JAVA IO

Übersicht

- Grundlagen
- Struktur der Klassen
- java.io.File
- Datenströme
- Lesen / Schreiben
- Strukturiertes Einlesen

Ein- und Ausgabe von Daten

Situation und Anforderungen

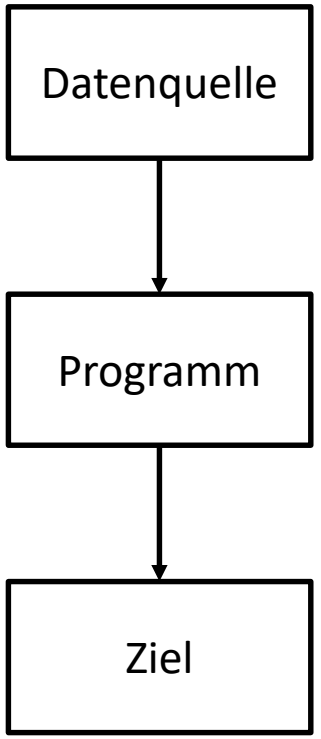
- Daten im flüchtigen Speicher während Laufzeit
- Sichern (Persistieren) erforderlich
- Einlesen ggf. zur Initialisierung des Programmzustands
- Kommunikation über Netzwerk

Input / Output

- Ein- und Ausgabe

Mögliche Quellen und Ziele von IO

- Datei auf Festplatte
- Dateien/Ressourcen auf entfernten Rechner
- Geräte (Bildschirm, USB, Serielle Schnittstelle, Drucker)
- andere Programme
- Speicherbereiche



