

# Programmieren I

## Input / Output (I/O)



Heusch 2. Bd.  
Ratz 19

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

# Die Klasse `File` (1)

- Ein *`File`-Objekt repräsentiert* einen (abstrakten) *Datei- oder Verzeichnis-Pfadnamen* im Dateisystem.  
Dabei kann es sich neben *existierenden* auch um (noch) *nicht existierende* Dateien und Verzeichnisse handeln.
- Ein `File`-Objekt *enthält Informationen über ein File oder Verzeichnis* (den (abstrakten) *Pfadnamen*).  
Das `File`-Objekt *ist nicht die Datei* bzw. das Verzeichnis *selbst*!
- Der *Pfadname* kann *absolut* (z.B. "`C:/xxx/tmp1`") oder *relativ* zum aktuellen Verzeichnis (*Working Directory*) sein.
  - Default-Wert des *Working Directory* in IDEs ist das Projekt-Verzeichnis  
Der Wert kann in der IDE überschrieben werden:
    - In Eclipse z.B. über: Rechtsklick auf Projekt > Run As > Run Config. > (x)= Arguments > Working directory
    - In NetBeans über: File > Project Properties > Run > Working Directory

## Die Klasse File (2)

- Konstruktoren der Klasse File:
  - `File(String pathname)`  
erzeugt ein File-Objekt mit dem angegebenen Pfadnamen.  
Der Pfadname kann absolut (kompletter Pfad) oder relativ (zum aktuellen Verzeichnis / Working Directory) angegeben werden.
  - `File(String parent, String child)`  
`File(File parent, String child)`  
erzeugt ein File-Objekt. Der Pfadname ergibt sich aus dem *Basisverzeichnis* parent und dem *weiteren Pfad-Teil* child.
- Beispiel:

```
File test = new File("tmp1");
```

# Klasse File – Methoden (1)

- **boolean createNewFile()**  
erzeugt eine neue, leere Datei mit dem im betreffenden File-Objekt gespeicherten Namen, falls diese Datei noch nicht existiert. Im Erfolgsfall wird `true` zurückgeliefert.
- **boolean mkdir()**  
legt neues Verzeichnis mit Pfadnamen des File-Objekts an.
- **boolean mkdirs()**  
erzeugt neues Verzeichnis mit dem Pfadnamen inklusive aller nichtexistierenden und notwendigen „Vater-Verzeichnisse“.
- **boolean exists()**  
testet, ob eine Datei/Verz. mit diesem Pfadnamen existiert.
- **boolean delete()**  
löscht die Datei bzw. das Verzeichnis.  
Ein *Verzeichnis* muss *leer* sein, damit es gelöscht wird.

## Klasse File – Methoden (2)

- **boolean equals(Object obj)**  
vergleicht den *Pfad* des betreffenden File-Objekts und des übergebenen Objekts obj auf Gleichheit.
- **int compareTo(File pathname)**  
vergleicht zwei Pfadnamen lexikografisch.
- **String getName()**  
liefert den Datei- bzw. Verzeichnisnamen des File-Objekts.
- **String getPath()**  
konvertiert den (abstrakten) Pfadnamen des File-Objekts in den Pfadname-String.
- **String getAbsolutePath()**  
liefert String mit dem *absolutem Pfadnamen* des File-Objekts.

## Klasse File – Methoden (3)

- **File getAbsolutePath()**  
liefert File-Objekt mit dem absolutem Pfadnamen des File-Objekts.
- **String getParent()**  
liefert den *Pfadnamen* des „Vater“-Verzeichnisses als String, oder null, wenn es kein solches gibt.
- **File getParentFile()**  
liefert das File-Objekt des „Vater“-Verzeichnisses oder null, wenn es kein solches gibt.
- **long getFreeSpace(), getTotalSpace(), getUsableSpace()**  
liefert die Anzahl unbelegter / aller / aller nutzbarer Bytes der *Partition*, in der sich das File befindet.

## Klasse File – Methoden (4)

- `boolean isAbsolute()`  
testet ob der Pfadname absolut ist.
- `boolean isDirectory()`  
testet ob es sich beim File-Objekt um Verzeichnis handelt.
- `boolean isFile()`  
testet ob es sich bei dem File-Objekt um eine normale Datei handelt.
- `boolean isHidden()`  
testet ob es sich bei der Datei um eine versteckte Datei handelt.
- `long lastModified()`  
liefert die *Zeit*, zu welcher diese Datei zuletzt geändert wurde.

## Klasse File – Methoden (5)

- `long length()`  
liefert die Länge dieser Datei.
- `String[] list()`  
liefert ein Array von Strings mit den *Dateien* und *Unterverzeichnissen* in diesem *Verzeichnis* (natürlich nur, wenn es sich um ein Verzeichnis handelt).
- `String[] list(FilenameFilter filter)`  
liefert ein Array von Strings aller Dateien und Verzeichnisse in diesem Verzeichnis, welche *filter entsprechen*.
- `File[] listFiles()`  
liefert ein Array von Files mit den *Dateien* im Verzeichnis.
- static `File[] listRoots()`  
listet alle verfügbaren Wurzeln des Dateisystems.



