Java - Dateien lesen und schreiben

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhbw.de, Raum 149B

Version: 1.0

Folien: https://delors.github.io/prog-adv-java-ea/folien.de.rst.html
https://delors.github.io/prog-adv-java-ea/folien.de.rst.html.pdf

Kontrollfragen: https://delors.github.io/prog-adv-java-ea/kontrollfragen.de.rst.html

Fehler melden: https://github.com/Delors/delors.github.io/issues

1. Einführung in E/A (🇺🇸 *I/O*) in Java

Architektur von Java I/O

- Verschiedene Quellen/Senken für die Daten
  - Tastatureingabe („Standard I/O“)
  - Datei(en) auf dem lokalen Rechner („File I/O“)
  - Datei(en)/Prozess(e) im Netzwerk („Network I/O“)
  - Hauptspeicher („Memory I/O“)
- Um einheitlich Daten auf unterschiedliche „Datenbehälter“ ein-/auszugeben, verwendet Java das Konzept der „Streams“ (Ströme): Eingabe- und Ausgabeströme (E/A,  *I/O*)
- Datenströme als eine Abstraktion von I/O „Geräten“
  - Verstecken Details über die Implementierung / Funktionsweise der einzelnen I/O-Geräte vor dem Java Programmierer

    (D. h. ob die Eingabe zum Beispiel eine Tastatur, Datei, anderes Programm, Netz, Hauptspeicher, … ist.)
  - Stellen Java-Programmen einheitliche Schnittstellen zum Lesen bzw. Schreiben von Daten zur Verfügung.

I/O-Ströme in Java

Lesen von Daten

Um Daten von einer externen Datenquelle zu lesen, öffnet ein Java-Programm einen Eingabestrom und liest die Daten *seriell* (nacheinander) ein:

Schreiben von Daten

Um Daten in eine externe Senke zu schreiben, öffnet ein Java-Programm einen Ausgabestrom und schreibt die Daten *seriell*.

Klassifizierung von Datenströmen

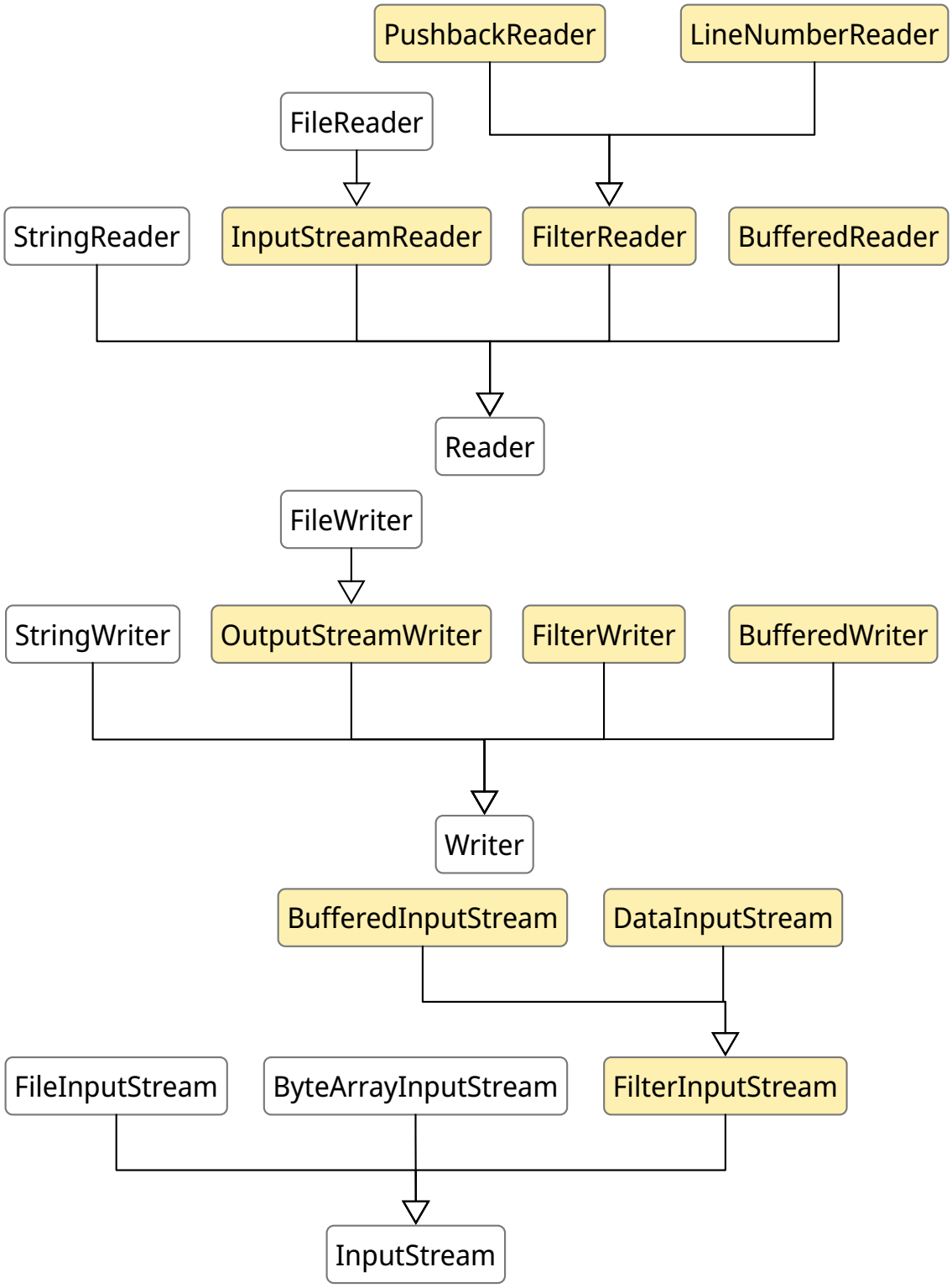
Nach Datentyp:

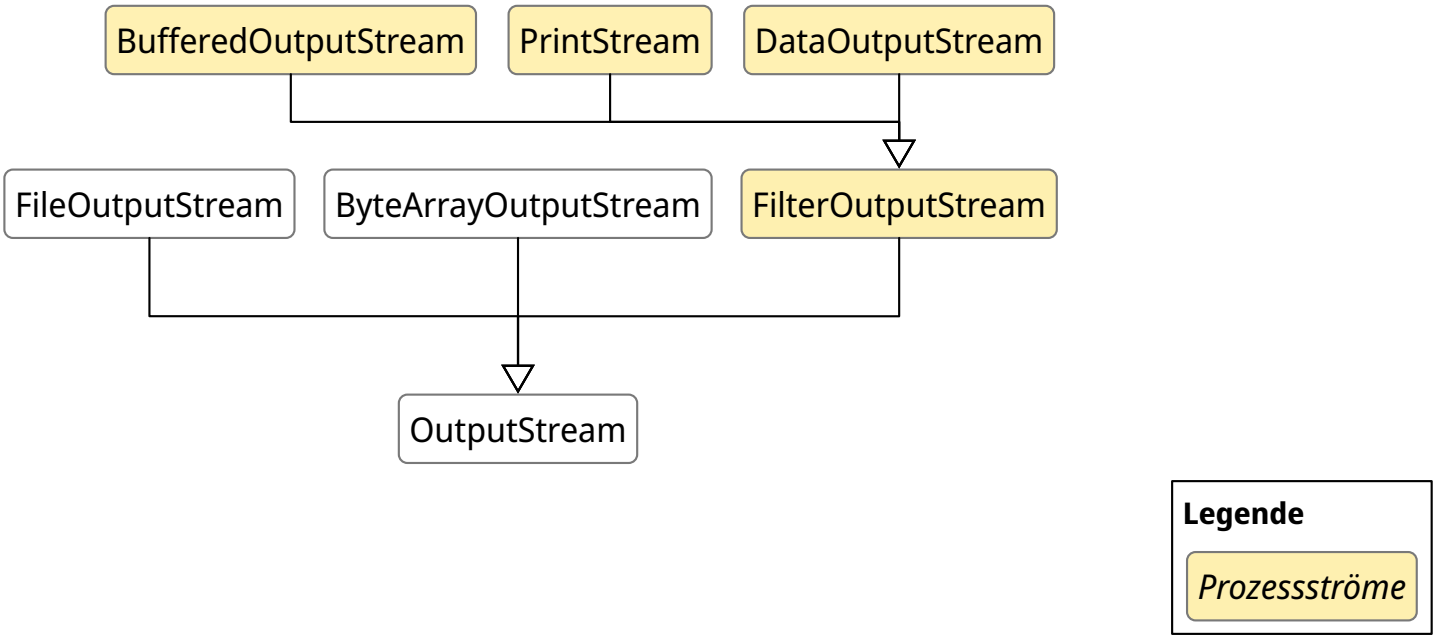
- **Zeichenströme:**
  - lesen / schreiben `char` (16-bit Unicode Zeichensatz).
    `java.io.Reader`/`java.io.Writer` stellen die Schnittstelle und eine partielle Implementierung von Zeichenströmen zur Verfügung.
    - Subklassen von `Reader`/`Writer` fügen neues Verhalten hinzu bzw. ändern dieses.
- **Byteströme:**
  - lesen / schreiben `byte`s (8-bit).
    Werden zum Lesen bzw. Schreiben von Binärdaten, z.B. Bildern, benutzt.
    - `java.io.InputStream`/`java.io.OutputStream`: gemeinsame Schnittstelle und partielle Implementierung für alle Ströme zum Lesen bzw. Schreiben von Bytes.
    - Alle anderen Byteströme sind Unterklassen davon.