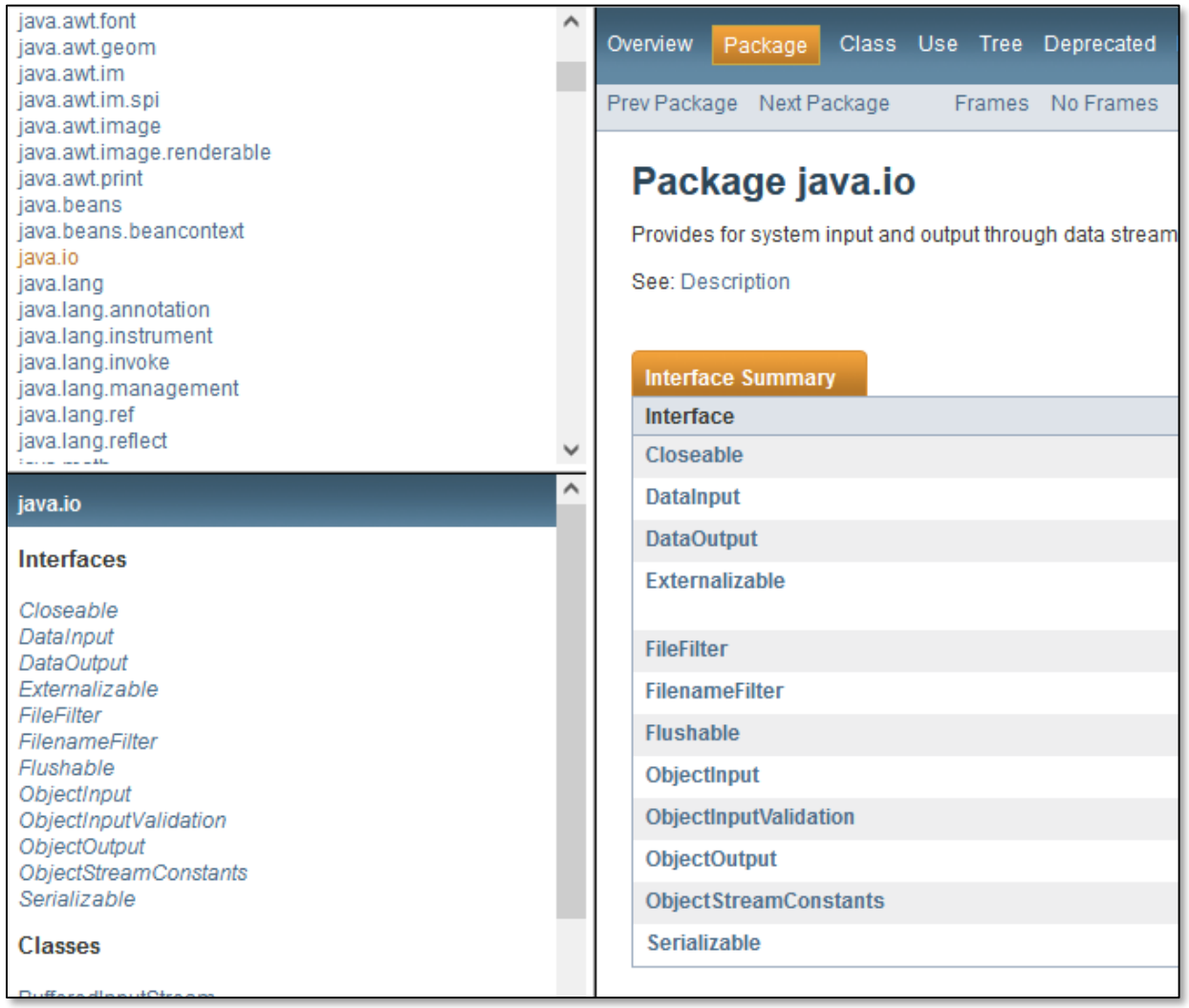
Verzeichnisse und Dateien

IO in Java

- Package **java.io** enthält Klassen für Ein- und Ausgabe
- Handhabung von Dateien und Verzeichnissen in **java.io.File** (Abstraktion bzgl. Plattform Win/Linux/etc.)

Pfad (Path) zu Datei

- spezielle Pfaddarstellung wird vorgesehen
- Linux/Unix: „home/username/docs“
- Windows: „c:\Users\username\Documents“
- Problem mit Laufwerksbuchstaben
  - Linux/Unix nur „/“ als Wurzel bekannt
  - Windows-Laufwerke separiert, statt Mounten



