



## 28. Funktionale Interfaces und Lambda-Ausdrücke

**ARINKO<sup>®</sup>**

- Ein Interface, das nur eine abstrakte Methode hat, nennt man **funktionales Interface** (da es nur eine Daseinsberechtigung, nur eine "Funktion" hat).
- Ob ein funktionales Interface tatsächlich als solches zu sehen ist, ergibt sich erst in der Betrachtung von Implementationsmöglichkeiten: wenn das Interface de facto mit einer Methode implementierbar ist, ist es ein funktionales Interface.
- Methoden, die nicht in die Beurteilung einfließen:
  - (indirekt) durch Object geerbte
  - statische Methoden
  - default Methoden
- Funktionale Interfaces können zur Kenntlichmachung über eine Annotation `@FunctionalInterface` ausgezeichnet werden.
- Die Annotation hat den Nebeneffekt, sollte kein solches funktionales Interface vorliegen, wird ein Compiler-Fehler gemeldet (hat also gewissen verifizierenden aber vorwiegend dokumentatorischen Charakter, analog zu `@Override`)

```
public interface Comparable<T>
{
    public int compareTo(T o);
}
```

nur eine Methode

```
public interface Comparator<T>
{
    public int compare(T o1,T o2);
    boolean equals(Object obj);
}
```

nur eine, nicht in Object  
vorkommende Methode

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T t);
}
```

eine Methode +  
Annotation

```
interface Foo<T, N extends Number>
{
    void m(T arg);
    void m(N arg);
}
```

nicht funktional, 2  
verschiedene Methoden

```
interface Bar extends Foo<String, Integer>
{
}
```

nicht funktional, de facto 2  
verschiedene Methoden

```
@FunctionalInterface
interface Baz extends Foo<Integer, Integer>
{
}
```

funktional, de facto nur  
eine Methode

- Ein Funktionstyp ist eine gewisse Sichtweise auf die Signatur und den Rückgabewert einer Methode, d.h. er beschreibt
  - die Parameterliste und deren Typen
  - den Rückgabebetyp
  - die Exceptions
- Der Funktionstyp ist relevant, um zu bestimmen, mit welcher Typisierung alle abstrakt geerbten Methoden eines funktionalen Interfaces realisiert werden müssten, damit nur eine einzige Methode (sozusagen die Schnittmenge) dabei herauskommt.
- Der Funktionstyp wird gerne so notiert:

`(Parameter) -> Rückgabewert (throws Exception)`

- Die Schnittmenge wird (grob) für den Satz an Methoden eines potentiellen funktionalen Interfaces wie folgt bestimmt:
  - die Implementierung hat eine Subsignatur von allen Methoden
  - der Rückgabewert ist ein Subtyp aller Rückgabewerte
  - die throws-Klausel enthält alle Exception-Typen, die in den throws-Klauseln des Methodensatzes genannt wurden und die zueinander typkompatibel sind

```
interface A
{
    List<String> foo(List<String> arg1, Integer arg2) throws IOException;
}

interface B
{
    List foo(List<String> arg1, Integer arg2) throws ClassNotFoundException,
                                                EOFException;
}

interface C extends A, B {}
```

- Der Funktionstyp von C ist:

```
(List<String>, Integer) -> List<String> (throws EOFException)
```

- Funktionale Programmierung ist ein Programmierparadigma, analog zu prozeduraler Programmierung oder objektorientierter Programmierung.
- Während prozedurale und objektorientierte Sprachen der **imperativen Programmierung** zugeordnet werden, zählt die funktionale Programmierung zu der **deklarativen Programmierung**.
- Imperativ: Abfolge von nacheinander auszuführenden Anweisungen
- Deklarativ: Spezifikation, *was* berechnet werden soll.
- Historische Grundlage ist der Lambda-Kalkül von Church
- Funktionale Programmierung arbeitet gerne mit Rekursion und hat i.d.R. keine Seiteneffekte (also keine Zustandsänderungen während eines Funktionsaufrufs)

- Java wurde mit der Version 8 "aufgerüstet" um Elemente funktionaler Programmierung.
- Java ist aber eine imperative, objektorientierte Sprache, was eigentlich ein Widerspruch ist.
- Daher sind funktionale Elemente in Java mit Vorsicht zu genießen, da weiterhin mit Objektzuständen gearbeitet werden kann!
- Java bietet die funktionalen Elemente dem Entwickler aus folgenden Gründen an:
  - Vermeidung von "boilerplate code" bei anonymen Klassen
  - Vermeidung von Methodendeklarationen
  - Vermeidung von Objektinstanziierungen
- Der Weg, der in Java beschritten wurde, nutzt die sog. **Lambda-Ausdrücke**.
- Anonyme Klassen können beispielsweise durch Lambda-Ausdrücke ersetzt werden, wenn sie aus einem funktionalen Interface stammen.
- Lambda-Ausdrücke sind aber keine Kurzschreibweise für anonyme Klassen, sie funktionieren nur nach ähnlichen Regeln
- Der Compiler generiert keine Artefakte für Lambda-Ausdrücke.



- Syntaktisch besteht ein Lambda-Ausdruck aus
  - einer durch Komma getrennten und optional von Klammern umgebenen Parameterliste,
  - dem Pfeilsymbol ->
  - gefolgt von einer Anweisung bzw. einem Block mit Anweisungen.

```
p -> expression;  
p -> { block }  
(p1, p2) -> expression;  
(p1, p2) -> { block }
```

- Lambda-Ausdrücke können verwendet werden
  - bei Zuweisungen
  - als Übergabeparameter
  - als Return-Wert
  - in einem Cast-Context

| Syntax                                | OK? | Erklärung  |
|---------------------------------------|-----|--|
| <code>(int x) -&gt; x+1</code>        | ✓   |  |
| <code>int x -&gt; x+1</code>          | ✗   | runde Klammern sind bei Typangabe erforderlich             |
| <code>(x) -&gt; x+1</code>            | ✓   | Compiler deduziert fehlenden Parametertyp aus dem Kontext. |
| <code>x -&gt; x+1</code>              | ✓   |  |
| <code>(int x, int y) -&gt; x+y</code> | ✓   |  |
| <code>(x,y) -&gt; x+y</code>          | ✓   |  |
| <code>x,y -&gt; x+y</code>            | ✗   | runde Klammern sind bei mehreren Parametern erforderlich   |
| <code>(x, int y) -&gt; x+y</code>     | ✗   | nicht mischen  |
| <code>() -&gt; 42</code>              | ✓   |  |
| <code>() -&gt; {}</code>              | ✓   | Keine Parameter, kein Rückgabebetyp                        |

- Lambdas sind Implementierungen von funktionalen Interfaces bzw. repräsentieren Instanzen eines funktionalen Interfaces.
- Dabei müssen sie intern nicht notwendigerweise Instanzen sein (der Compiler kann Lambdas z.B. auch als Methoden in der nutzenden Klasse implementieren, der Aufruf findet dann i.d.R. mit der JVM-Direktive `invokedynamic` statt – eine Aufrufart, die statische Typprüfung umgeht und Typsicherheit erst zur Laufzeit sicherstellt).
- Wichtige funktionale Interfaces für die Verwendung mit Lambdas:

```
java.util.Comparator  
java.util.function.Consumer  
java.util.function.Supplier  
java.util.function.Predicate  
java.util.function.Function
```

- Ein Consumer "verarbeitet" Daten und führt dazu beliebige Aktionen aus.
- Mit der Methode andThen können mehrere Aktionen verkettet werden.

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after){}
}
```

- Beispiel im Warenkatalog:

```
public void printAll()
{
    this.waren.forEach(w -> System.out.printf("%20s %6.2f EUR%n",
                                                w.getBezeichnung(), w.getPreis()));
}
```

- Die (neue) Methode forEach von Collection führt eine Aktion (einen Consumer) auf allen Elementen aus.

- Prädikate überprüfen Bedingungen und dienen vornehmlich der Filterung.

```
@FunctionalInterface
interface Predicate<T>
{
    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {}
    default Predicate<T> negate() {}
    default Predicate<T> or(Predicate<? super T> other) {}
    static <T> Predicate<T> isEqual(Object targetRef) {}
}
```

- Collections haben beispielsweise eine neue Funktionalität, die Elemente entfernt, wenn sie einer gewissen Bedingung genügen:

```
public void removeTooCheap(final double threshold)
{
    this.waren.removeIf(w -> w.getPreis() < threshold);
}
```

- Funktionen dienen der Transformation/Zuordnung.

```
@FunctionalInterface
interface Function<R, T>
{
    R apply(T t);

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before){}
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {}
    static <T> Function<T, T> identity() {}
}
```

- R ist der Resultatstyp, T der Eingangstyp.

```
final String[] namen = { "null", "eins", "zwei", ..., "neun" };
Map<Integer, String> zahlen = new HashMap<>();

for (int i = 0; i < 10; i++)
{
    String text = zahlen.computeIfAbsent(i, z -> namen[z]);
    System.out.println(text);
}
```

- Schwierige Testbarkeit (schwer isoliert zu testen)
- Wart- und Lesbarkeit *können* erschwert werden, v.a. bei komplexen und geschachtelten Ausdrücken.
- Schwieriges Debugging, da viel Information erst zur Laufzeit vorliegt.
- Fehlerbehandlung: Lambdas werden bei 0 beginnend hochgezählt...

