

# Programmieren II: Java

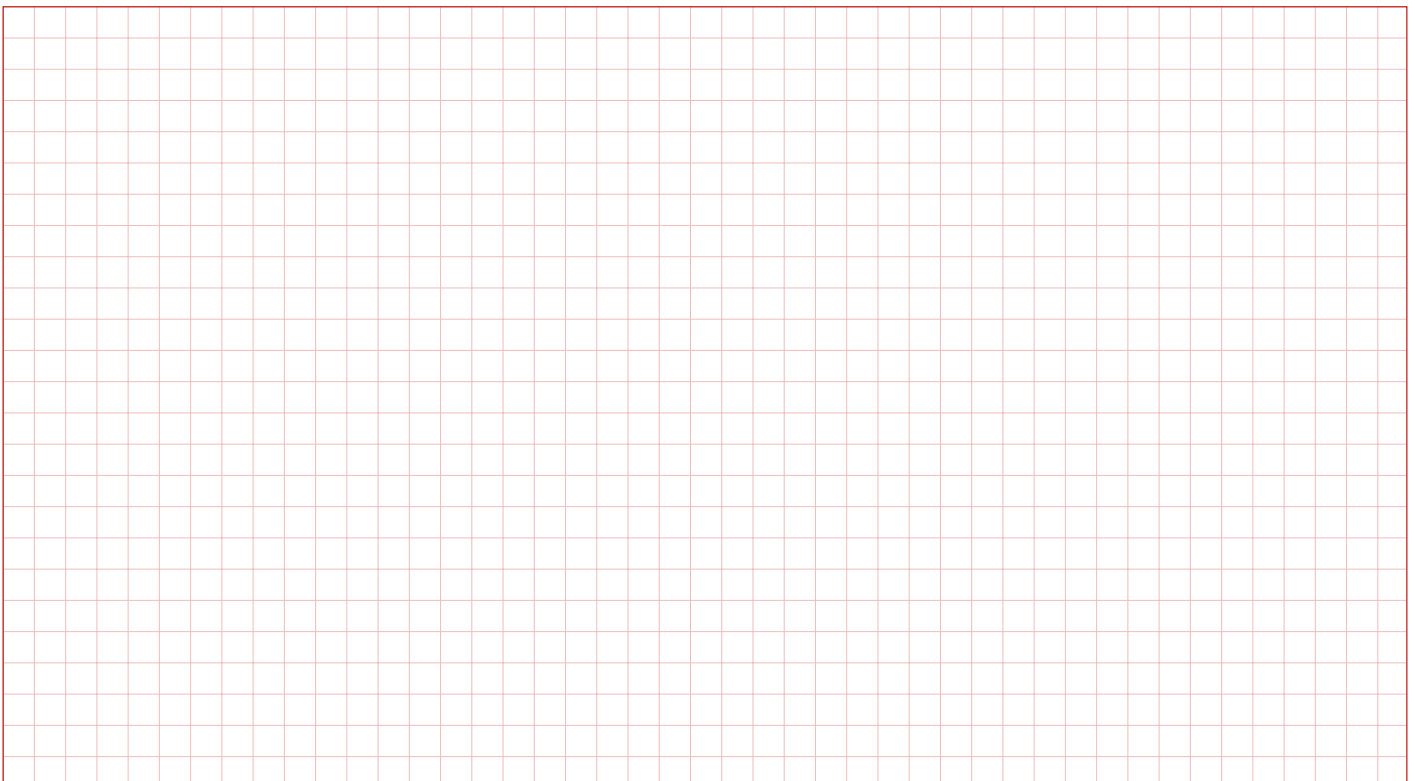
Ein- und Ausgabe

Prof. Dr. Christopher Auer

Sommersemester 2024



## Notizen



Motivation

Byteströme

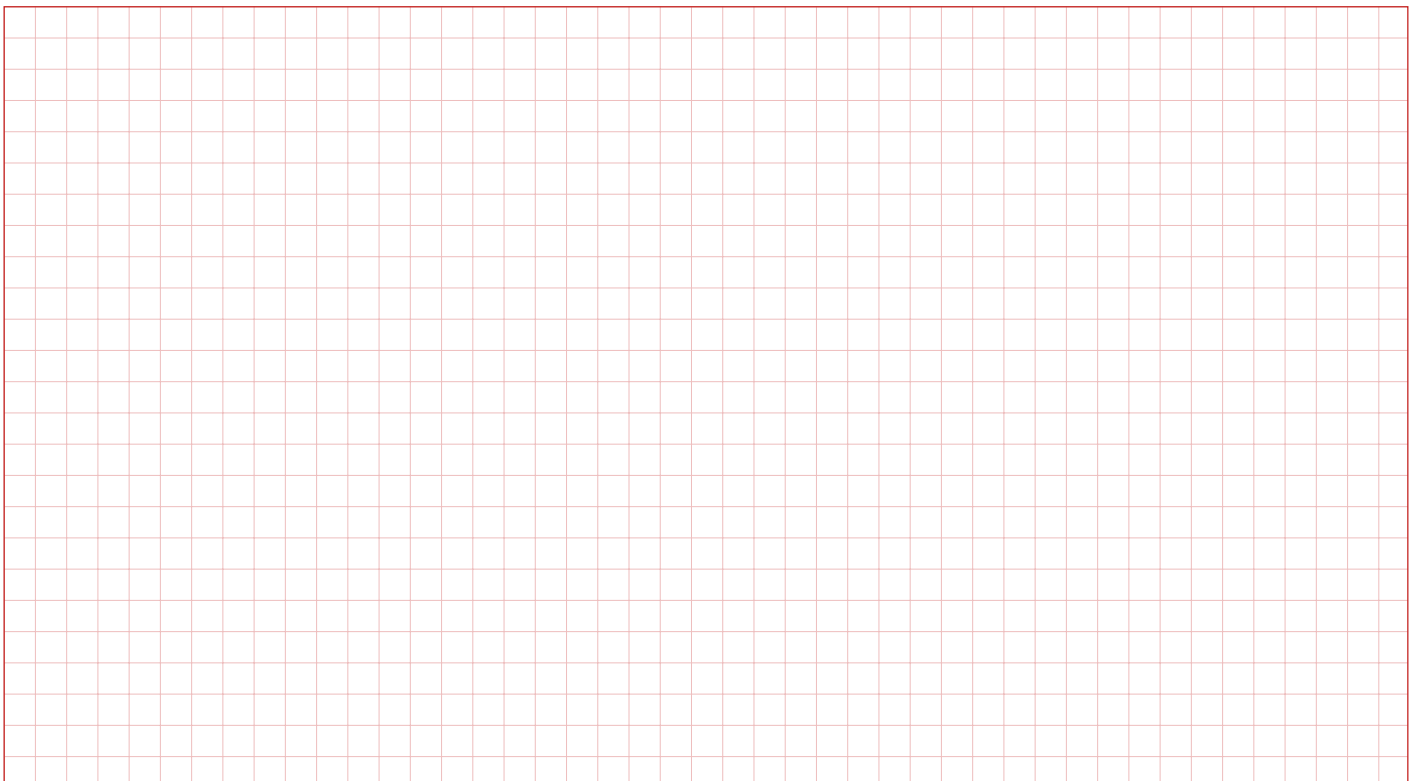
Text Ein- und Ausgabe

Automatic Resource Management

Dateien und Verzeichnisse

Zusammenfassung

## Notizen



# Inhalt

# Motivation

# Motivation

## Notizen

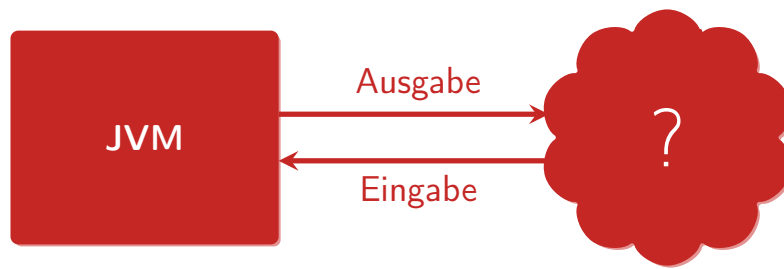
# Inhalt

# Motivation

# Motivation

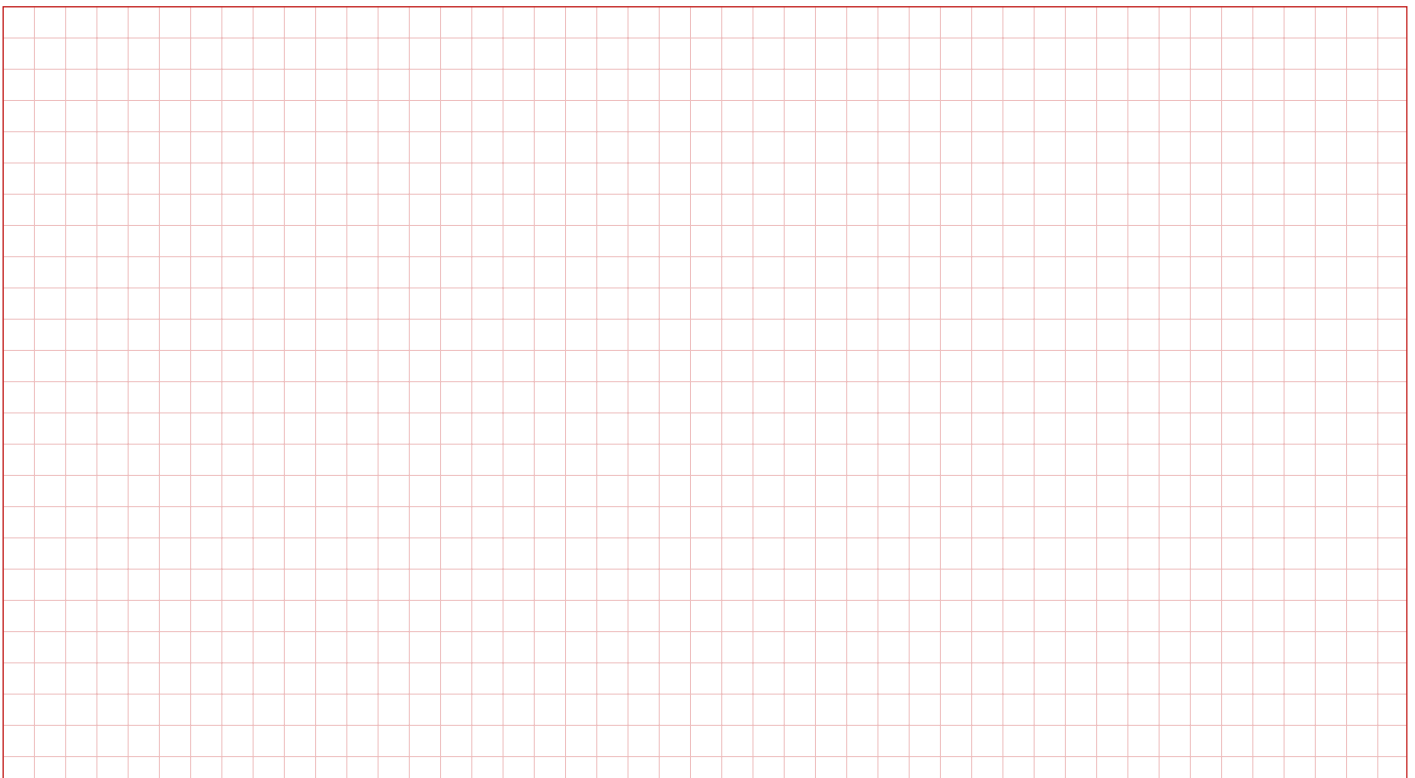
# Notizen

## Motivation



- ▶ JVM bildet **Abstraktionsschicht** zu **Betriebssystem**
  - ▶ Ein- und Ausgabeströme
  - ▶ **Dateisystem**: C:\Users\auer vs. /home/auer vs. /Users/auer
- ▶ Ein- und Ausgabe von/zu
  - ▶ **Standard Ein- und Ausgabe** (System.in, System.out)
  - ▶ **Dateien**: binär, Text, Devices, named PIPEs, etc.
  - ▶ **Netzwerk (IP/Bluetooth/etc.)**: Sockets, WebSockets, etc.
  - ▶ Andere **Prozesse**: PIPEs
  - ▶ ...

## Notizen



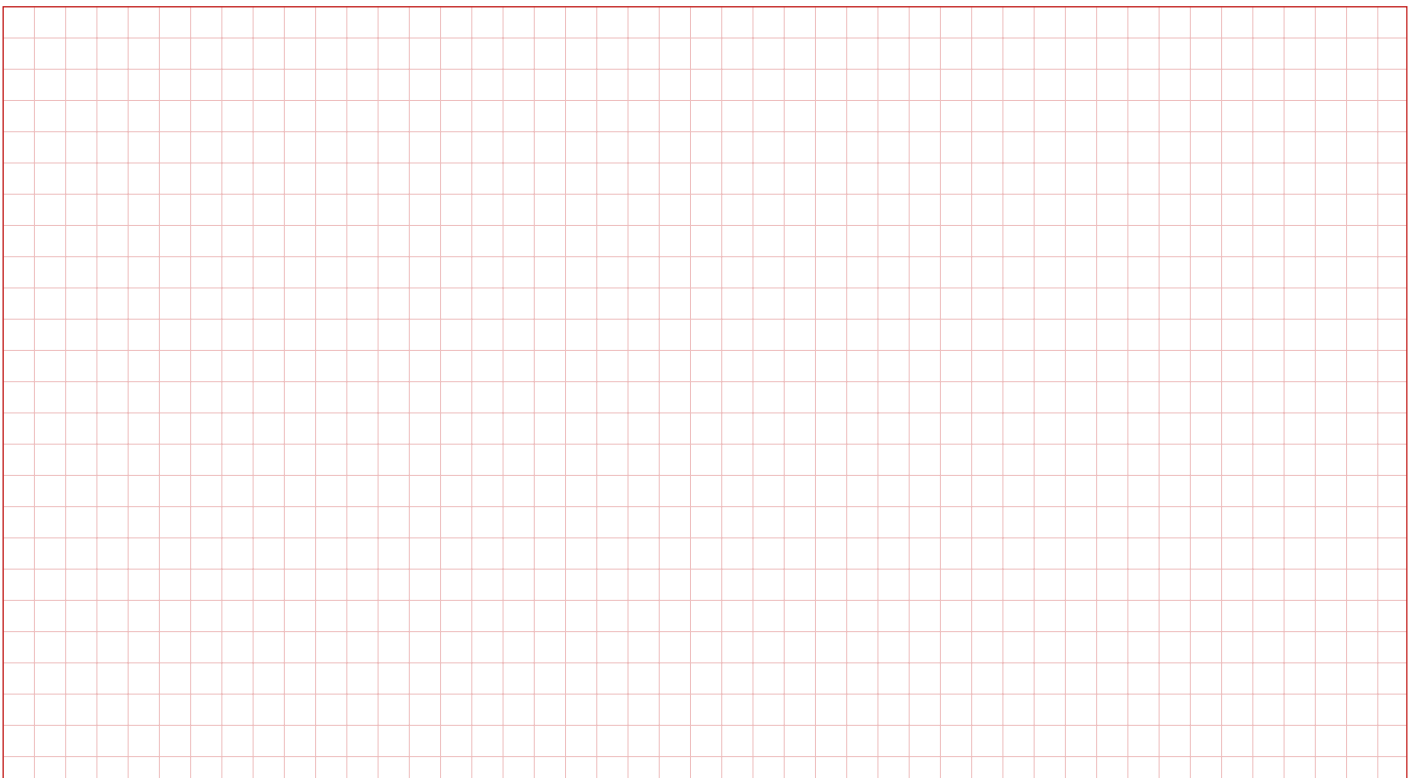
## Beispiel

- ▶ Eingabestrom ↗ `InputStream`
  - ▶ `int` `InputStream.read()` **liest** nächstes Byte (0—255)
  - ▶ -1 wenn Strom „zu Ende“
- ▶ Ausgabestrom ↗ `OutputStream`
  - ▶ `void` `OutputStream.write(int b)` **schreibt** nächstes Byte (0—255)
- ▶ `ioPlusOne(InputStream in, OutputStream out)`
  - ▶ **Liest** Byte für Byte aus in
  - ▶ **Addiert 1** (% 256)
  - ▶ **Schreibt** Byte in out

```
21 public static void ioPlusOne(InputStream in,  
22     OutputStream out) throws IOException {  
23     for (int i = in.read(); i >= 0; i = in.read()){  
24         out.write((i+1)%256);  
25     }  
26 }
```


ByteStreamExamples.java

## Notizen



## Beispiel — Standard Ein- und Ausgabe

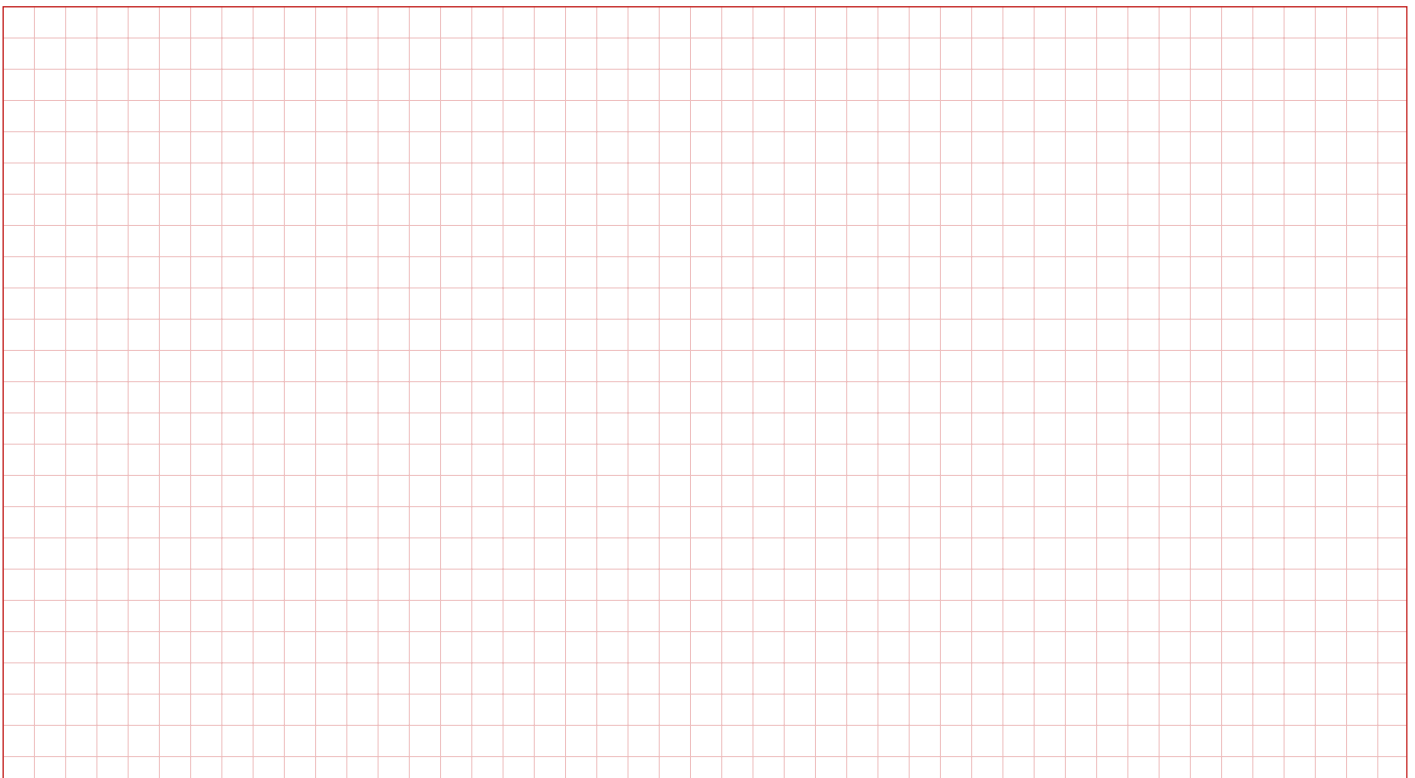
### ► Aufruf mit Standard Ein- und Ausgabe (Terminal)

```
33  runIoPlusOneStdInOut  
34 ioPlusOne(System.in, System.out);
```

 ByteStreamExamples.java

```
The cake is a LIE!<Ctrl-D/Ctrl-Z>  
Uif!dblf!jt!b!MJF"
```

## Notizen



## Beispiel — Datenströme aus und in Dateien

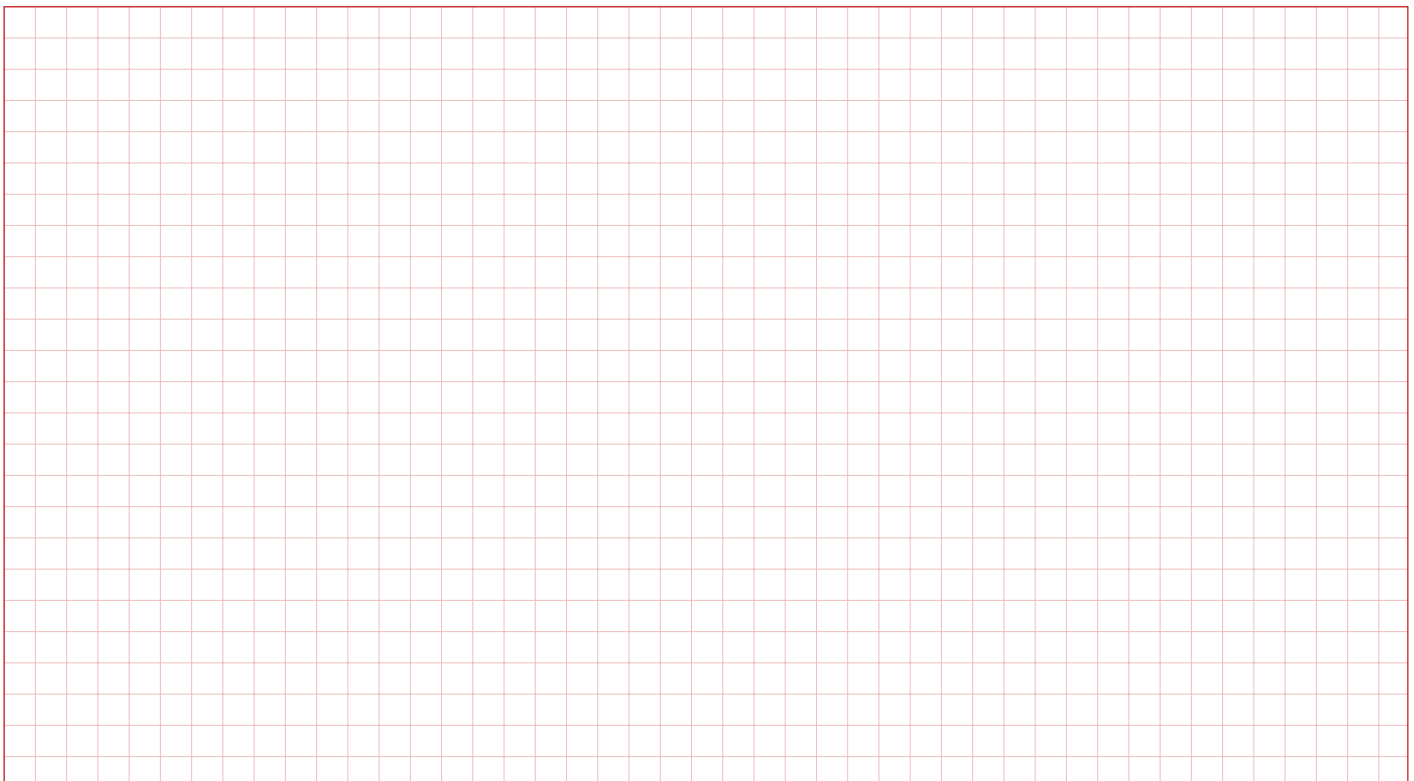
### ► Aufruf mit Datenströmen aus und in Dateien

```
41  runIoPlusOneFiles  
42 FileInputStream in = new FileInputStream("input.txt");  
43 FileOutputStream out = new FileOutputStream("output.txt");  
44 ioPlusOne(in, out);
```

 ByteStreamExamples.java

```
% echo "The cake is a LIE" > input.txt  
% gradle runIoPlusOneFiles  
% cat output.txt  
Uif!dblf!jt!b!MJF
```

## Notizen





## Beispiel — Datenströme aus dem Netzwerk


### ► Aufruf mit Datenströmen aus dem Netzwerk

#### ► „Server“ (extern in Terminal)

```
% echo "The cake is a LIE" | netcat -lp 12345
```

- „Hört“ auf Port 12345
- **Schreibt** „The cake is a LIE“ bei Verbindung auf Socket
- **Liest** Eingabe und gibt sie aus

#### ► ioPlusOneNetwork

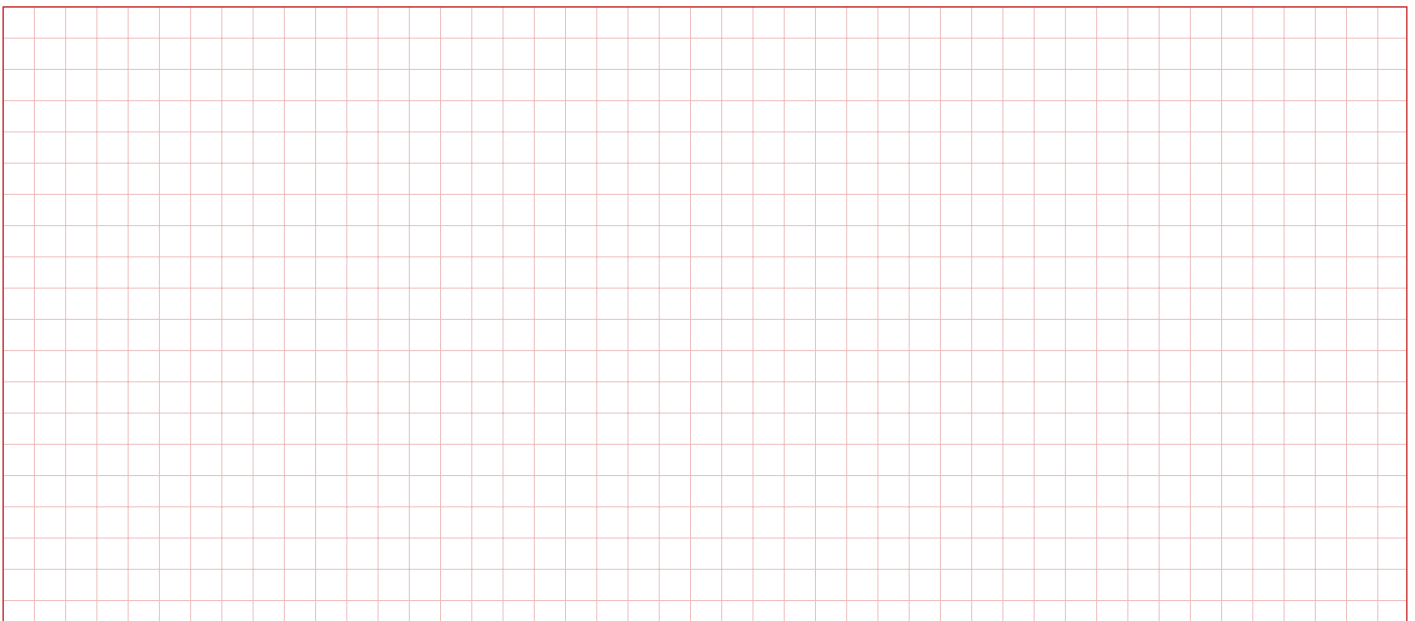
```
50  runIoPlusOneNetwork  
51 Socket socket = new Socket();  
52 socket.connect(  
53     new InetSocketAddress("localhost", 12345));  
54 ioPlusOne(socket.getInputStream(),  
55     socket.getOutputStream());
```

 ByteStreamExamples.java

- **Verbindung** mit Port 12345
- **Socket-Ströme** werden an ioPlusOne übergeben

## Notizen

- netcat ist ein Terminal-Programm für Unix mit dem man IP-Netzwerkverbindungen aufbauen kann.



## Beispiel — Datenströme aus dem Netzwerk

### ► Server

```
% echo "The cake is a LIE" | netcat -lp 12345
```

### ► Client

```
% gradle runIOPlusOneNetwork
```

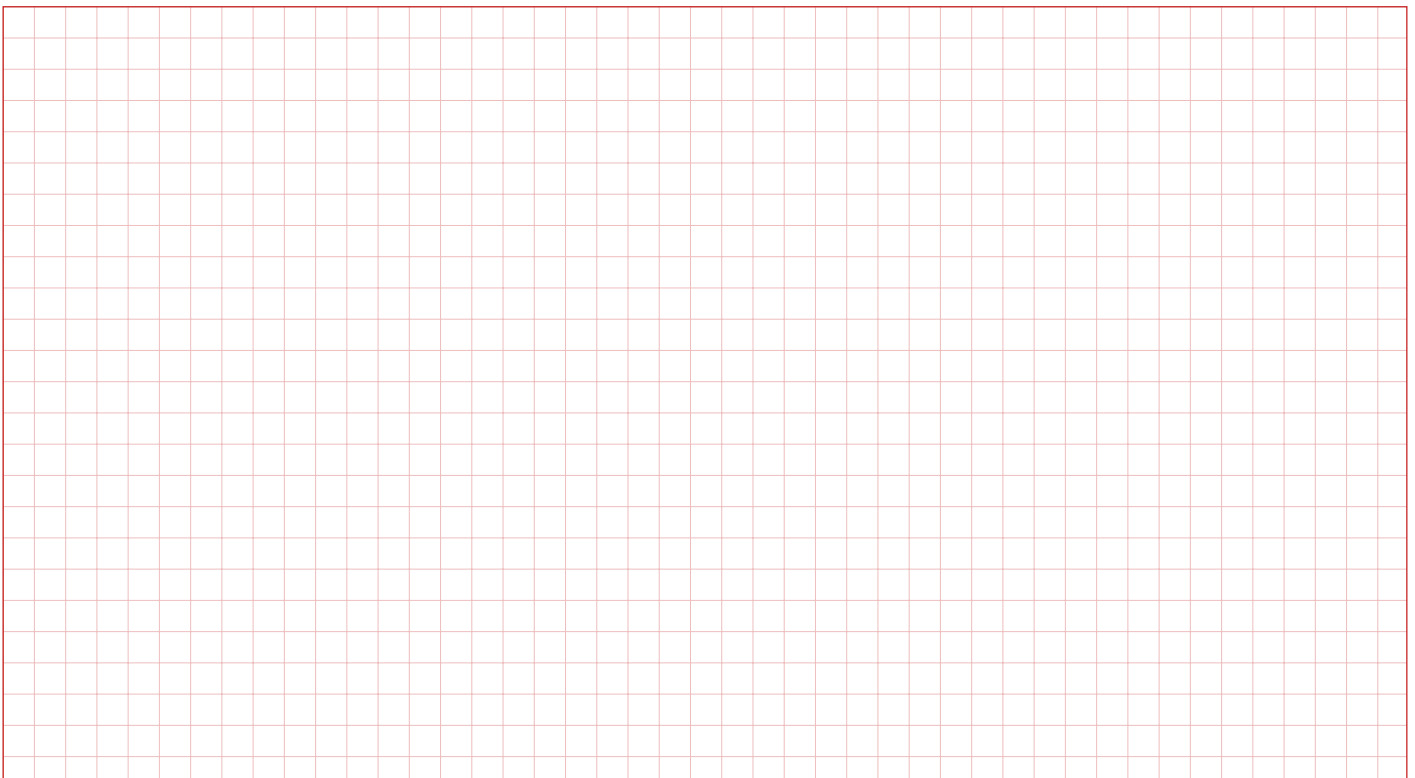
### ► Server

```
Uif!dblf!jt!b!MJF
```

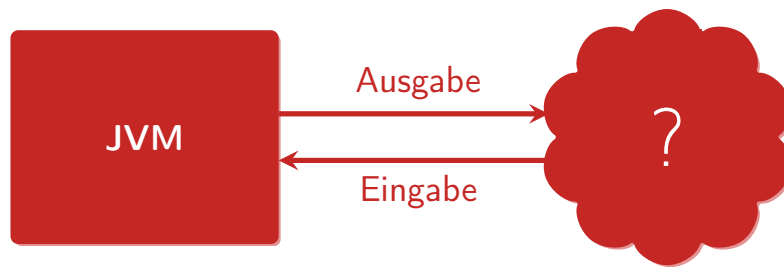
### ► Was passiert hier?



## Notizen

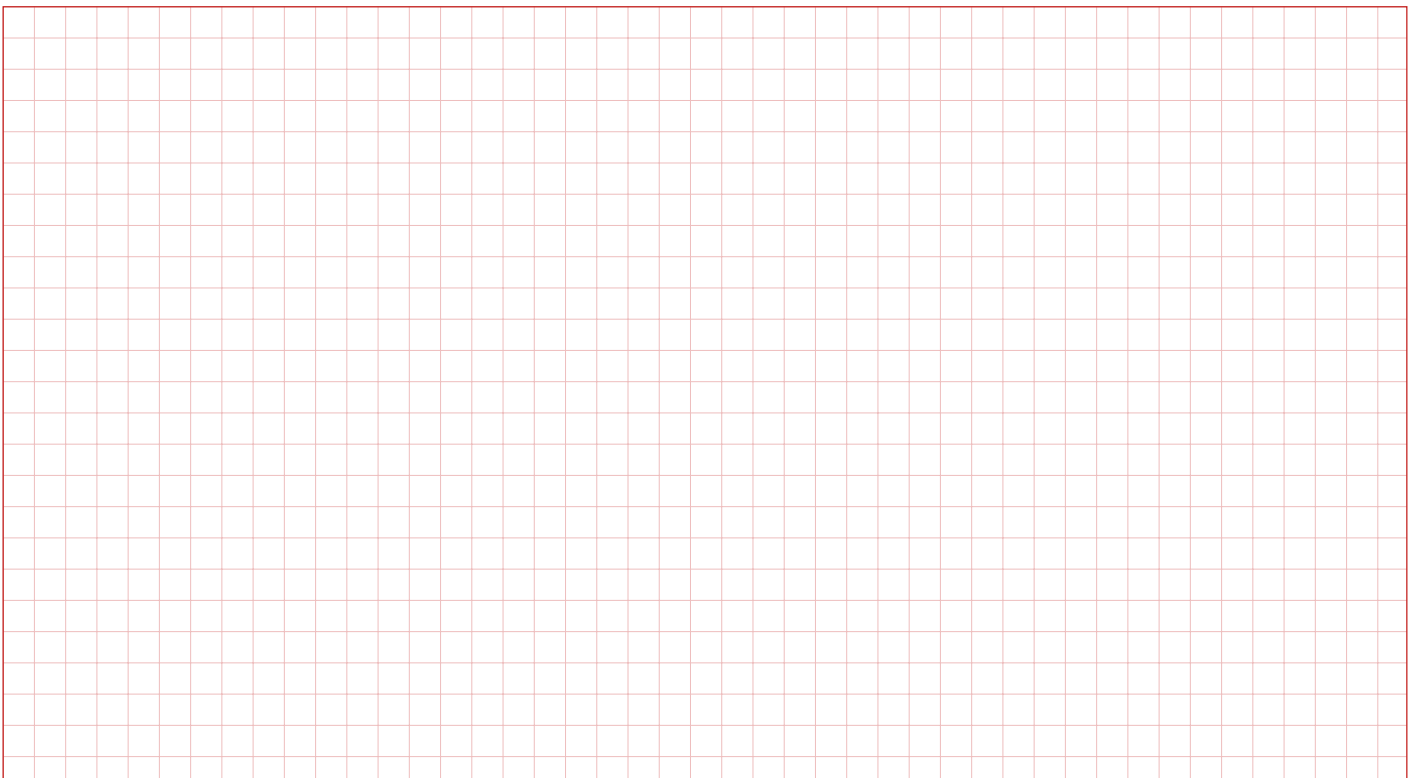


## Kleine Zusammenfassung



- ▶ Input/OutputStream heißen **Byteströme**
  - ▶ Ein- und Ausgabe
  - ▶ „Egal“ was **dahinterliegt** (Abstraktion)
- ▶ Als nächstes: **Byteströme** im **Detail**
  - ▶ Welche Byteströme gibt es?
  - ▶ Wie **arbeitet** man mit Byteströmen?

## Notizen

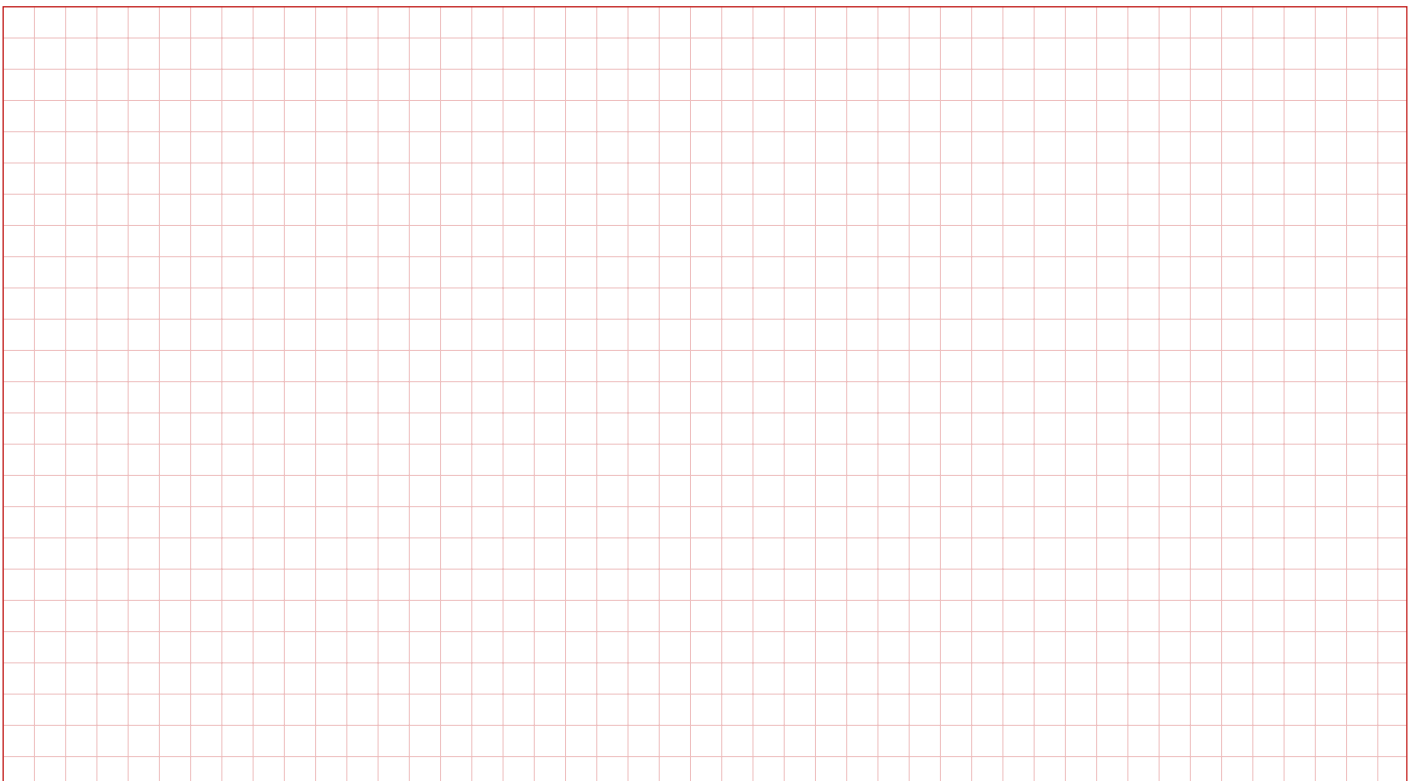


# Inhalt

## Byteströme

- Byteströme: Lesen und Schreiben
- Quellen für Eingabeströme
- Senken für Ausgabeströme
- Übersicht
- Filter
- Beispiel
- Zusammenfassung

## Notizen



# Inhalt

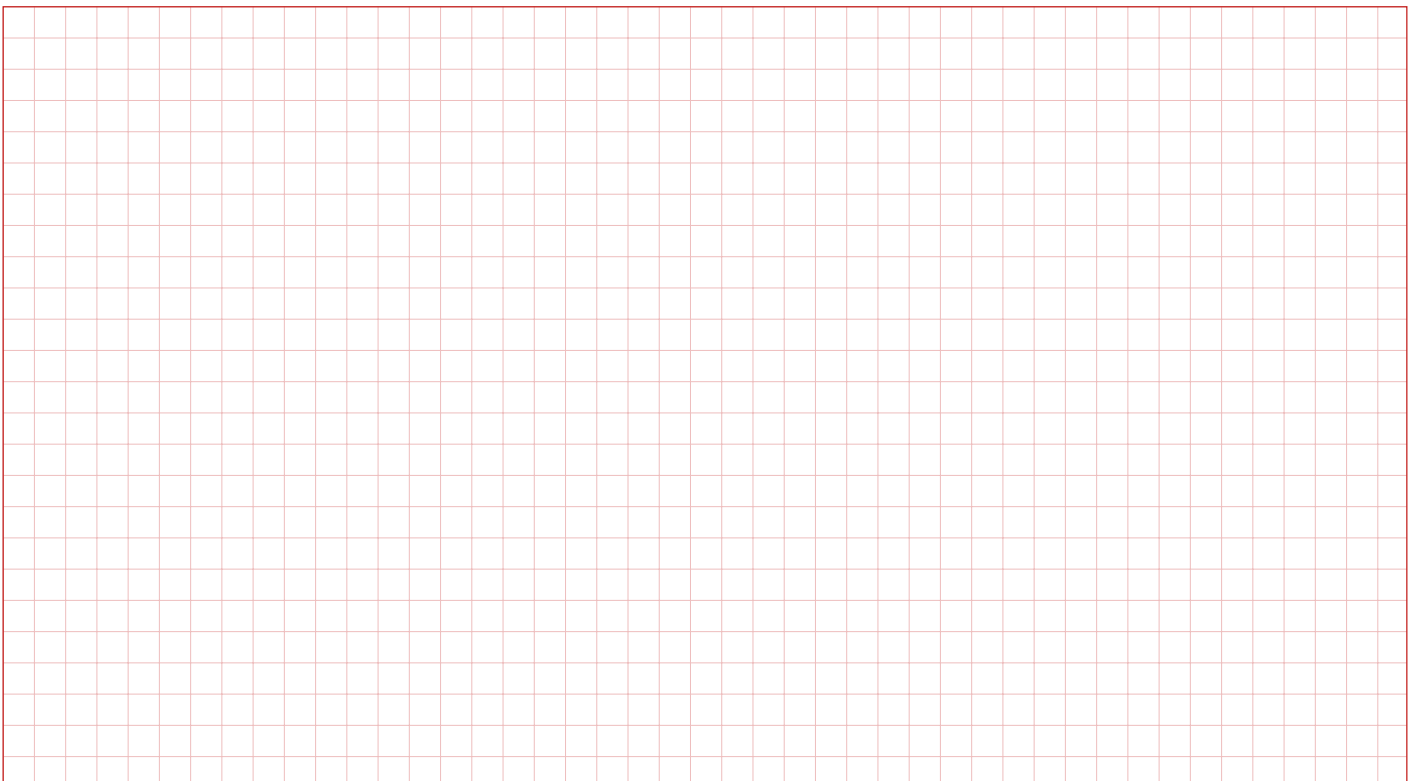
## Byteströme

Byteströme: Lesen und Schreiben

Eingabeströme: InputStream

Ausgabeströme: OutputStream

## Notizen



# Inhalt

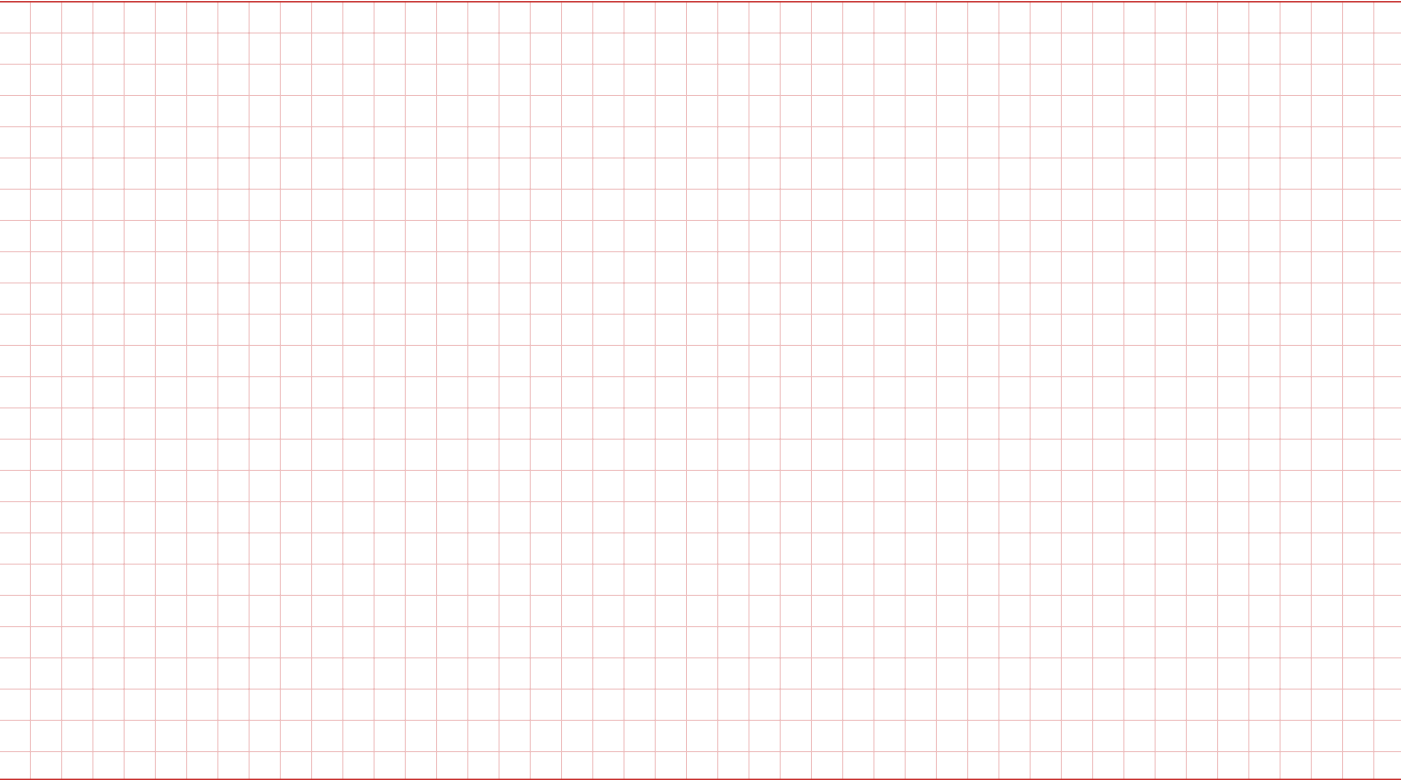
## Byteströme

Byteströme: Lesen und Schreiben

Eingabeströme: `InputStream`

Ausgabeströme: `OutputStream`

## Notizen



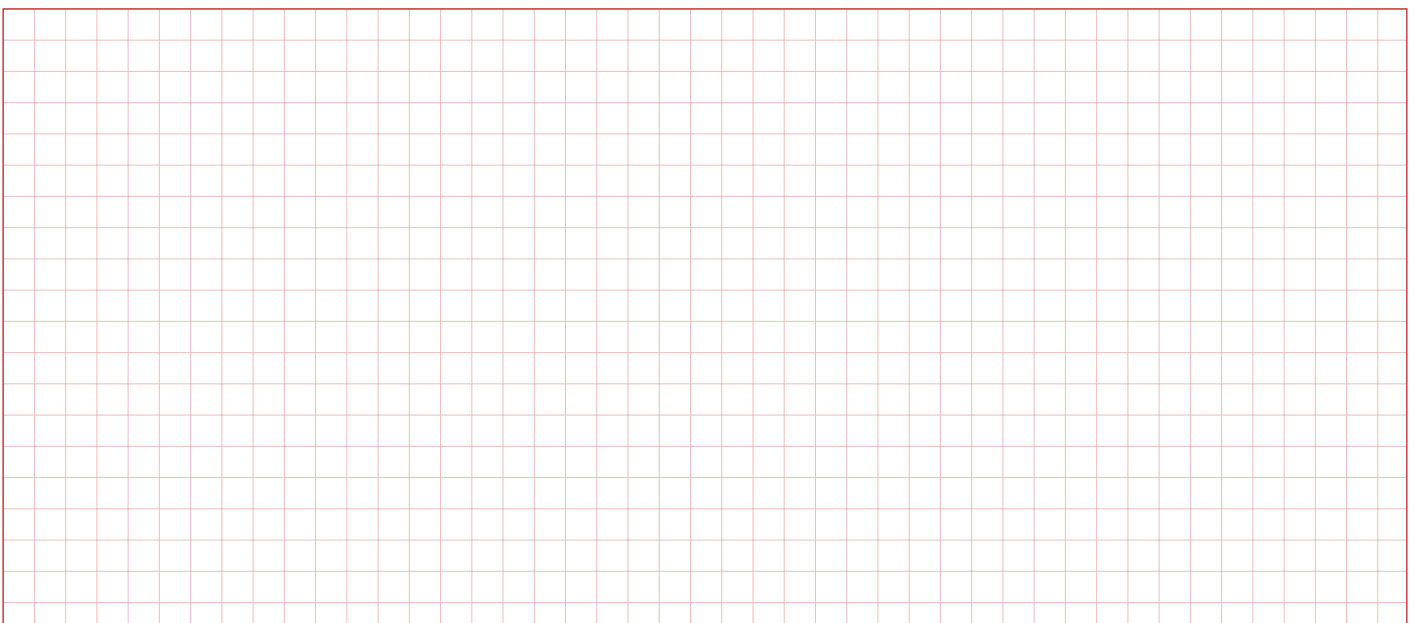
## InputStream

<<abstract>> InputStream	
+ read()	: int
+ read(b : byte[])	: int
+ read(b : byte[], off : int, len : int)	: int
+ skip(n : long)	: long
+ available()	: int
+ reset()	
+ close()	
+ mark(readlimit : int)	
+ markSupported()	: boolean

- ▶ Zum Lesen von **binären Daten** (bytes)
- ▶ **Abstrakte** Oberklasse aller Eingabeströme
  - ▶ Einzige **abstrakte Methode**: **int** read() (kennen wir schon)
  - ▶ **Rest**: kann, muss nicht, aber **sollte** überschrieben werden
- ▶ Methoden können [☞ IOException](#) (**geprüft**) werfen


## Notizen

- mark und markSupported sind die einzigen Methoden, die **keine** throws IOException Deklaration besitzen.

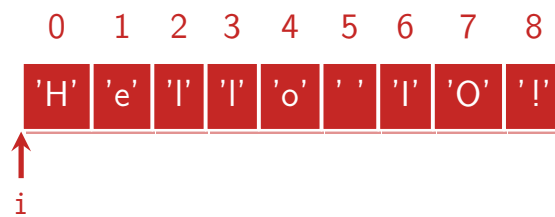


## InputStream — Beispiel

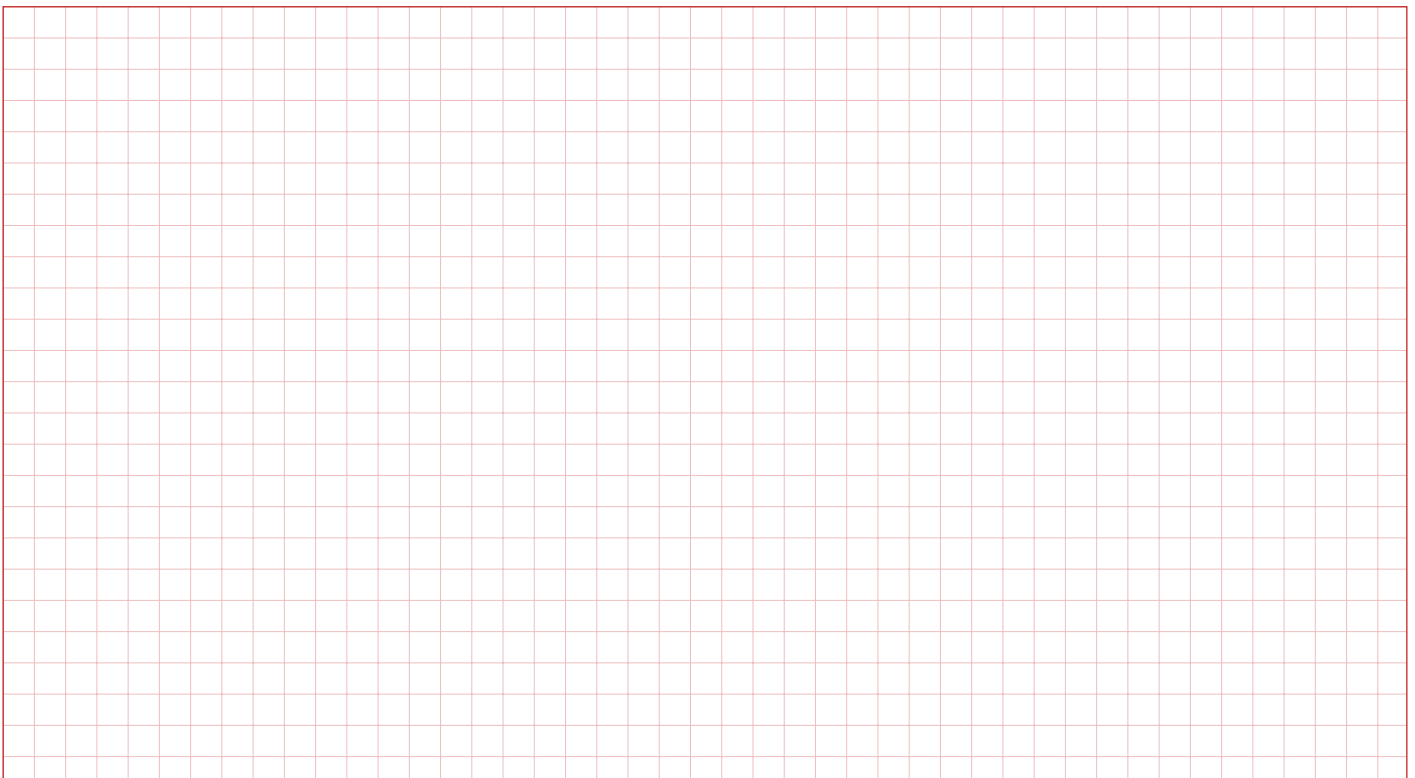
- Beispiel: [ByteArrayInputStream](#) liest Einträge aus einem **byte**-Array

```
63  runInputStreamExample  
64 final byte[] b = { 'H', 'e', 'l', 'l', 'o',  
65   ' ', 'I', 'O', '!' };  
66 ByteArrayInputStream i = new ByteArrayInputStream(b);
```

 `ByteStreamExamples.java`



## Notizen



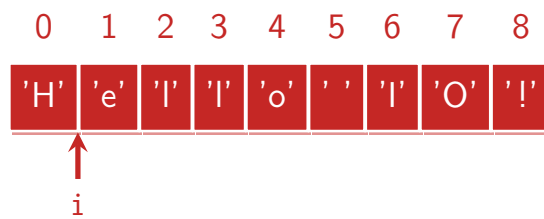


## InputStream.read

- ▶ **int** read() liest nächstes Byte (als **int**), -1 wenn am Ende angekommen
  - ▶ Liest nächstes Byte
  - ▶ **Rückgabe:** gelesenes Byte, -1 wenn am Ende angekommen
- ▶ **Beispiel**

```
70 out.println(i.read()); // 72 == 'H'
```

ByteStreamExamples.java



## Notizen

## InputStream.read

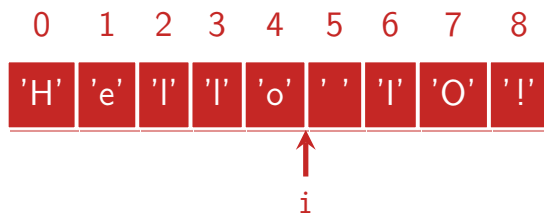
- ▶ `int read(byte[] buffer)`
  - ▶ Liest bis zu `buffer.length` viele **bytes** in `buffer`
  - ▶ Rückgabe: # gelesene **bytes**, -1 wenn am Ende angekommen

## ► Beispiel

```
74 byte[] buffer = new byte[4];
75 int n = i.read(buffer);
76 out.printf("n = %d: %s\n", n, Arrays.toString(buffer));
```

ByteStreamExamples.java

n = 4: [101, 108, 108, 111]



## Notizen

## InputStream.read

- ▶ `int read(byte[] buffer, int offset, int n)`
  - ▶ Liest bis zu `n` viele `bytes` in `buffer` ab `Index offset`
  - ▶ Rückgabe: # gelesene `bytes`, -1 wenn am Ende angekommen

### ▶ Beispiel

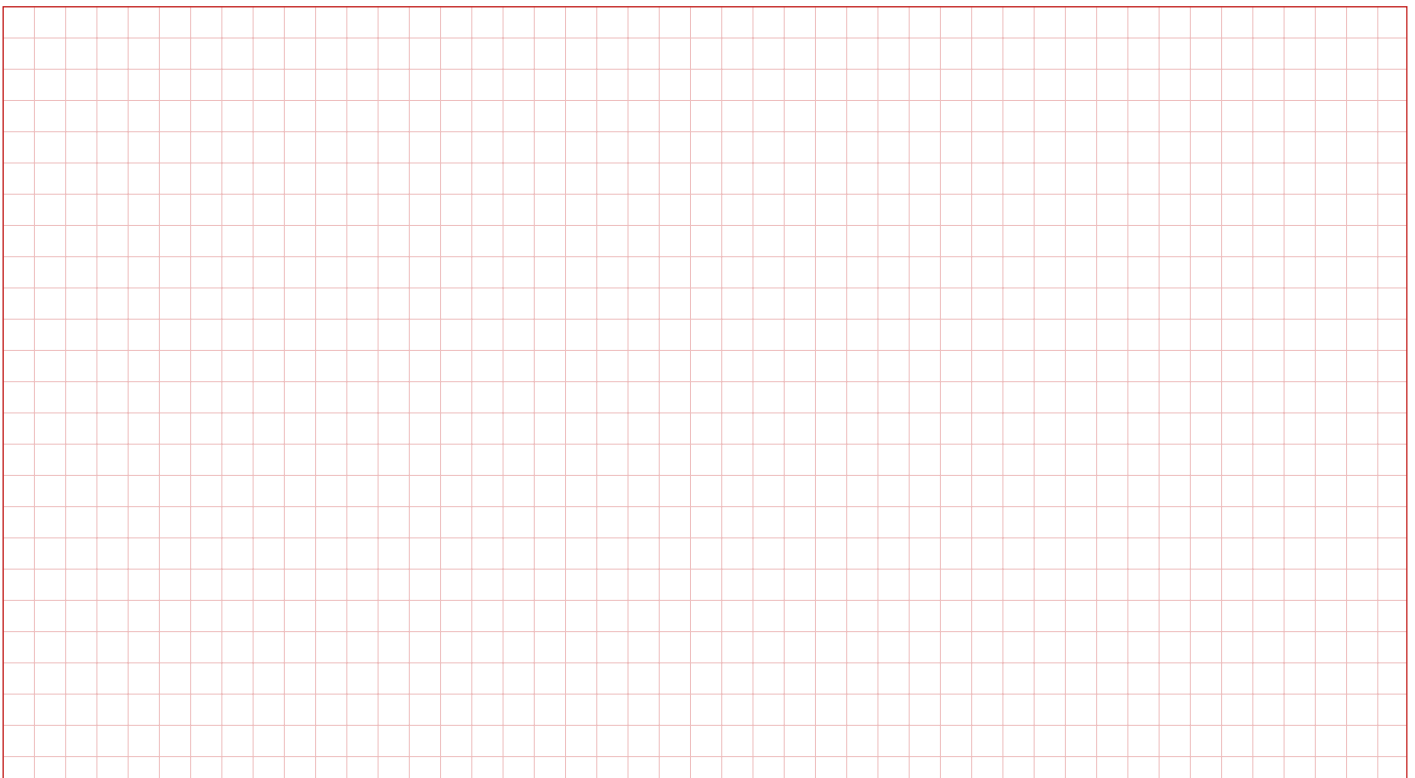
```
80 buffer = new byte[10];  
81 n = i.read(buffer, 3, 7);  
82 out.printf("n = %d: %s\n", n, Arrays.toString(buffer));
```

ByteStreamExamples.java

n = 4: [0, 0, 0, 32, 73, 79, 33, 0, 0, 0]



## Notizen

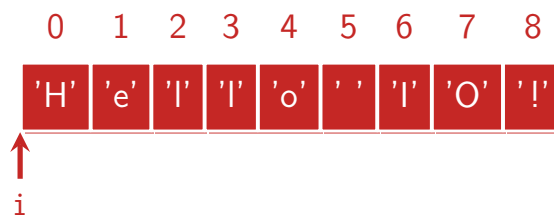


## InputStream.reset

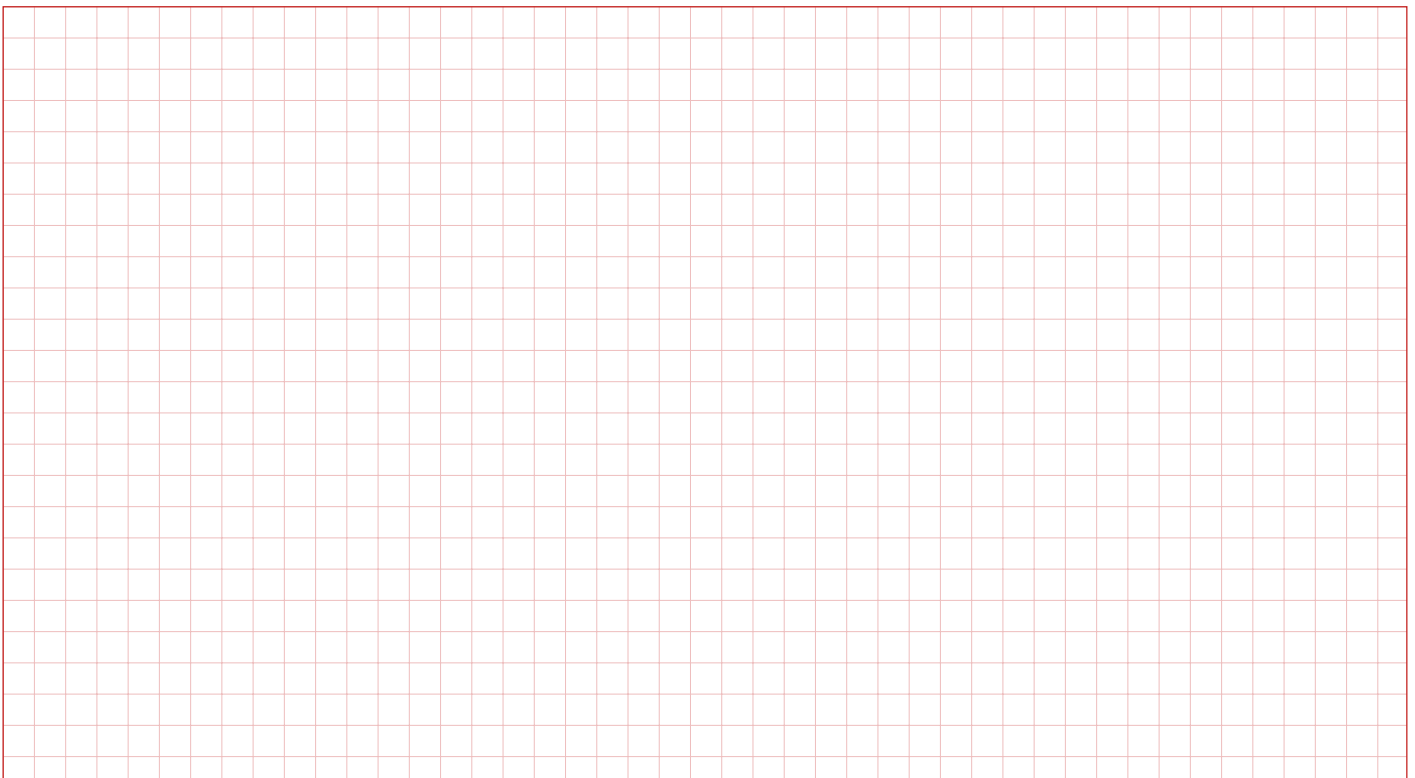
- ▶ **void reset()**
  - ▶ Setzt die Position „zurück“
  - ▶ **Drei** Möglichkeiten
    - ▶ `markSupported()== false` — **Anfang** des Streams (hängt von Stream ab)
    - ▶ `markSupported()== true` — Position als `mark(int)` aufgerufen wurde oder **Anfang**
    - ▶ `IOException` — **ungültig** für Stream
- ▶ **Beispiel**

86 `i.reset();`

`ByteStreamExamples.java`



## Notizen



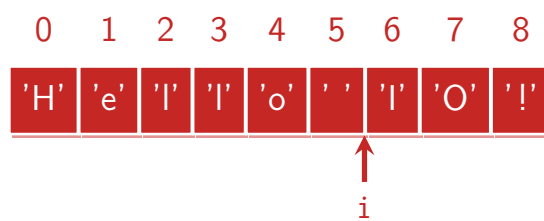
## InputStream.skip

- ▶ `long skip(long n)`
  - ▶ Versucht n **bytes** zu **überspringen**
  - ▶ **Rückgabe**: # übersprungener **bytes** ( $\leq n$ , 0 möglich)
- ▶ Beispiel

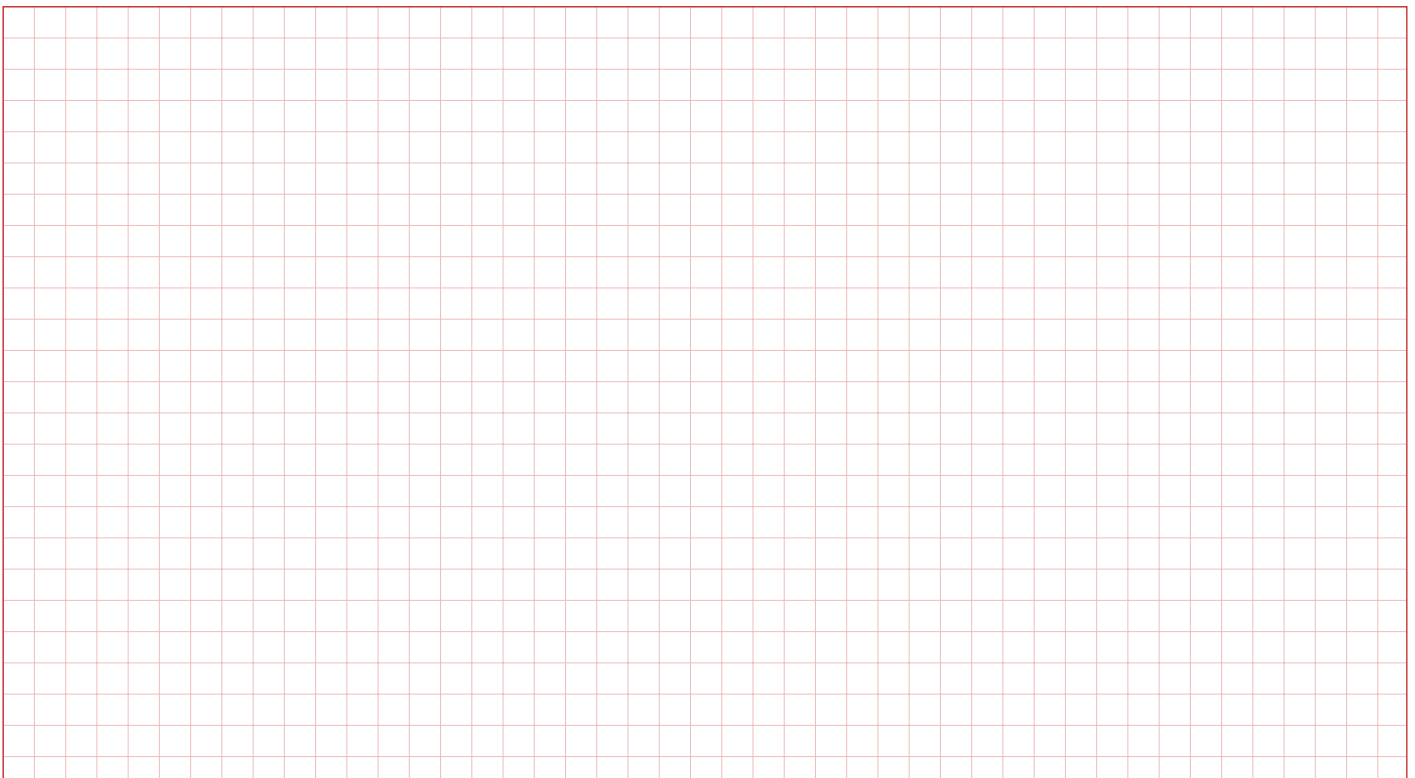
```
90 long l = i.skip(6);  
91 out.printf("l = %d\n", l);
```

ByteStreamExamples.java

l = 6



## Notizen



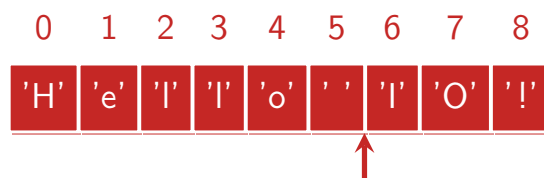
## InputStream.available

- ▶ Hinweis: read/skip können **blockieren**
  - ▶ **Aktueller Thread** wird angehalten bis wieder **Daten verfügbar** sind (Netzwerk, Festplatte, etc.)
- ▶ **int** available()
  - ▶ **Rückgabe**: „Schätzung“ # **byte** die **ohne Blockieren** von read/skip gelesen/übersprungen werden können
  - ▶ **Oft** # restliche **bytes**, aber **nicht immer**
- ▶ Beispiel