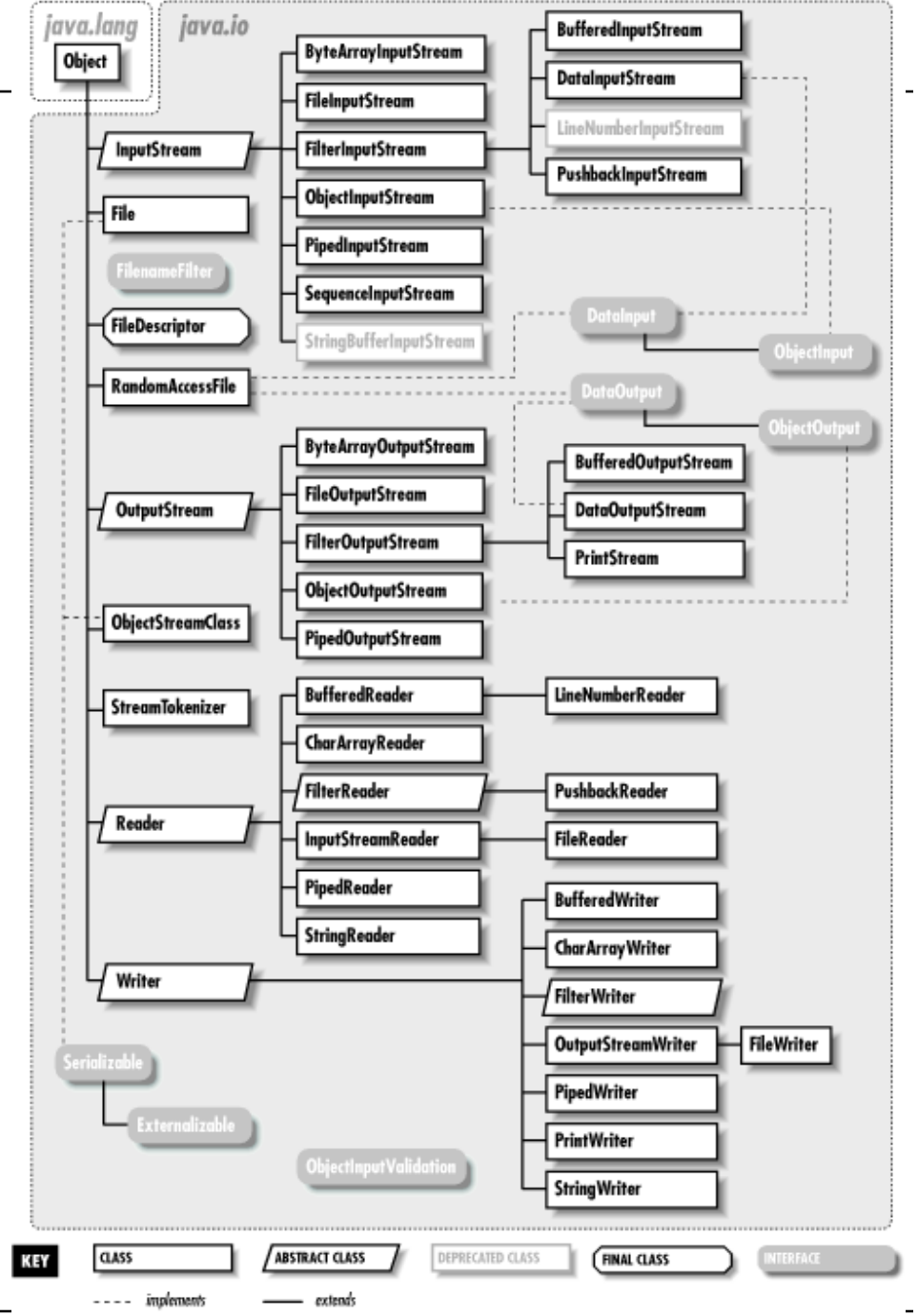
Übersicht Dateizugriff

Ausgangssituation

- Unterschiedliche Arten der Dateiarbeit erforderlich, z. B.
  - Daten in Datei schreiben / aus Datei lesen
  - Metainformationen lesen
  - Verzeichnis anlegen
  - Lesen und Schreiben unmittelbar hintereinander

Unterschiedliche Klassen verfügbar (Auszug)

- File
  - Datei erzeugen, umbenennen, löschen, Metainformationen
- FileInputStream / FileOutput Stream
  - Byteweise lesen und schreiben
- FileReader / FileWriter
  - Zeichenweise lesen und schreiben
- RandomAccessFile
  - In Datei Bereiche ändern
  - lesen/schreiben im Wechsel möglich



https://www.oreilly.com/library/view/learning-java/1565927184/ch10.html

JAVA.IO.FILE

java.io.File und Pfadangaben

Einsatzzweck File-Klasse

- Instanz von File repräsentiert Speicherort
- ermöglicht Dateiarbeit
- Speicherort muss (noch) nicht physikalisch existieren

Umsetzung von Pfadangaben

- Konkrete Pfadangaben konfigurierbar gestalten, statt fest programmieren

Beispiele (hier ausnahmsweise mit festen Pfadangaben)

```
// OS-unabhängig
File datei01 = new File("datendatei.txt"); // Datei im aktuellen Verzeichnis der Programmausführung
File datei02 = new File("../verz02");// relative Pfadangabe eines Verzeichnisses

// Windows
File datei03 = new File("c:\\daten\\temp"); // doppelter Backslash für Escaping
File datei04 = new File("c:/daten/temp");// unabhängig von Windows/Linux

// Linux
File datei05 = new File("/home/uname/dat.txt");// absolute Pfadangabe in Linux/Unix
File datei06 = new File("/mtn/usbmedia");// Pfad zu gemountetem USB-Medium
```

Pfadangaben plattformunabhängig

Herausforderung

- Plattformabhängigkeit schöpft Java-Potenzial nicht aus
- Abhängigkeit von OS erschwert Portierung
- Laufwerksbuchstaben und Pfadtrennzeichen erschweren Portierung

Möglichkeiten durch Java

- OS-spezifisches Pfadtrennzeichen abfragen

```
// Pfadtrennzeichen abfragen
char sepChar = File.separatorChar;              // Windows: '\\'; Unix: '/'
String sepString = File.separator;              // Windows: "\\"; Unix: "/"
String sepStringSys = System.getProperty("file.separator"); // über Systemeigenschaften
```

Hinweis: Die Zeichenkombination der zwei Backslashes („\\“) hat eine besondere Bedeutung. Der erste Backslash führt das Escaping aus, der zweite ist dann das eigentliche Zeichen. Deshalb ist ‘\\‘ auch nur ein Zeichen, nicht zwei.