Exception in thread "main" java.lang.ArithmeticException: / by zero  
at vortrag11.FunctionDemo.lambda\$2(FunctionDemo.java:33)  
at java.util.stream.ReferencePipeline\$3\$1.accept(...)

```
public static void main(String[] args)
{
    Collection<Integer> numbers = Arrays.asList(-3, -2, -1, 0, 1, 2, 3);
    int threshold = 24;

    List<Integer> list = numbers.stream().filter(number -> number >= 0)
        .map(number -> threshold / number)
        .filter(number -> number > 0)
        .collect(Collectors.toList());
    list.forEach(o -> System.out.println(o));
}
```

Zeile 33

- Eine Methodenreferenz wird benutzt, um auf den Aufruf einer Methode Bezug zu nehmen, ohne diesen Aufruf tatsächlich direkt durchzuführen.
- Die Evaluation einer Methodenreferenz (genauer: eines Methodenreferenz-Ausdrucks) erzeugt eine Instanz eines funktionalen Interfaces. Die Evaluation triggert nicht den Aufruf der entsprechenden Methode.
- Im Gegenteil: eine Methodenreferenz erlaubt es, den tatsächlichen Aufruf auf einen späteren Zeitpunkt zu verschieben.
- Die wichtigsten Methodenreferenzarten:

| Art der Referenz                             | Syntax                           |
|--|----------------------------------|
| Klassen-Methode                              | <code>Class::staticMethod</code> |
| Instanz-Methode                              | <code>Instance::method</code>    |
| Instanz-Methode ohne konkrete Instanzbindung | <code>Class::method</code>       |
| Konstruktor                                  | <code>Class::new</code>          |




- Methodenreferenzen können eingesetzt werden
  - wenn die Parameterliste mit der Parameterliste einer gegebenen Methode übereinstimmt oder
  - wenn die Parameterliste aus dem Kontext der umgebenden Methode und der gegebenen Methode ableitbar sind.

```
public void sortiere(boolean aufsteigend)
{
    Comparator<Ware> comp = null;

    if (aufsteigend)
        comp = Warenkatalog::vergleicheNachPreis;
    else
        comp = (w1, w2) -> Double.compare(w2.getPreis(), w1.getPreis());

    this.waren.sort(comp);
}

public static int vergleicheNachPreis(Ware w1, Ware w2)
{
    return Double.compare(w1.getPreis(), w2.getPreis());
}
```



Methodenreferenz

- Statische Methode

```
List<Ware> waren = warenkatalog.getWaren ();  
waren.forEach(System.out::println);
```

- Instanz-Methode

```
List<Ware> waren = warenkatalog.getWaren();  
  
ArrayList<Ware> list = new ArrayList<Ware>();  
waren.forEach(list::add);
```

- Instanz-Methode mit späterer Bindung

```
List<Ware> waren = warenkatalog.getWaren();  
  
waren.forEach(Ware::formatiertAusgeben);
```

## ■ Konstruktor

```
@FunctionalInterface
public interface WareProducer
{
    Ware produce(String nummer, String bezeichnung, double preis);
}
```

```
public class Ware
{
    ...
    public Ware(String nummer, String bezeichnung, double preis)
    {
    }
}
```

Die Analogie zum  
Konstruktor ist wichtig

```
List<Ware> waren = new ArrayList<>();
WareProducer wp = Ware::new;

waren.add(wp.produce("01019010", "Hammer", 19.00));
waren.add(wp.produce("01019020", "Zange", 17.00));
waren.add(wp.produce("01019030", "Schraubendreher", 12.00));
```



## 29. Stream-API

**ARINKO<sup>®</sup>**

- Sehr häufig hat man den Fall, dass man über einen gewissen „Strom“ an Daten verfügt, der abgearbeitet werden muss.
- Beispiele:
  - in einer Datei soll Zeile für Zeile ein gewisses Format ausgelesen und konvertiert werden
  - aus einer Datenbank werden einige gleichartige Daten gelesen, die verarbeitet werden sollen
  - in einer Collection liegen Daten vor, die iteriert und verarbeitet werden sollen
- Problem 1: in imperativen Programmen kann hier der Code schnell lang werden und die Tendenz zur Unübersichtlichkeit bekommen (vorsichtig ausgedrückt)
- Problem 2: sollte man zukünftig eine parallele Datenverarbeitung realisieren wollen, muss der Code komplett neu strukturiert werden.

- Waren einer bestimmten Kategorie sollen rabattiert werden.

```
public void rabattiereKategorie(Warenkategorie kategorie, double rabatt)
{
    for (Ware w : this.waren)
    {
        if (w.getKategorie() == kategorie)
        {
            double preis = w.getPreis();
            double neuerPreis = preis * (1-rabatt);
            w.setPreis(neuerPreis);
        }
    }
}
```