## Klasse File – Methoden (6)

- `boolean canRead()`  
testet, ob die *Anwendung* aus dieser Datei *lesen darf*.
  - `boolean canWrite()`  
testet, ob die Anwendung in diese Datei *schreiben darf*.
  - `boolean setReadOnly()`  
*markiert* diese Datei mit dem „nur Lesen“-Attribut, so dass für sie nur noch Lese-Operationen erlaubt sind.
  - `boolean setReadable(boolean readable, boolean ownerOnly)`  
setzt das Recht zum Lesen der Datei. Ist `ownerOnly == true`: Lese-Operationen nur für den Benutzer erlaubt.
- Ebenso: `setWritable`, `setExecutable`

## Klasse File – Methoden (7)

- `boolean setLastModified(long time)`  
setzt die „last-modified“-Zeit für diesen Pfadnamen.
- `boolean renameTo(File dest)`  
benennt diese Datei gemäß dest um.
- `String toString()`  
liefert den Pfadnamen als String.
- `URI toURI()` // früher: `toURL()` @deprecated!  
konvertiert diesen Pfadnamen in eine URI.  
(Uniform Resource Identifier, einheitlicher Bezeichner für Ressourcen)

Zu Details siehe API-Dokumentation der Klasse `File`.

# Beispiel: Erzeugen eines Verzeichnisses und eines Files

```
import java.io.*;

public class FileIOV1 {

    public static void main(String args[]) {
        File testDir = new File("testDir");
        testDir.mkdir();
        File testFile = new File(testDir, "testFile.txt");
        try {
            testFile.createNewFile();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

## Aufgaben zu Input/Output (1), 1. Aufgabe

## ***Ein- und Ausgabe über Streams (1)***

- Sämtliche Ein- und Ausgaben in Java laufen über (Daten-) **Ströme** (engl. Streams) ab.
- Ein (Daten-)Strom ist eine Verbindung zwischen einem Programm und einer **Datenquelle** bzw. einer **Datensenke** (Datenziel).
- Diese Verbindung (der Strom) läuft dabei stets nur in *einer Richtung* (uni-direktional).
  - Ein Strom kann an der Datenquelle Daten nur aufnehmen und an der Datensenke nur Daten abgeben.

## Ein- und Ausgabe über Streams (2)

- Für *Eingaben* muss ein Programm
  - zunächst einen *Strom*, der mit der Datenquelle verbundenen ist, *anlegen* und *öffnen*,
  - dann die ankommenden Informationen *sequentiell lesen* (wobei mit dem Lesen die Daten dem Strom *entnommen werden*) und
  - den *Strom* nach seiner Verwendung wieder *schließen*.
- Für *Ausgaben* gilt das Entsprechende mit einer Datensenke.
- In einem Java-Programm wird ein *Strom* durch ein *Stream-Objekt* repräsentiert.
- Für die *Verarbeitung verschiedenartiger Datenströme* stellt Java *zahlreiche Klassen* bereit, die sich auch kombinieren lassen.

# Basisklassen für die Ein- und Ausgabe-Ströme von Java

- Für Ein- und Ausgabe-Ströme hat Java vier *Basisklassen*:  
**InputStream** und **OutputStream**  
**Reader** und **Writer**
- InputStream und OutputStream unterstützen das Lesen und Schreiben von *Bytes* (Rohdaten)  
(Basisklassen für die *Byte-orientierte* Ein-/Ausgabe).
- Reader und Writer unterstützen das Lesen und Schreiben von *Zeichen*  
(Basisklassen für die *zeichen-orientierte* Ein-/Ausgabe).
- Diese vier Klassen sind direkte Unterklassen von Object.
- Sie liegen im Paket `java.io`.

# ***Input-Quellen für Ströme***

- Input kann aus verschiedenen Quellen kommen.
- Solche Quellen können sein:
  - Byte-Array (Puffer im Speicher)
  - Zeichenkette
  - Datei
  - Pipe (für Kommunikation zwischen Threads)
  - Andere Quellen,  
z.B. Netzwerk-Verbindungen, insbesondere Internet-Verbindungen

# Unterklassen der Basisklasse InputStream für diese Quellen

- Von der Basisklasse InputStream gibt es für die verschiedenen Quellen-Arten unterschiedliche Unterklassen:
  - ByteArrayInputStream  
zum Einlesen aus einem Byte-Array (Puffer im Speicher).
  - FileInputStream  
zum Einlesen aus einer Datei.
  - PipedInputStream  
z. Lesen von Daten, die durch PipedOutputStream erzeugt wurden
- Daneben gibt es weitere Unterklassen von InputStream.
- Außerdem gibt es Klassen, die ein InputStream-Objekt liefern, z.B. für Netzwerke die Klasse  
`java.net.URLConnection` mit der Methode  
`InputStream getInputStream()`
- Die Basisklassen InputStream etc. sind *abstrakte* Klassen, die Unterklassen sind *konkrete* Klassen.



# Output-Senken für Ströme

- Output kann in verschiedene Senken fließen.
- Solche Senken können sein:
  - Byte-Array
  - Zeichenkette
  - Datei
  - Pipe (für Kommunikation zwischen Threads)
  - andere Senken, wie z.B. Internet-Verbindungen

# Unterklassen der Basisklasse OutputStream für diese Senken

- Von OutputStream gibt es für die verschiedenen Arten von Senken unterschiedliche Unterklassen:
  - ByteArrayOutputStream  
zum Schreiben in ein Byte-Array (Puffer im Speicher).
  - FileOutputStream  
zum Schreiben in eine Datei.
  - PipedOutputStream  
zum Schreiben von Daten, die durch PipedInputStream gelesen werden können.
- Daneben gibt es weitere Unterklassen von OutputStream.
- Außerdem gibt es Klassen, die ein OutputStream-Objekt liefern, z.B. die Klasse `java.net.URLConnection` mit der Methode `OutputStream getOutputStream()`

# Die Basisklassen Reader und Writer und deren Unterklassen

- Die *Basisklassen* Reader und Writer sind *weitgehend identisch* mit den Basisklassen InputStream und OutputStream, allerdings Lesen bzw. Schreiben diese beiden Klassen **Zeichen** (statt *Bytes*).
- Ebenso sind auch *alle Unterklassen* jeweils *mehr oder weniger gleich*.
  - |              |              |
|--------------|--------------|
| StringReader | StringWriter |
| FileReader   | FileWriter   |
| PipedReader  | PipedWriter  |
| etc.         |              |

# Beispiel: Lesen eines Bytes aus einer existierenden Datei

