuellen Verzeichnisses .):

```
> java Test .
Datei 0: Test.java
Datei 1: Test.class
```

Formatierte Ausgabe

Die Ausgabe in Textform geschieht in einem Standardformat. Oft möchte man aber ein eigenes Format vorgeben, z.B. um Tabellen zu erstellen (10 Spalten breit, rechtsbündig, mit führenden Nullen,...)

Lösung:

`java.text.Format`

(mit direkten Unterklassen `DateFormat`, `MessageFormat`, `NumberFormat`)

```
import java.text.*;
public class FormatTest {
        public static void main (String [] args) {
                DecimalFormat df = new DecimalFormat ("0000.00E00");
                double x = 12.345;
                String s = df.format (x);
                System.out.println ("Unformatiert: " + x
                                    + ", Formatiert: " + s);
        }
}
```

Ausgabe:

```
Unformatiert: 12.345, Formatiert: 1234,50E-02
```