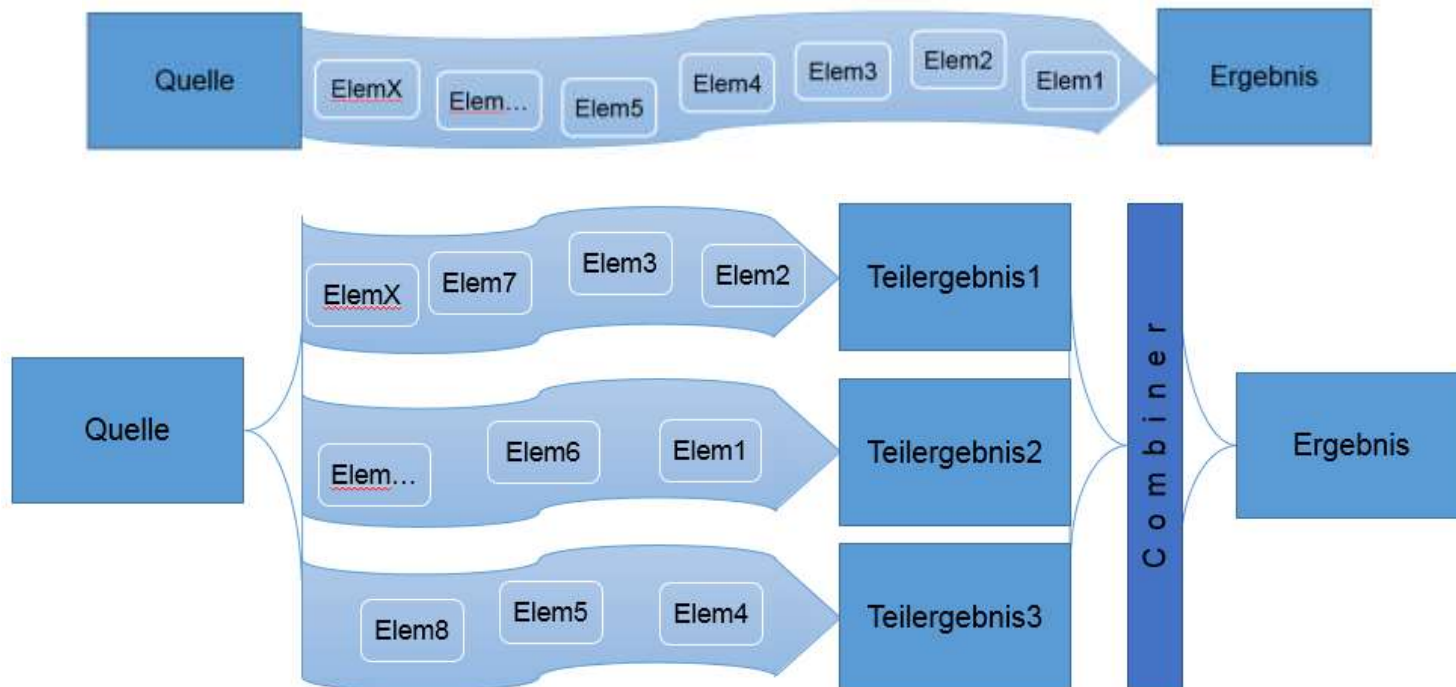
Iterieren

Filtern

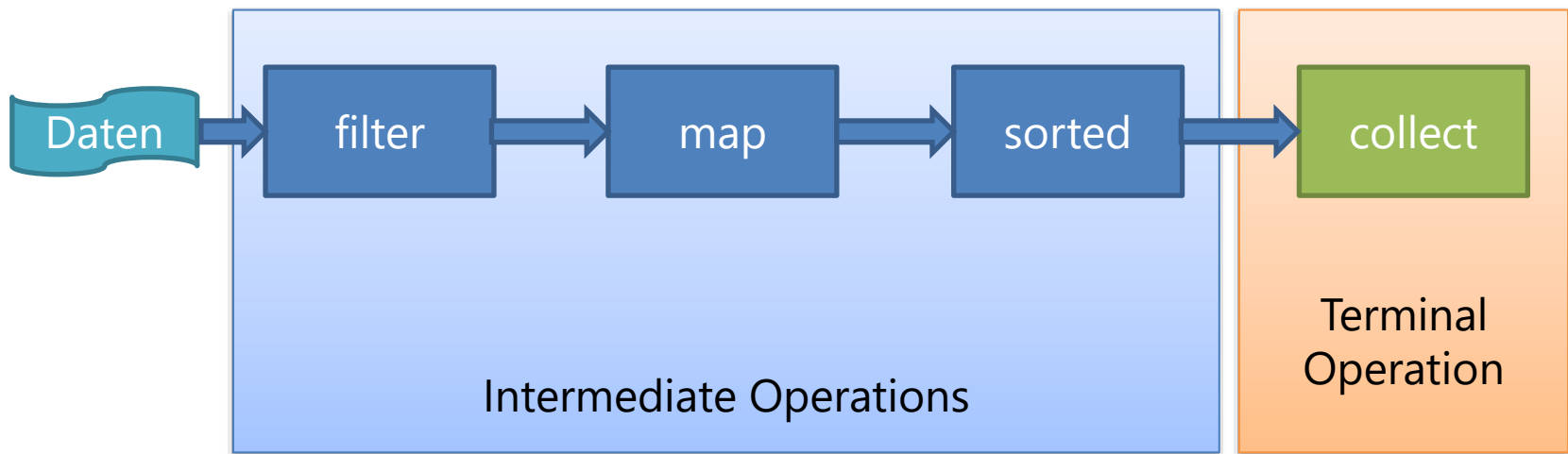
Verarbeiten

- Da die Waren voneinander unabhängig sind, könnte man die Daten auch parallel verarbeiten. Dazu müsste das Programm aber komplett umstrukturiert werden...

- Mit Java 8 wurde begleitend zu den Lambda-Ausdrücken auch das Stream-API eingeführt.
- Ein Stream besteht aus einer Folge von gleichartigen Elementen.
- Operationen auf Streamelementen können seriell (sequenziell) oder parallel erfolgen.



- Operationen auf Streams unterscheiden sich in
  - **intermediate operations**  
verarbeitet ein einzelnes Objekt aus dem Stream, die Methode gibt ein neues Streamobjekt für verkettete Weiterverarbeitung zurück
  - **terminal operations**  
beenden die Verarbeitung, z.B. indem sie einen Wert oder ein resultierendes Objekt konstruieren
- Grundsätzliche Arbeitsweise ist das Verketteten verschiedener intermediate Operations mit Abschluss durch eine terminal Operation, z.B.





|                         | Operation | Resultat          |
|-------------------------|-----------|-------------------|
| Intermediate Operations | distinct  | Stream<T>         |
|                         | filter    | Stream<T>         |
|                         | limit     | Stream<T>         |
|                         | map       | Stream<R>         |
|                         | sorted    | Stream<T>         |
| Terminal Operations     | allMatch  | boolean           |
|                         | anyMatch  | boolean           |
|                         | collect   | <i>spezifisch</i> |
|                         | count     | long              |
|                         | forEach   | void              |
|                         | reduce    | <i>spezifisch</i> |

- **filter** – die Daten im Stream werden der Prüfung durch ein Filterkriterium unterzogen. Der Resultatsstream enthält durch noch die Objekte, die der Filterbedingung genügen.

```
Stream<T> filter(Predicate<? super T> predicate);
```

- **sorted** – parameterlos oder mit Comparator aufrufbar. Der Resultatsstream enthält alle Elemente in der gewünschten Reihenfolge.

```
Stream<T> sorted();  
Stream<T> sorted(Comparator<? super T> comparator);
```

- Beispiel: filtere alle Waren, die weniger als 15 Euro kosten, heraus und sortiere die restlichen.

```
Stream<Ware> sortedWareStream =  
    this.waren.stream().filter(w -> w.getPreis() > 15.0).sorted();
```

- `distinct` – es wird ein neuer Stream generiert, der keine Duplikate (ermittelt mit `equals`) enthält.

```
Stream<T> distinct();
```

- `limit` – übernimmt nur die ersten `n` Elemente eines Streams in einen neuen Stream.

```
Stream<T> limit(long maxSize);
```

- Beispiel: eliminiere alle Duplikate und gebe davon maximal 10 Sätze zurück.

```
Stream<Ware> limitedStream = this.waren.stream().distinct().limit(10);
```

- map – Unterziehe die Daten einer transformierenden Function.
- Sollen Stream-Elemente auf primitive Datentypen abgebildet werden, gibt es dafür separate Methoden:

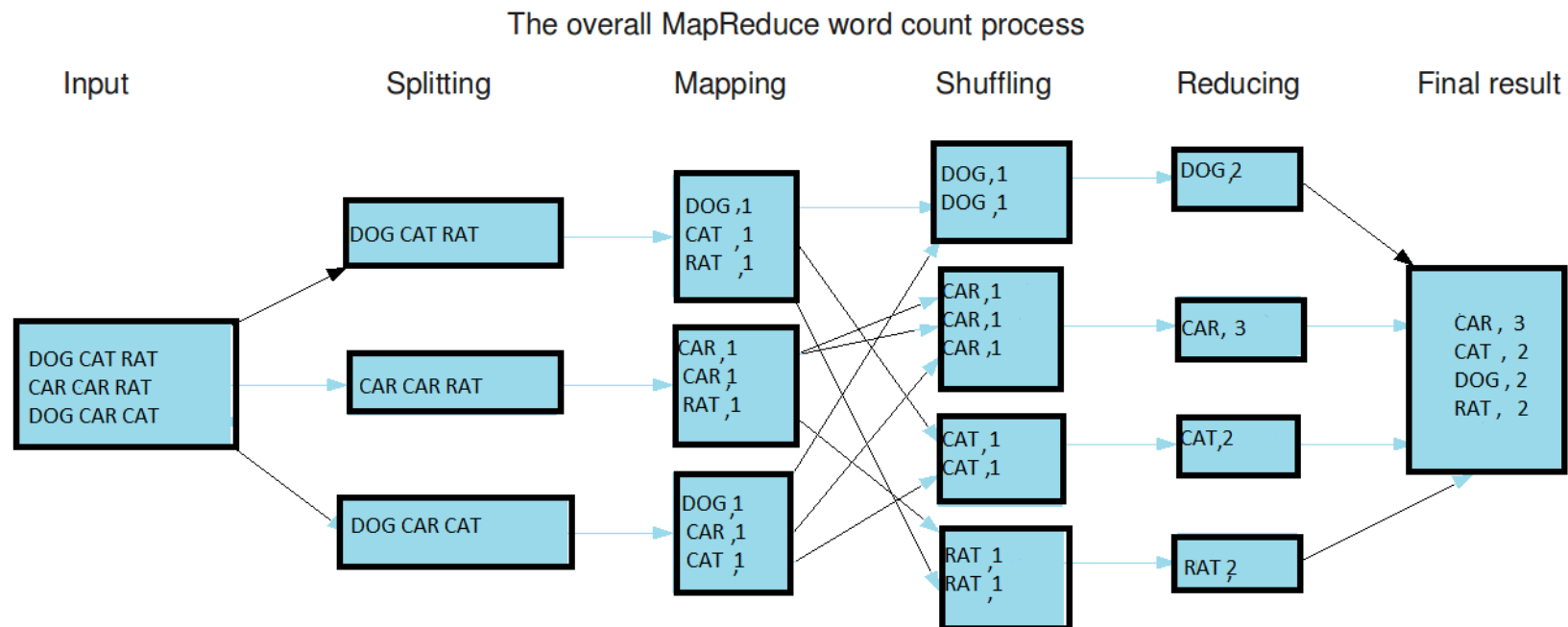
```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);  
IntStream mapToInt(ToIntFunction<? super T> mapper);  
LongStream mapToLong(ToLongFunction<? super T> mapper);  
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);  
  
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> m);  
IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper);  
LongStream flatMapToLong(Function<? super T, ? extends LongStream> mapper);  
DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> m);
```

- flatMap-Methoden nehmen durch die Funktion produzierte Streams auf und fügen diese in den gemeinsamen linearen Datenstrom ein.
- Beispiel: erzeuge einen Double-Datenstrom, der nur noch aus den Preisen der Waren besteht:

```
DoubleStream ds = this.waren.stream().mapToDouble(w -> w.getPreis());
```

- MapReduce ist ein Pattern für die (idealerweise parallele) Abarbeitung von großen Datenmengen.
- Es wird in vielen Produkten und Frameworks implementiert (Stichworte: Apache Hadoop und Spark)
- Die üblichen Schritte sind:
  1. Verteile die zu verarbeitenden Daten (parallel) („Split“-Phase)
  2. Führe auf den Daten transformierende, analysierende Operationen aus („Map“-Phase)
  3. Sammle Daten, die auf dem gleichen System verarbeitet werden sollen ein und gebe Daten, die für andere Systeme sind weiter („Shuffle“)
  4. Reduziere die angefallenen Daten auf Zwischenergebnisse („Reduce“-Phase)
- Die Aufteilung der Daten und die Zusammenführung sind vom jeweiligen Produkt implementiert und bestimmen maßgeblich dessen Performance und Leistungsfähigkeit.
- Die Phasen „Map“ und „Reduce“ werden vom Entwickler bestimmt.