```
import java.io.*;

public class FileIOV2 {
    public static void main(String args[]) throws IOException {
        int test;
        // Anlegen des FileInputStreams
        InputStream s = new FileInputStream("tmp0.txt");
        test = s.read(); // read() kann IOException werfen
        System.out.println("test: " + test);
        s.close(); // close() zum Schließen der Ressource;
                  // close() kann IOException werfen
    }
}
```



Hier *kein*  
Exception-  
Handling!

 Ausgabe

test: <ganzzahliger Wert des ersten Bytes der Datei>

# Exception-Handling & Schließen von Ressourcen

- Ressourcen **müssen** geschlossen werden, wenn sie nicht mehr gebraucht werden.
  - Erste Möglichkeit: Aufruf von `close()`, siehe Beispiel oben
  - Das Schließen kann aber *eine* `IOException` auslösen,  
→ try-catch-Anweisung um das `close()` nötig
- Seit Java 1.7 gibt es *try-with-resources* (try-Block mit Parameter)
  - Die im Parameter angegebenen Ressourcen werden nach dem try-Block *automatisch geschlossen*, und eventuell auftretende Exceptions werden wie üblich behandelt (in einem catch-Block).
  - Im Parameter lassen sich *eine oder auch mehrere Ressourcen* deklarieren. Beispiel:

 Ab Java 1.7

```
try ( InputStream is = new FileInputStream(source);  
      OutputStream os = new FileOutputStream(target) ) {  
    // Ströme zum Lesen und zum Schreiben  
} // Ressourcen werden automatisch geschlossen
```

# Hier vorheriges Beispiel *mit* Exception-Handling & automatischem Schließen der Ressource

```
import java.io.*;
```

```
public class FileIOV3 {  
    public static void main(String args[]) {  
        int test;  
        // Anlegen des FileInputStreams  
        try ( InputStream s = new FileInputStream("tmp0.txt") ) {  
            test = s.read();           // read() kann IOException werfen  
            System.out.println("test: " + test);  
        } catch (IOException e) {  
            System.err.println("I/O Error");  
        }  
    }  
}
```

Kein Weiterleiten von  
Exceptions mehr!

try-with-resources

 Ausgabe

