Hochschule **Rosenheim**
University of Applied Sciences

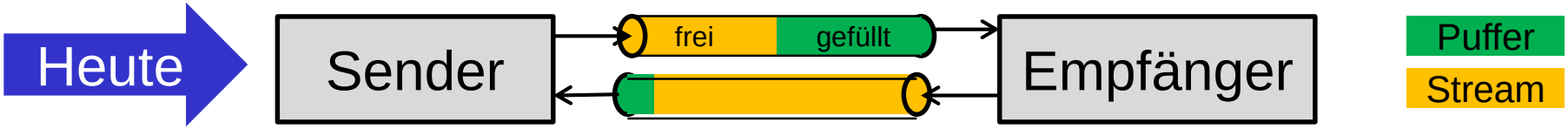
Verteilte Verarbeitung

Kapitel 2.2

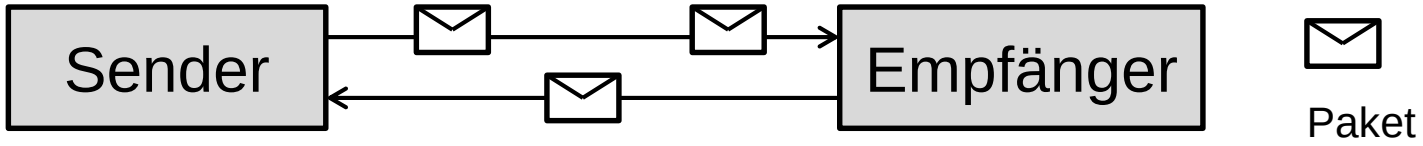
Streams

Kommunikation zwischen reaktiven Systemen

- Nachrichtenaustausch: Varianten
  - ***Stromorientiert*** (Streams / Pipes)

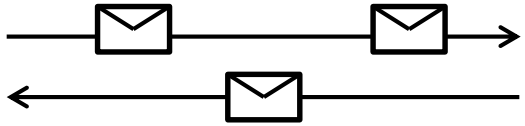


  - ***Paketorientiert*** (Datagramme / Nachrichten)



Stromorientiert vs. Nachrichtenorientiert

- Stromorientiert bzw. Verbindungsorientiert
  - ***Feste Verbindung*** zwischen einem Sender und einem Empfänger
  - Unidirektionale / Bidirektionale Ströme, gepuffert / ungepuffert
  - Kommunikation ist seriell (= Strom von Bytes),
    typisch auch für eingebettete Systeme (RS232, USB, …)
  - Zuverlässig, Reihenfolge bleibt erhalten
  - Wie in (alten) Telefonnetzen: GSM, Analoge Modems, …

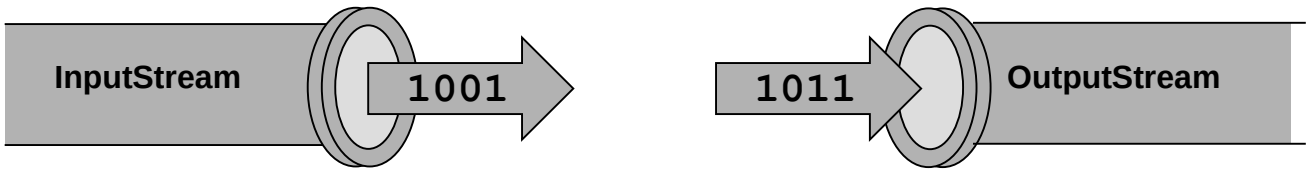


- Paketorientiert
  - ***Keine Verbindung***, Möglicherweise „Fire & Forget“
  - Paket wird übertragen bzw. über Netzwerk geroutet
  - Multicast / Broadcast möglich
  - ggf. unzuverlässig, ggf. geht Reihenfolge der Pakete verloren
  - Wie in (neuen) Telefonnetzen, LTE / (UMTS), TCP/IP, VoIP, …

Stromorientierte Kommunikation
Serielles Lesen und Schreiben von Daten

Idee in vielen Programmiersprachen:
Datenquellen und -ziele einheitlich behandeln