- Quelle: <http://a4academics.com/tutorials/83-hadoop/840-map-reduce-architecture>



- reduce – entspricht der Phase aus dem MapReduce-Pattern und soll den Strom auf einen Ergebniswert abbilden. Sie gibt es in 3 Formen, die am häufigsten verwendete ist

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

- identity – der Ausgangswert für die Akkumulation (und auch das Ergebnis, falls der Stream leer wäre)
- accumulator – eine Funktion, die einen weiteren Stream-Wert zum bereits akkumulierten Ergebnis hinzufügt
- Beispiel: berechne den Gesamtpreis aller Waren

```
Double gesamtpreis = this.waren.stream()  
    .map(w -> w.getPreis()).reduce(0.0, (p, n) -> p + n);
```

- In diesem Ausdruck ist

0.0 – der Ausgangswert

p – der bisher berechnete Wert

n – der neu hinzukommende Wert

- `collect` – gibt es in 2 Varianten. In der häufiger verwendeten werden die im Strom angesammelten Daten durch einen Collector aufgenommen.

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

- In der Klasse `java.util.stream.Collectors` gibt es Hilfsmethoden, die Collectoren anbieten, die beispielsweise sammelnde Collections erzeugen.
- Beispiel: filtere eine Liste von Waren nach Preisen > 15 Euro, sortiere alles und gebe das Resultat in eine Collection:

```
List<Ware> list = this.waren.stream().filter(w -> w.getPreis() > 15.0)  
                        .sorted().collect(Collectors.toList());
```



- `forEach` – erlaubt, über einem Stream nochmals abschließend eine Aktion auszuführen.

```
void forEach(Consumer<? super T> action);
```

- `count` – zählt die im Stream befindlichen Elemente

```
long count();
```

- Beispiel: stelle fest, wie viele Waren teurer als 15 Euro sind und gebe deren Bezeichnung aus:

```
long anzahl = this.waren.stream().filter(w -> w.getPreis() > 15).count();  
System.out.println("So viele Waren sind teurer als 15 Euro: " + anzahl);  
  
this.waren.stream().filter(w -> w.getPreis() > 15)  
                .forEach(w -> System.out.println(w.getBezeichnung()));
```

- anyMatch – überprüft, ob zumindest ein Element im Strom der gegebenen Bedingung entspricht. Der Strom muss nicht unbedingt vollständig untersucht worden sein.

```
boolean anyMatch(Predicate<? super T> predicate);
```

- allMatch – überprüft, ob alle im Strom befindlichen Objekte der Bedingung genügen. Auch hier müssen nicht notwendigerweise alle Objekte besucht worden sein.

```
boolean allMatch(Predicate<? super T> predicate);
```

- Beispiel: gibt es überhaupt Waren, die teurer als 50 Euro sind und haben alle Waren einen Preis > 0 Euro?

```
boolean teuer = this.waren.stream().anyMatch(w -> w.getPreis() > 50);  
boolean hatPreis = this.waren.stream().allMatch(w -> w.getPreis() > 0);
```

- Streams können sehr leicht parallel verarbeitet werden.
- Dazu muss nur anstelle von `stream()` die Methode `parallelStream()` eingesetzt werden. Für den Rest sorgt das Stream-API.
- Zu beachten ist, dass durch die Parallelverarbeitung einiges, aber nicht unbedingt alles schneller werden kann. z.B. ist paralleles Sortieren mit darauf folgender Weiterverarbeitung eher problematisch.
- Die Daten werden zur Parallelverarbeitung vom Stream-API automatisch in Teilbereiche zerlegt. Dafür sorgt ein sog. „Spliterator“.
- Man kann eigene Spliterators implementieren, hat i.d.R. aber damit nichts zu tun.
- Hier ein kleiner Micro-Benchmark mit 1.000.000 Waren-Objekten:

```
long time1 = System.currentTimeMillis();
boolean t1 = this.waren.stream().filter(w -> w.getPreis() < 50)
                           .allMatch(w -> w.getPreis() > 0);
long time2 = System.currentTimeMillis();
boolean t2 = this.waren.parallelStream().filter(w -> w.getPreis() < 50)
                           .allMatch(w -> w.getPreis() > 0);
long time3 = System.currentTimeMillis();
```

```
System.out.printf("Sequentiell: %4dms\n", time2-time1);
System.out.printf("Parallel:      %4dms\n", time3-time2);
```

|              |      |
|--------------|------|
| Sequentiell: | 52ms |
| Parallel:    | 34ms |



## 30. IO mit Java

**ARINKO<sup>®</sup>**

- Dieses Kapitel behandelt den Umgang von Java mit Dateien, dem Dateisystem und Dateiinhalten.
- Grundsätzlich wird in den Java-APIs unterschieden zwischen
  - Dateien (bzw. Pfade im Dateisystem) und
  - Dateiinhalten ("Streams", Reader/Writer)
- Das Dateihandling beinhaltet Erzeugen, Kopieren, Löschen, Verschieben und Rechteabfragen.
- Das Datei-Inhalts-Handling beinhaltet das Lesen und Schreiben.

- Wichtige Typen für die Eingabe/Ausgabe befinden sich in den Paketen
  - `java.io.*` (ab Java 1.x)
  - `java.nio.*` (ab Java 1.4)
  - `java.nio.file.*` (ab Java 7)
- Viele Methoden der Klassen in diesen Paketen zwingen den Anwender, ein Exception-Handling durchzuführen.
- Typische Exceptions:
  - `IOException` (Basisklasse)
  - `FileNotFoundException`
  - `InterruptedException`
  - `EOFException`

- File stammt aus Java 1.x.
- Seit Java 7 ist ein etwas komplexeres, aber mächtigeres Geflecht aus Klassen im `java.nio.file` Package vorhanden.
- Die Klasse `File` repräsentiert eine Schnittstelle zum File-System (Ordner und Dateien) der zugrundeliegenden Plattform.
- Ein `File`-Objekt repräsentiert einen Pfadnamen für ein Verzeichnis oder für eine Datei.
- `File`-Objekte sind unveränderlich.
- Daran befinden sich Methoden, um die Existenz von Dateien abzufragen, Verzeichnisse zu erzeugen, Inhalte von Verzeichnissen abzufragen, Dateien zu löschen.

- Bestandteil der Neuerung in Java 7.
- Path repräsentiert, ähnlich wie File, einen Pfad in einem Dateisystem.
- Warum ein zweites Datei-API?

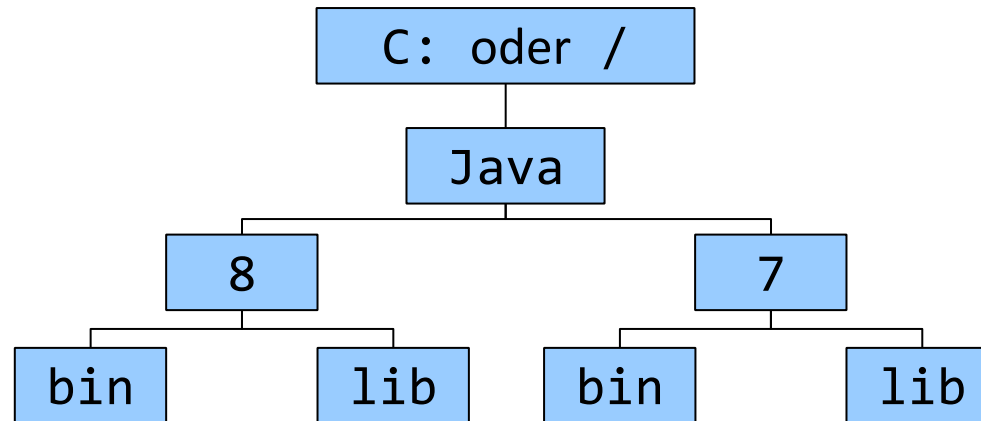
Prior to the Java SE 7 release, the `java.io.File` class was the mechanism used for file I/O, but it had several drawbacks.

- Many methods didn't throw exceptions when they failed, so it was impossible to obtain a useful error message.
- The rename method didn't work consistently across platforms.
- There was no real support for symbolic links.
- More support for metadata was desired, such as file permissions, file owner, and other security attributes.
- Accessing file metadata was inefficient.
- Many of the File methods didn't scale.
- It was not possible to write reliable code that could recursively walk a file tree and respond appropriately if there were circular symbolic links.

Quelle: Oracle Tutorial



- Ein Pfad stellt ein Dateisystem-Objekt (existent oder nichtexistent) dar.



- Beispiele:

/Java/7/bin

C:\Java\8\lib

/Java/9

/

C:\Java\8\bin\extras

- Pfade können konkrete Dateien und Verzeichnisse und auch über symbolische Links erreichbare Objekte repräsentieren.
- Pfade sind Betriebssystem-abhängig (was nicht bedeutet, dass man das Java-Programm nicht unabhängig schreiben kann!)
- Genauer: Pfade beziehen sich auf den File-System-Provider des jeweiligen JREs .
- Pfade können relativ oder absolut angegeben werden.
  - Relativ:  
8/bin                      Java\7\lib
  - Absolut:  
/Java/8/bin                C:\Java\7\lib
- Relative Pfade benötigen immer noch zusätzlichen Kontext, damit sie sinnvoll verwendet werden können.