```
test: <ganzzahliger Wert des ersten Bytes der Datei>
```

## Basisklasse InputStream – Einige Methoden (1)

- **int read()**  
liest *das nächste* Byte von diesem InputStream (und entnimmt es dabei). Liefert -1 bei Ende des Streams.
- **int read(byte[] b)**  
liest *mehrere* Bytes vom InputStream (max. `b.length` viele) und speichert sie in dem Array `b`.  
Rückgabewert: Anzahl der gelesenen Bytes
- **int read(byte[] b, int off, int len)**  
liest `len` Bytes von diesem InputStream in das Byte-Array `b` ab dem Index `off`.
- **int available()**  
liefert die Anzahl der *verfügbaren* Bytes, die sofort ohne Blockierungen gelesen oder übersprungen werden können.
- **long skip(long n)**  
überspringt `n` Bytes dieses InputStreams.

## Basisklasse InputStream – Einige Methoden (2)

- **void mark(int readAheadLimit)**  
kennzeichnet die aktuelle Position in diesem InputStream (readAheadLimit: Puffergröße).
- **void reset()**  
setzt die Leseposition dieses InputStreams wieder an die zuletzt mit der Methode mark() gekennzeichnete Stelle.
- **boolean markSupported()**  
true, wenn dieser InputStream die Methoden mark() und reset() unterstützt (Position merken/zurücksetzen).
- **void close()**  
schließt diesen InputStream und gibt die von ihm belegten Ressourcen frei.



# Basisklasse *OutputStream* – einige Methoden

- **void write(int b)**  
schreibt ein einzelnes *Byte* auf diesen *OutputStream*.  
Geschrieben werden die niederwertigen 8 Bits von *b*.
- **void write(byte[] b)**  
schreibt *b.length* Bytes vom Array *b* auf diesen *OutputStream*.
- **void write(byte[] b, int offset, int len)**  
schreibt *len* Bytes vom Array *b* ab Index *offset* auf diesen *OutputStream*.
- **void flush()**  
„spült“ (leert) einen eventuellen Puffer des *OutputStream*-Objekts  
(durch die sofortige Abarbeitung aller noch anstehenden Bytes).  
Wichtig bei *gepufferten* Ausgaben!
- **void close()**  
schließt diesen *OutputStream* und gibt die von ihm belegten  
Ressourcen frei.

# Die Klassen `FileWriter` und `FileReader`

## – Konstruktoren und Methoden –

- *Konstrukturen* der Klasse `FileWriter` (Auswahl):
  - `FileWriter(String fileName)`  
erzeugt einen *Zeichen*-Ausgabestrom zur *Datei* mit dem Namen `fileName`. *Der bisherige Inhalt der Datei wird gelöscht!*
  - `FileWriter(File file)`  
wie oben, hier Zeichen-Ausgabestrom zur Datei `file` (`File`-Objekt).
  - `FileWriter(String fileName, boolean append)`  
`FileWriter(File file, boolean append)`  
wie oben. Falls `append` den Wert `false` hat, wird der bisherige Inhalt der Datei gelöscht/überschrieben.  
Falls `append` den Wert `true` hat, werden die ausgegebenen Zeichen an den bereits bestehenden Datei-Inhalt *angehängt!*
- Die *Methoden* der Klasse `FileWriter` entsprechen den Methoden der Klasse `OutputStream` (bzw. `FileOutputStream`), aber Ausgabe von *Zeichen* bzw. *Zeichenketten* statt Bytes.
- Für die Klasse *`FileReader`* gilt Entsprechendes.

# Beispiel: Schreiben in eine Datei (zeichenbasiert)

```
import java.io.*;

public class FileIO1 {
    public static void main(String args[]) {
        String testFile = "test.txt"; // Name d. Datei im akt. Verz.
        try ( Writer fWriter = new FileWriter(testFile) ) {
            fWriter.write("Testline\n"); // s. Anmerkung unten
            fWriter.write("Second line\n");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

**Anmerkungen** zum Zeilentrenner: siehe Folien 31/32


# Beispiel: *Erstes Zeichen* aus der Datei lesen

```
import java.io.*;

public class FileIO2 {

    public static void main(String args[]) {
        String testFile = "test.txt";
        try ( Reader fReader = new FileReader(testFile) ) {
            int c = fReader.read(); // read() liefert int-Wert
            System.out.println("Read: " + (char) c);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Anmerkung: Die Methode `read()` von `Reader` gibt das gelesene Zeichen als `int`-Wert zurück. Am Dateiende liefert sie den Wert `-1`.

 **Ausgabe**  
Read: T

# Beispiel: *Alle* Zeichen aus der Datei lesen

```
import java.io.*;

public class FileIO3 {

    public static void main(String args[]) {
        String testFile = "test.txt";
        try ( Reader fReader = new FileReader(testFile) ) {
            int c;
            while ((c = fReader.read()) != -1) { // nächstes Z. einles.
                System.out.print((char) c);      // Wenn -1: Dateiende
            }
        } catch ( IOException e ) {
            e.printStackTrace();
        }
    }
}
```

 **Ausgabe**

```
Test line
Second line
```

# Weiteres Beispiel: Ausgabe anhängen an einen bereits existierenden Dateiinhalt

```
import java.io.*;

public class FileAppend {

    public static void main(String[] args) {
        try ( Writer out = new FileWriter("test.txt", true) ) {
            String s = "Neuer Text\n";
            out.write(s); // wird angehängt
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Die Datei `test.txt` wird nun *zum Anhängen geöffnet*, d.h. der existierende Inhalt bleibt erhalten.  
Ist die Datei nicht vorhanden, wird sie neu angelegt.
- Für `FileOutputStream` existiert ein analoger Konstruktor zum Anhängen

# Anmerkungen zum Zeilentrenner (Carriage Return und Line Feed) (1)

- Zeilentrenner sind in Unix/Linux, Windows und MacOS unterschiedliche Zeichen bzw. Zeichenkombinationen:

Unix/Linux	Windows	MacOS
\n (LF)	\r\n (CRLF)	\r (CR)

- Statt diese Zeichen(-kombinationen) direkt zu verwenden, sollte besser eine *Methode* aufgerufen werden, die den Zeilentrenner-String der jeweiligen Plattform liefert:

```
String newLine = System.lineSeparator();      oder
```

```
String newLine = System.getProperty("line.separator");
```

- Beispiel: Ausgabe in Folie 27 mit `lineSeparator()`:

```
fWriter.write("Testline" + System.lineSeparator());  
fWriter.write("Second line" + System.lineSeparator());
```

# Anmerkungen zum Zeilentrenner (Carriage Return und Line Feed) (2)

- Weitere Möglichkeit: Methode `println()` der Klassen `PrintWriter` (s. Folien 37ff.) oder `PrintStream` verwenden.