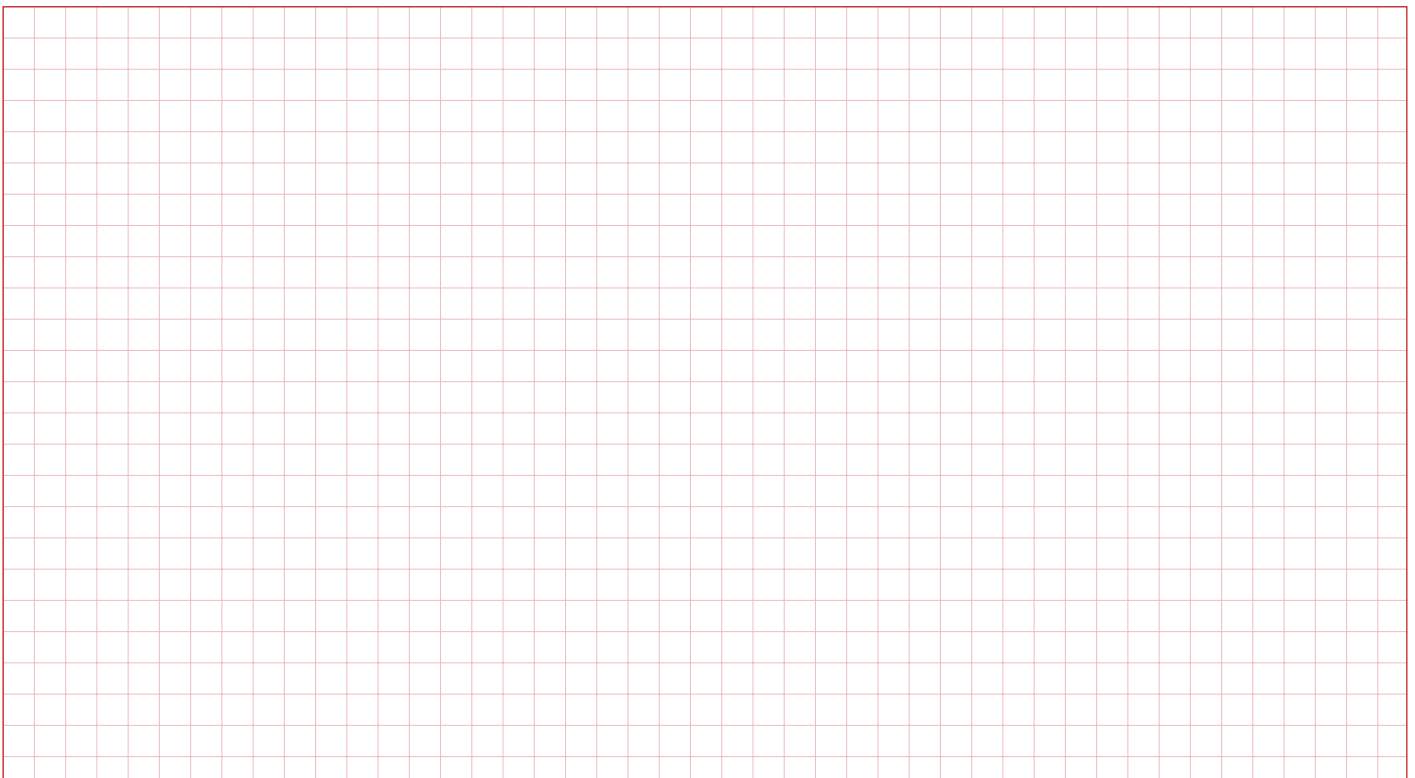
```
95 n = i.available();
96 out.printf("n = %d\n", n);
```

ByteStreamExamples.java

n = 3



## Notizen



## InputStream.mark/markSupported

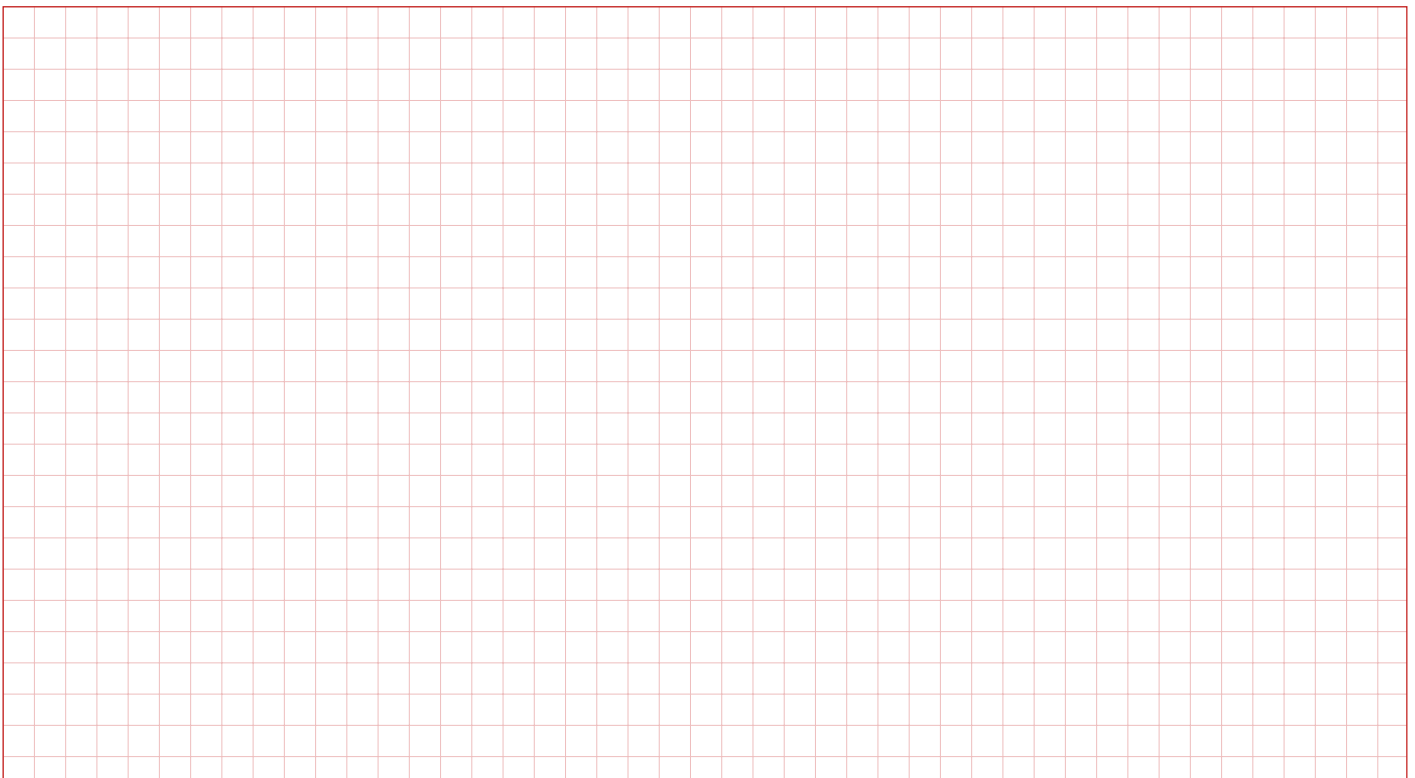
- ▶ **void mark(int readLimit)**
  - ▶ **Markiert** die aktuelle Position
  - ▶ **reset()** springt zu Markierung
  - ▶ **boolean markSupported()** liefert **true** wenn mark **unterstützt wird**
  - ▶ readLimit # verarbeiteter **bytes** bis Markierung **automatisch aufgehoben** wird (schont Ressourcen)
- ▶ **Beispiel**

```
100 i.mark(100);
101 i.skip(3);
102 out.printf("before: available = %d\n", i.available());
103 i.reset();
104 out.printf("after: available = %d\n", i.available());
```

ByteStreamExamples.java

```
before: available = 0
after: available = 3
```

## Notizen



## InputStream.close

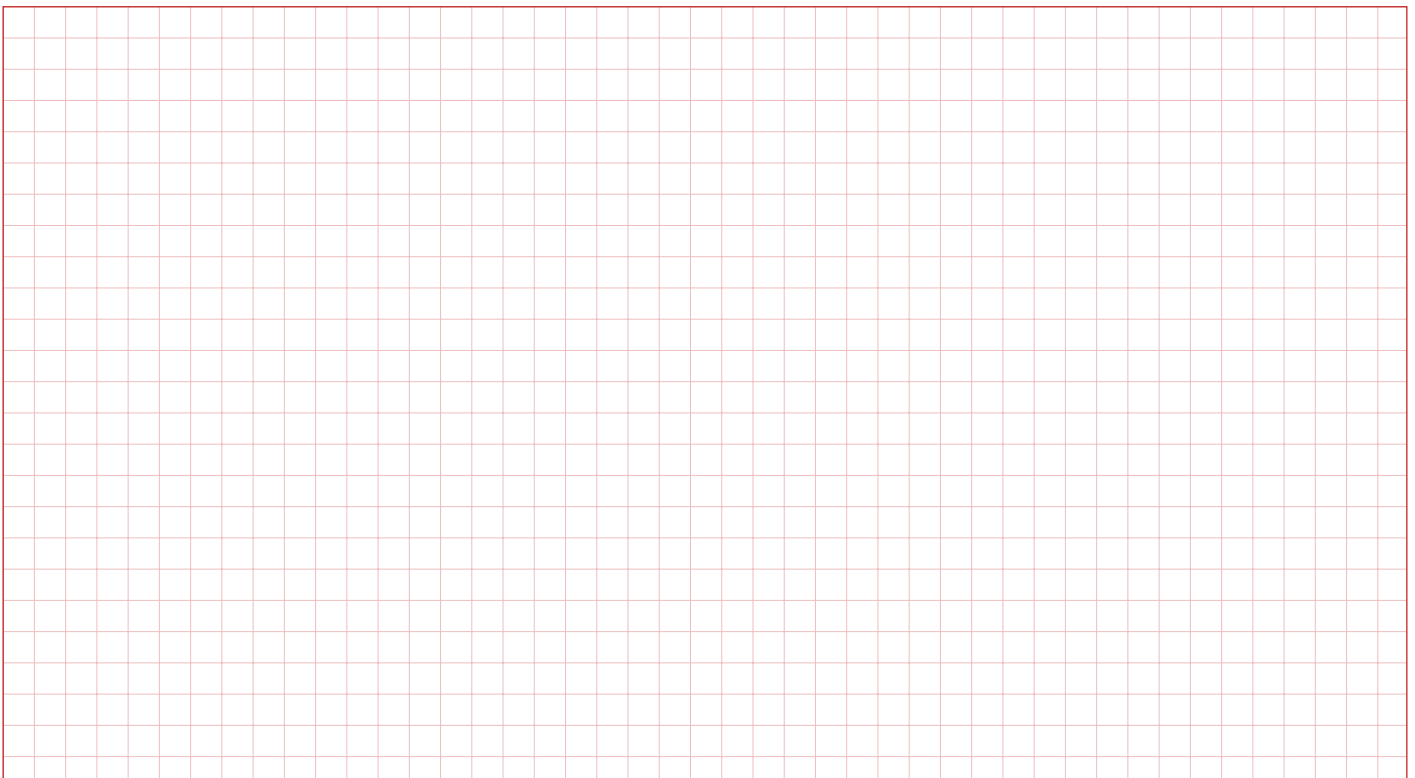
- ▶ **void close()**
  - ▶ Schließt den Stream und gibt Ressourcen frei
- ▶ Beispiel

```
108 i.close();
```

ByteStreamExamples.java

- ▶ Stream i.d.R. danach nicht mehr verwendbar
  - ▶ [FileInputStream](#) schließt Datei
  - ▶ [Socket](#) schließt Netzwerkverbindung
- ▶ Manche Streams funktionieren nach close immer noch
  - ▶ [ByteArrayInputStream](#)
- ▶ close stammt aus Interfaces [AutoCloseable](#) und [Closeable](#) (später)

## Notizen

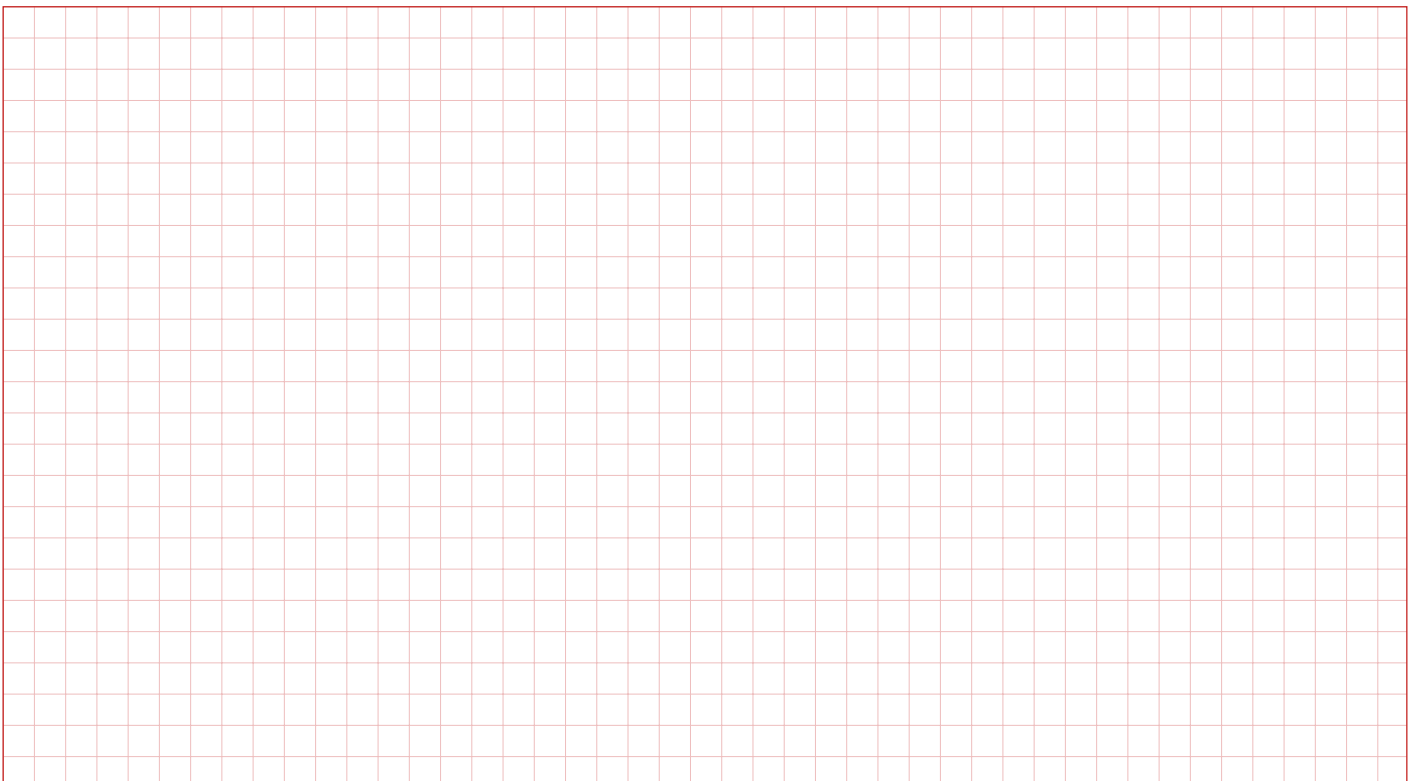




## Weitere Methoden

- ▶ **byte[]** readAllBytes() — liest alle **restlichen bytes**
- ▶ **int** readNBytes(**byte[]** b, **int** off, **int** len)/**byte[]** readNBytes(**int** len)
  - ▶ **Liest** bis zu len viele **bytes** in b ab off/und gibt gelesene **bytes** zurück
  - ▶ **Unterschied** zu read: **blockiert** bis mindestens len **bytes** gelesen wurden
- ▶ **void** skipNBytes(**long** n)
  - ▶ **Überspringt** bis zu n viele **bytes**
  - ▶ **Unterschied** zu skip: **blockiert** bis mindestens n übersprungen wurden
- ▶ **long** transferTo(OutputStream out)
  - ▶ **liest** alle Daten aus Eingabestrom und **schreibt** sie in Ausgabestrom out
  - ▶ **Rückgabe**: **Anzahl** transferierter **bytes**

## Notizen



# Inhalt

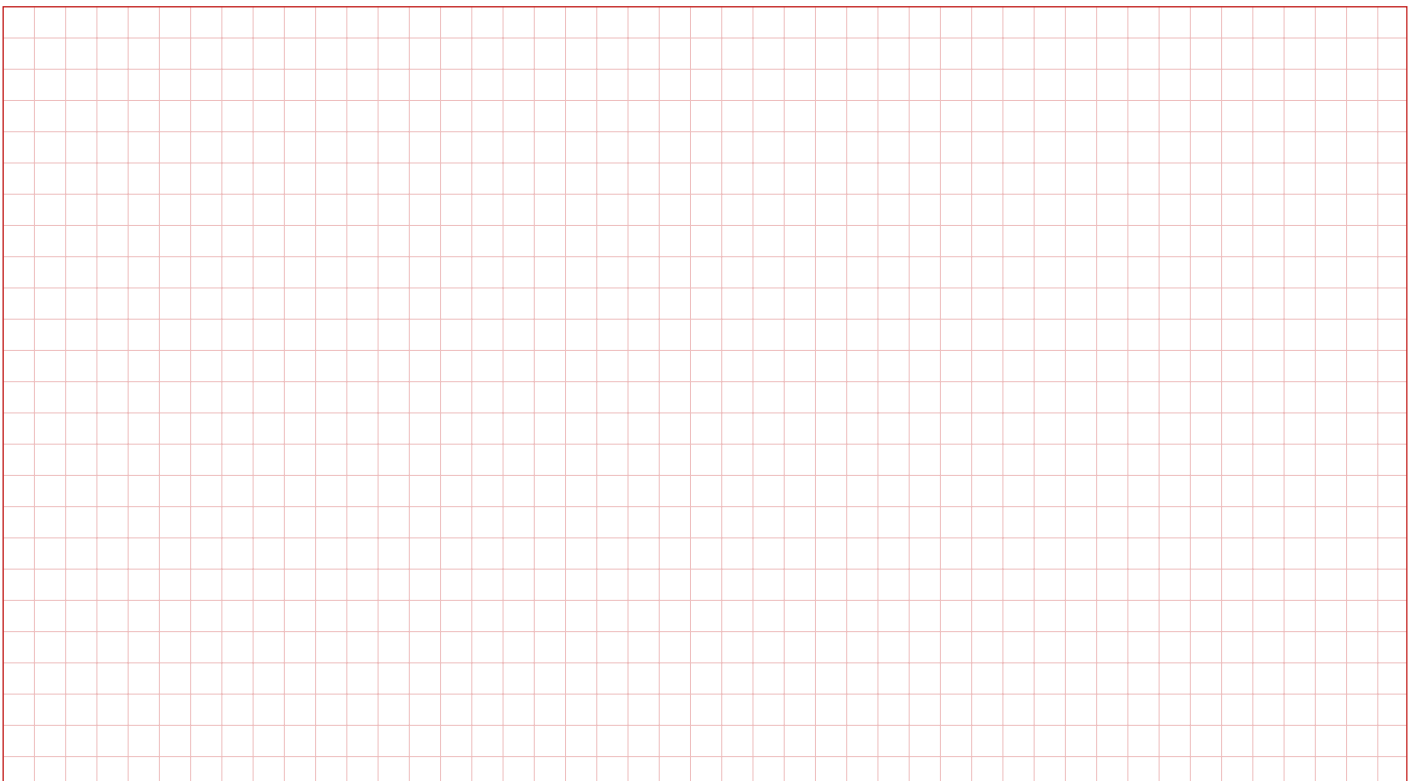
## Byteströme

Byteströme: Lesen und Schreiben

Eingabeströme: `InputStream`

Ausgabeströme: `OutputStream`

## Notizen

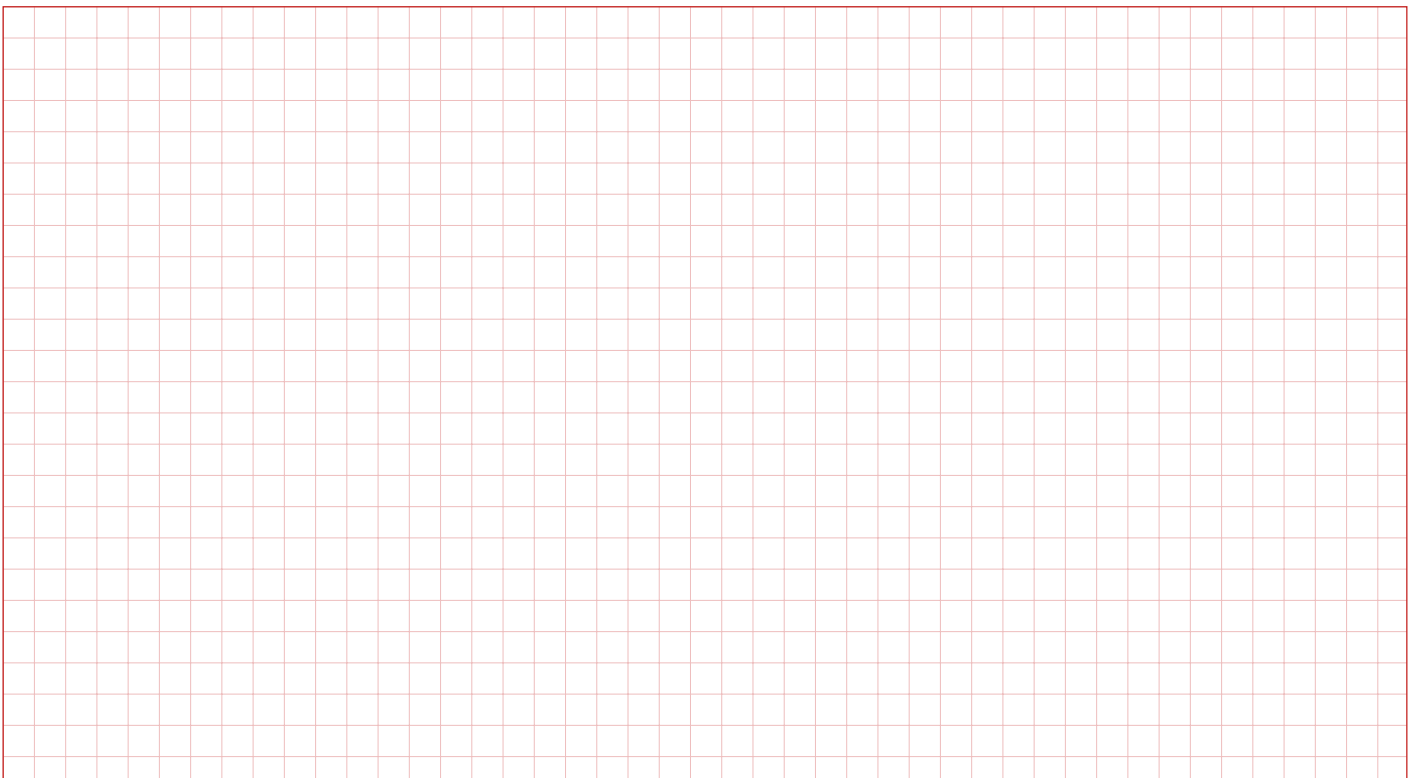


# OutputStream

<div>&lt;&lt;abstract&gt;&gt;</div> <div>OutputStream</div>
<div>+ write(b : int)</div> <div>+ write(b : byte[])</div> <div>+ write(b : byte[], off : int, len : int)</div> <div>+ flush()</div> <div>+ close()</div>


- ▶ Zum Schreiben von **binären Daten (bytes)**
- ▶ **Abstrakte** Oberklasse aller Ausgabeströme
  - ▶ Einzige **abstrakte Methode**: write(**int** b) zum Schreiben eines **einzelnen bytes**
  - ▶ **Rest**: kann, muss nicht, aber **sollte** überschrieben werden
- ▶ Alle Methoden können [☞ IOException](#) (**geprüft**) werfen

## Notizen

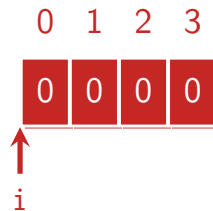


## OutputStream — Beispiel

- ▶ Beispiel: [ByteArrayOutputStream](#) schreibt **bytes** in einen **byte-Array**

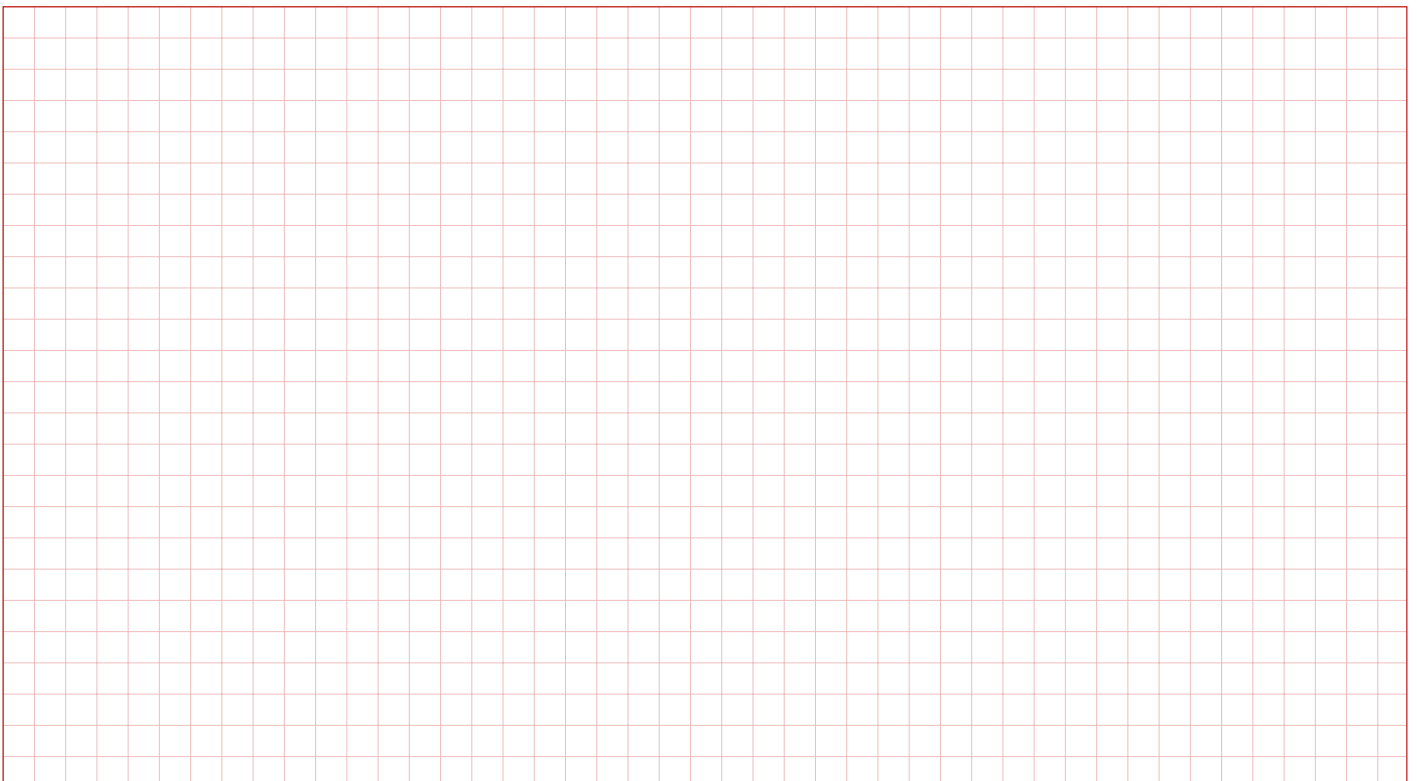
```
114  runOutputStreamExample  
115 ByteArrayOutputStream o = new ByteArrayOutputStream(4);
```

[ByteStreamExamples.java](#)



- ▶ Funktioniert ähnlich wie [ArrayList](#)
  - ▶ Konstruktor mit **initialer Kapazität**
  - ▶ **Kapazität** wird bei Bedarf **vergrößert**
- ▶ **byte[]** `toByteArray()` liefert **resultierenden byte-Array**

## Notizen



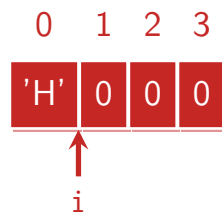
## OutputStream.write

- ▶ `void write(int b)` — schreibt byte in [OutputStream](#)
  - ▶ `int` wird zu Byte *gecastet*
- ▶ Beispiel

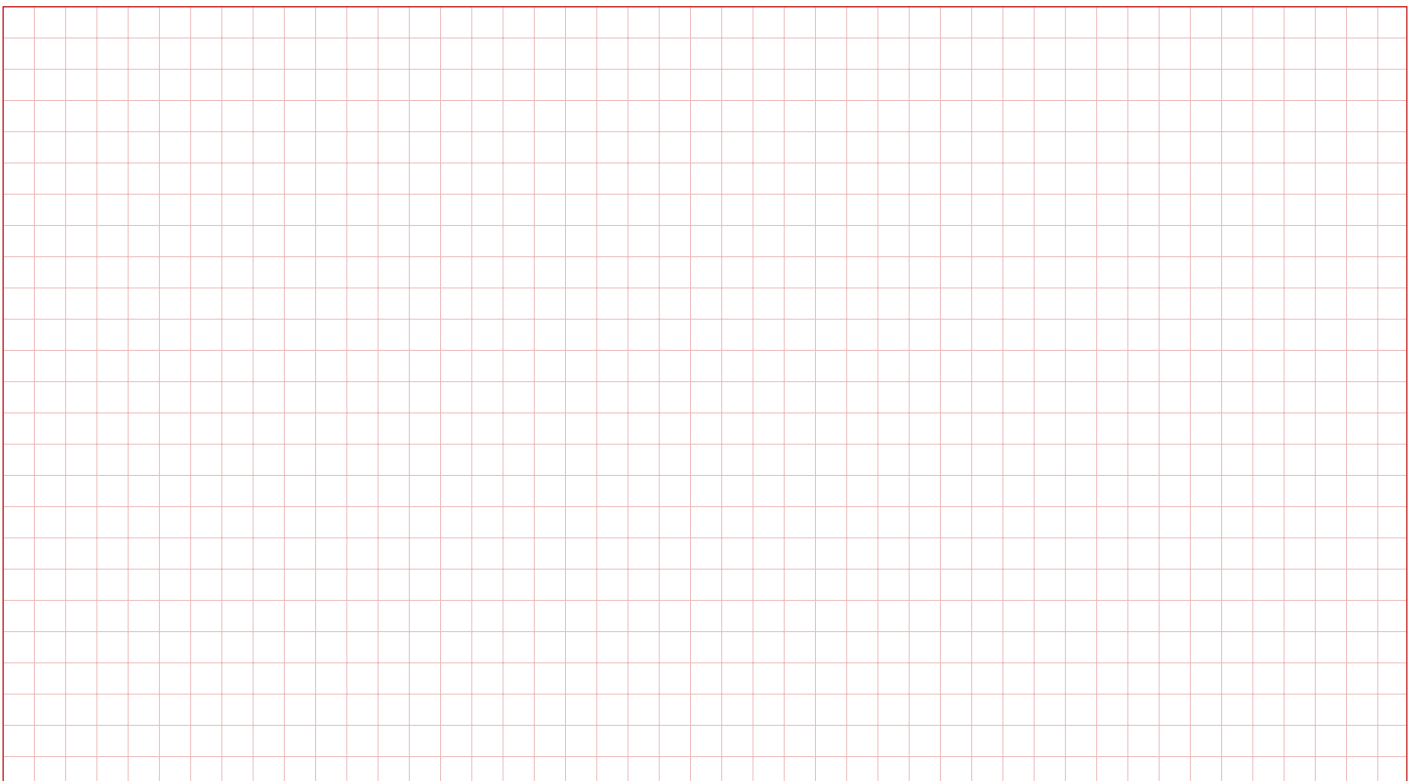
```
119 o.write('H');  
120 out.println(Arrays.toString(o.toByteArray()));
```

[ByteStreamExamples.java](#)

```
[72] // == 'H'
```



## Notizen



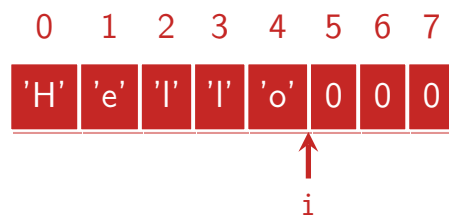
## OutputStream.write

- ▶ `void write(byte[] b)` und `void write(byte[] b, int offset, int length)`
  - ▶ **Schreibt bytes** aus `b` in Stream
  - ▶ `b.length` viele **oder** von `b[offset]` bis `b[offset+length-1]`
- ▶ Beispiel

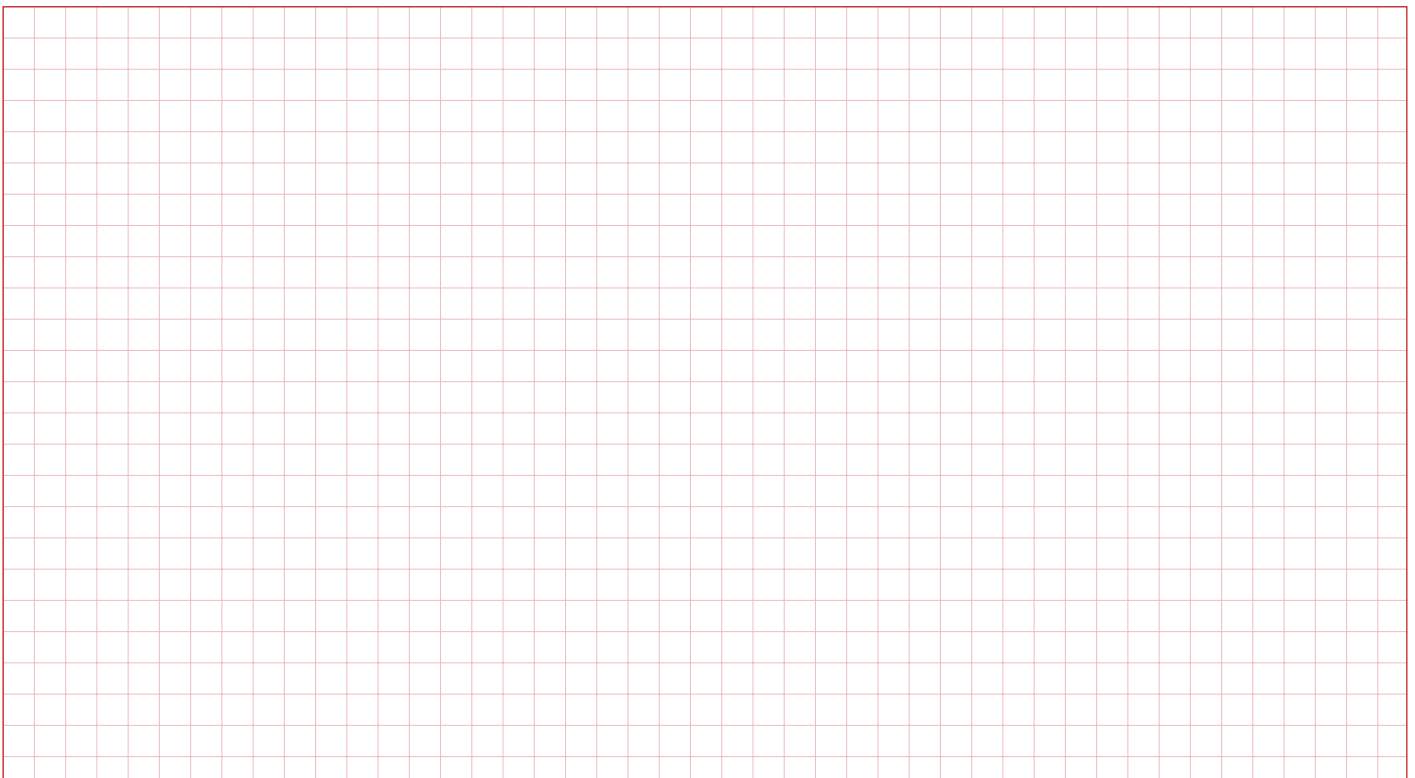
```
124 byte[] b = { 'e', 'l', 'l', 'o' };  
125 o.write(b);  
126 out.println(Arrays.toString(o.toByteArray()));
```

ByteStreamExamples.java

[72, 101, 108, 108, 111]



## Notizen



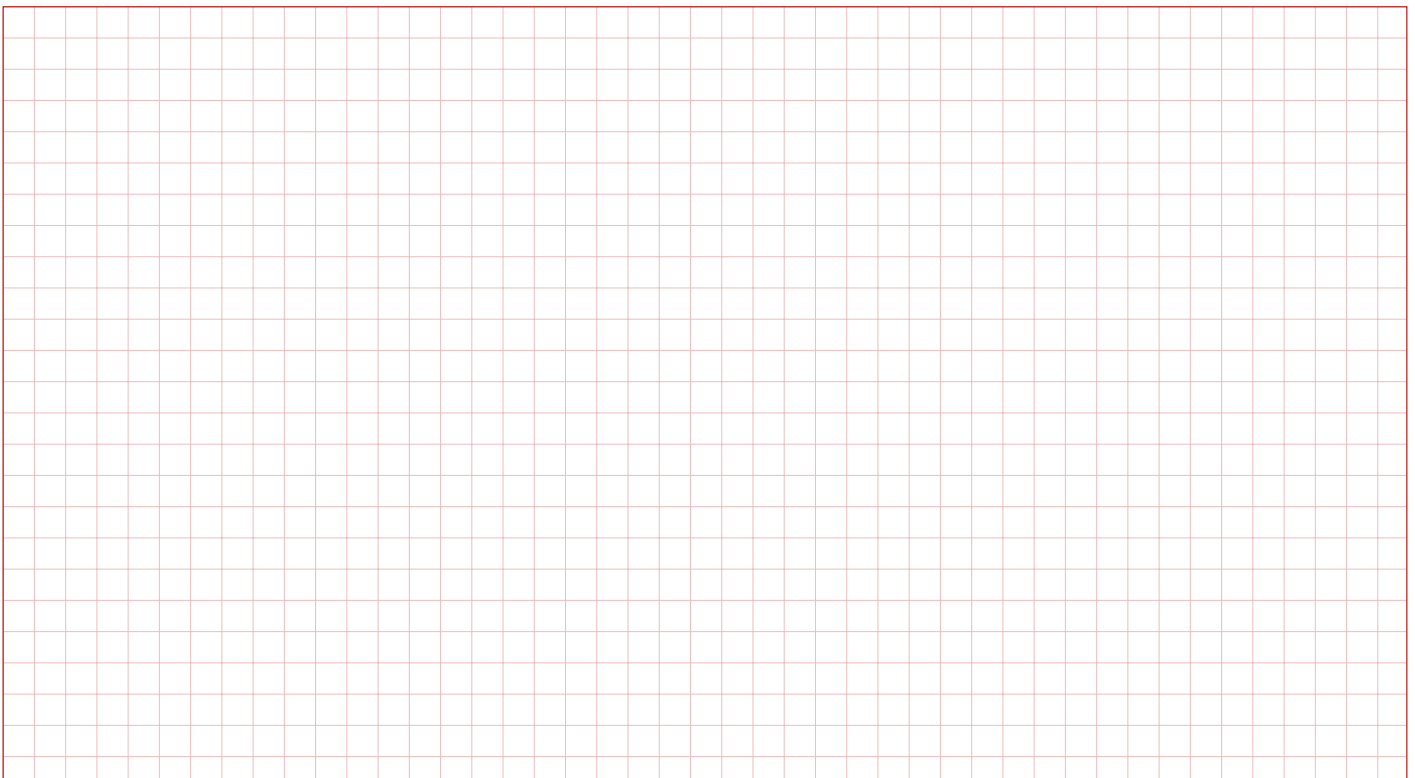
## OutputStream.flush/close

- ▶ `OutputStream.flush()`
  - ▶ Datenströme **puffern** für Effizienz
  - ▶ Daten werden vor eigentlichem Schreiben in **Puffer angesammelt**
  - ▶ ... und dann „**in einem Rutsch**“ geschrieben (Festplatte, Netzwerk, etc.)
  - ▶ `flush()` **erzwingt vorzeitiges Schreiben**
  - ▶ **Achtung:** Nach Rückkehr von `flush` **keine Garantie**, dass Daten angekommen sind
- ▶ `OutputStream.close()`
  - ▶ vgl. `InputStream.close()`: **Schließt** Datenstrom und gibt **Ressourcen frei**
  - ▶ Impliziter Aufruf von `flush()`
- ▶ `ByteArrayOutputStream.close/flush` hat **keine Auswirkung**

```
130 o.flush();  
131 o.close();
```

`ByteStreamExamples.java`

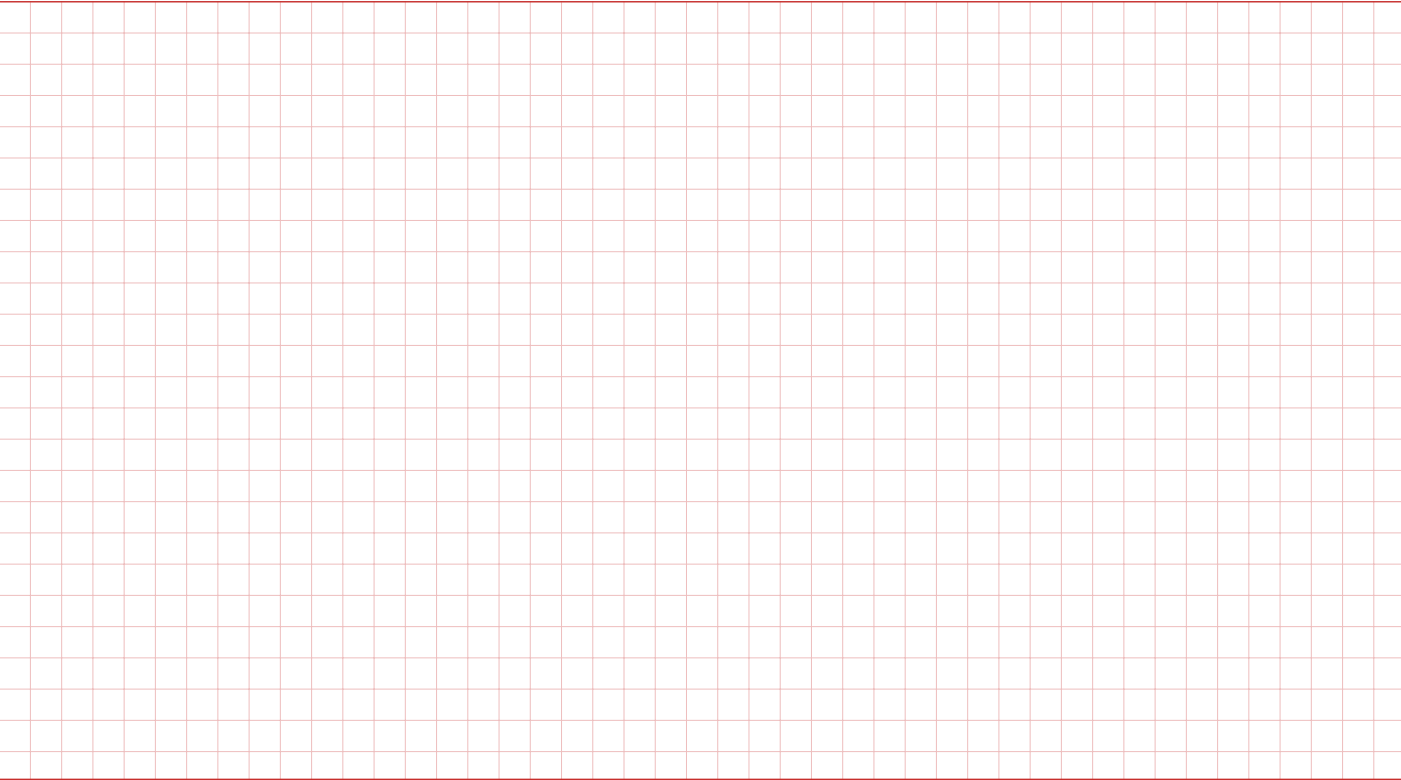
## Notizen



# Inhalt

- Byteströme
  - Quellen für Eingabeströme

## Notizen





# Beispielprogramm

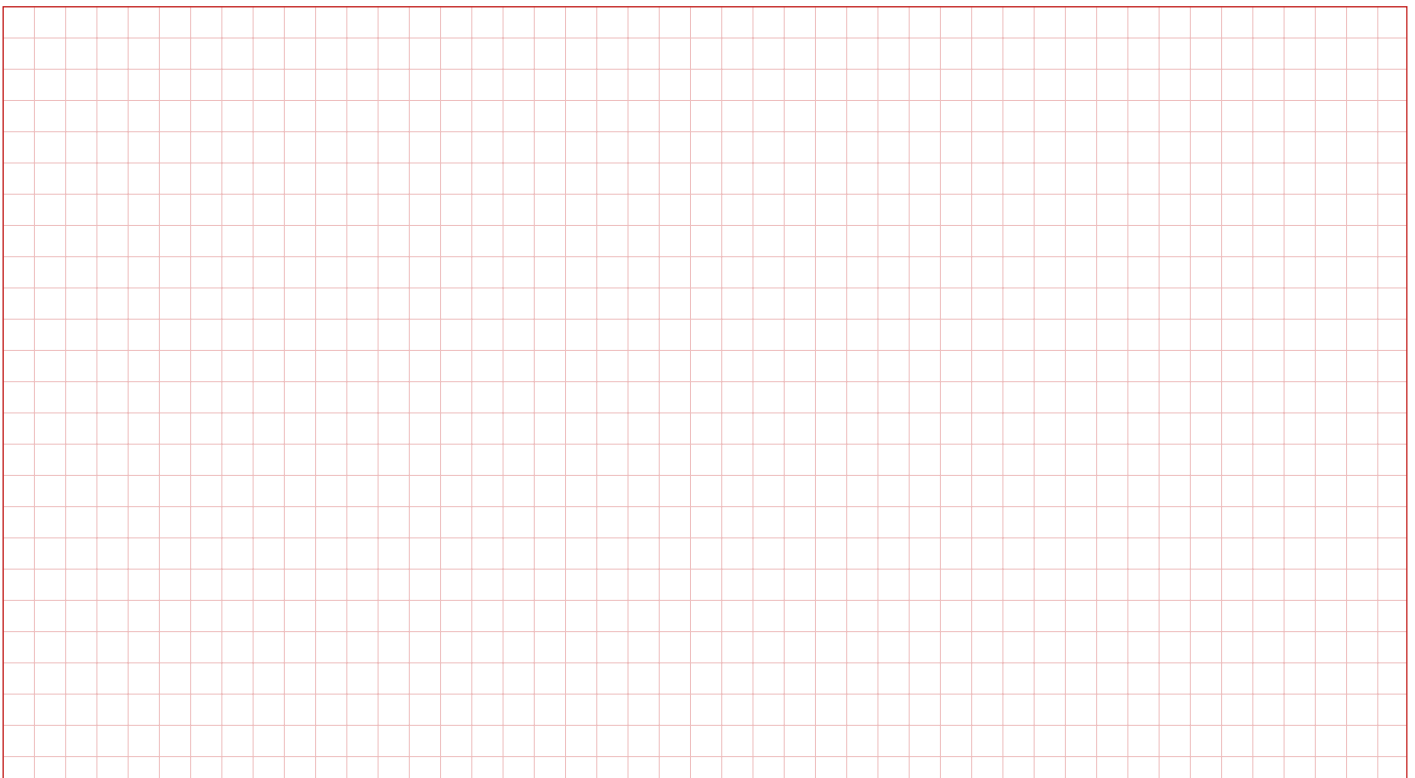
## ► Beispielprogramm

```
19 public static void readAndPrint(InputStream in) {  
20     try{  
21         int i;  
22         while ((i = in.read()) >= 0){  
23             out.printf("%d (%c)%n", i, (char) i);  
24         }  
25     } catch (IOException e){  
26         out.println(e.getMessage());  
27     }  
29 }
```


📄 Sources\SinksExamples.java

- Liest alle Zeichen und gibt sie aus
- Als `int` und `char`

## Notizen



## Standardeingabe

- ▶  `InputStream` `System.in`
  - ▶ Standardeingabestrom (schon gesehen)
  - ▶ Benutzereingaben auf Terminal, **umgeleitete** Eingabe

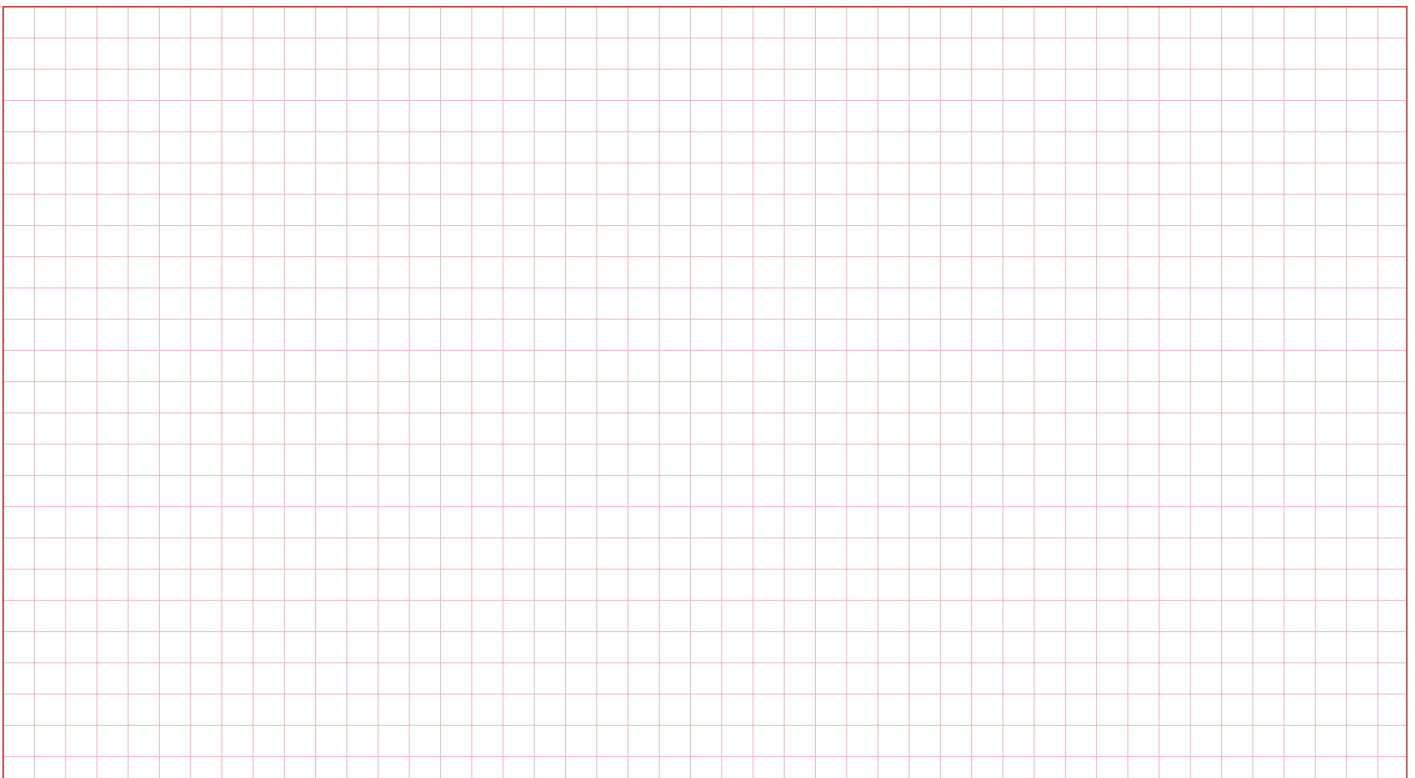
### ▶ Beispiel

```
35  runSourceSystemIn  
36 readAndPrint(System.in);
```

 SourcesSinksExamples.java

```
Java<ENTER> // Eingabe auf Terminal  
74 (J)  
97 (a)  
118 (v)  
97 (a)  
10 (  
)
```

## Notizen



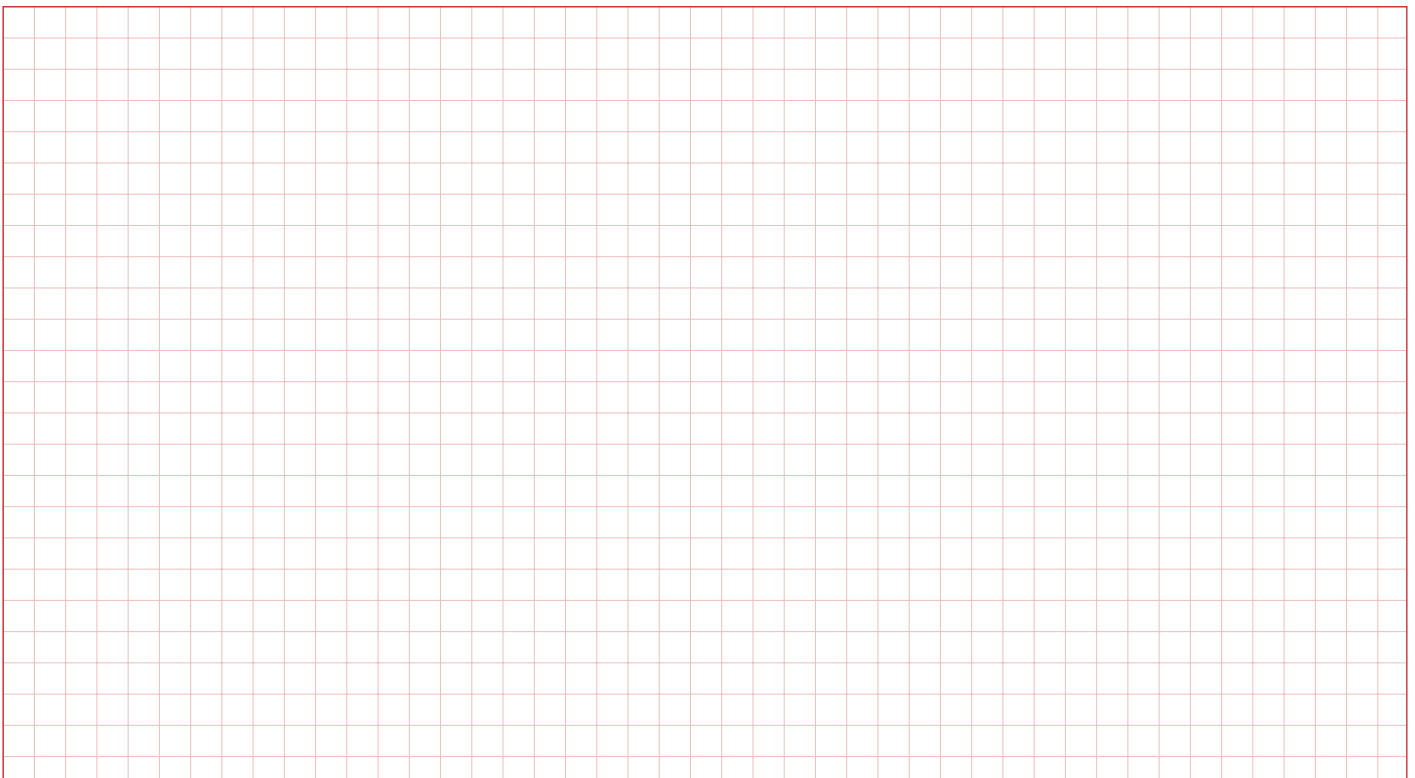
# Standardeingabestrom (umgeleitet)

## ► Umgeleiteter Eingabestrom (Linux-/Unix-Terminal)

```
echo "Java" | gradle runSourceSystemIn
74 (J)
97 (a)
118 (v)
97 (a)
10 (
)
```


- echo "Java" gibt Java<ENTER> aus
- | („Pipe“) leitet **Ausgabe** von echo in **Eingabe** von Java-Programm um
- Java-Programm liest **Eingabe** und gibt sie aus

## Notizen



## FileInputStream

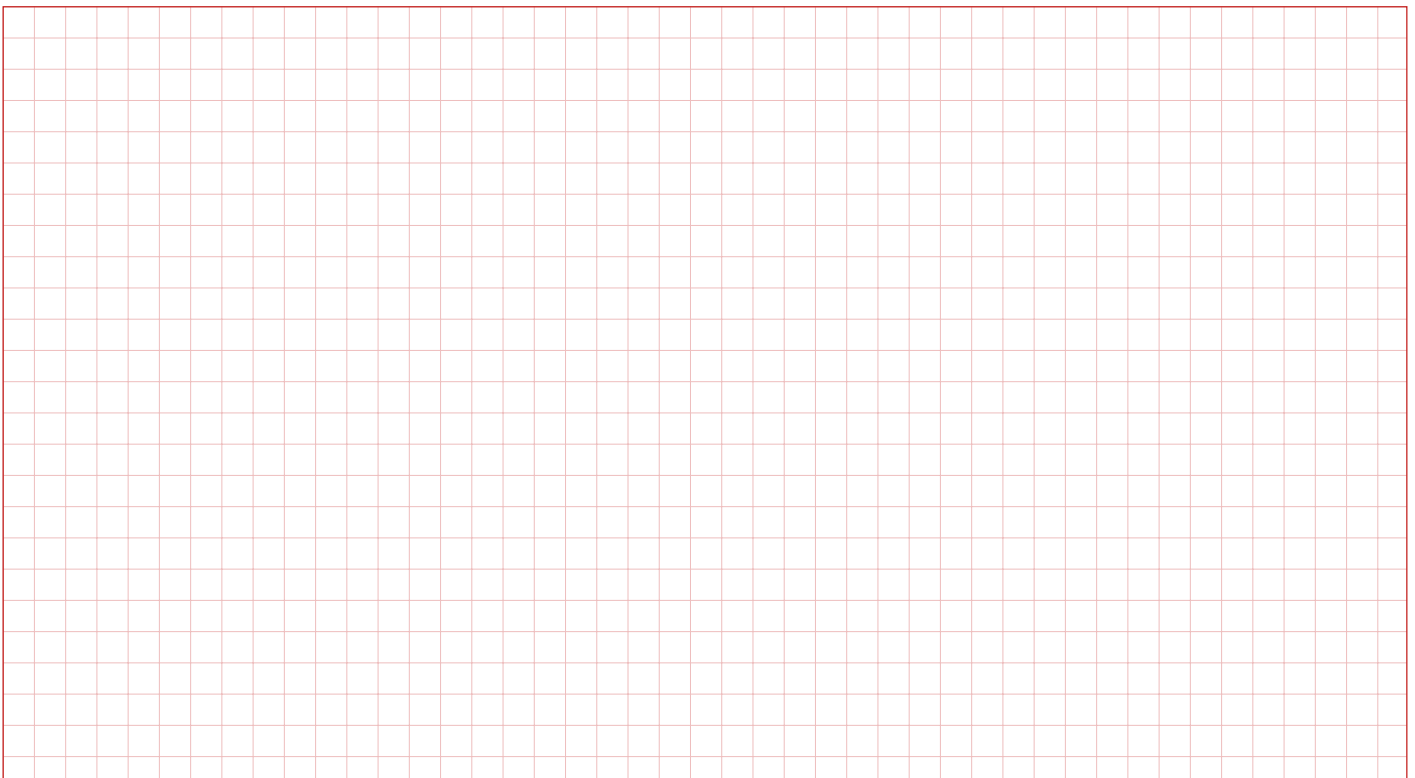
- ▶ [FileInputStream](#)
  - ▶ Eingabestrom aus **Datei** (schon gesehen)
- ▶ Beispiel

```
42  runSourceFileInputStream  
43 FileInputStream in = new FileInputStream("input.txt");  
44 readAndPrint(in);
```

 SourcesSinksExamples.java

```
echo "Java" > input.txt  
gradle runSourceFileInputStream  
74 (J)  
97 (a)  
118 (v)  
97 (a)  
10 (  
)
```

## Notizen



## ByteArrayInputStream

### ► [ByteArrayInputStream](#)

- Stellt **byte**-Array als Eingabestrom bereit (kenn wir auch schon)

### ► Beispiel

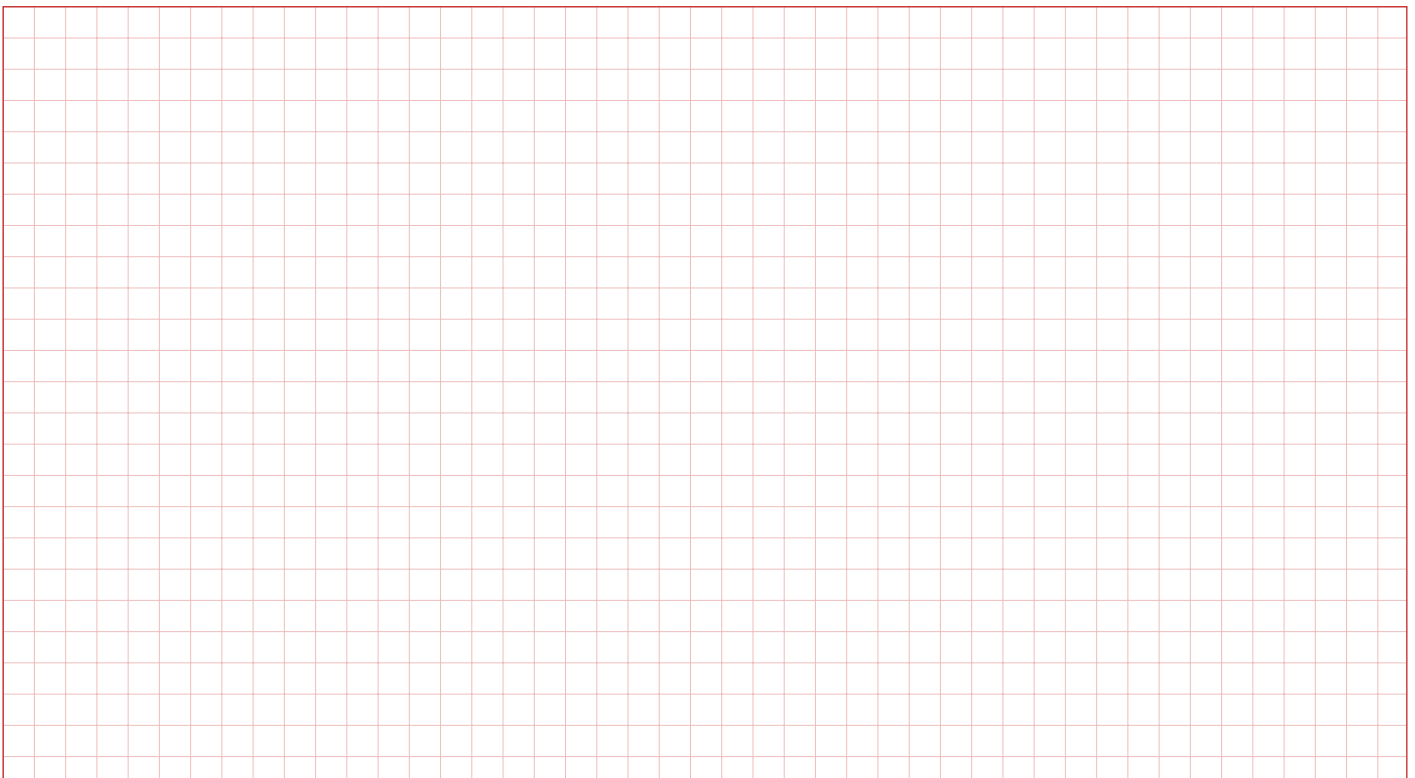
#### 50 runSourceByteArrayInputStream

```
51 byte[] a = new byte[] { 'J', 'a', 'v', 'a', '\n' };  
52 ByteArrayInputStream in = new ByteArrayInputStream(a);  
53 readAndPrint(in);
```

[SourcesSinksExamples.java](#)


```
74 (J)  
97 (a)  
118 (v)  
97 (a)  
10 (  
)
```

## Notizen



## PipedInputStream


### ► PipedInputStream

- Leitet  PipedOutputStream in  PipedInputStream weiter



- Lässt Java-Programm **intern** über Streams kommunizieren

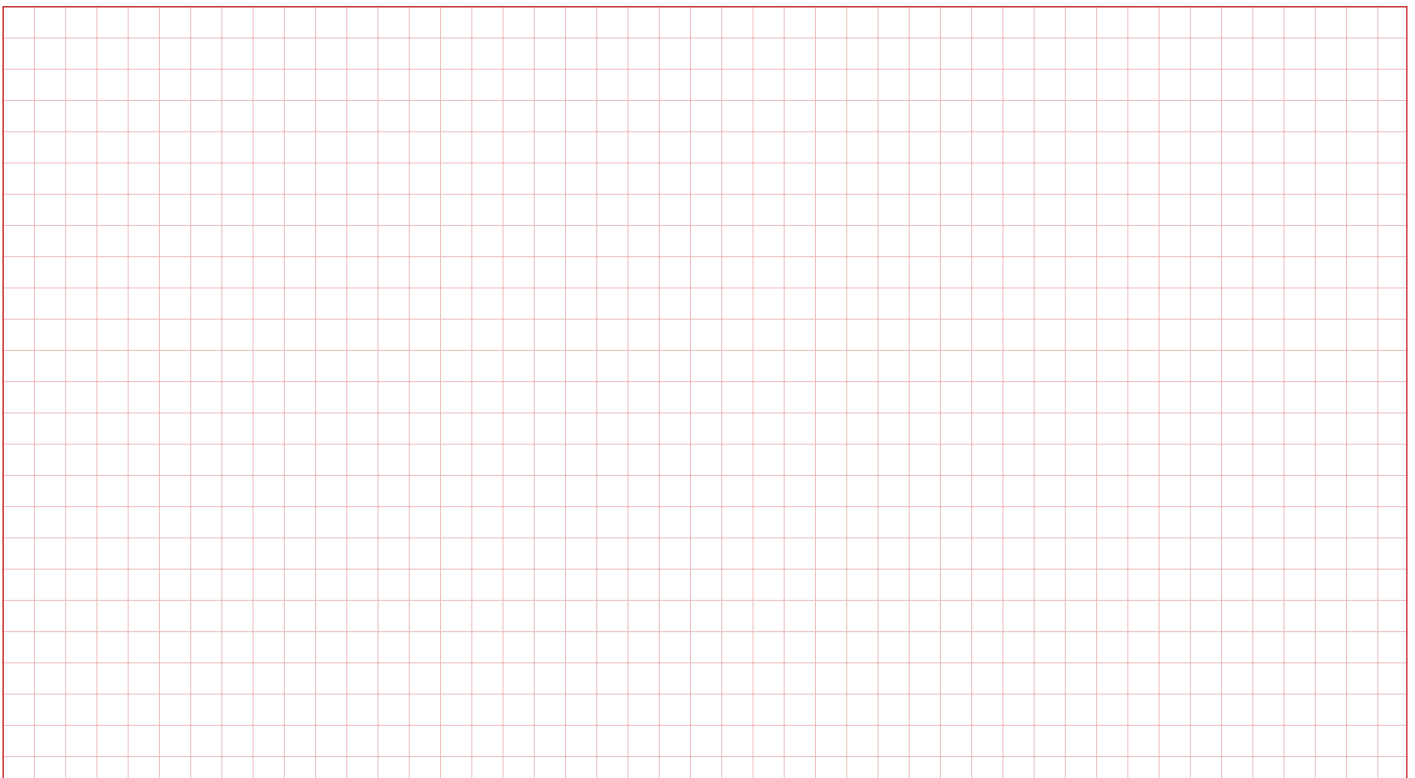
### ► Beispiel

```
59  runSourcePipedInputStream  
60 PipedOutputStream out = new PipedOutputStream();  
61 PipedInputStream in = new PipedInputStream(out);  
62 out.write( new byte[] { 'J', 'a', 'v', 'a', '\n' } );  
63 readAndPrint(in);
```

 SourcesSinksExamples.java

```
74 (J)  
97 (a)  
...
```

## Notizen



# Inhalt

- Byteströme
- Senken für Ausgabeströme

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

# Beispielprogramm

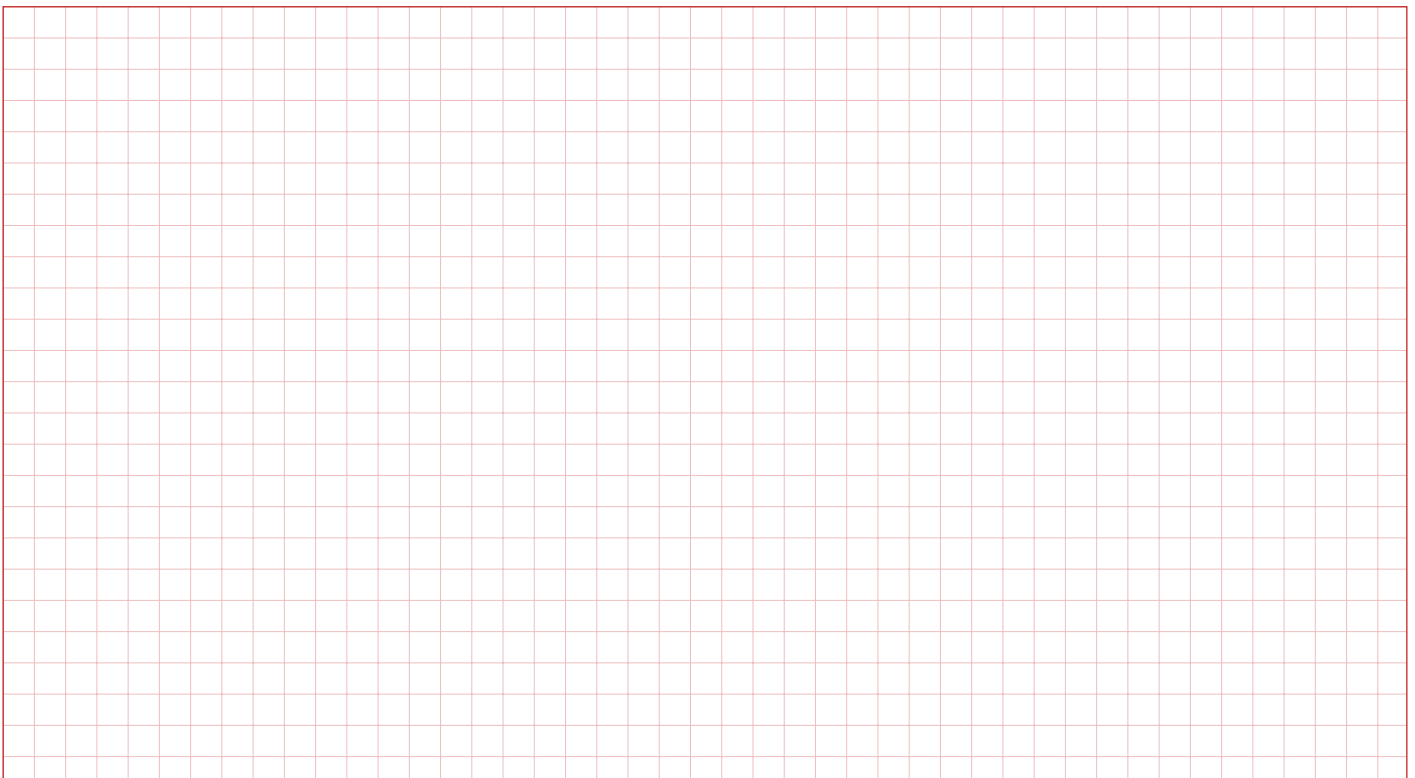
## ► Beispielprogramm

```
68 public static void writeJava(OutputStream out) {  
69     try{  
70         out.write(new byte[] { 'J', 'a', 'v', 'a', '\n' } );  
71         out.close();  
72     }catch (IOException e){  
73         System.err.println(e.getMessage());  
74     }  
75 }
```

