

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);  
    }  
}
```

java.io + java.nio

■ Für das Handling von **Dateien** gibt es **zwei „Wege“**

Das Original: `java.io` + `File`

- Für die „Basics“ vollkommen ausreichend
- Blockierendes I/O
- Fehlerbehandlung nicht ganz optimal
- Probleme bei symbolischen Links
- Probleme bei Verzeichnissen mit vielen Unterverzeichnissen / Dateien
- **Aber:** Noch häufig verwendet in Beispielen & Code „den man so findet“ → Auslassen keine Option!

Neuer: `java.nio` + `Path`

- Adressiert Probleme die mit `java.io` identifiziert wurden
- Ergänzt `java.io` teilweise
- Asynchrone/Nicht-Blockierende I/O-Operationen unterstützt
- UTF-8 als Standard-Z Zeichensatz
- Erweiterte Datei-Operationen
- **Aber:** Alles wäre zu Umfangreich → wir zeigen die Basics im Vergleich mit `java.io`

IO



NIO: Java 1.4
NIO.2: Java 1.7

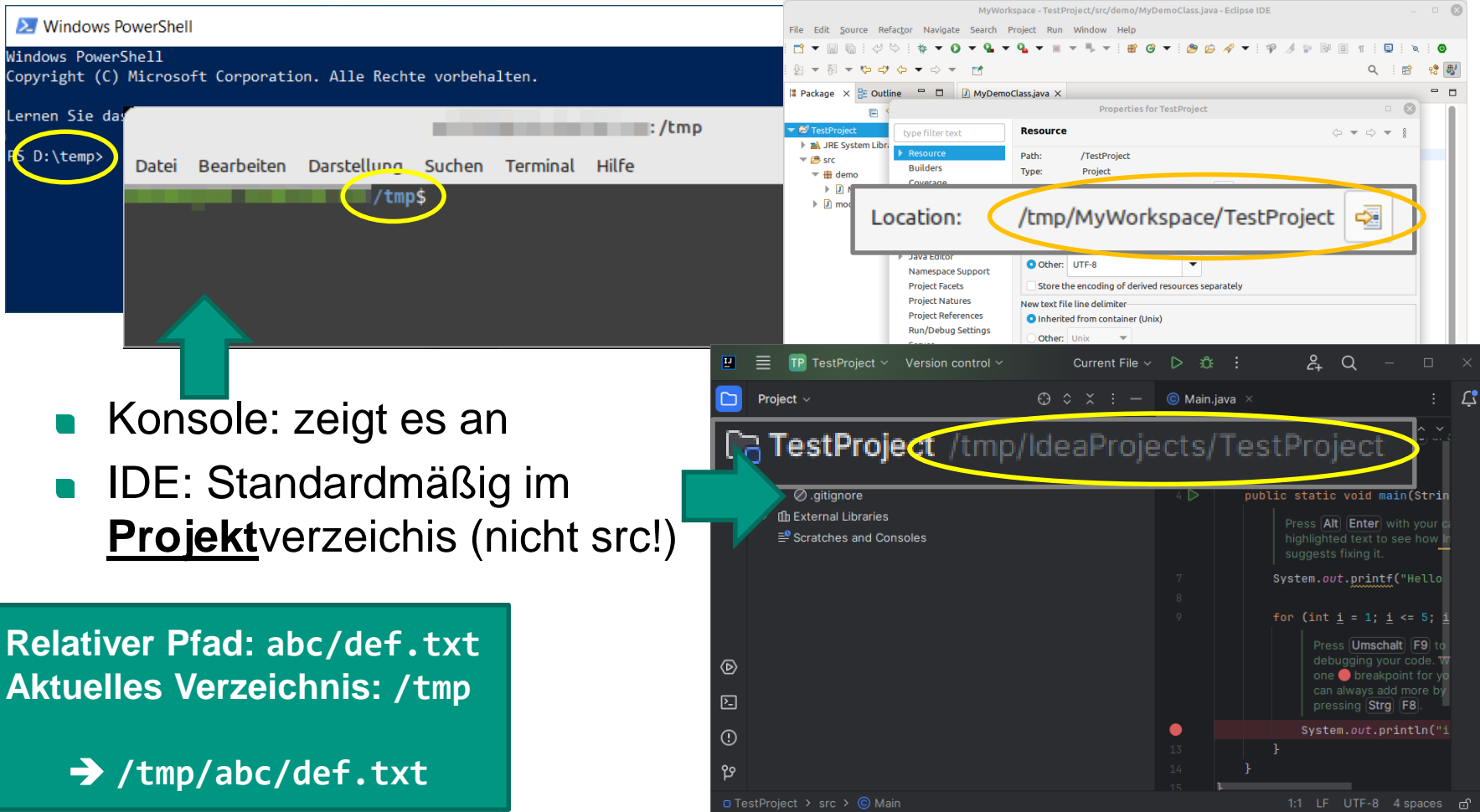
NIO

Vorab: Verzeichnis- und Dateipfade (1)

- **Verzeichnisse** haben einen Namen und ggf. ein Eltern-Verzeichnis in dem sie liegen
 - Beides zusammen ist der **Verzeichnispfad**
- **Dateien** werden über das Verzeichnis in dem sie gespeichert sind und ihrem Namen angesprochen
 - Beides zusammen ist der **Dateipfad**
- Dateipfade gibt es in 2 Varianten:
 - **Absolut** → von überall eindeutig (bspw. C:\temp\abc\def.txt)
 - Beginnt unter Windows mit **\$LAUFWERK:**
 - unter Unix-Systemen mit / (forward slash)
 - **Relativ** → Angabe abhängig davon wo man sich „gerade befindet“ (bspw. abc/def.txt)

Vorab: Verzeichnis- und Dateipfade (2)

■ Und wo befinde ich mich gerade?



The composite image illustrates the current directory and project location across different environments:

- Windows PowerShell:** The prompt shows the current directory is `D:\tmp` (circled in yellow). The command prompt is `/tmp$` (circled in yellow).
- Eclipse IDE:** The Properties window for the TestProject shows the Location as `/tmp/MyWorkspace/TestProject` (circled in yellow).
- IntelliJ IDEA:** The Project view shows the project location as `/tmp/IdeaProjects/TestProject` (circled in yellow).

Green arrows point from the PowerShell prompt and the IntelliJ IDEA project view to the text box below.

Relativer Pfad: `abc/def.txt`
Aktuelles Verzeichnis: `/tmp`
➔ `/tmp/abc/def.txt`

- 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!
- Konstruktoren:
 - **File**(String pathname) `File test = new File("tmp.txt");`
erzeugt ein File-Objekt mit dem angegebenen Pfadnamen.
 - **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.

Die Schnittstelle Path



- Analog zu `File`-Objekten repräsentiert ein `Path` nur die „Adresse“ zu einem Verzeichnis bzw. einer Datei
- `Path`-Instanzen werden im Gegensatz zu `Files` nicht mit einem eigenen Konstruktor erzeugt
 - Dafür gibt es die Hilfsklasse `Paths`

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

- `Path`-Instanzen enthalten im Gegensatz zu `Files` auch keine Informationen über das Verzeichnis oder die Datei bereit
 - Dafür gibt es die Hilfsklasse `Files`

Vergleich: Ein paar wichtige Funktionalitäten (1)