- Zur einfachen Verwendung von Path (Interface!) existiert die Klasse Paths.

```
public final class Paths
{
    public static Path get(String first, String... more);
    public static Path get(URI uri);
}
```

- Beispiel:

```
Path path = Paths.get("C:\\");
```

```
Path path = Paths.get("/opt/Java");
```

```
Path path = Paths.get(System.getProperty("user.home"), "logs", "test.log");
```

- Einige Funktionalitäten eines Paths (unvollständig):

```
public interface Path extends Comparable<Path>, Iterable<Path>, Watchable
{
    boolean isAbsolute();
    Path getRoot();
    Path getFileName();
    Path getParent();
    int getNameCount();
    Path getName(int index);
    Path subpath(int beginIndex, int endIndex);
    boolean startsWith(Path other);
    boolean startsWith(String other);
    boolean endsWith(Path other);
    boolean endsWith(String other);
    Path normalize();
    URI toUri();
    Path toAbsolutePath();
    Path toRealPath(LinkOption... options) throws IOException;
    File toFile();
    Iterator<Path> iterator();
    int compareTo(Path other);
    String toString();
}
```

```
// Microsoft Windows syntax
Path path = Paths.get("C:\\Java\\SE\\8\\bin\\java.exe");

// Unix syntax
Path path = Paths.get("/Java/SE/8/bin/java");

System.out.format("toString: %s\n", path.toString());
System.out.format("getFileName: %s\n", path.getFileName());
System.out.format("getName(0): %s\n", path.getName(0));
System.out.format("getNameCount: %d\n", path.getNameCount());
System.out.format("subpath(1,3): %s\n", path.subpath(1, 3));
System.out.format("getParent: %s\n", path.getParent());
System.out.format("getRoot: %s\n", path.getRoot());
```

```
toString: C:\Java\SE\8\bin\java.exe
getFileName: java.exe
getName(0): Java
getNameCount: 5
subpath(1,3): SE\8
getParent: C:\Java\SE\8\bin
getRoot: C:\
```

```
toString: /Java/SE/8/bin/java
getFileName: java
getName(0): Java
getNameCount: 5
subpath(1,3): SE/8
getParent: /Java/SE/8/bin
getRoot: /
```

- Die Verbindung abstrakter Pfadinformation zu direkten Tätigkeiten auf dem File-System erledigt die Klasse Files.
- Files ermöglicht auch die Erzeugung von Stream-Objekten, die bis Java 6 ausschließlich über Konstruktoren zu erzeugen waren.
- Einige Methoden:

```
public static long size(Path) throws IOException
public static void delete(Path) throws IOException
public static boolean deleteIfExists(Path) throws IOException
public static Path createDirectory(Path,FileAttribute[]) ...
public static Path createTempFile(Path,String,String,FileAttribute[]) ...
public static boolean exists(Path,LinkOption[])
public static boolean notExists(Path,LinkOption[])
public static boolean isDirectory(Path,LinkOption[])
public static boolean isRegularFile(Path,LinkOption[])
public static boolean isHidden/Executable/Readable/Writable(Path) ...
public static Path copy(Path,Path,CopyOption[]) throws IOException
public static Path move(Path,Path,CopyOption[]) throws IOException
```

```
Path path = Paths.get("test.txt");

if (Files.exists(path) && Files.size(path) == 0)
{
    System.out.println("Wohl leer...");
}
```

```
Path path = Paths.get("test.txt");

if (Files.notExists(path))
{
    Files.createFile(path);
}
else
{
    Path path2 = Paths.get("text-copy.txt");
    Files.copy(path, path2);
}
```

- Der Übergang zwischen "alter" und "neuer" Welt ist sicher noch öfter erforderlich:
  - Code, der alte Klassentypen als Parameter benötigt
  - Code, der alte Klassentypen als Resultat zurückliefert
- Der Übergang ist über Konvertierungs-Methoden möglich.

```
File file = oldCodeReturningFile();  
Path path = file.toPath();  
  
File convertedFile = path.toFile();  
oldCodeDemandingFile(convertedFile);
```



- Das File-API arbeitet plattformunabhängig.
- Pfade können als Windows- oder Unix-Pfade angegeben werden.
- „best practice“ ist, Unix-Pfade zu verwenden.
  - Windows-Pfade brauchen doppelten \ im String (Escape-Sequenz)
- Path ist aber in sich nicht System-unabhängig.
  - Der gleiche Pfad von einem Solaris-System gegen einen Windows-Pfad verglichen, ist nicht gleich!
  - Nur auf dem selben System erzeugte Pfade mit unterschiedlicher Syntax können gleich sein.

- Über die Klasse Files sind auch Meta-Informationen zu Dateien abrufbar
- Viele "übliche" Eigenschaften, sind direkt als Methoden in Files abrufbar, z.B.

`Files.isHidden(path)`

- File-System-Attribute, die spezifisch sind bzw. sein können, werden etwas allgemeiner abgefragt, z.B. alle:

`Map<String, Object> attrs = Files.readAttributes(path, "*");`

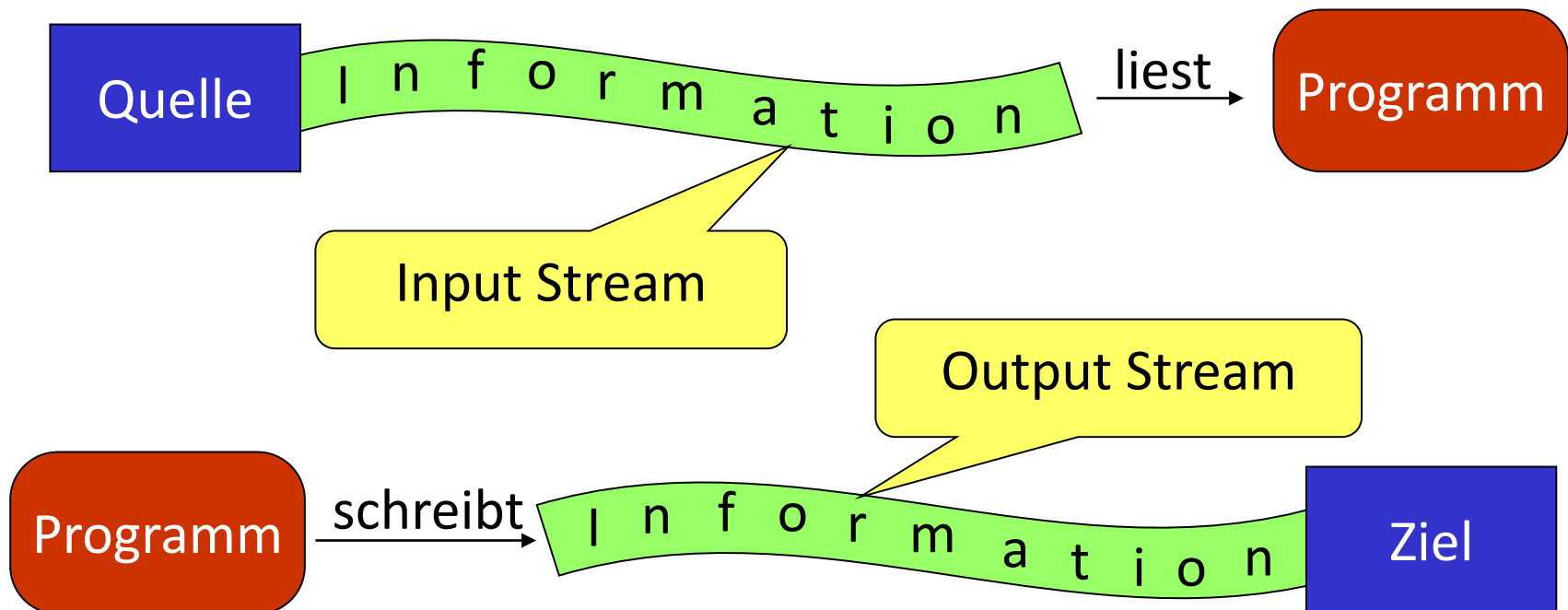
- Attribute werden auch System-spezifisch in sog. Views zusammengefasst:

`BasicFileAttributes, DosFileAttributes, PosixFileAttributes`

```
BasicFileAttributes attr = Files.readAttributes(file,  
                                                BasicFileAttributes.class);  
  
System.out.println("creationTime: " + attr.creationTime());  
System.out.println("lastAccessTime: " + attr.lastAccessTime());  
System.out.println("lastModifiedTime: " + attr.lastModifiedTime());  
System.out.println("isDirectory: " + attr.isDirectory());  
System.out.println("isOther: " + attr.isOther());  
System.out.println("isRegularFile: " + attr.isRegularFile());  
System.out.println("isSymbolicLink: " + attr.isSymbolicLink());  
System.out.println("size: " + attr.size());
```

```
DosFileAttributes attr = Files.readAttributes(file,  
                                                DosFileAttributes.class);  
  
System.out.println("isReadOnly is " + attr.isReadOnly());  
System.out.println("isHidden is " + attr.isHidden());  
System.out.println("isArchive is " + attr.isArchive());  
System.out.println("isSystem is " + attr.isSystem());
```

- Eingabe und Ausgabe in Java beruhen auf einem Stream-Konzept:  
Zwischen Programm und Medium ist ein Stream geschaltet.