📄 SourcesSinksExamples.java

- **Schreibt** „Java\n“ in den Ausgabestrom
- **Schließt** den Strom


## Notizen





## System.out und System.err

- ▶ [System.out](#): Nutzerausgaben auf Terminal, umgeleitete Ausgabe
- ▶ [System.err](#): Fehlerausgabe für Fehlermeldungen
- ▶ Beispiel

```
81  runSinkSystemOutErr
82 writeJava(System.out);
83 writeJava(System.err);
```

 SourcesSinksExamples.java

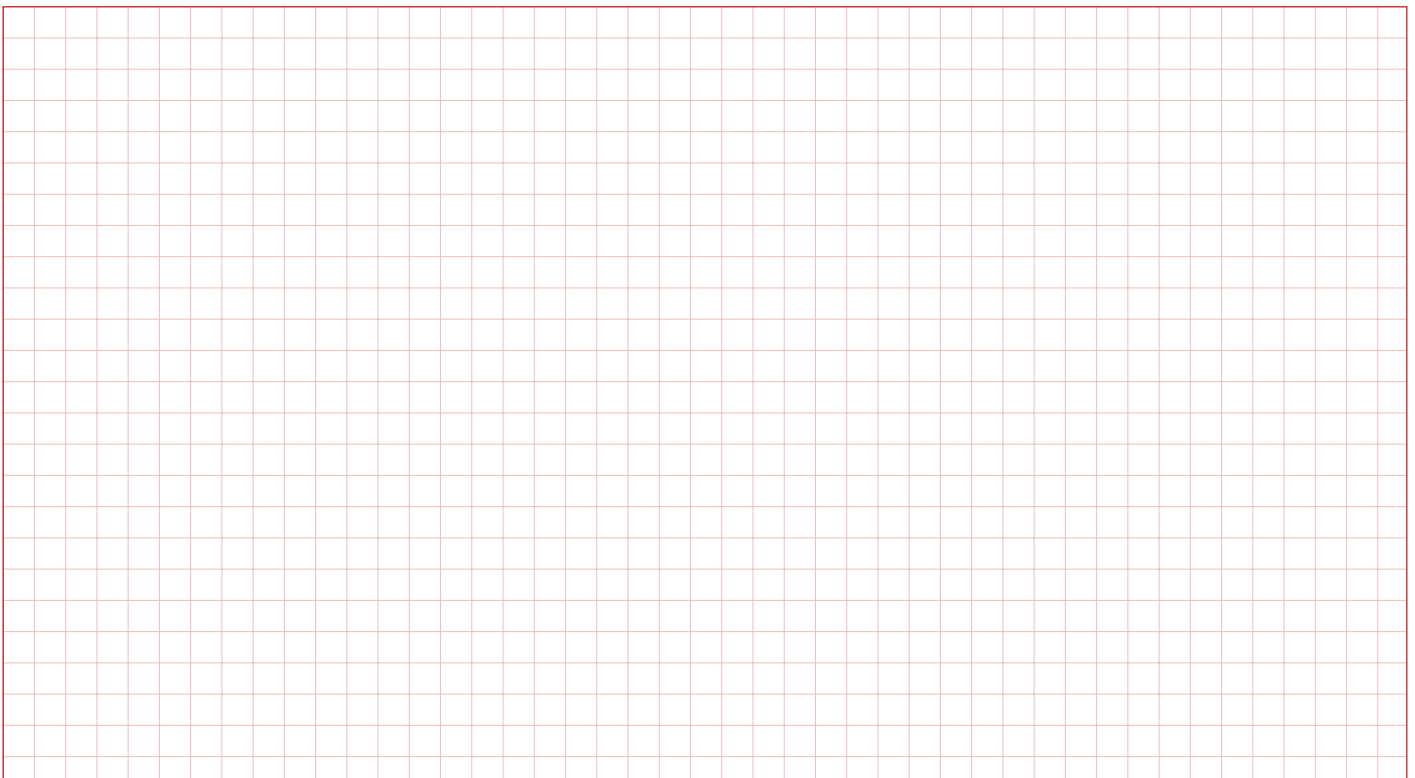
Java

Java

- ▶ Beispiel umgeleitete Ausgabe

```
$ gradle runSinkSystemOutErr 1> out.txt 2> err.txt
$ cat out.txt
Java
$ cat err.txt
Java
```


## Notizen



## FileOutputStream

- ▶ [FileOutputStream](#)
  - ▶ Ausgabestrom in Dateien

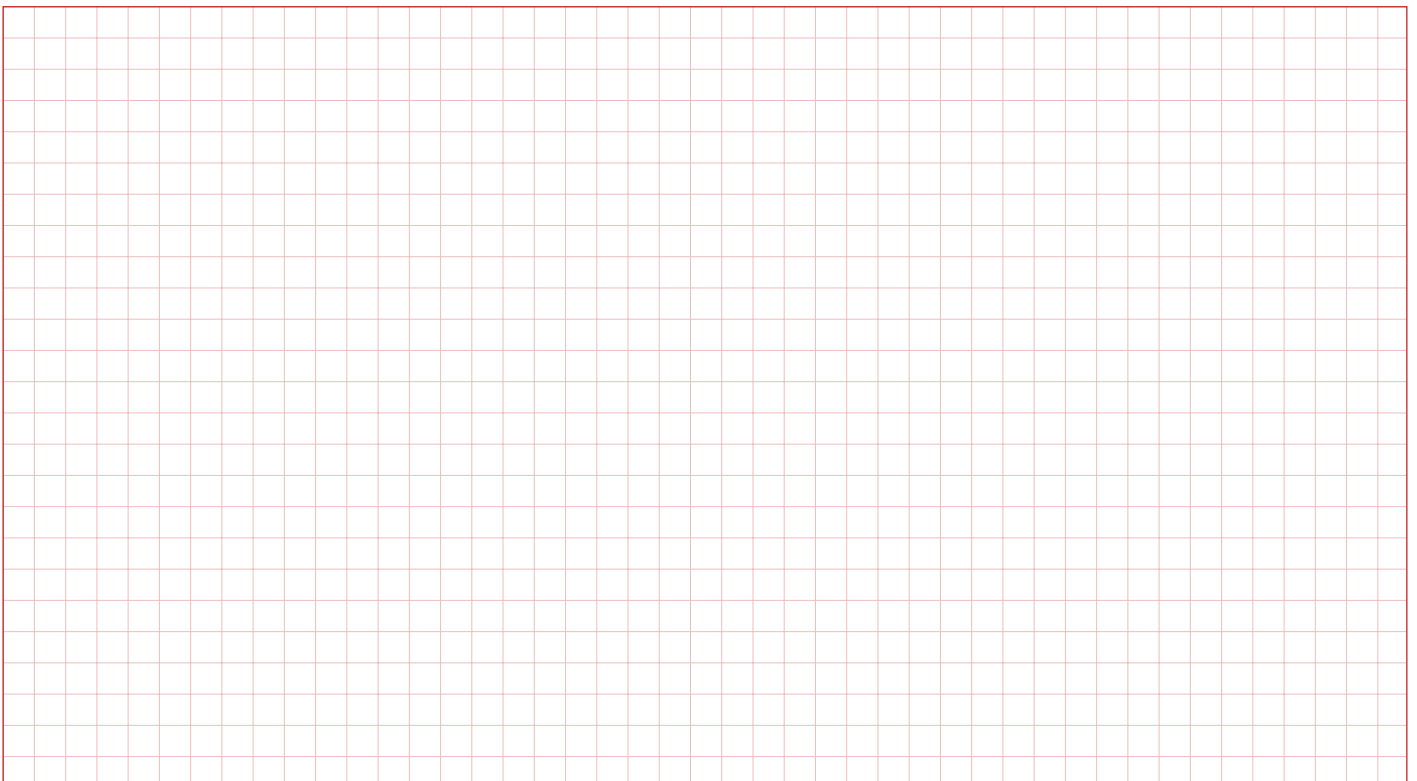
### ▶ Beispiel

```
89  runSinkFileOutputStream  
90 FileOutputStream out = new FileOutputStream("output.txt");  
91 writeJava(out);
```

 SourcesSinksExamples.java

```
$ gradle runSinkFileOutputStream  
$ cat output.txt  
Java
```


## Notizen



## ByteArrayOutputStream

- ▶ [ByteArrayOutputStream](#)
  - ▶ Ausgabestrom in **byte**-Array

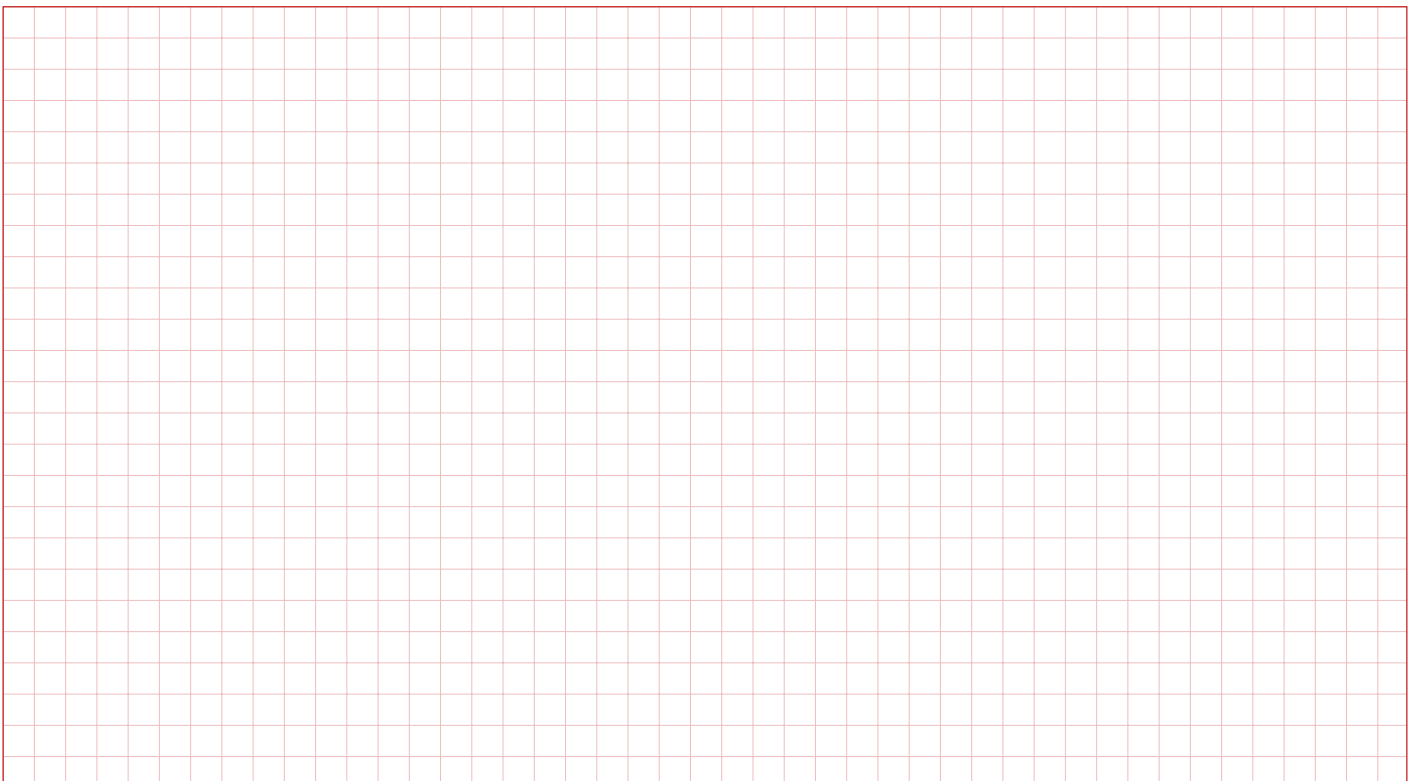
### ▶ Beispiel

```
97  runSinkByteArrayOutputStream  
98 ByteArrayOutputStream out = new ByteArrayOutputStream();  
99 writeJava(out);  
100 System.out.println(Arrays.toString(out.toByteArray()));
```

 SourcesSinksExamples.java

[74, 97, 118, 97, 10]

## Notizen



# Inhalt

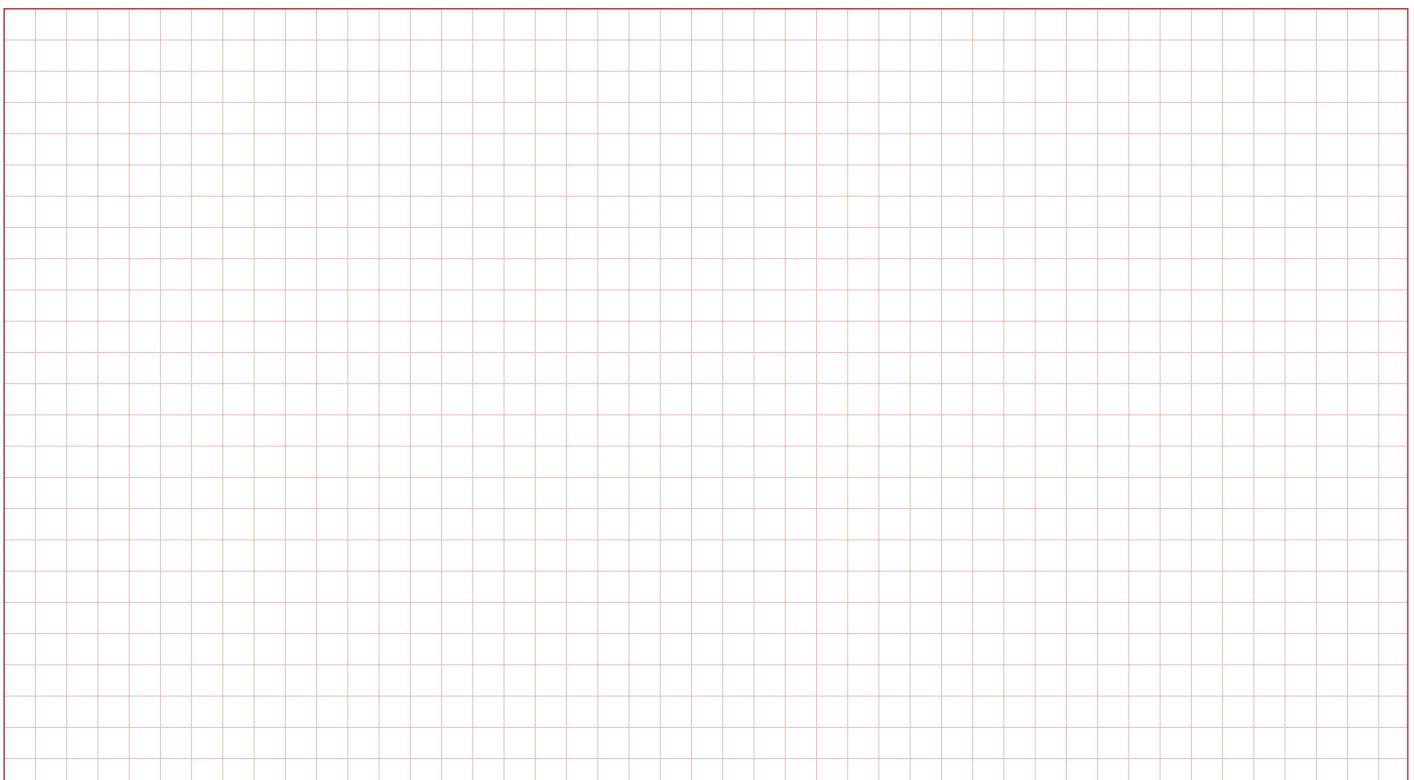
# Byteströme

## Notizen

# Übersicht

Stream	Ziel
<a href="#">↗</a> <code>System.in/out/err</code>	Standardein-/ausgabe und Fehlerstrom
<code>FileIn/OutputStream</code>	Datei
<code>ByteArrayIn/OutputStream</code>	<b>byte</b> -Array
<code>PipedIn/OutputStream</code>	PipedOut/InputStream
<a href="#">↗</a> <code>Socket.getIn/OutputStream</code>	Netzwerkverbindung
...	...

## Notizen



# Inhalt

## Byteströme

### Filter

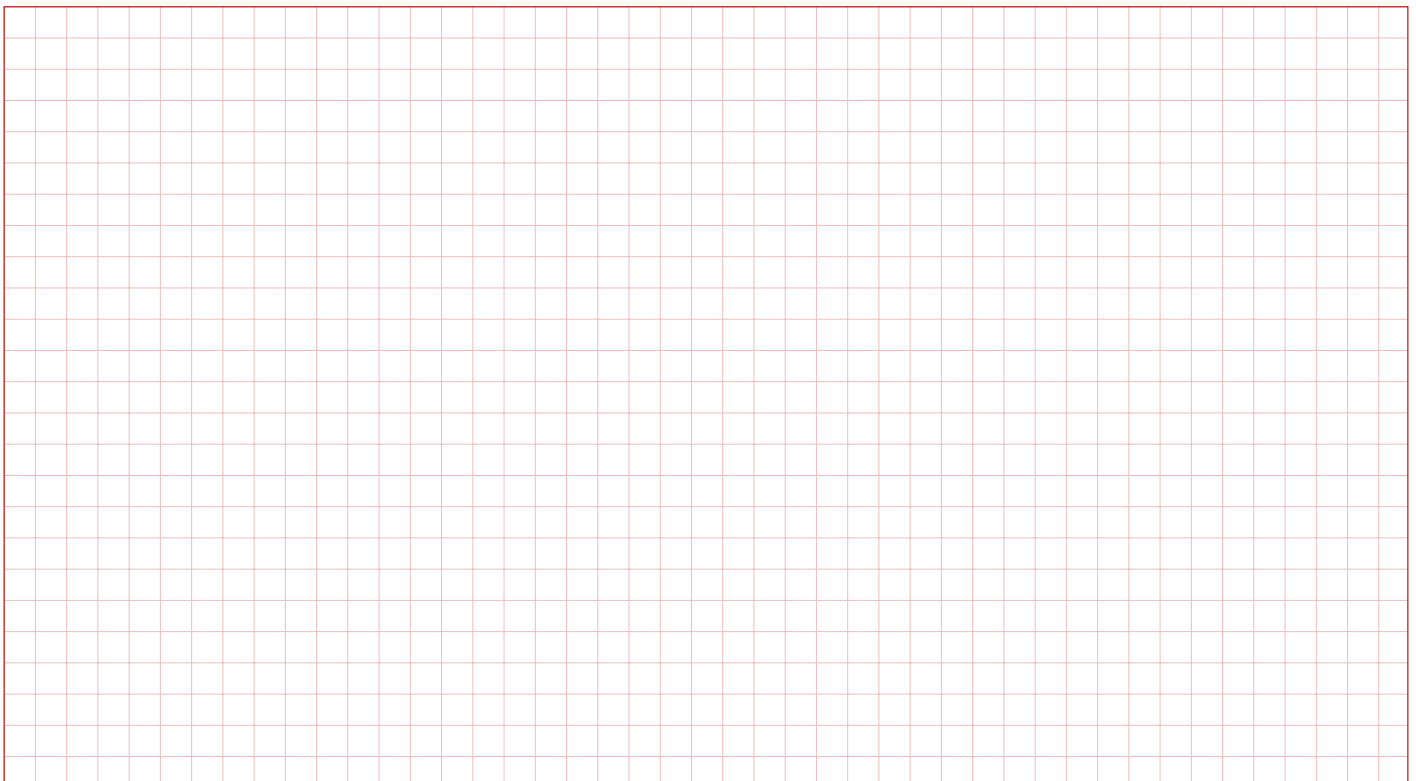
- Motivation

- Input/OutputFilterStream

- Beispiel: DataIn/OutputStream

- Hintereinanderschalten von Filtern

## Notizen



# Inhalt

## Byteströme

### Filter

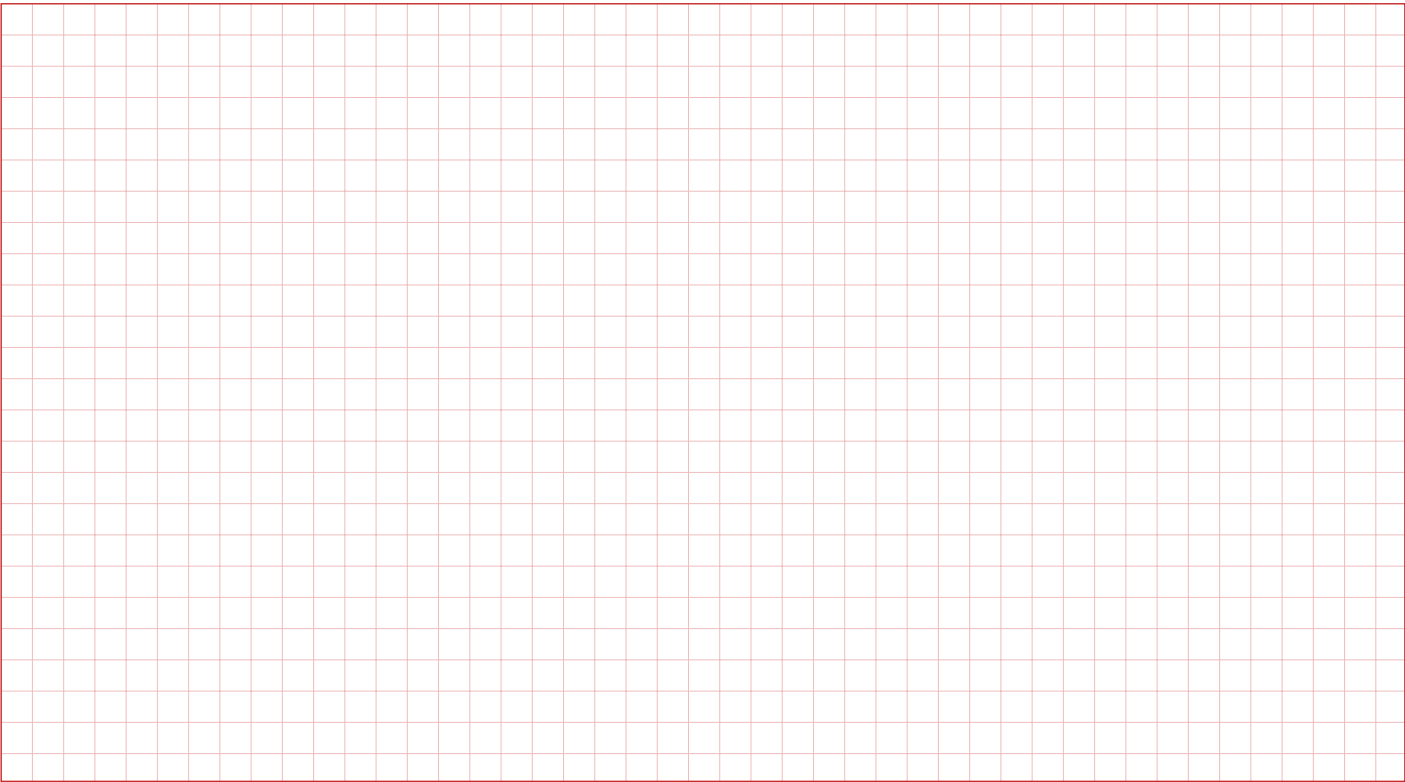
#### Motivation

Input/OutputFilterStream

Beispiel: DataIn/OutputStream

Hintereinanderschalten von Filtern

## Notizen



## Motivation

- ▶ Linux-Tool gzip
  - ▶ Liest von der **Standardeingabe**
  - ▶ **Komprimiert** mit gzip-Algorithmus
  - ▶ **Schreibt** komprimierte Daten auf **Standardausgabe**
- ▶ **Beispiel** auf der Linux-Kommandozeile

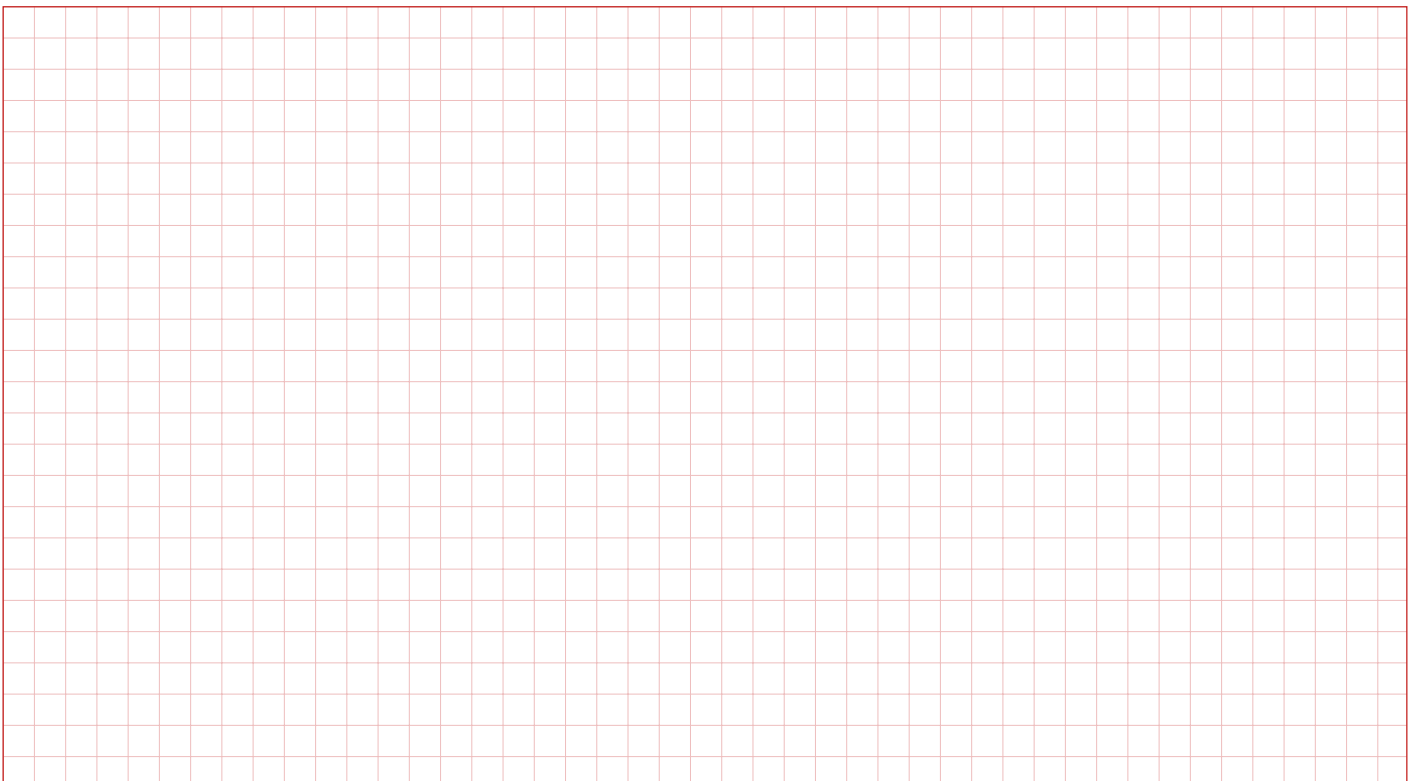
```
$ gzip -k lorem-ipsuM.txt
$ du -hs lorem-ipsuM.txt lorem-ipsuM.txt.gz
16K    lorem-ipsuM.txt
4.0K   lorem-ipsuM.txt.gz
```

- ▶ Prinzip



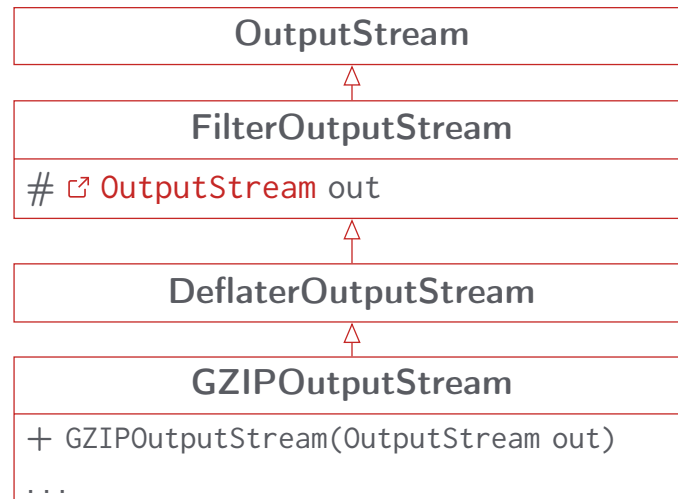
- ▶ gzip **transformiert** Daten
- ▶ Prinzip in Java: **Filter**
- ▶ **Ziel**: Java-gzip Implementierung

## Notizen





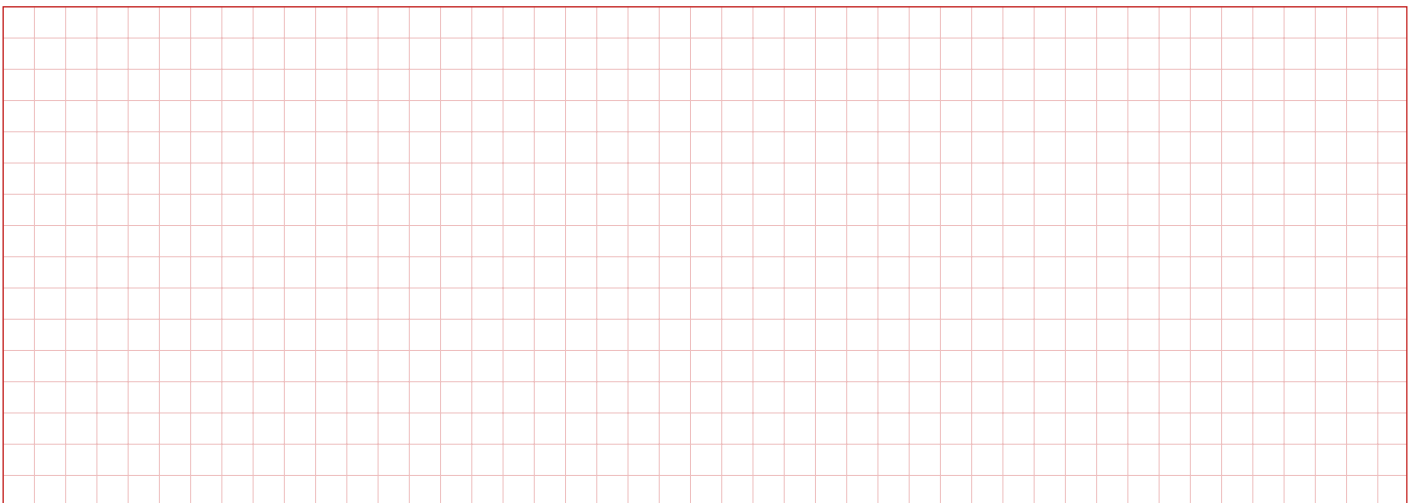
## 🔗 GZIPOutputStream



- ▶ 🔗 FilterOutputStream — **Ausgabefilter**, filtert Daten vor Schreiben nach out
- ▶ 🔗 DeflaterOutputStream — allgemeiner **Kompressions-Ausgabefilter**, komprimiert Daten vor Schreiben
- ▶ 🔗 GZIPOutputStream — Kompressions-Ausgabefilter basierend auf **gzip-Algorithmus**

## Notizen

- FilterOutputStream *filtert/modifiziert* die geschriebenen Daten und schreibt sie dann in out.
- DeflaterOutputStream allgemeine Schnittstelle für Filter, die Daten *komprimieren*.
- GZIPOutputStream komprimiert Daten nach gzip-Algorithmus und schreibt die komprimierten Daten in out.



## Java-gzip

- Komprimiert Datei mit Pfad args[0] nach args[0]+".gz"

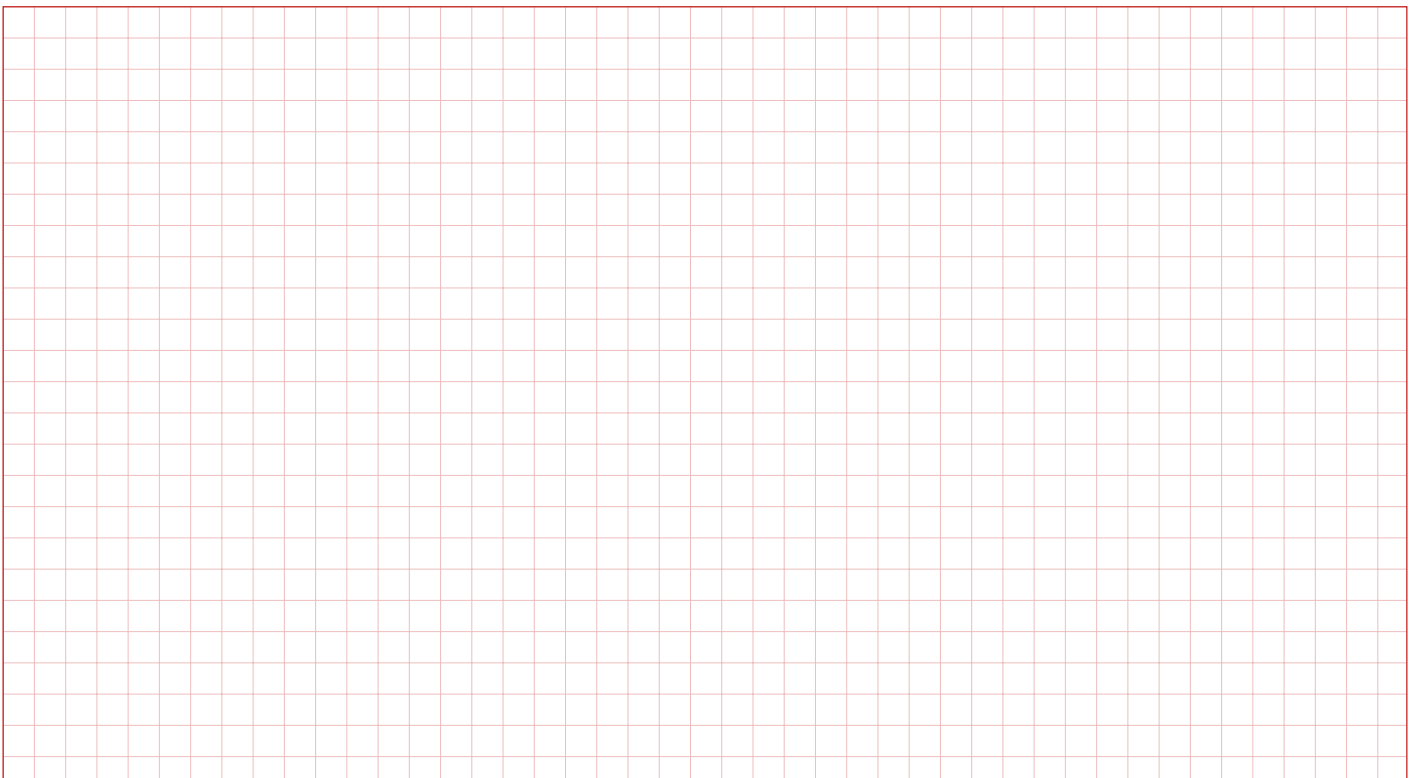
```
19 public static void main(String args[])
20     throws IOException {
21
22     var in = new FileInputStream(args[0]);
23     var out = new FileOutputStream(args[0] + ".gz");
24     var gzipFilter = new GZIPOutputStream(out);
25
26     in.transferTo(gzipFilter);
27
28     in.close();
29     gzipFilter.close(); // closes out as well
30 }
```

ByteStreamFilterExamples.java

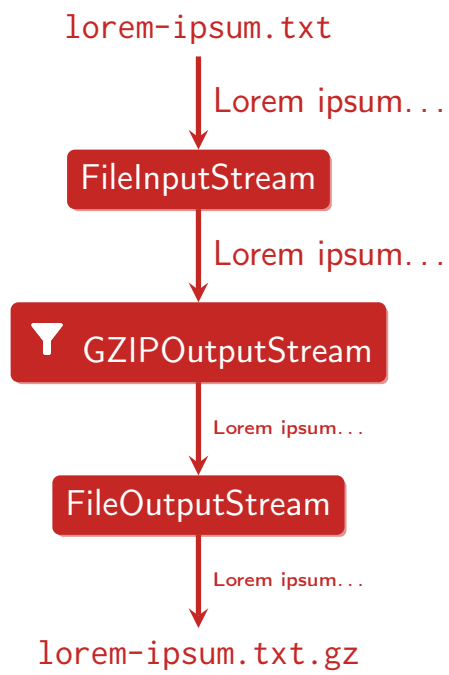
- Aufruf

```
$ gradle runGzipExample --args="lorem-ipsum.txt"
```

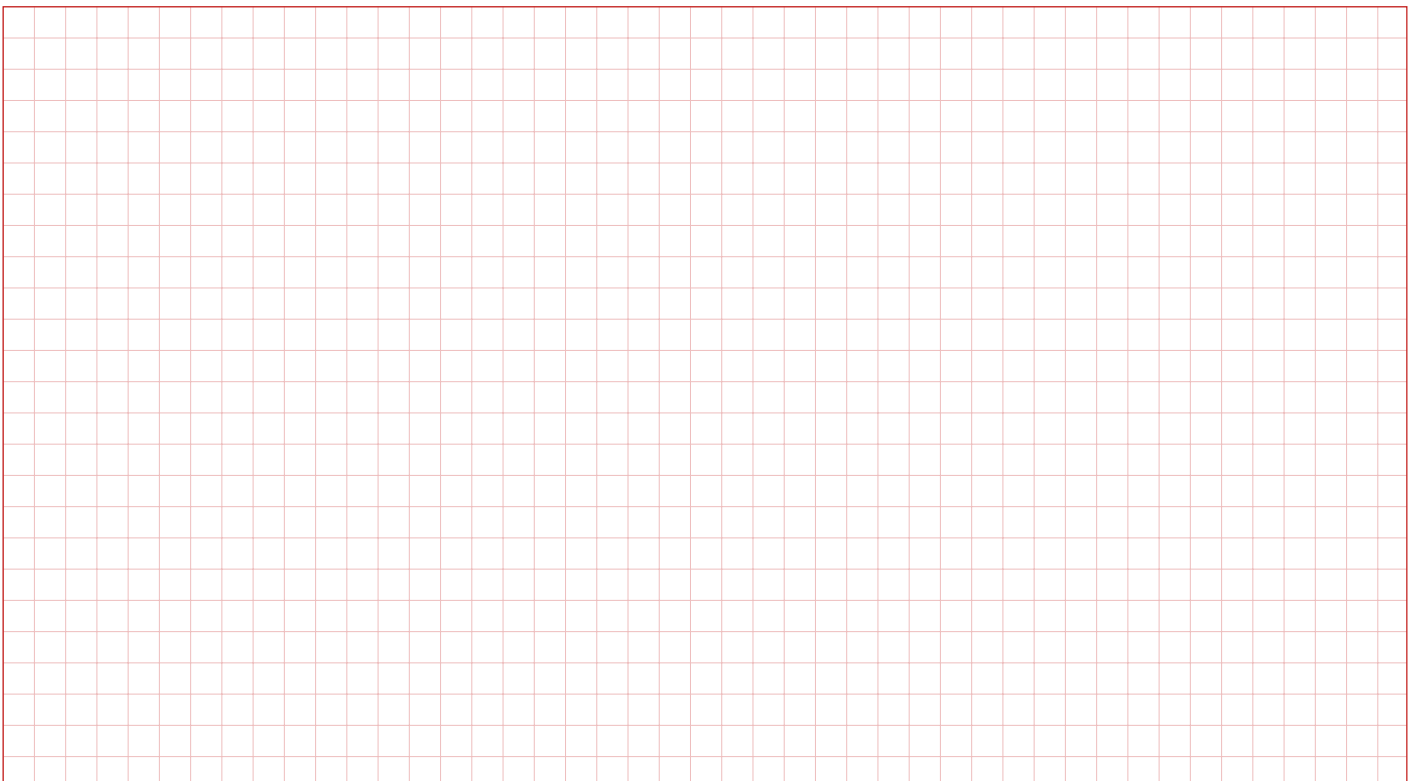
## Notizen



## Java-gzip — Veranschaulichung



### Notizen



# Inhalt

## Byteströme

### Filter

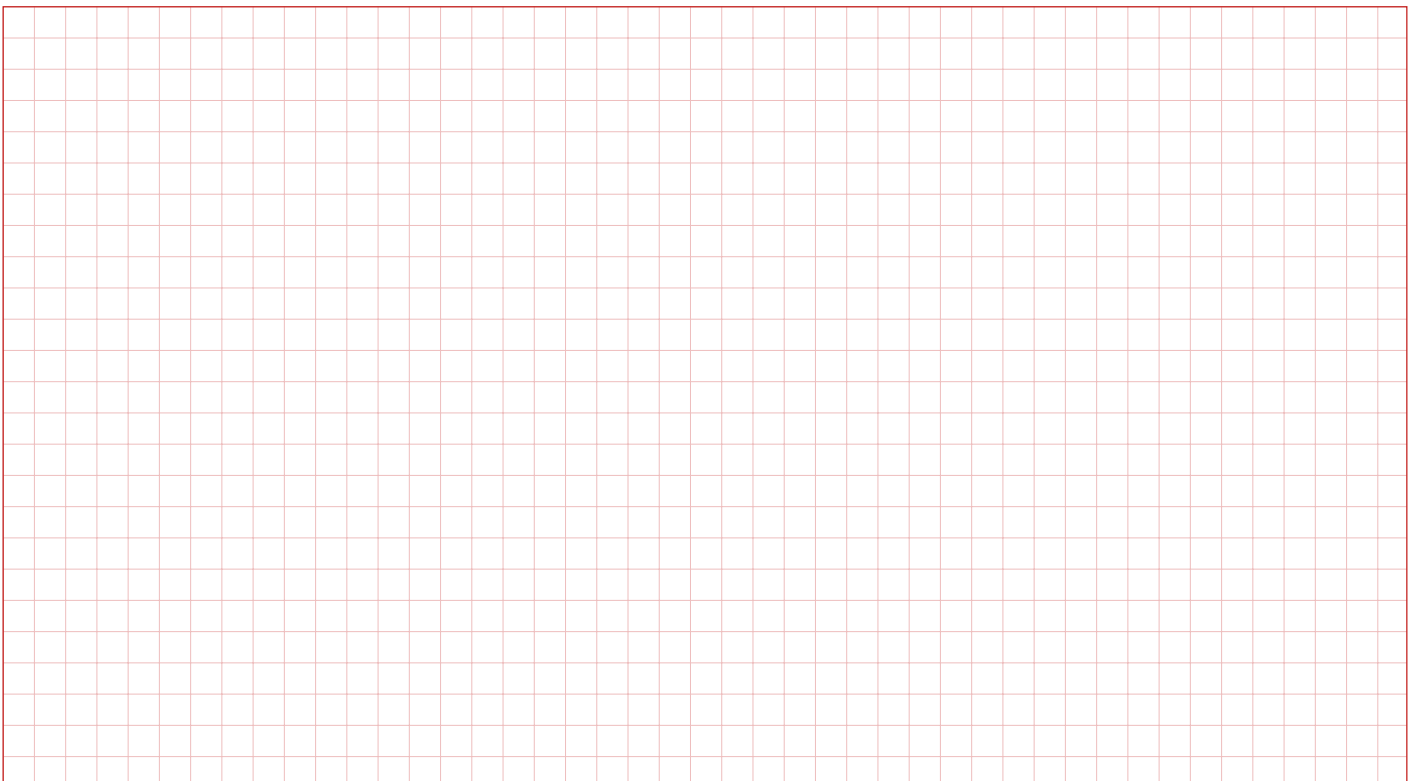
Motivation

Input/OutputFilterStream

Beispiel: DataIn/OutputStream

Hintereinanderschalten von Filtern

## Notizen



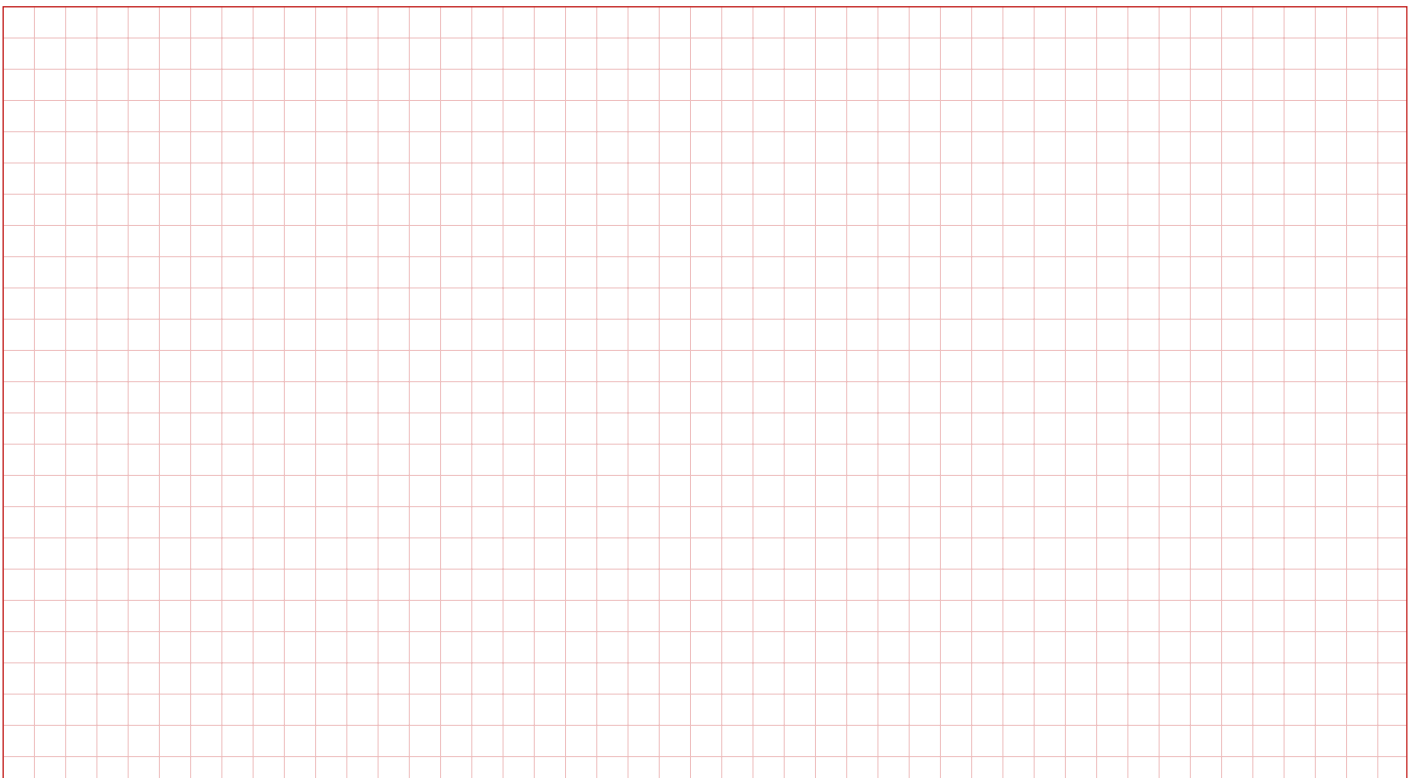
## Input/OutputStreamFilterStream

- ▶ Filter können auf **Ein- und Ausgabeströmen** agieren

FilterInputStream	FilterOutputStream
# in : InputStream	# out : OutputStream

- ▶ [↗](#) FilterOutputStream
  - ▶ Daten werden **transformiert...**
  - ▶ ...und dann in out **geschrieben**
- ▶ [↗](#) FilterInputStream
  - ▶ Daten werden aus in **gelesen...**
  - ▶ ...und dann **transformiert**

## Notizen

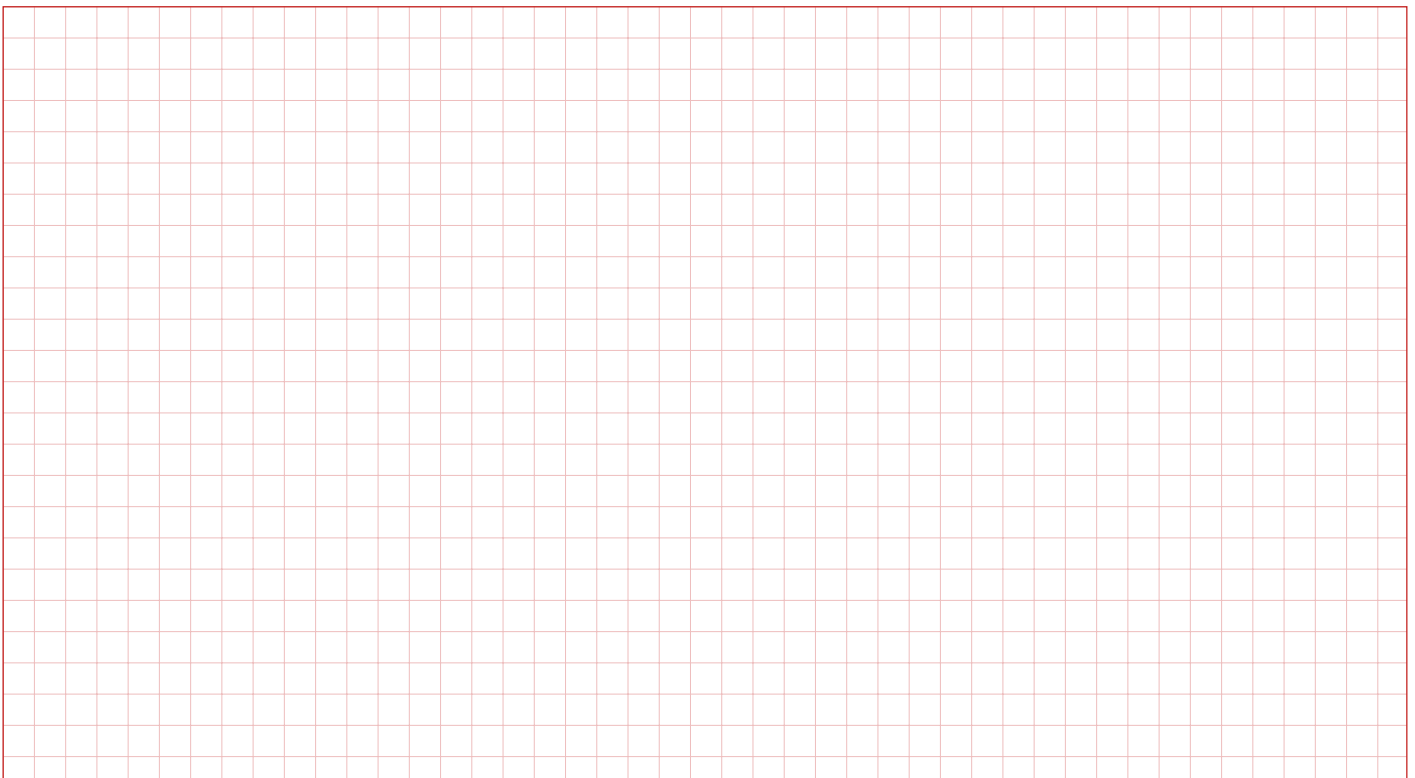


## Filter-Implementierungen

### ► „Echte“ Filter (verändern Daten)

Klasse	Funktion
CipherIn/OutputStream	Ent-/Verschlüsseln von Daten
DeflaterIn/OutputStream	Kompression von Daten
InflaterIn/OutputStream	Dekompression von Daten
GZIPIn/OutputStream	gzip-Kompression
ZipIn/OutputStream	ZIP-Kompression

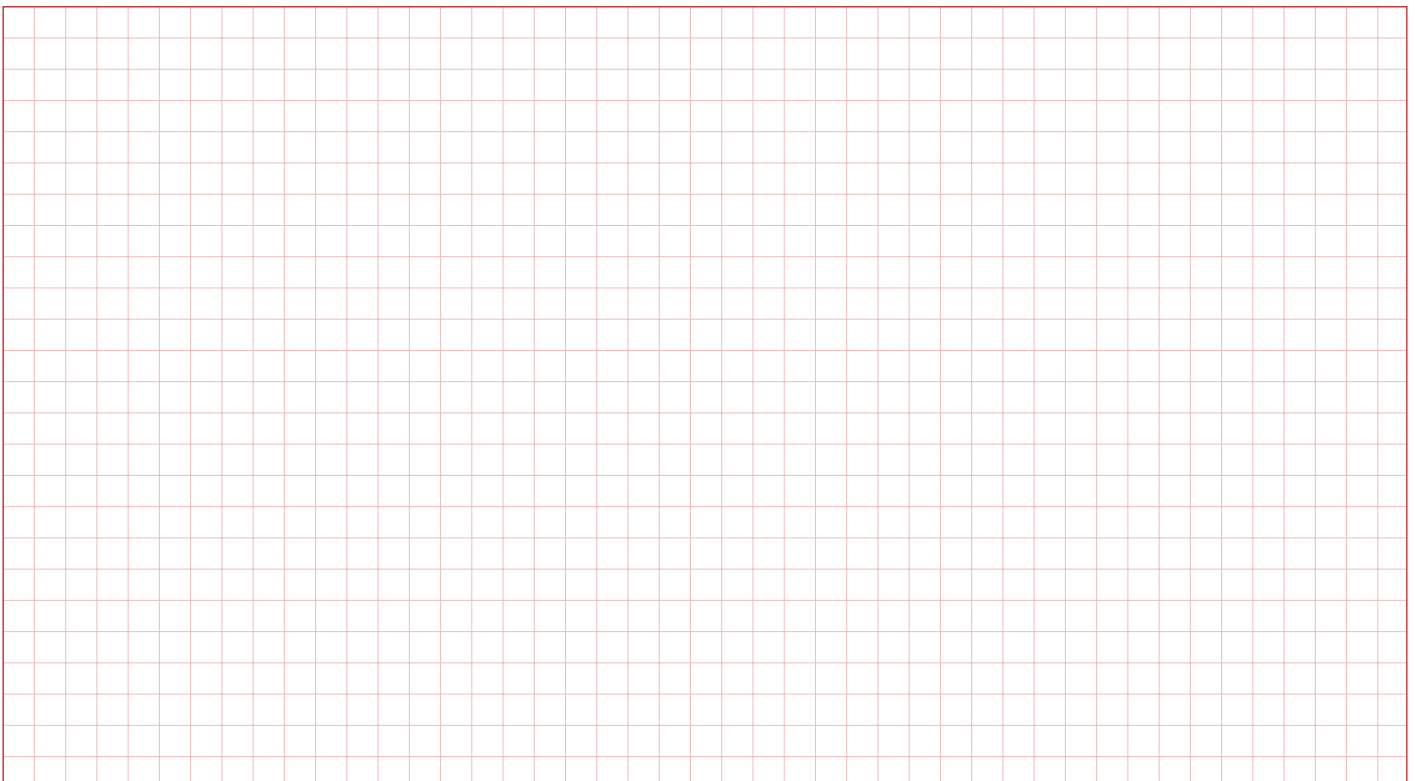
## Notizen





- ▶ **Filter** zum „Aufbohren“ von normalen Streams (Daten bleiben **unverändert**)
  - ▶ `BufferedInput/OutputStream`  
puffert Daten für effizienteres Lesen/Schreiben (+ `mark/reset` für [InputStream](#)s)
  - ▶ `CheckedInput/OutputStream`  
berechnet **Checksumme** (z.B. CRC32)
  - ▶ `DigestInput/OutputStream`  
berechnet **Digests** (z.B. MD5, SHA-1)
  - ▶ [LineNumberInputStream](#)  
zählt **Zeilennummer** mit
  - ▶ [PrintStream](#)  
mit Methoden zur **Textausgabe** von Java-Datentypen
  - ▶ `DataIn/OutputStream`  
mit Methoden zur **binären Ausgabe** von Java-Datentypen
  - ▶ Siehe [FilterInputStream](#) und [FilterOutputStream](#) für mehr

## Notizen



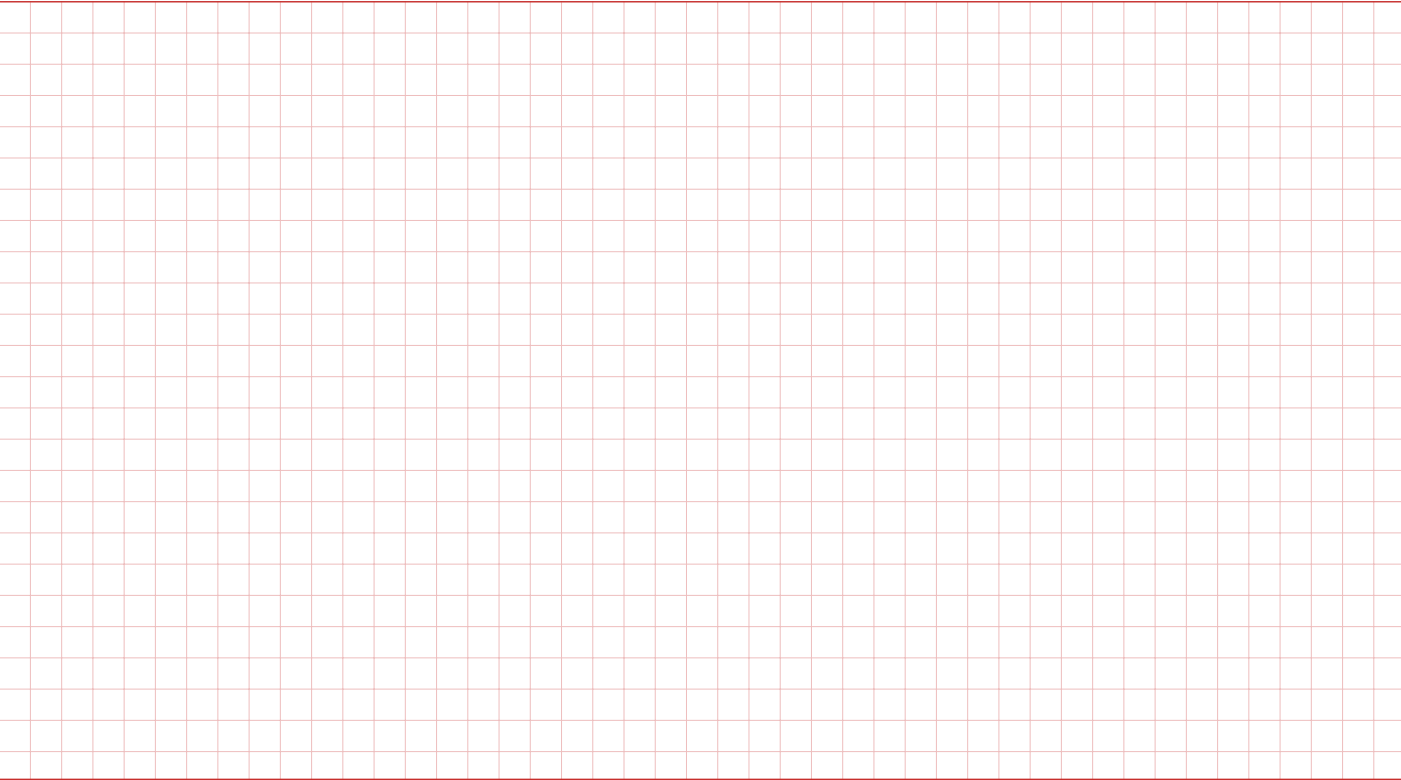
# Inhalt

## Byteströme

### Filter

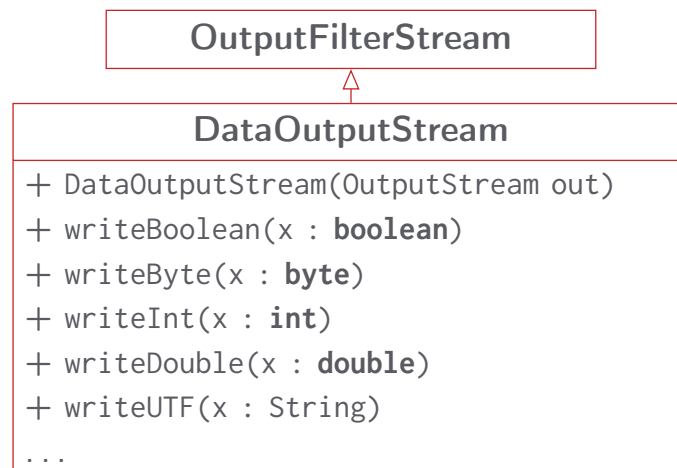
- Motivation
- Input/OutputFilterStream
- Beispiel: DataIn/OutputStream
- Hintereinanderschalten von Filtern

## Notizen





## DataOutputStream



### ► [DataOutputStream](#)

- `write*`-Methode für jeden **primitiven Typen**
- `writeUTF/Bytes/Chars`-Methoden für [String](#)
- Konvertiert und schreibt **Binärdaten** (nicht „human-readable“)

## Notizen

A large grid area for taking notes, consisting of a 20x20 grid of squares.


## DataOutputStream

- `writeData` schreibt ein paar Daten in [DataOutputStream](#)

```
35 public static void writeData(DataOutputStream out)
36     throws IOException {
37     out.writeInt(42);
38     out.writeDouble(Math.PI);
39     out.writeBoolean(true);
40     out.writeUTF("Java!");
41 }
```

ByteStreamFilterExamples.java

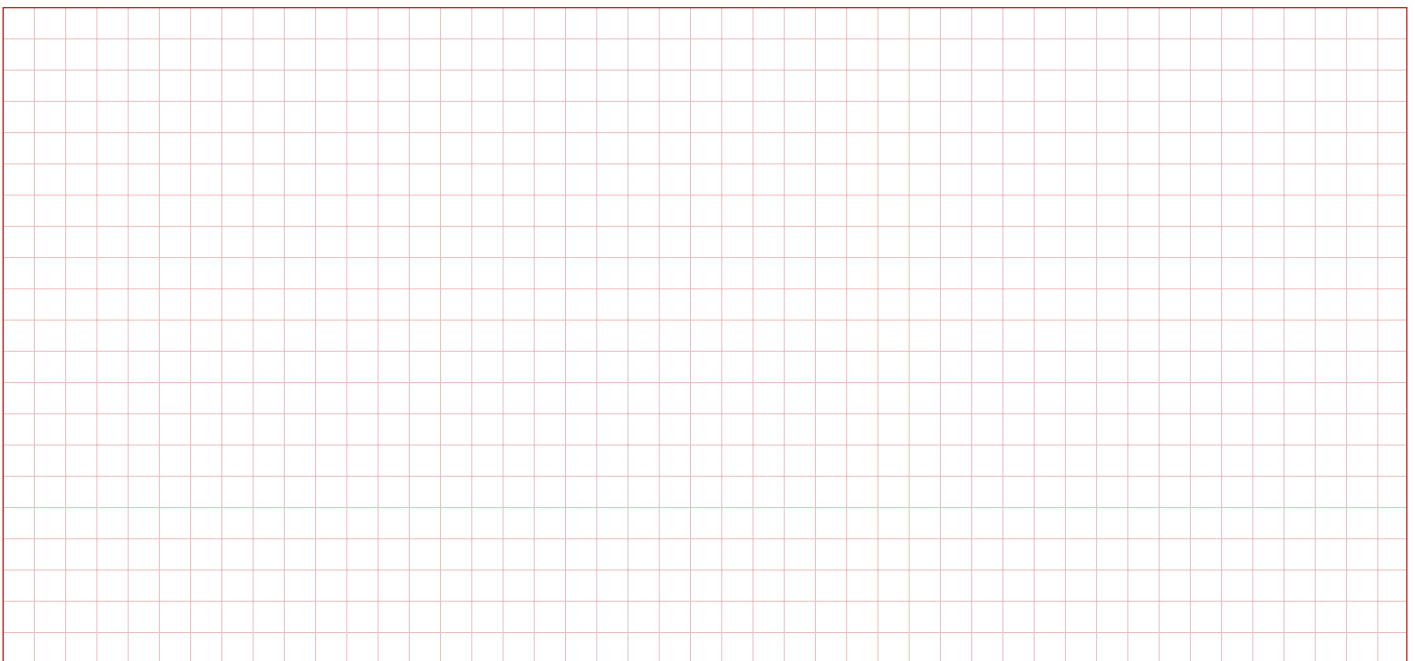
- Aufruf

```
59  runDataOutputStreamExample
60 var fileOut = new FileOutputStream("data.bin");
61 var dataOut = new DataOutputStream(fileOut);
62 writeData(dataOut);
```

ByteStreamFilterExamples.java

## Notizen

- `hexdump` ist ein Kommandozeilen-Tool das den Inhalt einer Datei im Hexadezimalformat ausgibt.



