



# 1-17 Mit Character- und Bytestreams arbeiten

---



SEW 3

DI Thomas Helml



# Inhalt

---

- 17.1 Grundlagen zu Streams
  - 17.2 Character-Streams schreiben
  - 17.3 Character-Streams lesen
  - 17.4 Mit Character-Streams und Textdateien arbeiten
  - 17.5 Character-Streams puffern
  - 17.6 Mit Character-Streams primitive Datentypen schreiben und lesen
  - 17.7 Character-Streams filtern
  - 17.8 Character-Streams für Strings und Character-Arrays
  - 17.9 Mit Byte-Streams arbeiten
  - 17.10 Übung
  - 17.11 Objektserialisierung
-



# 17.1 Grundlagen zu Streams

---

## II Input-Stream (Eingabestrom)

- IT kann Daten aus beliebiger Quelle lesen (Tastatur, Datei, String, Netzwerk ...)

## II Output-Stream(Ausgabestrom)

- IT kann Daten in beliebiges Ziel schreiben
- IT Streams sind gleich zu behandeln, unabhängig von Quelle/Ziel
- IT Stream geht immer nur in eine Richtung



# Stream-Typen

---

II Man unterscheidet:

IT **Character-Streams**

IT transportieren Unicode-Zeichen (16 Bit); Ein-/Ausgabe von Text

IT **Byte-Streams**

IT transportieren Byte(s) (je 8 Bit)



# Streams und ihre Klassen

Stream-Typ	Transportbreite	Für die Eingabe verwendete ...		Für die Ausgabe verwendete ...	
		... Superklasse	... Interfaces	... Superklasse	... Interfaces
<b>Character-Stream</b>	16 Bit (1 Unicode-Zeichen)	Reader	Closable, Readable	Writer	Closable, Flushable, Appendable
<b>Byte-Stream</b>	8 Bit (1 Byte)	InputStream	Closable	OutputStream	Closable, Flushable

## ④ Interfaces:

- ④ **Closable** zum Schließen von Ein- bzw. Ausgabeströmen
- ④ **Readable** zum Lesen von Character-Streams
- ④ **Flushable** zum Leeren von Ausgabeströmen
- ④ **Appendable**, um Daten an Character-Streams anzuhängen



# Fehlerbehandlung

---

- II I/O Operationen können Fehler verursachen
  - IT z.B. beim Öffnen, Lesen, Schreiben, Schließen
  - IT Fehler lösen IOException aus
  - IT müssen behandelt werden (try-catch)



# Schließen von Ressourcen

---

- alle geöffneten Ressourcen müssen geschlossen werden
  - korrektes Schließen ist aufwendig
- Ab Java 7: **try with resources** (selbstschließende Ressourcen)

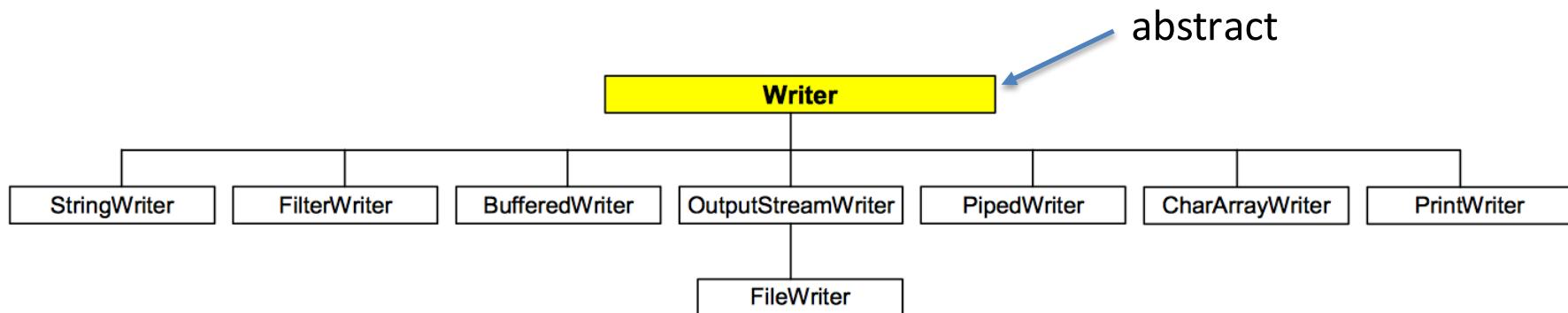
```
try (OutputStream out =
      new FileOutputStream(...)) {
    out.write(...);
}
```