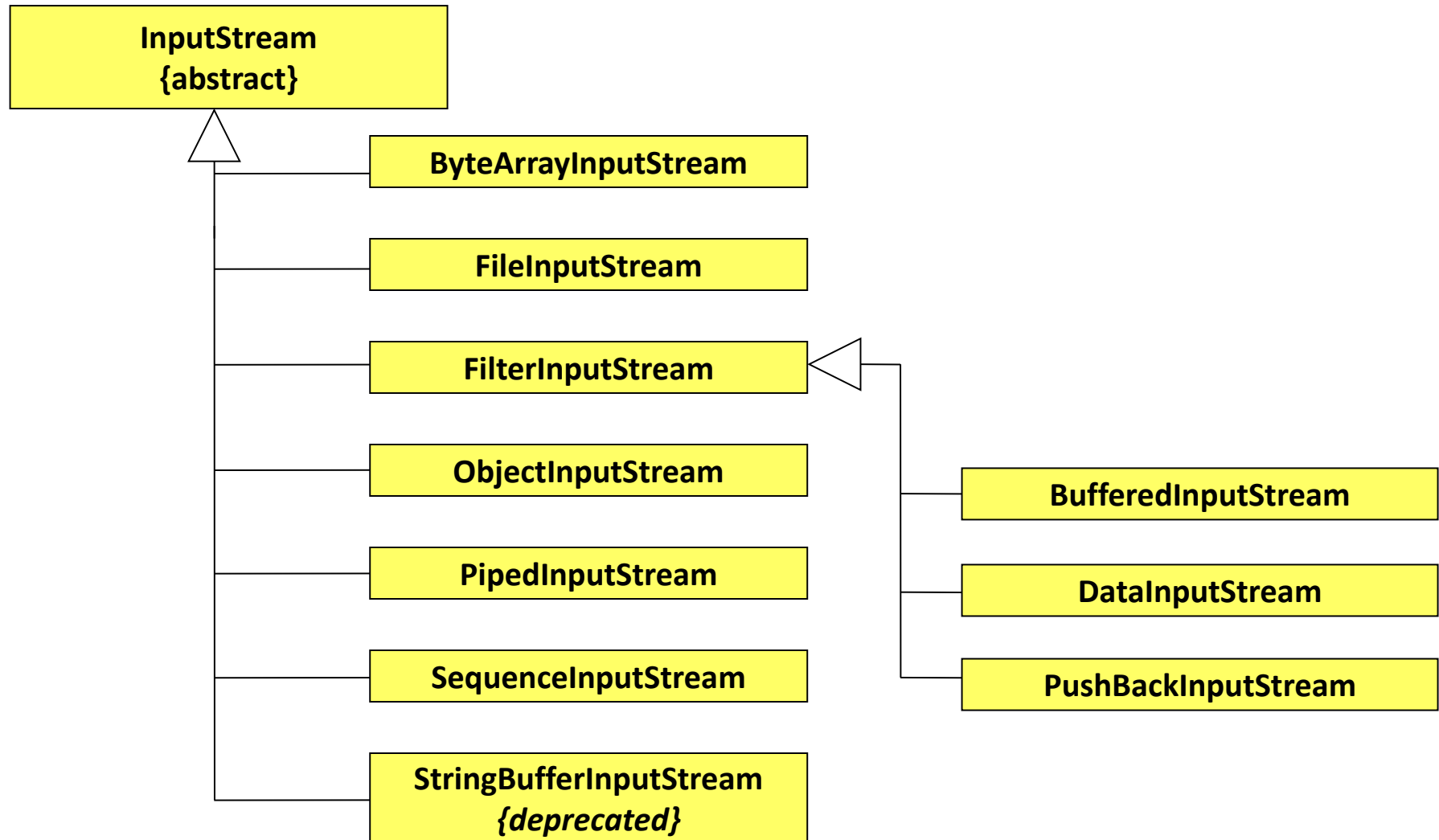


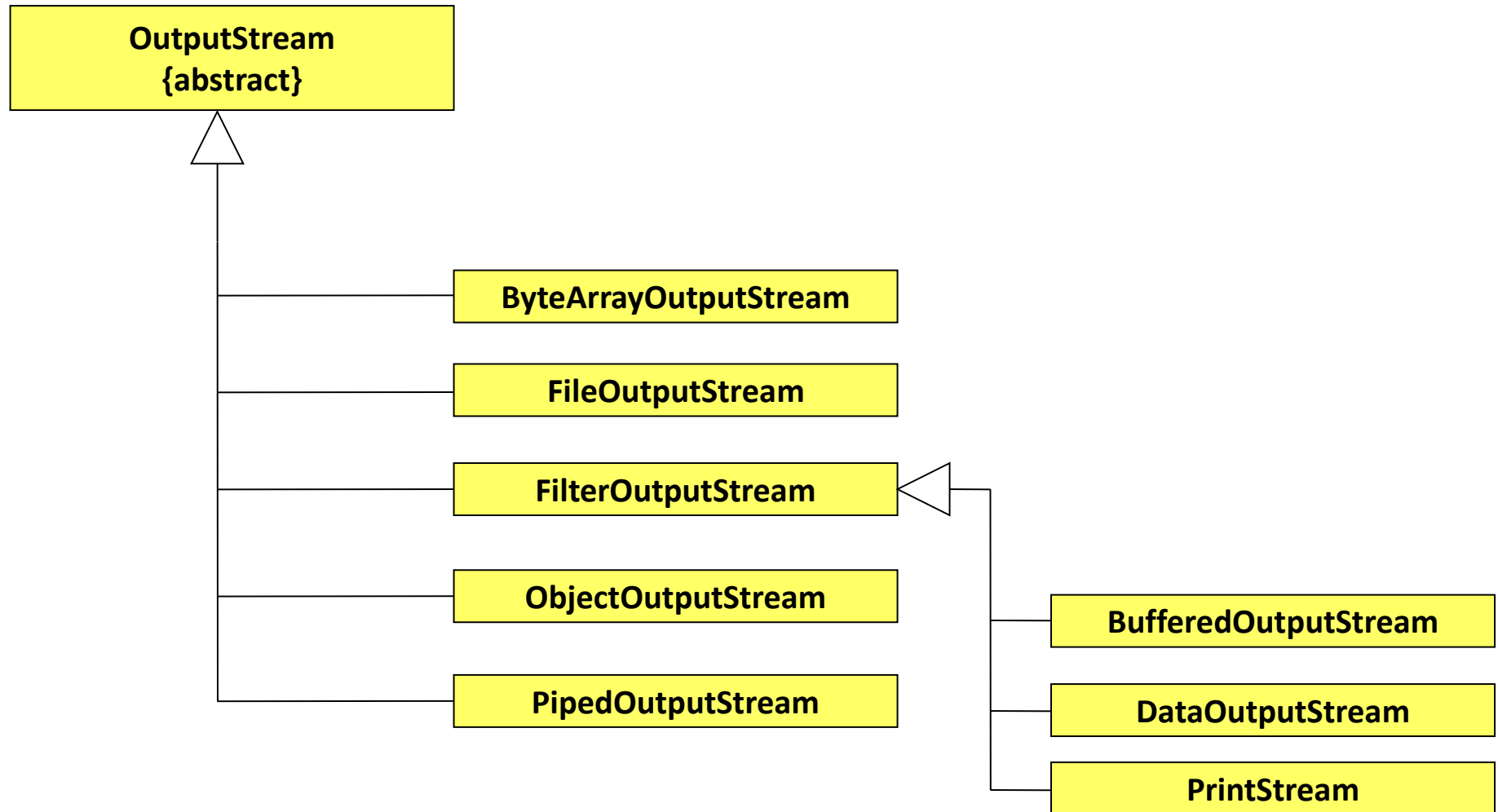
- Man hat das "klassische" Vorgehen, nur dass Streams Objekte sind:
  - Stream-Objekt öffnen,
    - bis Java 6: mit new
    - ab Java 7: alternativ über new oder `Files.newXXX()`
  - Beispiele für Files:

```
BufferedWriter writer = Files.newBufferedWriter(path);  
BufferedReader reader = Files.newBufferedReader(path);  
InputStream is = Files.newInputStream(path);
```