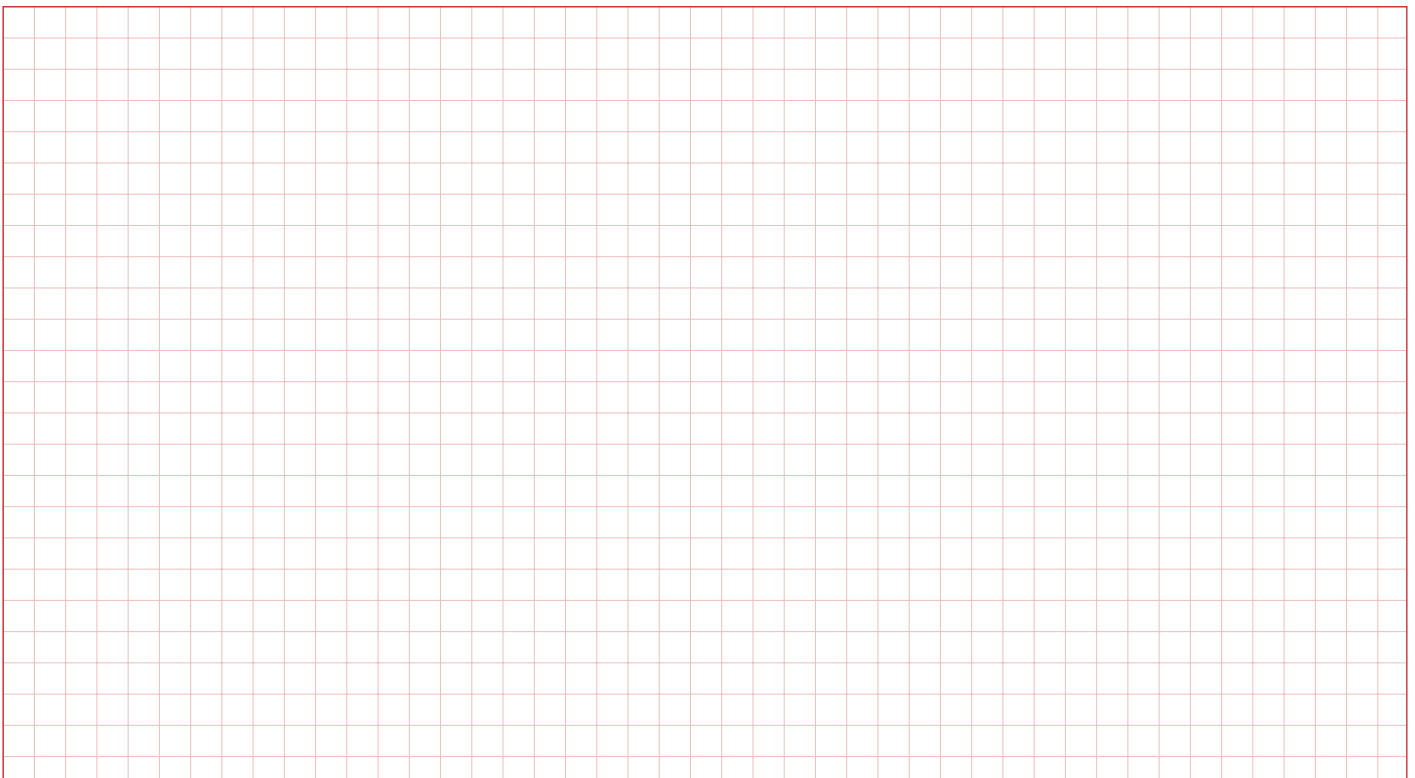
## DataOutputStream

### ► Resultat (mit hexdump)

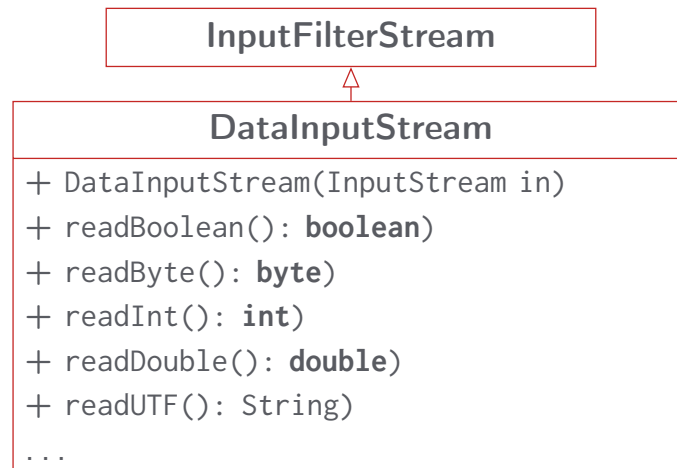
```
$ hexdump data.bin
00 00 00 2a 40 09 21 fb 54 44 2d 18 01 00 05 4a
61 76 61 21
```

- **(int)** 42 → 00 00 00 2a
- **(double)** Math.PI → 40 09 21 fb 54 44 2d 18
- **(boolean)** true → 01
- **(String)** "Java!" → 05 4a 61 76 61 21 (05 für Länge, dann Zeichen)

## Notizen

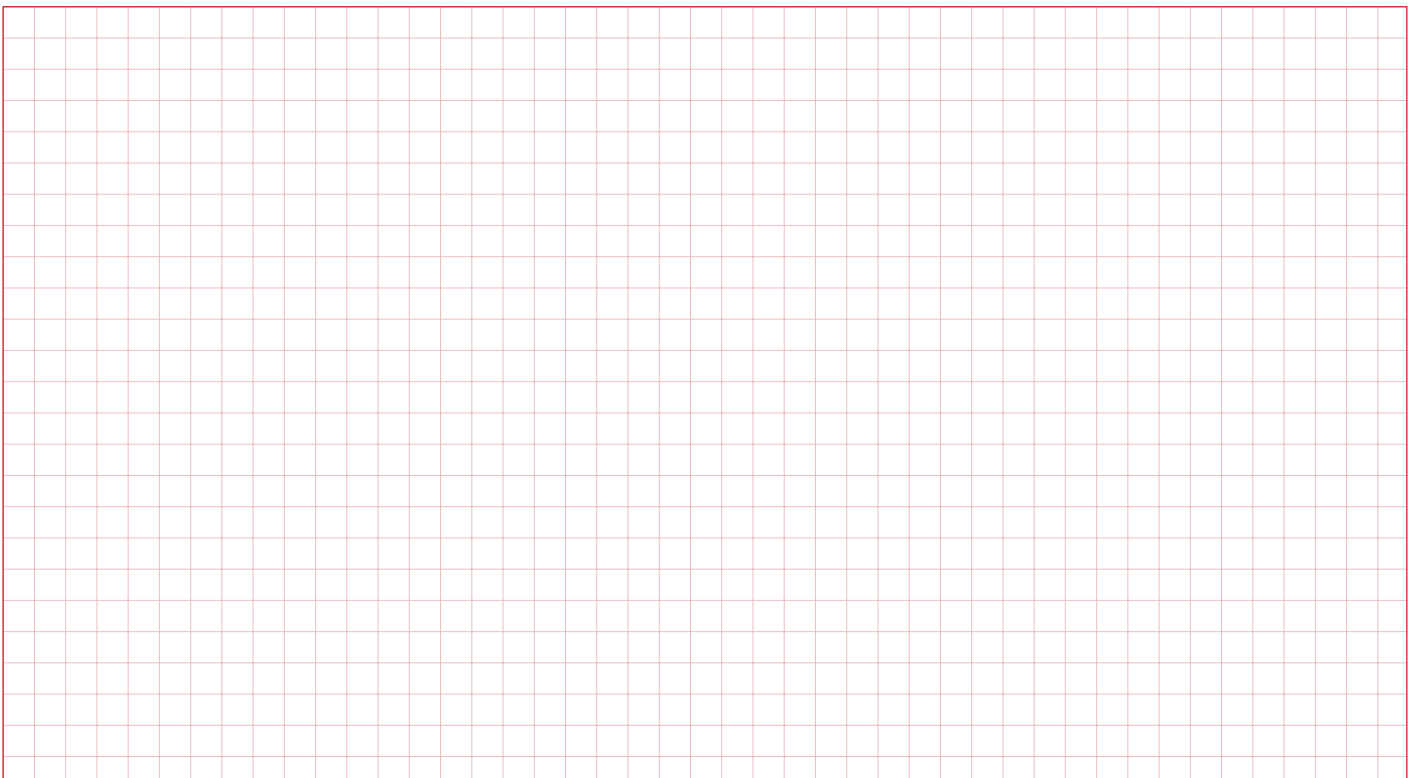


## DataInputStream



- ▶ [↗ DataInputStream](#)
  - ▶ Gegenstück zu [↗ DataOutputStream](#)
  - ▶ `read*`-Methode für jeden primitiven Typen
  - ▶ `readUTF/Bytes/Chars`-Methoden für [↗ String](#)
  - ▶ Liest und konvertiert Binärdaten in primitive Typen

## Notizen



## DataInputStream

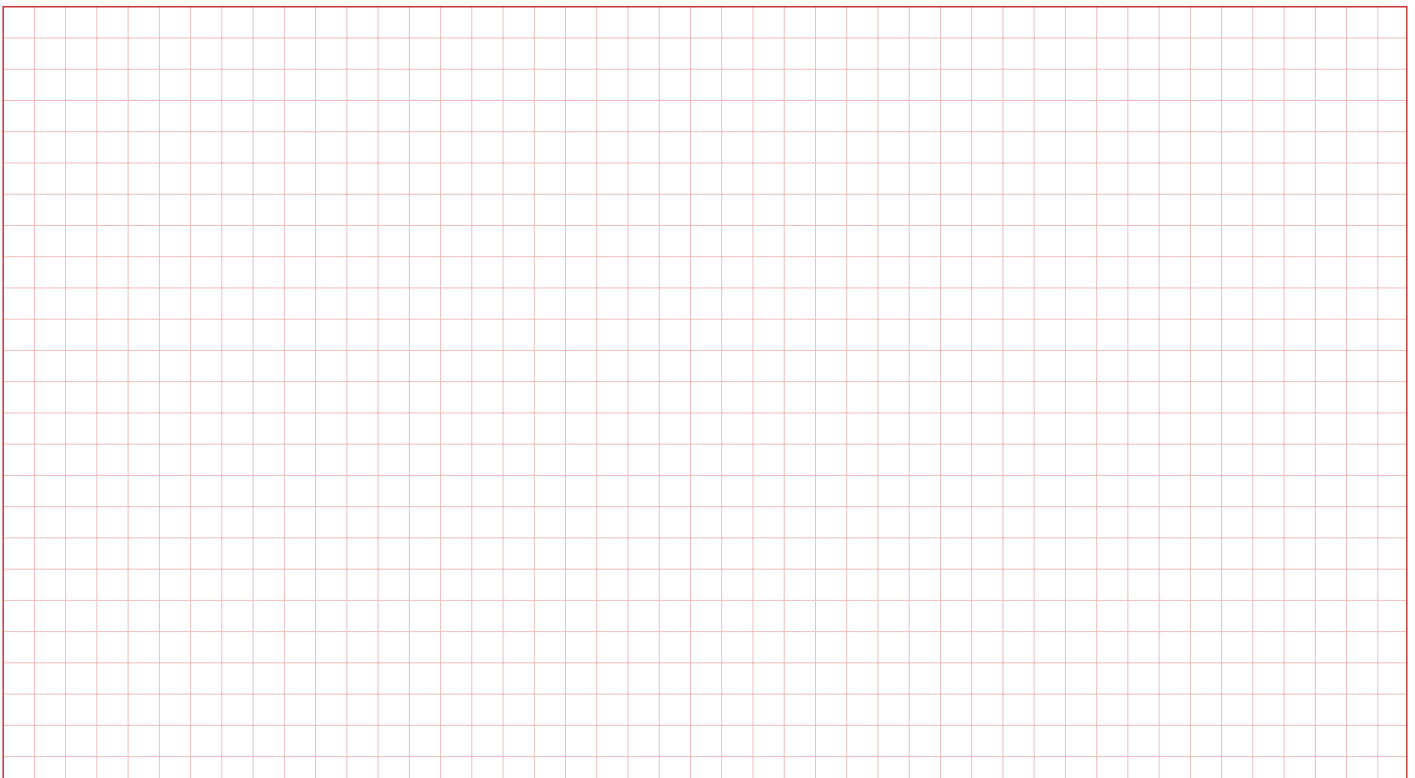
- readData liest die geschriebenen Daten aus [DataInputStream](#)

```
45 public static void readData(DataInputStream in)
46     throws IOException {
47     int i = in.readInt();
48     double pi = in.readDouble();
49     boolean b = in.readBoolean();
50     String s = in.readUTF();
51     out.printf("i=%d, pi=%f, b=%b, s=%s\n", i, pi, b, s);
52 }
53
54 }
```

[ByteStreamFilterExamples.java](#)


- **Achtung:** Lesereihenfolge **muss** Schreibreihenfolge entsprechen

## Notizen



## DataInputStream

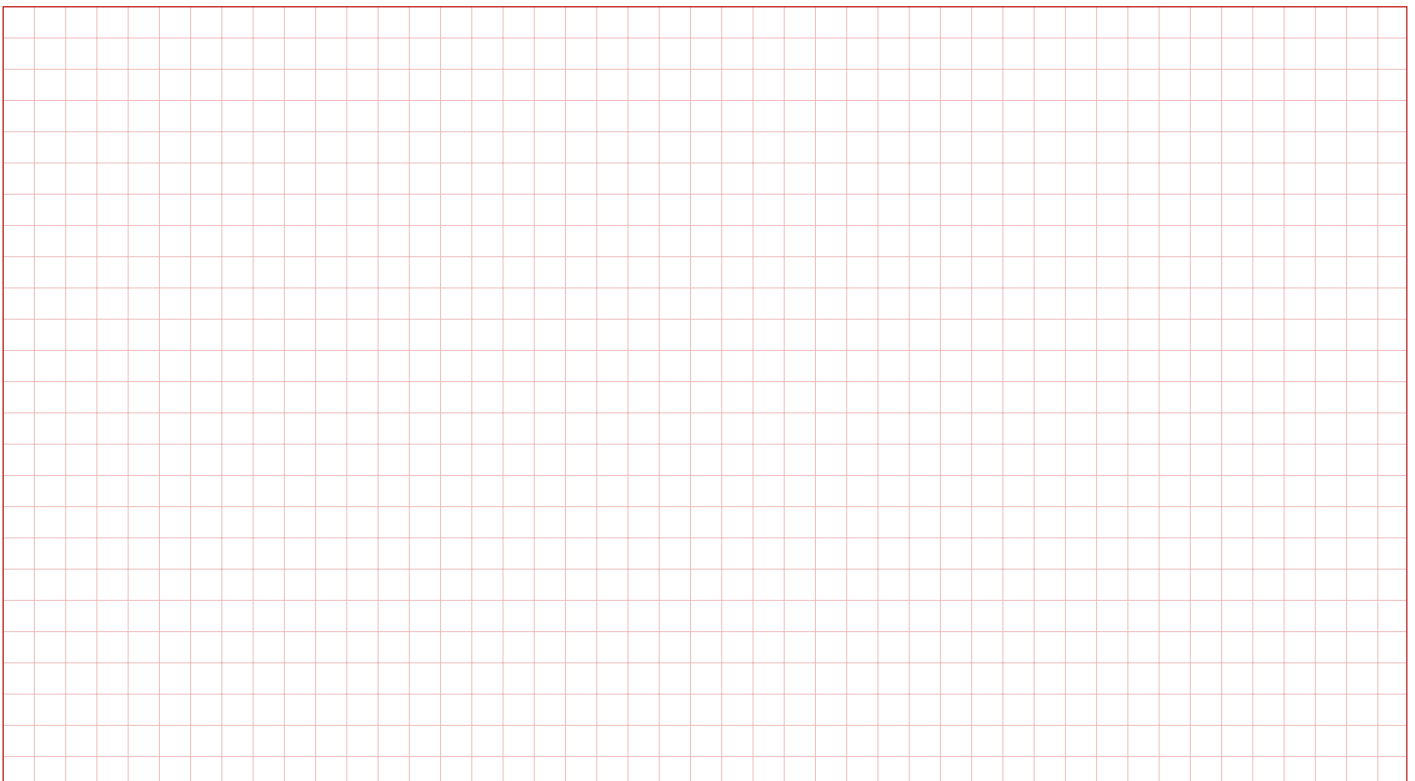
### ► Aufruf

```
68  runDataInputStreamExample  
69 var fileIn = new FileInputStream("data.bin");  
70 var dataIn = new DataInputStream(fileIn);  
71 readData(dataIn);
```

 ByteStreamFilterExamples.java

i=42, pi=3,141593, b=true, s=Java!

## Notizen

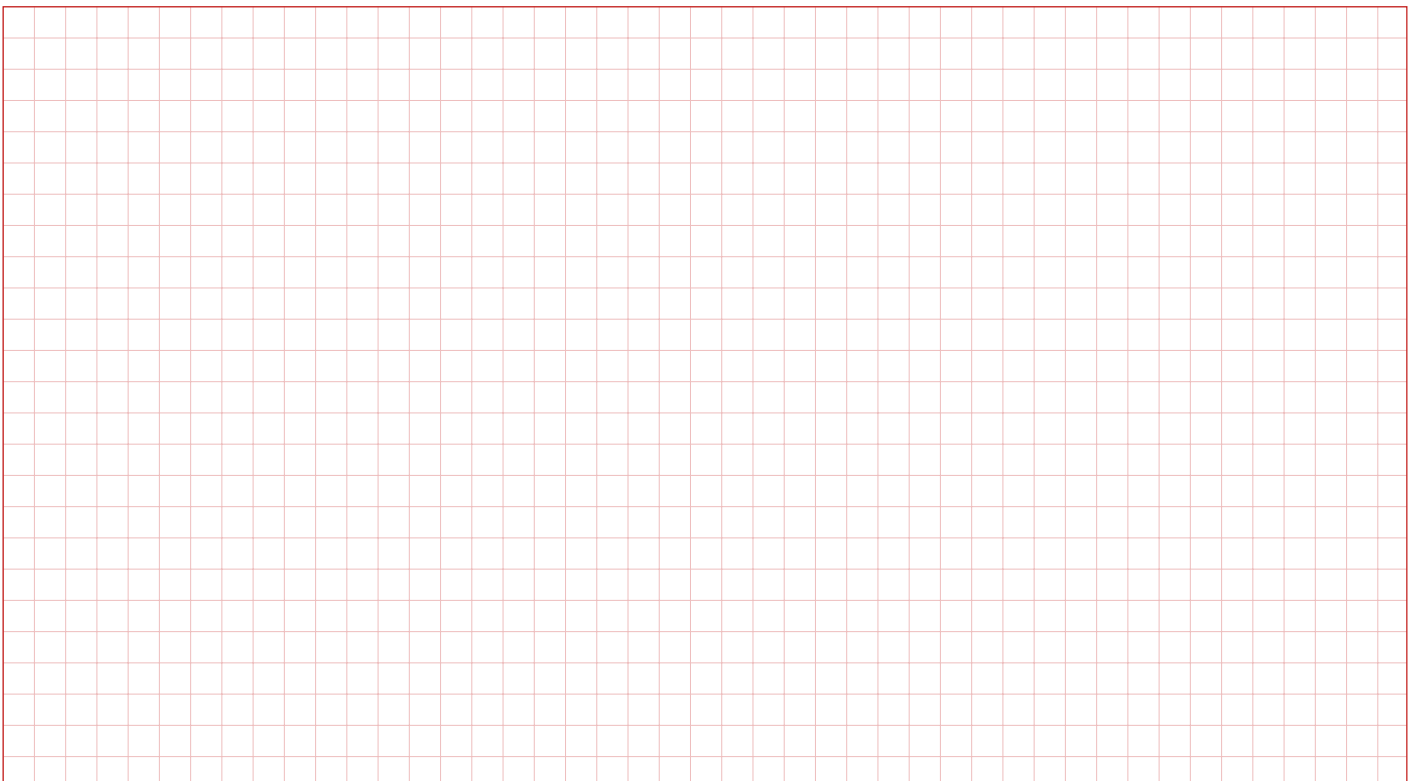


## Hinweise zu Binärdaten

- ▶ **In Beispiel:** Quelle/Senke war Datei
- ▶ **Allgemein** Input/OutputStream, z.B. auch **Netzwerk**
- ▶ Vor- und Nachteile von Binärdaten

Vorteile	Nachteile
geringer Speicherbedarf	nicht „human-readable“
zeit-/speichereffizient	nicht portabel (in Java schon)

## Notizen



# Inhalt

## Byteströme

### Filter

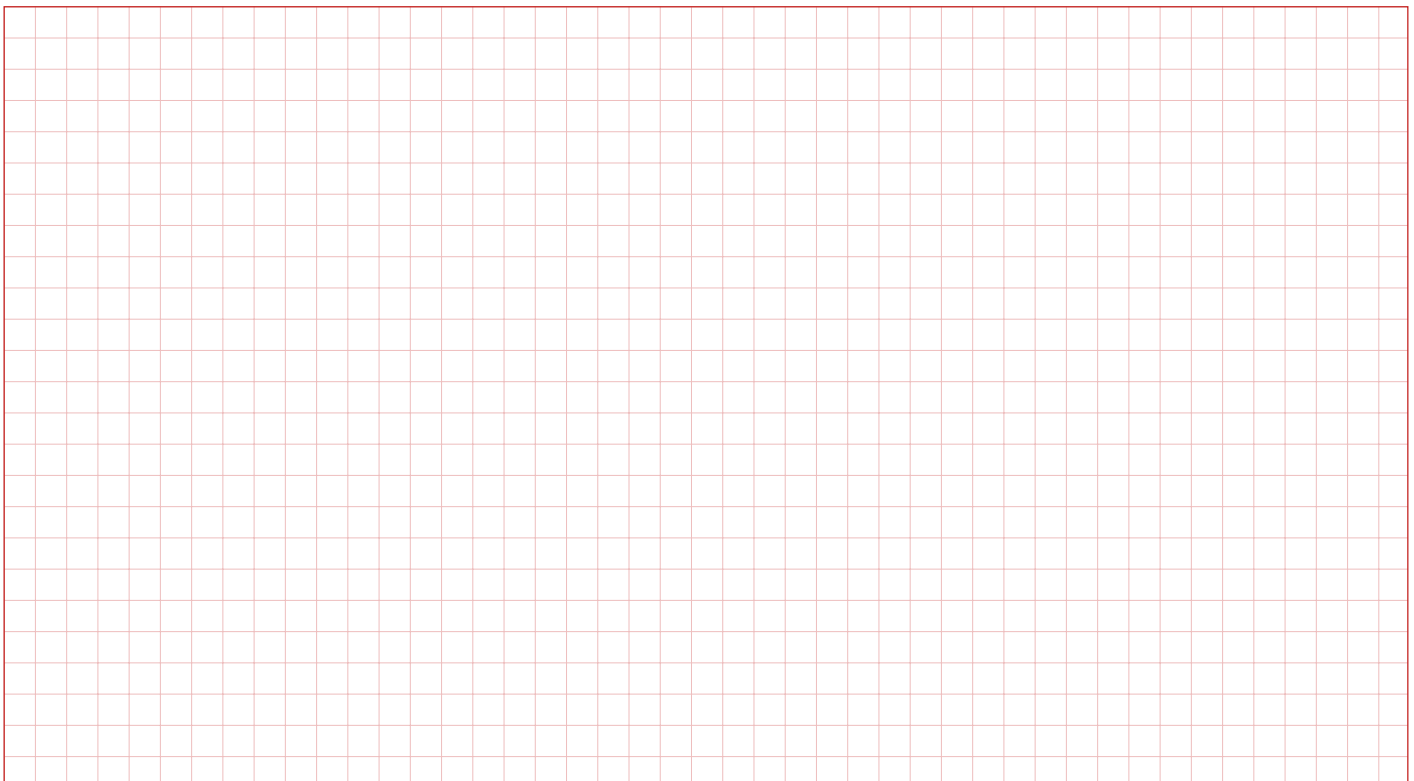
Motivation

Input/OutputFilterStream

Beispiel: DataIn/OutputStream

Hintereinanderschalten von Filtern

## Notizen






## Hintereinanderschalten

- Filter können **kombiniert** werden

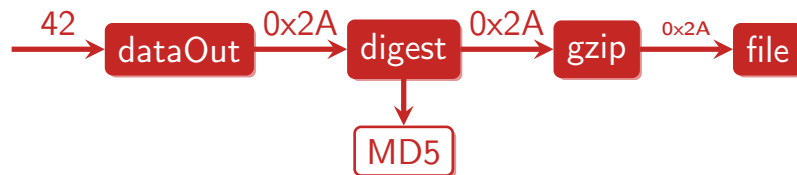


- Beispiel

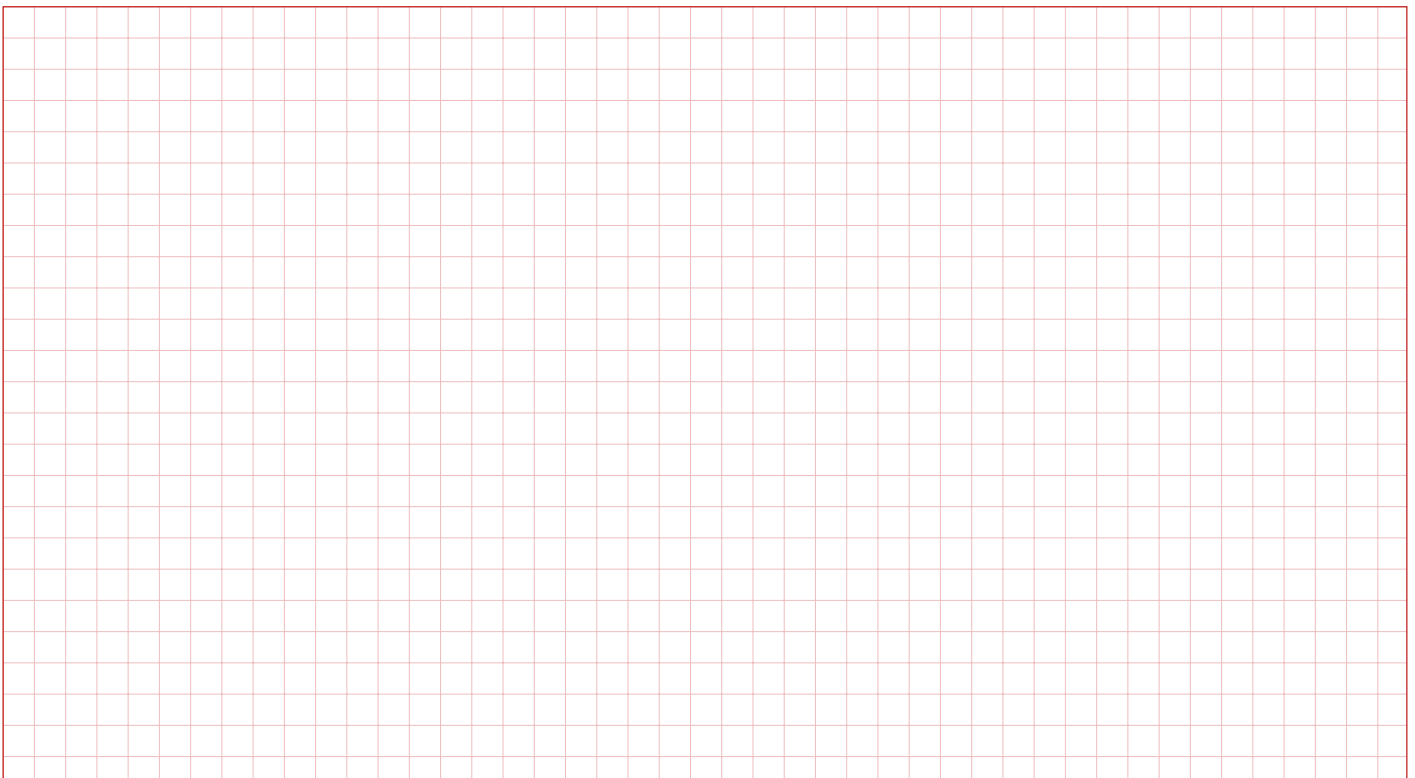
```
77  runCombinedFiltersExample  
78 var file = new FileOutputStream("data.bin");  
79 var gzip = new GZIPOutputStream(file);  
80 var digest = new DigestOutputStream(gzip,  
81     MessageDigest.getInstance("MD5"));  
82 var dataOut = new DataOutputStream(digest);  
83 writeData(dataOut);
```

 ByteStreamFilterExamples.java

- Beispiel `dataOut.writeInt(42)`



## Notizen



# Inhalt

# Byteströme

## Beispiel

## Notizen

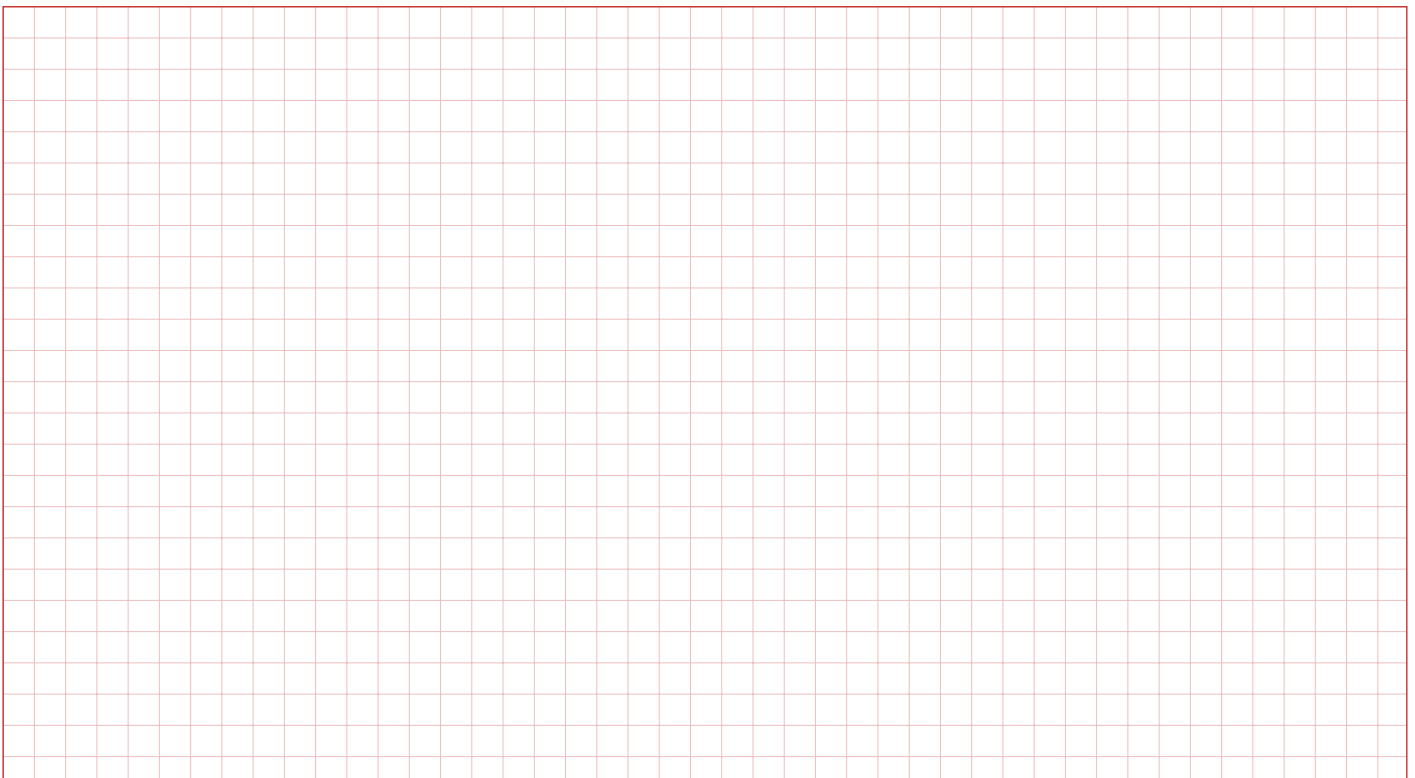
## Beispiel: Kopieren

- Kopieren von `InputStream` nach `OutputStream` mit Performancevergleich
- 1. Version: Kopieren „byte für byte“

```
25 public static long copyByteByByte(InputStream in,  
26     OutputStream out) throws IOException {  
27     long count = 0;  
28     int b;  
29     do{  
30         b = in.read();  
31         if (b >= 0){  
32             out.write(b);  
33             count++;  
34         }  
35     } while (b >= 0);  
37     return count;  
38 }
```


PerformanceExample.java

## Notizen



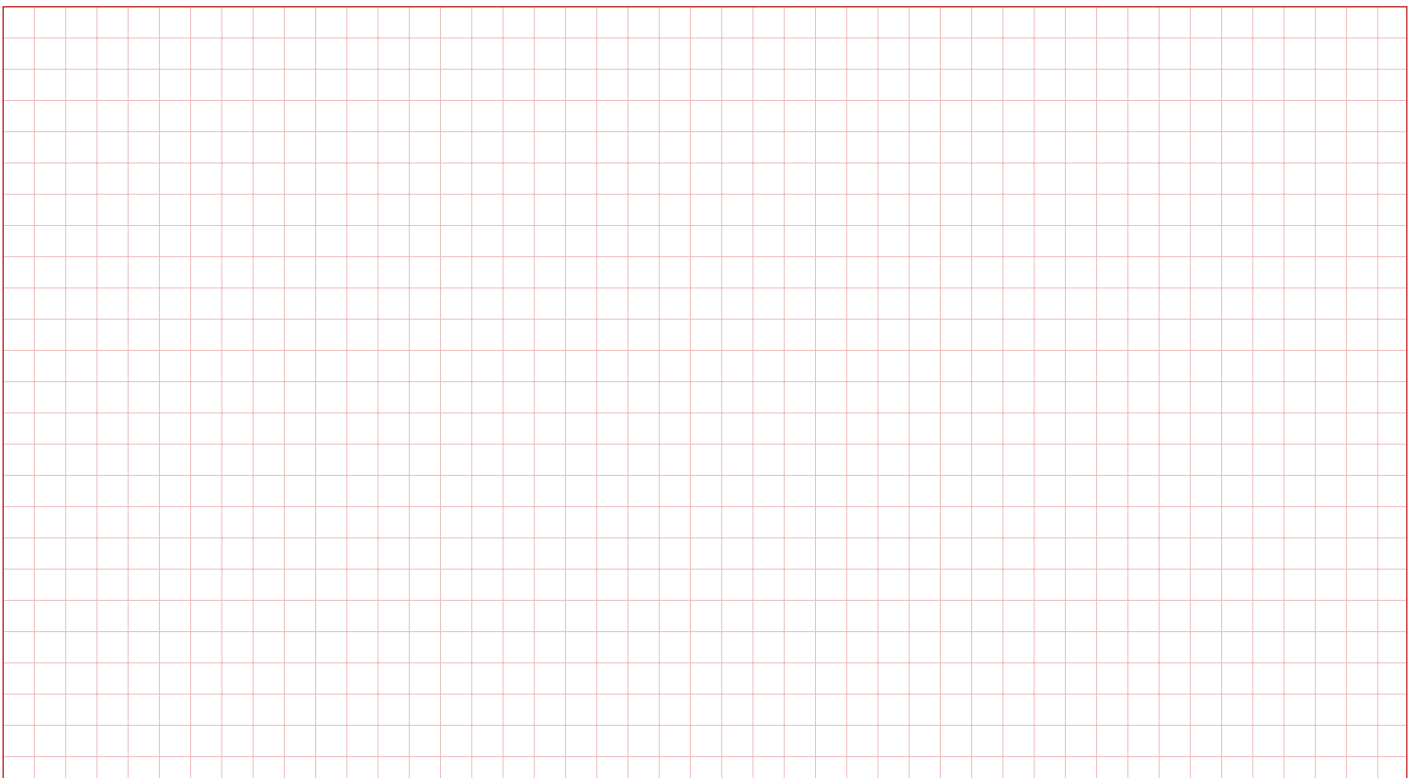
## Kopieren „byte für byte“

► Aufruf: Datei kopieren

```
43  runCopyFileByteByByte  
44 FileInputStream in = new FileInputStream("input-file");  
45 FileOutputStream out =  
46     new FileOutputStream("output-file");  
48 long startTime = System.currentTimeMillis();  
49 long count = copyByteByByte(in, out);  
50 long elapsed = System.currentTimeMillis() - startTime;  
52 // Gibt Infos zur Laufzeit und Datenrate aus  
53 printPerformanceInfo(count, elapsed);  
55 in.close();  
56 out.close();
```

 PerformanceExample.java

## Notizen



# Kopieren „byte für byte“

## ► Ergebnis (64 MB Datei)

```
Time: 300,332000 s  
Size: 64,000000 MB  
Rate: 0,213098 MB/s
```

## ► Sehr langsam!

## ► Gründe

- Sehr viele Methodenaufrufe (read, write)
- Sehr viele Hardwarezugriffe: immer nur ein Byte

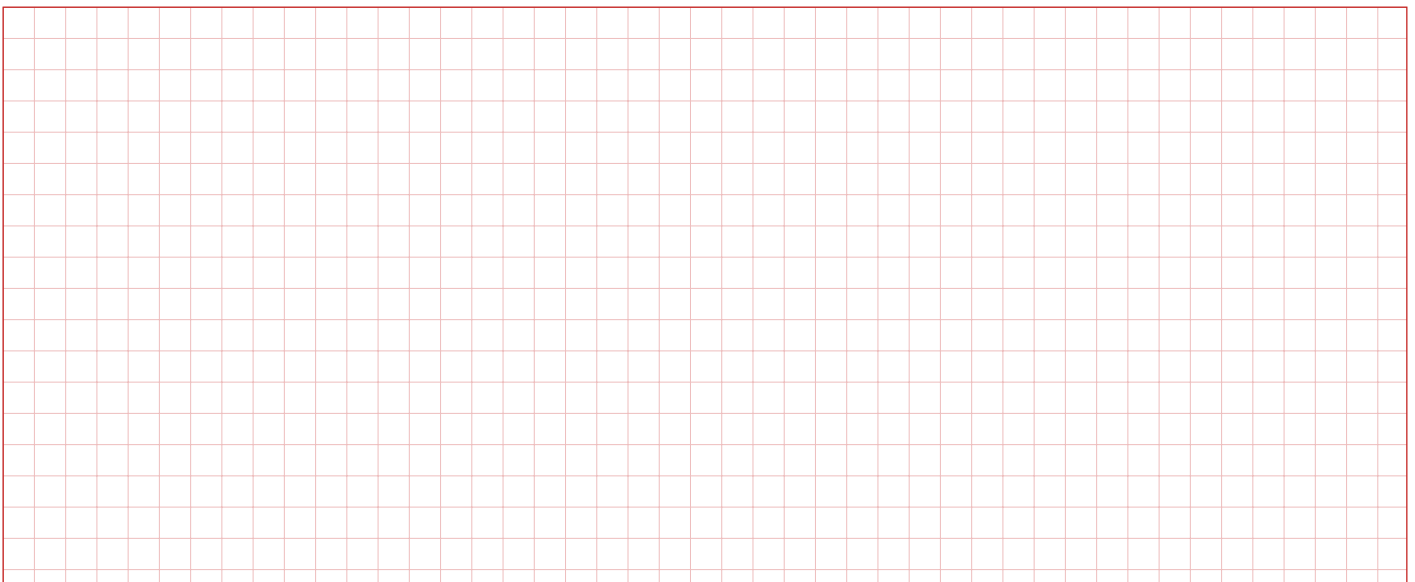
## ► Wie können wir die Performance verbessern?

## ► Idee

- Wir lesen **mehrere bytes** in **byte-Array**...
- ...und schreiben diese **in einem in den** [OutputStream](#)

## Notizen

- Dieser Test und alle folgenden Tests wurden auf einem Apple MacBook Pro 2019 unter macOS 10.15.5 ausgeführt. Auf anderer Hardware und anderen Betriebssystemen kann es zu anderen Ergebnissen kommen.



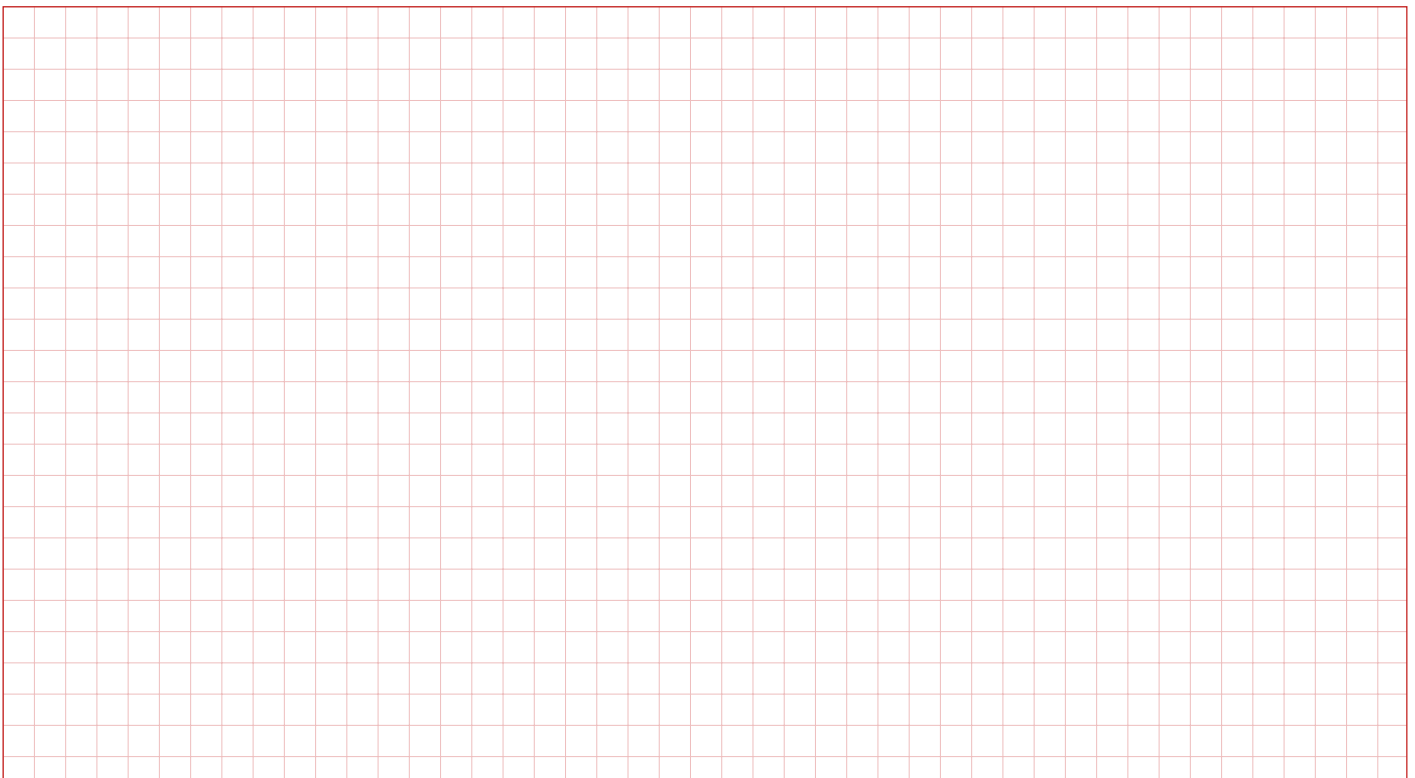
## Kopieren mit Puffer

### ► 2. Version: Kopieren mit Puffer (byte[])

```
62 public static long copyBuffer(InputStream in,
63     OutputStream out, byte[] buffer)
64     throws IOException {
65     long count = 0;
66     int readCount;
67
68     do {
69         readCount = in.read(buffer);
70         count += readCount;
71
72         if (readCount > 0)
73             out.write(buffer, 0, readCount);
74     } while (readCount > 0);
75     return count;
76 }
77 }
```

📄 PerformanceExample.java

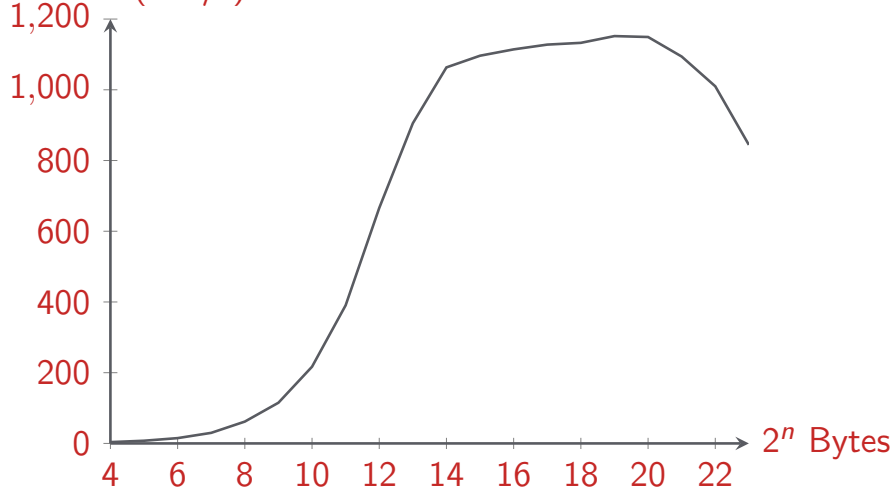
## Notizen



## Kopieren mit Puffer

- ▶ Dateigröße: 1 GB
- ▶ Aufruf mit `buffer.length=16,32,64 Bytes,...,8MB`

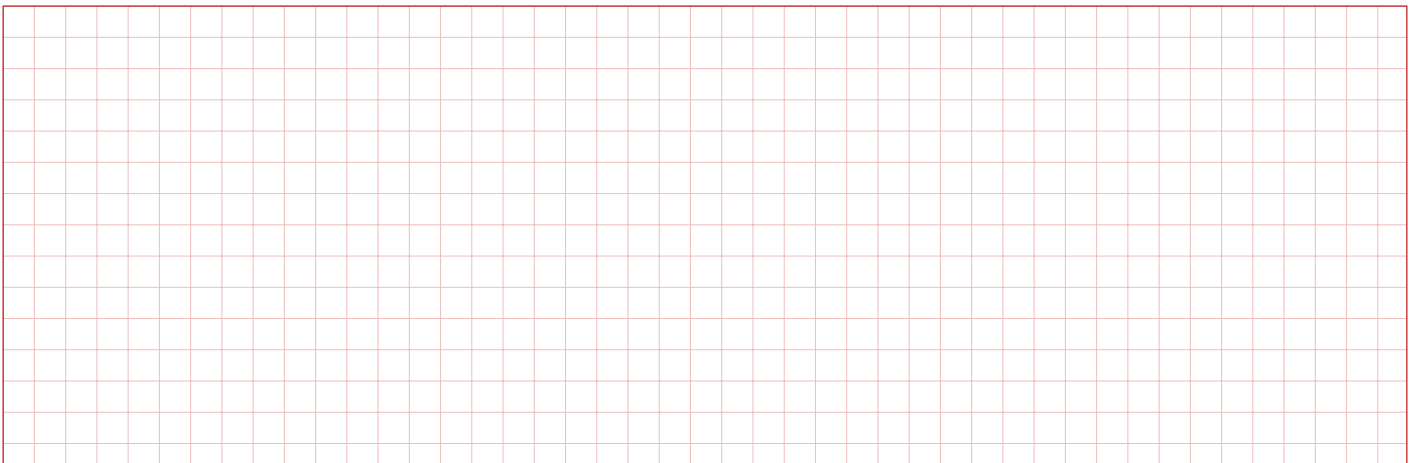
Datenrate (MB/s)



- ▶ Durchsatz steigt mit Puffergröße (ungefähr linear)...
- ▶ ...bis max. Durchsatz von 1150 MB/s (256 KB bis 1 MB)
- ▶ Effizienz sinkt dann sogar wieder ab (Grund: „caching“)

## Notizen

- Der Grund warum der Effizienz ab einer gewissen Puffergröße wieder absinkt ist der Zugriff auf den Cache der CPU: Häufig benutzte Hauptspeicherbereiche (wie unser Puffer) werden auf einen Cache-Speicher der CPU ausgelagert. Dieser ist besonders schnell, aber auch in der Größe sehr beschränkt. Wird diese Größe überschritten kommt es zu verhältnismäßig langsamen Zugriffen auf den Hauptspeicher („cache misses“). Je größer der Puffer, desto mehr Zugriffe auf den Hauptspeicher erfolgen und desto schlechter wird der Durchsatz.



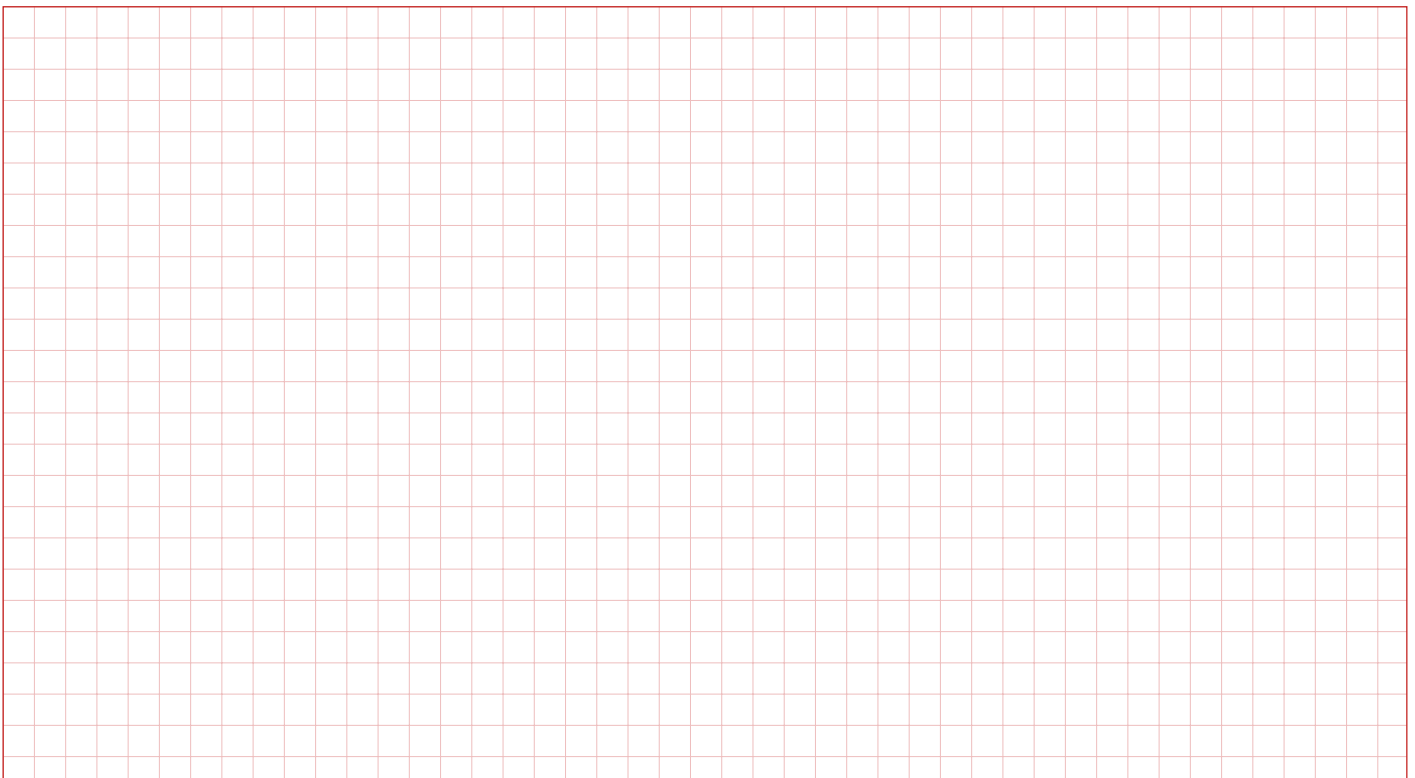
## Kopieren mit BufferedIn/OutputStream

- ▶ 3. Version: **BufferedIn/OutputStream**
- ▶ Implementieren **Pufferung**
- ▶ Gut wenn **Quelle/Senke nicht gepuffert** ist
- ▶ **Reduziert** Zugriffe auf „Hardware“
- ▶ **BufferedIn/OutputStream** „verpacken“ andere Streams

```
var in = new BufferedInputStream(  
    new FileInputStream("input-file"));  
var out = new BufferedOutputStream(  
    new FileOutputStream("output-file"));
```

- ▶ **Vergleich** mit „byte für byte“-Variante
  - ▶ **FileIn/OutputStream**: 0.213 MB/s
  - ▶ **BufferedIn/OutputStream**: 73 MB/s
- ▶ **Viel schneller**
- ▶ Immer noch **langsamer** als mit **eigenem Puffer**
- ▶ Grund: immer noch **viele Methodenaufrufe** von read/write

## Notizen





## Kopieren mit `InputStream.transferTo`

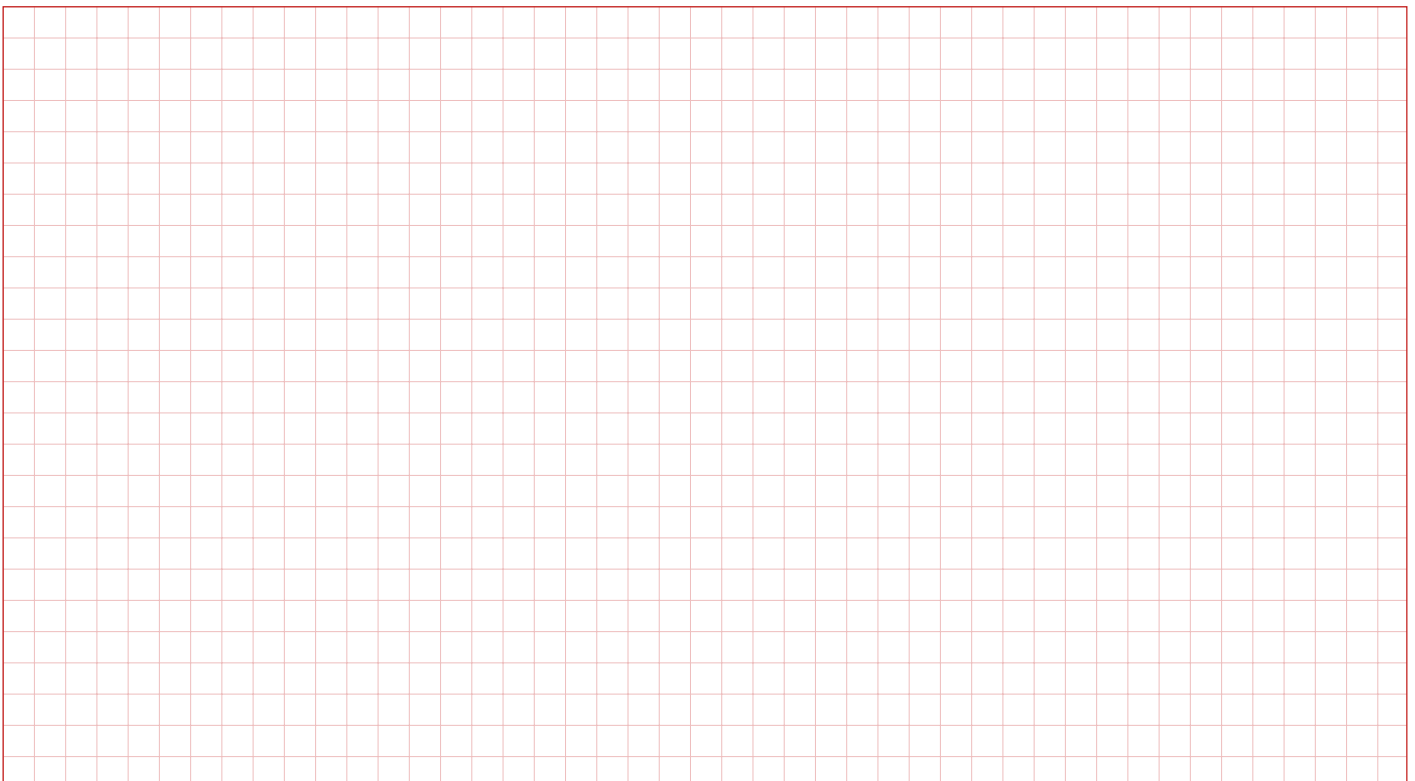
- ▶ 4. Version: `InputStream.transferTo(OutputStream out)`
  - ▶ Liest **alles** aus `InputStream`, schreibt **alles** in `out`
  - ▶ Rückgabe: Anzahl transferierter **bytes**
- ▶ Anwendung

```
120 public static long copyTransferTo(InputStream in,  
121     OutputStream out) throws IOException {  
122     return in.transferTo(out);  
123 }
```

📄 PerformanceExample.java

- ▶ **Transferrate**: 1047 MB/s
- ▶ Nutzt intern **Puffer**
- ▶ Vergleichbar mit **Variante 2** (eigener **byte**-Puffer)

## Notizen

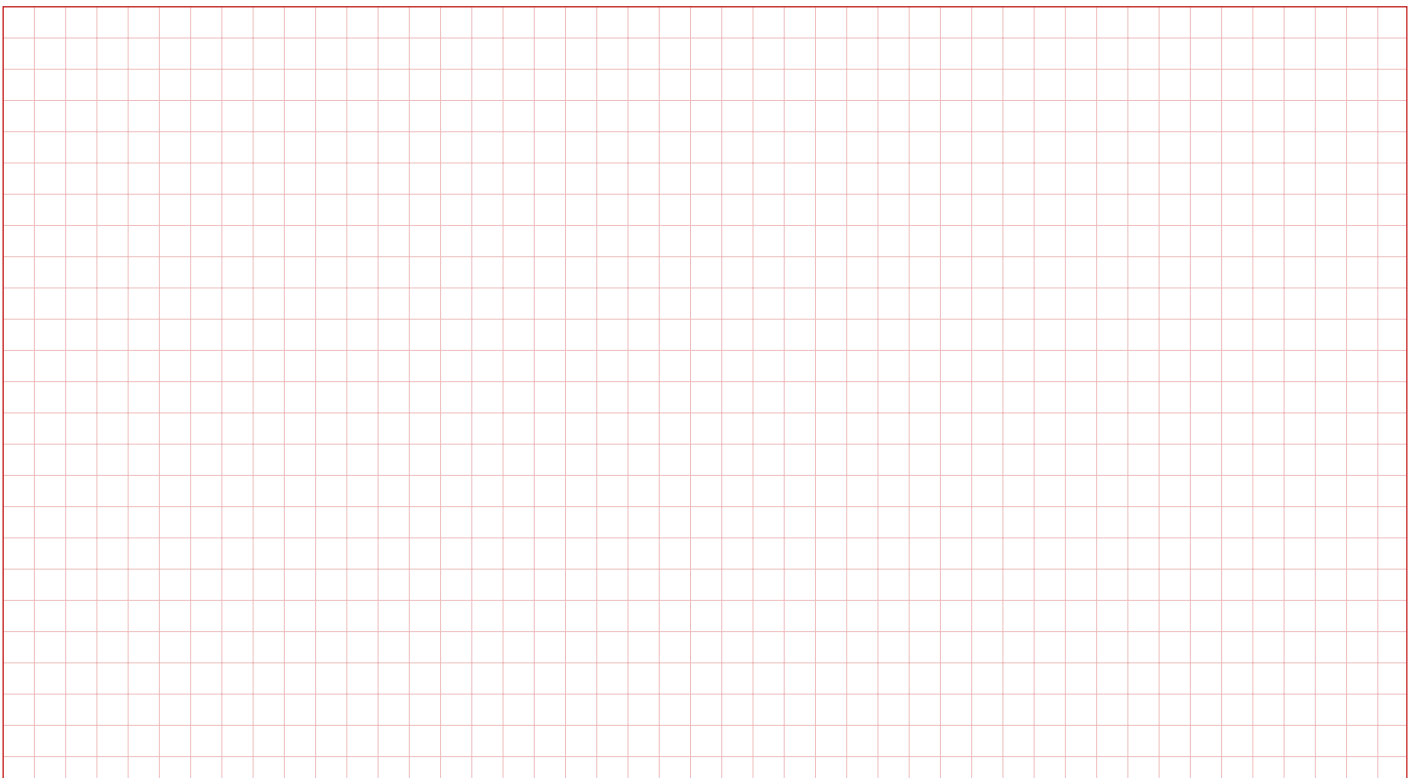


## Vergleich

Variante	Beschreibung	Test-Transferrate
2.	<b>byte</b> -Puffer (256 KB bis 1 MB)	1150 MB/s
4.	↗ <b>InputStream.transferTo</b>	1047 MB/s
3.	BufferedIn/OutputStream	73 MB/s
1.	Jedes <b>byte</b> einzeln	0.2 MB/s

- ▶ Zahlen mit **Vorsicht** genießen!
- ▶ **Abhängig** von
  - ▶ **Hardware**: CPU (Caches), Festplatte, Arbeitsspeicher
  - ▶ **Betriebssystem**
  - ▶ **Java-Implementierung**: Details der Implementierung in In/OutputStream
- ▶ **Im Test**: Macbook Pro 2019, macOS 10.15.5, Oracle JDK 13

## Notizen



# Inhalt

# Byteströme

## Zusammenfassung

75

## Notizen

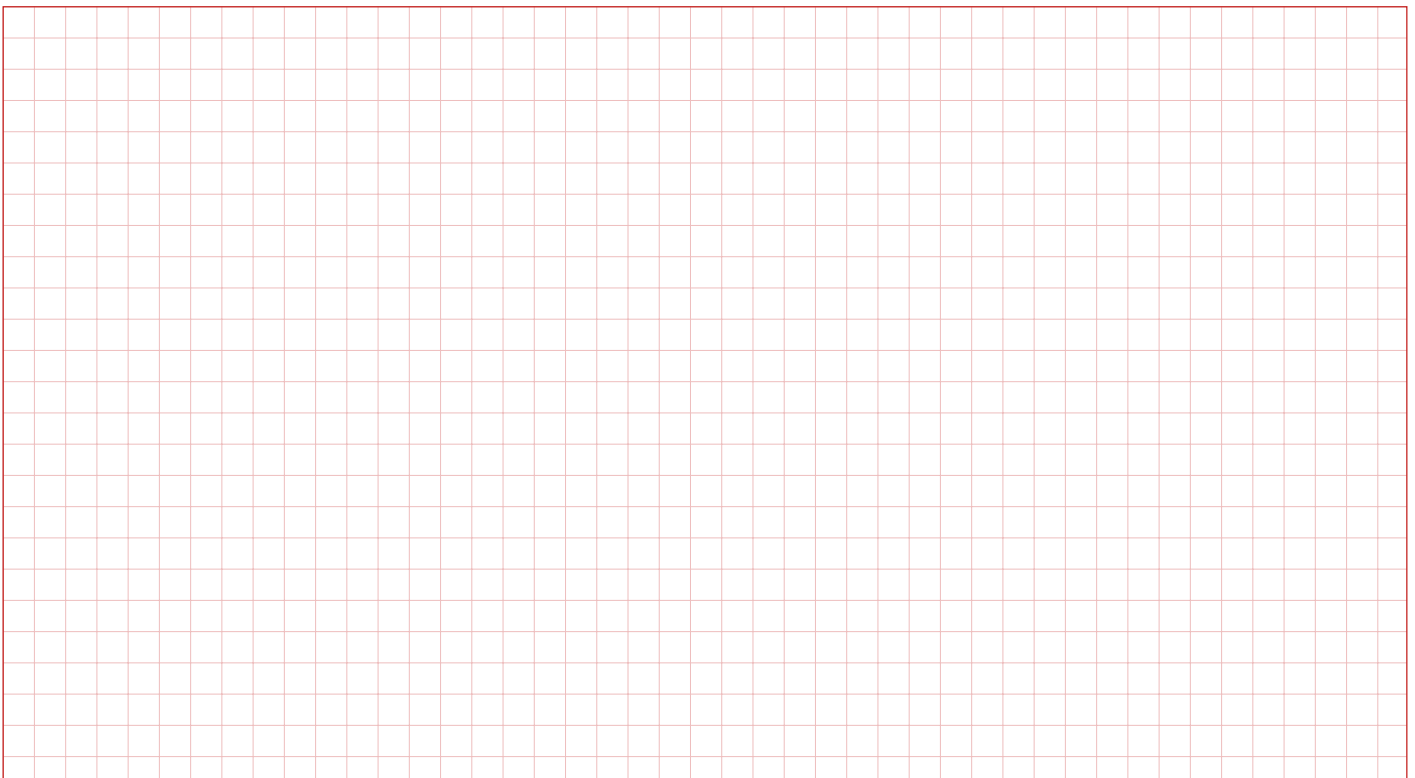
## In/OutputStream

<<abstract>>  
**InputStream**

<<abstract>>  
**OutputStream**

- ▶ Abstrakte Schnittstellen zum **Lesen und Schreiben von bytes**
- ▶ **Byteströme, Binärdaten**
- ▶ Wichtige Methoden
  - ▶ close
  - ▶ ↗ **InputStream**: read, skip, reset
  - ▶ ↗ **OutputStream**: write, flush
- ▶ Werfen ↗ **IOException** (später mehr zu Exception-Handling)
- ▶ **Blockieren** aufrufenden Programm-Thread
- ▶ Quellen/Senken
  - ▶ **Dateien**: FileIn/OutputStream
  - ▶ **byte-Arrays**: ByteArrayIn/OutputStream
  - ▶ **Programm-interner** Austausch: PipedIn/OutputStream
  - ▶ **Netzwerk**: ↗ **Socket**.getIn/OutputStream()
  - ▶ ...

## Notizen

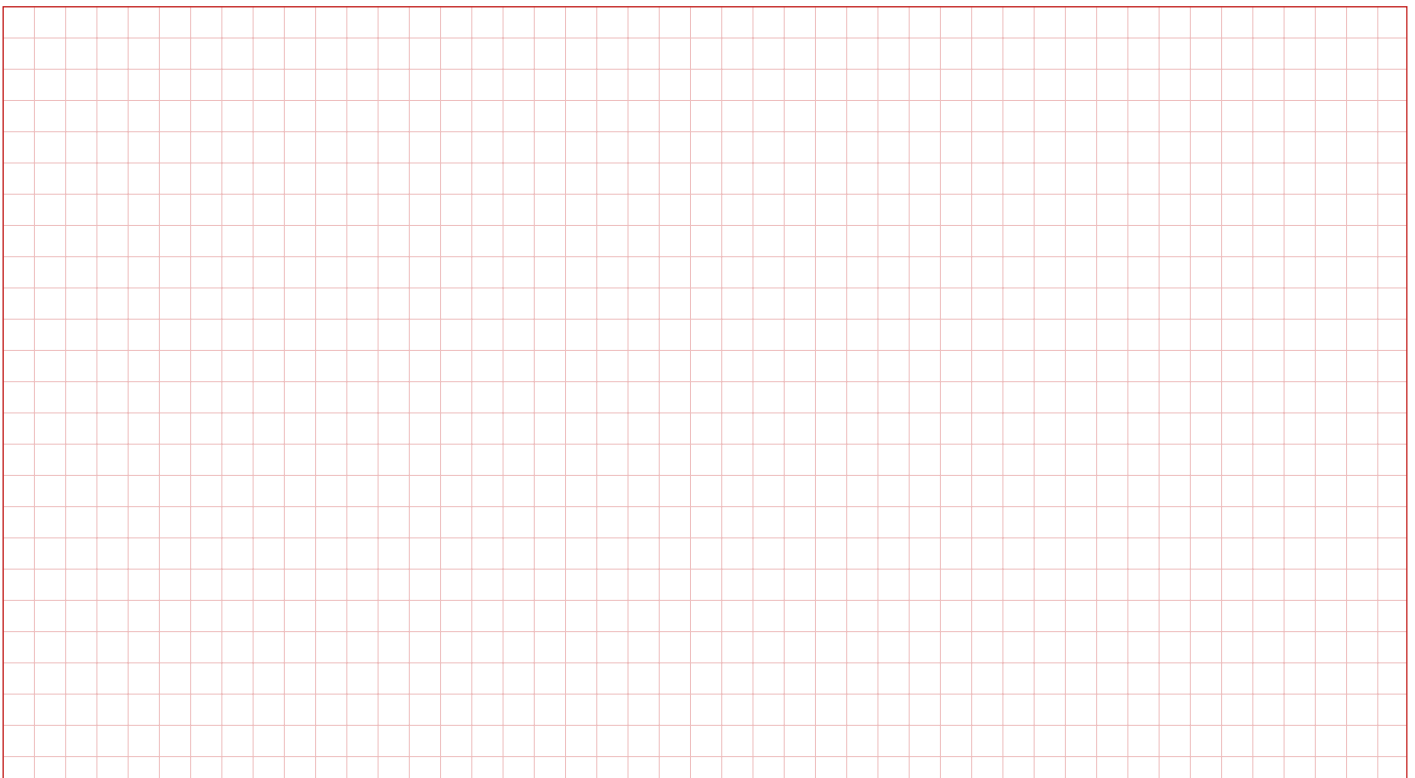


## Filter



- ▶ [FilterInputStream.read](#)
  - ▶ **bytes** von **in** **lesen**
  - ▶ **verarbeiten** (eventuell **transformieren**)
  - ▶ **weitergeben**
- ▶ [FilterOutputStream.write](#)
  - ▶ **bytes** **verarbeiten** (eventuell **transformieren**)
  - ▶ **in** **out** **schreiben**
- ▶ **Beispiele**
  - ▶ „**Echte**“ Filter: GZIPIn/OutputStream, CipherIn/OutputStream
  - ▶ Mehr **Funktionalität**: BufferedIn/OutputStream, DataIn/OutputStream, DigestIn/OutputStream
- ▶ Filter können **hintereinandergeschaltet** werden

## Notizen

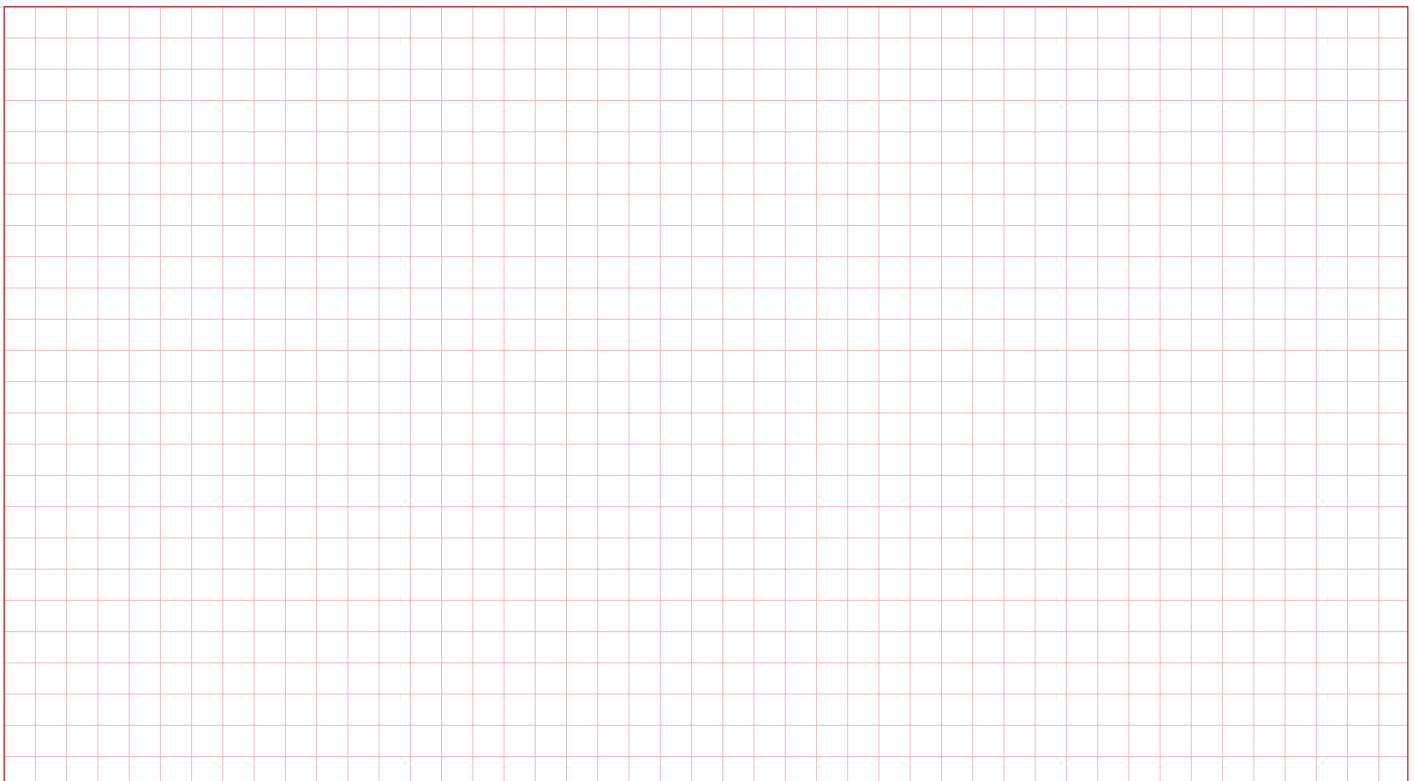


# Inhalt

## Text Ein- und Ausgabe

- Byteströme vs. Text
- Charsets und Encoding
- Reader und Writer
- Reader und Writer Quellen und Senken
- Filter
- Formatierte Textausgabe
- Einlesen von Daten: „Parsing“
- Zusammenfassung

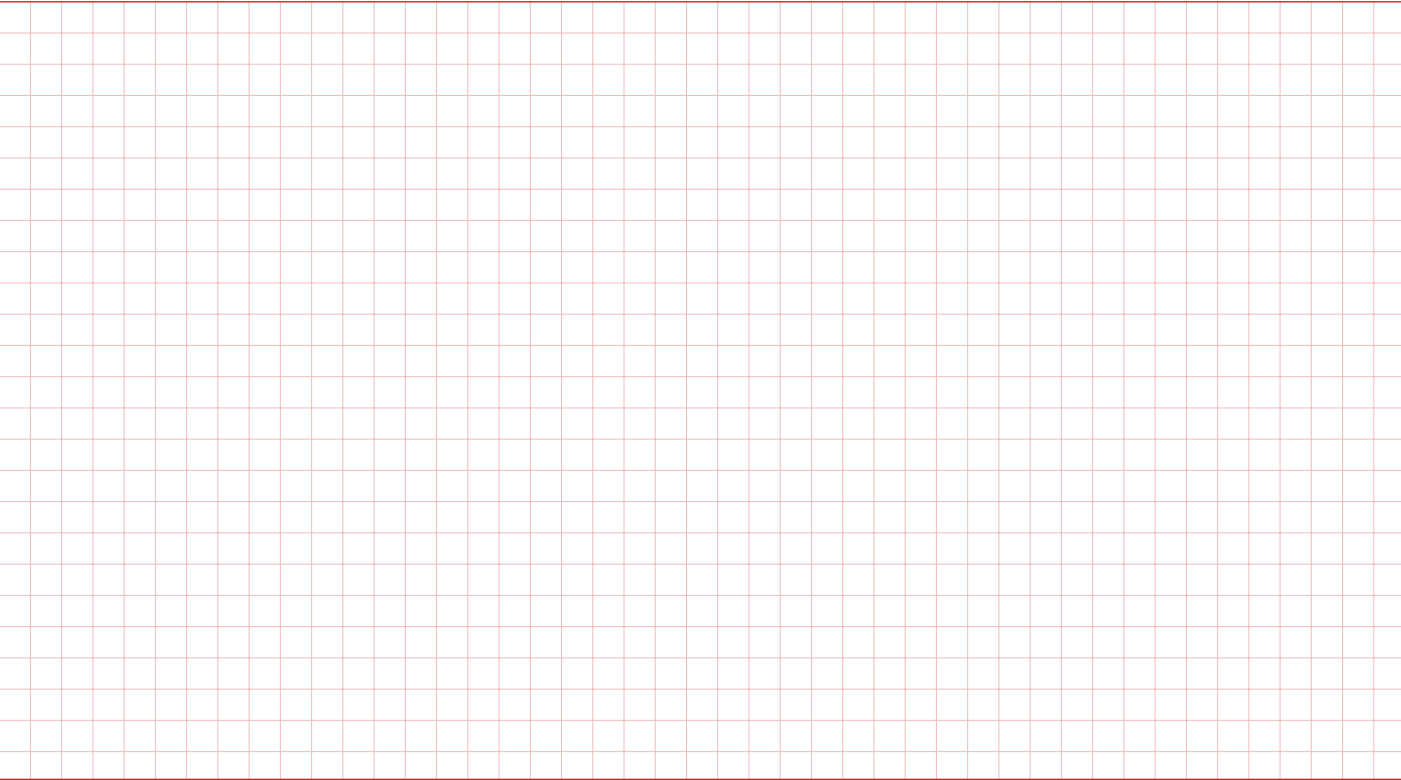
## Notizen



# Inhalt


Text Ein- und Ausgabe  
Byteströme vs. Text

## Notizen




## Byteströme vs. Text

- ▶ Bisher: **byte**-Arrays (**Byteströme**)
- ▶ Jetzt: **char**-Arrays (**Text**)
- ▶ „Was ist der **Unterschied**?“
- ▶ **Beispiel** anhand von Schreiben eines **ints**
  - ▶ Bytestrom

```
26  runWriteIntByteStream  
27 var file = new FileOutputStream("answer.bin");  
28 var data = new DataOutputStream(file);  
29 data.writeInt(42);
```

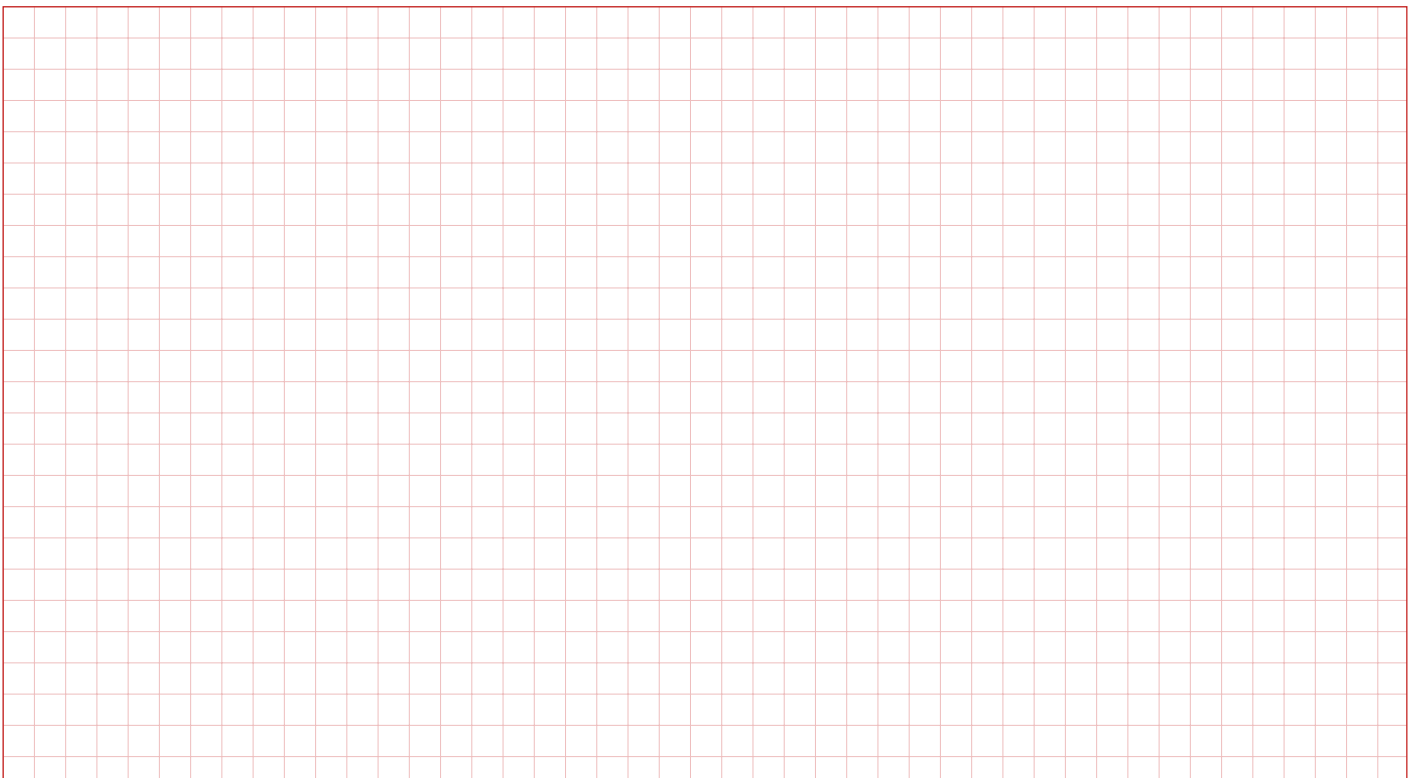
 ReaderWriterExamples.java

- ▶ Text (Details **später**)

```
36  runPrintIntText  
37 var file = new PrintWriter("answer.txt");  
38 file.print(42);
```

 ReaderWriterExamples.java

## Notizen





# Byteströme vs. Text

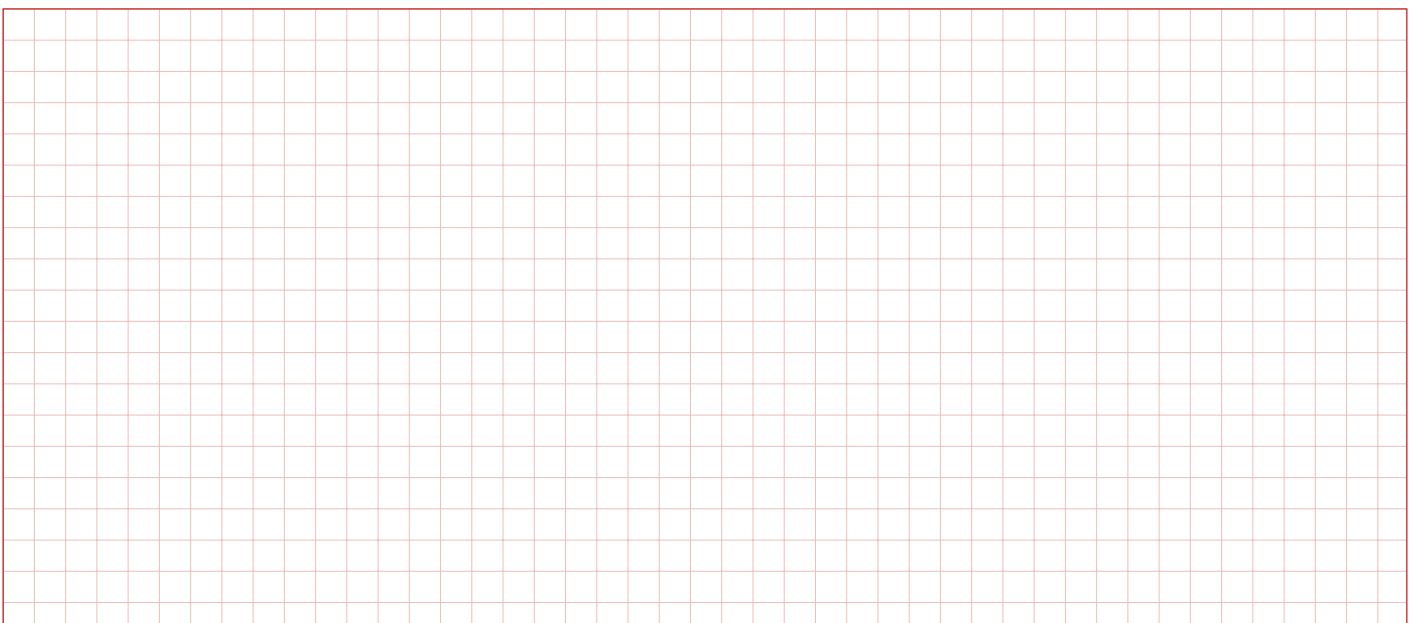
## Inhalte

	answer.bin	answer.txt
Binär (0x)	00 00 00 2a 34 32	
Text	"*"	"42"

- Interpretation **binär**
  - 00 00 00 2a entspricht **interner** Darstellung von (**int**) 42
  - 34 32 entspricht den **ASCII-Zeichen** für „4“ und „2“
- Interpretation als **Text**
  - 00 00 00 2a entspricht der **Zeichenkette** `\0\0\0*`
  - 42 entspricht der **Zeichenkette** `"42"`
- Daten müssen unterschiedlich **interpretiert** werden

## Notizen

- `\0` entspricht dem 0-Zeichen, das bspw. in C als Abschlussmarkierung von Zeichenketten in `char[]` verwendet wird.