- die in `try (...)` geöffnete Ressource wird automatisch geschlossen

## 17.2 Character-Streams schreiben

---

### II Hierarchie der Klassen für die Ausgabe





# Methoden der abstrakten Superklasse Writer

<code>Writer()</code>	Konstruktor der Klasse <code>Writer</code> , der das Öffnen des Ausgabestroms realisiert
<code>close()</code>	Schließt den Ausgabestrom
<code>flush()</code>	Leert den Ausgabestrom
<code>write(char[ ] cbuf, int off, int len)</code>	Schreibt einen Teil eines Arrays von Zeichen
<code>write(char[ ] cbuf)</code>	Schreibt ein Array von Zeichen in den Ausgabestrom
<code>write(int c)</code>	Schreibt ein einzelnes Zeichen (2 Byte) in den Ausgabestrom, wobei als Zeichen die zwei niederwertigen Bytes des <code>int</code> -Parameters interpretiert werden
<code>write(String str)</code>	Schreibt eine Zeichenkette in den Ausgabestrom
<code>write(String str, int off, int len)</code>	Schreibt beginnend beim Offset <code>off</code> eine Zeichenkette der Länge <code>len</code> in den Ausgabestrom
<code>append(char c)</code>	Hängt ein Zeichen an; Rückgabewert: <code>Writer</code>
<code>append(CharSequence csq)</code>	Hängt eine Zeichenfolge an; Rückgabewert: <code>Writer</code> . <code>CharSequence</code> beschreibt eine nur lesbare Folge von Zeichen.
<code>append(CharSequence csq, int start, int end)</code>	Hängt einen Teil einer Zeichenfolge an, beginnend mit dem Zeichen an der Stelle des Index <code>start</code> und endend ein Zeichen vor dem Index <code>end</code> ; Rückgabewert: <code>Writer</code>

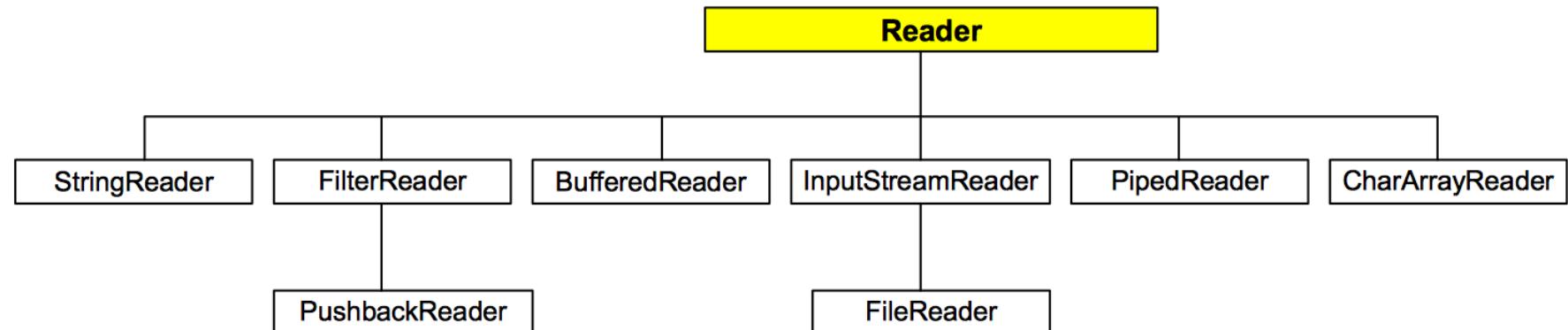


# Abgeleitete Klassen der Klasse Writer

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-/Aus-gabeströme von Threads (gleichzeitig ausgeführte Programmmoduln) 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

# 17.3 Character-Streams lesen

## II Hierarchie der Klassen für die Eingabe





# Methoden der abstrakten Superklasse Reader

<code>Reader()</code>	Konstruktor, der das Öffnen des Eingabestroms realisiert
<code>close()</code>	Schließt den Eingabestrom
<code>mark()</code>	Markiert die aktuelle Position im Eingabestrom
<code>boolean markSupported()</code>	Überprüft, ob der Eingabestrom Markierungen unterstützt
<code>reset()</code>	Zurücksetzen des Eingabestroms
<code>long skip(long n)</code>	Überspringt die nächsten <code>n</code> Zeichen, wobei <code>n</code> größer als 0 sein muss
<code>int read()</code>	Liest ein einzelnes Zeichen aus dem Eingabestrom und gibt dieses als <code>int</code> -Wert zurück. Gibt die <code>read</code> -Methode den Wert -1 zurück, ist das Dateiende erreicht. Für Leseoperationen in der Schleife können Sie diese Eigenschaft als Abbruchkriterium nutzen.
<code>int read(char[] cbuf)</code>	Liest ein oder mehrere Zeichen aus dem Eingabestrom in das als Parameter übergebene <code>char</code> -Array. Der zurückgegebene <code>int</code> -Wert entspricht der Anzahl der übertragenen Zeichen.
<code>int read(char[] cbuf, int off, int len)</code>	Liest Zeichen aus dem Eingabestrom in einen Teil des als Parameter übergebenen <code>char</code> -Arrays



# Abgeleitete Klassen der Klasse Reader

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-/Ausgabe- ströme von Threads (gleichzeitig ausgeführte Programmmodulen) verwendet.
StringReader	Liest die Zeichen aus einem String-Puffer



## 17.4 Mit Character-Streams und Textdateien arbeiten

---

- ⑩ OutputStreamWriter ist abstrakte Klasse
- ⑩ FileWriter VON OutputStreamWriter abgeleitet
  - ⑩ wird für Ausgabe in Dateien eingesetzt
  - ⑩ Konstruktoren:
    - ⑩ FileWriter(String fileName)
    - ⑩ FileWriter(String fileName, boolean append)
    - ⑩ FileWriter(File file)
  - ⑩ Datei wird im aktuellen Benutzerverzeichnis erzeugt oder überschrieben
  - ⑩ Aktuelles Benutzerverzeichnis auslesen:
    - ⑩ System.getProperty("user.dir")