- Datenquelle/Datenziel = ***Strom von Bytes / Zeichen***
- ***Sequenzielles*** Lesen und Schreiben in diese Ströme
- Stream abstrahiert Datei, Hauptspeicher, Konsole, Socket, …
- Je nach Datenquelle/ziel andere ***Stream*** bzw. ***Writer/Reader*** Klasse in Java (Java: im JDK6 > 60 Stream-Klassen!)
- Schön: Filter / Transformation einbaubar



Datenquellen und -ziele können sein

- Konsole + Tastatur + (Maus)
- Dateien (Files, zip)
- Hauptspeicher, Byte Arrays
- Pipes (zur Kommunikation zwischen Threads)
- ***Sockets (zur Kommunikation über ein Netzwerk)***

- Spezielle Streams für Multimedia / XML / …

Streams in Java: java.io

Java und Streams

Java unterscheidet Streams

- zum Lesen und zum Schreiben
  (**`InputStream`** bzw. **`OutputStream`**)
- für ***Binär-*** bzw. ***Textdaten (UTF 16)***
  (**`InputStream`** / **`OutputStream`**, bzw. **`Reader`** / **`Writer`**)
- für bestimmte ***Datenquellen***
  (z.B. **`FileReader`**, **`CharArrayReader`**)
- mit spezieller ***Funktionalität***,
  etwa gepuffertes oder gefiltertes Lesen
  (**`BufferedReader`**, **`FilterReader`**)

Streams sind auch: **`System.in`**, **`System.out`** und **`System.err`**

Umgang mit Strömen

- Lesen und Schreiben erfolgt häufig byteweise
  (Achtung, das beißt sich mit der Hardware, die arbeitet in Blöcken)

```
InputStream is = null;
is = new FileInputStream(„XY.dat");
// is = System.in;          // Konsole
// is = new Socket(…);      // Netzwerk

int c = 0;
while ((c = is.read()) != -1) {  // -1 bedeutet: EOF
   // Irgendwas mit c machen
   // Sonderzeichen, etwa ‚\n' oder ‚\r' selber
   // behandeln
   // Byteweise – arbeiten: schlechte Performance
}
```

- Während des Lesens:
  - z.B. Umwandlung in **`char`**
  - Behandlung von Sonderzeichen (EndOfLine, LineFeed)
  - Pufferung

Umgang mit Strömen /2

Freigabe von Ressourcen bei Reaktiven Systemen

- Alle Ströme werden automatisch geöffnet, wenn die Instanzen kreiert werden
- Ströme müssen explizit (finally-Block) mit „close“ geschlossen werden (da sie sonst unnötig Ressourcen belegen)
- Ressource = Netzwerkverbindung, Dateihandle, DB-Verbindung, Speicher

```
InputStream is = null;
try {
    is = new FileInputStream(„XY.dat“);
    int c = 0;
    while ((c = is.read()) != -1) {
        …
    }
} finally {
    try {is.close();} catch(Exception ex) { … }
}
```

