Programmierung 2

DHBW Mannheim - WWI17SCB - May 2018

Slides

https://goo.gl/U45wY3

7. Input/Output & Files

Input/Output - Lernziele

* Streams und ihre Verwendung kennenlernen
* Grundlegende Stream-Methoden unterscheiden
* Buffered Streams und ihre Vorteile kennen
* Dateilese- und Schreiboperationen mit Streams ausführen
* Properties-Dateien schreiben und lesen können

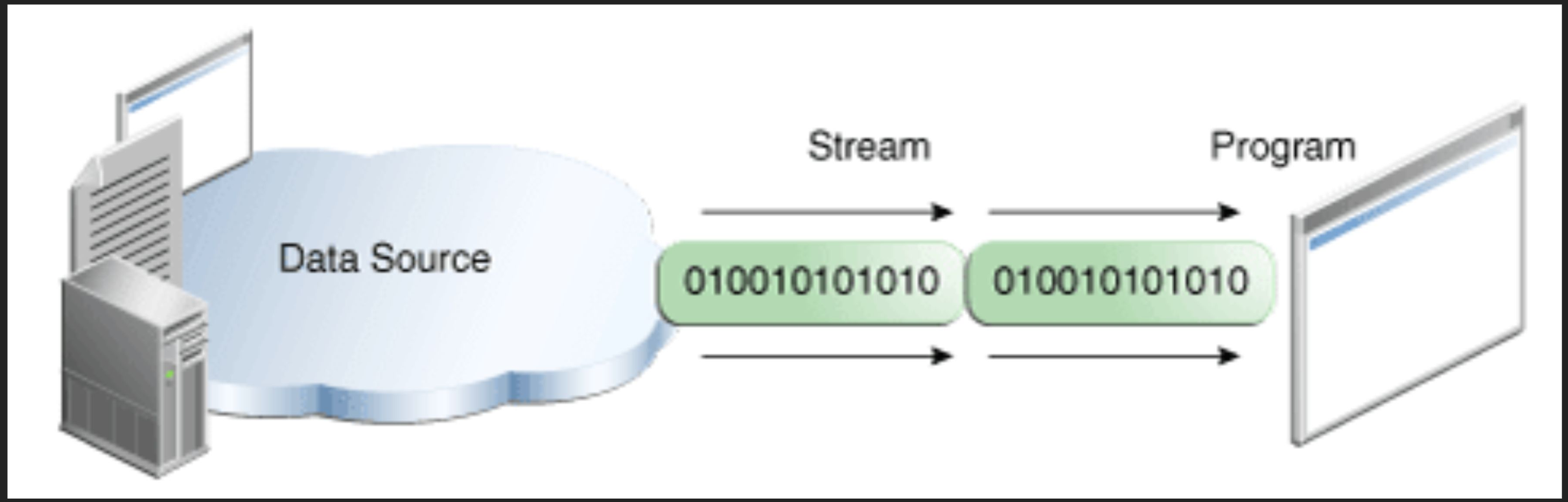
Was sind Streams?

Was sind Streams?

* Streams sind **Sequenzen von Daten**.
* Ein Stream repräsentiert eine Quelle von Eingabedaten oder ein Ziel von Ausgabedaten.
* Streams können verschiedenste Quellen bzw. Ziele haben, z.B.:
  * Dateien auf der Festplatte
  * Netzwerkverbindungen
  * Externe Geräte (z.B. USB- oder Bluetooth-Verbindungen)
  * Den Hauptspeicher (RAM)