# Ein FileWriter-Objekt erzeugen

```
try {  
    FileWriter fw = new FileWriter("testCharFile.dat");  
    fw.write("1. Zeile: Test Ausgabe \r\n2. Zeile: in eine Datei");  
    fw.close();  
} catch (IOException ioex){  
    System.out.println(ioex.getMessage());  
}
```

- alle I/O Operationen müssen innerhalb eines try-catch Blocks stehen
- Konstruktor erzeugt File, sofern nicht vorhanden, dann wird es überschrieben
  - `write(...)` schreibt String in Datei
  - mit `close()` wird die Datei geschlossen



# Die Klassen InputStreamReader und FileReader

---

- ④ InputStreamReader ist abstrakte Klasse
- ④ FileReader VON InputStreamReader abgeleitet
  - ④ wird für Lesen aus Dateien eingesetzt
  - ④ Konstruktoren:
    - ④ FileReader(String fileName)
    - ④ FileReader(File file)



# FileReader

---

```
String text = "";  
int x = 0;  
try {  
    FileReader fr = new FileReader("testCharFile.dat");  
    while ((x = fr.read()) != -1)  
        text += (char)x;  
    fr.close();  
} catch (IOException io){  
    System.out.println(io.getMessage());  
}  
System.out.println(text);
```

- IT read() liest zeichenweise ein



# Brückenklassen zur Konvertieren von Character- in Byte-Streams

---

- ④ InputStreamReader und OutputStreamWriter stellen sogenannte **Brückenklassen** zwischen Character-Streams und Byte-Streams dar
    - ④ beim Lesen erfolgt Umwandlung von Byte-Streams in Character-Streams
    - ④ beim Schreiben werden Character-Streams in Byte-Streams umgewandelt
  - ④ durch Umwandlung von Unicode-Zeichen in Bytes wird Platzbedarf beim Speichern reduziert
-



## 17.5 Character-Streams puffern

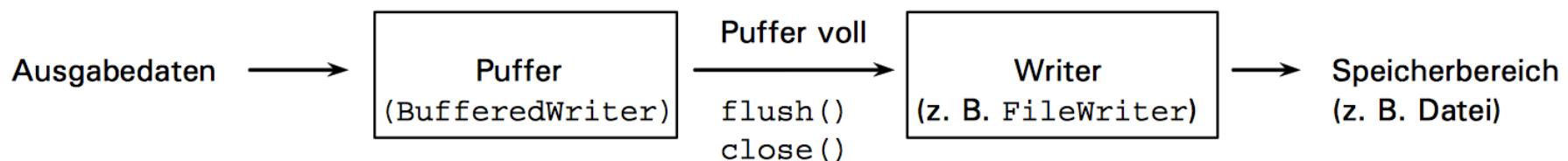
---

- ④ Ein/Ausgabeströme werden aufgrund der Performance meist gepuffert
- ④ Zwischenspeicherung in Puffer (Cache)
- ④ Weiterleitung, wenn Puffer voll oder Puffer geleert wird (flush)
- ④ Klassen: BufferedReader/BufferedWriter

# Character-Streams zur Pufferung von Ausgabeströmen schachteln

---

- ④ für Pufferung von Ausgabeströmen ist es erforderlich, Character-Streams zu schachteln
- ④ `BufferedWriter()` muss anderes Writer-Objekt übergeben werden (z.B. `FileWriter`)
- ④ wird `write` von `BufferedWriter()` aufgerufen, so wird das erst bei Leerung des Puffers an `FileWriter` weitergeleitet





# Konstruktoren von BufferedWriter

---

- ④ BufferedWriter(Writer out)
- ④ BufferedWriter(Writer out, int sz)
  - ④ sz gibt die Größe des Puffers an



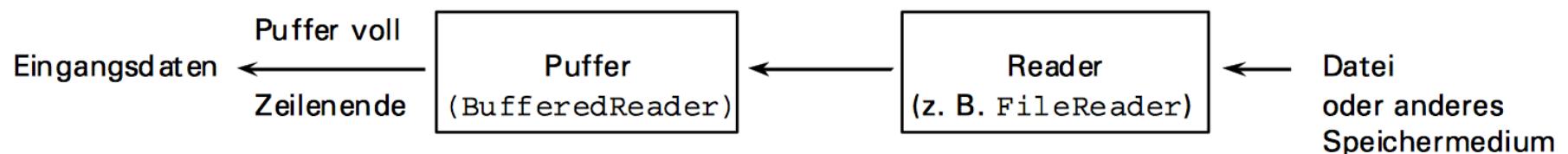
# BufferedWriter Beispiel

```
char[] ca = {'k', 'n', 'r', '0', '1'};  
try {  
    BufferedWriter bw = new BufferedWriter(new FileWriter("testCharBuffer.dat"));  
  
    for (int i = 0; i < 100; i++) {  
        bw.write(ca);  
        bw.newLine();  
    }  
    bw.close();  
} catch(IOException io){...}
```

- IT Character Array wird 100x in Datei ausgegeben
- IT erst mit .close() werden die Daten geschrieben

# Eingabeströme puffern

- II Prinzip gleich wie bei Ausgabestrom



- IT mehrere Bytes gleichzeitig lesen -> Lesegeschwindigkeit steigt
- IT Konstruktoren analog zu BufferedWriter:
  - IT BufferedReader(Reader in)
  - IT BufferedReader(Reader in, int sz)