```
File dir =  
    new File("/tmp/a");  
File file =  
    new File("/tmp/b.txt");
```

```
Path dirPath =  
    Paths.get("/tmp/a");  
Path filePath =  
    Paths.get("/tmp/b.txt");
```

Was	File	Path
Verzeichnis anlegen	<code>dir.mkdir()</code>	<code>Files.createDirectory(dirPath)</code>
Verzeichnisse anlegen	<code>dir.mkdirs()</code>	<code>Files.createDirectories(dirPath)</code>
Wenn das Verzeichnis <code>/tmp/abc/def/ghi</code> angelegt werden soll und bisher „nur“ <code>/tmp</code> existiert wird die erste Variante fehlschlagen. Bei der zweiten Variante werden alle „Zwischenverzeichnisse“ ebenfalls angelegt.		
Test ob „Adresse“ ein existierendes Verzeichnis ist	<code>dir.isDirectory()</code>	<code>Files.isDirectory(dirPath)</code>
Test ob „Adresse“ eine existierende Datei ist	<code>dir.isFile()</code>	<code>Files.isRegularFile(filePath)</code>

Vergleich: Ein paar wichtige Funktionalitäten (2)

```
File dir =  
    new File("/tmp/a");  
File file =  
    new File("/tmp/b.txt");
```