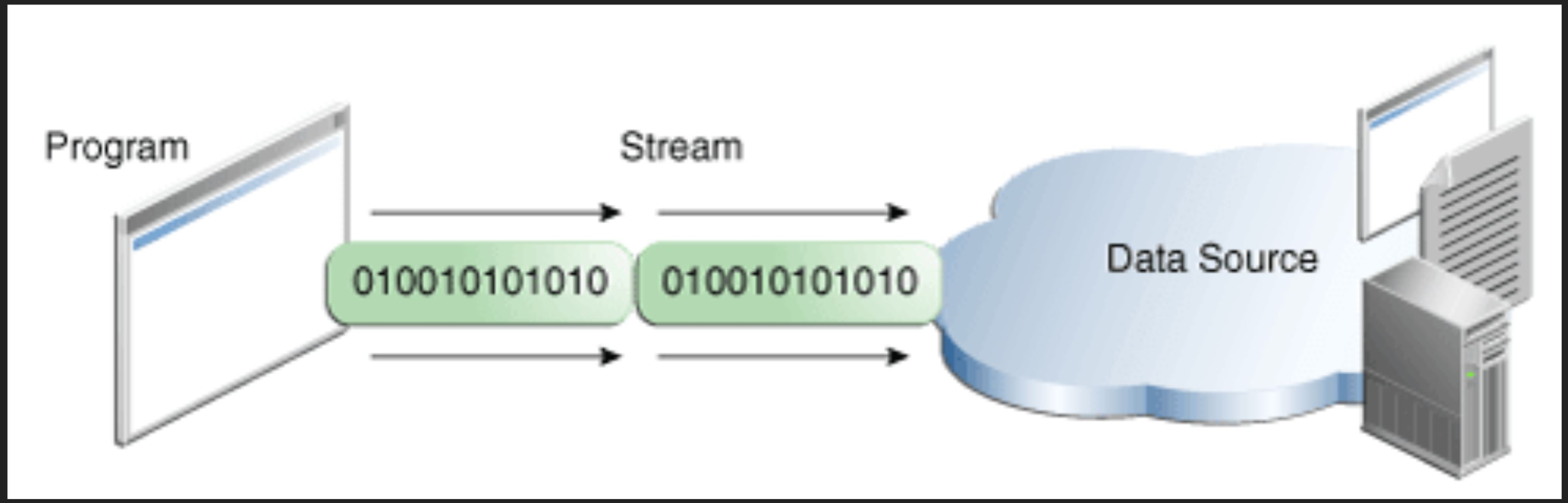
Was sind Streams?

* Ein Input Stream wird verwendet um Daten aus einer Quelle ins Programm einzulesen.



Was sind Streams?

* Ein Output Stream wird verwendet um Daten aus dem Programm in ein Ziel zu schreiben.



Grundlegende Stream-Methoden / Byte-Streams

Input Streams

`java.io.InputStream`

* `public abstract int read() throws IOException`

  Liest das nächste Byte aus dem Stream oder -1, wenn es keine Daten mehr gibt.

* `public void close() throws IOException`

  Schließt einen Stream und gibt die damit angeforderten Systemressourcen frei. Sehr wichtig, um keine Memory Leaks zu erzeugen!

Output Streams

`java.io.OutputStream`

* `public void write(byte[] b) throws IOException`

  Schreibt die Bytes in b[] in den Output-Stream.

* `public void flush() throws IOException`

  Schreibt alle im Stream verbleibenden Daten in das Ziel des Output-Streams.

* `public void close() throws IOException`

  Schließt einen Stream und gibt die damit angeforderten Systemressourcen frei.