Beispiel:

```
// Ausgabe von Array String[] buffer in Datei test2.txt
try (PrintWriter pw = new PrintWriter("test2.txt")) {
    for (int i = 0; i < buffer.length; i++) {
        pw.println(buffer[i]);
    }
} catch ( IOException e ) { /* error handling */ }
```

- Möglichkeit bei *formatierter Ausgabe* mit der Methode `printf()` dieser Klassen:  
Format-Spezifizierer `%n` verwenden (statt `\n`).

**Aufgaben zu Input/Output (1), 2. Aufgabe**



# Elementare und höherwertige I/O-Klassen

- Die bisher behandelten Klassen bieten eher „*elementare*“ (einfache) *Ein-/Ausgabe-Funktionen*,
  - wie z.B. Öffnen/Schließen eines Stroms, byteweise bzw. zeichenweise Ein-/Ausgabe
- Daneben gibt es in Java eine Reihe von Klassen mit „*höherwertigeren*“ *Ein-/Ausgabe-Funktionen*, die zusätzliche Funktionalität bietenden, wie z.B.
  - die Ein-/Ausgabe primitiver Datentypen in ihrer Binärdarstellung (z.B. `long`- oder `double`-Daten)
  - gepufferte Ein-/Ausgabe.

# Einordnung am Beispiel von InputStream- und Reader-Klassen

## ■ Klassen mit „elementaren“ Funktionen

- FileInputStream
- ByteArrayInputStream
- PipedInputStream
  
- FileReader
- CharArrayReader
- StringReader
- PipedReader

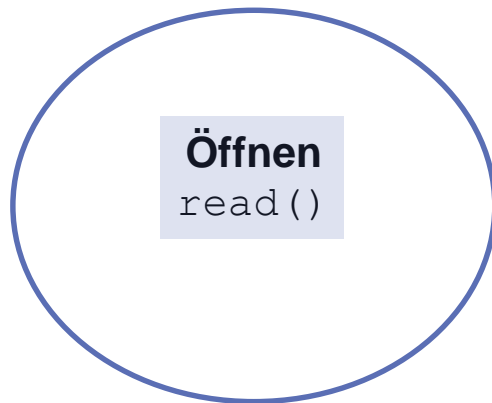
## ■ Klassen mit „höherwertigen“ Funktionen

- DataInputStream
- BufferedInputStream
- FilterInputStream
- SequenceInputStream
  
- BufferedReader
- InputStreamReader
- FilterReader
- LineNumberReader

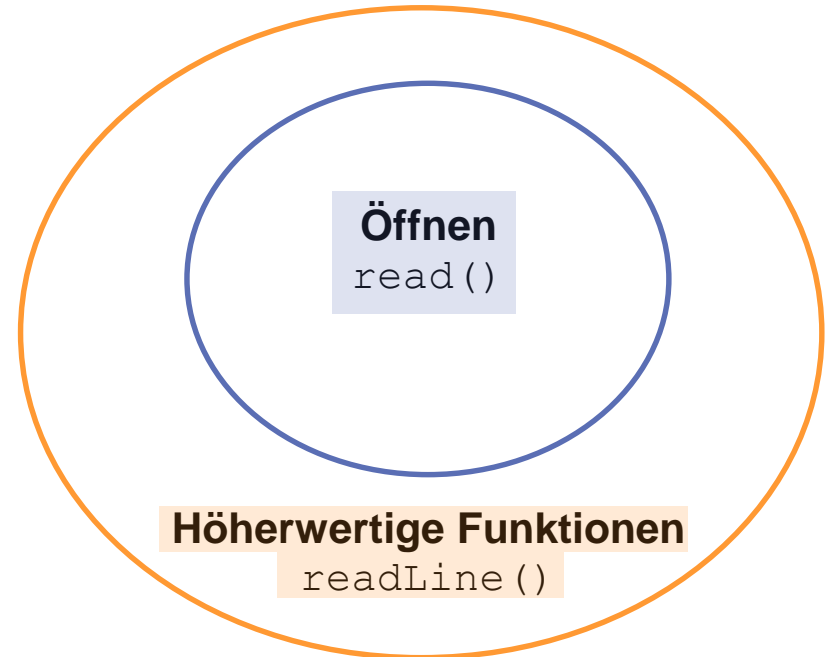
Typische  
Verkettung:

```
FileReader fr = new FileReader("test.txt");  
BufferedReader br = new BufferedReader(fr);
```

# Verkettung verdeutlicht am Beispiel der zwei o.g. Reader-Klassen



```
FileReader fr =  
    new FileReader("test.txt");  
fr.read();
```



```
FileReader fr =  
    new FileReader("test.txt");  
BufferedReader br =  
    new BufferedReader(fr);  
br.readLine();
```

# Im Weiteren näher betrachtete Klassen mit "höherwertigen Funktionen"

- Im Folgenden werden vier häufig verwendete Klassen mit "höherwertigen" Funktionen näher betrachtet:
  - Klasse `PrintWriter`
  - Klasse `BufferedReader`
  - Klasse `DataOutputStream`
  - Klasse `DataInputStream`

# Die Klasse `PrintWriter`

- Diese Klasse stellt funktional mächtigere Methoden für die *Formatierung* und *zeichenorientierte Ausgabe* von primitiven Datentypen und Objekten bereit (`print()`, `println()`, `printf()` etc.).