```
Path dirPath =  
    Paths.get("/tmp/a");  
Path filePath =  
    Paths.get("/tmp/b.txt");
```

Was	File	Path
Existenz prinzipiell überprüfen	<code>dir.exists()</code> <code>file.exists()</code>	<code>Files.exists(dirPath)</code> <code>Files.exists(filePath)</code>
Umbenennen /verschieben	<code>file.renameTo(otherFile)</code>	<code>Files.move(filePath, otherFilePath)</code>
Löschen	<code>file.delete()</code> <code>dir.delete()</code>	<code>Files.delete(filePath)</code> <code>Files.delete(dirPath)</code>
Wenn ein Verzeichnis gelöscht wird, dürfen darin keine Unterordner oder Dateien mehr liegen.		
In absolute File-/Path-Instanzumwandeln	<code>file.getAbsoluteFile()</code> <code>dir.getAbsoluteFile()</code>	<code>filePath.toAbsolutePath()</code> <code>dirPath.toAbsolutePath()</code>

Vollständige Dokumentation:

<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/File.html>

Vollständige Dokumentation:

<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/file/Files.html>

Beispiel: Verzeichnis und Datei erzeugen (1)



```
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();
        }
    }
}
```

Beispiel: Verzeichnis und Datei erzeugen (2)