Beispiel: Datei kopieren mit Streams

`java.io.FileInputStream & java.io.FileOutputStream`

* Erben direkt von InputStream bzw. OutputStream

* Low-Level API, um auf beliebige Streams zuzugreifen

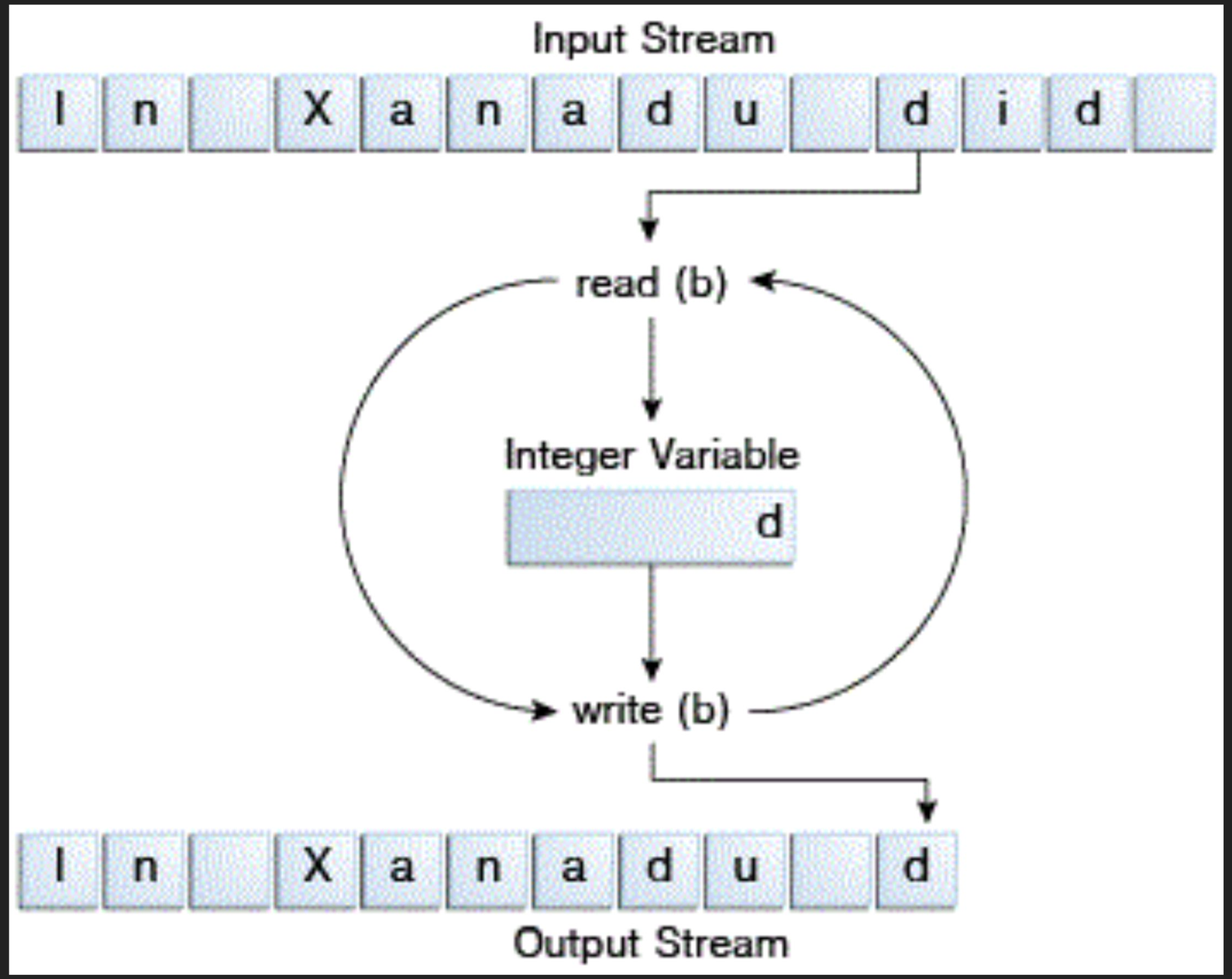
Beispiel: Datei kopieren mit Streams

```
1  FileInputStream in = null;
2  FileOutputStream out = null;
3
4  try {
5    URL samuel = StreamDemo1.class.getResource("Samuel_L_Ipsum.txt");
6    in = new FileInputStream(samuel.getFile());
7    out = new FileOutputStream("/Users/bastian/Desktop/copy.txt");
8    int c;
9
10   while ((c = in.read()) != -1) {
11     out.write(c);
12   }
13 } finally {
14   if (in != null) { in.close(); }
15   if (out != null) { out.close(); }
16 }
```

Try-With-Resources

```
URL samuel = StreamDemo2.class.getResource("Samuel_L_Ipsum.txt");
try (
  FileInputStream in = new FileInputStream(samuel.getFile());
  FileOutputStream out = new FileOutputStream("/Users/bastian/Desktop/copy.txt");
) {
  int c;
  while ((c = in.read()) != -1) {
    out.write(c);
  }
}
```

Beispiel: Datei kopieren mit Streams



Byte-Streams

* Byte-Streams sind sehr einfach gebaut und operieren auf der untersten logischen Datenebene - einzelnen Bytes
* Es gibt bessere Streams, die für bestimmte Datentypen optimiert sind, z.B. Character Streams für Dateien mit Text
* Byte Streams sollten nur verwendet werden, wenn kein "höherer" Stream zur Verfügung steht
* Alle höheren Streams sind auf Byte-Streams aufgebaut

Höhere Stream-Abstraktionen

Beispiel: Datei kopieren mit Character Streams

`java.io.FileReader & java.io.FileWriter`

* Erben indirekt von InputStream bzw. OutputStream

* Low-Level API, um auf Dateien zuzugreifen

Beispiel: Datei kopieren mit Character Streams

```
1  URL samuel = StreamDemo2.class.getResource("Samuel_L_Ipsum.txt");
2  try (
3    FileReader in = new FileReader(samuel.getFile());
4    FileWriter out = new FileWriter("/Users/bastian/Desktop/copy.txt");
5  ) {
6    int c;
7    // c enthält hier 2-4 bytes character anstelle von jeweils 1 byte
8    while ((c = in.read()) != -1) {
9      out.write(c);
10   }
11 }
```

Zeilenweises Lesen von Dateien

`java.io.BufferedReader & java.io.PrintWriter`

* BufferedReader "puffert" die Eingabe eines Streams - d.h. nicht jede Lese-Operation führt zu einem Dateizugriff. Stattdessen werden immer n Bytes gelesen (gebuffert) - das führt zu weniger Systemzugriffen
* PrintWriter kann Ausgabestrings formatieren und fängt Ausgabefehler ab - wirft also keine Exceptions
* Beide können jeweils mit einem zugrunde liegenden Stream initialisiert werden