Nach Struktur der Ströme:

- **Datensenkeströme:**
  - Daten werden direkt von „physikalischer“ Datenquelle gelesen bzw. auf „physikalische“ Datensenke geschrieben
- **Prozessströme:**
  - Daten werden von anderen Strömen gelesen bzw. auf andere Ströme geschrieben
  - Daten werden nach dem Lesen bzw. vor dem Schreiben gefiltert, gepuffert, bearbeitet, usw.

Hierarchie der Zeichenströme für I/O




BufferedOutputStream
PrintStream
DataOutputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
OutputStream
Legende
Prozessströme

Allgemeines Vorgehen beim Lesen/Schreiben von Dateien

Lesen:

1. Öffne einen Strom
   Ströme werden beim Erzeugen automatisch geöffnet
2. Lese Daten, solange nötig und es noch Daten gibt
3. Schließe den Strom
   Beim Beenden des Lesens ist der Strom durch `close()` zu schließen.

Schreiben:

1. Öffne einen Strom
2. Solange es noch Daten gibt, schreibe Daten
3. Schließe den Strom
   Beim Beenden des Schreibens ist der Strom durch `close()` zu schließen.

Zur Erinnerung

Lesen/Schreiben von/auf Strömen haben unabhängig von Datentyp und Quelle bzw. Senke die gleiche Form.

Schreiben und Lesen von Daten

```
public interface InputStream {
    public int read()
    public int read(byte[] bbuf)
    public int read(byte[] bbuf, int offset, int len)
}
```

```
public interface Reader {
    public int read()
    public int read(char[] cbuf)
    public int read(char[] cbuf, int offset, int len)
}
```

Achtung!

Es gilt immer zu überprüfen ob auch die erwartete Anzahl an Bytes/Zeichen gelesen wurde.

```
public interface OutputStream {
    public int write(int b)
    public int write(byte[] bbuf)
    public int write(byte[] bbuf, int offset, int len)
}
```

```
public interface Writer {
    public int write(int c)
    public int write(char[] cbuf)
    public int write(char[] cbuf, int offset, int len)
}
```

Achtung!

Es gilt immer zu überprüfen ob auch die erwartete Anzahl an Bytes/Zeichen geschrieben wurde.

Dateiströme in Java

- Dateiströme sind Ein-/Ausgabe-Ströme, deren Quellen/Senken Dateien im Dateisystem sind:
- `FileReader` / `FileWriter` für Lesen / Schreiben von Zeichen aus/in Dateien
- `FileInputStream` / `FileOutputStream` für Lesen / Schreiben von Bytes von/in Dateien
- Ein Dateistrom kann erzeugt werden, indem man die Quelle- bzw. Senke-Datei durch eines der folgenden Objekte als Parameter des Strom-Konstruktors übergibt:
  - Dateiname (`String`)
  - Datei-Objekt (`java.io.File`)
  - Dateibeschreibung (`java.io.FileDescriptor`)
- Die Klasse `java.nio.file.Files` bietet weitere Methoden (z. B. `newInputStream(...)`, `newBufferedWriter(...)`) zum Lesen und Schreiben von Dateien als Streams an.

Einfaches (ineffizientes) Beispiel

```
1 void print(String fileName) throws IOException {
2     try (FileReader in = new FileReader(fileName)) {
3         int b;
4         while ((b = in.read()) != -1) System.out.print(b);
5     }
6 }
```

Prozessströme

- Ein Prozess-Strom enthält einen anderen (Daten- oder Prozess-)Strom

  Dieser dient als Quelle bzw. Senke.

- Prozess-Ströme ändern Daten oder bieten Funktionalität:
  - Zwischenspeichern (Puffern) von Daten
  - Zählen der gelesenen/geschriebenen Zeilen
  - Konvertierung zwischen Byte und Zeichen
  - Kompression, ...

Pufferströme

- Ein Pufferstrom (z. B. `BufferedInputStream` oder `BufferedOutputStream`) kapselt einen anderen Datenstrom und einen internen Puffer
- Beim ersten Lesen wird der Puffer vollständig gefüllt
  - Weitere Lese-Operationen liefern Bytes vom Puffer zurück, ohne vom unterliegenden Strom tatsächlich zu lesen.
  - Bei leerem Puffer wird erneut vom unterliegenden Strom gelesen
- Beim Schreiben werden die Daten zuerst in dem internen Puffer gespeichert, bevor sie in den unterliegenden Strom geschrieben werden.
  - Nur wenn der Puffer voll ist, wird auf den unterliegenden Strom geschrieben
  - Sowie bei explizitem Aufruf der Methode `flush()` oder `close()`.

