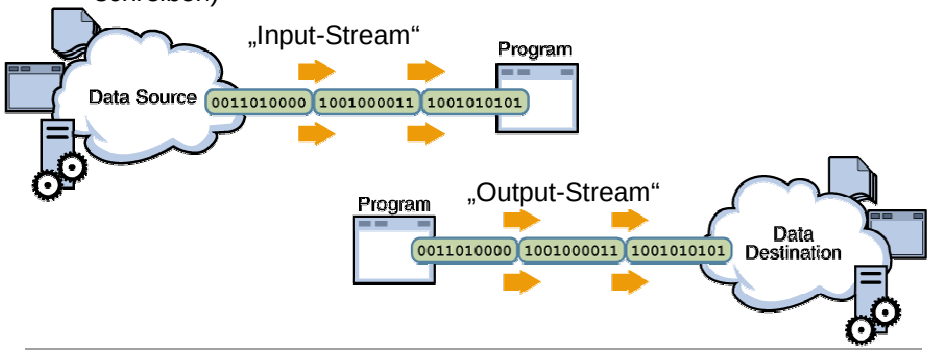
Streams

Thomas Philipp

Prinzip der Datenströme (Streams)

- **Data stream**: Kontinuierliche Abfolge von Daten
- **Ende** eines Datenstroms ist im Voraus **nicht bekannt**
- **Kein direkter Zugriff auf einzelne Datensätze/Elemente** (Prinzip: „Immer schön der Reihe nach“)
- **Daten** können **nur in eine Richtung** gesendet werden (lesen oder schreiben)



Prinzip der Datenströme (Streams)

- **Grundlegende Vorgehensweise**:
  - **Verknüpfung zum Stream herstellen (z.B. Datei öffnen)**
  - **Bearbeiten der Daten (lesen/schreiben)**
  - **Schließen des Streams (z.B. Datei schließen)**
- **Grundprinzip der Streams ist unabhängig von Datenquelle/Speicherort**
  - **Input von: Datei, Tastatur, Netzwerkverbindung, …**
  - **Output nach: Bildschirm, Datei, String, …**

Allgemeines

- **Stream**: Ist eine **Abstraktion der Information**, die von einem „Gerät“ geliefert bzw. an „Gerät“ geschickt wird
- Java: Umfangreiche Klassenbibliothek für Ein-/Ausgabe
- Paket: **java.io.*;**
- **Ein-/Ausgaben** sind von Natur aus **fehleranfällig**
  - Aufrufe von Ein- und Ausgabemethoden müssen mit **try-catch** geklammert werden
  - entsprechende **Exceptions** können ausgelöst werden
  - Exceptions sind Unterklassen von der Klasse **IOException**

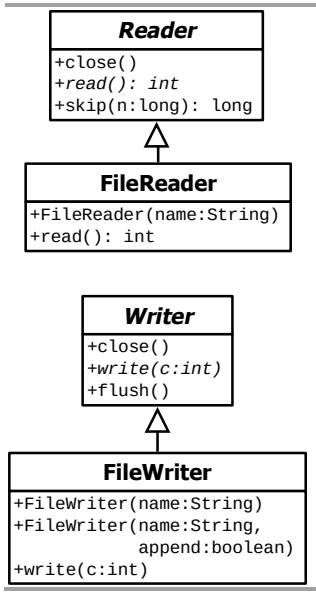
Streams und Java-Klassen

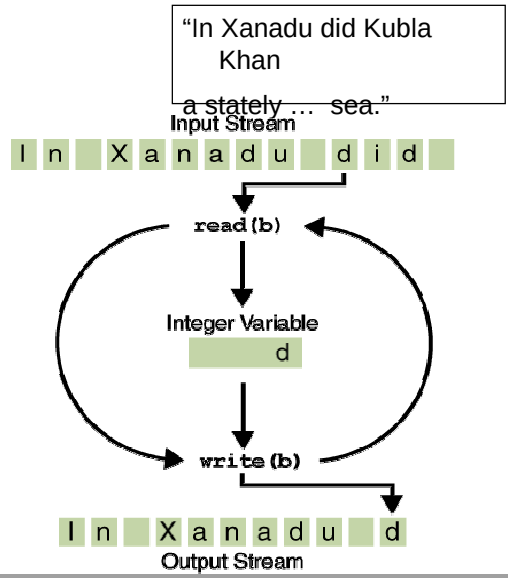
- Unterscheidung nach Art der Daten:
  - **Byte-Streams** (Binärdaten)
  - **Character-Streams** (Text)
- Unterscheidung nach Richtung:
  - **Eingabe-Streams**
  - **Ausgabe-Streams**
- **Java-Oberklassen:**

	Character (Transportbreite 16 Bit)	Byte (Transportbreite 8 Bit)
Ausgabe	Writer	OutputStream
Eingabe	Reader	InputStream

- **Je nach Art des Geräts bzw. Art der Ein-/Ausgabe: Verwendung der passenden Unterklasse dieser 4 Oberklassen**