Wichtige Methoden in Klasse File /1

Grundlegende Methoden für Dateiarbeit (Verzeichnisse und Dateien)

```
// Instanz von File muss nicht physisch vorhanden sein
File inputDatei = new File("c:\\daten\\inputfile.txt");

// Pruefen, ob die Datei (oder auch Pfad) vorhanden ist
boolean inputDateiExists = inputDatei.exists();

// Laenge der Datei ermitteln
long fileSize = inputDatei.length();

// Instanz von File muss nicht physisch vorhanden sein
File tiefesVerzeichnis = new File("c:/ein/tiefes/verzeichnis/mit/langem/pfad");

// falls Verzeichnis c:/ein/tiefes/verzeichnis/mit/langem/ vorhanden ist,
// wird nur letztes Verzeichnis erstellt
boolean mkDirOk = tiefesVerzeichnis.mkdir();

// falls mehrere uebergeordnete Verzeichnisse noch nicht vorhanden sind,
// werden auch uebergeordnete Verzeichnisse mit erstellt
boolean mkDirsOk = tiefesVerzeichnis.mkdirs();
```

Wichtige Methoden in Klasse File /2

Grundlegende Methoden für Dateiarbeit (Verzeichnisse und Dateien)

```
// Umbenennen oder Bewegen einer Datei
boolean renameOk = inputDatei.renameTo(new File("c:\\daten\\inputfile_neu.txt"));

// Datei oder Verzeichnis loeschen
boolean deleteOk = inputDatei.delete();

// Pruefen ob File-Instanz ein Verzeichnis oder eine Datei ist
boolean isDirectory = tiefesVerzeichnis.isDirectory();

// Verzeichnis lesen
File auszulesendesVerzeichnis = new File("c:/daten/temp");

// Verzeichnis als Array von Strings
String[] fileNames = auszulesendesVerzeichnis.list();

// Verzeichnis als Array von Files
File[]   files = auszulesendesVerzeichnis.listFiles();
```

Verzeichnisinhalt rekursiv ausgeben

Rekursion für Abstieg in Verzeichnishierarchie

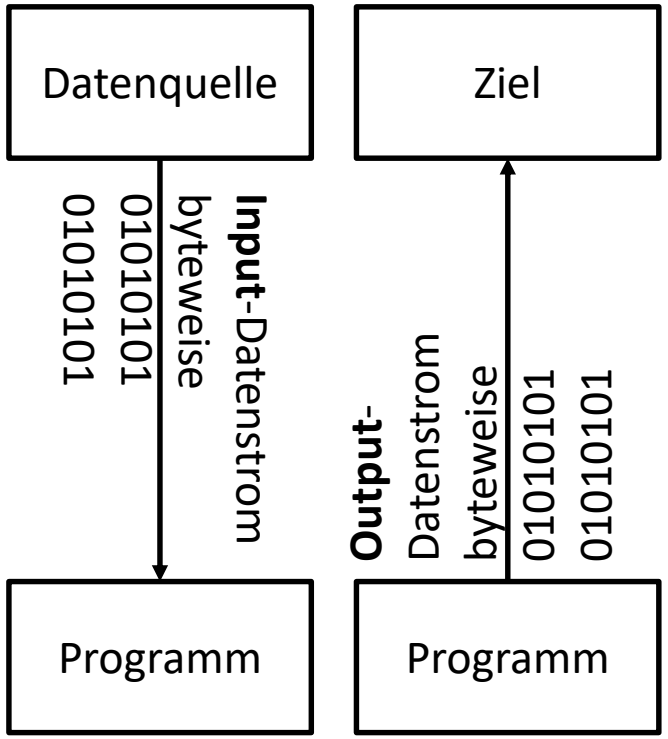
```
public static void main(String[] args) {
     // Verzeichnis lesen
     File auszulesendesVerzeichnis = new File("c:/daten/temp/");
     printAllJavaFiles(auszulesendesVerzeichnis, 1);
}

public static void dateienRekursivAusgeben(File directory, int verzeichnistiefe) {
     if(directory.isDirectory()) {
          printDateiname(directory, verzeichnistiefe);
          File[] subDirectories = directory.listFiles();
          for (File file : subDirectories) {
               dateienRekursivAusgeben(file, verzeichnistiefe+1);
          }
     } else {
          printDateiname(directory, verzeichnistiefe);
     }
}

public static void printDateiname(File directory, int depth) {
     if(directory.isDirectory()) { for(int i=0; i<depth; i++) { System.out.print("- "); }
     } else { for(int i=0; i<depth; i++) { System.out.print("   "); }}
     System.out.println(directory.getName());
}
```