```
import java.io.IOException;
import java.nio.file.*;
```

```
public class CreateDirAndFile2 {
```

```
    public static void main(String args[]) {
```

```
        Path testDir = Paths.get("testDir");
```

```
        try {
```

```
            Files.createDirectory(testDir);
```

```
            Path testFile = testDir.resolve("testFile.txt");
```

```
            Files.createFile(testFile);
```

```
        } catch (IOException ex) {
```

```
            ex.printStackTrace();
```

```
        }
```

```
    }
```

```
}
```

**Auch Verzeichnis
anlegen** wirft im
Fehlerfall eine
IOException



ÜBUNG

INPUT/OUTPUT (1)

1. AUFGABE

Ein- und Ausgabe über Datenströme (1)

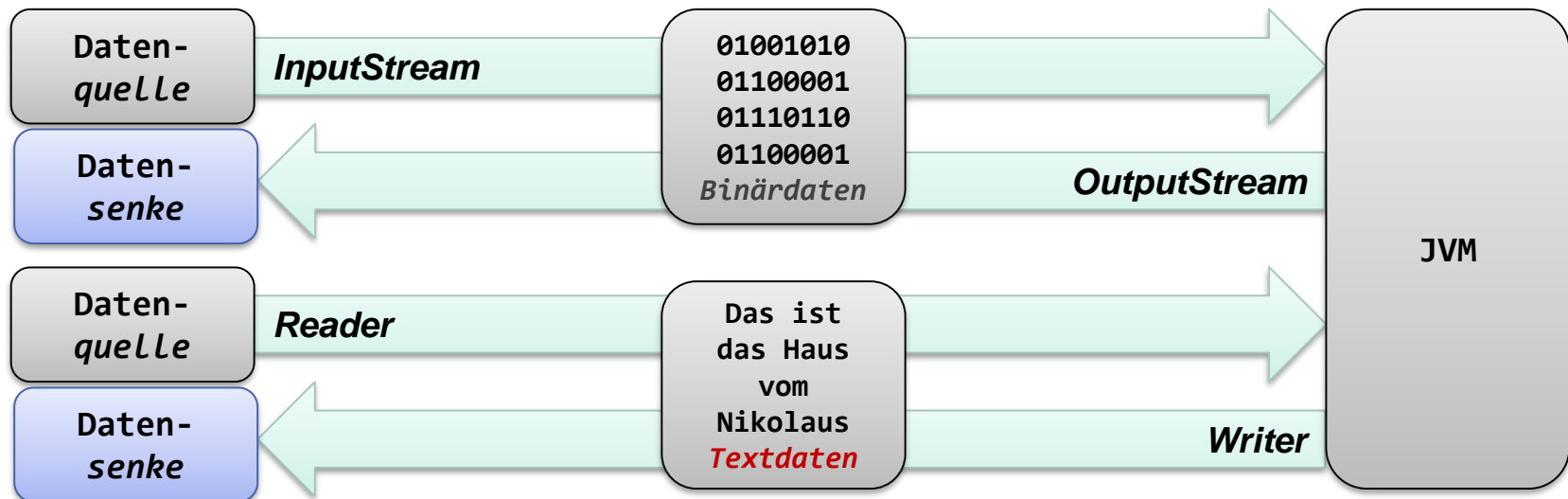
- Sämtliche Ein- und Ausgaben in Java laufen über (Daten-) **Ströme** (engl. Streams) ab.
- Ein (Daten-)Strom ist eine Verbindung zwischen einem Programm und einer **Datenquelle** bzw. einer **Datensenke** (Datenziel).
- Diese Verbindung (der Strom) läuft dabei stets nur in *einer Richtung* (uni-direktional).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

Ein- und Ausgabe über Datenströme (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 Ein- und Ausgabe-Ströme

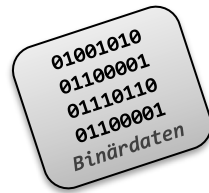
- Für Ein- und Ausgabe-Ströme hat Java vier *Basisklassen*:
InputStream und **OutputStream** (*Binärdaten*)
Reader und **Writer** (*Zeichen-orientierte Daten*)



- Diese vier Klassen sind direkte Unterklassen von **Object**
- Sie liegen im Paket **java.io**
→ aber auch von **NIO** verwendet!

Basisklasse InputStream – Einige Methoden

- **int read()**
liest *das nächste* Byte von diesem InputStream (und entnimmt es dabei). Liefert -1 bei Ende des Streams.
- **int read(byte[] b)**
int read(byte[] b, int off, int len)
liest *mehrere* Bytes vom InputStream (max. b.length viele) und speichert sie in dem Array b.
Optional: Offset und von b.length abweichende Menge
Rückgabewert: Anzahl der gelesenen Bytes
- **long skip(long n)**
überspringt n Bytes dieses InputStreams.
- **void close()**
InputStream schließen und Ressourcen freigeben

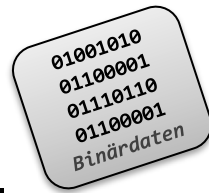


Nicht mehr
direkt
verwenden!

Vollständige Dokumentation: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/InputStream.html>

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)**