Beispiel – Textdatei kopieren





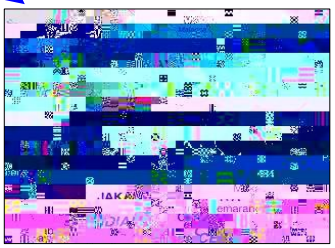
Beispiel – Textdatei kopieren

```
public static void copy(String src, String dest) {
    // Variante 2 mit "try-with-resources" Statement:
    // - Alle im try deklarierten Ressourcen werden am
    //   Blockende automatisch wieder geschlossen!
    try (   FileReader in = new FileReader(src);
            FileWriter out = new FileWriter(dest)) {
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
public static void main(String[] args) {
    copy("xanadu.txt", "xanadu_copy.txt");
}
```


Character-Streams und Binärdaten ?!

```
public static void copy(String src, String dest) {
    try (FileReader in = new FileReader(src);
         FileWriter out = new FileWriter(dest)) {
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } catch (IOException e) {
    }
}
public static void main(String[] args) {
    copy("java.jpg", "java_copy.jpg");
}
```

è Reader/Writer **sind nicht für Binärdaten geeignet**, da gelesene Bytes anhand eines charsets in Unicode-Character umgesetzt werden!



Beispiel: Ausgeben von eingelesenen Texten

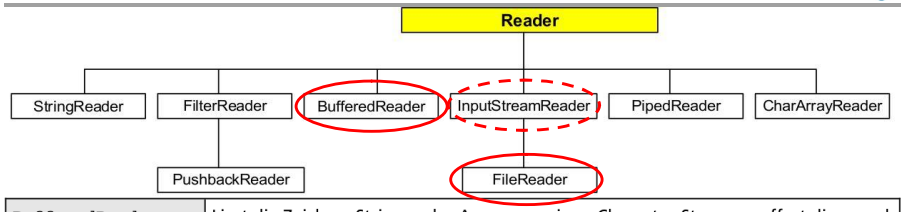
Wo sind in dem Quelltext „Streams“ zu finden?

Vordefiniertes InputStream-Objekt

Vordefiniertes PrintStream-Objekt

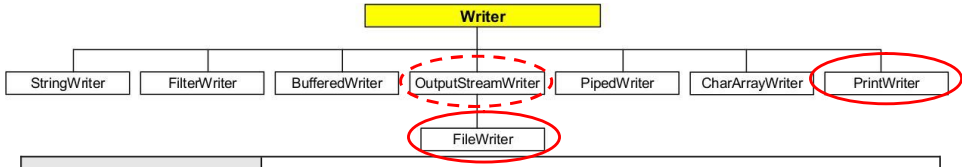
```
…
public static void main(String[] args) {
   Scanner aScanner = new Scanner ( System.in );
   String   zeile;
   do {
       zeile = aScanner.nextLine();

       System.out.println(zeile);
   } while ( !zeile.equalsIgnoreCase("exit") );
   System.out.println("Tschüss...");
}
```


Character-Streams für die Eingabe



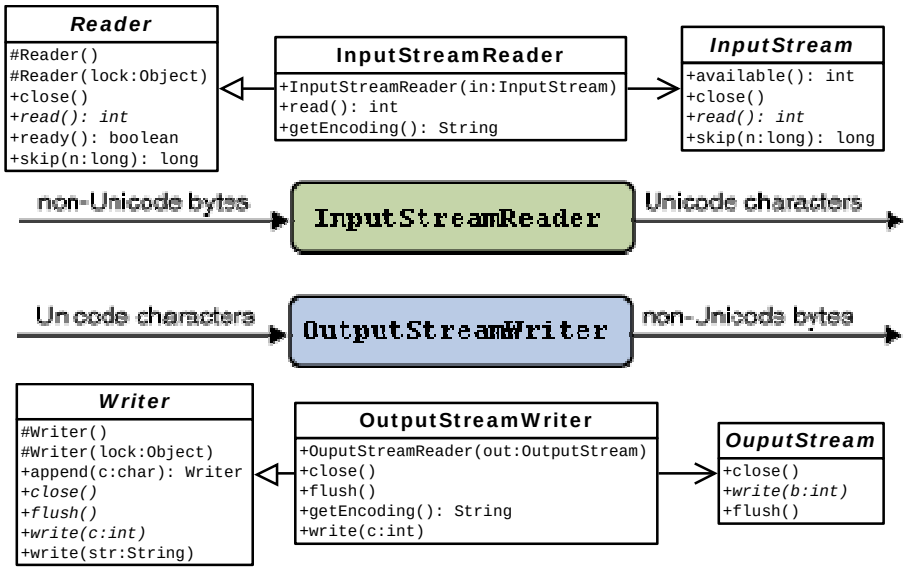
BufferedReader	Liest die Zeichen, Strings oder Arrays aus einen Character-Stream, puffert diese und ermöglicht so ein effektives Lesen
CharArrayReader	Die Zeichen werden in ein Character-Array gelesen.
FilterReader	Filtert Character-Streams während des Lesens (abstrakt)
PushbackReader	Von der Klasse FilterReader abgeleitete Klasse, die die erlaubten Zeichen liest und sie wieder in den Eingabestrom zurückschiebt
InputStreamReader	Der Eingabestrom wird als Byte-Stream gelesen und in einen Character-Stream umgewandelt.
FileReader	Von der Klasse InputStreamReader abgeleitete Klasse, die das Lesen aus einer Datei realisiert
PipedReader	Dient zum Lesen aus einer Pipe. Pipes werden zur Realisierung der Ein-/Ausgabeströme von Threads (gleichzeitig ausgeführte Programmmodule) verwendet. Die Arbeit mit Threads ist nicht Gegenstand dieses Buches.
StringReader	Liest die Zeichen aus einem String-Puffer