# Eingabeströme puffern

- ④ **String readLine()**
  - ④ zusätzliche Methode (nur bei BufferedReader)
  - ④ liest Textzeile ein und gibt String zurück
  - ④ Zeilenende wird \n und \r\n erkannt, aber nicht zurückgegeben
  - ④ wird nichts gelesen (z.B. Ende der Datei) wird null zurückgegeben



# Eingabeströme puffern

```
String s;
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("testCharBuffer.dat"));
    try {
        while ((s = br.readLine()) != null)
            System.out.println(s);
    } finally {
        if (br != null)
            br.close();
    }
} catch (IOException io) {
    System.out.println(io.getMessage());
}
```

- ⚠ **try {} finally {} -> tritt Fehler beim Lesen auf, wird Ressource ordnungsgemäß geschlossen**



# Eingaben von der Konsole als String lesen

- II mit BufferedReader kann String von der Konsole eingelesen werden

```
BufferedReader br = new BufferedReader(  
        new InputStreamReader(System.in));  
String s = br.readLine();
```

- IT InputStreamReader erledigt Konvertierung in Unicode
- IT System.in -> Standardeingabe
- IT readLine() liest zeilenweise bis ENTER



## 17.6 Mit Character-Streams primitive Datentypen schreiben und lesen

---

- ④ Primitive Datentypen schreiben
  - ④ in bisherigen Writer-Klassen gibt es nur `write()`
  - ④ PrintWriter besitzt eigene Methoden für primitive Datentypen
    - ④ `print(boolean b)`
    - ④ `print(double d)`
    - ④ `print(Object obj)`
    - ④ `print(String s)`
    - ④ `...`



# Primitive Datentypen schreiben

- II Methoden `println` zusätzlich noch Zeilenende
  - IT `println(boolean b)`
  - IT `println(double d)`
  - IT `println(Object obj)`
  - IT ...
- IT `printf` ermöglicht formatierte Ausgabe (wie in C):
  - IT `printf(String format, Object...args)`



# Primitive Datentypen schreiben

```
try {  
    PrintWriter pw = new PrintWriter("testCharPrint.dat");  
    //String schreiben  
    pw.println("Ausgabe des Flächeninhalts für Kreise mit");  
    for (int r = 1; r <= 10; r++) {  
        pw.print("Radius r = " + r + ": "); //String schreiben  
        pw.println(PI * r * r);           //double-Wert schreiben  
    }  
    pw.close();  
} catch (IOException io){  
    System.out.println(io.getMessage());  
}
```

## Alternativ:

```
pw.printf("Radius r = %d: %g%n", r, PI * r * r); //formatierte Ausgabe
```



# Primitive Datentypen lesen

---

- II Vordefinierte Klasse zum Lesen von primitiven Datentypen **gibt es nicht!**
  - II Eingabe von String: BufferedReader
  - II Zahlen eingeben: als String einlesen und dann konvertieren



# Primitive Datentypen lesen

```
try {  
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
    System.out.println("Geben Sie eine Integerzahl ein: ");  
    // String einlesen  
    String s = br.readLine();  
    // String umwandeln/parsen in entsprechende Wrapper-Klasse  
    int x = Integer.parseInt(s);  
    System.out.println("Die eingegebene Zahl heisst: " + x);  
  
} catch (IOException io) {  
    System.out.println(io.getMessage());  
    // Exception tritt auf, wenn keine Integer-Zahl eingegeben wird -> Behandlung  
} catch (NumberFormatException ex){  
    System.out.println("Fehlerhafte Integerzahl gelesen: " + ex.getMessage());  
}
```



## 17.7 Character-Streams filtern

---

- ④ FilterWriter/FilterReader (abstrakt!) können eigene Filter definiert werden
  - ④ Beispiel: Kleinbuchstaben in Großbuchstaben umwandeln
  - ④ eigene Klasse, die von FilterWriter bzw. FilterReader ableitet
  - ④ read() / write() muss überschrieben werden



# Gelesene Zeichen oder Zeichenfolgen wieder zurückschreiben

---

- ④ PushbackReader leitet von FilterReader ab
- ④ Anwendung:
  - ④ Zeichen/-folge lesen, die nur auf bestimmte Kriterien geprüft werden sollen
  - ④ mit PushbackReader können diese in den Eingabestrom zurück gestellt werden
  - ④ gelesene Zeichen werden in sog. Pushback-Puffer zwischengespeichert
  - ④ zurückschreiben mit Methode `unread()`



## 17.8 Character-Streams für Strings und Character-Arrays

---

- ④ CharArrayWriter/StringWriter schreiben nicht in Datei, sondern in Character- bzw. String-Puffer
  - ④ Puffer wachsen automatisch, wenn Daten in Strom geschrieben werden
  - ④ Konstrukturen
    - ④ CharArrayWriter()
    - ④ CharArrayWriter(int initialSize)
    - ④ StringWriter()
    - ④ StringWriter(int initialSize)