Zeilenweises Lesen von Dateien

```
1  URL samuel = StreamDemo2.class.getResource("Samuel_L_Ipsum.txt");
2  try (
3    BufferedReader in = new BufferedReader(new FileReader(samuel.getFile()));
4    PrintWriter out = new PrintWriter(new FileWriter("/Users/bastian/Desktop/copy.t
5  ) {
6    // Jetzt lesen wir Strings, also höhere Abstraktionsebene!
7    String l;
8    while ((l = in.readLine()) != null) {
9      out.println(l);
10   }
11 }
```

Gepufferte Streams

Gepufferte Streams

* Alle Low-Level-Streams nutzen "unbuffered I/O" - das heißt: Jeder Lese- bzw. Schreibbefehl (read() / write()) wird direkt an das OS weitergereicht und erzeugt dort eine blockierende Systemoperation
* Das sorgt für sehr ineffizienten Datenzugriff --> langsam!
* Java kennt deshalb gepufferte Streams. Sie lesen immer eine gewisse Zahl an Bytes aus der Quelle (Input) bzw. sammeln Ausgabe, bis eine bestimmte Zahl an Bytes erreicht ist (Output). Das erfolgt in einem Speicherbereich namens **Buffer / Puffer**
* Ist der Puffer voll (schreiben) bzw. leer (lesen), erfolgt Systemoperation

Gepufferte Streams flushen

* Manchmal macht es Sinn, gepufferte Output-Streams manuell zu "flushen" - also explizit an bestimmten Stellen zu schreiben.
* Manche Steams unterstützen "Autoflushing" - dann wird z.B. bei jedem "println" ein flush ausgelöst.

BufferedInputStream	Byte	BufferedOutputStream
BufferedReader	Character	BufferedWriter

Coding Time!

I/O - Codingaufgabe 1

15m

* Erstellt eine Textdatei mit beliebigem Inhalt und macht sie für euer Programm zugreifbar (YourClass.class.getResource())
* Lest die Textdatei mithilfe von BufferedReader ein und gebt ihren Inhalt auf der Konsole aus
* Das Lesen sollte zeilenweise passieren
* Verwendet try-with-resources

I/O - Codingaufgabe 2

15m

* Lest beliebig viele Zeilen Eingabe von der Konsole und schreibt sie mithilfe von BufferedWriter/FileWriter zeilenweise in eine Datei.
* Die Eingabe von "0" soll den Schreibbefehl beenden und den Inhalt der Datei ausgeben.
* Verwendet Scanner zum lesen der Konsole.
* Flusht den Writer, bevor ihr das Programm beendet (flush())
* Verwendet zum Lesen der geschriebenen Datei wie gehabt den BufferedReader/FileReader

Weitere Streams

* Es gibt weitere Arten von Streams, z.B.:
  * Datenstreams (Interfaces DataInput / DataOutput) für alle primitiven Datentypen (boolean, char, byte, short, int, long, float und double)
  * Objektstreams - hiermit können Objekte, die das Interface **Serializable** implementieren, weggeschrieben (serialisiert) bzw. aus der serialisierten Form wieder als Objekt erzeugt (hydriert) werden --> ObjectInputStream / ObjectOutputStream

Properties

Properties

* Properties-Dateien sind besonders formatierte Dateien, die Java direkt in ein verständliches Objekt einlesen kann - mithilfe der Klasse Properties
* Properties ist eine Map und wird daher mit der Collection API bedient

```
1 # Properties-Dateien können Kommentare enthalten - praktisch!
2 # Sie sind nach dem Key-Value-Prinzip aufgebaut.
3
4 pi: 3.14
5 spielername: Helga
6 # = geht auch als separator
7 student=fleissig!
```

Beispiel: Properties-Datei lesen

```
1  Properties properties = new Properties();
2  URL propsURL = PropertiesExample.class.getResource("test.properties");
3  try (
4    InputStream in = propsURL.openStream();
5  ) {
6    properties.load(in);
7    // Ausgabe aller properties
8    Iterator<String> it = properties.stringPropertyNames().iterator();
9    while (it.hasNext()) {
10     String key = it.next();
11     System.out.println(key + ": " + properties.getProperty(key));
12   }
13 }
```

```
1 spielername: Helga
2 student: fleissig!
3 pi: 3.14
4
5 Process finished with exit code 0
```

Klausur

Feedback