void write(byte[] b, int off, int len)
schreibt die Bytes aus Puffer b auf den OutputStream
Optional: Offset und von b.length abweichende Menge
- **void flush()**
„spült“ (leert) einen eventuellen Puffer des OutputStream-Objekts (durch die sofortige Abarbeitung aller noch anstehenden Bytes). Wichtig bei *gepufferten* Ausgaben!
- **void close()**
OutputStream schließen und Ressourcen freigeben



Nicht mehr
direkt
verwenden!

Vollständige Dokumentation: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/OutputStream.html>



Basisklasse Reader – Einige Methoden

- `int read()`
liest *das nächste Zeichen* von diesem Reader (und entnimmt es dabei). Liefert -1 bei Ende des Datenstroms.
- `int read(char[] b)`
`int read(char[] b, int off, int len)`
liest *mehrere Zeichen* vom Reader (max. `b.length` viele) und speichert sie in dem Array `b`.
Optional: Offset und von `b.length` abweichende Menge
Rückgabewert: Anzahl der gelesenen Zeichen
- `long skip(long n)`
überspringt `n` Zeichen dieses Readers.
- `void close()`
Reader schließen und Ressourcen freigeben

Nicht mehr
direkt
verwenden!

Vollständige Dokumentation: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/Reader.html>

Basisklasse Writer – Einige Methoden

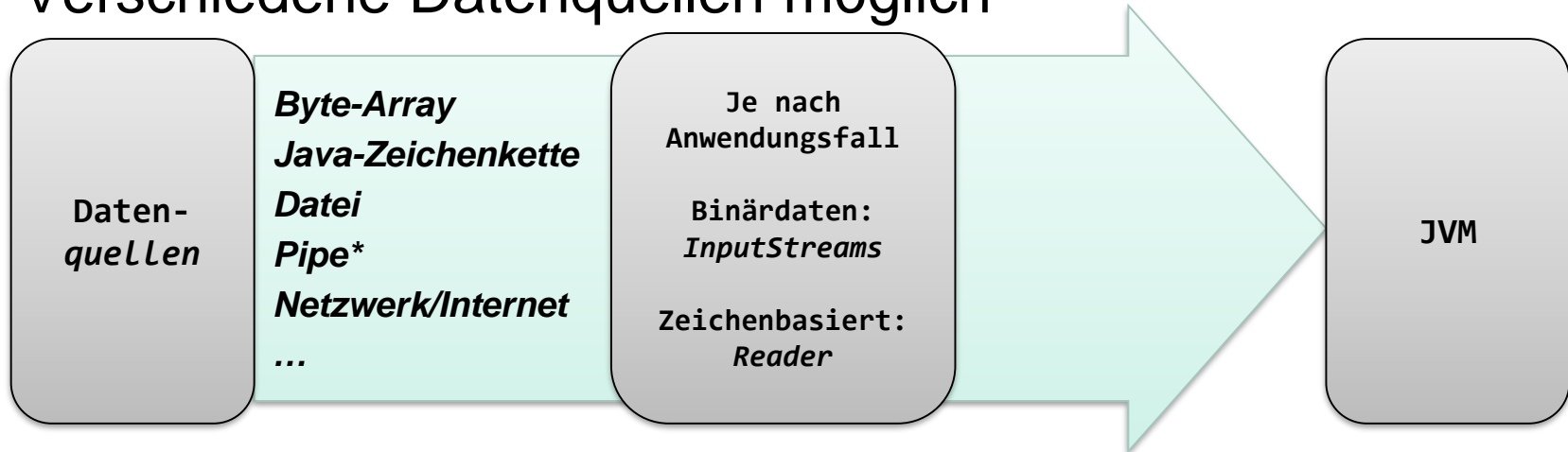
- `void write(char c)`
schreibt ein einzelnes *Zeichen* auf diesen Writer
- `void write(char[] b)`
`void write(char[] b, int off, int len)`
schreibt die Zeichen aus Puffer b auf den Writer
Optional: Offset und von b.length abweichende Menge
- `void write(String s)`
schreibt die Zeichenkette s auf den Writer
- `void flush()`
analog zu OutputStream
- `void close()`
Writer schließen und Ressourcen freigeben

Nicht mehr
direkt
verwenden!

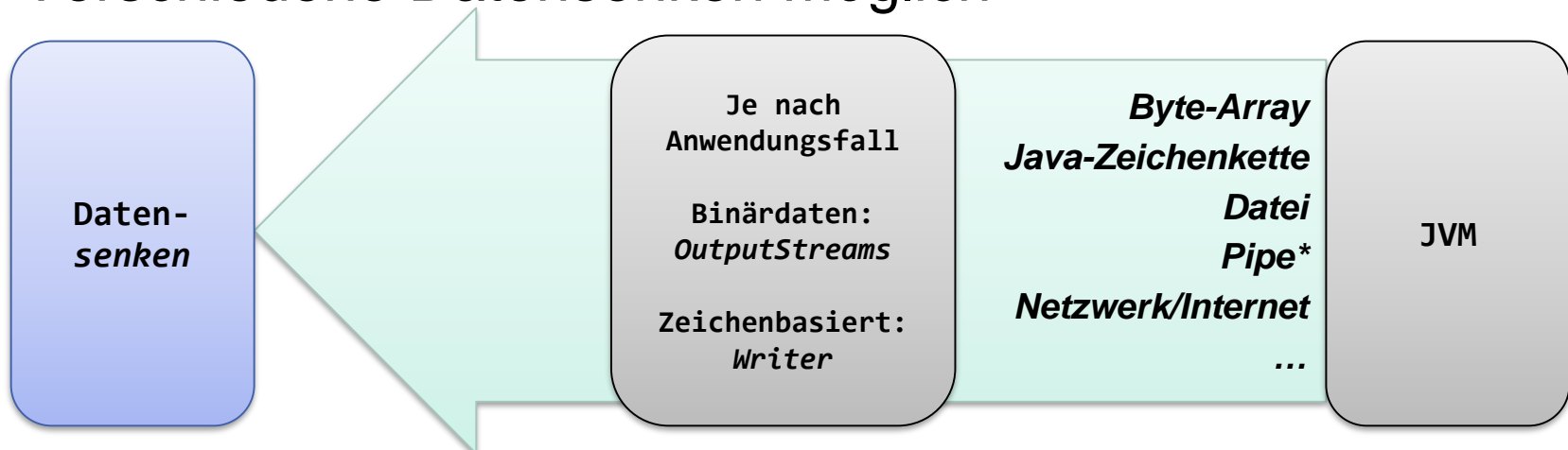
Vollständige Dokumentation: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/Writer.html>

Ein- und Ausgabetypen

■ Verschiedene Datenquellen möglich



■ Verschiedene Datensenken möglich



* Kommunikation zwischen Threads

Inputs und Outputs: Dateien

	Binäre Dateien (bspw. Bilder)	Textdateien
Datei lesen	FileInputStream	FileReader
Datei schreiben	FileOutputStream	FileWriter

- Besonderheit beim Schreiben: Entscheidung ob
 - die komplette Datei *überschrieben* werden soll
 - der neue Inhalt am Ende der Datei *angehängt* werden soll

IO