# In Strings und Character-Arrays schreiben

## II Methoden der Klasse CharArrayWriter

<code>char[] toCharArray()</code>	Kopiert die Daten des Streams in ein Character-Array
<code>String toString()</code>	Konvertiert die Daten des Streams in einen String
<code>void writeTo(Writer out)</code>	Schreibt die Daten des Streams in einen anderen Character-Stream
<code>int size()</code>	Gibt die aktuelle Puffergröße zurück
<code>void reset()</code>	Leert den Puffer

## II Methoden der Klasse StringWriter

<code>StringBuffer getBuffer()</code>	Gibt den aktuellen Inhalt des Streams als String-Buffer zurück
<code>String toString()</code>	Konvertiert den aktuellen Inhalt des Streams in einen String



# In Strings und Character-Arrays schreiben

---

```
String s = "Java macht Spass";
StringWriter sw = new StringWriter();
int len = s.length();
for (int i = 0; i < len; i++) {
    s = s.substring(0, s.length() - 1);
    sw.write(s + "\n");
}
System.out.println(sw.toString());
```



# Aus Strings und Character-Arrays lesen

---

## II Klassen zum Lesen: CharArrayReader / StringReader

### III Konstrukturen:

- `CharArrayReader(char[ ] buf)`
- `CharArrayReader(char[ ] buf, int offset, int length)`
- `StringReader(String s)`



# Aus Strings und Character-Arrays lesen

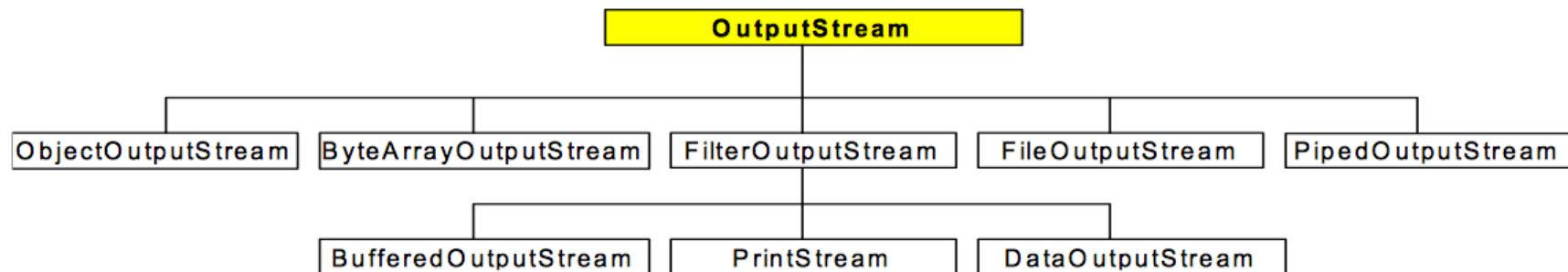
---

```
String s = "Java macht Spass";
StringReader sr = new StringReader(s);
int z;
try {
    while ((z = sr.read()) != -1)
        System.out.println((char)z);
} catch (IOException io){
    System.out.println(io.getMessage());
}
```

## 17.9 Mit Byte-Streams arbeiten

---

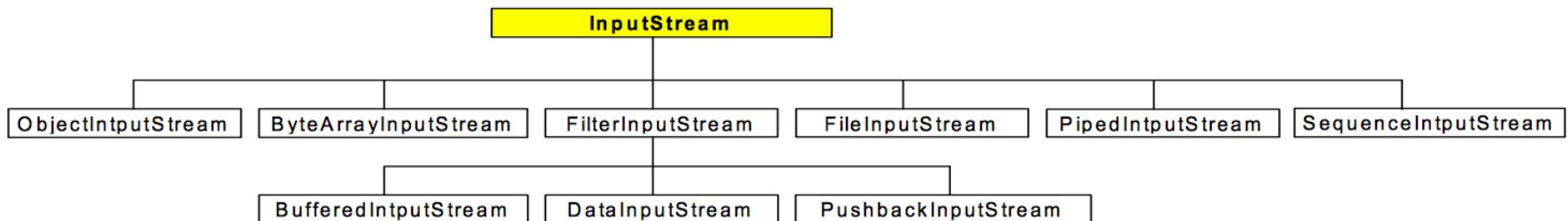
- II Byte-Streams transportieren je 8 Bit (1Byte)
- II alle Klassen sind von `OutputStream` abgeleitet



# Hierarchie der Klassen für die Eingabe

---

- II alle Klassen für Eingabe von Byte-Streams sind von `InputStream` abgeleitet





# Klassen für Byte- und Character-Streams im Vergleich

---

- II Für viele Klassen der Byte- bzw. Character-Streams gibt es Pendants
  - IT gleiche Methodenaufrufen
  - IT unterschiedliche Verarbeitungsbreite (8/16Bit)
  - IT Ausnahmen:
    - T `StringReader/StringWriter`
    - T `InputStreamReader/OutputStreamWriter`
    - T `ObjectInputStream/ObjectOutputStream`



# Klassen für Byte- und Character-Streams im Vergleich

Ausgabe-Streams		Eingabe-Streams	
Byte-Streams	Character-Streams	Byte-Streams	Character-Streams
OutputStream	Writer	InputStream	Reader
FileOutputStream	FileWriter	FileInputStream	FileReader
BufferedOutputStream	BufferedWriter	BufferedInputStream	BufferedReader
ByteArrayOutputStream	CharArrayWriter	ByteArrayInputStream	CharArrayReader
FilterOutputStream	FilterWriter	FilterInputStream	FilterReader
PipedOutputStream	PipedWriter	PipedInputStream	PipedReader
PrintStream	PrintWriter		
		PushbackInputStream	PushbackReader