Character-Streams für die Ausgabe



BufferedWriter	Schreibt Zeichen, Strings oder Arrays in einen gepufferten Character-Stream und ermöglicht so ein effektives Schreiben
CharArrayWriter	Die Zeichen werden in ein Character-Array geschrieben. Dieser Puffer (das Array) wächst automatisch während des Schreibens.
FilterWriter	Schreibt gefilterte Character-Streams (abstrakt)
OutputStreamWriter	Schreibt die Zeichen in einen Ausgabestrom und wandelt dabei den Character-Stream in einen Byte-Stream um
FileWriter	Eine von OutputStreamWriter abgeleitete Klasse, die die Ausgabe in eine Datei realisiert
PipedWriter	Dient zur Ausgabe in eine Pipe. Pipes werden zur Realisierung der Ein-/Ausgabeströme von Threads (gleichzeitig ausgeführte Programmmodule) verwendet. Die Arbeit mit Threads ist nicht Gegenstand dieses Buches.
PrintWriter	Gibt alle primitiven Datentypen und Strings im Textformat aus
StringWriter	Schreibt Zeichen in einen String-Puffer

Brückenklassen: byte- ó char-Streams



Beispiel: Zeilenweise aus Streams lesen

```
public static void main(String[]
  try {
      BufferedReader br;
      br = new BufferedReader(
                new InputStreamReader(System.in));
      String zeile;
      while ( (zeile=br.readLine()) != null){
            .....;
      }
  } catch (IOException e) {
      e.printStackTrace();
  }
}
```

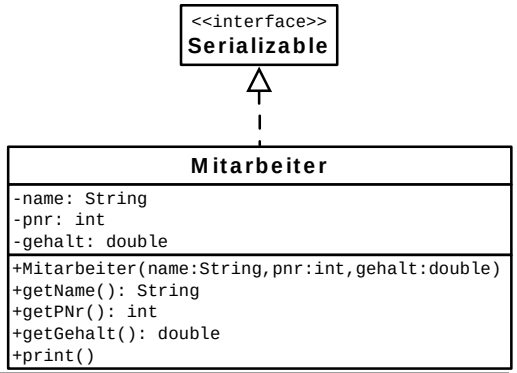
InputStreamReader wird zur Umwandlung eines Byte-Streams (System.in) in einen Char-Stream benötigt

Binärdaten lesen/schreiben: DataStream-Kl.

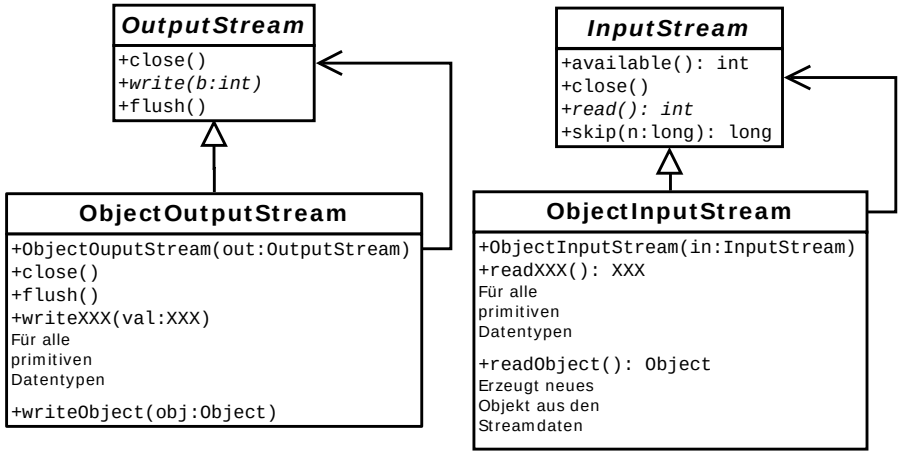
- Mögliche Lösung: Verwendung der Klassen **DataOutputStream** bzw. **DataInputStream** (**è** siehe DataStorage.java)
- **Problem/Wunsch**: Speichern von „ganzen“ Objekten der Klasse Mitarbeiter, nicht nur der einzelnen Elemente (name, pnr, gehalt)

è **ObjectOutputStream / ObjectInputStream**

è **Klasse muss als „serialisierbar“ markiert werden (Interface Serializable)**



ObjectStreams



ObjectStreams

- Bei der Verwendung von **`ObjectStreams`** werden u.U. automatisch ganze Objektbäume geschrieben und gelesen. In der Regel werden dabei nur Klassen berücksichtigt die das **`Interface Serializable`** implementieren.