```
File f = new File("myfile.txt");
new FileWriter(f);           // Überschreiben
new FileWriter(f, true);    // Anhängen
```

NIO

```
Path p = Paths.get("myfile.txt");
Files.newBufferedWriter(p);           // Überschreiben
Files.newBufferedWriter(p,
    StandardOpenOption.APPEND); // Anhängen
```

Anmerkung: was BufferedWriter ist kommt später ☺

Exception-Handling & Schließen von Ressourcen

- Ressourcen **müssen** geschlossen werden, wenn sie nicht mehr gebraucht werden.
- 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:

IO

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



Ab Java 1.7

Beispiel: Erstes Zeichen einer Datei lesen



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

```
public class ReadFirstChar {
```

```
    public static void main(String args[]) {
```

```
        String testFile = "test.txt";
```

```
        // Anlegen des FileReaders zum Lesen von textbasierten Dateien
```

```
        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();
```

```
        }
```

```
    }
```

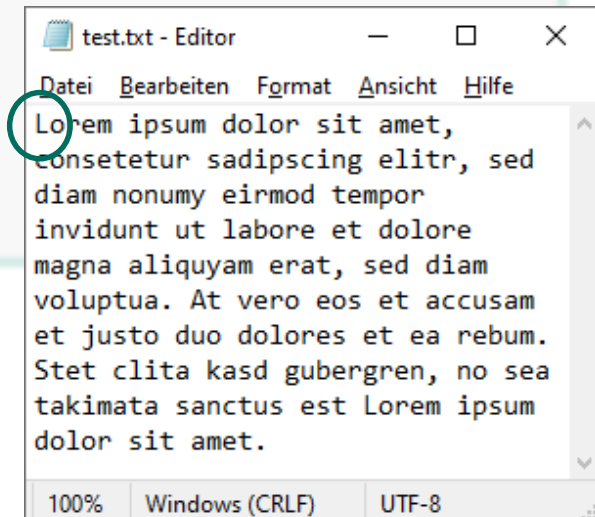
```
}
```



Eingeschränktes Beispiel zum „warm werden“. Die „richtigen“ Beispiele folgen später!

Ausgabe

Read: L



ÜBUNG

INPUT/OUTPUT (1)

2. AUFGABE

Anmerkungen zum Zeilentrenner (Carriage Return und Line Feed)

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

Unix/Linux/MacOS X	Windows
<code>\n</code> (LF)	<code>\r\n</code> (CRLF)

- Der Zeilentrenner des Systems auf dem das Java-Programm ausgeführt wird lässt sich zur Laufzeit auslesen:

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

oder

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

- Höherwertige Ein-/Ausgabeklassen kümmern sich darum *plattformunabhängig* automatisch

Anmerkung: Früher bei „klassischem“ MacOS: `\r` (CR)

Elementare und höherwertige I/O-Klassen (1)

- 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.
 - Automatische Behandlung des Zeilentrenners
 - gepufferte Ein-/Ausgabe.
 - die Ein-/Ausgabe primitiver Datentypen in ihrer Binärdarstellung (z.B. `long`- oder `double`-Daten)

Elementare und höherwertige I/O-Klassen (2)

- Bisherige Klassen mit elementaren Funktionen
 - FileInputStream / FileOutputStream
 - FileReader / FileWriter
- Klassen mit „höherwertigen“ Funktionen
 - PrintStream
 - Nur Schreiben!*
 - PrintWriter
 - BufferedInputStream/BufferedOutputStream
 - BufferedReader/BufferedWriter
- Typische Verkettung:

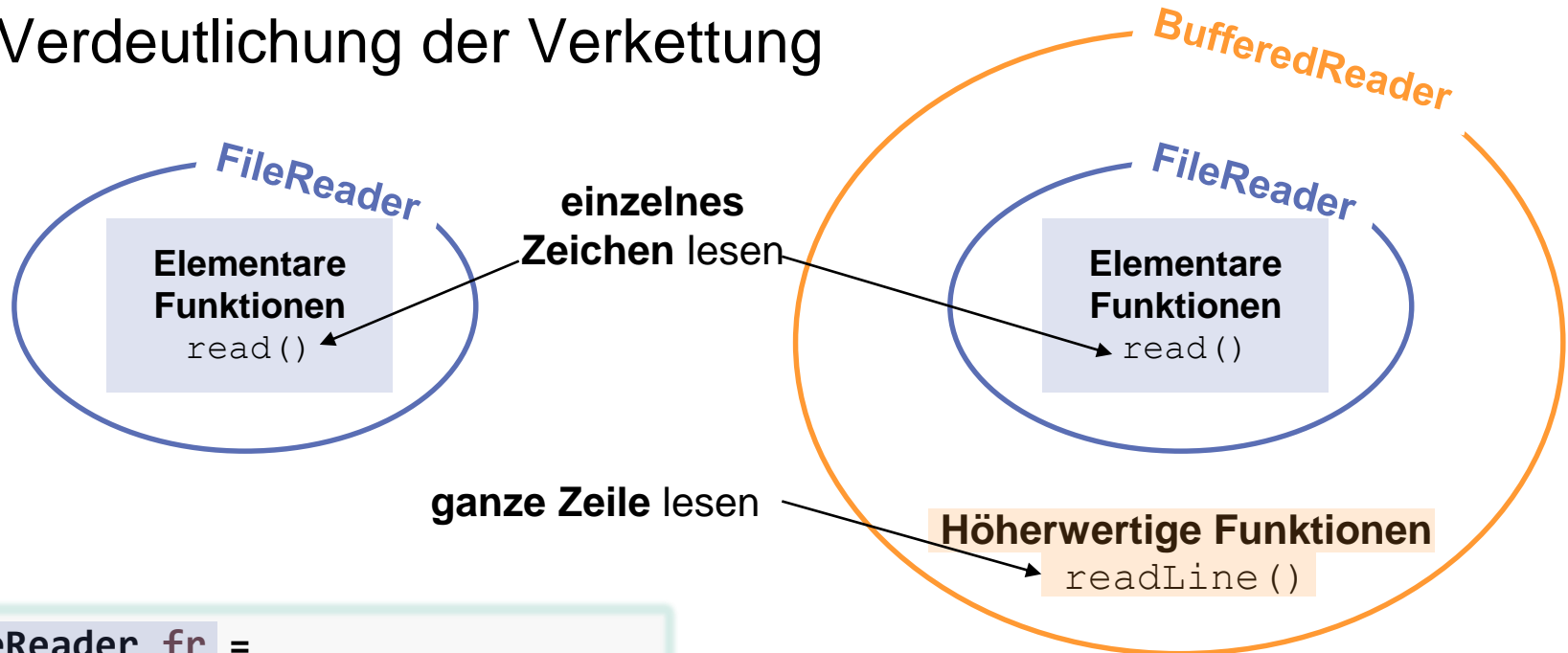
höherwertige
Methoden

Zusätzlich:
gepufferte
Ein-/Ausgabe

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

Elementare und höherwertige I/O-Klassen (3)

■ Verdeutlichung der Verkettung



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

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

Einige höherwertige Funktionen – `PrintWriter`

- Schwerpunkt: *Formatierung & zeichenorientierte Ausgabe*
 - `void print(type x)`
erzeugt dem Datentyp `type` entsprechende *String-Darstellung* für den Wert `x` und schreibt sie auf die Ausgabe
 - `void println()`
schreibt einen *Zeilenwechsel-String* auf die Ausgabe
 - `void println(type x)`
ruft `print(x)` und anschließend `println()` auf