DATENSTRÖME (STREAMS)

Datenströme in Java

Abstraktion der Ein-/Ausgabe

- verschiedene Stream-Arten
  - File…, Buffered…, Piped…, etc.
- Trennung von Input- und Output-Richtung
- Lesen in Datei abschnittsweise
- Sequenz von aufeinander folgenden Abschnitten



Abstraktion

- konkrete Datenquelle ist irrelevant (Netzwerk, Datei, etc.)
- Lesen von Bytes in Klassen mit *Stream
- Lesen (Schreiben) von Zeichen in Klassen mit *Reader (*Writer)

Lesen/Schreiben in Dateien

- FileInputStream/FileOutputStream -> Lesen/Schreiben von Bytes
- FileReader/FileWriter -> Lesen/Schreiben von Zeichen

Lesen einer Datei

- FileInputStream für byteweises Lesen
- read() liest ein Byte
  - Rückgabe des Bytes an aktueller Position und weiter zur nächsten Position
  - Rückgabe -1, falls Ende erreicht wurde
- Beispiel hier:
  - Byte lesen und ausgeben (bis Ende wiederholen)

```
public class DateiBytesLesen {
     public static void main(String[] args) {
          FileInputStream fis;
          try {
               fis = new FileInputStream(new File("c:/daten/temp/testtext.txt"));
               int input = fis.read();
               while(input!=-1) {
                    System.out.print((char)input);
                    input = fis.read();
               }
          } catch (IOException e) {
               // Fehlermeldung ausgeben
               e.printStackTrace();
          }
     }
}
```

Lesen einer Datei in Puffer und Buchstabe e zählen

```
public class BuchstabeEzaehlen {
     public static void main(String[] args) {
          FileInputStream fis;
          File datei;
          int eCounter=0;
          try {
               datei = new File("c:/daten/temp/testtext.txt");
               fis = new FileInputStream(datei);
               byte[] pufferArray = new byte[(int) datei.length()];
               // Lesen in Datei bis Puffer voll oder Ende erreicht
               int anzahlBytesGelesen = fis.read(pufferArray);
               fis.close();
               int counter = 0;
               while(counter<anzahlBytesGelesen&&counter<pufferArray.length) {
                    if((char)pufferArray[counter]=='e') {
                         eCounter++;
                    }
                    counter++;
               }
          } catch (IOException e) {
               // Fehlermeldung ausgeben
               e.printStackTrace();
          }
          System.out.print("Anzahl Buchstabe e: "+eCounter);
     }
}
```

- FileInputStream für byteweises Lesen einer angegebenen Anzahl Bytes
- read(byte) liest Bytes in Array
  - Füllen des Arrays mit Bytes aus Datei
- Beispiel hier:
  - Puffer-Array aus Datei füllen
  - Byte aus Puffer lesen und ‚e' zählen

Datei zeilenweise lesen

```
public class DateiZeilenweiseLesen {
     public static void main(String[] args) {
          FileReader fr;
          try {
               fr = new FileReader("c:/daten/temp/testtext.txt");
               BufferedReader br = new BufferedReader(fr);

               int counter=0;
               String eineZeile = br.readLine();
               while(eineZeile!=null) {
                    System.out.println("Zeile "+counter+"\t"+eineZeile);
                    eineZeile=br.readLine();
                    counter++;
               }
               br.close();
          } catch (IOException e) {
               // Fehlermeldung ausgeben
               e.printStackTrace();
          }
     }
}
```

- FileReader in BufferedReader verpacken
- BufferedReader ermöglicht zeilenweises Lesen (Puffern und Ende einer Zeile erkennen)
- readLine() gibt
  - Zeileninhalt zurück, oder
  - null falls Ende erreicht
- Beispiel hier:
  - Variable zeilenweise zuweisen
  - Variable unmittelbar ausgeben