## Byteströme vs. Text

	Byteströme	Text
Primitiver Datentyp	byte	char
Lesbarkeit	„machine-readable“	„human-readable“
Platzbedarf	kompakt	hoch
Informationsverlust	exakt	evtl. Verlust
Verarbeitungseffizienz	hoch	niedrig („parsing“)
Portabilität	evtl. hardwareabhängig	hoch
Beispiele	JPEG, MP4	XML, JSON

### Notizen

- Zu „Platzbedarf“: In dem vorherigen Beispiel war der Speicherbedarf des binären Formates mit 4 Byte höher als der des Textformats mit nur 2 Byte. Allerdings ist das dem gewählten Beispiel geschuldet: Bei größeren Zahlen und Gleitkommazahlen ist der Verbrauch im Textformat höher. Zu dem verwenden Textformate strukturierende Elemente (z.B. Tags in XML) und Formatierung für die Lesbarkeit, die den Speicherbedarf im Vergleich zu Binärformaten stark anwachsen lässt.
- Zu „Verlust“: Ein Informationsverlust im Textformat kann auftreten, wenn bspw. Gleitkommazahlen in eine Textdarstellung zur Basis 10 überführt werden. Diese Konvertierung führt in der Regel zu Informationsverlust im Vergleich zur binären, internen Darstellung.



# Inhalt

## Text Ein- und Ausgabe

### Charsets und Encoding

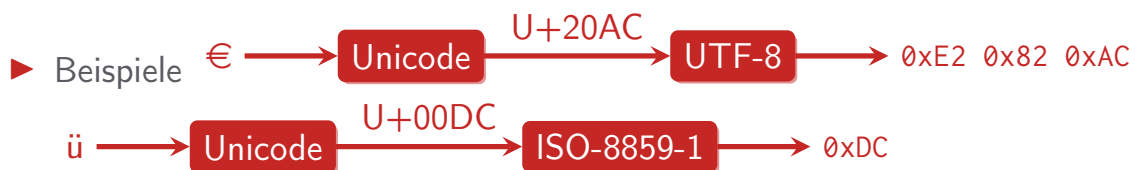
# Notizen

# Charsets und Encoding

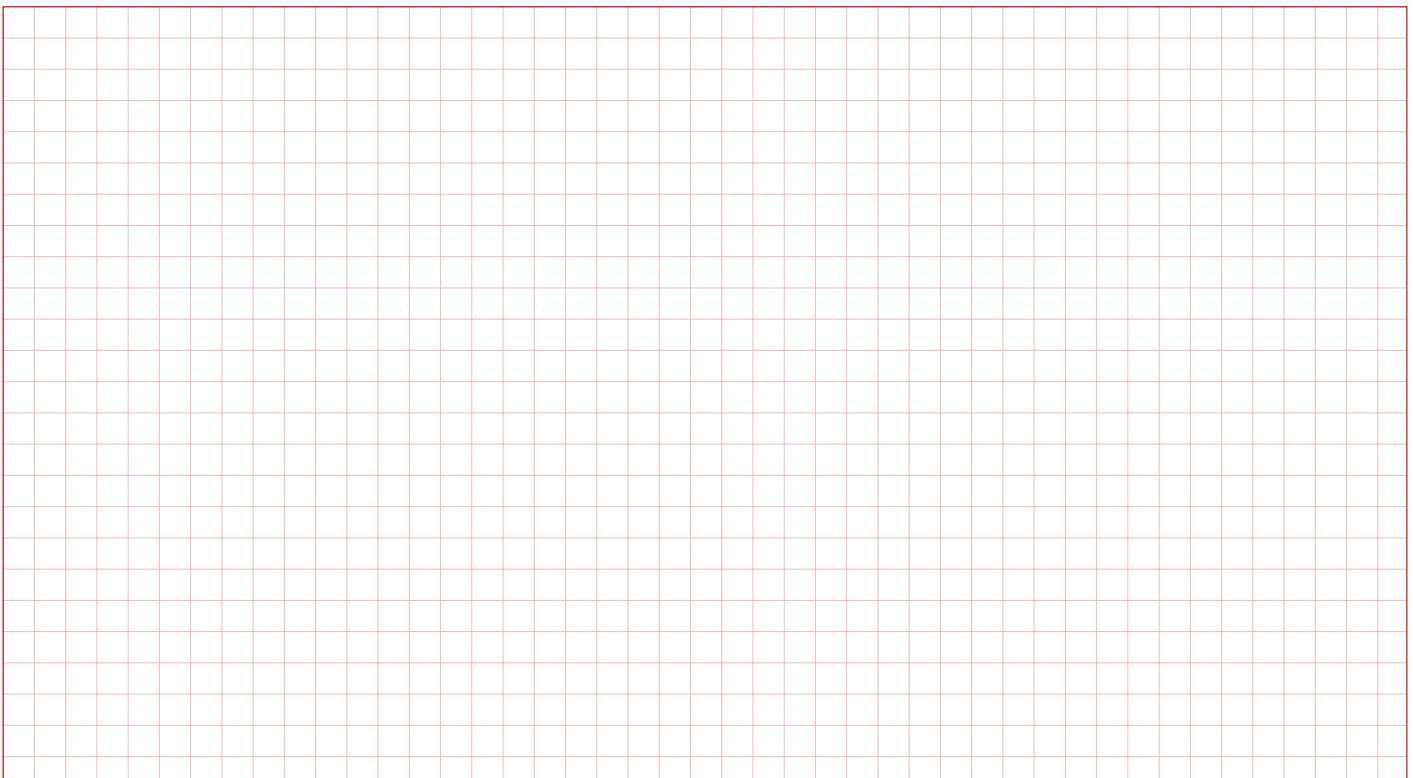
- ▶ Was ist der Unterschied zwischen „charset“ (Zeichensatz) und „encoding“?
- ▶ Wie kommt man von einem Zeichen (z.B. €) zu seiner Codierung als Bytesequenz (byte[])?
- ▶ Prinzip



- ▶ Charset bildet (abstraktes) Symbol/Zeichen auf Code (Zahl) ab
- ▶ Encoding bildet Code auf Bytesequenz ab



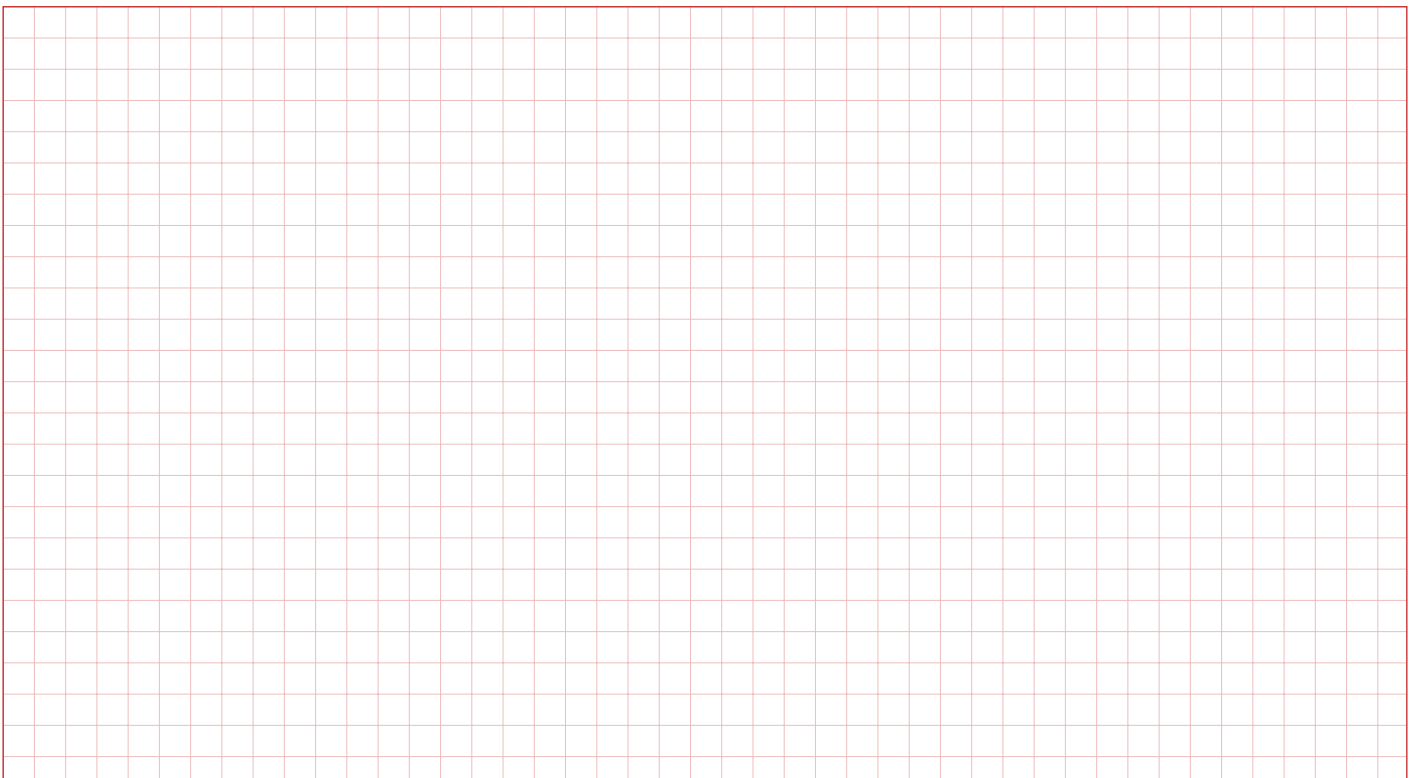
## Notizen



## char in Java


- ▶ Zur Erinnerung
  - ▶ Primitiver Typ **char** für Zeichen
  - ▶ Zwei Byte
  - ▶ Positive ganze Zahl von 0x0000 bis 0xFFFF
  - ▶ Literale: 'ü', '\u00FC' (entspricht Unicode)
- ▶ Internes Encoding von **char**/[↗](#) **String**
  - ▶ UTF-16 mit fixer Länge (zwei Bytes)
  - ▶ Entspricht 1:1 dem Wert des **char**
- ▶ Beispiel ü → **Unicode**  $\xrightarrow{\text{\u00DC}}$  **UTF-16** → 0x00 0xDC
- ▶ Merken für die nächsten Kapitel!
- ▶ Hinweise
  - ▶ Unicode zu Zeiten von Java 1.0: 0x0000 bis 0xFFFF
  - ▶ Das reicht nicht aus für alle Sprachen
  - ▶ Unicode heute: 0x0000 bis 0x10FFFF
  - ▶ In Java mit **char** nicht möglich (nur über **int**)

## Notizen



# Die Klasse Charset

- ▶ Die Klasse [Charset](#)
  - ▶ Modelliert ein **Encoding** (ungünstiger Name, hat aber Gründe)
  - ▶ Kann **encodieren**: **char[]**/[String](#) → **byte[]**
  - ▶ Kann **decodieren**: **byte[]** → **char[]**/[String](#)
  - ▶ Verwaltet alle **unterstützten Encodings** (kein Konstruktor)
- ▶ Auflisten **aller unterstützten Encodings**

```
14  runCharsetListEncodings  
15 for (Charset charset : Charset.availableCharsets().values())  
16     out.println(charset.name());
```

 CharsetExamples.java

```
...  
UTF-16  
...  
ISO-8859-1  
...
```

## Notizen



## Die Klasse Charset


- ▶ `defaultCharset()` liefert Standard-Encoding des Systems

```
22  runCharsetDefaultEncoding  
23 out.println(Charset.defaultCharset().name());
```

 CharsetExamples.java

```
UTF-8 // macOS
```

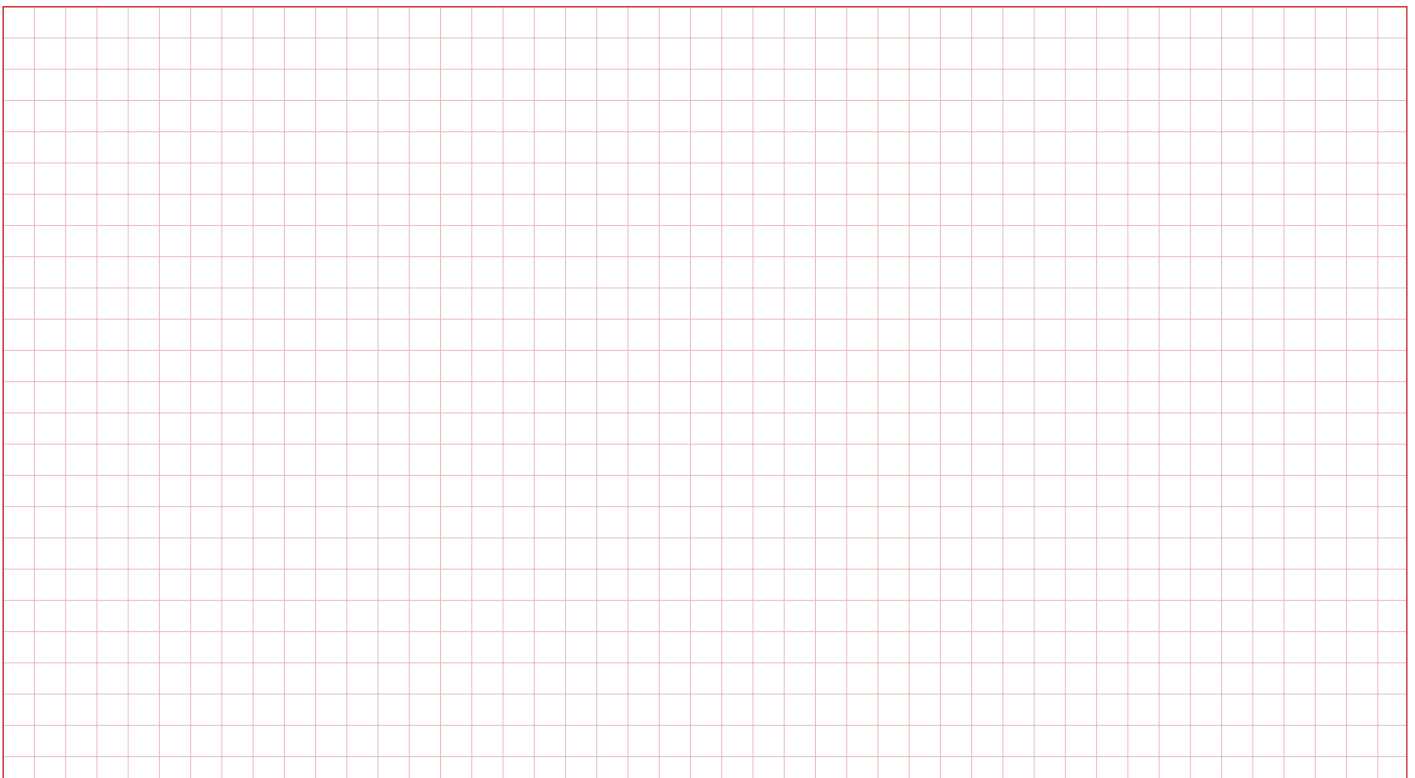
- ▶ Beispiel für Encodieren/Decodieren

```
29  runCharsetEncodeDecode  
30 Charset iso8859 = Charset.forName("ISO-8859-1");  
31 ByteBuffer b = iso8859.encode("Süßölgefäß");  
32 out.println(Arrays.toString(b.array()));  
33 CharBuffer c = iso8859.decode(b);  
34 out.println(c.toString());
```

 CharsetExamples.java

```
[83, -4, -33, -10, 108, 103, 101, 102, -28, -33]  
Süßölgefäß
```

## Notizen



## Inhalt

## Text Ein- und Ausgabe

### Reader und Writer

## Notizen



# Reader und Writer

## ► Kurzfassung

	Lesen	Schreiben
<b>byte</b>	<a href="#">InputStream</a>	<a href="#">OutputStream</a>
<b>char</b>	<a href="#">Reader</a>	<a href="#">Writer</a>

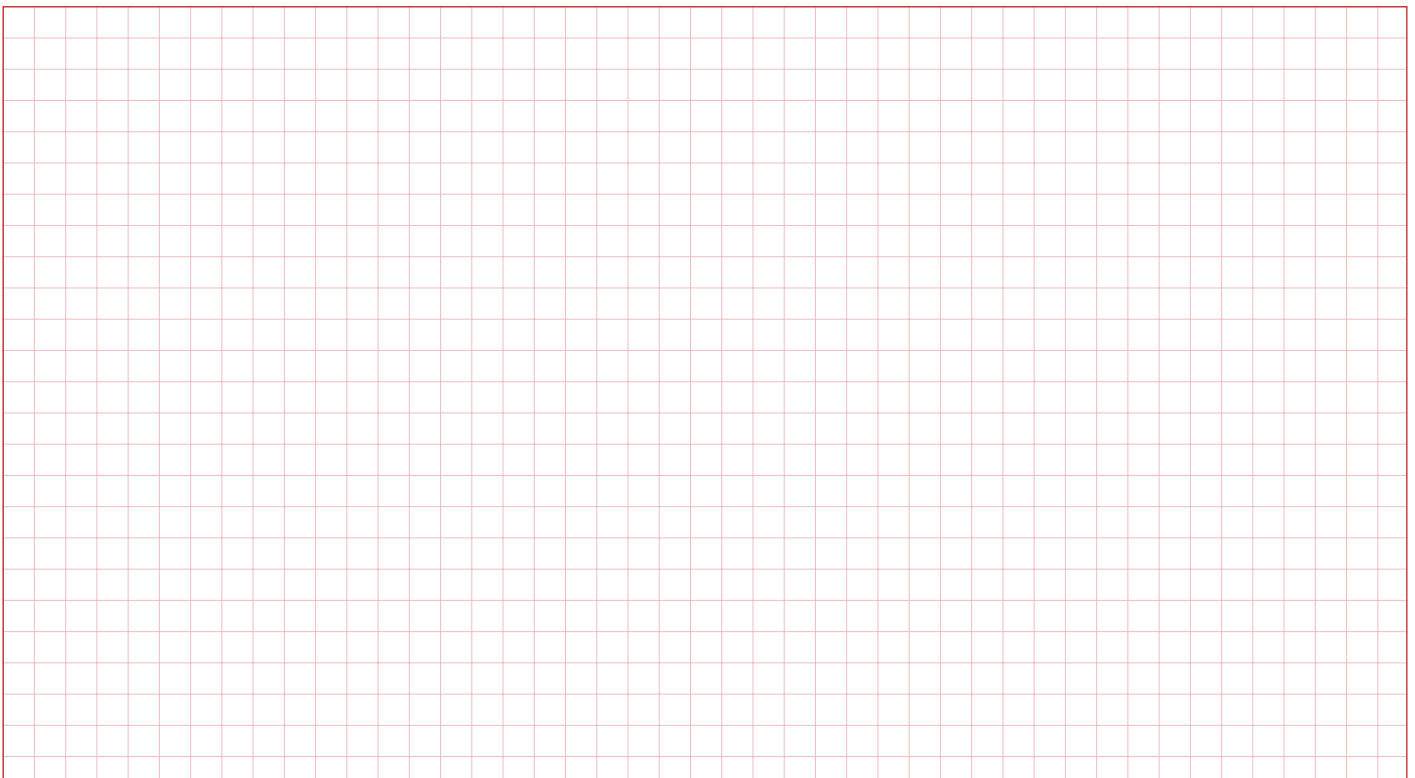
## ► [Reader](#)

- Ähnliche Schnittstelle wie [InputStream](#)
- **byte** → **char**

## ► [Writer](#)

- Ähnliche Schnittstelle wie [OutputStream](#)
- **byte** → **char**

## Notizen



## Reader

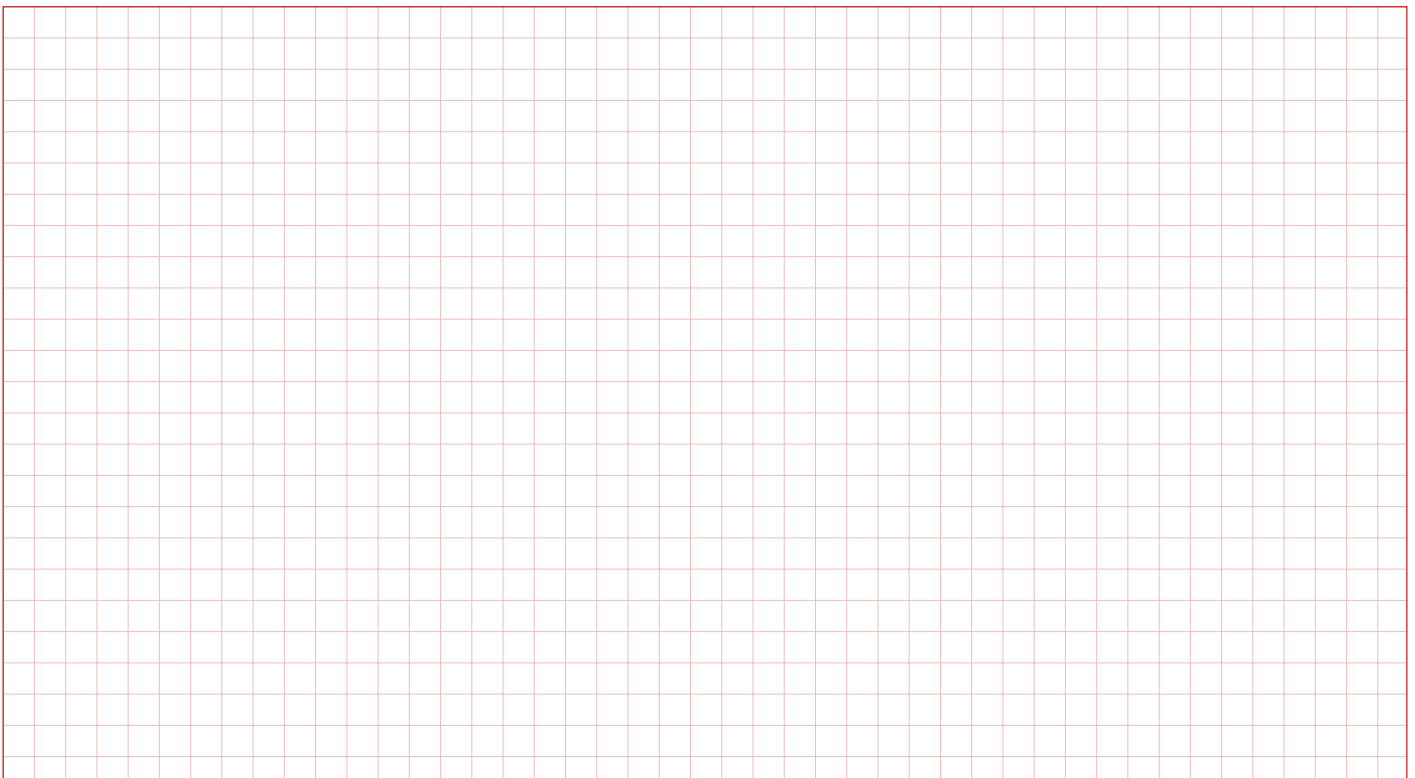
<<abstract>>

Reader

```
+ read() : int
+ read(c : char[]): int
+ read(c : char[], off : int, len : int): int
+ skip(n : long): long
+ mark(readAheadLimit : int)
+ markSupported(): boolean
+ reset()
+ close()
+ ready(): boolean
+ transferTo(w : Writer): long
...
```

- ▶ Methoden werfen [IOException](#) und blockieren
- ▶ **Abstrakt:** `int read(char[] c, int o, int l)` und `close()`
- ▶ `boolean ready()` entspricht `int InputStream.available()`

## Notizen



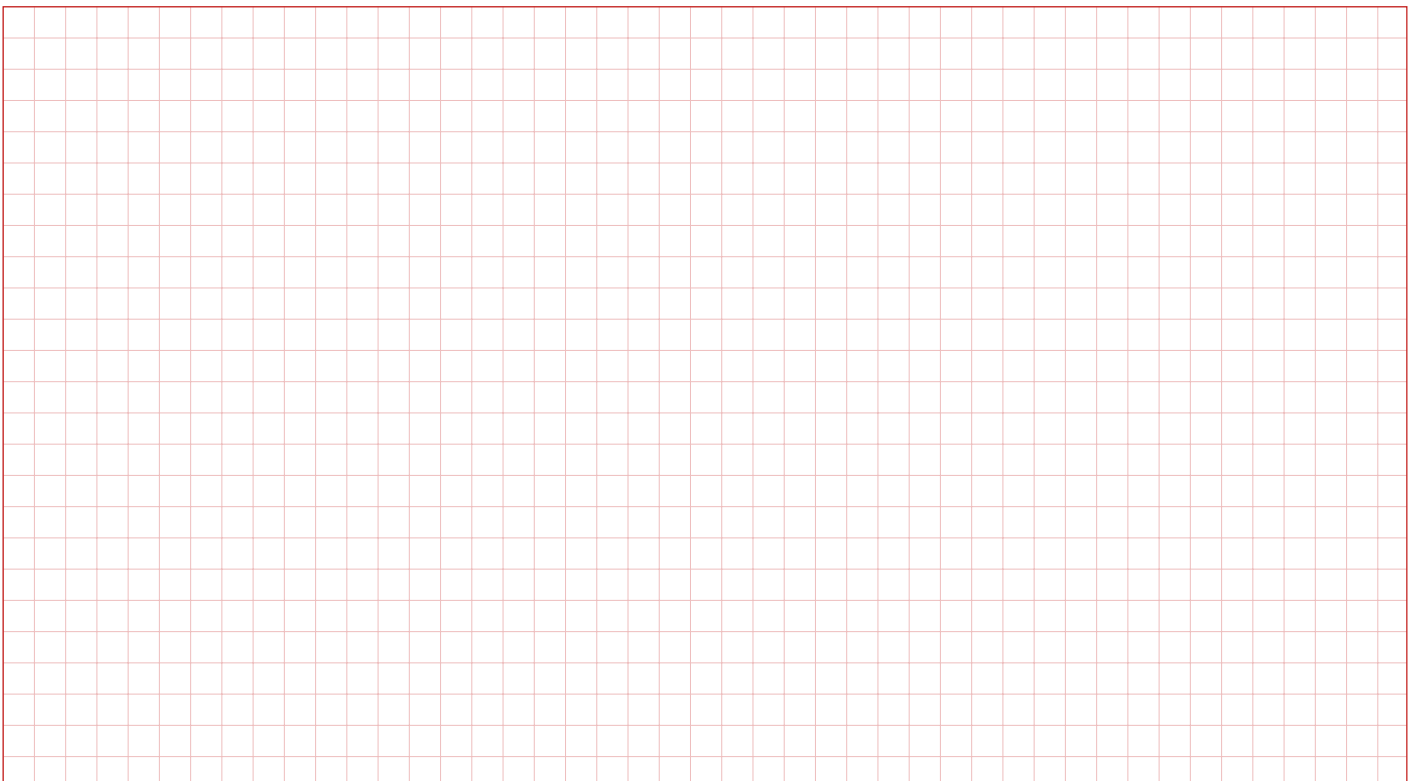
## Writer

<<abstract>>  
**Writer**

```
+ write(c : int)  
+ write(c : char[])  
+ write(c : char[], off : int, len : int)  
+ write(s : String)  
+ write(s : String, off : int, len : int)  
+ flush()  
+ close()  
...
```

- ▶ Methoden werfen [IOException](#) und blockieren
- ▶ Abstrakte Methoden
  - ▶ write(**char**[] c, **int** off, **int** len)
  - ▶ flush() und close()

## Notizen



# Inhalt

Text Ein- und Ausgabe

Reader und Writer Quellen und Senken

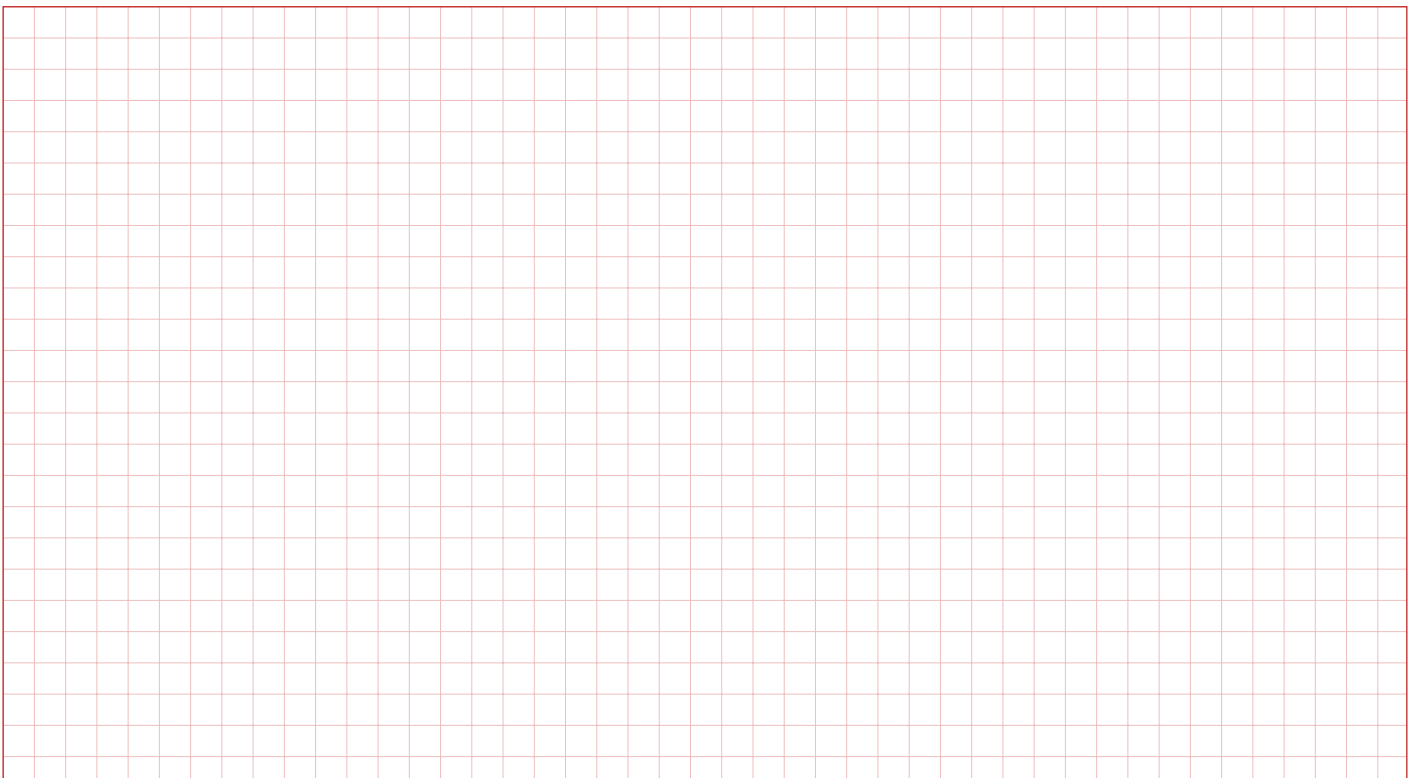
## Notizen

## Quellen und Senken

Reader/Writer	Ziel	Encoding
↗ <code>FileReader/Writer</code>	Datei	ja
↗ <code>CharArrayReader/Writer</code>	<code>char</code> -Array	nein
↗ <code>PipedReader/Writer</code>	↗ <code>PipedWriter/Reader</code>	nein
↗ <code>OutputStreamWriter</code>	↗ <code>OutputStream</code>	ja
↗ <code>InputStreamReader</code>	↗ <code>InputStream</code>	ja
...	...	...


- ▶ Bedeutung: siehe Quellen/Senken bei Byteströmen
- ▶ Ein **Encoding** kann immer angegeben werden, wenn
  - ▶ Quelle oder Ziel in „rohen Daten“ (**byte**-Strömen) enden
  - ▶ Beispiel ↗ `FileReader/Writer` in Datei
  - ▶ Default-Encoding: ↗ `Charset.defaultCharset()`

## Notizen



## Beispiel: FileWriter/Reader

► [FileWriter](#) mit ISO-8859-1 Encoding

```
46  runFileWriterEncoding
47 Charset iso8859 = Charset.forName("ISO-8859-1");
48 FileWriter out = new FileWriter("output.txt", iso8859);
49 out.write("Süßölgefäß");
50 out.close();
```


ReaderWriterExamples.java

```
$ cat output.txt
S???lgef?? // auf Terminal mit UTF-8 Encodierung
$ file output.txt
output.txt: ISO-8859 text, with no line terminators
```

## Notizen

## Beispiel: FileWriter/Reader

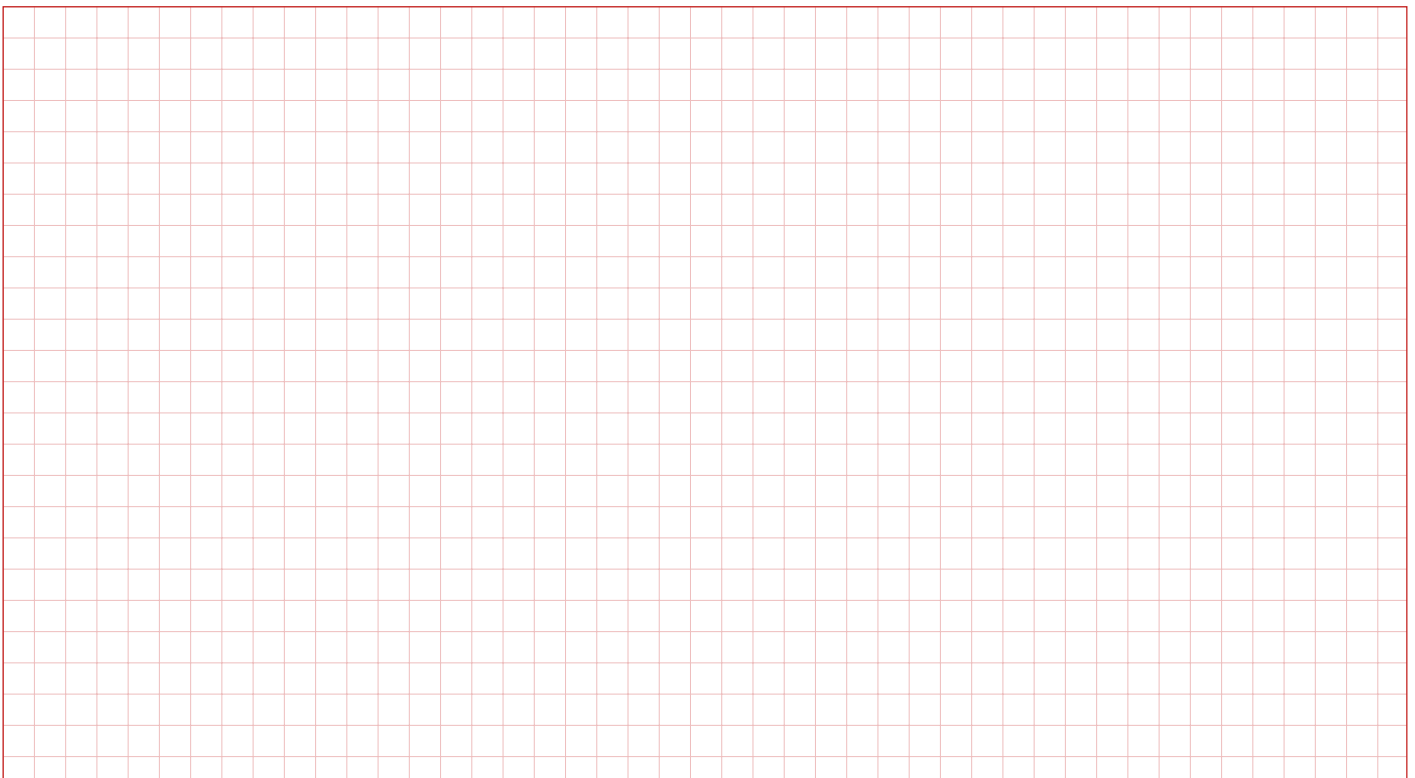
### ► [FileReader](#) mit ISO-8859-1 Encoding

```
56  runFileReaderEncoding  
57 char[] c = new char[1024]; // magic number  
58 Charset iso8859 = Charset.forName("ISO-8859-1");  
59 FileReader in = new FileReader("output.txt", iso8859);  
60 int count = in.read(c);  
61 out.printf("Gelesen: %d character%n", count);  
63 // erstelle String aus c[0..count-1]  
64 String s = new String(c, 0, count);  
65 out.println(s);  
67 in.close();
```

 ReaderWriterExamples.java

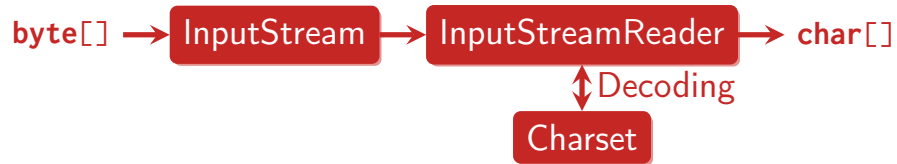
```
Gelesen: 10 character  
Süßölgefäß
```

## Notizen

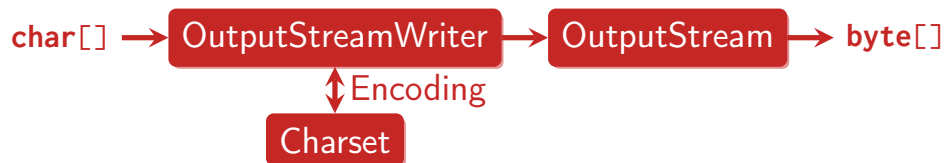


## InputStreamReader und OutputStreamWriter

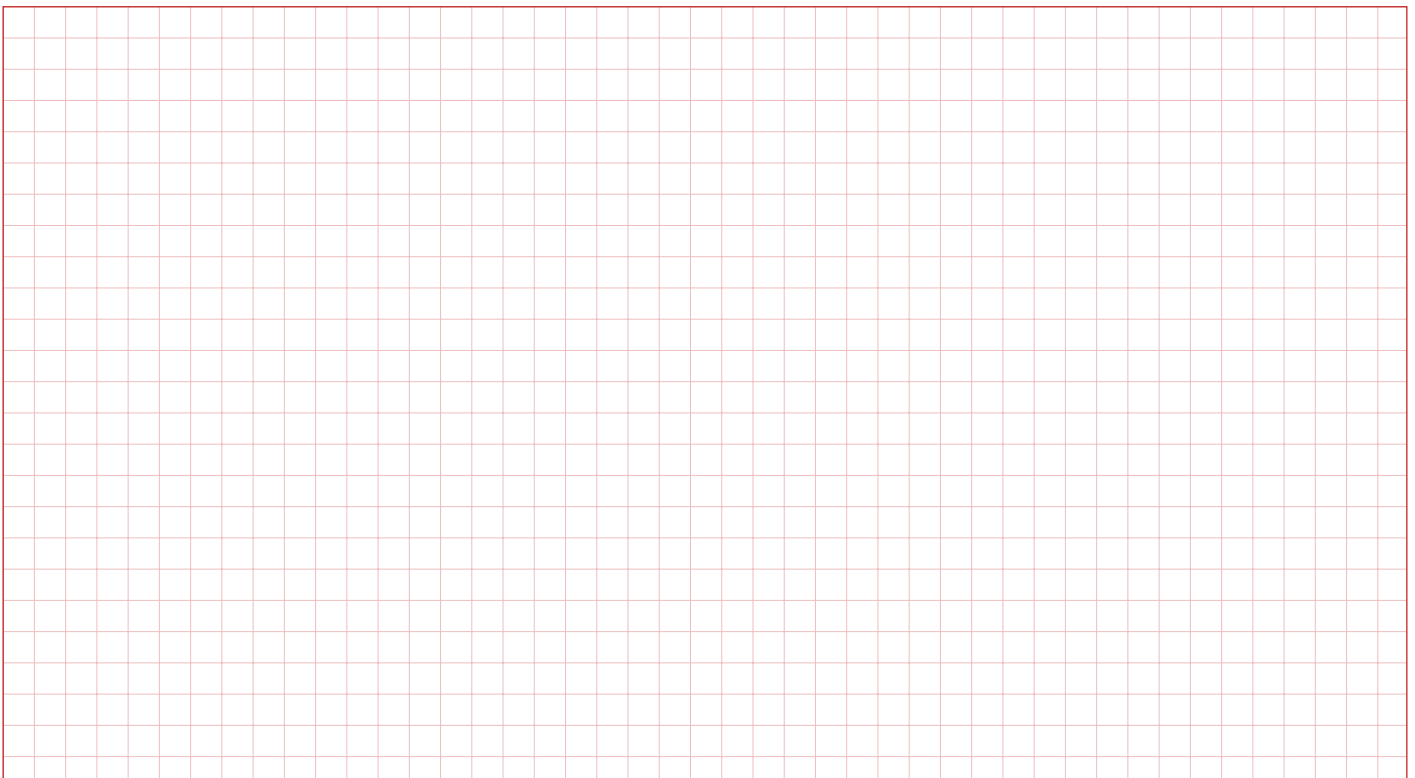
- ▶ Problem: Manchmal steht nur `InputStream` oder `OutputStream` zur Verfügung
- ▶ Wie bringt man `Reader`/`Writer` mit `InputStream`/`OutputStream` zusammen?
- ▶ `InputStreamReader`



- ▶ `OutputStreamWriter`



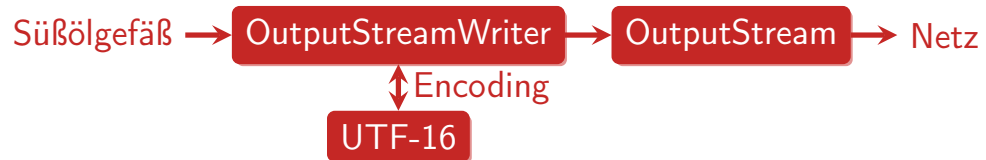
### Notizen



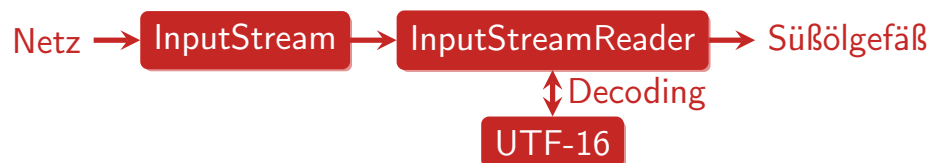


## Beispiel: Sockets

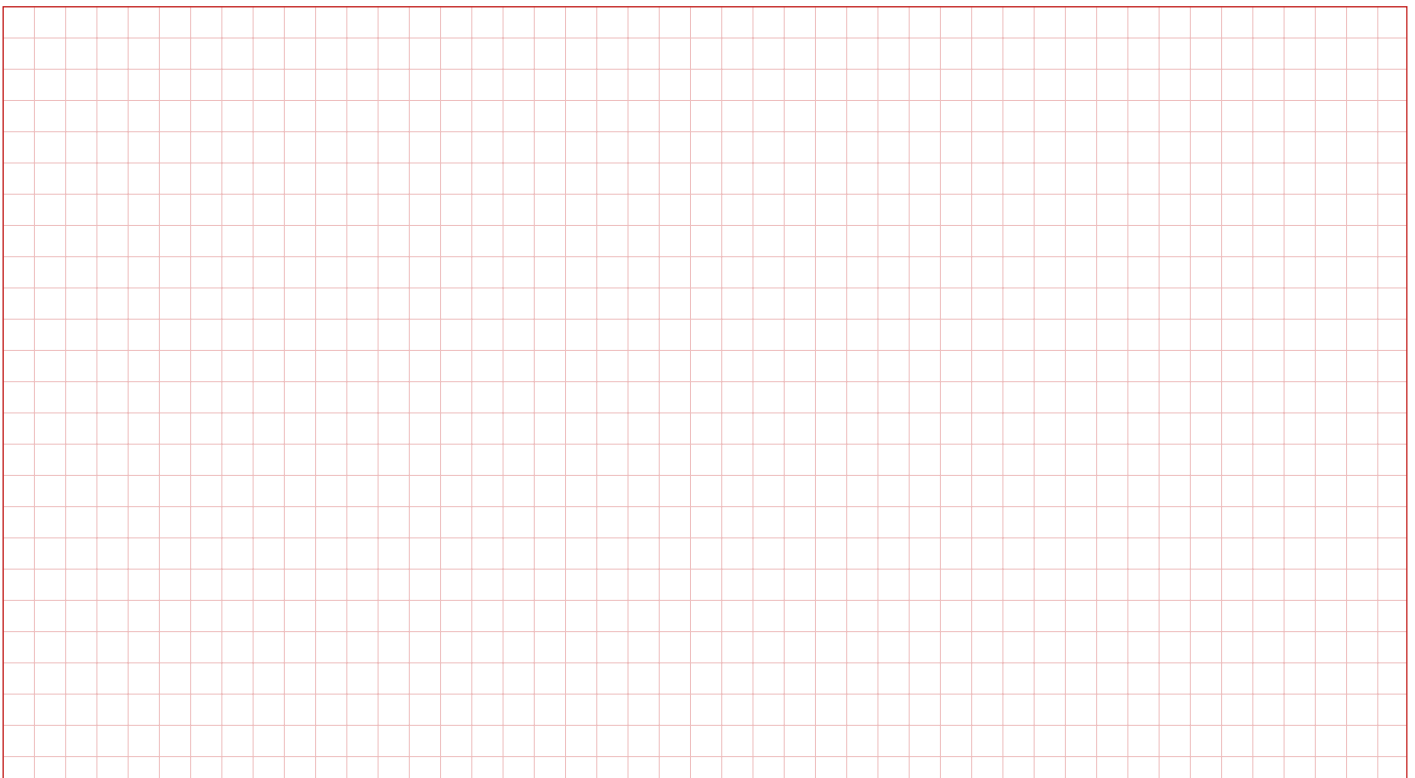
- ▶ `Socket`/`ServerSocket` zur Netzwirkommunikation
- ▶ Keine Angst: Hier nur oberflächlich
- ▶ „Problem“: `Socket` bietet nur `InputStream`/`OutputStream` an
- ▶ Wir sollen **encodierte Strings** schreiben!
- ▶ Client **verbindet sich** und **schreibt String** „Süßölgefäß“ (UTF-16)



- ▶ Server **horcht auf Verbindungen** und **liest String** (UTF-16)




## Notizen



## Beispiel: Sockets

### ► Server

```
73  runInputStreamReaderExample  
74 ServerSocket server = new ServerSocket(12345);  
76 out.println("Waiting for incoming connections...");  
77 Socket connection = server.accept();  
79 InputStream inputStream = connection.getInputStream();  
80 InputStreamReader reader = new InputStreamReader(inputStream, "UTF-16");  
82 char[] b = new char[1024];  
83 int count = reader.read(b);  
84 String s = new String(b, 0, count);  
85 out.println(s);  
87 reader.close();  
88 server.close();
```

 ReaderWriterExamples.java

## Notizen



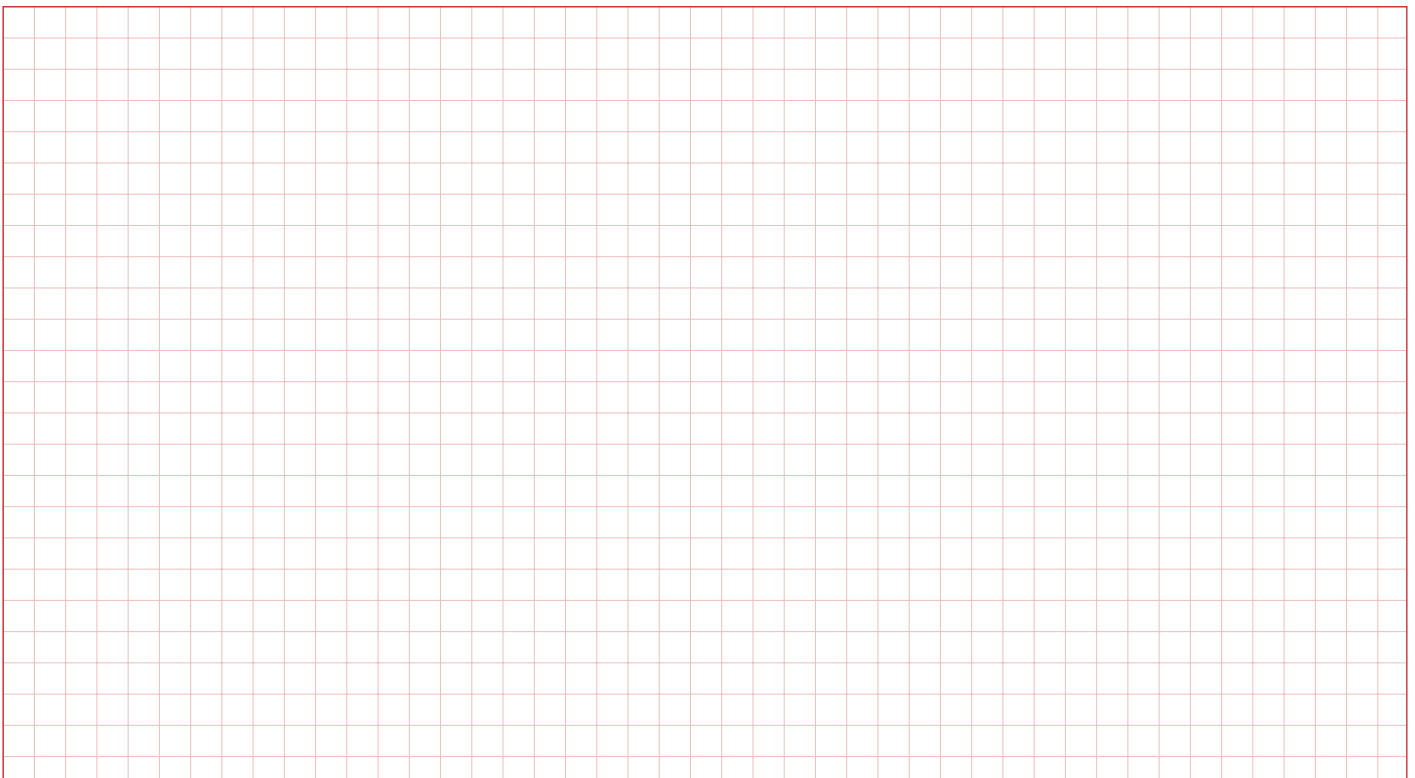
## Beispiel: Sockets

### ► Client

```
95  runOutputStreamWriterExample  
96 Socket client = new Socket();  
97 out.println("Connecting...");  
98 client.connect(new InetSocketAddress("localhost", 12345));  
100 OutputStream outputStream = client.getOutputStream();  
101 OutputStreamWriter writer = new OutputStreamWriter(outputStream, "UTF-16");  
103 writer.write("Süßölgefäß");  
104 out.println("done...");  
106 writer.close();  
107 client.close();
```

 ReaderWriterExamples.java

### Notizen



## Beispiel: Sockets

- ▶ Ausführen **Server** (blockiert)

```
Waiting for incoming connections...
```

- ▶ Ausführen **Client** (in zweitem Terminal)

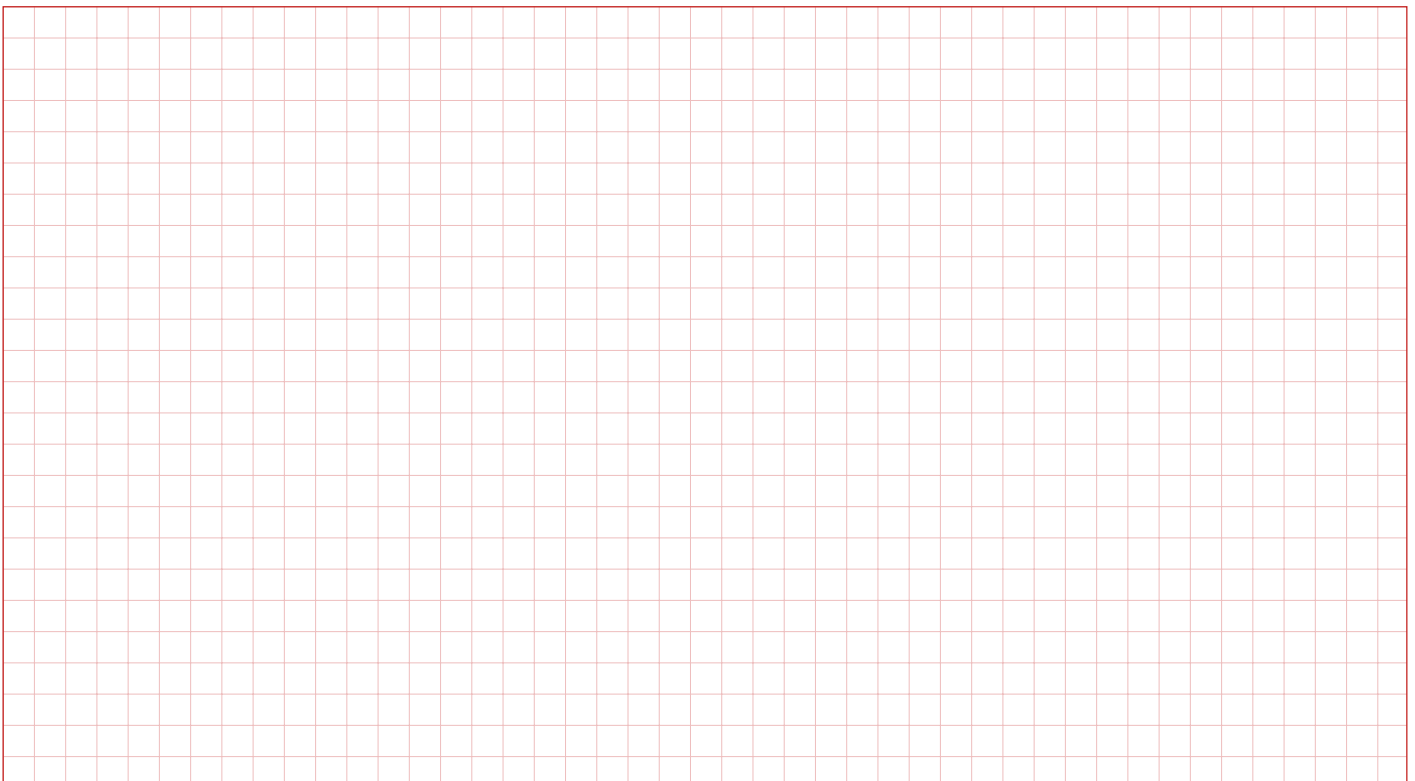
```
Connecting...  
done
```

- ▶ **Server**

```
Waiting for incoming connections...  
Süßölgefäß
```

- ▶ **Übung:** Was passiert wenn die **Encodings** nicht zusammenpassen?

## Notizen

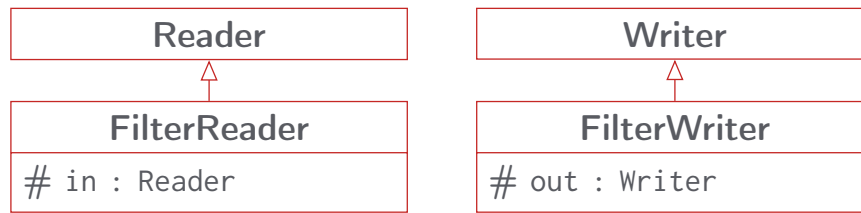


# Inhalt

## Text Ein- und Ausgabe

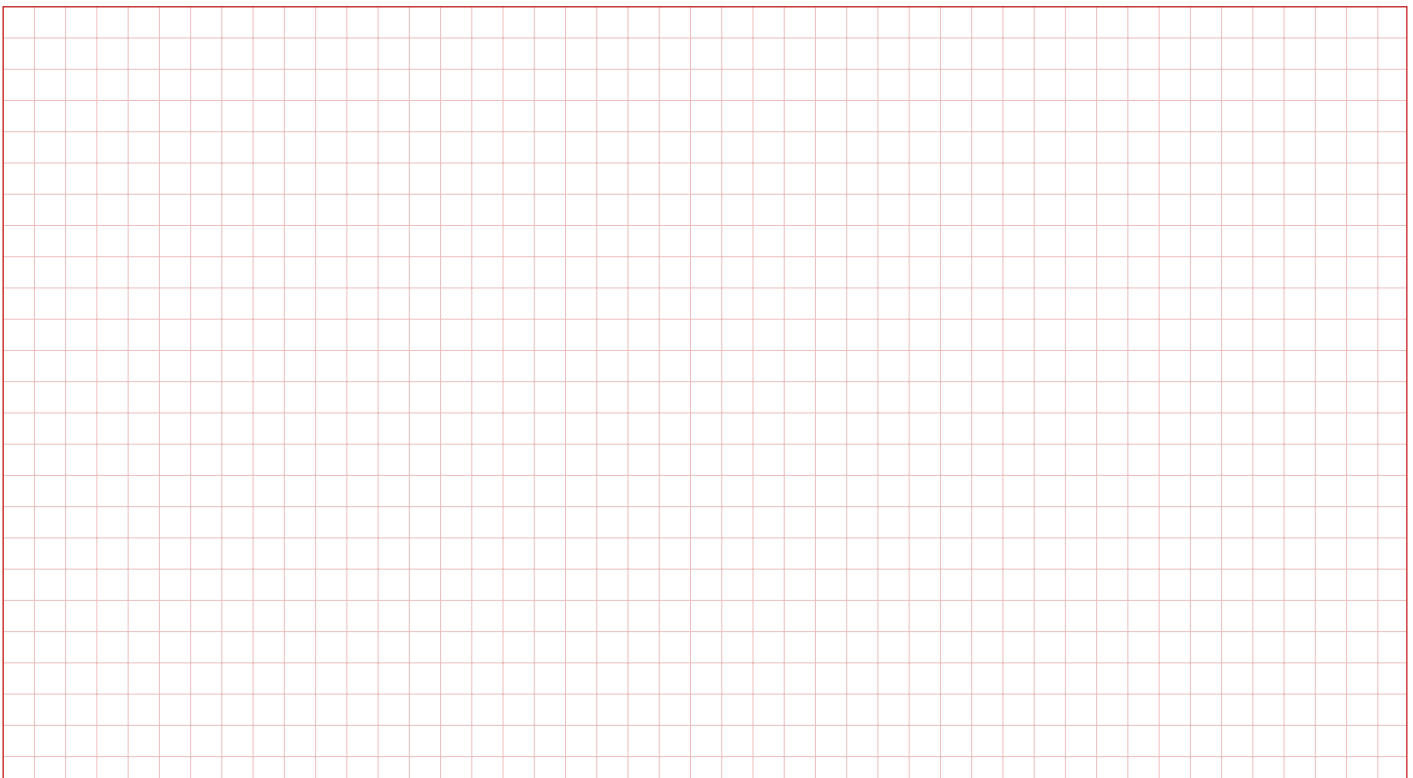
# Notizen

## FilterReader und FilterWriter



- ▶ Prinzip wie bei `FilterIn/OutputStream`
- ▶ `FilterReader`
  - ▶ `chars` werden aus `in` **gelesen**...
  - ▶ ...und dann (eventuell) transformiert
- ▶ `FilterWriter`
  - ▶ `chars` werden (eventuell) transformiert...
  - ▶ ...und dann nach `out` **geschrieben**...