# Standardeingabe und -ausgabe mit Byte-Streams

- ④ InputStream/PrintStream: verwendet für Standardein- und ausgabe
  - ④ Implementieren die Methoden `read` und `print`

Typ des Byte-Streams	Definition der Klassenkonstanten	Aufruf
Standard-Eingabestrom	<code>public static final InputStream in</code>	<code>System.in</code>
Standard-Ausgabestrom	<code>public static final PrintStream out</code>	<code>System.out</code>
Standard-Fehlerausgabestrom	<code>public static final PrintStream err</code>	<code>System.err</code>



# Standardeingabe und -ausgabe mit Byte-Streams

---

```
byte[ ] b = new byte[1];

try {
    System.out.print("Bitte geben Sie ein Zeichen ein: ");
    System.in.read(b);
    System.out.println((char)b[0] + " hat den ASCII-Code " + b[0]);
} catch (IOException io){
    System.out.println(io.getMessage());
}
```



# Standardeingabe und -ausgabe mit Byte-Streams

- II Falls mehrere Ein-und Ausgaben codiert werden – statischer Import von System:

```
import static java.lang.System.*;  
  
...  
out.print("Bitte geben Sie ein Zeichen ein: ");  
in.read(b);  
out.println((char)b[0] + " hat den ASCII-Code " + b[0]);  
...
```



# Byte-Streams mit den Methoden print und println ausgeben

- II print/println sind für jeden primitiven Datentypen, String und Object definiert:

- II print(char b) / println(char b)
- II print(double d) / println(double d)
- II print(int i) / println(int i)
- II print(Object obj) / println(Object obj)
- II print(String s) / println(String s)
- II ...



# Byte-Streams mit der Methode read lesen

```
int read()  
int read(byte[] b)  
int read(byte[] b, int off, int len)
```

- ohne Parameter: nächstes Zeichen auslesen
- read mit byte-Array => maximal so viele Zeichen gelesen, wie Array aufnehmen kann
  - Der Rest verbleibt im Eingabestrom (Eingabepuffer)
  - Zeichen aus dem Eingabepuffer löschen:  
`System.in.skip(System.in.available());`



# Byte Streams mit Scanner lesen

---

- II Alternativ kann für primitive Datentypen die Klasse Scanner verwendet werden:

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt();  
long l = sc.nextLong();  
double ...
```



# DataInputStream / DataOutputStream

---

- II Findet Verwendung bei: Lesen/Schreiben von Binärdaten in maschinenunabhängiger Form
  - IT readByte() / writeByte(...)
  - IT readInt() / writeInt(...)
  - IT readFloat() / writeFloat(...)
  - IT readUTF() / writeUTF(...)
  - IT ...



# 17.10 Übung Arbeit mit Streams

---

- ① Lesen Sie in einem Programm *Measurements.java* 20 Messreihen zu je 10 Messwerten aus der Datei *valuesIn.dat*. Geben Sie die einzelnen Messwerte durch Leerzeichen getrennt am Bildschirm aus. Schließen Sie jede Messreihe mit einem Hinweis *Ende der Messreihe xx* ab und geben Sie die nächste Messreihe in einer neuen Zeile aus. Schreiben Sie alle Daten in die Datei *valuesOut.dat*.  
Die Ein- und Ausgabe soll gepuffert erfolgen. Die Dateien *valuesIn.dat* und *valuesOut.dat* sollen sich im Ordner *com\herdt\java8\kap17* befinden bzw. dort gespeichert werden.  
Um bei der Bildschirmausgabe alle Zahlen rechtsbündig untereinander zu schreiben, können Sie in der `printf`-Methode die Codierung wie im folgenden Beispiel verwenden: `%4d` bewirkt, dass der angegebene Parameter auf 4 Zeichen Breite rechtsbündig geschrieben wird.
- ② Erstellen Sie eine Anwendung *ReadWriteData.java* zur Erfassung von Personendaten. Dazu sind über die Tastatur folgende Daten abzufragen: Name, Geschlecht, Größe (in cm) und Gewicht (in kg). Verwenden Sie dazu die Klasse `BufferedReader` und nutzen Sie nicht die Klasse `StdInput`!  
Schreiben Sie die eingegebenen Werte in eine Datei *personal.dat* im Ordner *com\herdt\java8\kap17*. Verwenden Sie dazu die überladene Methode `println` der Klasse `PrintWriter`.  
Das Programm soll so gestaltet werden, dass Daten von mehreren Personen erfasst werden können.



# 17.10 Übung Arbeit mit Streams

- ③ Erstellen Sie eine Filterklasse `MyFilterWriter`, welche die Ausgaben in eine Datei so filtert, dass nur Zahlen und Groß- und Kleinbuchstaben in die Datei geschrieben werden. Alle anderen Zeichen sind durch das Zeichen \* zu ersetzen. Gehen Sie dabei wie folgt vor:

- ✓ Leiten Sie Ihre Filterklasse von der Klasse `FilterWriter` ab.
- ✓ Rufen Sie im Konstruktor Ihrer Filterklasse den Konstruktor der Superklasse mit dem übergebenen `Writer`-Objekt auf.
- ✓ Überschreiben Sie die drei `write`-Methoden der Klasse `FilterWriter`.

Testen Sie Ihre Filterklasse in einem Programm `FilterCharacters.java`. Dazu soll über die Tastatur eine Zeichenkette eingegeben werden, die dann gefiltert in die Datei `filter.dat` im Ordner `com\herdt\java8\kap17` geschrieben wird.

Hinweise:

- ✓ Die Filterung braucht nur in der Methode `write(int c)` durchgeführt zu werden. Das zu schreibende Zeichen ist durch Aufruf der `write`-Methode der Superklasse auszugeben. In den anderen beiden `write`-Methoden können Sie dann diese `write`-Methode aufrufen.
- ✓ Die ASCII-Codes für die herauszufilternden Zeichen liegen in folgenden Bereichen:

kleiner 48	und
zwischen 57 und 65	und
zwischen 90 und 97	und
größer 122	
- ✓ Das Zeichen \* hat den ASCII-Code 42.



# 17.10 Übung Arbeit mit Streams

---

- ④ Erstellen Sie auf Basis des Programms `PrintWriterTest.java` das Programm `PrintWriterTest2.java` und verwenden Sie zur Erstellung des PrintWriter-Objekts den Konstruktor mit den Parametern `out` und `autoflush`. Testen Sie den Parameter `autoflush` für die Werte `true` und `false`. Um die Auswirkung des Wertes `false` auf dem Standardausgabegerät zu testen, können Sie beispielsweise eine einfache Zählschleife einfügen.



# Objektserialisierung

---

## II Dauerhafte Speicherung eines Objekts

- IT Textdatei, z.B. CSV
- IT Nachteile:
  - IT Vererbungshierarchie geht u.U. verloren
  - IT Referenzen auf andere Objekte müssen berücksichtigt werden
  - IT Eigenes „Format“ muss überlegt werden



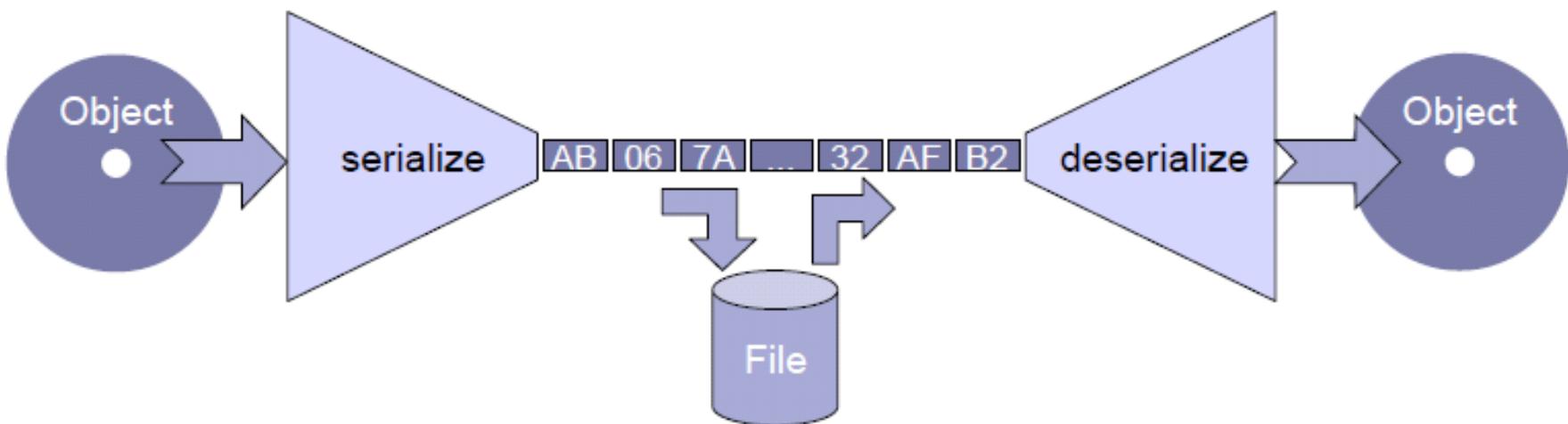
# Objektserialisierung

---

- ④ Speicherung in „genormten“ Format:
    - ④ Struktur, Inhalt, verknüpfte Objekte sind rekonstruierbar
    - ④ Speichern: Objekt-Serialisierung
    - ④ Laden: Objekt-Deserialisierung
  - ④ Serialisierung: Objekte in Strom zu schreiben/lesen
  - ④ Klassen:
    - ④ `java.io.ObjectOutputStream`
    - ④ `java.io.ObjectInputStream`
-

# Objektserialisierung

**Speichern von Objekten in einem Byte-Stream und Wiederherstellen des Objektes aus einem Bytestream.  
(Persistieren von Objekten)**





# Objektserialisierung

---

## II Konstruktoren:

- public ObjectOutputStream(OutputStream out)  
throws IOException
- public ObjectInputStream(InputStream in)  
throws IOException, StreamCorruptedException

II Objekte können mit `writeObject(Object o)` bzw. `Object readObject()` durch beliebige Streams geschickt werden -> werden serialisiert



# Serializable

---

- II Klassen, deren Objekte serialisierbar sein sollen, müssen Interface  
`java.io.Serializable` implementieren
- IT reines „Markerinterface“
- IT Achtung: nicht alle Klassen der Java API sind serialisierbar



# Serialisierung

---

- II Alle serialisierbaren Attribute einer Klasse werden automatisch mit serialisiert
  - IT z.B. bei Collections muss nur die Collection serialisiert werden
- II Ausnahme: statische Attribute werden nicht serialisiert
- II Attribute, die nicht persistiert werden sollen kann man mit transient kennzeichnen



# Serialisierung

Konstruktor	<code>ObjectOutputStream(OutputStream out)</code>
Schreiben primitiver Typen	<code>void writeChar(int v)</code> <code>void writeInt(int v)</code> <code>void writeFloat(float v)</code> ...
Schreiben von Objekten	<code>void writeObject(Object v)</code> // auch Arrays, String, Collections können hier übergeben werden



# Deserialisierung

Konstruktor	<code>ObjectInputStream(InputStream in)</code>
Schreiben primitiver Typen	<code>char readChar()</code> <code>int readInt()</code> <code>float readFloat()</code> ...
Schreiben von Objekten	<code>Object readObject()</code> // Object wird zurückgegeben, daher casten!



# Beispiel