Umgang mit Strömen /2

Freigabe von Ressourcen bei Reaktiven Systemen

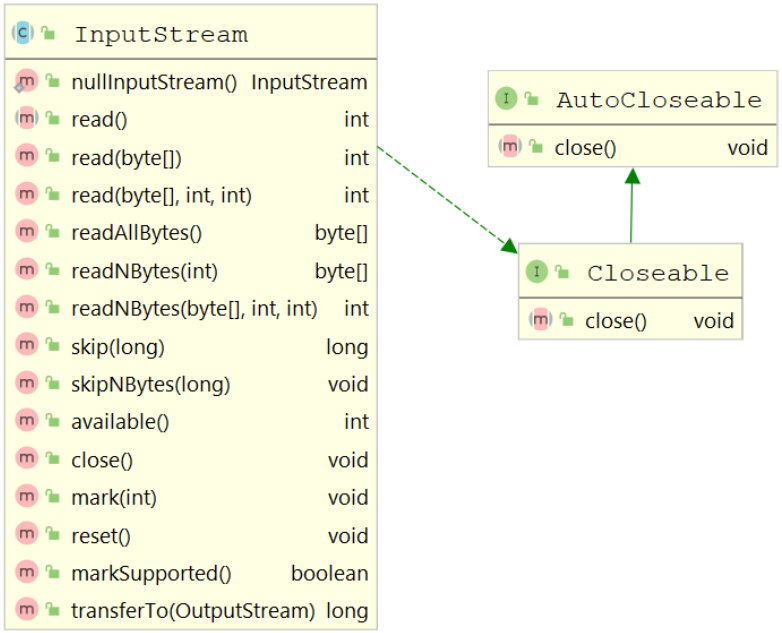
- Seit Java 7 etwas besser zu programmieren …
- Neue try-catch Klammer mit automatischer Ressourcen Freigabe
- Ressource muss allerdings das Closable Interface implementieren

```
try (InputStream is = new FileInputStream("XY.dat")) {
    int c = 0;
    while ((c = is.read()) != -1) {
       // Zeichenweise
    }
}
catch(IOException ex) {
    System.err.println("Fehler:" ex.getMessage());
}
```

Umgang mit Ressourcen bei Reaktiven Systemen

- Im Laufe des Betriebs allokiert ein Reaktives System Ressourcen (z.B. in Form von Streams)
  - Hauptspeicher
  - Netzwerkverbindungen
  - Datei-Handles
- Da das Reaktive System nicht terminiert:
  - Wichtig: Akribisch darauf achten, dass allokierte Ressourcen ***in jedem Fall wieder freigegeben*** werden (***auch im Fehlerfall***)
  - Freigabe in der Regel im ***`finally`***-Block, bzw. try(…) {} catch () {}
  - Sonst sind die Ressourcen irgendwann verbraucht und das System bleibt stehen oder wird sehr langsam (z.B. wg. Swapping, da z.B. Speicherlöcher entstanden sind)

Byteweise Lesen: *InputStream*



API

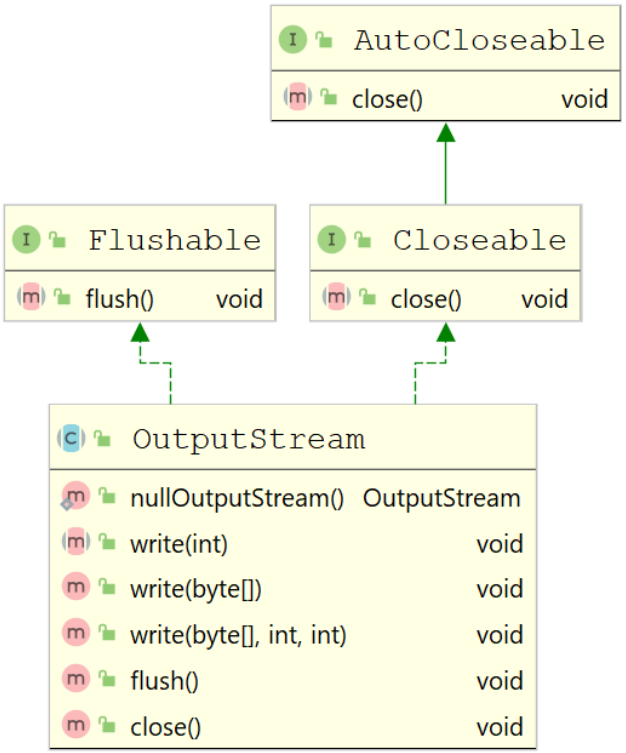
Byteweise

```
InputStream in = …;
int c = 0;
while ((c = in.read()) != -1) { // -1 = EOF
   System.out.println("Reading Byte: " + c);
}
in.close();
```

Weitere Beispiele für InputStreams

- **`FileInputStream`** Lesen aus Dateien
- **`ByteArrayInputStream`** Lesen aus ByteArray
- **`ObjectInputStream`** Deserialisieren von Objekten
- **`DataInputStream`** Komfortablere Funktionen
- **`SequenceInputStream`** Verkettung von InputStreams