- Vererbungshierarchie:

```
java.lang.Object
|
+--java.io.Writer
|
+--java.io.PrintWriter
```

- `public class PrintWriter extends Writer`

# Klasse `PrintWriter` – Konstruktoren

- `PrintWriter(Writer out)`  
erzeugt einen *Zeichen-Ausgabestrom*, der über dem *Zeichen-Ausgabestrom* `out` liegt.  
Der *Zeilen-Ausgabepuffer* wird nicht automatisch geleert!
- `PrintWriter(Writer out, boolean flush)`  
wie oben. Falls `flush` den Wert `true` hat, wird der *Zeilen-Ausgabepuffer* bei `println()`-Aufrufen automatisch geleert.
- `PrintWriter(OutputStream out)`  
erzeugt einen *Zeichen-Ausgabestrom*, der über dem *Byte-Ausgabestrom* `out` liegt.
- `PrintWriter(OutputStream out, boolean flush)`
- `PrintWriter(File file)`  
Parameter `file`: Datei, in die ausgegeben wird.
- `PrintWriter(String fileName)`

# Klasse `PrintWriter` – Methoden (Auswahl)

- `void print(type x)`  
erzeugt eine dem Datentyp `type` entsprechende *String-Darstellung* für den Wert `x` und schreibt diese in den Ausgabestrom.
- `void println()`  
schreibt einen *Zeilenwechsel-String* in diesen Ausgabestrom.
- `void println(type x)`  
ruft `print(x)` und anschließend `println()` auf.
- `PrintWriter printf(String format, Object... args)`  
schreibt eine *formatierte* Zeichenkette in diesen Ausgabestrom.
- `PrintWriter append(CharSequence csq)`  
schreibt die Zeichenfolge `csq` in den Strom.
- `void flush()`  
leert („spült“) den Puffer
- `void close()`  
schließt den Strom. Ruft zuerst `flush()` auf.

# Die Klasse BufferedReader (1)

- *Gepufferte* Eingabe- und Ausgabe-Klassen dienen zur Verbesserung der *Performance* der Ein- und Ausgabe.
- Hier: Am Beispiel von `BufferedReader`
  - Wenn *sehr viele Zeichen* von einer Datenquelle gelesen werden, kann dies *ineffizient* sein, wenn jedes Zeichen in einem *eigenen Vorgang* eingelesen wird.
  - Für *jedes einzelne Zeichen* muss dann z.B. die entsprechende Verbindung mit dem Speichermedium bzw. der Netzwerkumgebung separat aufgebaut und wieder abgebaut werden.
  - Die Klasse `BufferedReader` ermöglicht es, eine *ganze Reihe* von gelesenen Zeichen in einem internen *Puffer zwischenzuspeichern*.
  - Dadurch können *ganze Sequenzen* von Zeichen zu *größeren Blöcken* zusammengefasst und in *einem* Vorgang eingelesen werden.



# Die Klasse `BufferedReader` (2)

## ■ Vererbungshierarchie:

```
java.lang.Object
|
+--java.io.Reader
|
+--java.io.BufferedReader
```

## ■ `public class BufferedReader extends Reader`

## ■ Konstruktoren:

- `BufferedReader(Reader in)`
- `BufferedReader(Reader in, int size)`  
size gibt die Größe des Eingabepuffers vor.

# Klasse BufferedReader – Methoden

- `int read()`
- `int read(char[] cbuf, int off, int len)`
- `String readLine()`
- `boolean ready()`  
wahr, wenn das nächste `read()` garantiert nicht wegen nicht vorhandener Daten wartet (blockiert)
- `boolean markSupported()` unterstützt merken / zurücksetzen einer Leseposition, entspr. wie in `InputStream`
- `void mark(int readAheadLimit)`
- `void reset()`
- `long skip(long n)`
- `void close()`

# Beispiel: Schreiben in Datei mit BufferedWriter

```
import java.io.*;
import de.dhbwka.java.exercise.classes.Polynomial; // Polyn. 2. Grades
                                                    // s. Aufgaben "Klassen (2)"

public class FileWriteOut {
    public static void main(String[] args) {
        try (BufferedWriter out
              = new BufferedWriter(new FileWriter("tmp2"))) {
            Polynomial p = new Polynomial(1, 0, 1);
            for (double x = -3.0; x <= 3.1; x += 0.5) {
                String str = x + " " + p.f(x) + System.lineSeparator();
                out.write(str);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Die Daten stehen hier als *Folge von Characters* in der Datei tmp2. Die Datei kann mit einem Editor gelesen werden.
- Vergleiche mit Beispiel zu DataOutputStream auf Folie 52.
- Die Ausgabe könnte auch ohne *Pufferung* (ohne BufferedWriter) erfolgen (nur FileWriter), wäre dann aber ineffizienter.

# Beispiel: Lesen aus Datei mit BufferedReader

```
import java.io.*;

public class FileReadIn {

    public static void main(String[] args) {
        try (BufferedReader br =
            new BufferedReader(new FileReader("tmp2"))) {
            while (br.ready()) {
                String line = br.readLine();
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Ergebnis des Beispiels

## ■ Ausgabe auf der Konsole:

```
-3.0 10.0  
-2.5 7.25  
-2.0 5.0  
-1.5 3.25  
-1.0 2.0  
-0.5 1.25  
0.0 1.0  
0.5 1.25  
1.0 2.0  
1.5 3.25  
2.0 5.0  
2.5 7.25  
3.0 10.0
```