SCHREIBEN IN DATEI

Beispiel FileWriter

```
public class TextInDateiSchreiben {
     public static void main(String[] args) {
          FileWriter fw = null;
          // Datei initial erzeugen und schreiben
          try {
               // nur zum Schreiben oeffen, vorherige Daten gehen verloren
               fw = new FileWriter(new File("c:/daten/temp/testtext_02.txt"));
               fw.write("Das ist ein zweiter Testtext.\n");
               fw.close();
          } catch (IOException e) {
               e.printStackTrace();
          }

          // Datei zum Anhaengen oeffnen und Text anhaengen
          try {
               // zum Anhaengen oeffen, vorherige Daten bleiben erhalten
               fw = new FileWriter(new File("c:/daten/temp/testtext_02.txt"),true);
               fw.write("Das ist eine zweite Zeile im Testtext.\n");
               fw.close();
          } catch (IOException e) {
               e.printStackTrace();
          }
     }
}
```

- FileWriter für zeichenweises Schreiben einer angegebenen Anzahl Bytes
- write(…) schreibt in Datei
  - Zeichenkette
  - Einzelnes Zeichen
  - Array von Zeichen
- Beispiel hier:
  - Schreiben eines Strings
  - Anhängen eines weiteren Strings

Beispiel BufferedWriter

```
public class DateiZeilenweiseSchreiben {
     public static void main(String[] args) {
          FileWriter fw;
          try {
               fw = new FileWriter("c:/daten/temp/testtext_03.txt");
               BufferedWriter bw = new BufferedWriter(fw);

               String eineZeile;
               for(int i=0; i<5; i++) {
                    eineZeile="Das ist die Zeile "+i;
                    bw.write(eineZeile);
                    bw.newLine(); // Zeilenende plattformspezifisch
               }

               bw.close();
          } catch (IOException e) {
               // Fehlermeldung ausgeben
               e.printStackTrace();
          }
     }
}
```

- FileWriter in BufferedWriter verpacken
- BufferedWriter ermöglicht Schreiben von Strings und Zeilenumbruch
- write(String) schreibt String in Datei
- newLine() hängt plattformspezifisches Zeilenende an
- Beispiel hier:
  - Zeileninhalt erzeugen und schreiben
  - newLine() an Zeilenende

```
Das ist die Zeile 0
Das ist die Zeile 1
Das ist die Zeile 2
Das ist die Zeile 3
Das ist die Zeile 4
```

STRUKTURIERTE DATEN AUS DATEI LESEN

Beispiel Objekte einlesen

```
public class DateiInObjekteEinlesen {
    public static void main(String[] args) {
        List<DatenContainer> listeDerObjekte = new ArrayList<DatenContainer>()
        FileReader fr;
        try {
            fr = new FileReader("c:/daten/temp/testobjekte.txt");
            BufferedReader br = new BufferedReader(fr);

            listeDerObjekte = new ArrayList<DatenContainer>();

            String eineZeile = br.readLine();
            while(eineZeile!=null && !eineZeile.trim().equals("")) {
                StringTokenizer st = new StringTokenizer(eineZeile, ";");

                int nummer = Integer.parseInt(st.nextToken().trim());
                String bezeichnung = st.nextToken();
                String inhalt = st.nextToken();
                int wert = Integer.parseInt(st.nextToken().trim());

                DatenContainer dc = new DatenContainer(nummer, bezeichnung,
                listeDerObjekte.add(dc);

                eineZeile=br.readLine();
            }
            br.close();
        } catch (...
```

- FileReader in BufferedReader verpacken
- BufferedReader ermöglicht zeilenweises Lesen (Puffern und Ende einer Zeile erkennen)
- readLine() gibt
  - Zeileninhalt zurück, oder
  - null falls Ende erreicht
- Beispiel hier:
  - Variable zeilenweise zuweisen
  - StringTokenizer für zerlegen der Zeichenkette
  - Variable unmittelbar ausgeben

```
1;Bezeichnung 1;Inhaltsangabe 1; 23
2;Bezeichnung 2;Inhaltsangabe 2; 42
3;Bezeichnung 3;Inhaltsangabe 3; 4224
4;Bezeichnung 4;Inhaltsangabe 4; 666
5;Bezeichnung 5;Inhaltsangabe 5; 777
```

Zusammenfassung

- Java-Klassen für Dateiarbeit vorhanden
- Metainformationen auslesen
- Dateiarbeit durchführen
- Lesen / Schreiben mit verschiedenen Möglichkeiten