Byteweise schreiben: *OutputStream*



API

- **`flush()`** erzwingt Leeren der Schreibpuffer
- Stream nach der Verwendung mit **`close()`** freigeben
- Bei **`close`** erfolgt **`flush`** automatisch

Weitere Beispiele für OutputStreams

- **`FileOutputStream`** Schreiben in Dateien
- **`ByteArrayOutputStream`** Schreiben in ByteArray
- **`ObjectOutputStream`** Serialisieren von Objekten
- **`DataOutputStream`** Komfortablere Funktionen
- **`PrintStream`** Komfortablere Funktionen

Beispiel für die Schachtelung von Streams
I/O Performance – Puffer verwenden!

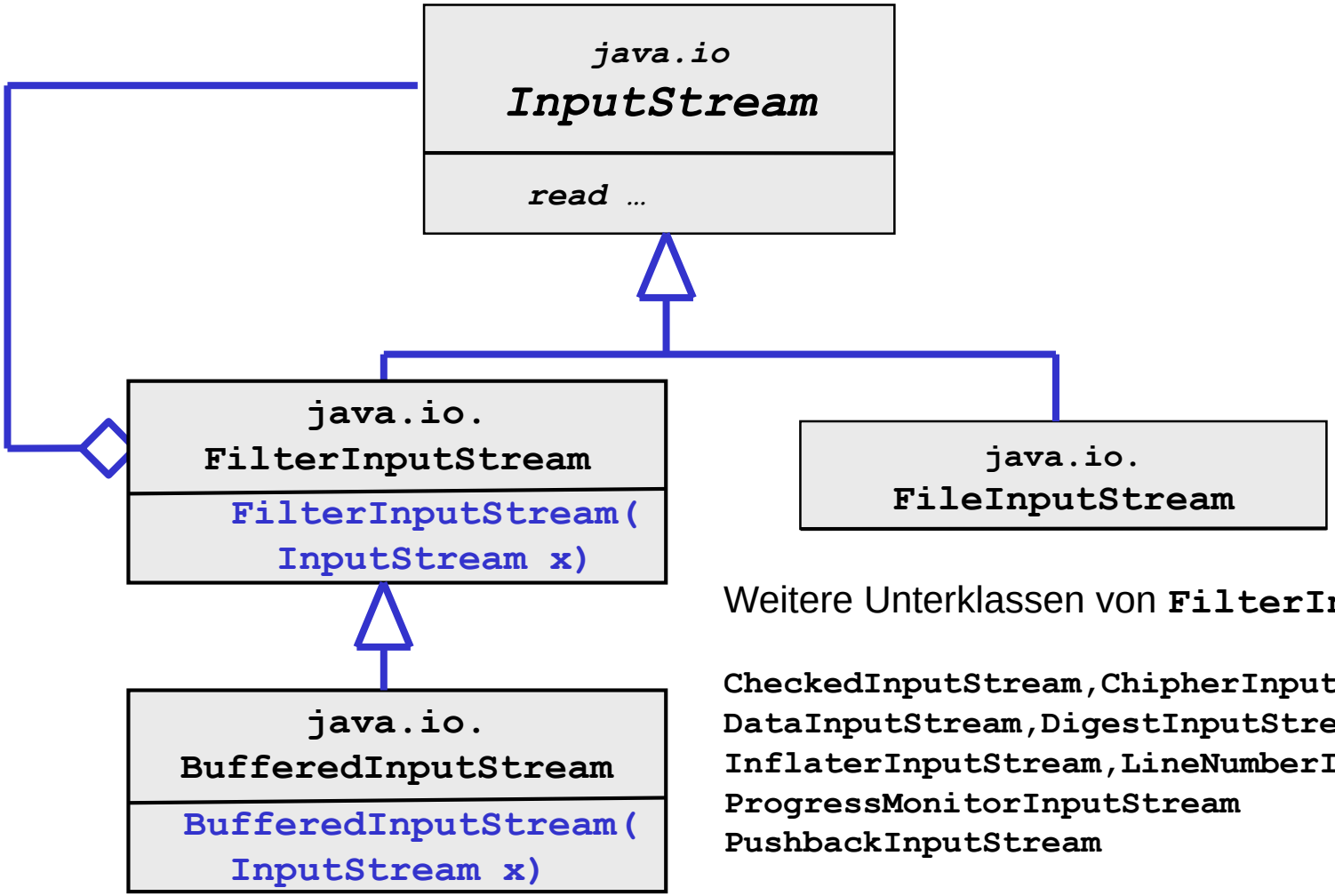
Benutzen Sie ***Puffer***, wo es möglich ist.

```
// …
try {
   in = new FileReader(inFile);
   out = new FileWriter(outFile);
   BufferedInputStream inBuffered = new BufferedInputStream(in);
   BufferedOutputStream outBuffered =
             new BufferedOutputStream(out);

   int c = 0;
   while ((c = inBuffered.read()) != -1) {
      outBuffered.write(c);
   }
} finally {
// …
```