## Notizen



# Filter



Klasse	Funktion	Oberklasse
↗ <code>BufferedReader/Writer</code>	Zeilenpufferung	↗ <code>Reader/Writer</code>
↗ <code>LineNumberReader</code>	Zeilennummerierung	↗ <code>BufferedReader</code>
<code>PushBackReader</code>	Lesen mit zurückspringen	↗ <code>FilterReader</code>
↗ <code>PrintWriter</code>	Formatierte Ausgabe	↗ <code>Writer</code>


- ▶ Nur `PushBackReader` leitet von ↗ `FilterReader` ab
- ▶ Aber alle sind konzeptionell Filter

## Notizen



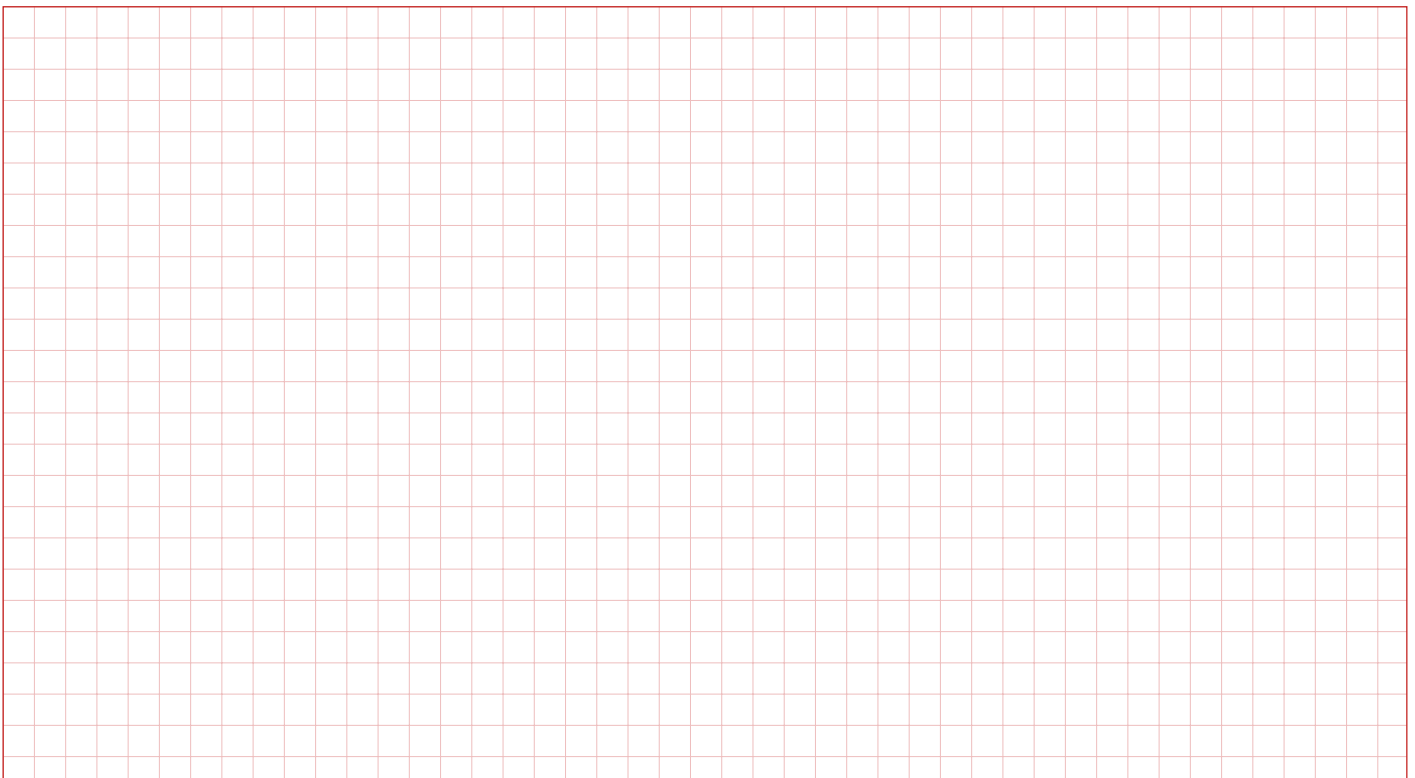
## BufferedReader/Writer

- ▶  `BufferedReader/Writer`
  - ▶ Zeilenorientiertes Lesen und Schreiben
  - ▶ Pufferung (in `char[]`) für Effizienz
- ▶  `String` `BufferedReader.readLine()` liest Zeile (`null` wenn Ende)
- ▶ `void` `BufferedWriter.newLine()` neue Zeile
- ▶ Beispiel schreibt Quadratzahlen in `squares.txt`

```
15  runBufferedWriterExample  
16 FileWriter file = new FileWriter("squares.txt");  
17 BufferedWriter out = new BufferedWriter(file);  
19 for (int i = 0; i < 10; i++) {  
20     out.write(i + "^2 = " + (i*i));  
21     out.newLine();  
22 }  
24 out.close();
```

 `TextFilterExamples.java`


## Notizen





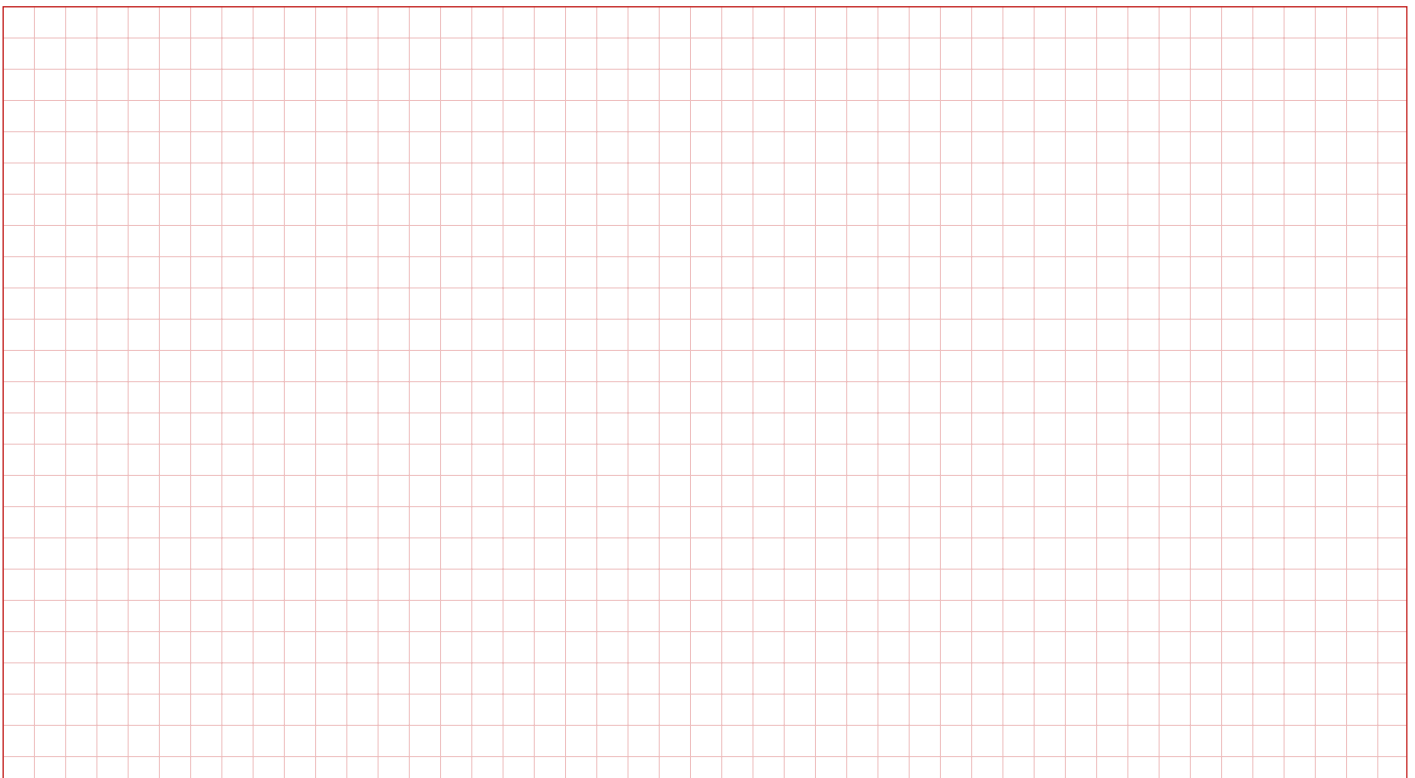
## BufferedReader/Writer

- ▶ [LineNumberReader](#) extends `BufferedReader`
- ▶ Zählt **Zeilennummern**
- ▶ Beispiel

```
30  runLineNumberReaderExample
31 FileReader file = new FileReader("squares.txt");
32 LineNumberReader in = new LineNumberReader(file);
34 String line;
36 do{
37     int n = in.getLineNumber();
38     line = in.readLine();
40     if (line != null)
41         out.printf("Zeile %d: %s\n", n, line);
42 } while (line != null);
44 in.close();
```

 `TextFilterExamples.java`

## Notizen



# Inhalt

# Text Ein- und Ausgabe

## Formatierte Textausgabe

# Notizen

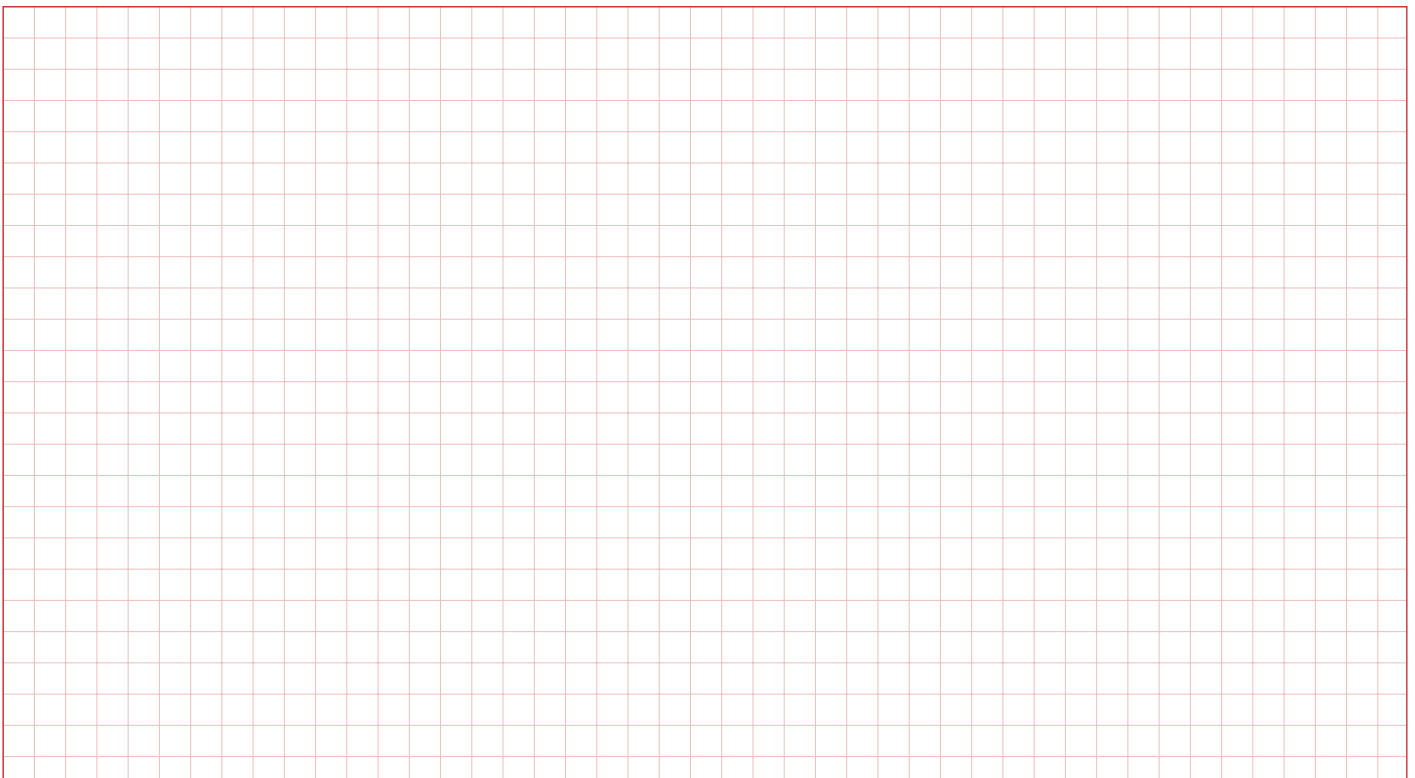
## Formatierte Textausgabe

- ▶ Wie können wir Java-Datentypen „**menschenlesbar**“ **ausgeben**?
- ▶ **Antwort** kennen wir eigentlich schon

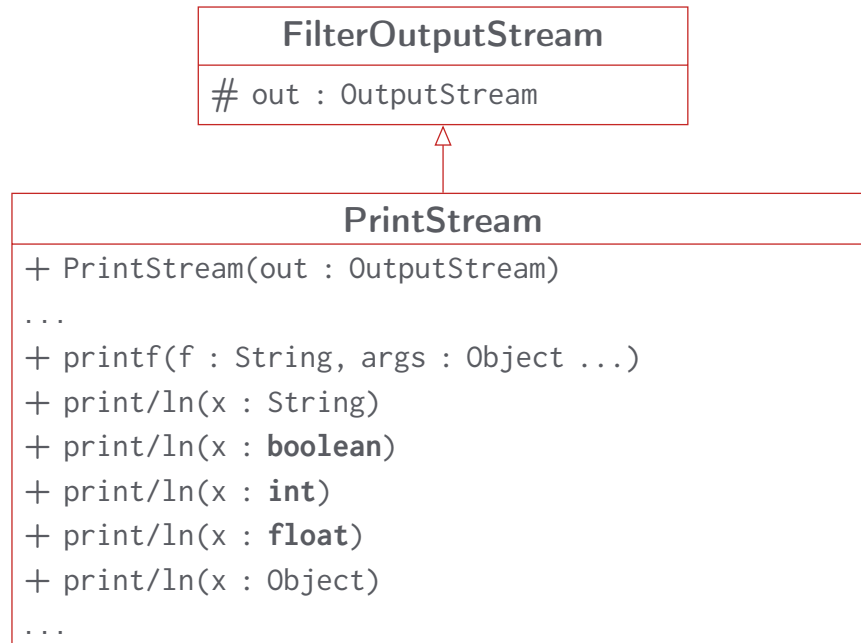
```
System.out.printf("i=%d, pi=%f, s=%s",  
42, Math.PI, someString);
```

- ▶ **Bisher** Ausgabe in **Ausgabestrom** [↗](#) `System.out`
- ▶ [↗](#) `System.out` ist vom Typ [↗](#) `PrintStream`
- ▶ **Jetzt** allgemeiner in [↗](#) `OutputStream` und [↗](#) `Writer`
  - ▶ Formatierte Ausgabe in **Dateien**, **Puffer**, **Netzwerk**, etc.
  - ▶ Zwei Klassen
    - ▶ [↗](#) `PrintStream` für [↗](#) `OutputStream`
    - ▶ [↗](#) `PrintWriter` für [↗](#) `Writer`

## Notizen

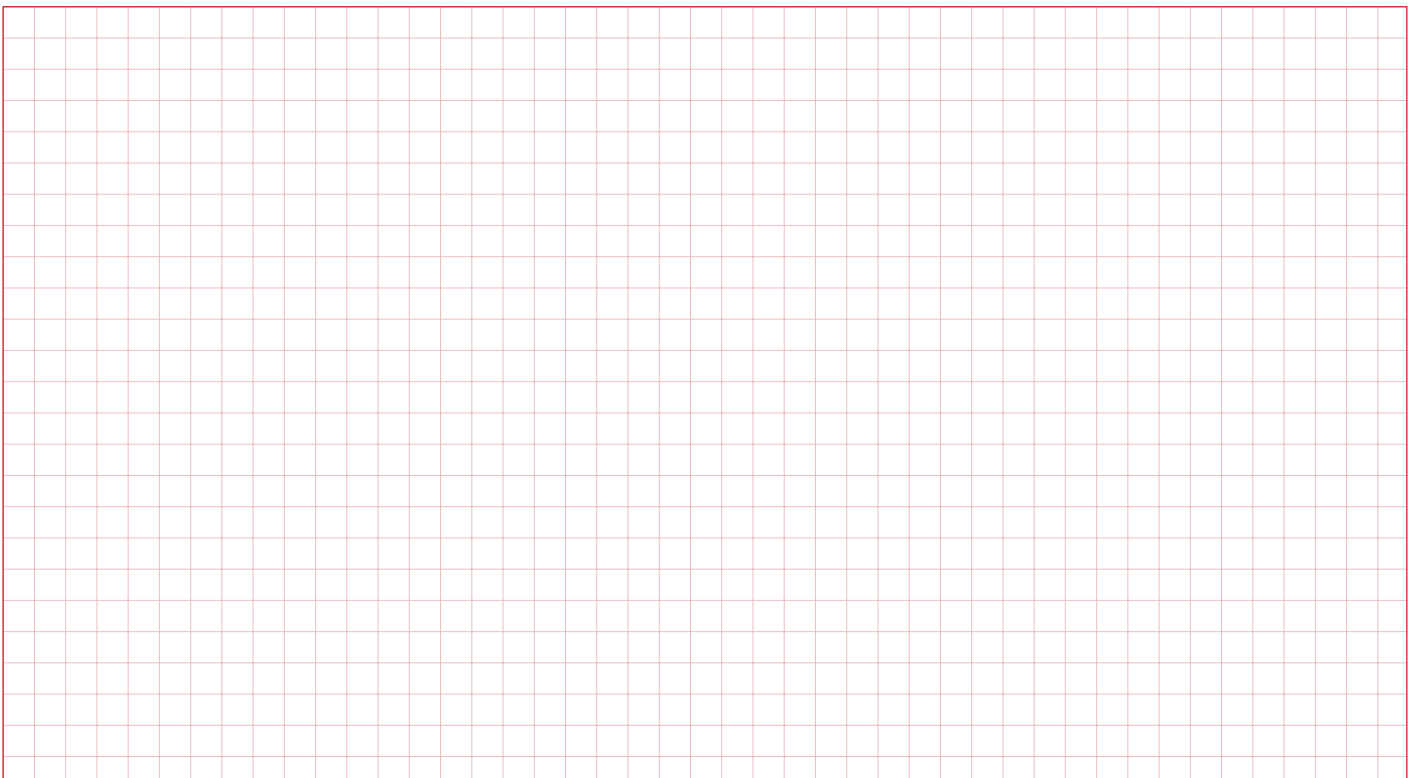


## PrintStream

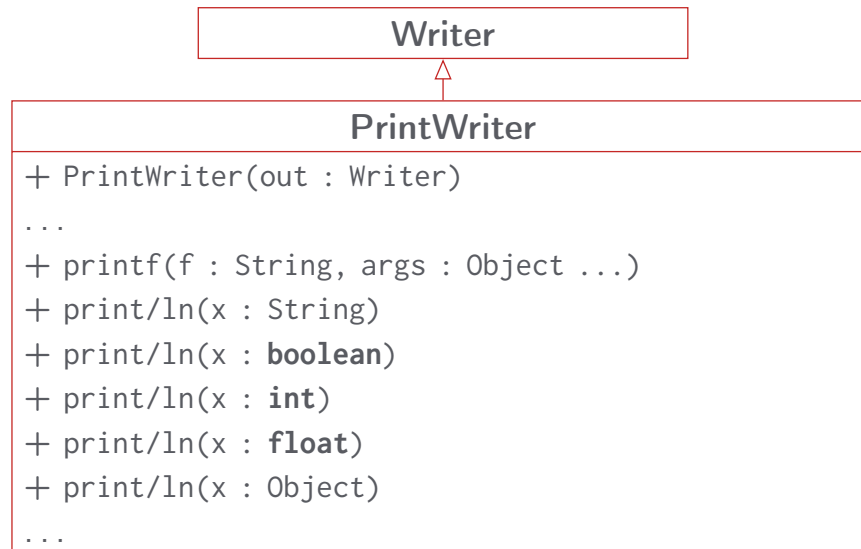


- ▶ Für [OutputStreams](#) (z.B. [System.out](#))
- ▶ Optional: [Charset](#) (Default: [Charset.defaultCharset\(\)](#))
- ▶ Funktionsweise `print/f/ln`: siehe [Grundlagenkapitel](#)!

## Notizen

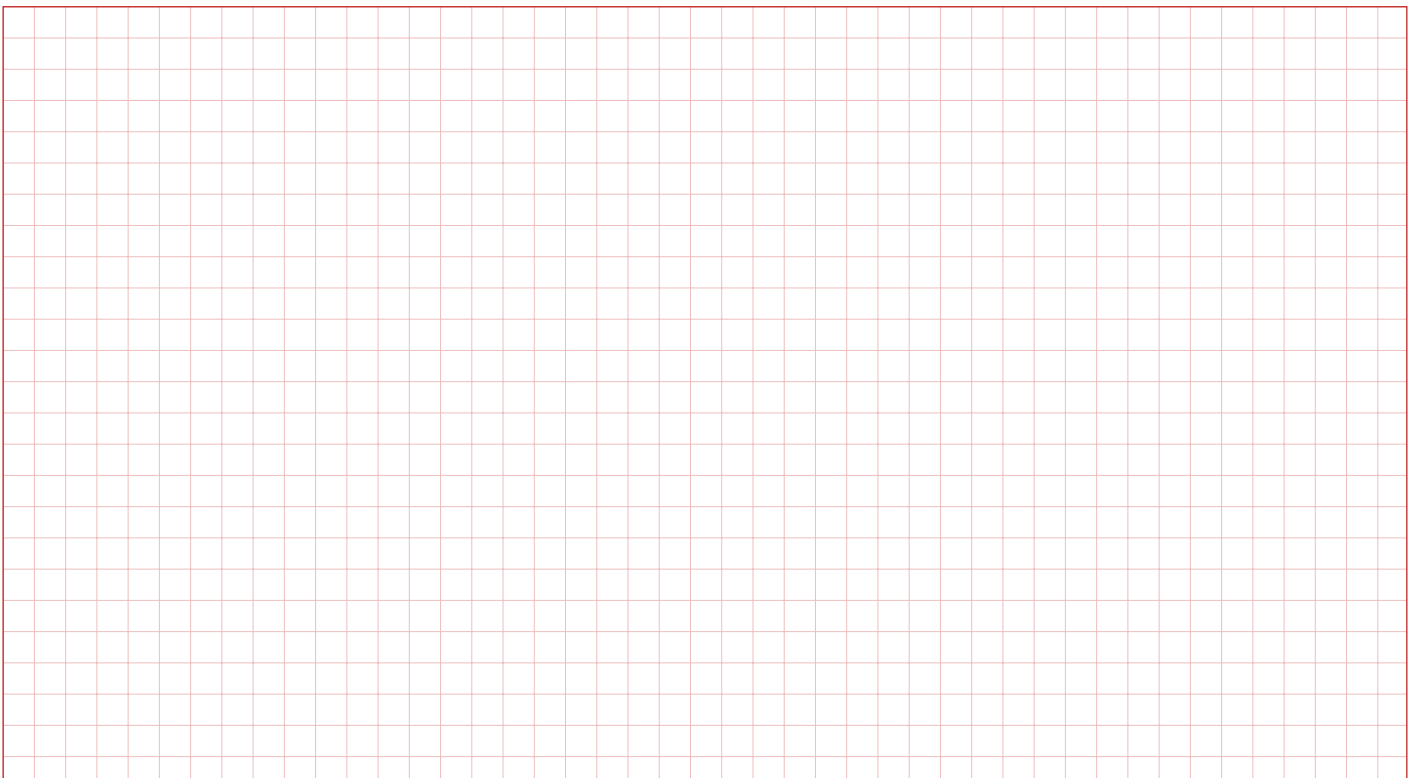


## PrintWriter



- ▶ Für [OutputStreams](#) und [Writer](#)
- ▶ Gleiche Methoden wie [PrintStream](#)
- ▶ Optional: [Charset](#) für [OutputStream](#)

## Notizen



## Beispiel PrintStream: CSV-Datei

- ▶ Beispiel aus **Collections**-Kapitel: **Bestände von Items**

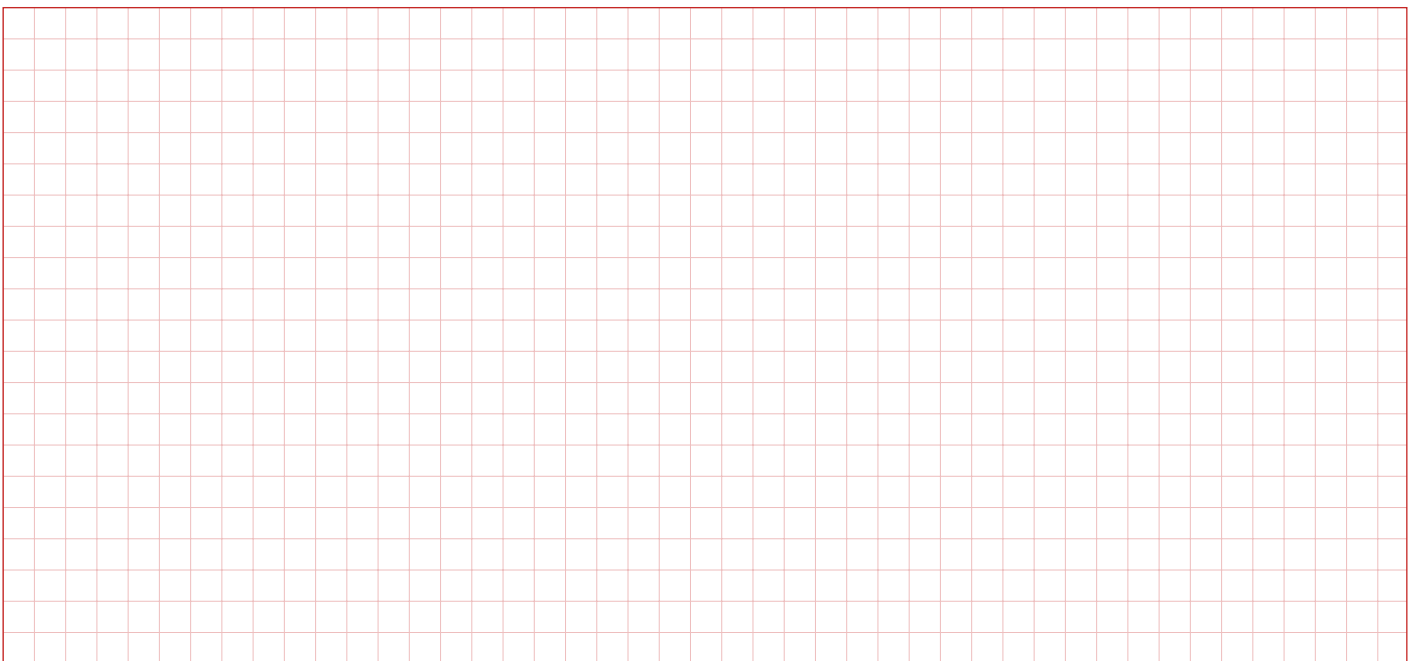
```
var salad = new Item("Salat", 2);  
var choc = new Item("Schokolade", 1);  
var milk = new Item("Milch", 2);  
var toiletpaper = new Item("Toilettenpapier", 3);  
var stock = Map.of(  
    salad, 10,  
    choc, 50,  
    milk, 30,  
    toiletpaper, 2);
```

- ▶ Wir sollen stock in einer **CSV-Datei** speichern

```
Salat;2;10  
Schokolade;1;50  
Milch;2;30  
Toilettenpapier;3;2
```

### Notizen

- CSV steht für „comma separated values“



## Beispiel PrintStream: CSV-Datei

► exportToCSV schreibt [Map<Item,Integer>](#) in [OutputStream](#)

```
17 public static void exportToCSV(Map<Item,Integer> stock,
18     OutputStream outputStream) throws IOException {
19     var out = new PrintStream(outputStream);
20     for (var entry : stock.entrySet()){
21         Item item = entry.getKey();
22         int amount = entry.getValue();
23
24         out.printf("%s;%d;%d\n",
25             item.getName(),
26             item.getPrice(),
27             amount);
28     }
29     out.close();
30 }
31
32 }
```

[PrintExamples.java](#)

## Notizen



## Beispiel `PrintStream`: CSV-Datei

- ▶ Funktionsweise von `exportToCSV`

- ▶ `OutputStream` wird in `PrintStream` verpackt

```
var out = new PrintStream(outputStream);
```

Encoding: `Charset.defaultCharset()`

- ▶ Einträge von `stock` werden durchlaufen

```
for (var entry : stock.entrySet()){  
    Item item = entry.getKey();  
    int amount = entry.getValue();
```

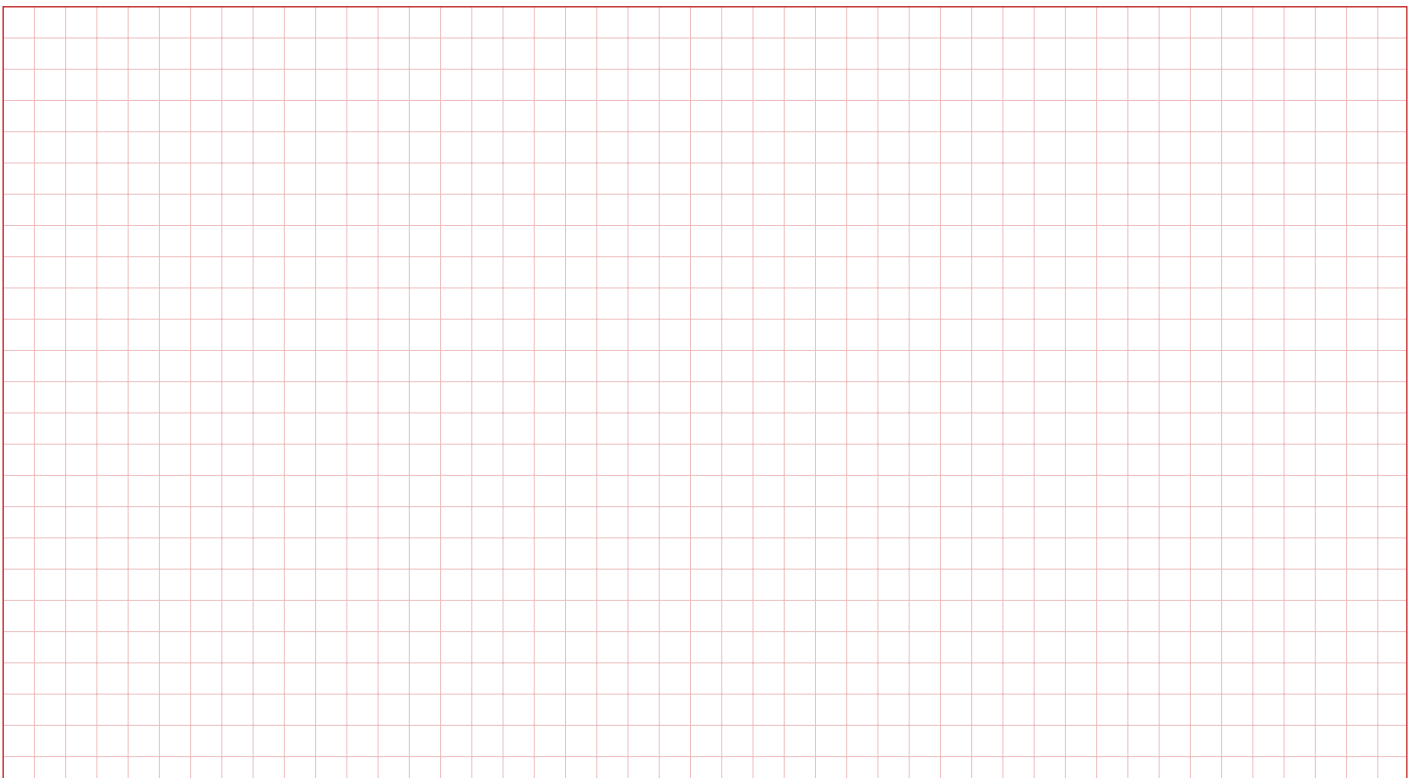
- ▶ Formatierte Ausgabe mit `printf`

```
out.printf("%s;%d;%d\n",  
    item.getName(),  
    item.getPrice(),  
    amount);
```

- ▶ Hinweis

- ▶ `exportToCSV(stock, System.out)` würde CSV auf Standardausgabe ausgeben

## Notizen





## Beispiel PrintWriter: JSON-Datei

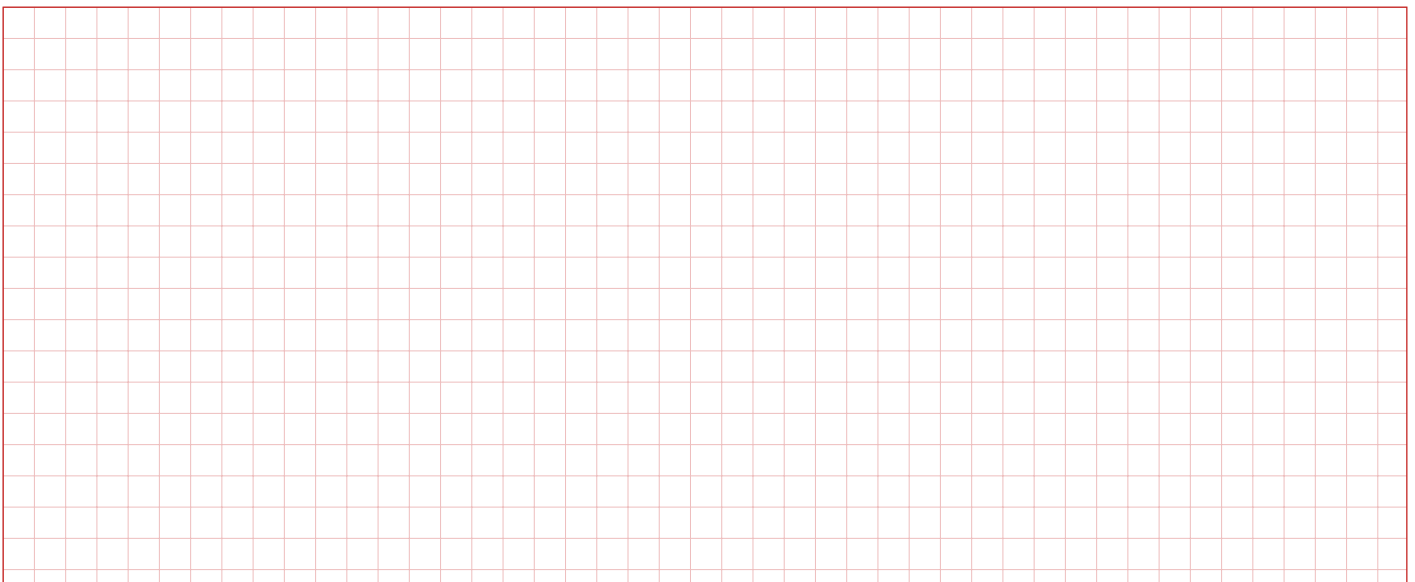
- ▶ Wir wollen `PrintWriter` verwenden um stock in eine **JSON-Datei** schreiben
- ▶ Ergebnis

```
{ "stock":  
  [  
    { "name" : "Salat", "price": 2, "amount": 10 },  
    { "name" : "Schokolade", "price": 1, "amount": 50 },  
    { "name" : "Milch", "price": 2, "amount": 30 },  
    { "name" : "Toilettenpapier", "price": 3, "amount": 2 }  
  ]  
}
```

- ▶ Siehe `exportToJson` in `PrintExamples.java`

## Notizen

- JSON steht für „JavaScript Object Notation“ und wurde ursprünglich verwendet um Objekte in der Sprache JavaScript in Menschen lesbarer Form als Text abzulegen oder zu übertragen. Mittlerweile wird JSON auch in anderen Programmiersprachen verwendet.



## Beispiel PrintWriter: JSON-Datei

Funktionsweise von exportToJson

- ▶ Ähnlich zu exportToCSV
- ▶ Schreibt in [Writer](#) statt [OutputStream](#)
- ▶ Verpackt [Writer](#) in [PrintWriter](#)

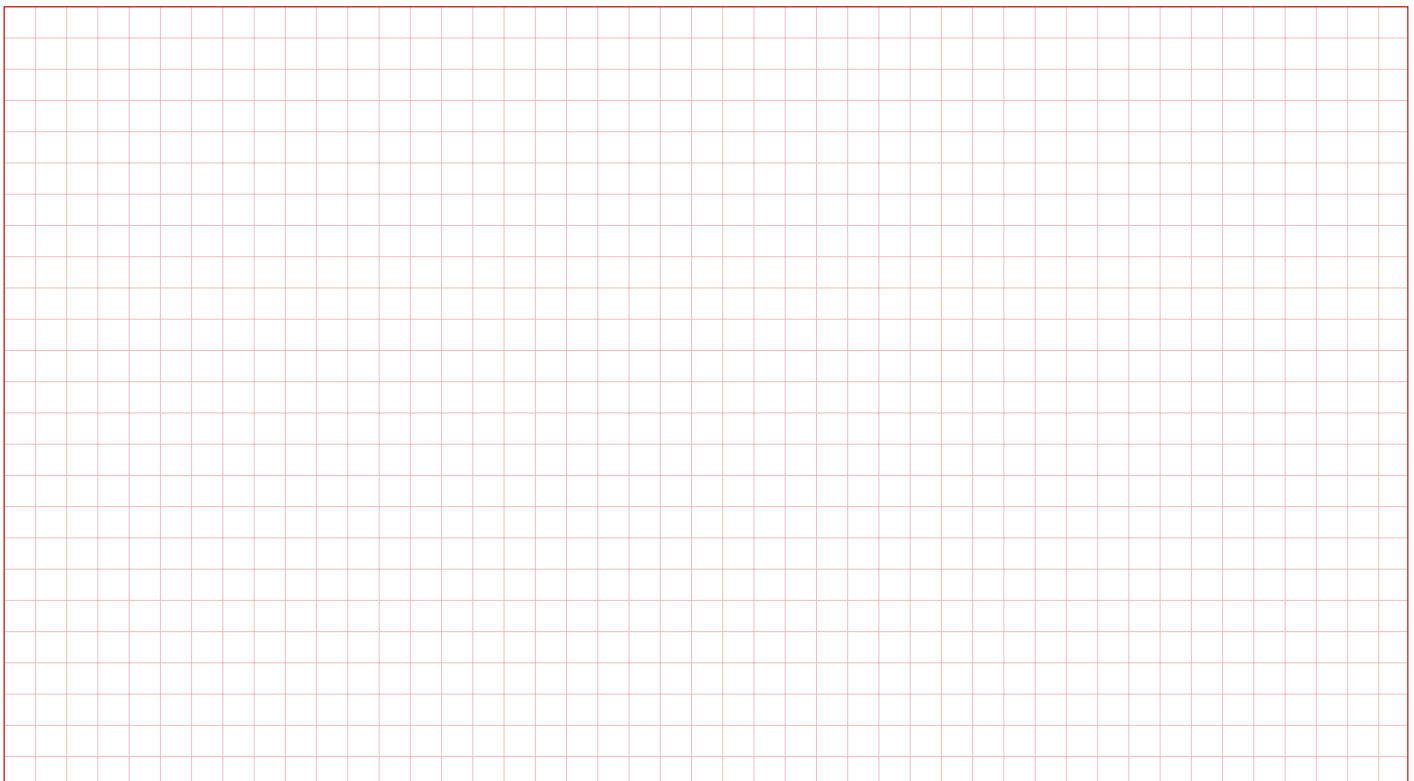
```
var out = new PrintWriter(writer);
```

- ▶ Generiert **JSON-Ausgabe**

```
/* ... */  
out.printf(" { \"name\" : \"%s\", \"price\": %d, \"amount\": %d }",  
    item.getName(),  
    item.getPrice(),  
    amount);  
/* ... */
```

- ▶ Mühsam! **JSON-Library** verwenden!

## Notizen



# Inhalt

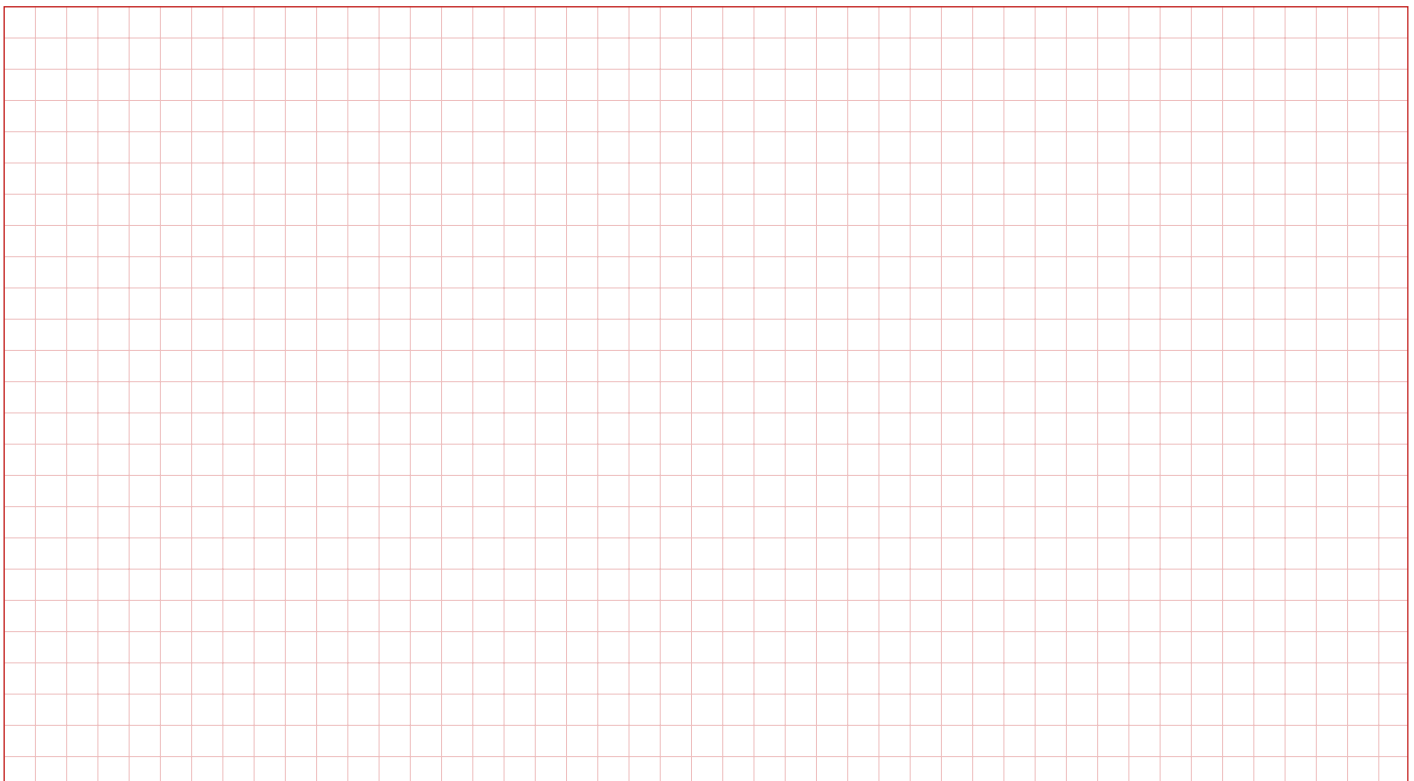
## Text Ein- und Ausgabe

Einlesen von Daten: „Parsing“

Parsing zu Fuß

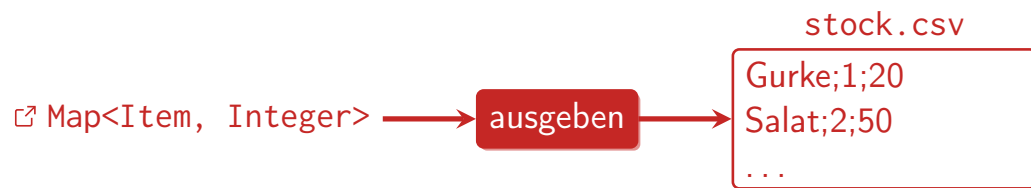
Einlesen mit Scanner

## Notizen

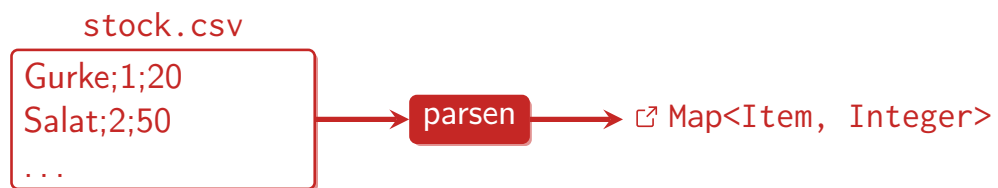


# Parsing

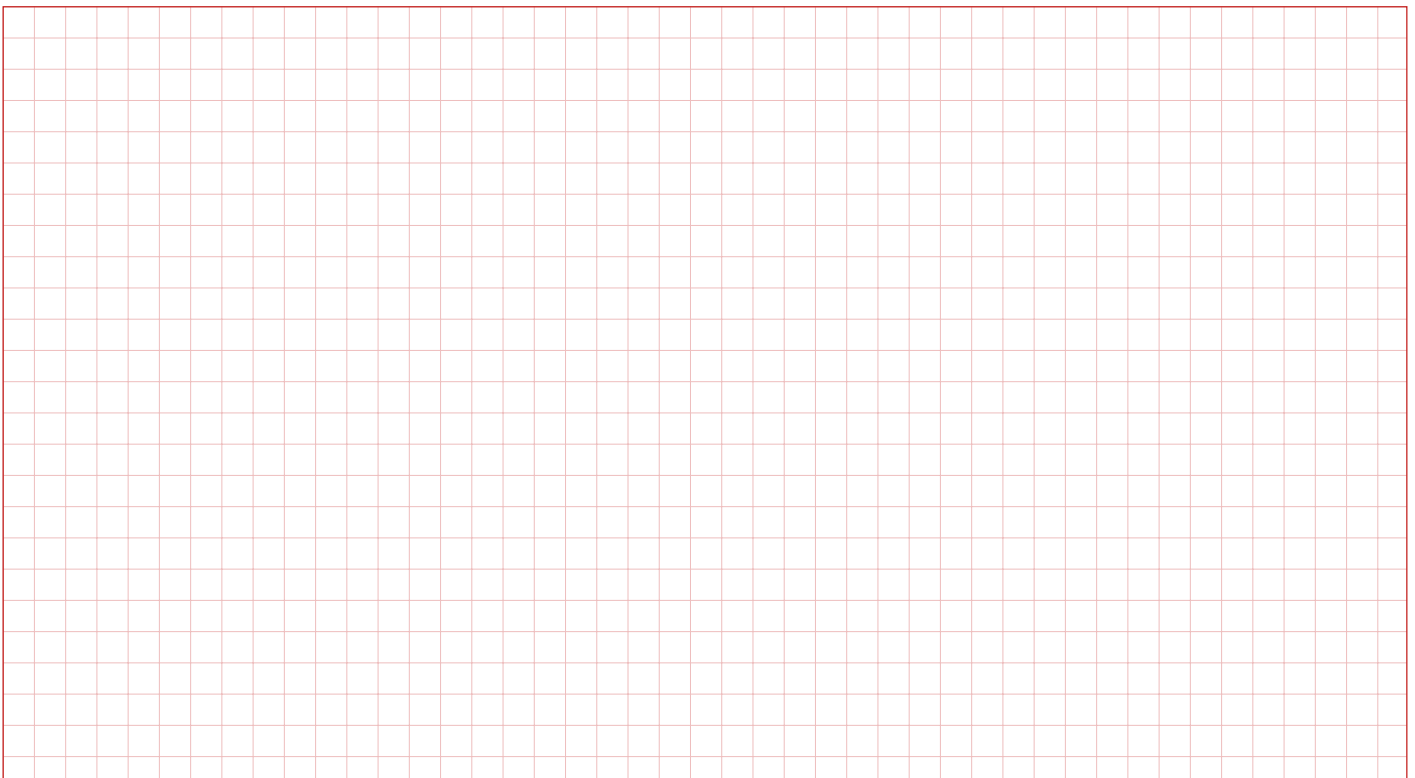
- Bisher: Formatierte Ausgabe



- Jetzt: Einlesen aus Textformat („parsing“)



## Notizen



# Inhalt

## Text Ein- und Ausgabe

## Einlesen von Daten: „Parsing“

# Parsing zu Fuß

# Notizen

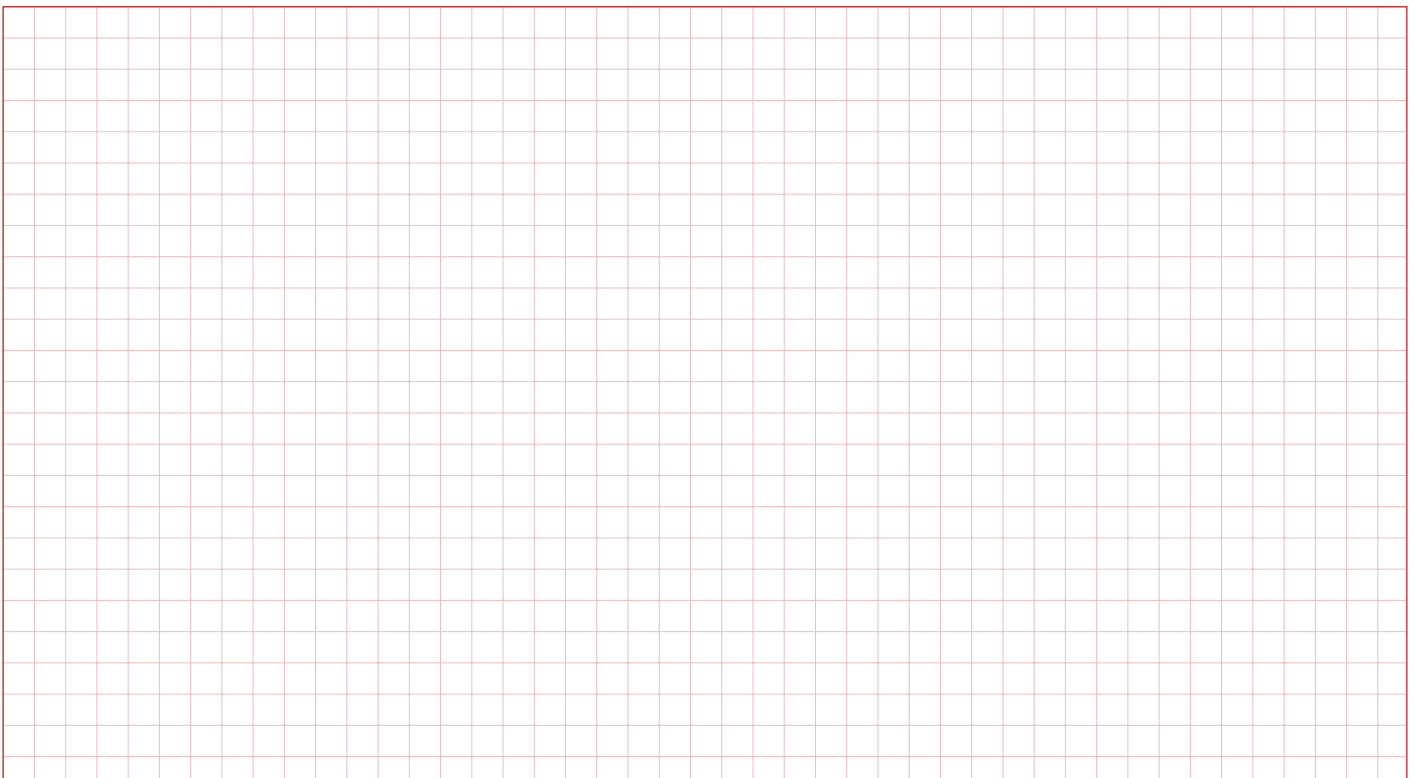
# Parsing zu Fuß

- ▶ Manuelles Parsen mit
  - ▶ Lesen des Inhalts in `String`
  - ▶ Meist zeileweise mit `BufferedReader`
  - ▶ `parse-Methoden` der Wrapperklassen

```
Integer.parseInt(String s)
Double.parseDouble(String s)
...
```

- ▶ `String-Methoden`
  - ▶ `indexOf` und Co. — suchen im `String`
  - ▶ `trim` — „whitespaces“ am Anfang/Ende entfernen
  - ▶ `split` — Aufteilen an Trennzeichen
  - ▶ ...
- ▶ Nur für einfache Formate!

## Notizen



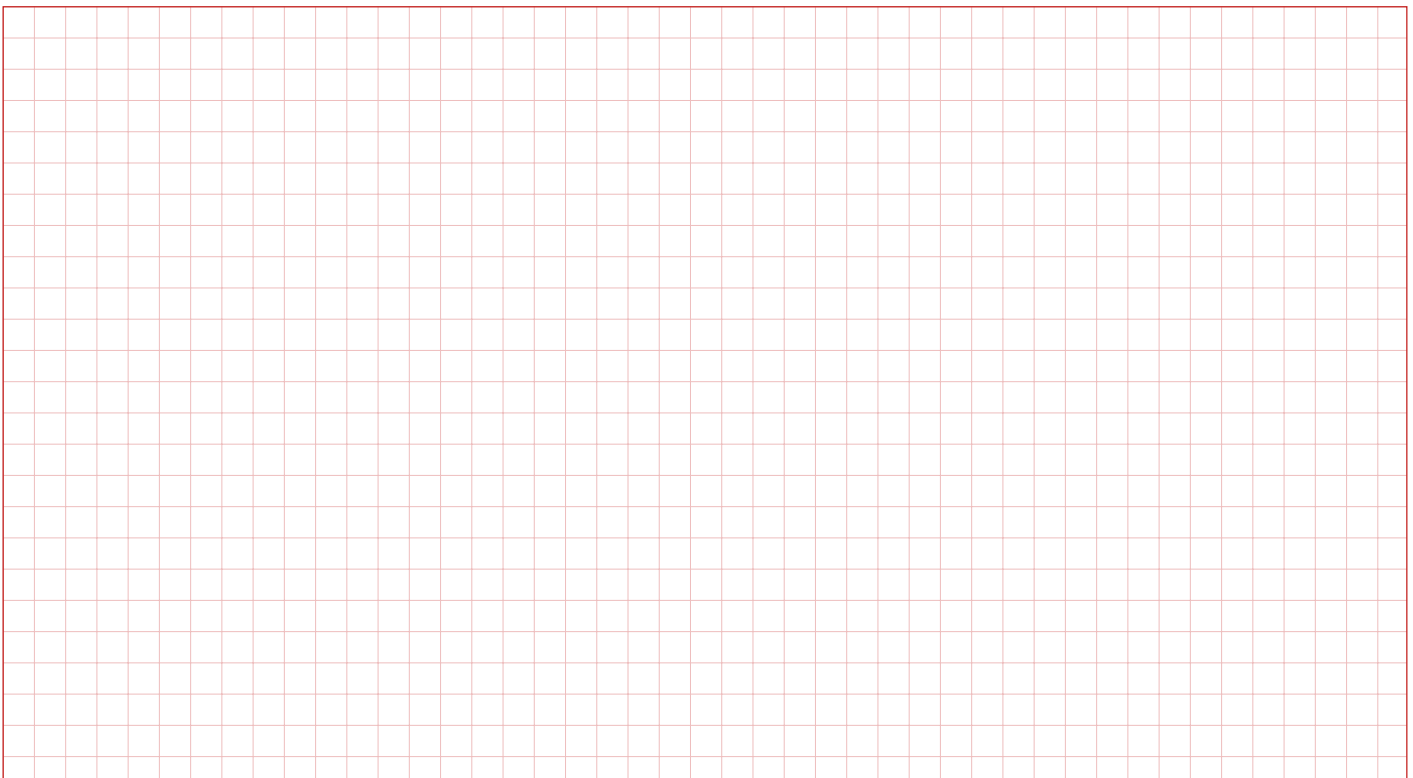
## Beispiel: CSV einlesen

- ▶ Zur Erinnerung: CSV-Datei von vorher

```
Salat;2;10  
Schokolade;1;50  
Milch;2;30  
Toilettenpapier;3;2
```

- ▶ Ziel: Einlesen in `Map<Item, Integer>`
- ▶ Lösung:
  - ▶ `parseCSV` und `parseCSVExample` in `ParseExample.java`
  - ▶ Ausführen mit `runParseCSVExample` (vorher `runExportToCSVExample` ausführen!)
- ▶ Funktionsweise von `parseCSV`
  - ▶ Datei mit `BufferedReader` öffnen
  - ▶ Zeile für Zeile lesen
  - ▶ Zeile an `;` auftrennen (`String.split`)
  - ▶ Werte parsen
    - ▶ Name: `String` übernehmen
    - ▶ Preis/Anzahl: `int` mit `Integer.parseInt` einlesen

## Notizen



# Inhalt

# Text Ein- und Ausgabe

## Einlesen von Daten: „Parsing“

## Einlesen mit Scanner

# Notizen



# Scanner

## ► Zur Erinnerung

- [Scanner](#) bisher zum Einlesen von Benutzereingaben

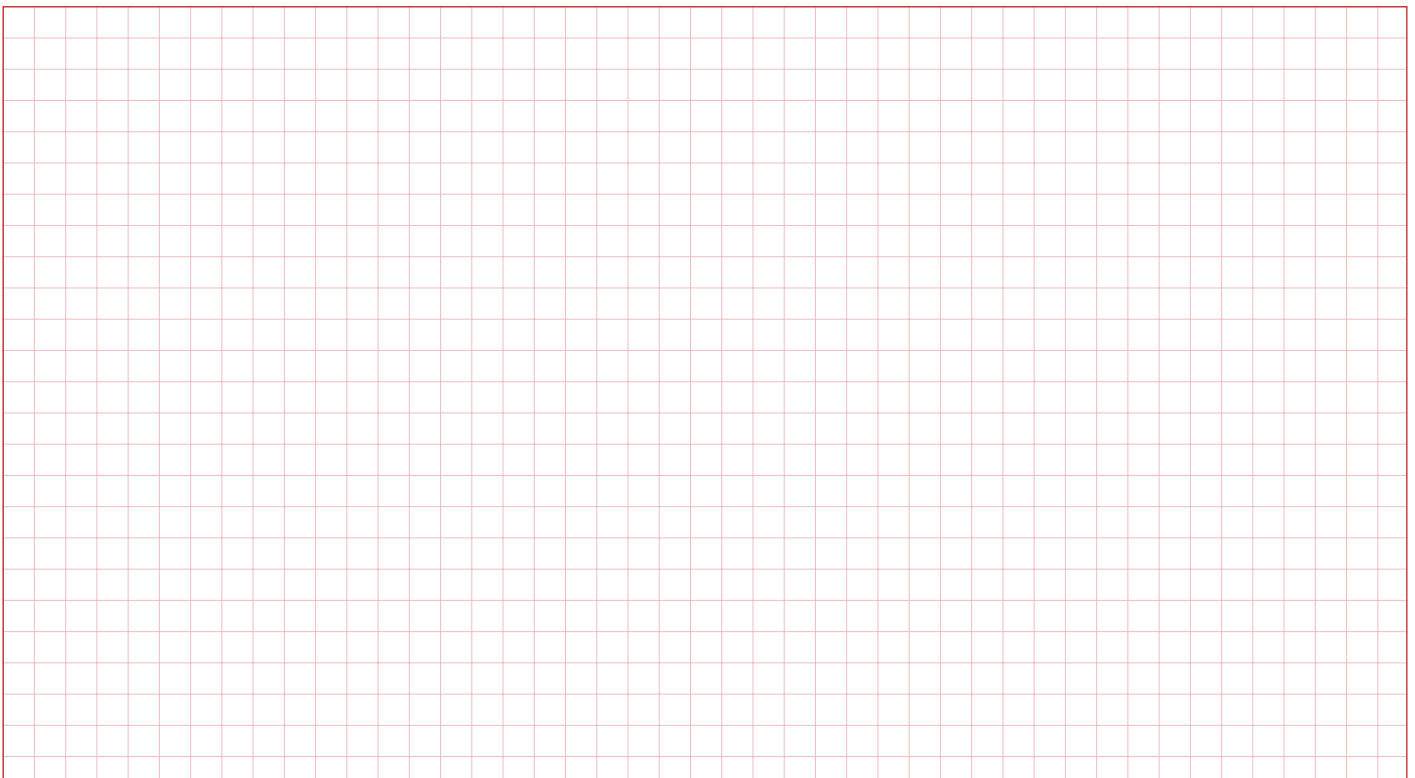
```
scanner.nextInt()  
scanner.nextDouble()
```

- [Scanner](#) funktioniert auf [InputStream](#)

## ► Vorteil: [Scanner](#) kann Trennelemente berücksichtigen

- [Scanner.useDelimiter\(String regex\)](#) verwendet regex als Trennelement
- Allgemein regulärer Ausdruck
- Hier: `scanner.useDelimiter("(;|\\R)")`
- Entspricht: ; oder ein Zeilenvorschub

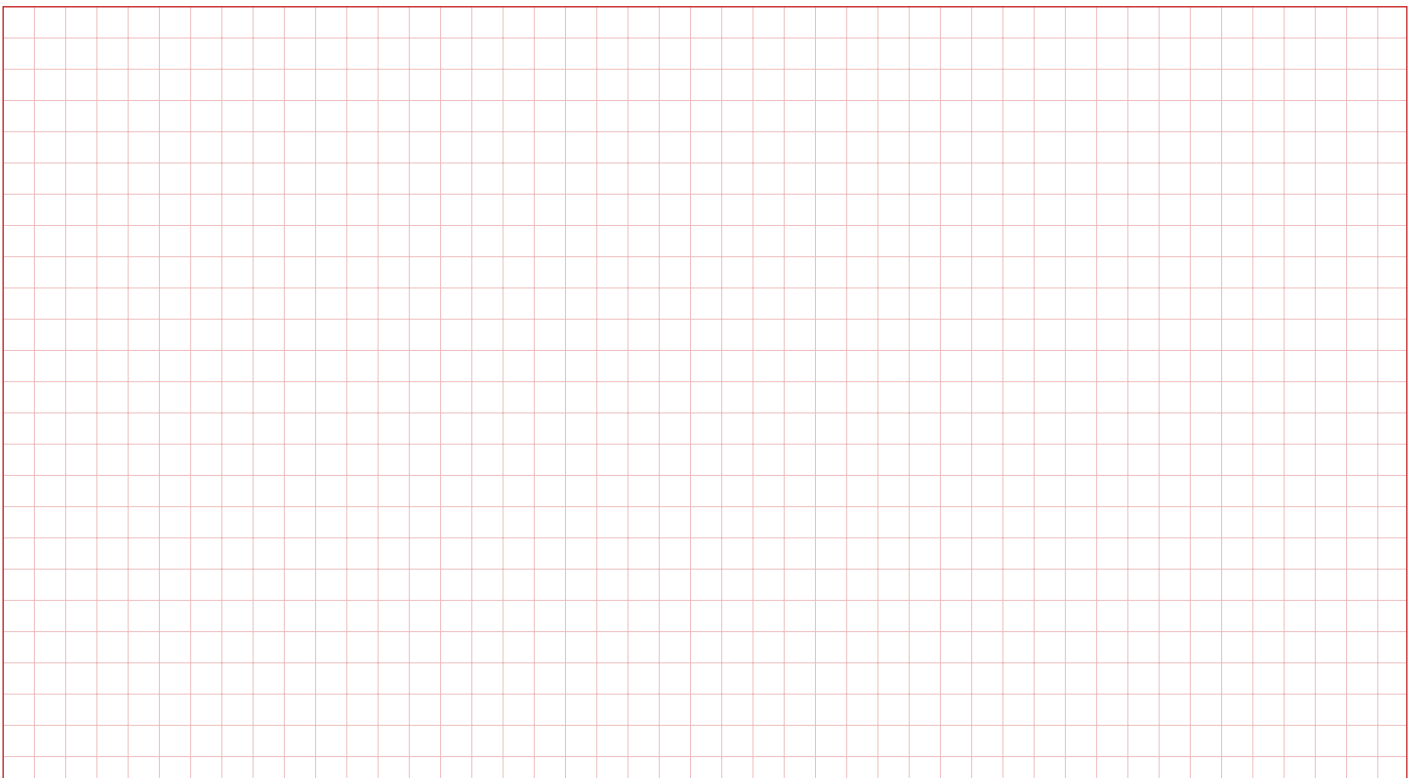
## Notizen



## Einlesen mit Scanner

- ▶ Lösung:
  - ▶ `parseCSVScanner` und `parseCSVScannerExample` in `ParseExample.java`
  - ▶ Ausführen mit `runParseCSVScannerExample` (**vorher** `runExportToCSVExample` **ausführen!**)
- ▶ Funktionsweise von `parseCSVScanner`
  - ▶ Datei als `FileInputStream` öffnen
  - ▶ `FileInputStream` in `Scanner` „verpacken“
  - ▶ Trennelement auf `"(;|\\R)"` setzen
  - ▶ Werte parsen
    - ▶ `Name` mit `hasNext()` und `next()`
    - ▶ `Preis/Anzahl` mit `hasNextInt()` und `nextInt()`
- ▶ Was ist „schöner“?
  - ▶ `split`-Variante?
  - ▶ `Scanner`-Variante?
- ▶ Beides **nicht toll** — am besten fertige Libraries verwenden

## Notizen



# Inhalt

# Text Ein- und Ausgabe

## Zusammenfassung

# Notizen

# Zusammenfassung



- ▶ **Code:** Zeichen → Zahl (Code)
- ▶ **Encoding:** Code → Bytesequenz
- ▶ **Java**
  - ▶ **Code:** Unicode
  - ▶ **Encoding:** UTF-16
- ▶ **Charset**
  - ▶ Implementiert **Encodings**
  - ▶ **Verfügbare Encodings**
- ▶ **Hinweis:**
  - ▶ Auf **Encodings** achten!
  - ▶ Besonders bei **Dateien**

## Notizen



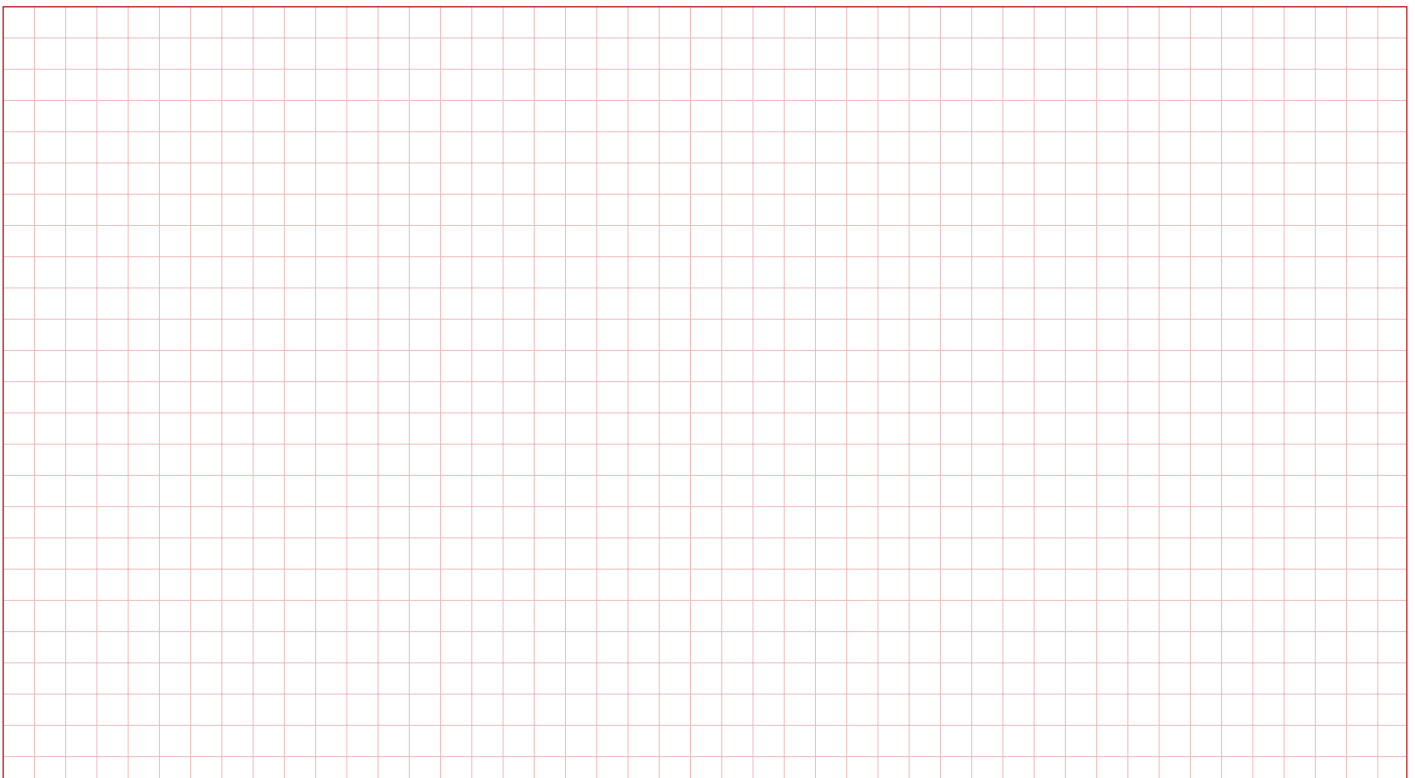
# Zusammenfassung

<<abstract>>  
**Reader**

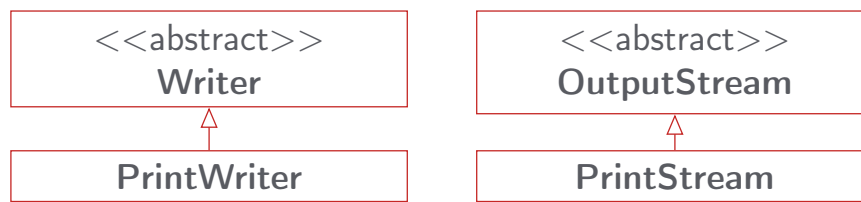
<<abstract>>  
**Writer**

- ▶ [Reader](#)/[Writer](#) sind ähnlich zu `InputStream`/`OutputStream` mit:
  - ▶ `char[]` statt `byte[]`
  - ▶ Encoding
- ▶ Filter wie bei `InputStream`/`OutputStream`
  - ▶ [BufferedReader](#)/[BufferedWriter](#) — **zeilweises** Lesen/Schreiben
  - ▶ [LineNumberReader](#) — lesen mit **Zeilennummern**
- ▶ Quellen/Senken wie bei **Byteströmen**
- ▶ Brückenklassen
  - ▶ [InputStreamReader](#) ist ein [Reader](#) der aus [InputStream](#) liest
  - ▶ [OutputStreamWriter](#) ist ein [Writer](#) der in [OutputStream](#) schreibt

## Notizen

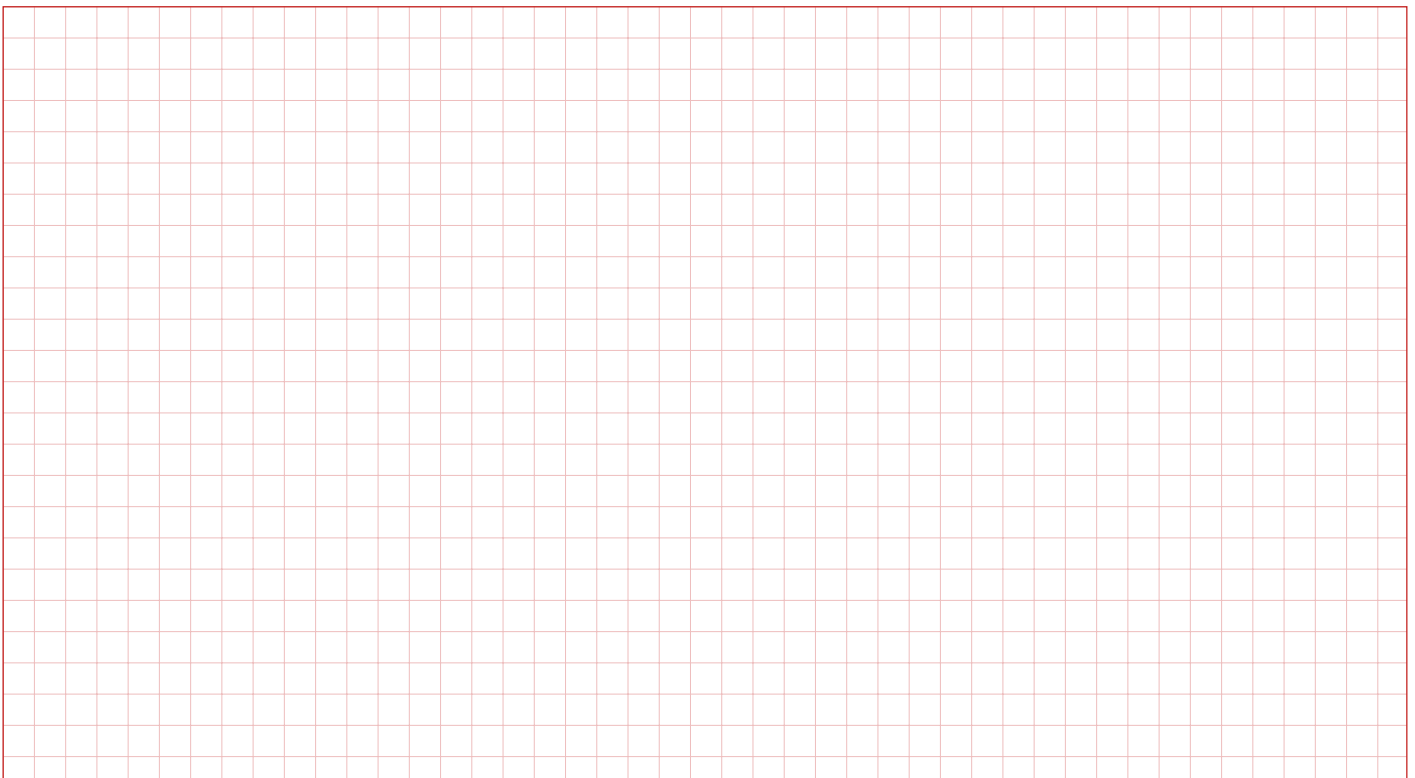


## Zusammenfassung



- ▶ **Formatierte Ausgabe**
  - ▶ `PrintWriter` für `Writer`
  - ▶ `PrintStream` für `OutputStream`
- ▶ **Wichtige Methoden**
  - ▶ `print` bzw. `println` (mit **neuer Zeile**)
  - ▶ `printf`
- ▶ **Parsing**
  - ▶ Zu Fuß: Wrapper-parse- und `String`-Methoden
  - ▶ Alternative/Hilfe: `Scanner`
  - ▶ Externe Library

## Notizen



# Inhalt

## Automatic Resource Management

- Motivation
- try mit Ressourcen

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

# Inhalt

# Automatic Resource Management

# Motivation

# Notizen



## Ein Geständnis

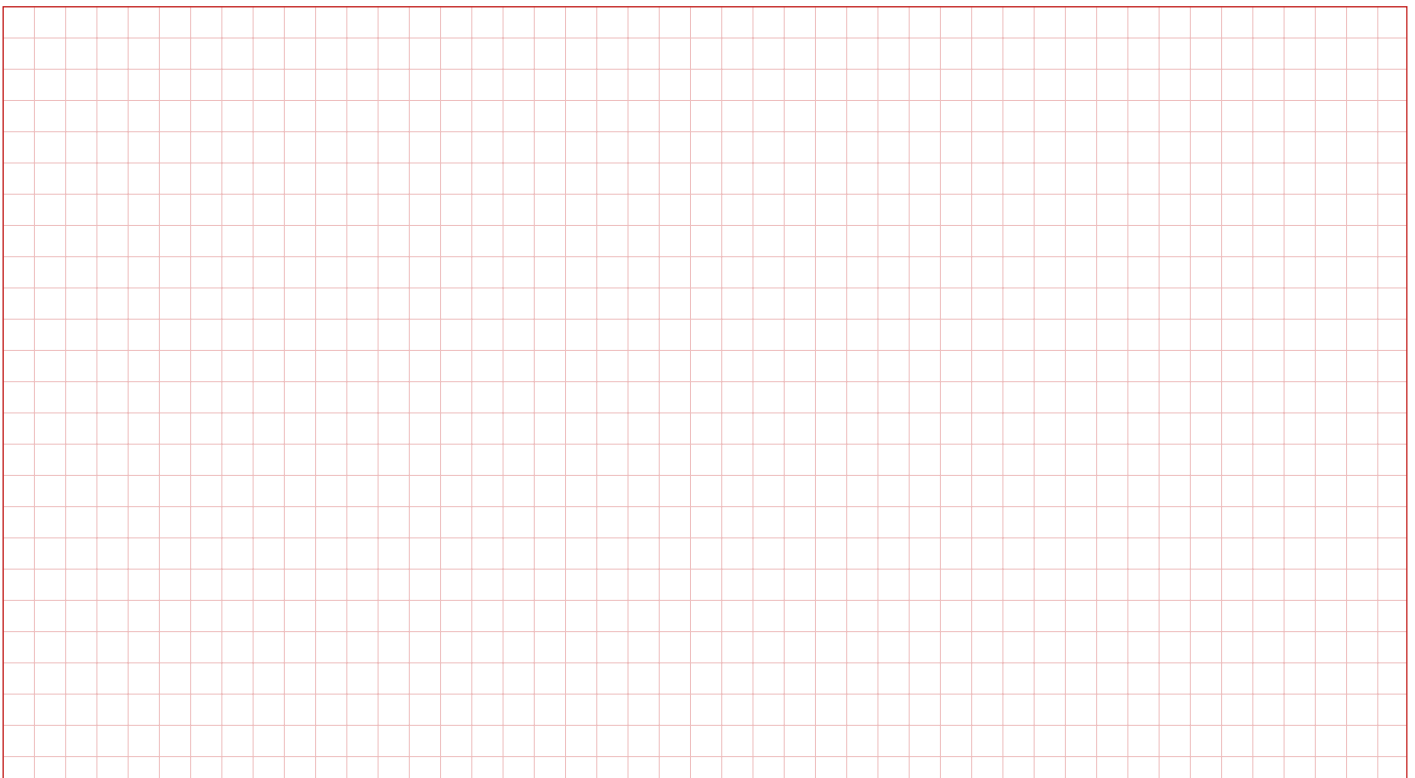
- ▶ **Geständnis:** Alle bisherigen Beispiel waren **sehr unsauber!**

```
12 public static void copyFile(String from, String to)
13     throws IOException {
14     var in = new FileInputStream(from);
15     var out = new FileOutputStream(to);
17     in.transferTo(out);
19     in.close();
20     out.close();
21 }
```

ARMExamples.java

- ▶ **new** FileIn/OutputStream **belegt** Betriebssystem-Ressourcen
- ▶ **Manuelle Freigabe** über `close()`
- ▶ **Jeder Methodenaufruf** kann [↗](#) `IOException` werfen
- ▶ → `close()` wird eventuell **nicht** aufgerufen

## Notizen



## „Lösung“

### ► „Lösung“ mit **finally**

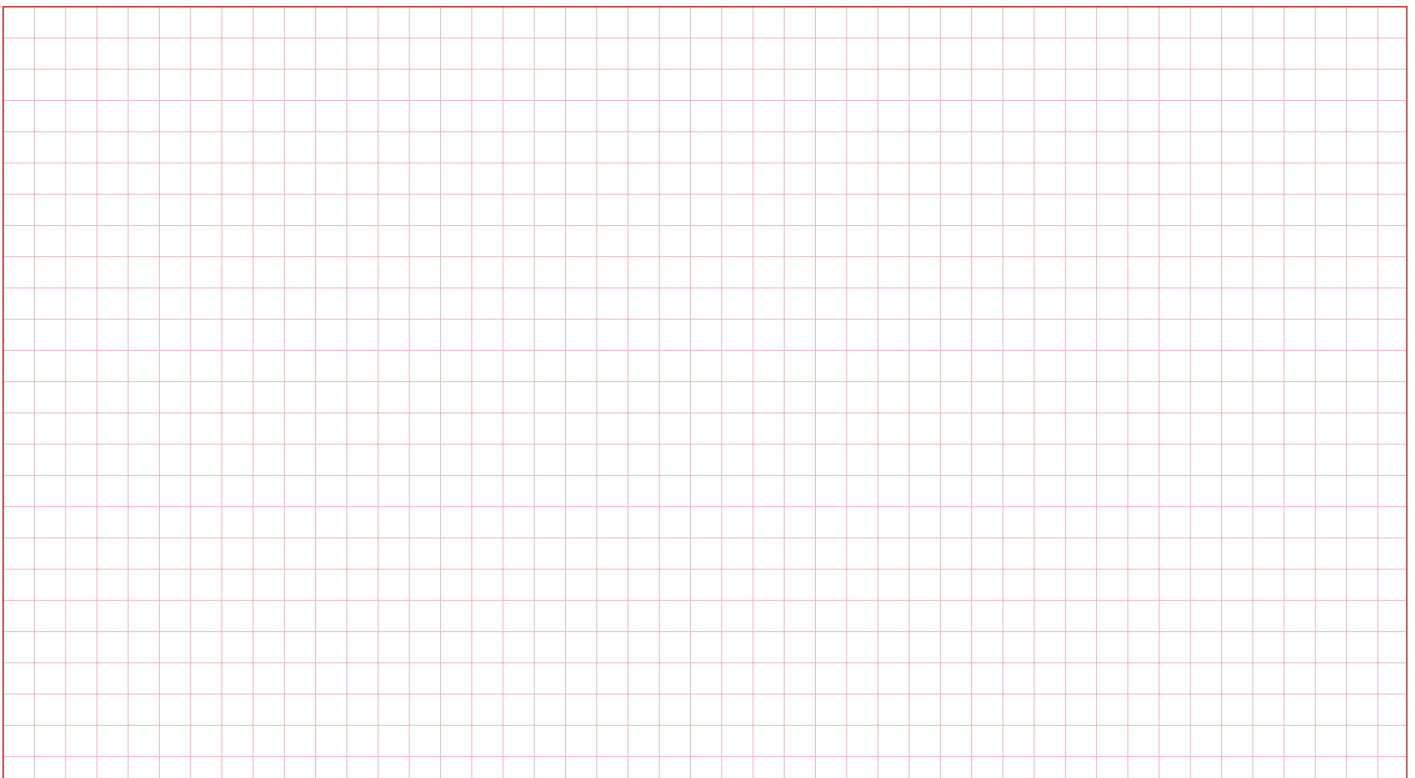
```
25 public static void copyFileWithFinally(  
26     String from, String to) throws IOException {  
27     FileInputStream in = null;  
28     FileOutputStream out = null;  
29     try{  
30         in = new FileInputStream(from);  
31         out = new FileOutputStream(to);  
32         in.transferTo(out);  
33     } finally {  
34         if (in != null) in.close();  
35         if (out != null) out.close();  
36     }  
37 }
```

ARMExamples.java

### ► Probleme

- **Unschön**: Mehr und aufwändigerer Code
- close() kann auch [☞ IOException](#) werfen!