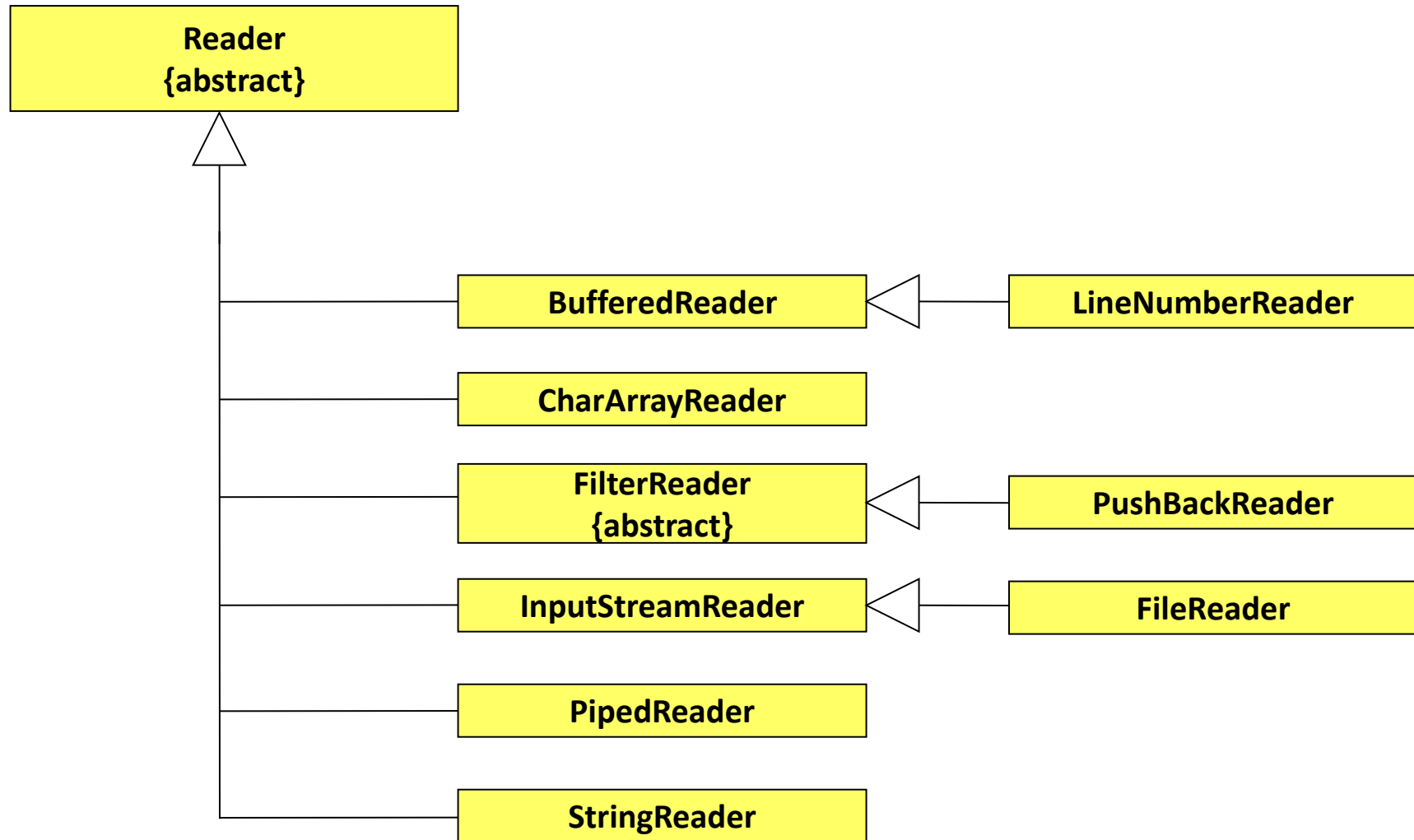
- Schreiben/lesen als Methoden dieser Objekte.
- Stream-Objekt schließen.

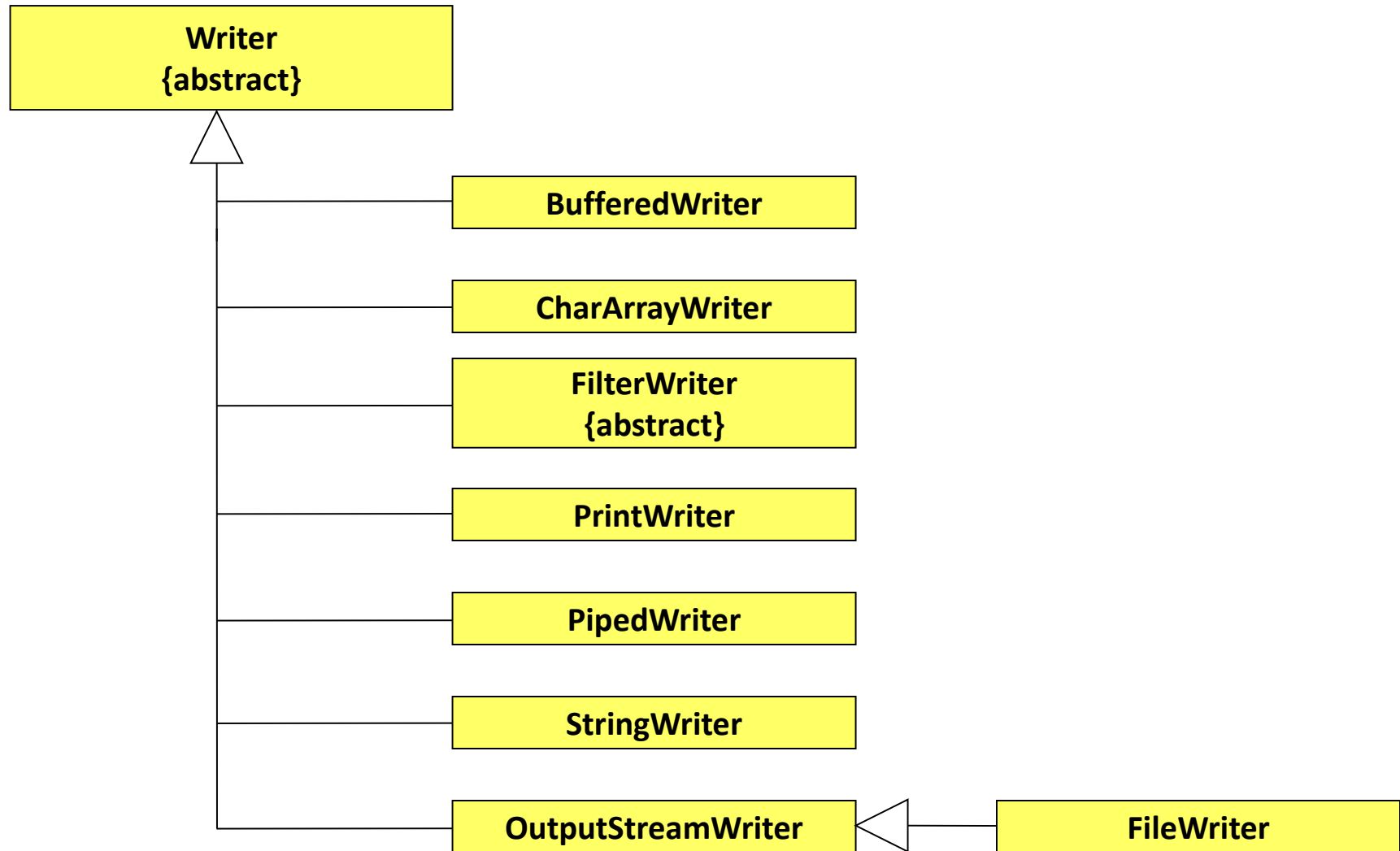
- Es gibt **Byte Streams** und **Character Streams**.
- Byte Streams
  - Input Streams bzw. Output Streams.
  - Einheiten sind 8-Bit Bytes.
  - Geeignet für Binär-Daten wie Images und Sounds.
- Character Streams:
  - Readers bzw. Writers.
  - Einheiten sind 16-Bit Characters.
  - Geeignet für textuelle Information, da alle Zeichen des Unicode Character Set verarbeitet werden können.











- Konstruktoren von FileWriter:

```
public FileWriter(String fileName) throws IOException  
public FileWriter(File file) throws IOException
```

- Konstruktoren von CharArrayWriter:

```
public CharArrayWriter()  
public CharArrayWriter(int initialSize)
```

- Konstruktoren von PrintWriter:

```
public PrintWriter(Writer out)  
public PrintWriter(OutputStream out)
```

- Für kleine Textdateien gibt es mit dem Java-7-API eine Vereinfachung, die in vielen Fällen genügen wird.
- Hierfür werden keine Ströme benötigt!

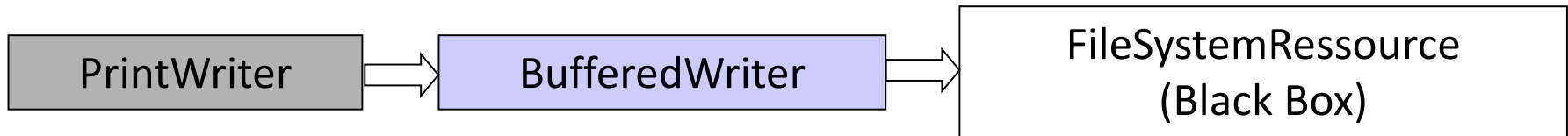
```
Path path = Paths.get("texte.txt");

List<String> texte = new ArrayList<>();
texte.add("Zeile 1");
texte.add("Zeile 2");
texte.add("Zeile 3");

Files.write(path, texte);

List<String> content = Files.readAllLines(path);
for (String string : content)
{
    System.out.println(string);
}
```

- Kann/soll die Datei nicht auf einmal in den Speicher geladen werden oder handelt es sich um keine Textdatei oder soll der Dateiinhalt einer anderen Klasse unaufbereitet übergeben werden, müssen Ströme verwendet werden.
- Für textbasierte Ausgaben eignet sich `PrintWriter` (verwandt mit `PrintStream`, der durch `System.out` bekannt sein sollte...)
- Für textbasierte (zeilenweise) Eingaben ist v.a. `BufferedReader` interessant, für Sonderaufgaben mit Parsing noch die Klasse `Scanner`.
- Die grundsätzliche Verfahrensweise mit Strömen in Java ist, dass man sie sich passend "ineinandersteckt"!