Filter-Streams und das Decorator Pattern

Umgang mit Strömen: Veredelung
(das Decorator Pattern)



Weitere Unterklassen von **`FilterInputStream`**:

```
CheckedInputStream,ChipherInputStream,
DataInputStream,DigestInputStream,
InflaterInputStream,LineNumberInputStream,
ProgressMonitorInputStream
PushbackInputStream
```

Decorator Eigenschaften

- Erweitern von Funktionalität ohne eine direkte Subklasse zu bilden
  - Flexibles Modell zur Erweiterung
  - Mitten in einer Vererbungshierarchie einsetzbar
- Decorator (hier: ***`FilterInputStream`***)
  - Hat genau eine dekoriertes Objekt (hier vom Typ ***`InputStream`***)
  - ***Implementiert dasselbe Interface***
    - ergänzt / erweitert Funktionalität, und/oder
    - reicht Funktionen an dekoriertes Objekt weiter
  - Verwendet nur „öffentliche“ Methoden des dekorierten Objekts, kein Zugriff auf private und (teilw.) protected

Beispiel für die Schachtelung von Streams
nach Decorator Pattern

- Zeilenweise Schreiben in eine Datei

```
File outFile = new File(fileName);
OutputStream out = new FileOutputStream(outFile);
BufferedOutputStream outBuffered =
                              new BufferedOutputStream(out);
PrintStream outConvenient = new PrintStream(outBuffered);

outConvenient.println(„Ja hallo erstmal“);
```

- Zeilenweise Lesen

```
File inFile = new File(fileName);
InputStream  in = new FileInputStream(inFile);
InputStreamReader inReader = new InputStreamReader(in);
BufferedReader bufferedReader = new BufferedReader(inReader);

String line = "";
while ((line = bufferedReader.readLine()) != null){
```

Ein eigener **`Filter`**

```
public class NummerFilter extends FilterOutputStream {
  public NummerFilter(OutputStream out) { super(out); }

  public void write(int c) throws IOException {
    switch (c) {
        case '1': writeString("eins ");break;
        case '2': writeString("zwei ");break;
        // …
        default: super.write(c);
  } }

  private void writeString(String str) throws IOException {
    for (int i=0; i < str.length(); i++) {
        super.write(str.charAt(i)); // Delegation an super
  } }

  public static void main(String[] args) throws IOException {
    OutputStream ausg = new NummerFilter(System.out);
    PrintStream ps = new PrintStream(ausg);
    ps.println("Meine Telefonnummer ist: 089/1234567-890");
    ausg.flush(); // Puffer leeren
} }
```

Nützliche Hilfsklassen

Angenehm: `PrintStream`

```
public class PrintStream extends FilterOutputStream
   implements Appendable, Closeable {

      boolean checkError()
      void print(…)
      void println(…)
      …
```