## Beispiel: Lesen in eine Datenstruktur

- Normalerweise werden die gelesenen Daten zur weiteren Verarbeitung in einer *Datenstruktur* (z.B. Array oder ArrayList) gespeichert, z.B.:

```
import java.util.ArrayList; // fertige Datenstruktur (Liste)

// ...
ArrayList<String> lines = new ArrayList<>(); // zweites Semester!
try ( BufferedReader br =
        new BufferedReader(new FileReader("tmp2")) ) {
    while (br.ready()) {
        String line = br.readLine();
        lines.add(line); // String-Daten an Liste anhängen
    }
} catch (IOException e) {
    // error
}
// ...
```

Die String-Daten können dann  
„ausgepackt“ und verarbeitet werden.

**Aufgaben zu Input/Output (2)**

# Die Klasse `DataOutputStream`

- `DataOutputStream` und `DataInputStream` ermöglichen es, Werte *primitiver Datentypen* in ihrer *Binärdarstellung* zu schreiben und zu lesen.

- Vererbungshierarchie von *`DataOutputStream`*:

```
java.lang.Object
|
+-- java.io.OutputStream
|
+-- java.io.FilterOutputStream
|
+-- java.io.DataOutputStream
```

- `public class DataOutputStream`  
    `extends FilterOutputStream implements DataOutput`
- Konstruktor: `DataOutputStream(OutputStream out)`  
erzeugt einen *Binärdaten-Ausgabe-Strom*, der den zugrundeliegenden Ausgabe-Strom `out` nutzt.

# Klasse DataOutputStream - Methoden (1)

- `void writeBoolean(boolean v)`
- `void writeByte(int v)` // ebenso `write(int v)`
- `void writeBytes(String s)` // niederw. Bytes d. Zeichen
- `void writeChar(int v)`
- `void writeChars(String s)`
- `void writeShort(int v)`
- `void writeInt(int v)`
- `void writeLong(long v)`
- `void writeFloat(float v)`
- `void writeDouble(double v)`



## Klasse DataOutputStream - Methoden (2)

- `void flush()`  
zum Leeren des Puffers
- `void close()`  
zum Schließen des Ausgabestroms. Ruft zuerst `flush()` auf.
- `int size()`  
liefert die *Anzahl* der bisher in diesen Ausgabestrom geschriebenen Bytes.

Siehe auch: Methoden der Basisklasse OutputStream.

# Die Klasse `DataInputStream`

- Vererbungshierarchie von `DataInputStream`:

```
java.lang.Object
|
+--java.io.InputStream
|
+--java.io.FilterInputStream
|
+--java.io.DataInputStream
```

- `public class DataInputStream`  
    extends `FilterInputStream` implements `DataInput`
- Konstruktor: `DataInputStream(InputStream in)`  
    erzeugt einen *Binärdaten-Eingabe-Strom*, der den zugrundeliegenden Eingabe-Strom `in` nutzt.

# Klasse DataInputStream – Methoden

- `boolean readBoolean()`
- `char readChar()`
- `byte readByte()`
- `short readShort()`
- `int readInt()`
- `long readLong()`
- `float readFloat()`
- `double readDouble()`
- `int skipBytes(int n)` // um n Bytes zu überspringen
- `void readFully(byte[] b)`  
versucht, das *gesamte* Array b zu füllen; sonst Exception
- `void readFully(byte[] b, int off, int len)`

Siehe auch die Methoden der Basisklasse `InputStream`.

# Beispiel: double-Werte als *Binärdaten* in eine Datei schreiben mit DataOutputStream

```
import java.io.*;
import de.dhbwka.java.exercise.classes.Polynomial; // Polynom 2. Grades
                                                    // s. Aufg. "Klassen (2)"

public class FileDataOut {
    public static void main(String args[]) {
        try ( DataOutputStream out =
                new DataOutputStream(new FileOutputStream("bindata")) ){
            Polynomial p = new Polynomial(1, 0, 1);
            for (double x = -3.0; x <= 3.0; x += 0.5) {
                out.writeDouble(x);
                out.writeDouble(p.f(x)); // s. Aufgaben zu "Klassen"
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Die Daten stehen hier in Form von Binärdaten (in double-Darstellung) in der Datei bindata.
- Diese Datei ist mit einem Editor nicht lesbar.
- Vergleiche mit Beispiel zu BufferedWriter auf Folie 43.

# Beispiel: double-Werte als *Binärdaten* aus einer Datei lesen mit DataInputStream

```
import java.io.*;

public class FileDataIn {
    public static void main(String args[]) {
        try ( DataInputStream in =
                new DataInputStream(new FileInputStream("bindata")) ){
            double x, y;
            boolean eof = false;
            while ( !eof ){
                try {
                    x = in.readDouble();
                    y = in.readDouble();
                    System.out.println(x + " " + y);
                } catch (EOFException e) {
                    eof = true;
                }
            }
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

EOFException wird hier bewusst als Abbruchkriterium verwendet.