 - `PrintWriter printf(String format, Object... args)`
schreibt eine *formatierte* Zeichenkette auf die Ausgabe
 - `PrintWriter append(CharSequence csq)`
schreibt die Zeichenfolge `csq` in den Strom

Vollständige Dokumentation: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/PrintWriter.html>

Einige höherw. Funktionen – BufferedWriter

■ Schwerpunkt: *gepufferte Ausgabe für bessere Performance*

■ `void newLine()`

schreibt den Zeilentrenner des Systems auf dem das Java-Programm aktuell läuft auf die Ausgabe

Vorteil gepufferter Ausgabe

- Wenn sehr viele Zeichen auf eine Datensenke geschrieben werden, kann dies ineffizient sein, jedes Zeichen in einem eigenen Vorgang zu schreiben → Für jedes einzelne Zeichen muss z.B. die entsprechende Verbindung mit dem Speichermedium bzw. der Netzwerkkumgebung separat auf- und wieder abgebaut werden.
- Die Klasse `BufferedWriter` ermöglicht es, eine ganze Reihe von vor dem eigentlichen Schreibvorgang auf das „echte“ Medium in einem internen Puffer zwischenzuspeichern.
- Dadurch können ganze Sequenzen von Zeichen zu größeren Blöcken zusammengefasst und in einem Vorgang geschrieben werden.

Vollständige Dokumentation: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/BufferedWriter.html>

Einige höherw. Funktionen – BufferedReader

- Schwerpunkt: *gepufferte Eingabe für bessere Performance*
 - `String readLine()`
ließt die Eingabe bis zum nächsten Zeilentrenner (unabhängig davon ob dies LF, CRLF oder CR ist!)
Rückgabewert `null` signalisiert Ende der Eingabe(-datei)
 - `boolean ready()`
wahr, wenn das nächste `read()` garantiert nicht wegen nicht vorhandener Daten wartet (blockiert)

Vorteil gepuffert Eingabe

Die Vorteile der gepufferten Ausgabe gelten in die „andere Richtung“ analog für die gepufferte Eingabe!

Es ist wesentlich effizienter ganze Blöcke von Zeichen in einen internen Puffer zu lesen als jedes Zeichen einzeln.

Vollständige Dokumentation: <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/BufferedReader.html>

Beispiele

- Es folgen nun einige Beispiele (endlich mehr Code!)
- Wo möglich gibt es jeweils 2 Varianten
 - Standard-IO
 - Alternative mit NIO
- Verstehen als Nachschlagewerk

Datei lesen mit BufferedReader (1)

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class BufferedReaderExampleIO {

    public static void main(String[] args) {
        File f = new File("test.txt");
        try (BufferedReader br = new BufferedReader(new FileReader(f))) {
            while (br.ready()) {
                String line = br.readLine();
                System.out.println("Line Read: " + line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Öffnen des
BufferedReader
mit Standard-IO

Datei lesen mit BufferedReader (2)

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class BufferedReaderExampleNIO {
    public static void main(String[] args) {
        Path p = Paths.get("test.txt");
        try (BufferedReader br = Files.newBufferedReader(p)){
            while ( br.ready() ) {
                String line = br.readLine();
                System.out.println("Line Read: " + line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Öffnen des
BufferedReader
mit NIO

Datei lesen mit Files-Methoden