- Zeilenweise Schreiben beliebiger Daten über print/println
- `System.out` ist ein `PrintStream`
- Übersetzung der Daten in „Byte-Würste“,
  Bei Strings: Default Character Encoding der Plattform
- Automatisches `flush()` bei new line
- IO-Exception wird niemals geworfen, Fehler über `checkError()`
- Stand heute: Eher `PrintWriter` verwenden (Zeichensätze)

Formatierte Ausgabe

- Besonderheit (Gruß aus C): **`format()`**–Methode der Klasse String, **`PrintStream.printf(), PrintStream.format()`**

```
String s = String.format(
   "Vorname: %s - Nachname %7s.",
   "Hubert", "Kah" );
System.out.println( s );
```

- Ausgabeformatierung: Beispiel ISO-Datum (Problem mit den führenden 0-en)

```
int jahr = 2017; int monat = 2; int tag = 7;
System.out.printf("%04d-%02d-%02d",jahr,monat,tag);
```

- Ausgabeformatierung mit Fließpunkt-Zahlen

```
double PI = 3.1415973d;
String zweiStellen = String.format("%.2f", PI);
System.out.println("PI: " + zweiStellen);
```

Strings/Ausgaben formatieren
Eine kleine Auswahl der Möglichkeiten

- Symbole = Platzhalter für Variablenwerte und Formatierungsanweisungen
- Beispiele (Conversion)

`%s`	Zeichenkette (wie sie ist, String)
`%d`	Dezimalzahl (int, long)
`%f`	Fließpunktzahl (float, double)
`%t`	Datum / Zeit (java.util.Date)
`%n`	Zeilenvorschub
`%%`	Prozentzeichen

- Formatierung:

`%10s`	Zeichenkette Länge 10, vorne Leerzeichen
`%-10s`	Zeichenkette Länge 10, hinten Leerzeichen
`%04d`	Dezimalzahl, 4 Stellen, führende 0
`%3.7f`	Fließpunktzahl mit 3 Vor- und 7 Nachkommastellen

Formatierte Eingabe: Scanner

- java.util.Scanner zerlegt Datenstrom in Tokens
- Token können ausgelesen werden mit
  - String nextLine(), String nextString()
  - int nextInt(), long nextLong(), etc.
- Abfrage, ob Token vorhanden mit hasNextLine() (analog für andere Datentypen)
- Suchen mit regulären Ausdrücken möglich mit next(String regexp)

Beispiel:

```
Scanner s = new Scanner(System.in);
while (true) {
   String zeile = s.nextLine();
   System.out.println("Zeile:" + zeile);
}
```

Java.nio: Channel, Puffer, Selektoren

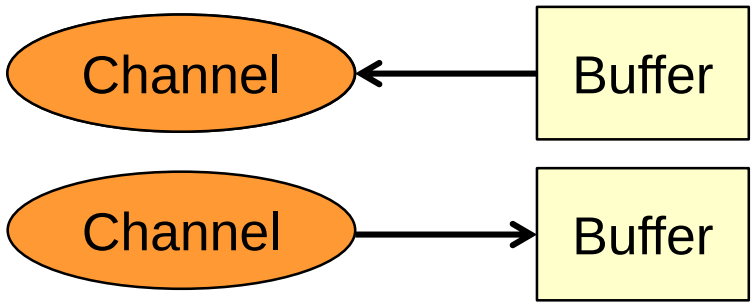
Nur zur Information wird nicht besprochen

Java NIO und Java NIO 2

- = New I/O, eingeführt in Java 4, deutlich erweitert in Java 7
- Java 4 NIO – JSR 53:
  - Gepuffertes Lesen und Schreiben explizite Puffer
- Java 7 NIO.2 – JSR 203:
  - Besserer Umgang mit Dateien
  - Besserer Umgang mit Pfaden in Verzeichnisstruktur