!! Wichtig

Die richtige Puffergröße kann die Geschwindigkeit beim Lesen und Schreiben von Dateien **erheblich** beeinflussen (5-10x). Die richtige Puffergröße ist von vielen Faktoren abhängig liegt aber vermutlich zwischen 8KB und 64KB.

Einfaches Beispiel

```java
try(
    FileOutputStream fos = new FileOutputStream("Test.tmp");
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    DataOutputStream out = new DataOutputStream(bos)
) {
    out.writeInt(9);
    out.writeDouble(Math.PI);
    out.writeBoolean(true);
}
```

```
$ hexdump Test.tmp
0000000 0000 0900 0940 fb21 4454 182d 0001
```

Architektur der I/O API

Ströme können ineinander verschachtelt werden.

Abstraktionsebenen, bei denen unterliegende „primitive“ Ströme von umschließenden („höheren“, komfortableren) Strömen benutzt werden („Prozessströme“).

```
// erzeugt gepufferten, komprimierenden Dateiausgabestrom
OutputStream out = new FileOutputStream(<filename>);
var bout = new BufferedOutputStream(out);
var zout = new ZipOutputStream(bout);
// … mehr Eigenschaften koennen dynamisch hinzugefuegt werden
// Die Stromeigenschaften sind unsichtbar fuer Klienten
```

Die Technik, mit der erreicht wird, dass Ströme beliebig zur Laufzeit kombiniert werden können, ist nicht nur im Kontext von Strömen von Interesse. Es handelt sich um eine generelle Technik, um Objekte dynamisch mit Features zu erweiterten.

In der Softwaretechnik werden solche Techniken in der Form so genannter *Design Patterns* (*Entwurfsmuster*) dokumentiert; d. h. wiederverwendbare, dokumentierte Designideen.

Die oben genannte Technik bei Streams ist als „Decorator Pattern“ bekannt.

Übung

1.1. Datei lesen und ausgeben

Schreiben Sie ein Programm, dass eine Textdatei liest und die Zeilen in der Konsole ausgibt. Schreiben Sie vor jede Zeile die Zeilennummer.

Beispiel

Für folgende Datei (Autor: ChatGPT):

```
In Java springt der Code so leicht,
Klammern tanzen, Ziel erreicht,
Fehler? Nur ein Abenteuer vielleicht!
```

sollte folgende Ausgabe erzeugt werden:

```
1: In Java springt der Code so leicht,
2: Klammern tanzen, Ziel erreicht,
3: Fehler? Nur ein Abenteuer vielleicht!
```

2. Java Streams und I/O

java.nio.file.Files

Neben den traditionellen I/O-Klassen (seit Java 1.x) gibt es auch die Möglichkeit Dateien als Streams zu lesen und zu schreiben (`java.nio.file.Files`).

```
1  package java.nio.file;
2
3  public class Files {
4      /** Read all lines from a file as a Stream. */
5      static Stream<String> lines(Path path)
6
7      /** Read all lines from a file as a Stream. */
8      static Stream<String> lines(Path path, Charset cs)
9
10     // ...
11 }
```

Achtung!

Diese Streams müssen explizit geschlossen werden (`close()`), da sie Ressourcen verbrauchen.

Übung

2.1. Streamverarbeitung von Dateien

Schreiben Sie ein Programm, das eine Textdatei liest und die Zeilen in der Konsole ausgibt. Jeder Zeile soll weiterhin die Zeilennummer vorangestellt werden. Verwenden Sie dazu die Klasse *Files* und die Methode *lines*.

Beispiel

Für folgende Datei (Autor: ChatGPT):

```
In Java springt der Code so leicht,
Klammern tanzen, Ziel erreicht,
Fehler? Nur ein Abenteuer vielleicht!
```

sollte folgende Ausgabe erzeugt werden:

```
1: In Java springt der Code so leicht,
2: Klammern tanzen, Ziel erreicht,
3: Fehler? Nur ein Abenteuer vielleicht!
```

Übung

2.2. Durchsuchen von Dateien

Schreiben Sie ein Programm (Sie können die JShell benutzen), dass alle Textdateien (z. B. *.rxt, *.md oder *.java) eines Verzeichnisses in Hinblick auf das Vorkommen eines bestimmten Wortes (z. B. Java) durchsucht. Geben Sie den Namen der Datei und eine Zeilennummer aus, in der das Wort vorkommt. Parallelisieren (`parallel()`) Sie die Suche wenn möglich.

Relevante API: `Files.walk`, `File.toPath()`, `Files.isRegularFile`, `Files.lines`, `Stream.filter`, `Stream.map`, `Stream.findAny`, `Optional.isPresent`, `Optional.get`, `Optional.empty`

Hinweis

Sie müssen ggf. `IOException`s explizit behandeln und in solchen Fällen zum Beispiel `Optional.empty()` zurückgeben.