# Überblick über das Paket `java.io`

- Hier noch ein Überblick über das Paket `java.io` (Java 8) :
  - Klassen
  - Interfaces
  - Exceptions


# Klassen von java.io (1)

- **BufferedInputStream**
- **BufferedOutputStream**
- **BufferedReader**
- **BufferedWriter**
- **ByteArrayInputStream**
- **ByteArrayOutputStream**
- **CharArrayReader**
- **CharArrayWriter**
- **Console**
- **DataInputStream**
- **DataOutputStream**

`// String readLine()`

## Klassen von java.io (2)


- **File**
- **FileDescriptor**
- **FileInputStream**
- **FileOutputStream**
- **FilePermission**
- **FileReader**
- **FileWriter**
- **FilterInputStream**
- **FilterOutputStream**
- **FilterReader**
- **FilterWriter**
- **InputStream** // System.in
- **InputStreamReader**
- **LineNumberInputStream** // @deprecated

 Stand: Java 1.8



## Klassen von java.io (3)

- `LineNumberReader` // Unterklasse von `BufferedReader`
- `ObjectInputStream` // Einlesen von Objekten
- `ObjectInputStream.GetField`
- `ObjectOutputStream` // Ausgabe von Objekten
- `ObjectOutputStream.PutField`
- `ObjectStreamClass`
- `ObjectStreamField`
- **`OutputStream`**
- `OutputStreamWriter`
- `PipedInputStream`
- `PipedOutputStream`
- `PipedReader`
- `PipedWriter`
- **`PrintStream`** // **`System.out`, `System.err`**

 Stand: Java 1.8

## Klassen von java.io (4)

- **PrintWriter**
- **PushbackInputStream**
- **PushbackReader**
- **RandomAccessFile** // Lesen und Schreiben!
- **Reader**
- **SequenceInputStream**
- **SerializablePermission**
- **StreamTokenizer** // Liest Datei im C++-Stil
- **StringBufferInputStream** // @deprecated
- **StringReader**
- **StringWriter**
- **Writer**

## *Interfaces von java.io*

- `Closable` `// close()`
- `DataInput`
- `DataOutput`
- `Externalizable`
- `FileFilter`
- `FilenameFilter`
- `Flushable` `// flush()`
- `ObjectInput` `// Object readObject()`
- `ObjectInputValidation`
- `ObjectOutput` `// writeObject(Object obj)`
- `ObjectStreamConstants`
- `Serializable`

# *Exceptions in java.io*

- `CharConversionException`
- `EOFException`
- `FileNotFoundException`
- `InterruptedIOException`
- `InvalidClassException`
- `InvalidObjectException`
- `IOException`
- `NotActiveException`
- `NotSerializableException`
- `ObjectStreamException`
- `OptionalDataException`
- `StreamCorruptedException`
- `SyncFailedException`
- `UncheckedIOException`
- `UnsupportedEncodingException`
- `UTFDataFormatException`
- `WriteAbortedException`
- Möglicher Error: `IOError`  
(für schwerwiegende  
Ausnahmesituationen)



**Stand: Java 1.8**

# Anmerkung: Das Paket `java.nio`

- Erweiterung der I/O-Funktionalität um:
  - Reguläre Ausdrücke
  - Memory-mapped I/O und Datei-Locks
  - Zeichen-Codierer und -Decodierer
    - Konvertierung zwischen (externen) Bytes und internen (UTF-16/Unicode) Zeichen
  - Puffer und Kanäle (Buffers, Channels)
    - Neue I/O-Abstraktion: Pufferbasierte I/O
  - I/O-Multiplexing / asynchrone I/O (Selector)
    - Ereignis-orientierte I/O
- Im Rahmen dieser Vorlesung gehen wir auf `java.nio` selbst nicht ein.
- Im Sommersemester werden einige Aspekte von NIO.2 (seit Java 7) behandelt.

# Direkter Vergleich: Vollständiges *Exception-Handling* ohne und mit *try-with-resources*

```
InputStream in = null;
try {
    in = new FileInputStream("a.txt");    // FileNotFoundException möglich
    // IO-Operationen
}
catch (IOException e) {
    e.printStackTrace();
}
finally {
    try {
        if ( in != null ){
            in.close();
        }
    } catch (IOException e){ /* ... */ }
}
```

 Bis Java 1.6

```
try (InputStream in = new FileInputStream("a.txt")){
    // IO-Operationen
} catch (IOException e) {
    e.printStackTrace();
}
```

 Ab Java 1.7

# Ergänzung: Gleichartige Ausgabe in Datei oder nach System.out

- Basiert darauf, dass `System.out` (Standard-Ausgabe-Strom) eine Referenz auf ein `PrintStream`-Objekt enthält. Beispiel:

```
PrintStream ps;  
boolean fileOutput = true;    // Ausgabe-Schalter  
// ...  
  
// Ausgabestrom an Variable ps zuweisen  
if (fileOutput) {             // Ausgabe in Datei  
    ps = new PrintStream(new FileOutputStream(outfile));  
}  
else {                         // Ausgabe in Standard-Ausgabe-Strom  
    ps = System.out;          // (ist ebenfalls PrintStream)  
}  
  
// Nun die Ausgabe für beide Fälle  
ps.println("Just a test...");
```

Aufgaben zu Input/Output (3)  
Aufgaben zu Input/Output JTail