Konzept der Channel und der Buffer

- Channel (java.nio.channels.Channel)
  - Ähnlich zu Stream, aber Duplex, Streams häufig Simplex
  - Repräsentiert offene Verbindung zu Datei, Hardware, Socket etc.
  - Liest/Schreibt in einen oder mehrere Buffer (java.nio.Buffer)
  - Kann asynchron verwendet werden
- Implementierungen
  - FileChannel, ByteChannel
  - DatagramChannel
  - SocketChannel, ServerSocketChannel



Beispiel: Lesen einer Datei

```
RandomAccessFile datei = new RandomAccessFile("beispiel.txt", "rw");
FileChannel inChannel = datei.getChannel();

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {
   buf.flip();
   while (buf.hasRemaining()) {
      System.out.print((char) buf.get());
   }

   buf.clear();
   bytesRead = inChannel.read(buf);
}
datei.close();
```

Puffer im Hauptspeicher

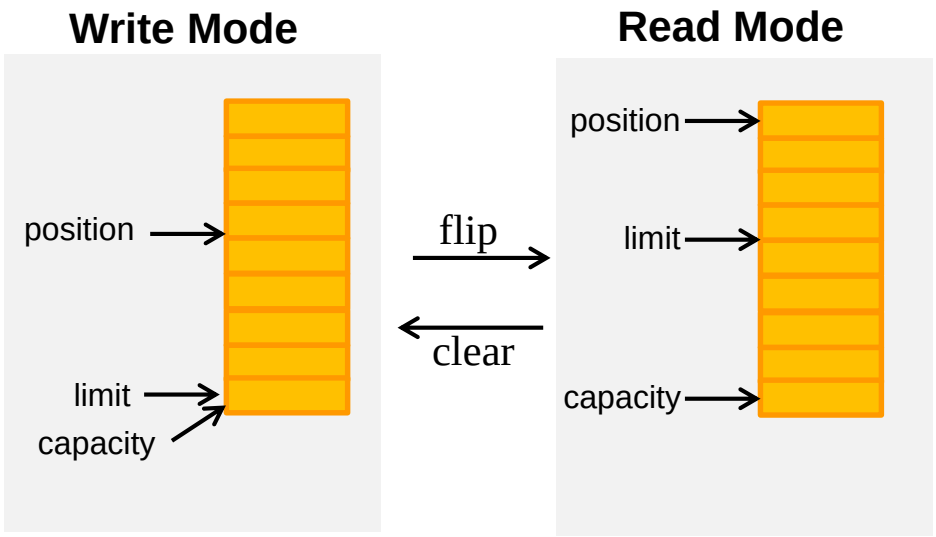
Einlesen aus Datei in den Puffer

Verwenden eines Puffers:
1. Daten in Puffer holen
2. buf.flip() aufrufen
3. Daten aus Puffer auslesen
4. buf.clear() aufrufen

filp() sorgt dafür, dass genau der in den Buffer eingelesene Teil wieder ausgelesen werden kann.
(Kapazität auf letzte Position, Leseposition wieder auf 0)

Buffer

- = Speicherbereich (byte-Array)
  - In den ein Channel schreiben kann
  - Aus dem ein Channel lesen kann
  - Größe wird mit allocate(), allocateDirect() festgelegt
- Drei Informationen: position, limit, capacity



- Typen von Buffern
  - ByteBuffer
  - IntBuffer
  - LongBuffer
  - …

Direkter Transfer von Daten zwischen Channels

- Direkter Datentransfer zwischen zwei Dateien mit transferFrom-Methode

```
RandomAccessFile quelleFile = new RandomAccessFile(quelle, "rw");
FileChannel quelleChannel = quelleFile.getChannel();

RandomAccessFile zielFile = new RandomAccessFile(ziel, "rw");
FileChannel zielChannel = zielFile.getChannel();

zielChannel.transferFrom(quelleChannel, 0, quelleChannel.size());
```

Selektoren

- Nützlich bei ***nicht-blockierendem IO***
  - Mehr dazu im Foliensatz über Sockets
- Channel können dort registriert werden
- Selector reagiert darauf wenn Channel
  - Daten zum Lesen hat
  - Bereit zum Schreiben ist
  - In weiteren Fällen: vgl. Socket Foliensatz