## Notizen



## „Lösung“?

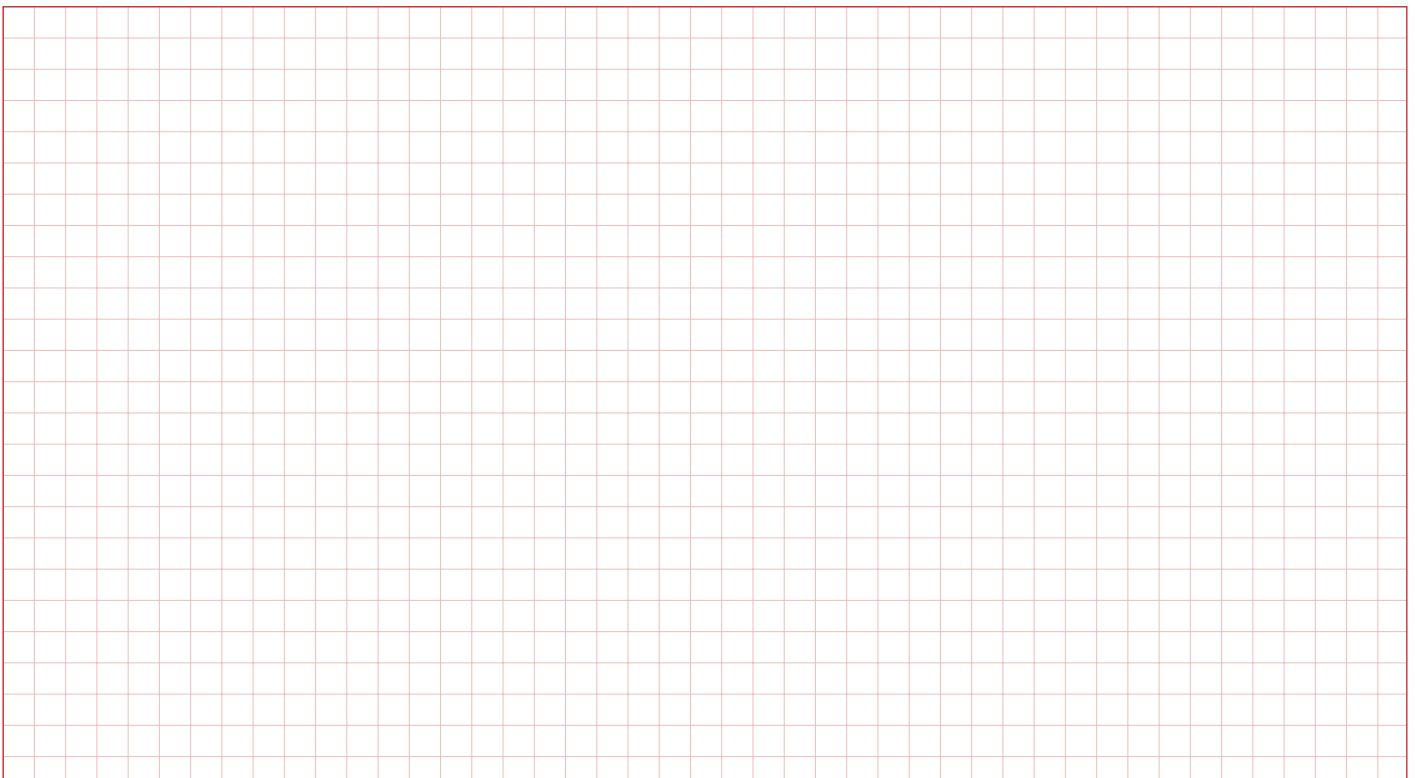
### ► „Lösung“?

```
45 try{
46     in = new FileInputStream(from);
47     out = new FileOutputStream(to);
48     in.transferTo(out);
49 } finally {
50     try{
51         if (in != null) in.close();
52         if (out != null) out.close();
53     } finally {
54         if (in != null) in.close();
55         if (out != null) out.close();
56     }
57 }
```

ARMExamples.java

- Noch **aufwändigerer** Code!
- Was ist wenn **close()** in zweitem **finally** [↗](#) **IOException** wirft?
- Wir brauchen eine **grundsätzliche** Lösung!

## Notizen



# Inhalt

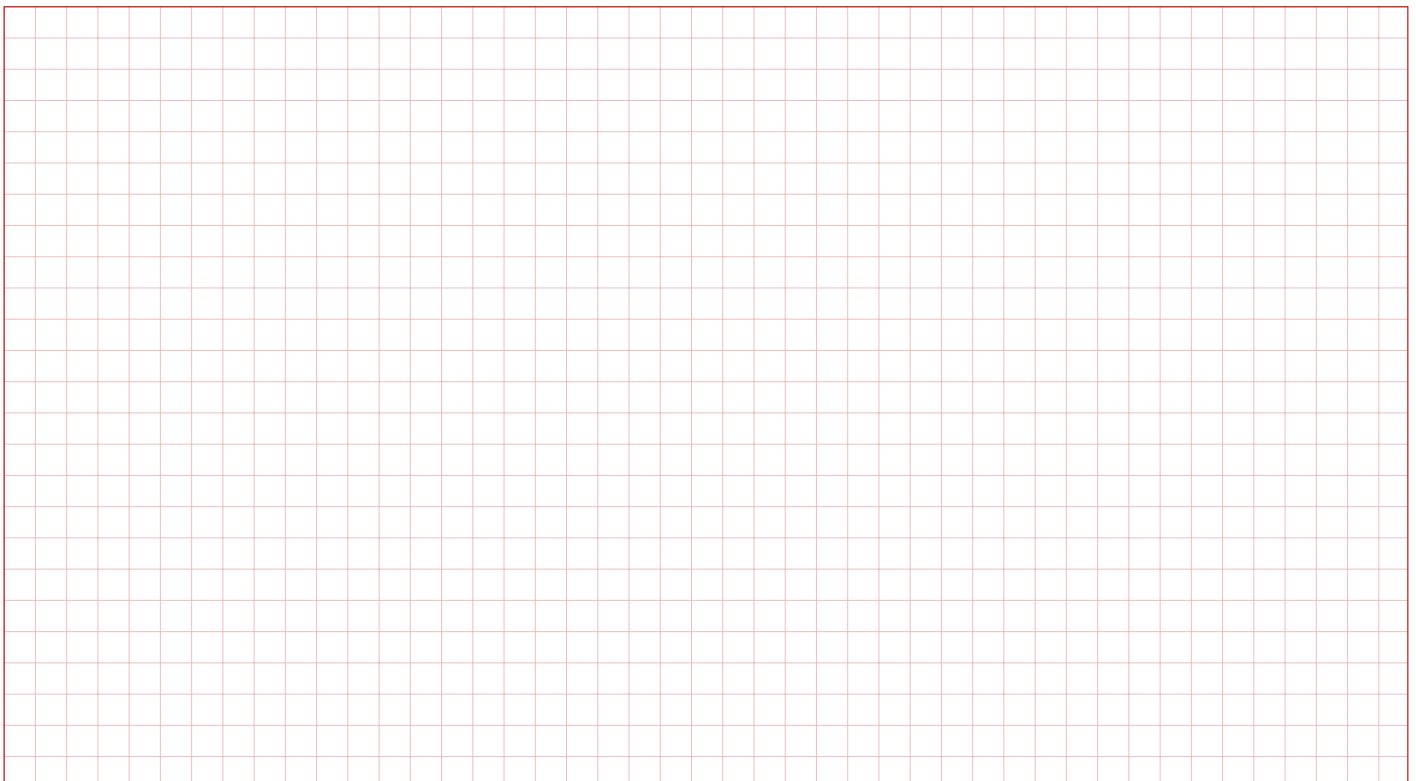
## Automatic Resource Management

- try mit Ressourcen

  - Behandlung von Ausnahmen

  - AutoCloseable and Closeable

## Notizen



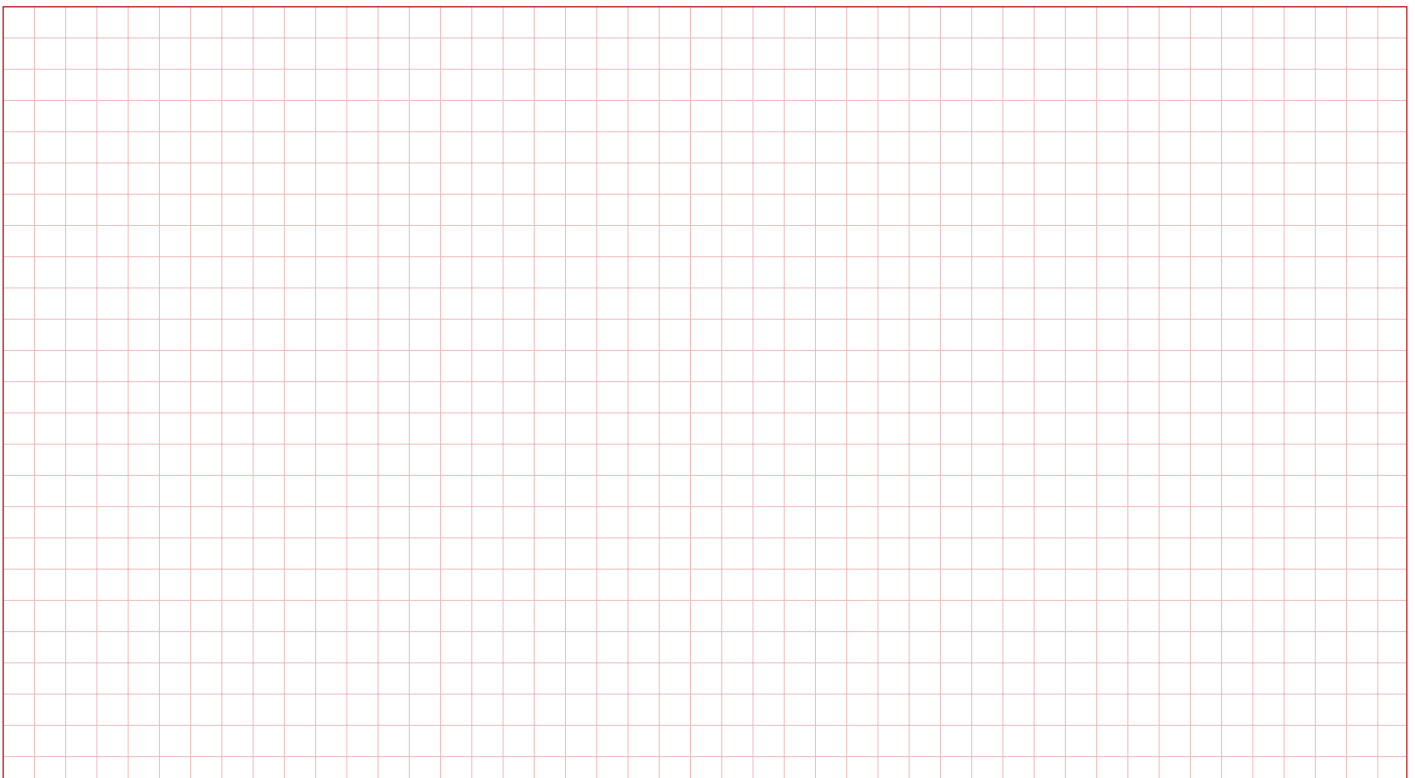
## try mit Ressourcen

- ▶ „Ressourcen“
  - ▶ Objekte mit Betriebssystem-Ressourcen
  - ▶ Werden **nicht** von Garbage Collector freigegeben
  - ▶ Expliziter Methodenaufruf **close()**
  - ▶ **Beispiel**: Datei-Handles, Netzwerkverbindungen
- ▶ Wie kann Aufruf von **close()** **garantiert werden**?
  - ▶ Auftreten von **Exceptions**
  - ▶ Vorzeitigem **return**
  - ▶ **Exceptions** bei **close()**
- ▶ „Automatic Resource Management“: **try-with**

```
try (Resource1 r1 = new Resource1();
    Resource2 r2 = new Resource2();
    ...
    ResourceN rN = new ResourceN()){
    /* ... */
}
```

Implizit **immer**: **rN.close(), ..., r2.close(), r1.close()**

## Notizen



## try-with

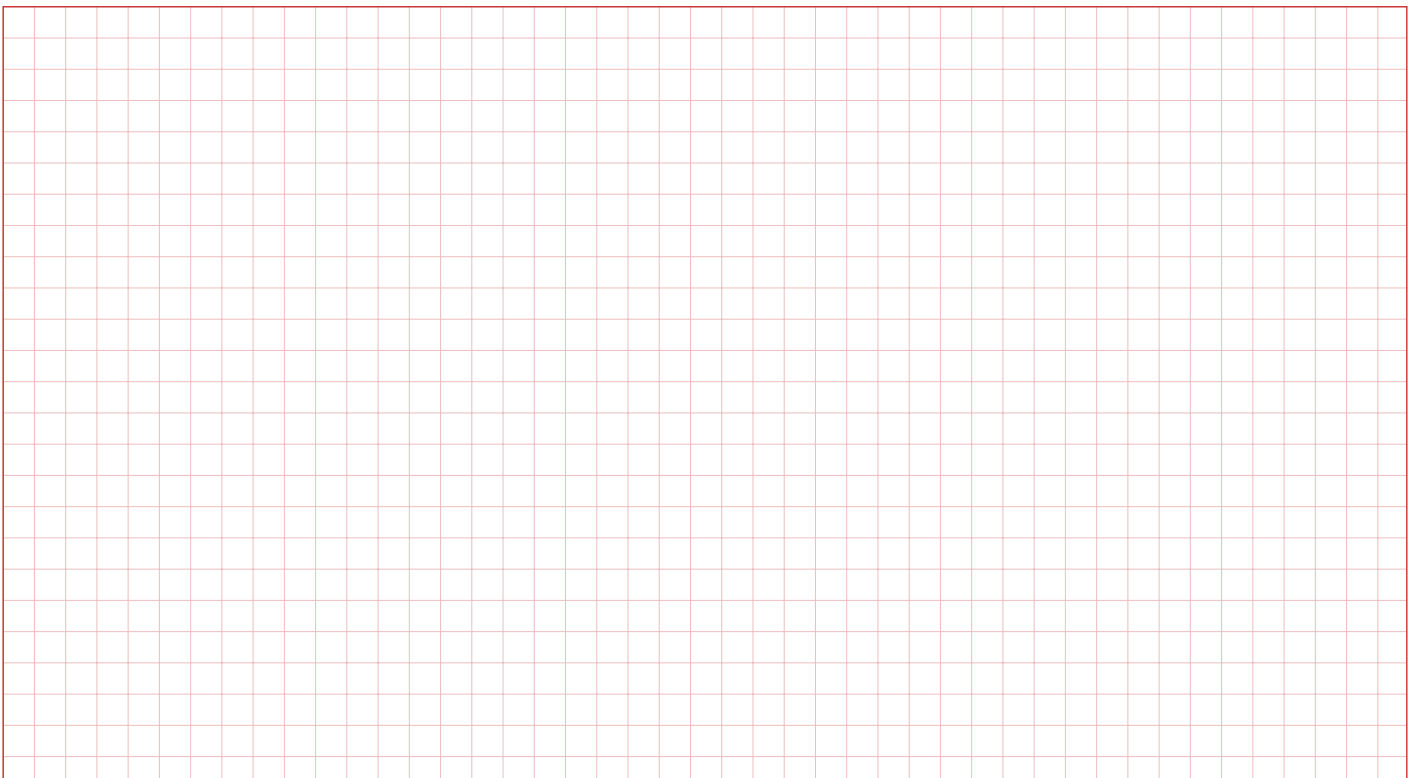
### ► Anwendung auf **Beispiel**

```
64 try (var in = new FileInputStream(from);  
65      var out = new FileOutputStream(to)){  
66     in.transferTo(out);  
67 }
```

ARMExamples.java

- Sehr **kompakt**
- `close()` wird **immer** aufgerufen
- `new FileOutputStream(to)` **scheitert** → nur `in.close()`
- `in/out.close()` **scheitert** → anderes `close()` **wird aufgerufen**
- **Hinweise:** Resourcevariablen sind...
  - **final**
  - nur in **try**-Block sichtbar

## Notizen



# Inhalt

## Automatic Resource Management

- try mit Ressourcen
- Behandlung von Ausnahmen
- AutoCloseable and Closeable

## Notizen

A large rectangular area filled with a fine grid of light gray lines, intended for taking handwritten notes. The grid consists of approximately 30 columns and 40 rows of small squares.

## try mit Ressourcen und Exceptions

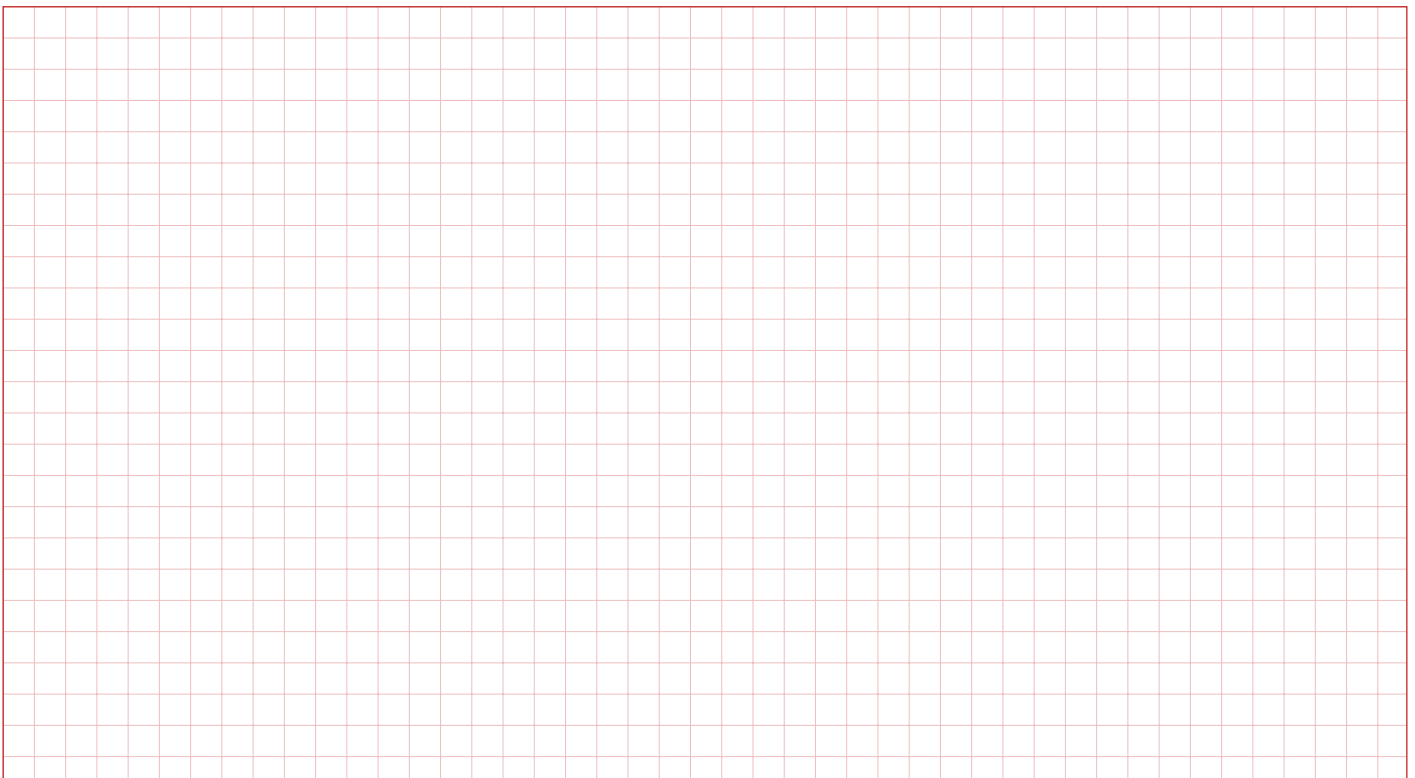
- ▶ Fangen von **Exceptions** wie gewohnt

```
74 try (var in = new FileInputStream(from);  
75       var out = new FileOutputStream(to)){  
76     in.transferTo(out);  
77 } catch (IOException e){  
78     err.println(e.getMessage());  
79 }
```

ARMExamples.java

- ▶ Abfolge
  - ▶ `IOException` tritt auf
  - ▶ `in/out.close()` wird **aufgerufen**
  - ▶ **catch-Block** wird ausgeführt

### Notizen





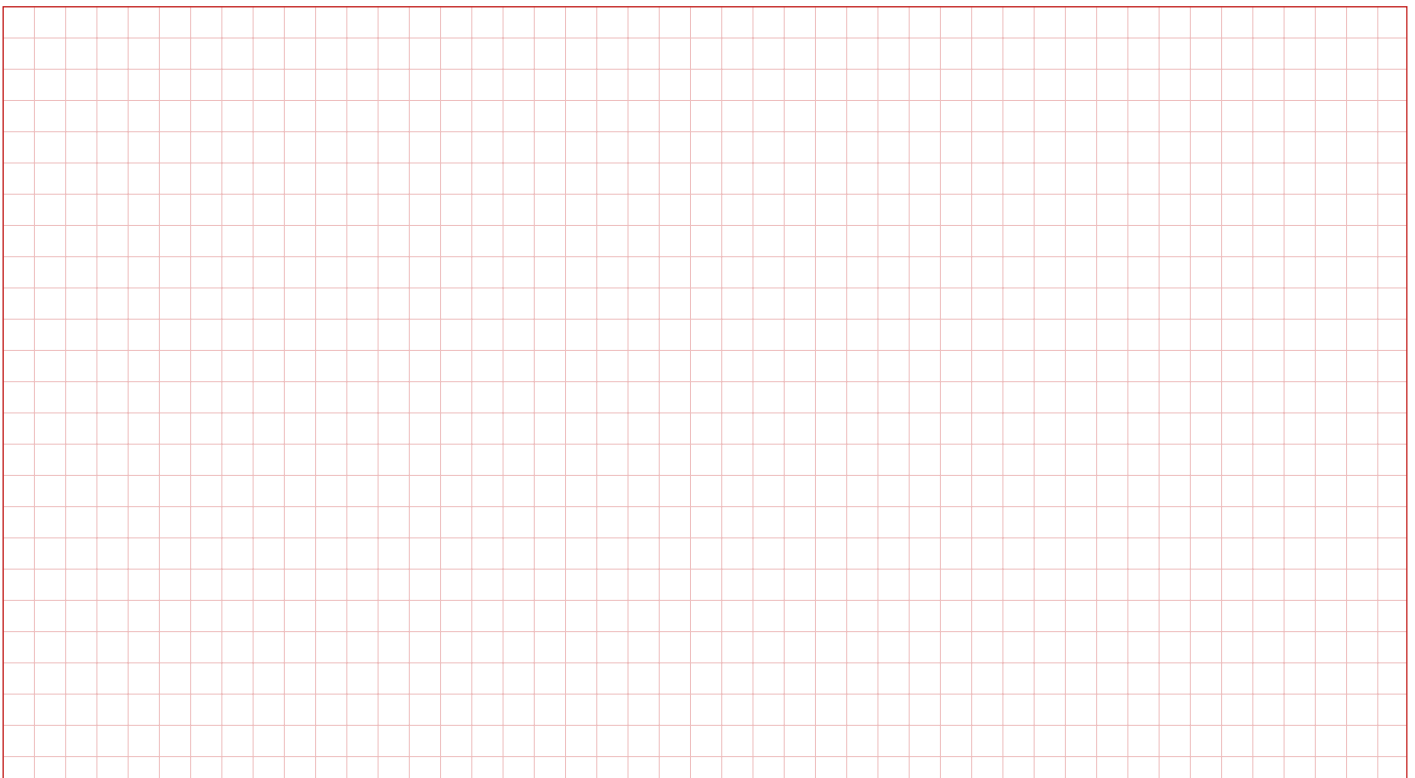
## try mit Ressourcen und unterdrückten Exceptions

- ▶ Abfolge (mit kleiner **Änderung**)
  - ▶ `IOException` tritt auf
  - ▶ `in/out.close()` → `IOException`!
  - ▶ **catch-Block** wird ausgeführt
- ▶ Was passiert wenn `close` wieder `IOException (s)` wirft?
  - ▶ `close`-Exceptions werden **unterdrückt**
  - ▶ `Throwable[] e.getSuppressed()` liefert **unterdrückte Exceptions**

```
90 catch (IOException e){  
92     err.println(e.getMessage());  
94     for (Throwable t : e.getSuppressed())  
95         err.println(t.getMessage());  
97 }
```

ARMExamples.java

## Notizen



# Inhalt

# Automatic Resource Management

## try mit Ressourcen

## AutoCloseable and Closeable

# Notizen

# AutoCloseable

<code>&lt;&lt;interface&gt;&gt;</code>
<b>AutoCloseable</b>
<code>+ close() : void</code>

► Klassen mit freizugebenden Ressourcen

- Implementieren ↗ `AutoCloseable`
- Ressourcenfreigabe in `close()`

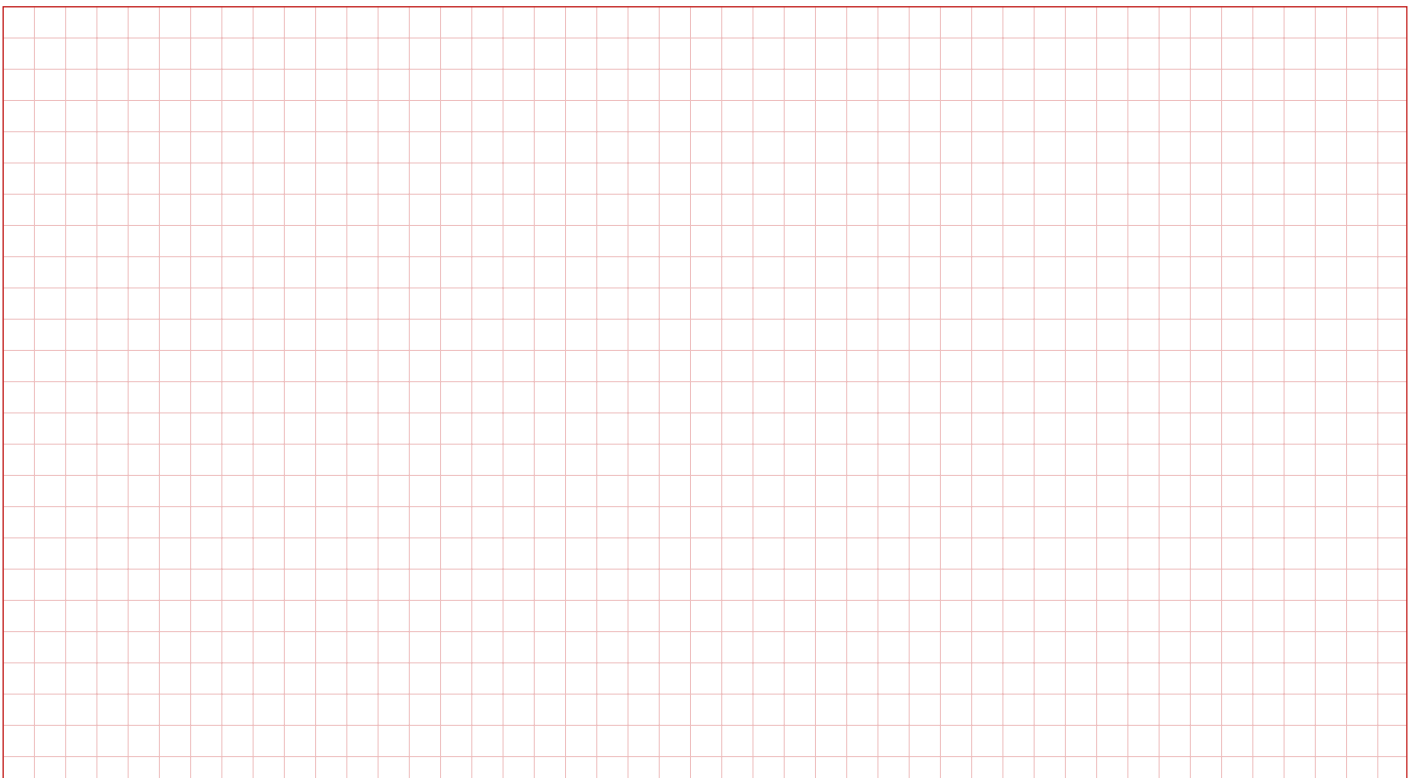
► Erlaubt Verwendung in `try-with`

► Signatur

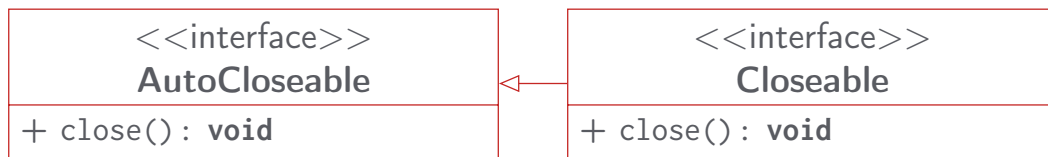
`void close() throws Exception`

- `close()` darf beliebige `Exception` werfen
- Kovarianz: Spezialisierung bei Implementierung erlaubt

## Notizen



## Closeable



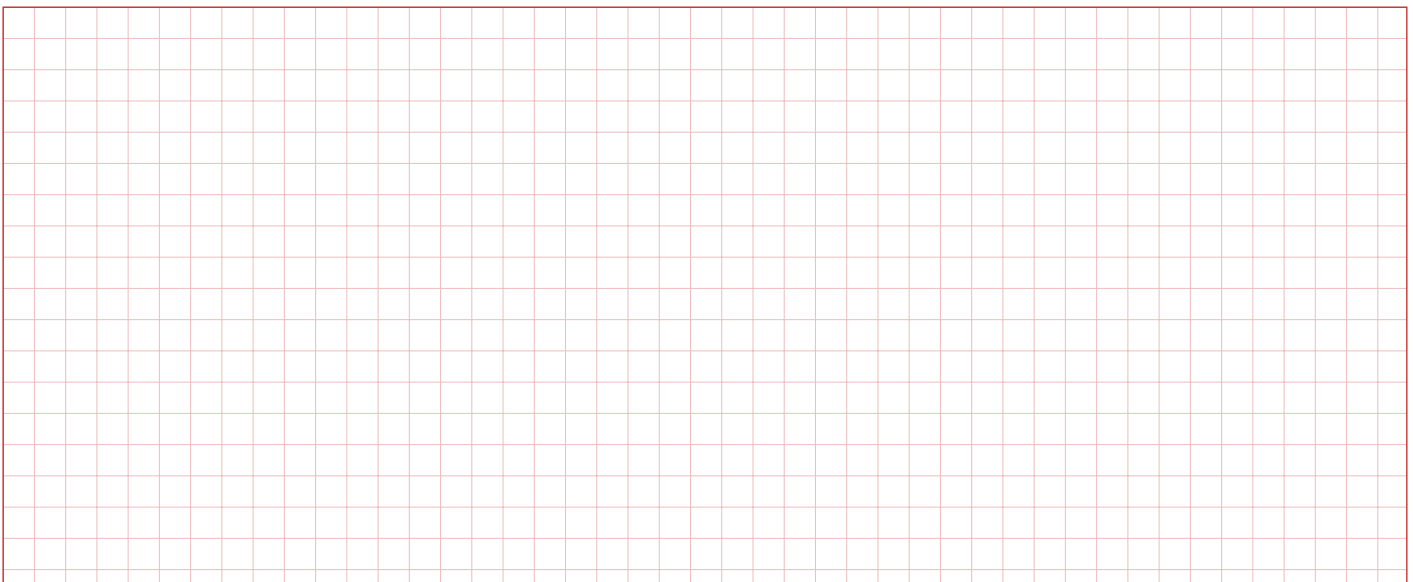
- ▶ Signatur von `Closeable.close()`

```
void close() throws IOException
```

- ▶ `Closeable` existiert länger als `AutoCloseable` !!!
  - ▶ Historisch zuerst `Closeable`
  - ▶ Erkenntnis: zu restriktiv
    - ▶ `IOException` zu speziell
    - ▶ `Closeable.close()` muss idempotent sein
  - ▶ `AutoCloseable.close()` allgemeiner
- ▶ `Closeable` extends `AutoCloseable` erhielt Kompatibilität existierender `Closeable`-Verwendungen

## Notizen

- „Idempotent“ heißt in diesem Kontext, dass `close()` mehr als einmal ohne Nebeneffekte aufgerufen werden kann. Bspw. schließt der erste Aufruf von `close()` einen Stream. Der zweite Aufruf hat dann keine Auswirkungen mehr, der Stream bleibt einfach geschlossen.



# Inhalt

## Dateien und Verzeichnisse

- Arbeiten mit Dateipfaden
- Durchlaufen von Verzeichnissbäumen

## Notizen

A large rectangular area filled with a light gray grid pattern, intended for taking notes. The grid consists of small squares, with a slightly larger margin at the top for a header.

# Inhalt

## Dateien und Verzeichnisse

- Arbeiten mit Dateipfaden
  - Eigenschaften von Dateien und Verzeichnissen
  - Datei- und Verzeichnis-Operationen
  - Dateien lesen und schreiben

## Notizen

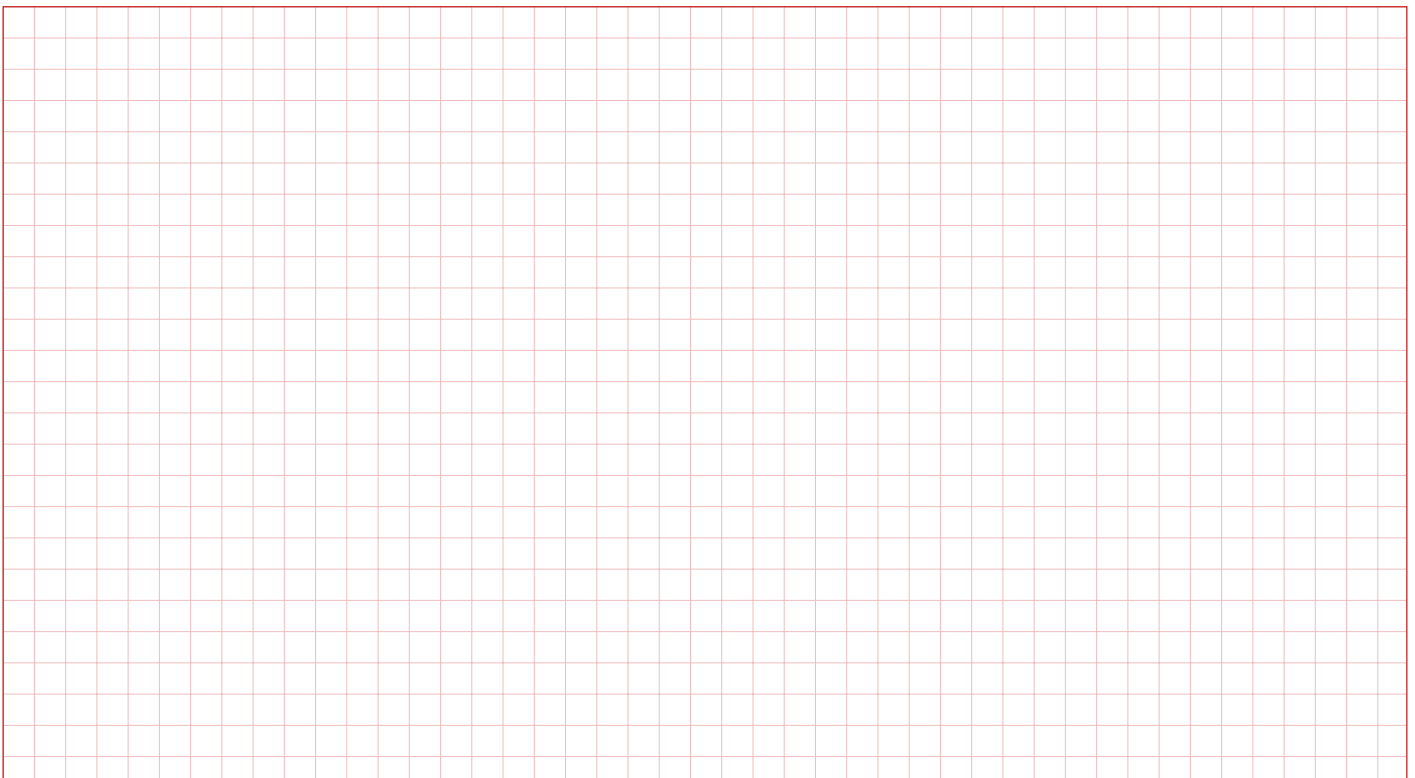
A large rectangular area filled with a light gray grid of squares, intended for taking notes. The grid consists of approximately 30 columns and 30 rows of squares.

## Path

- ▶ Bisher: Inhalte von Dateien **lesen/schreiben**
- ▶ Jetzt
  - ▶ Dateipfade
  - ▶ **Eigenschaften** von Verzeichnissen/Dateien
  - ▶ **Umbenennen, Kopieren, Erstellen** von Verzeichnissen/Dateien
  - ▶ **Durchsuchen** von Verzeichnissen
- ▶ `java.nio.file.Path` für **Pfade**
- ▶ **Allgemeiner Aufbau:**
  - ▶ Sequenz von **Verzeichnis-** und **Dateinamenelemente**
  - ▶ Getrennt durch **Trennzeichen** (/ oder \)
  - ▶ Eventuell **Wurzelement** am Anfang
- ▶ **Beispiele** mit / als Trennzeichen

```
<e1>/<e2>/.../<eN>  
<root>/<e1>/<e2>/.../<eN>
```

## Notizen



## Erstellen von Path-Objekten

- ▶ Documents/Thesis.doc (**relativer** Pfad)

```
Path p = Path.resolve("Documents/Thesis.doc");
```

- ▶ Dokumente\Rezepte\Geheimes Waffelrezept.txt (**relativer** Pfad)

```
Path p = Path.of("Dokumente", "Rezepte", "Geheimes Waffelrezept.txt");
```

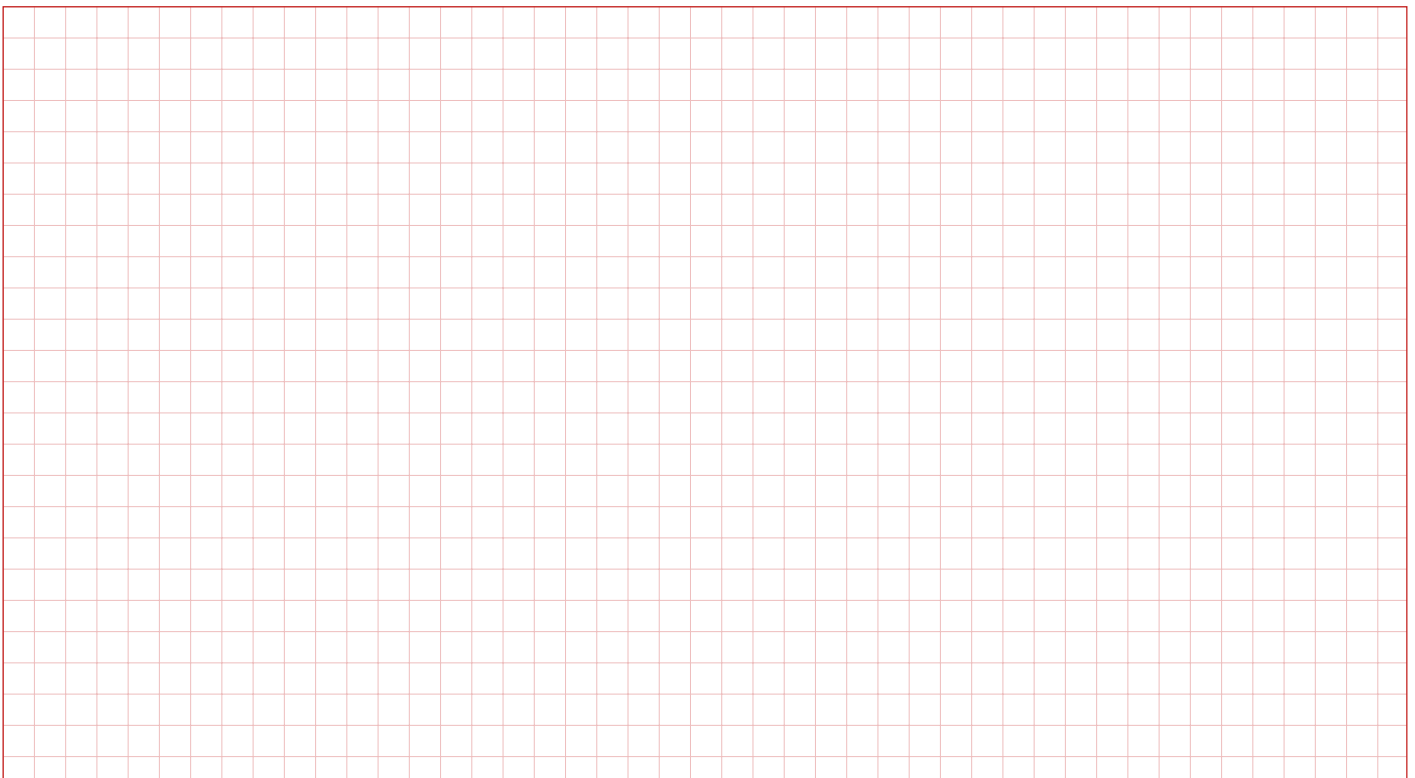
- ▶ /home/auer/workspace/java1/solutions (**absoluter** Pfad)

```
Path p = Paths.get("/", "home", "auer", "workspace", "java1", "solutions");
```

- ▶ C:\Windows\System32\cmd.exe (**absoluter** Pfad)

```
Path p = FileSystems.getDefault().getPath("C:\\Windows\\System32\\cmd.exe");
```

## Notizen





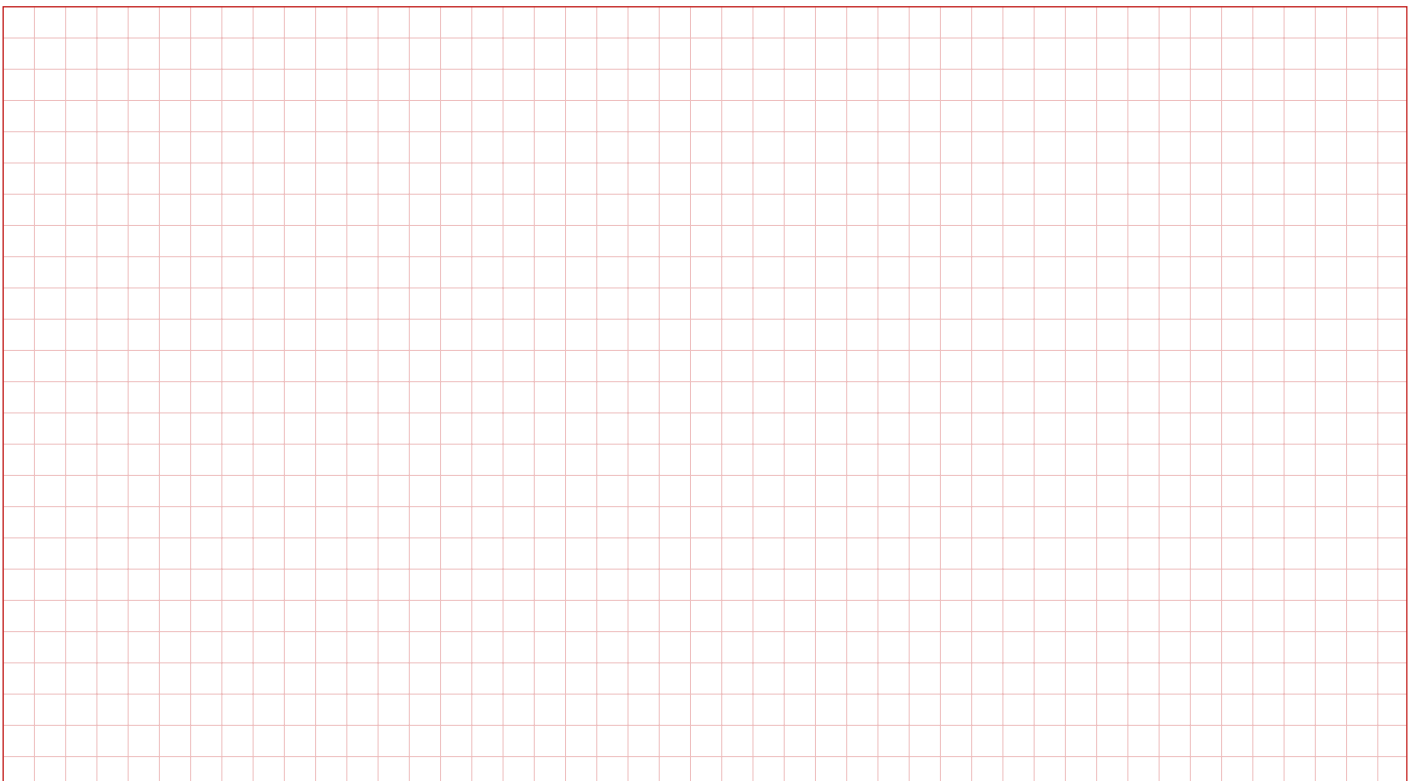
## Path: Nützliche Methoden

- ▶ [Path](#) `getFileName()` — liefert **Datei-/Verzeichnisname**
- ▶ [Path](#) `getParent()` — liefert **Elternelement**
- ▶ **boolean** `isAbsolute/Relative()` — **true** wenn **absoluter/relativer** Pfad
- ▶ [Path](#) `toAbsolutePath()` — umwandeln in **absoluten** Pfad
- ▶ [Path](#) `relativize(Path p)` — umwandeln in Pfad **relativ zu p**
- ▶ [Path](#) `normalize()` — entfernt **redundante** Bestandteile

```
/home/../../etc/./passwd -> /etc/passwd
```


- ▶ [Path](#) `getName(int i)` — gibt **i-ten Bestandteil** des Pfads zurück
- ▶ **boolean** `starts/endsWith(String/Path p)` — **true** wenn Pfad mit p **beginnt/endet**
- ▶ [Iterator](#)<Path> `iterator()` — liefert [Iterator](#) über **Pfad-Bestandteile**

## Notizen



## Path: Beispiel

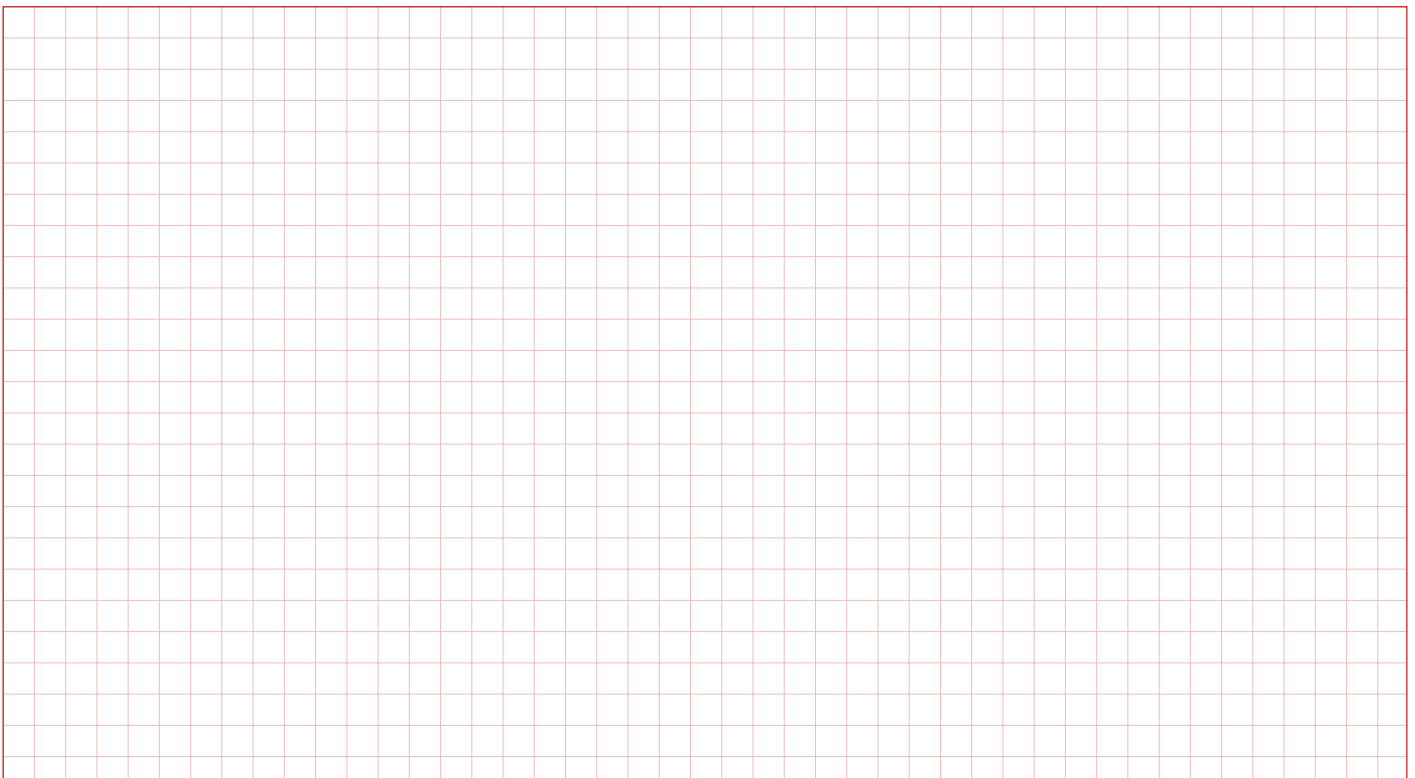
### ► Beispiel

```
44  runPathExampleUnix/Windows
45 Path path = Paths.get(pathString);
46 out.printf("toString(): %s\n", path.toString());
47 out.printf("getFileName(): %s\n", path.getFileName());
48 out.printf("getParent(): %s\n", path.getParent());
50 out.print("iterator: ");
51 for (Path p : path)
52     out.print(p + " ");
53 out.println();
55 out.printf("isAbsolute: %b\n", path.isAbsolute());
56 out.printf("toAbsolutePath(): %s\n", path.toAbsolutePath());
57 /* ... */
```

 PathExamples.java

- runPathExampleUnix für **Linux und Co.**
- runPathExampleWindows für **Windows**

## Notizen



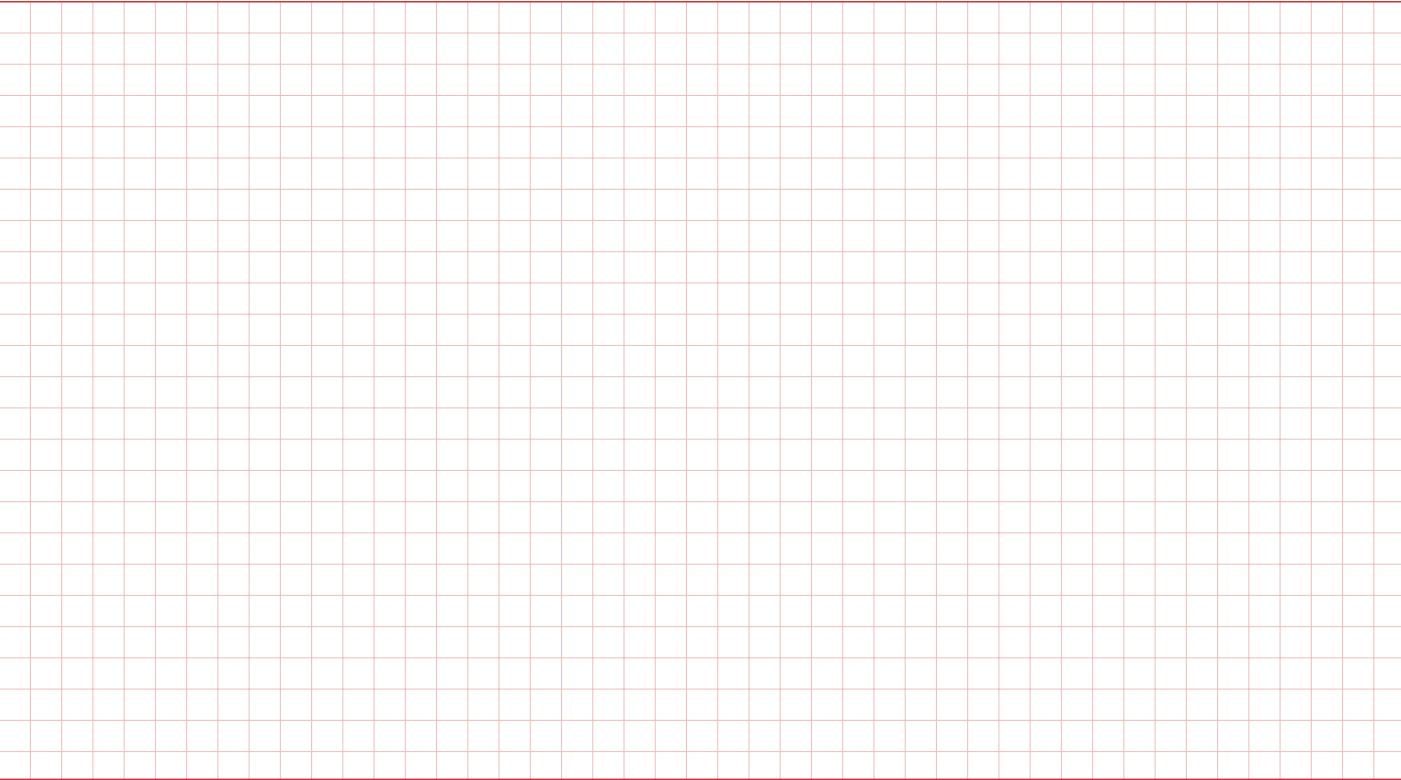
# Inhalt

## Dateien und Verzeichnisse

### Arbeiten mit Dateipfaden

- Eigenschaften von Dateien und Verzeichnissen
- Datei- und Verzeichnis-Operationen
- Dateien lesen und schreiben

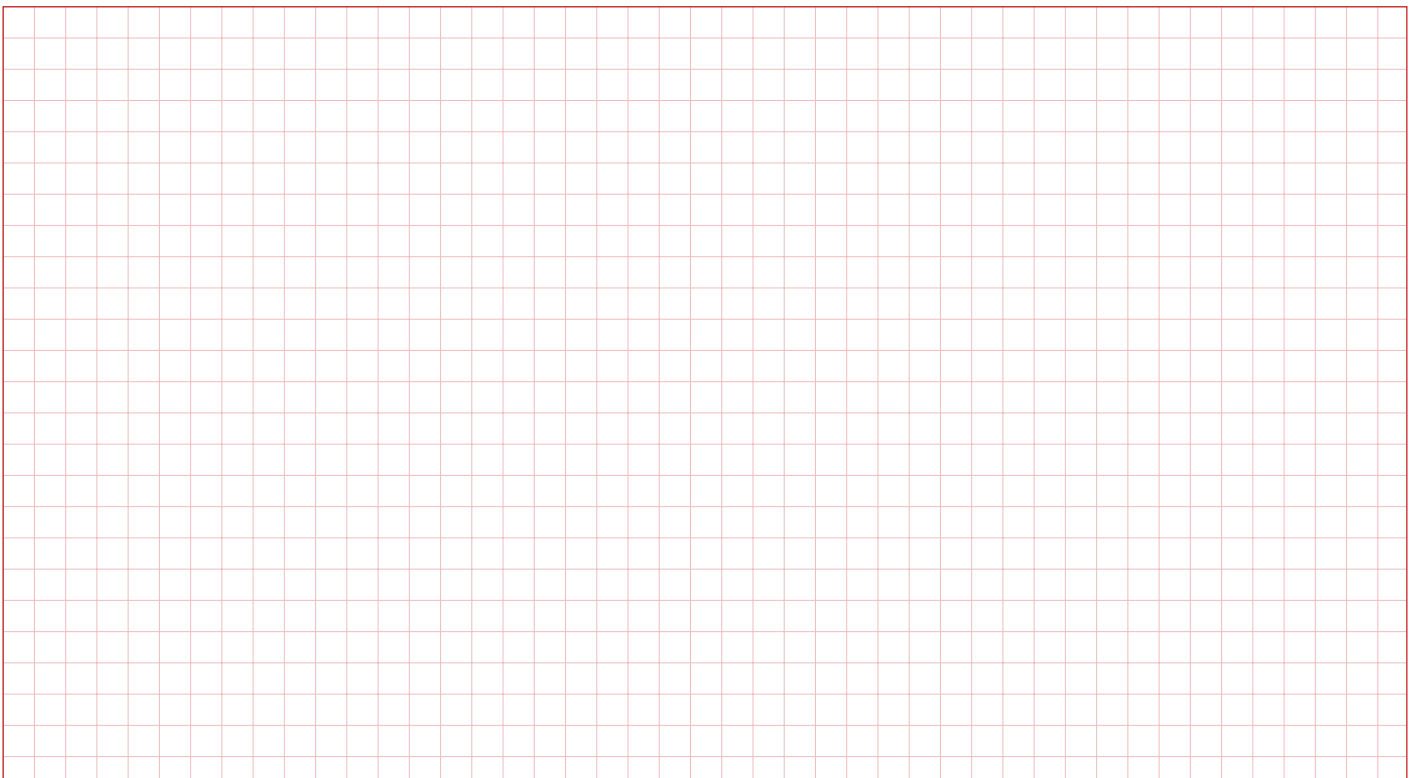
## Notizen



# Path

- ▶ Was macht man mit einem `Path`-Objekt?
- ▶ Hinweis: `Path` ist nur ein Pfad
  - ▶ Kann auf Datei, Verzeichnis oder was anderes verweisen
  - ▶ Datei/Verzeichnis muss nicht existieren
- ▶ Eigenschaften mit der Klasse `Files` abfragen
  - ▶ `static boolean exists(Path p)` — `true` wenn existent, sonst `false`
  - ▶ `static boolean isDirectory/RegularFile(Path p)` — `true` wenn Verzeichnis/reguläre Datei, sonst `false`
  - ▶ `static long size(Path p)` — Länge
  - ▶ `static boolean getOwner(Path p)` — gibt Eigentümer zurück
  - ▶ `static boolean isHidden/isExecutable/isReadable(Path p)` — `true` wenn versteckt/ausführbar/lesbar, sonst `false`
  - ▶ ...