```
// schreiben
try (ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream("objekte.ser"))){
    oos.writeInt(12345);
    oos.writeObject("Heute ist ");
    oos.writeObject(new Schueler());
}
catch (IOException e) {...}

//lesen
try (ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream("objekte.ser"))){
    int i = ois.readInt();
    String s = (String) ois.readObject();
    Schueler s = (Schueler) ois.readObject();
}
catch (IOException e) {...}
```



# Bedingungen zur Objektserialisierung

- ① Klasse muss Interface **Serializable** implementieren
  - ① Markerinterface (d.h. keine Methoden deklariert)
  - ① die meisten Klassen der Klassenbibliothek sind serialisierbar
  - ① mit Schlüsselwort **transient** können Attribute von Serialisierung ausgenommen werden
- ① **writeObject** speichert automatisch:
  - ① Attribute, die nicht-statisch und nicht-transient sind
  - ① referenzierte Objekte (rekursiv)
- ① alternativ ist (De-)Serialisierung selbst implementierbar:
  - ① **private void writeObject(ObjectOutputStream oos)**
  - ① **private void readObject(ObjectInputStream ois)**
- ① Lesen muss in gleicher Reihenfolge wie Schreiben erfolgen



# Streams - Übersicht

## I/O Streams

Type of I/O	Streams	Description
Memory	CharArrayReader CharArrayWriter ByteArrayInputStream ByteArrayOutputStream	Use these streams to read from and write to memory. You create these streams on an existing array and then use the read and write methods to read from or write to the array.
	StringReader StringWriter StringBufferInputStream	Use StringReader to read characters from a String in memory. Use StringWriter to write to a String. StringWriter collects the characters written to it in a StringBuffer, which can then be converted to a String. StringBufferInputStream is similar to StringReader, except that it reads bytes from a StringBuffer.
	PipedReader PipedWriter PipedInputStream PipedOutputStream	Implement the input and output components of a pipe. Pipes are used to channel the output from one thread into the input of another.
	FileReader FileWriter FileInputStream FileOutputStream	Collectively called file streams, these streams are used to read from or write to a file on the native file system.
Concatenation	N/A SequenceInputStream	Concatenates multiple input streams into one input stream.
Object Serialization	N/A ObjectInputStream ObjectOutputStream	Used to serialize objects.
	N/A DataInputStream DataOutputStream	Read or write primitive data types in a machine-independent format.



# Streams - Übersicht

## I/O Streams

Type of I/O	Streams	Description
Counting	LineNumberReader LineNumberInputStream	Keeps track of line numbers while reading.
Peeking Ahead	PushbackReader PushbackInputStream	These input streams each have a pushback buffer. When reading data from a stream, it is sometimes useful to peek at the next few bytes or characters in the stream to decide what to do next.
Printing	PrintWriter PrintStream	Contain convenient printing methods. These are the easiest streams to write to, so you will often see other writable streams wrapped in one of these.
Buffering	BufferedReader BufferedWriter BufferedInputStream BufferedOutputStream	Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source. Buffered streams are typically more efficient than similar nonbuffered streams and are often used with other streams.
Filtering	FilterReader FilterWriter FilterInputStream FilterOutputStream	These abstract classes define the interface for filter streams, which filter data as it's being read or written.
Converting between Bytes and Characters	InputStreamReader OutputStreamWriter	A reader and writer pair that forms the bridge between byte streams and character streams. An InputStreamReader reads bytes from an InputStream and converts them to characters, using the default character encoding or a character encoding specified by name. An OutputStreamWriter converts characters to bytes, using the default character encoding or a character encoding specified by name and then writes those bytes to an OutputStream. You can get the name of the default character encoding by calling System.getProperty("file.encoding").
Counting	LineNumberReader LineNumberInputStream	Keeps track of line numbers while reading.



# Quellenverzeichnis

---

- ② Java 8 – Grundlagen Programmierung, Herdt Verlag
- ③ Stream Überblick aus: <https://www.math.uni-hamburg.de/doc/java/tutorial>