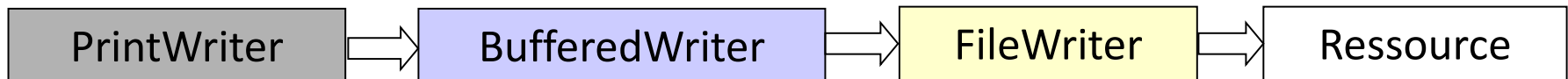


```
Path path = Paths.get("texte.txt");
PrintWriter writer = new PrintWriter(Files.newBufferedWriter(path));

writer.printf("%s mit %s\n", "IO", "Java");

writer.close();
```

Java 7+



```
FileWriter fileWriter = new FileWriter("texte.txt");
BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
PrintWriter writer = new PrintWriter(bufferedWriter);

writer.printf("%s mit %s\n", "old IO", "Java");

writer.close();
```

&lt;= Java 6

```
print(boolean b)           println(boolean b)
print(char c)              println(char c)
print(char[] ac)           println(char[] ac)
print(int i)               println(int i)
print(long l)              println(long l)
print(double d)            println(double d)
print(float f)             println(float f)
print(String str)          println(String str)
print(Object obj)          println(Object obj)
                           println()

printf(String format, Object ... params)
```

- Lesen mit BufferedReader.
- BufferedReader stellt die Methode `readLine()` zur Verfügung.



```
// Alternativ "old io":  
// FileReader fileReader = new FileReader("texte.txt");  
// BufferedReader reader = new BufferedReader(fileReader);  
BufferedReader reader = Files.newBufferedReader(path);  
  
String line = null;  
  
while ((line = reader.readLine()) != null)  
{  
    System.out.println(line);  
}  
  
reader.close();
```



- Lesen per Java 8 Stream-Konzept und Lambda-Expression:

```
BufferedReader reader = Files.newBufferedReader(path);  
reader.lines().forEach(a -> System.out.println(a));  
reader.close();
```

- Etwas kompakter als die ältere Variante.
- Dahingehend eleganter, dass die unschöne Vereinbarung im Kopf der while-Schleife (Zuweisung + Abfrage auf einmal) entfällt.
- Wird „spannend“, wenn der Inhalt der Schleife anspruchsvoller wird...

- Die Tastatureingabe wird über das InputStream-Objekt `System.in` zur Verfügung gestellt.
- Ein `InputStream` ist ein Byte-Stream.
- Man müsste gegebenenfalls ein eigenes „Character-Encoding“ beim Übergang zu Characters durchführen.

```
byte[] bytes = new byte[80];  
System.in.read(bytes);  
String s = new String(bytes);  
String ls = System.getProperty("line.separator");  
s = s.substring(0, s.indexOf(ls));
```

alles bis zum  
nächsten Line-  
Separator

- Das Encoding kann man mit einem Objekt der Klasse `InputStreamReader` bekommen.
- `InputStreamReader` ist ein Character Stream.
- Wenn man außerdem noch mit einem `BufferedReader` arbeitet, ist das zeilenweise Lesen verbunden mit einer Pufferung.

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



ganze Zeile

- Die Bildschirmausgabe wird über das PrintStream-Objekt System.out zur Verfügung gestellt.
- Die Klasse PrintStream ist ein Byte Stream.  
Das hat historische Gründe (ist seit Java 1.0 so, da gab es noch keine Character-Streams)
- PrintStream sollte nicht zum Schreiben von Characters verwendet werden.
- Man kann stattdessen folgendermaßen auf einen PrintWriter übergehen:

```
PrintWriter out = new PrintWriter(System.out);  
out.println("Ü ist in verschüdenen Sprachen häufig anzutröffen.");  
out.flush(); // je nach Ausgabemedium sieht man sonst nichts
```

- Wenn Sie mit dem JDK in einer (Windows)-Shell arbeiten, werden bei der Ausgabe über **System.out** problematische Sonderzeichen je nach eingestelltem Encoding nicht richtig ausgedruckt.

```
C:\Java\DHBW>java Bildschirmausgabe
■ ist in versch³denen Sprachen hõufig anzutr÷ffen.

C:\Java\DHBW>
```

- Abhilfe schafft folgende Stream-Kette:

```
PrintWriter out = new PrintWriter(  
    new OutputStreamWriter(System.out, "Cp850"));  
out.println("Ü ist in verschüdenen Sprachen häufig anzutröffen.");  
out.flush();  
  
// Alternativ  
PrintWriter out = new PrintWriter(  
    new OutputStreamWriter(System.out, Charset.forName("IBM850")));  
out.println("Ü ist in verschüdenen Sprachen häufig anzutröffen.");  
out.flush();
```

```
C:\Java\DHBW>java Bildschirmausgabe  
Ü ist in verschüdenen Sprachen häufig anzutröffen.
```

```
C:\Java\DHBW>
```

- IO-Operationen werfen oft Exceptions.
- Dies erschwert die Programmierung, denn es muss darauf geachtet werden, dass das Schließen des Stromes auf jeden Fall im Code erreicht wird.

```
Path path = Paths.get("texte.txt");
```

```
BufferedWriter writer = Files.newBufferedWriter(path);
```

Exception

```
writer.newLine();
```

Exception

```
writer.close();
```

Exception

```
Path path = Paths.get("texte.txt");
BufferedWriter writer = null;

try
{
    writer = Files.newBufferedWriter(path);
    writer.newLine();
}
catch (IOException e)
{
    // do something
}
finally
{
    if (writer != null)
    {
        try
        {
            writer.close();
        }
        catch (IOException e)
        {
            // do something if close doesn't work
        }
    }
}
```

- Beispiel mit "korrektem" Fehlercode.
- Es wird beachtet, dass der Strom auch wieder geschlossen wird.
- Allerdings kann er auch noch nie korrekt geöffnet worden sein...
- Umständlicher als die erste Version...



- Abhilfe schafft das mit Java 7 eingeführte try-with-resources Statement.
- Dieses Statement verarbeitet jeden Ressource-Typ, der das Interface `AutoCloseable` implementiert.
- Syntax:

```
try ( resources; ) {}  
[ catch (...){} ]  
[ finally {} ]
```

- Arbeitsweise:  
es wird ein generierter `finally`-Block gebildet, in dem für alle `AutoCloseables` im `resources`-Abschnitt
  - die `close()`-Methode aufgerufen wird
  - und das null-sicher

```
try ({VariableModifier} R Identifier = Expression ...)  
Block
```

```
{  
    final {VariableModifierNoFinal} R Identifier = Expression;  
    Throwable #primaryExc = null;  
    try ResourceSpecification_tail  
        Block  
    catch (Throwable #t) {  
        #primaryExc = #t;  
        throw #t;  
    } finally {  
        if (Identifier != null) {  
            if (#primaryExc != null) {  
                try {  
                    Identifier.close();  
                } catch (Throwable #suppressedExc) {  
                    #primaryExc.addSuppressed(#suppressedExc);  
                }  
            } else {  
                Identifier.close();  
            }  
        }  
    }  
}
```

- Konkret in unserem Beispiel:

```
Path path = Paths.get("texte.txt");
```

```
try (BufferedWriter writer = Files.newBufferedWriter(path))  
{  
    writer.newLine();  
}  
catch (IOException e)  
{  
    // do something  
}
```

!

kein finally mit close() erforderlich

Zusammenfassend gilt folgendes für die Fehlerbehandlung bei IO:

- Seien Sie gründlich!
- Versuchen Sie, nicht „trivialen“ Code zu schreiben (wie in vielen Beispielen in diesem Vortrag), sondern in Projekten IO-Code in Framework-artige Strukturen zu bringen, die Fehlerbehandlung absichern und systematisieren.
- Nutzen Sie die Angebote in Java mit try-with-resources, aber rechnen Sie in catch-Blöcken mit null-Werten.
- Gilt eigentlich immer, aber hier besonders: keine leeren Catch-Blöcke. Lassen Sie Ihren Code nicht Exceptions „verschlucken“. Die Fehlersuche wird sonst zum Albtraum.
- Gruppieren Sie Fehler, die Sie gleichartig behandeln können und machen Sie sich die verschiedenen Arten von Fehlern, die Sie bekommen können, bewusst.

Stichwort: muss/kann/darf/soll ich eine NullPointerException gleichbehandeln wie eine FileNotFoundException?

- Da gerade IO-Code IO-Operationen und Fehlercode vermischt, versuchen Sie (gilt überall, hier aber besonders) keine weiteren Dinge, wie z.B. fachliche Datenverarbeitung, in der selben Methode unterzubringen.