## Notizen

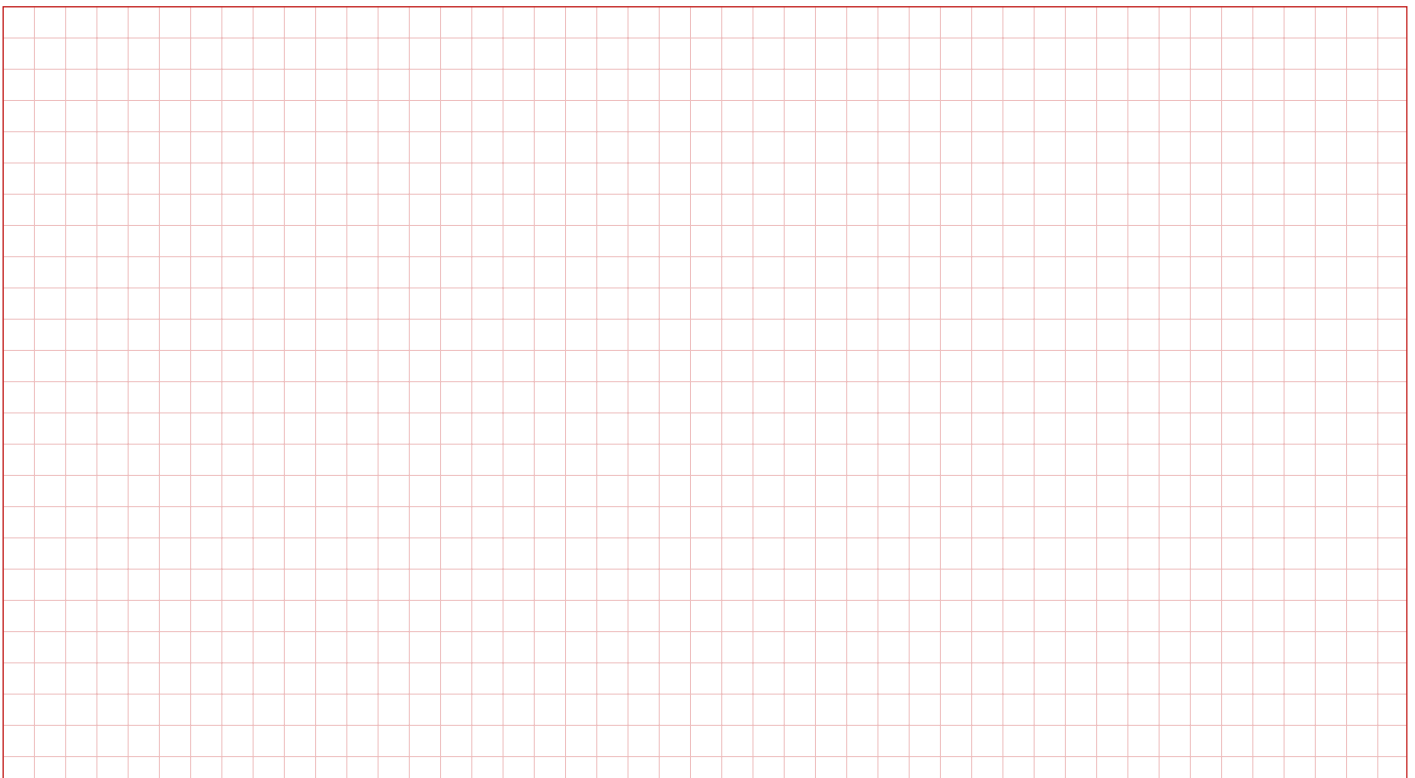


## Path

- ▶ `printPathProperties` in `PathExamples.java` gibt **Eigenschaften** eines `Path`-Objekts aus
- ▶ Beispiel

```
gradle runPrintPathProperties --args="build.gradle"
path: build.gradle
exists: true
isDirectory: false
isExecutable: false
isHidden: false
isReadable: true
isRegularFile: true
isSymbolicLink: false
isWritable: true
size: 367 bytes
getOwner: chris
getLastModifiedTime: 2020-07-23T11:37:13.340527613Z
```

## Notizen



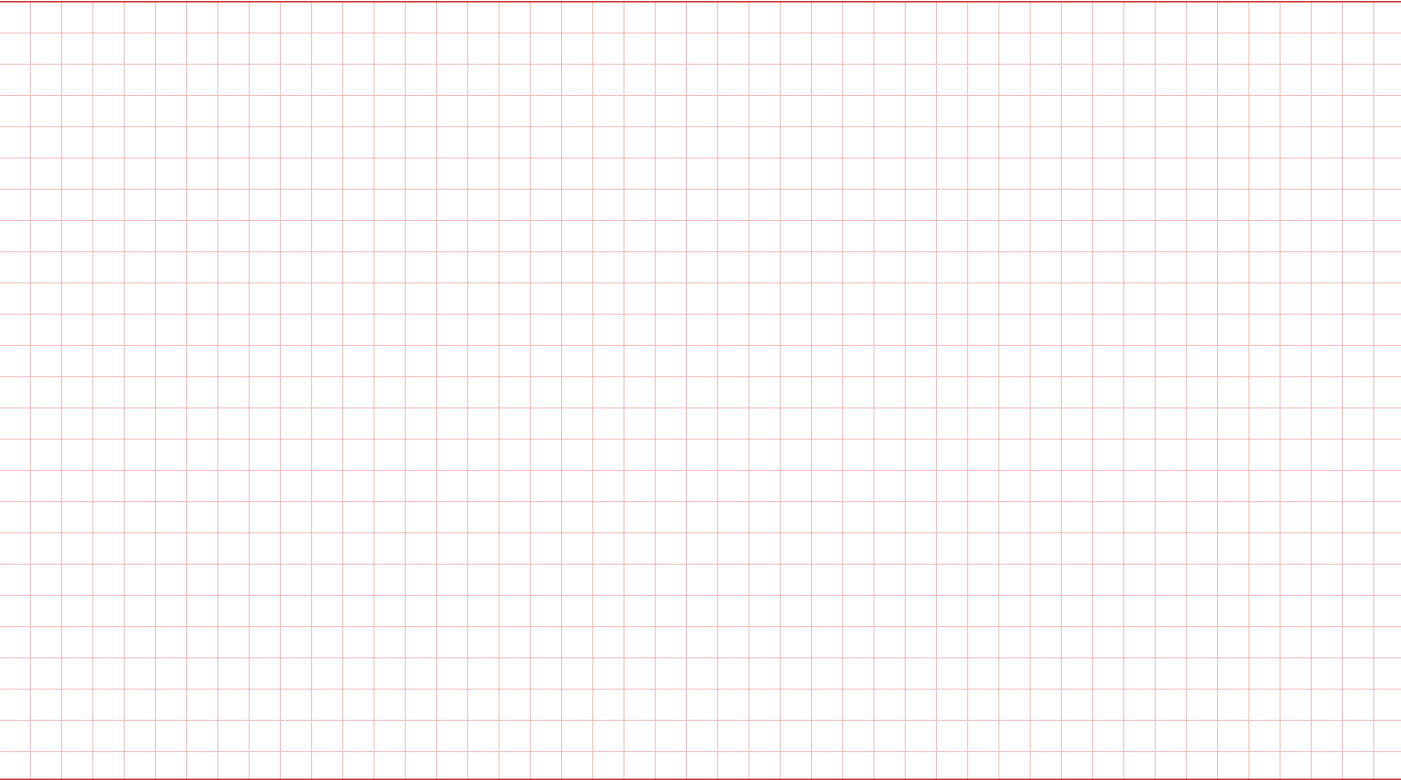
# Inhalt

## Dateien und Verzeichnisse

### Arbeiten mit Dateipfaden

- Eigenschaften von Dateien und Verzeichnissen
- Datei- und Verzeichnis-Operationen
- Dateien lesen und schreiben

## Notizen



## Datei- und Verzeichnis-Operationen

Operation	Beschreibung
<code>createDirectory</code>	Verzeichnis erstellen (ohne „Zwischenschritte“)
<code>createDirectories</code>	Verzeichnis erstellen (mit „Zwischenschritten“)
<code>createFile</code>	Datei erstellen
<code>createLink</code>	symbolischen Link erstellen (Unix)
<code>createTempDirectory</code>	temporäres Verzeichnis erzeugen
<code>createTempFile</code>	temporäre Datei erzeugen
<code>delete</code>	<b>existente</b> Datei entfernen
<code>deleteIfExists</code>	Datei entfernen (ohne Exception)
<code>move</code>	Datei/Verzeichnis verschieben

► **Beispiel:** Methode `pathOperations` in `PathOperations.java`

### Notizen



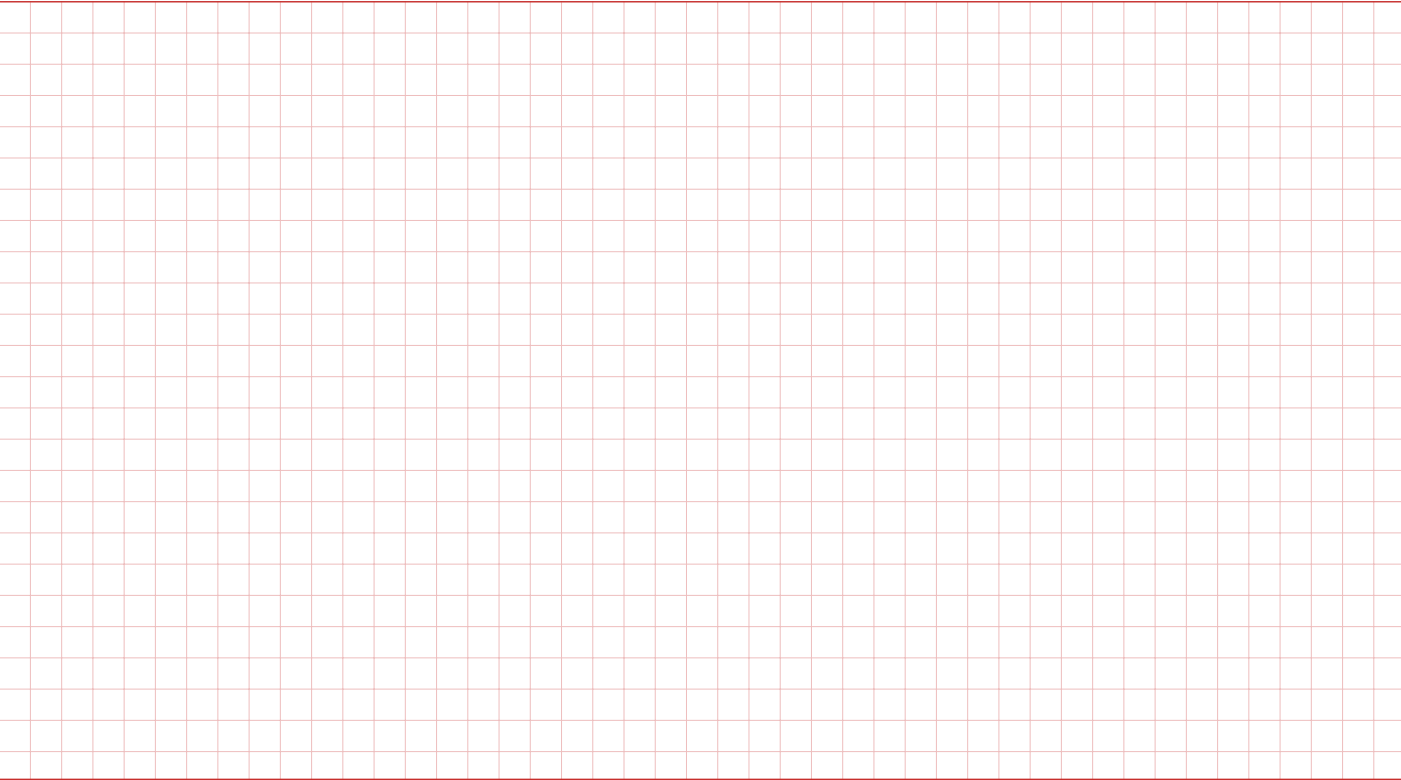
# Inhalt

## Dateien und Verzeichnisse

### Arbeiten mit Dateipfaden

- Eigenschaften von Dateien und Verzeichnissen
- Datei- und Verzeichnis-Operationen
- Dateien lesen und schreiben

## Notizen





# Dateien lesen und schreiben

► [Files](#) bietet bequeme Methoden zum Lesen/Schreiben von kleinen Dateien

Methode	Beschreibung
<code>readAllBytes</code>	liest Inhalt in <code>byte[]</code>
<code>readString</code>	liest Inhalt in <a href="#">String</a>
<code>readAllLines</code>	liest Zeilen in <a href="#">List&lt;String&gt;</a>
<code>write</code>	schreibt <code>byte[]</code>
<code>writeString</code>	schreibt <a href="#">String</a> / <a href="#">CharSequence</a>


► Optionen zum Schreiben der Datei (siehe [Dokumentation](#))

Option	Bedeutung
<a href="#">StandardOpenOption.CREATE</a>	Datei neu erstellen
<a href="#">StandardOpenOption.APPEND</a>	Inhalt anhängen
<a href="#">StandardOpenOption.WRITE</a>	zum Schreiben öffnen

## Notizen

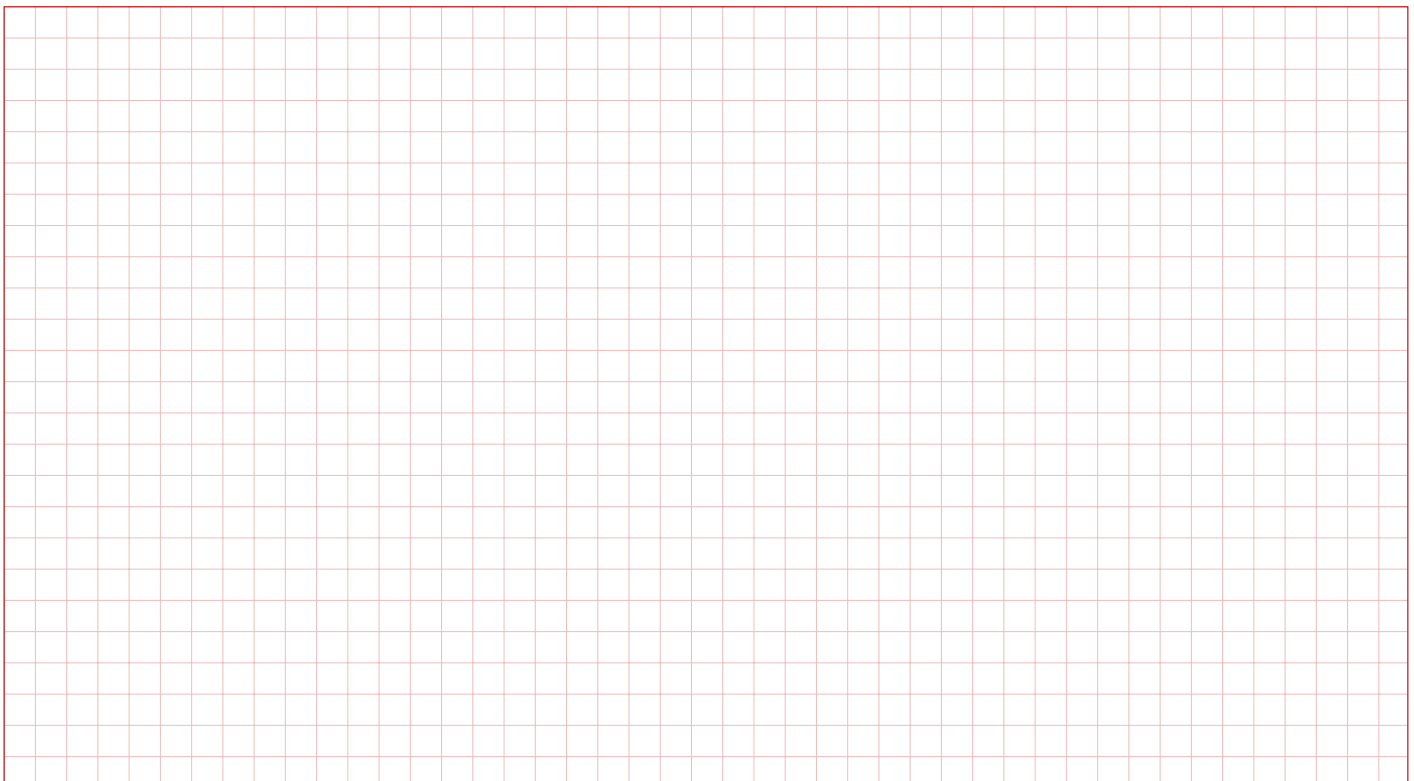
# Dateien lesen und schreiben

## ► Beispiel

```
104  runReadWriteFile  
105 Path p = Paths.get("output.txt");  
107 for (int i = 0; i < 100; i++)  
108     Files.writeString(p,  
109         "A work and no play makes Jack a dull boy.\n",  
110         StandardOpenOption.APPEND,  
111         StandardOpenOption.CREATE);  
113 var lines = Files.readAllLines(p);  
115 for (String line : lines)  
116     out.println(line);
```

 PathExamples.java

## Notizen



## Zusammenarbeit mit IO-Klassen

- ▶ [Files](#) bietet Methoden für Zugriff über **Streams/Readers/Writers**

Methode	Bedeutung
<code>newInputStream(Path p, ...)</code>	erzeugt <a href="#">InputStream</a>
<code>newOutputStream(Path p, ...)</code>	erzeugt <a href="#">OutputStream</a>
<code>newBufferedReader(Path p, ...)</code>	erzeugt <a href="#">Reader</a>
<code>newBufferedWriter(Path p, ...)</code>	erzeugt <a href="#">Writer</a>

- ▶ Bei [Reader](#)/[Writer](#) optional mit [Charset](#)
- ▶ [Files](#) besitzt noch **viel mehr Methoden** (siehe [Dokumentation](#))

### Notizen



# Inhalt

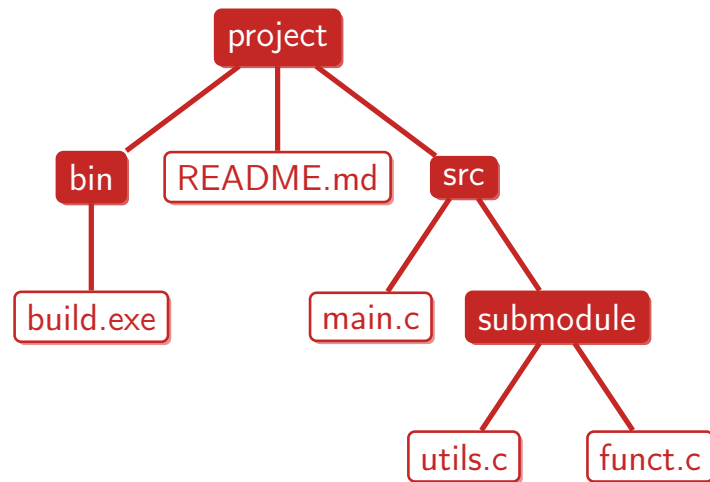
Dateien und Verzeichnisse

Durchlaufen von Verzeichnissbäumen

## Notizen

## Durchlaufen von Verzeichnissbäumen

- Verzeichnisse, Unterverzeichnisse und Dateien bilden einen Baum



- Baumdurchlauf für Suche, rekursive Kopien, Verzeichnisbaum komprimieren, ...

## Notizen

# Durchlaufen von Verzeichnissbäumen

## ► Methode `Files.walkFileTree`

```
Path walkFileTree(Path start, FileVisitor<Path> visitor)
```

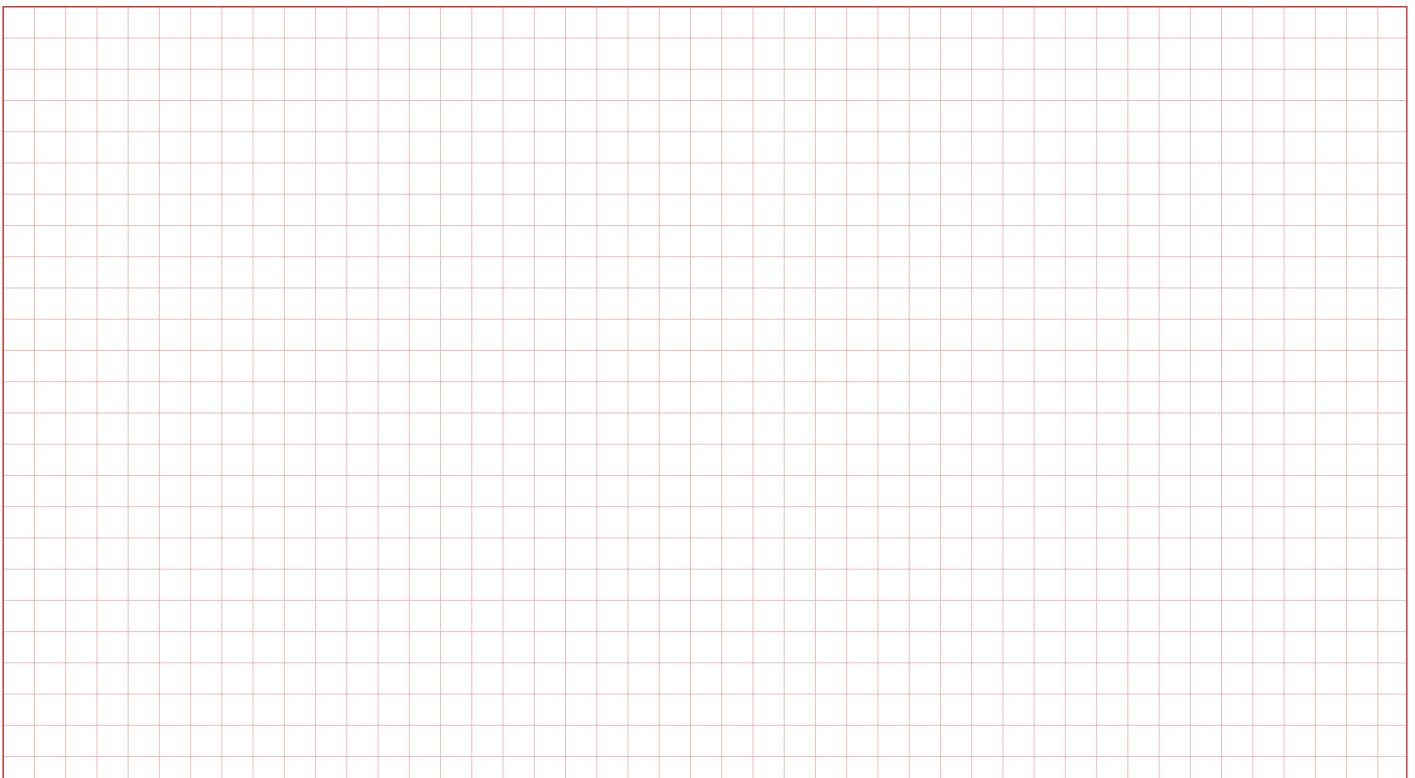
- Beginnt **rekursiven Durchlauf** ab `start`
- Ruft **Methoden von `visitor`** für gefundene Verzeichnisse/Dateien auf

## ► **interface** `FileVisitor`

- Müssen **wir implementieren**
- Vier Methoden der Form `FileVisitResult visit(Path p, ...)`
  - `Path p` — **gefundenes Objekt**
  - **enum** `FileVisitResult` — **Rückmeldung** an `walkFileTree`

Wert	Bedeutung
CONTINUE	Weitermachen
TERMINATE	Abbrechen
SKIP_SIBLINGS	„Geschwister“ von <code>p</code> überspringen
SKIP_SUBTREE	Verzeichnis <code>p</code> überspringen

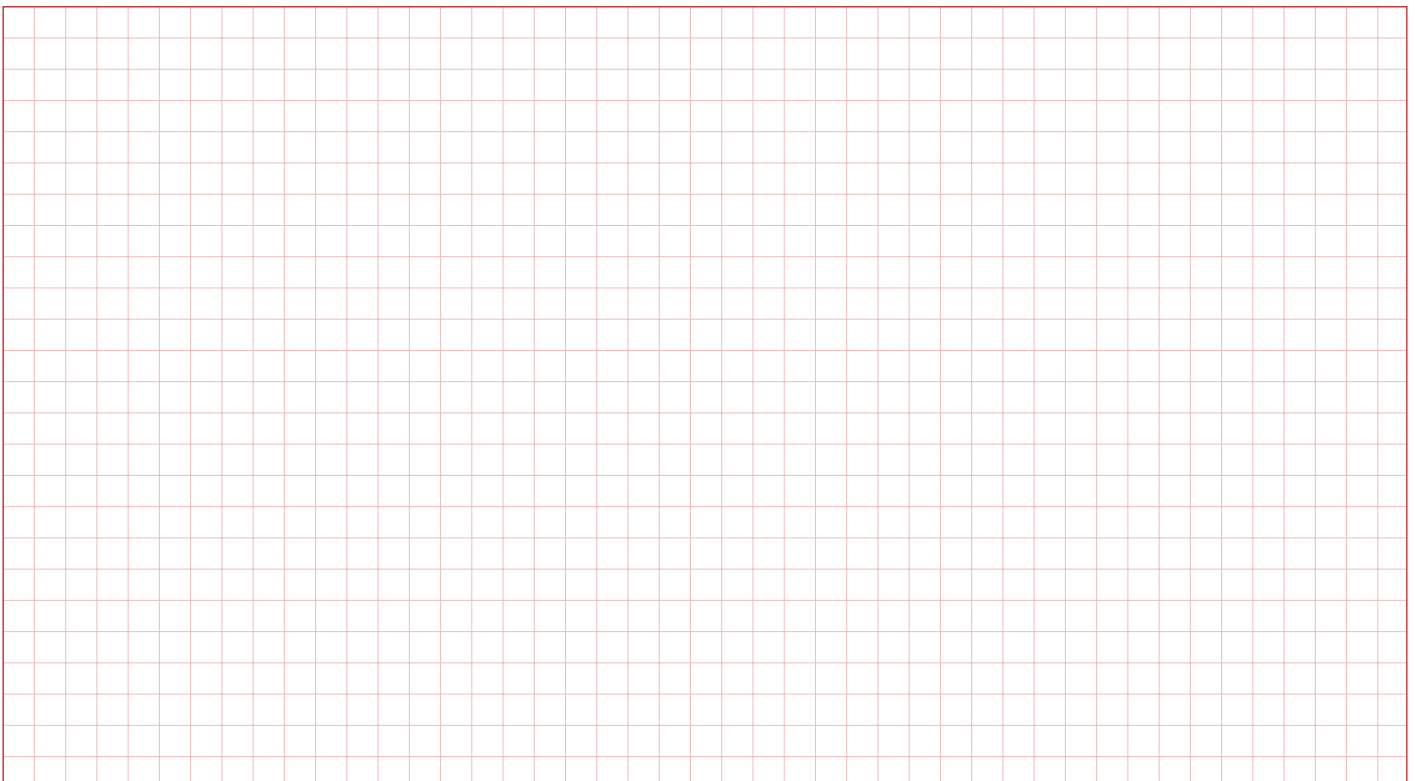
## Notizen



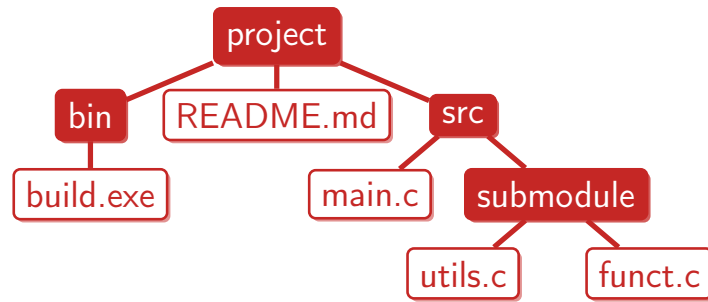
## FileVisitor

- ▶ `preVisitDirectory(Path dir, BasicFileAttributes attr)`  
Aufruf **bevor Verzeichnis path** durchlaufen wird
- ▶ `postVisitDirectory(Path dir, IOException e)`  
Aufruf **nachdem Verzeichnis path** durchlaufen wurde
- ▶ `visitFile(Path dir, BasicFileAttributes attr)`  
Aufruf **wenn Datei path** gefunden wurde
- ▶ `visitFileFailed(Path dir, IOException e)`  
Aufruf **wenn Datei path nicht bearbeitet** werden konnte

## Notizen

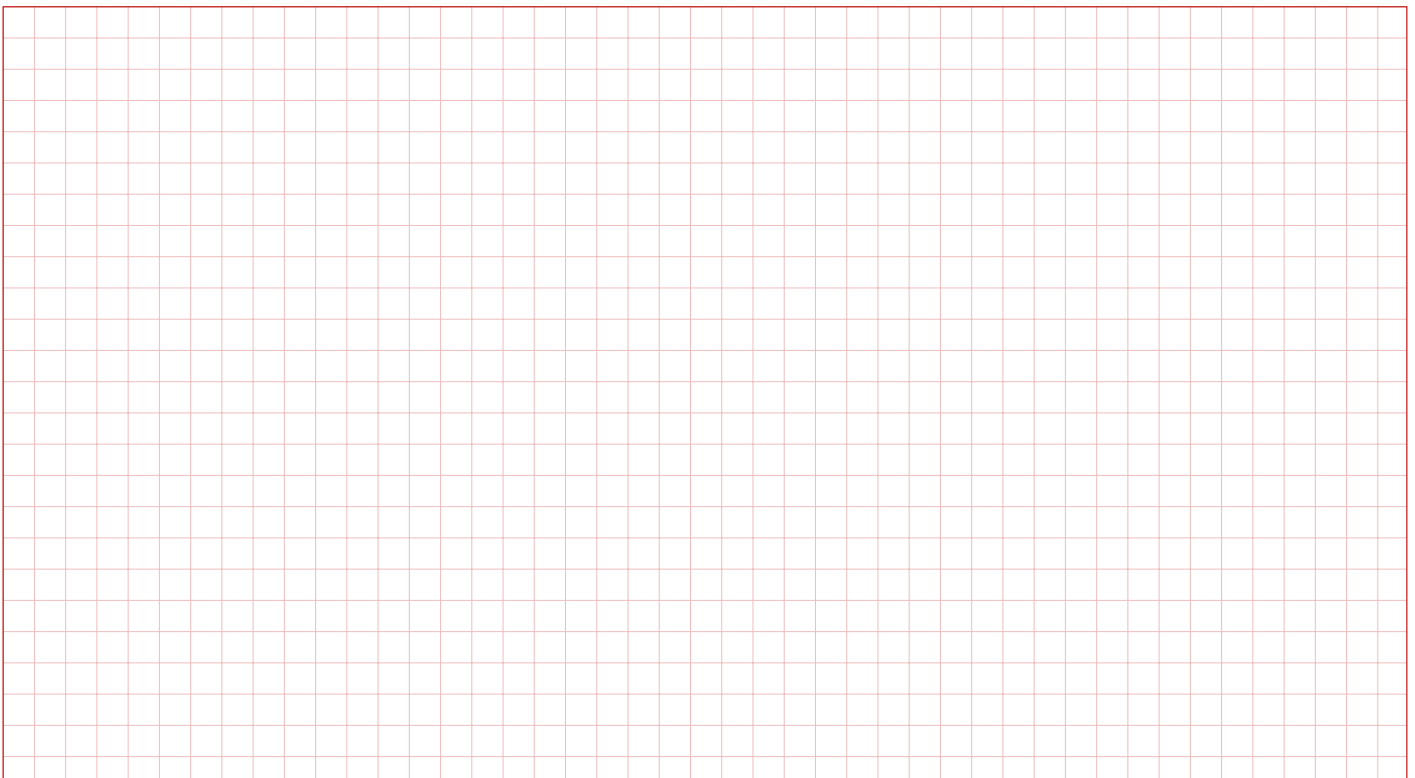


## FileVisitor



1. `preVisitDir("project")`
2. `preVisitDir("bin")`
3. `visitFile("build.exe")`
4. `postVisitDir("bin")`
5. `visitFile("README.md")`
6. `preVisitDir("src")`
7. `visitFile("main.c")`
8. `preVisitDir("submodule")`
9. `visitFile("utils.c")`
10. `visitFile("funct.c")`
11. `postVisitDir("submodule")`
12. `postVisitDir("src")`
13. `postVisitDir("project")`

## Notizen





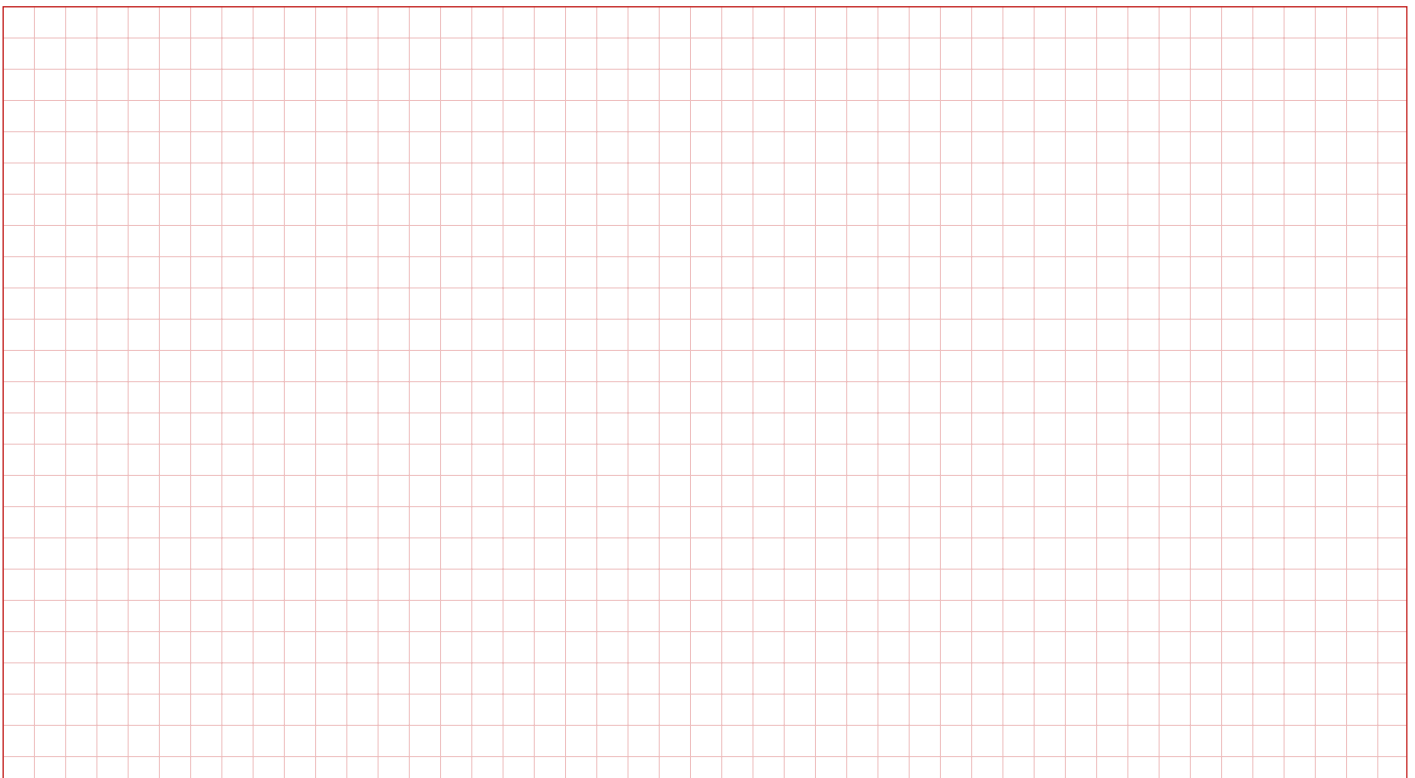
## Beispiel: Dateien suchen

- ▶ Programm das **Dateien** anhand **Suchschlüssel** findet
- ▶ **Beispiel:**

```
% java FileSearchExample de Week
Found files:
de/hawlandshut/java1/oopbasics/WeekdayExamples.java
de/hawlandshut/java1/oopbasics/WeekdayBetaUtils.java
de/hawlandshut/java1/oopbasics/WeekdayBeta.java
de/hawlandshut/java1/oopbasics/Weekday.java
de/hawlandshut/java1/oopbasics/WeekdayAlpha.java
de/hawlandshut/java1/oopbasics/WeekdayGamma.java
```

- ▶ Suche von Dateien mit **Week** im Namen
- ▶ In **de** und **Unterverzeichnissen**

## Notizen



## Beispiel: Dateien suchen

- ▶ Main-Klasse FileSearchExample
- ▶ Geschachtelte statische Klasse FileSearchVisitor

```
34 private static class FileSearchVisitor
35     implements FileVisitor<Path>{
```

FileSearchExample.java

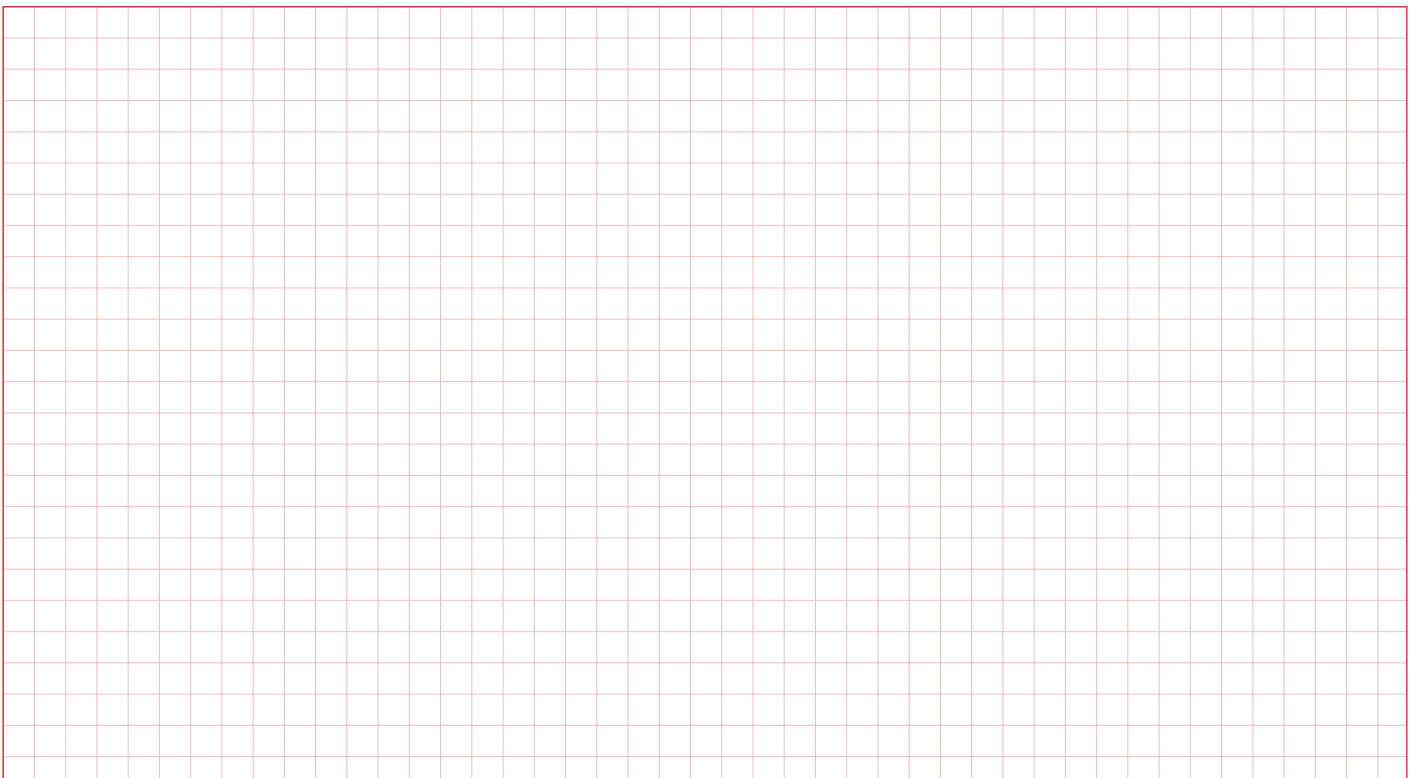
- ▶ Attribute

```
39 private String searchKey;
40 private List<Path> foundFiles;
42 private FileSearchVisitor(String searchKey){
43     this.searchKey = searchKey;
44     foundFiles = new LinkedList<Path>();
45 }
```

FileSearchExample.java

- ▶ searchKey — Suchschlüssel
- ▶ foundFiles — Liste mit gefundenen Dateien

## Notizen



## Beispiel: Dateien suchen

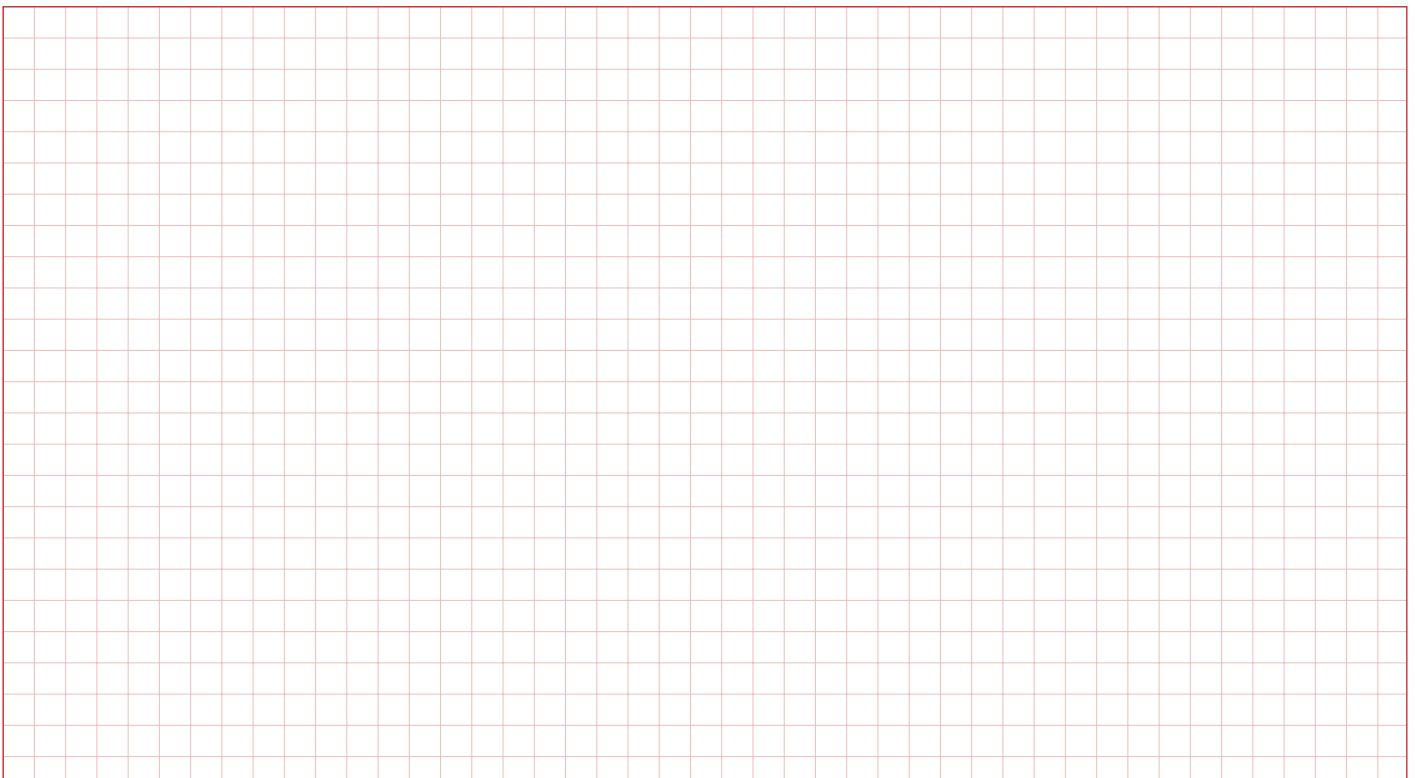
### ► preVisitDirectory

```
53 @Override public FileVisitResult preVisitDirectory(  
54     Path dir, BasicFileAttributes attrs) {  
56     out.println("Entering " + dir);  
58     return FileVisitResult.CONTINUE;  
59 }
```

FileSearchExample.java

- Macht **Ausgabe** (optional)
- Teilt Aufrufer mit **fortzufahren** (↗ `FileVisitResult.CONTINUE`)

## Notizen



## Beispiel: Dateien suchen

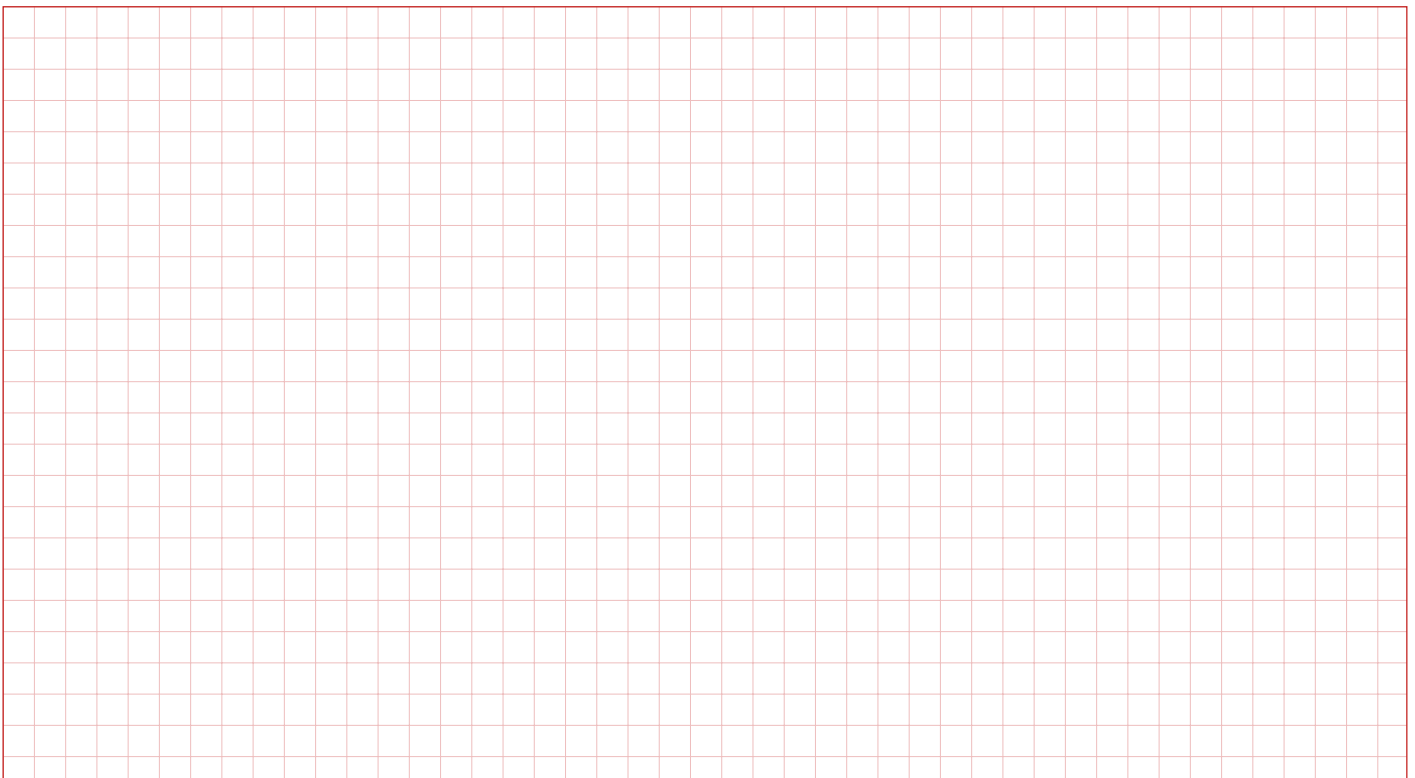
### ► postVisitDirectory

```
82 @Override public FileVisitResult postVisitDirectory(  
83     Path dir, IOException exc) {  
84     out.println("Leaving " + dir);  
86     if (exc != null){  
87         err.println(exc.getMessage());  
88         return FileVisitResult.TERMINATE;  
89     }else  
90         return FileVisitResult.CONTINUE;  
91 }
```

FileSearchExample.java

- Macht **Ausgabe** (optional)
- [IOException](#) → **Fehler** beim Durchlaufen
  - Fehler **ausgeben**
  - Durchlauf mit FileSearchVisitor.TERMINATE **abbrechen**
  - **Hinweis:** FileSearchVisitor.CONTINUE **auch möglich!**
- Ansonsten **weitermachen**

## Notizen



## Beispiel: Dateien suchen

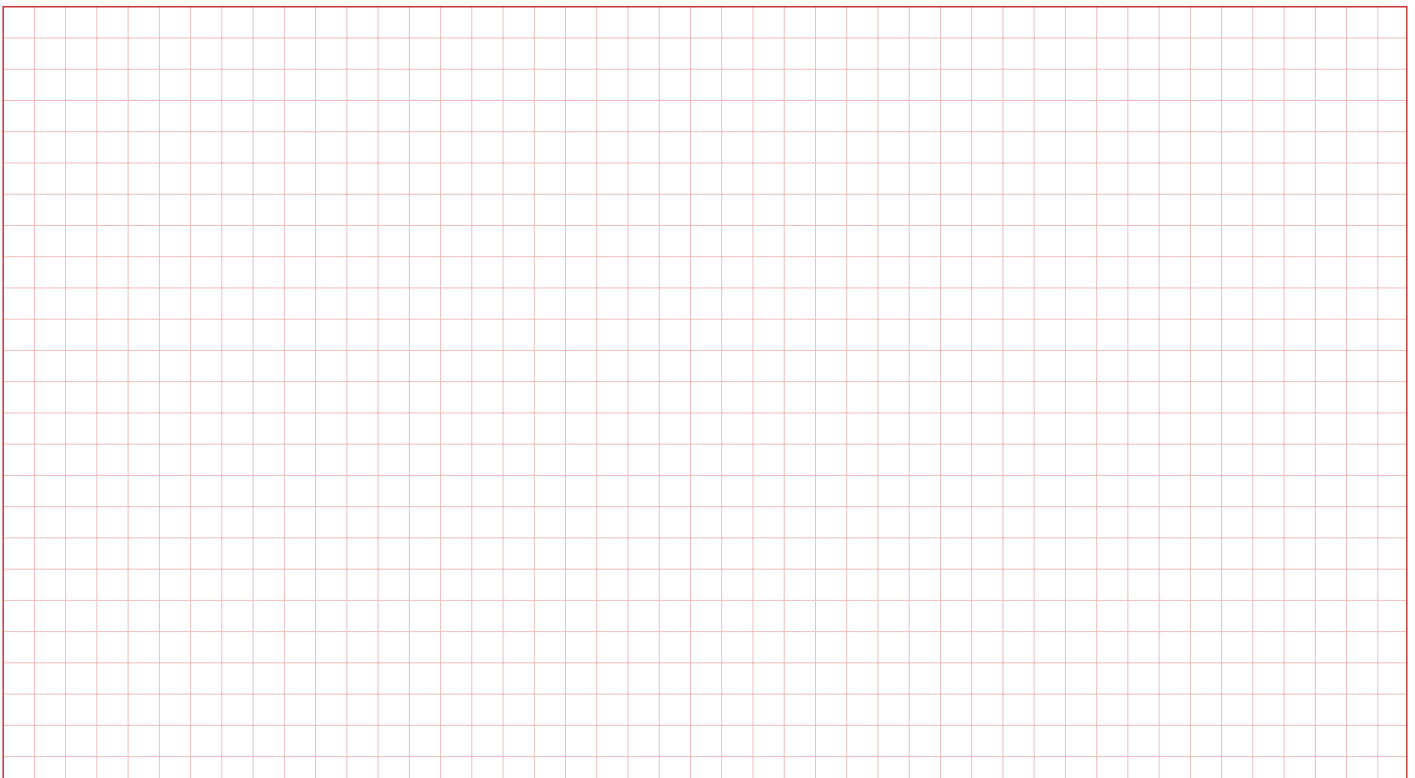
### ► visitFile

```
63 @Override public FileVisitResult visitFile(  
64     Path file, BasicFileAttributes attrs) {  
66     if (file.toString().contains(searchKey))  
67         foundFiles.add(file);  
69     return FileVisitResult.CONTINUE;  
70 }
```

FileSearchExample.java

- Prüft ob **Suchschlüssel** in Dateinamen ist
- Ja → zu **results** hinzufügen
- Teilt Aufrufer mit **fortzufahren** (↗ `FileVisitResult.CONTINUE`)

## Notizen



## Beispiel: Dateien suchen

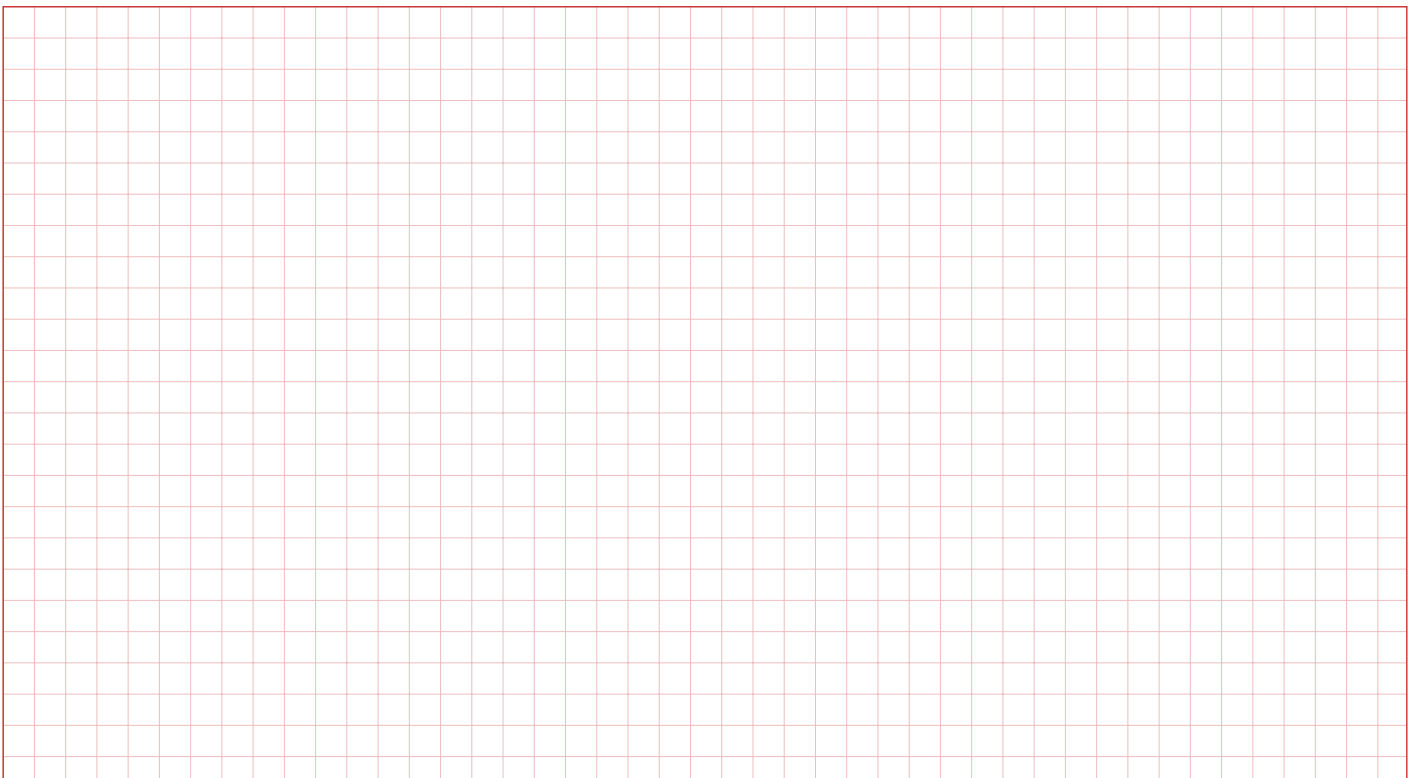
### ► visitFileFailed

```
74 @Override public FileVisitResult visitFileFailed(  
75     Path file, IOException exc) {  
76     err.println(exc.getMessage());  
77     return FileVisitResult.TERMINATE;  
78 }
```

FileSearchExample.java

- Gibt **Fehler** aus
- Teilt Aufrufer mit **abzubrechen** (↗ `FileVisitResult.TERMINATE`)

## Notizen



## Beispiel: Dateien suchen

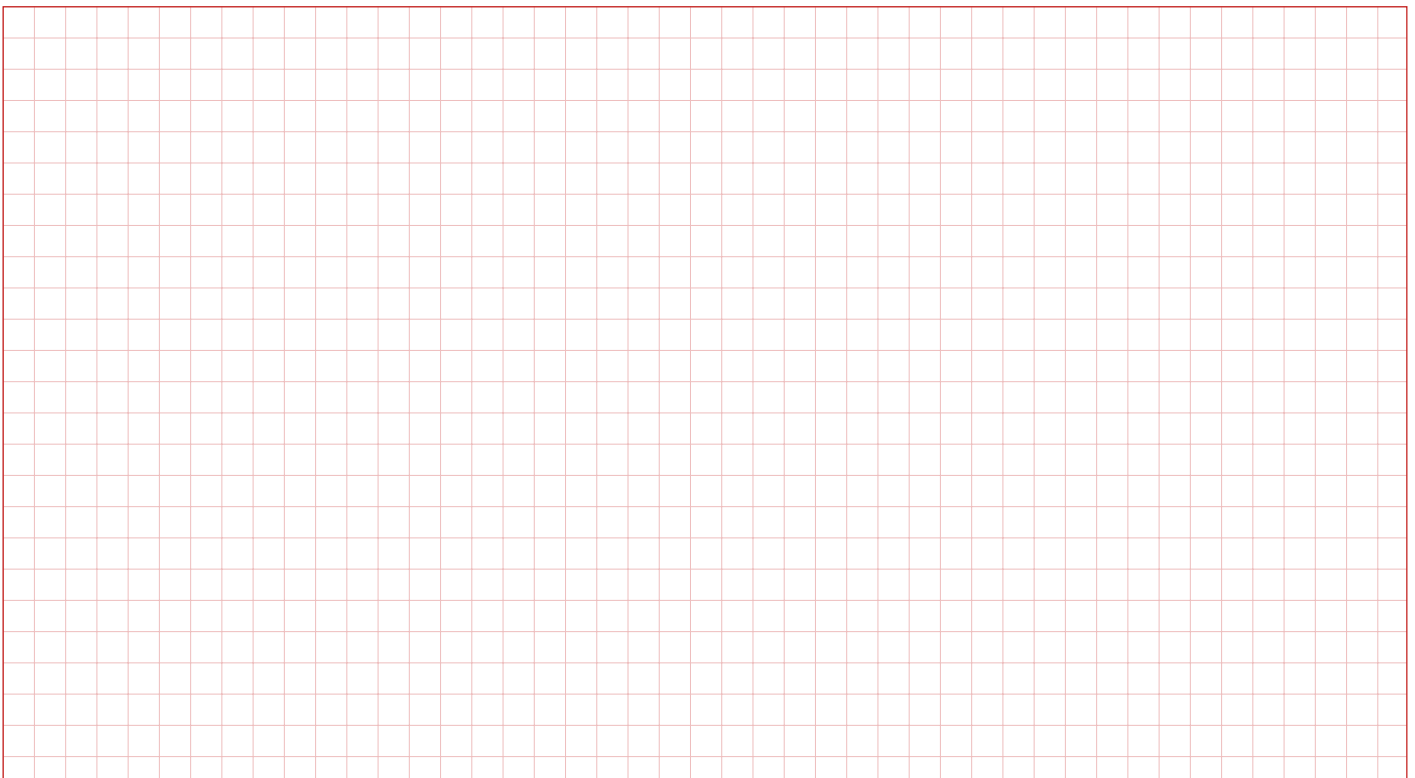
### ► Aufruf in main

```
19 public static void main(String[] args) throws IOException {  
20     Path start = Paths.get(args[0]);  
21     String searchKey = args[1];  
23     FileSearchVisitor visitor = new FileSearchVisitor(searchKey);  
25     Files.walkFileTree(start, visitor);  
27     out.println("Found files:");  
28     for (Path p : visitor.getFoundFiles())  
29         out.println(p);  
30 }
```

FileSearchExample.java

- Erstellt Instanz von `FileSearchVisitor`
- Aufruf von `Files.walkFileTree` mit `FileSearchVisitor`

## Notizen



# Inhalt

## Zusammenfassung

# Ein- und Ausgabe

# NIO vs. Streams

## Notizen



## Inhalt

## Zusammenfassung

### Ein- und Ausgabe

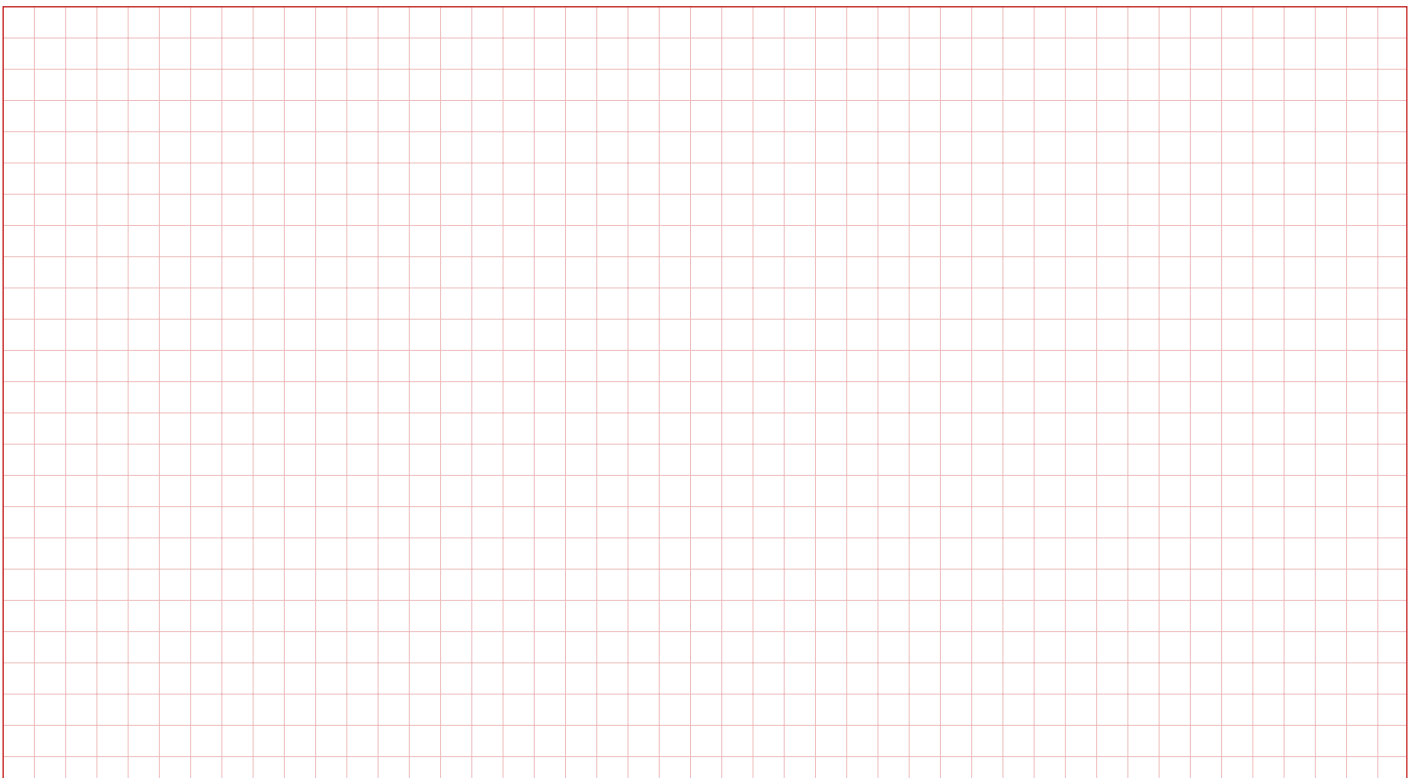
## Notizen

## Zusammenfassung

<b>InputStream</b>	<b>OutputStream</b>
+ read(...)	+ write(...)
<b>Reader</b>	<b>Writer</b>
+ read(...)	+ write(...)

- ▶ **Byteströme:** [InputStream](#) und [OutputStream](#)
  - ▶ Lesen und Schreiben von **byte-Arrays** (Binärdaten)
- ▶ **Text:** [Reader](#) und [Writer](#)
  - ▶ Lesen und Schreiben von **char-Arrays**
  - ▶ Textdaten mit **Encoding** ([Charset](#))
- ▶ **Quellen/Senken:** [System.in/out](#), Dateien, Netzwerk, Puffer
- ▶ **Filter:** Kompression, Verschlüsselung, Digests (z.B. MD5), ...
- ▶ **Formatierte Ausgabe:** [PrintStream](#) und [PrintWriter](#)
- ▶ **Parsing:** manuell, [Scanner](#)

## Notizen

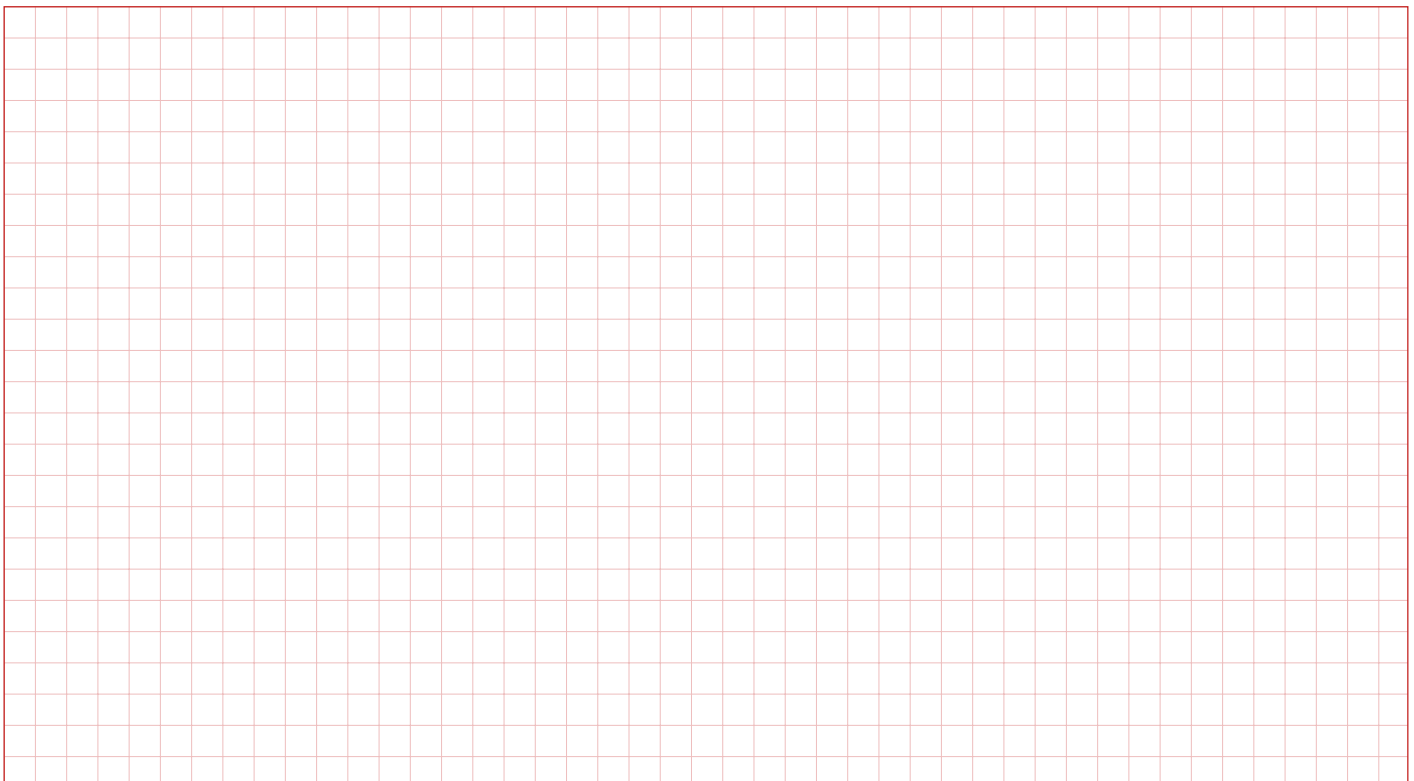


## Zusammenfassung

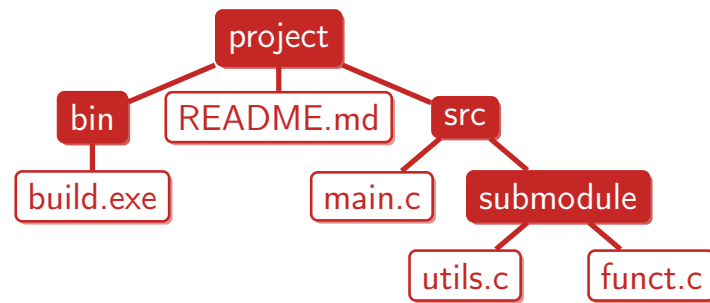
```
try (var in = new FileInputStream("input.txt");
    var out = new FileOutputStream("output.txt")){
    /* ... */
}
```

- ▶ „Automatic Resource Management“: **try-with**
  - ▶ Ressourcen müssen **close()** freigegeben werden
  - ▶ Problematisch bei Ausnahmen
  - ▶ **try-with** garantiert Aufruf von **close()**
- ▶ Interfaces [Closeable](#) und [AutoCloseable](#)
  - ▶ Definieren **close()**
  - ▶ Ermöglicht Verwendung in **try-with**
  - ▶ [Closeable](#) ist restriktiver (**throws** `IOException`)
  - ▶ [AutoCloseable](#) ist allgemeiner (**throws** `Exception`)

## Notizen



## Zusammenfassung



- ▶ Dateien und Verzeichnisse
- ▶ [Path](#)-Klasse modelliert allgemeine Datei-/Verzeichnispfade
- ▶ [Files](#)-Klasse
  - ▶ Abfrage von **Eigenschaften** (isDirectory, isExecutable, etc.)
  - ▶ **Dateioperationen** (copy, move, etc.)
  - ▶ ...
- ▶ **Durchsuchen** von Verzeichnissbäumen mit [Files.walkFileTree](#)
  - ▶ Implementieren von [FileVisitor](#)-Interface
  - ▶ [FileVisitor](#)-Methoden werden von walkFileTree **aufgerufen**

## Notizen



## Inhalt

## Zusammenfassung

### NIO vs. Streams

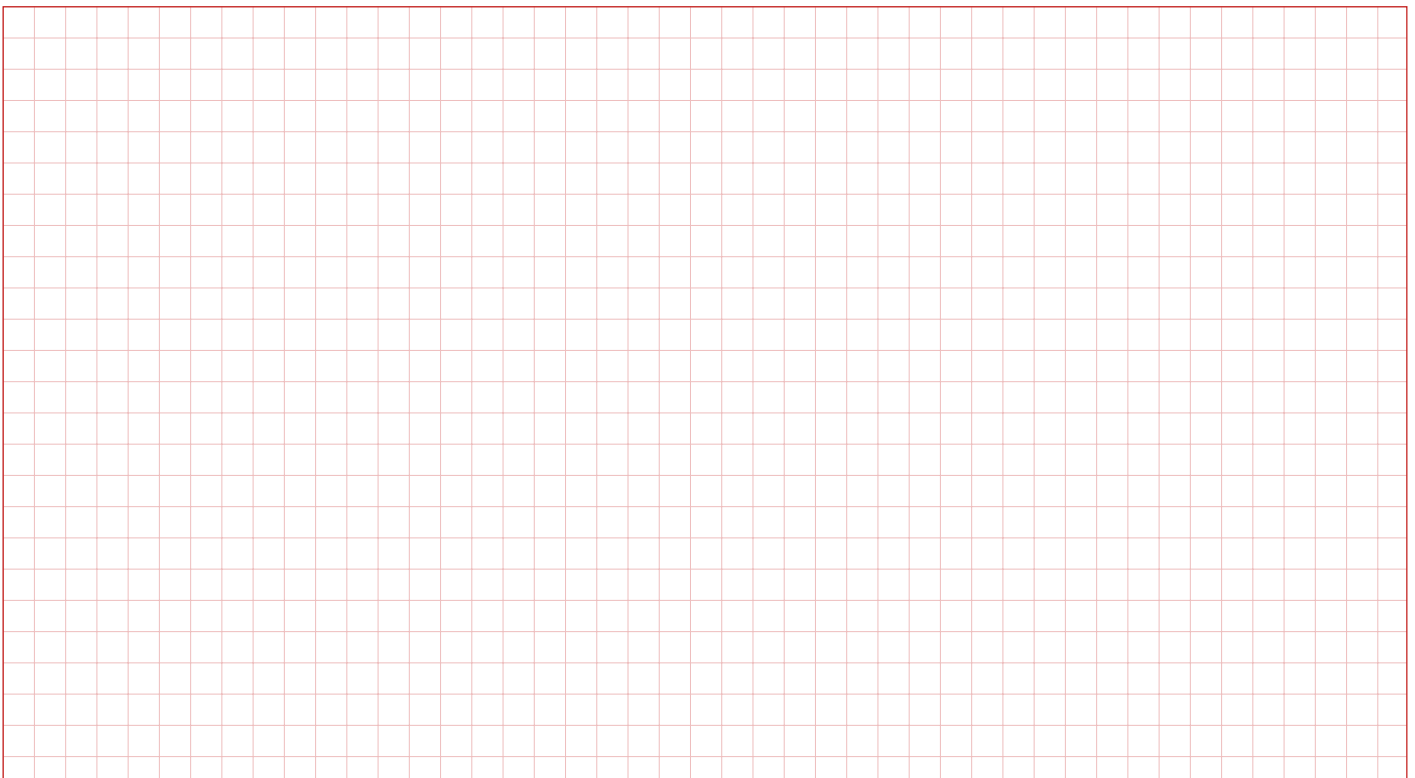
173

## Notizen

## NIO vs. Streams

- ▶ Historie
    - ▶ Zuerst `java.io` und **Streams** (hier vorgestellt)
    - ▶ Dann `java.nio` mit [↗](#) **Channels**
  - ▶ Zur Erinnerung
    - ▶ `read/write/...` **blockieren** auf Streams
- ```
in.read(buffer);
```
- ▶ Ausführer **Thread** (Programmfaden) ist **blockiert**
  - ▶ Beispiel: **Internetserver** hat viele **Verbindungen** (Streams)
    - ▶ Für jede Verbindung **ein Thread** (der blockiert)?
    - ▶ Zu **viele Threads** und **Ressourcenverschwendung**
    - ▶ **Skaliert nicht!**
  - ▶ Anderer **Mechanismus** gefragt → NIO
  - ▶ (Einfache Anwendungen: Streams **ausreichend!**)

## Notizen



## NIO vs. Streams

| Streams           | java.nio          |
|-------------------|-------------------|
| Stream-orientiert | Puffer-orientiert |
| blockierend       | nicht-blockierend |
| —                 | ↗ Selector        |

- ▶ Datenverarbeitung
  - ▶ **Stream-orientiert:** **bytes/chars** werden gelesen und sind dann **verarbeitet**
  - ▶ **Puffer-orientiert:** Daten liegen in **Puffer** und erlauben **vor- und zurückspringen**
- ▶ Blockierend
  - ▶ **Stream**-Methodenaufrufe können **blockieren**
  - ▶ **Channel**-Methodenaufrufe **blockieren nicht**
- ▶ ↗ **Selector**
  - ▶ Überwacht mehrere ↗ **Channel** s in einem **Thread**
  - ▶ Ermittelt welcher ↗ **Channel** Daten **lesen/schreiben** kann

## Notizen