```
import java.io.IOException; import java.nio.file.Files; import java.nio.file.Path; import java.nio.file.Paths; import java.util.List;

public class ReadAllExamplesNIO {
    public static void main(String[] args) {
        Path p = Paths.get("test.txt");

        // kein try-with-resources, Files.readXYZ-Methoden lesen komplette
        // Datei und kümmern sich um das Schliessen der Ressourcen
        try {
            // der komplette Inhalt der Textdatei als Zeichenkette
            String entireFileContent = Files.readString(p);

            // Jede Zeile als Eintrag in einer Liste
            // mehr zu Listen: 2. Semester :-)
            List<String> allLines = Files.readAllLines(p);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Komplette
Datei
einlesen

Datei schreiben mit BufferedWriter (1)

Komplette Datei überschreiben (IO)

```
import java.io.BufferedWriter; import java.io.File;
import java.io.FileWriter; import java.io.IOException;

public class BufferedWriterExampleIO {

    public static void main(String[] args) {
        File f = new File("test.txt");
        try (BufferedWriter bw =
            new BufferedWriter(new FileWriter(f))) {
            bw.write("Ich bin die erste Zeile");
            bw.newLine();
            bw.write("Ich bin die zweite Zeile");
            bw.newLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Öffnen des
BufferedWriter
mit Standard-IO

Wenn die Datei noch nicht
existiert wird sie automatisch
angelegt!

Datei schreiben mit BufferedWriter (2)

Inhalt an Datei anhängen (IO)

```
import java.io.BufferedWriter; import java.io.File;
import java.io.FileWriter; import java.io.IOException;

public class BufferedWriterExampleAppendIO {

    public static void main(String[] args) {
        File f = new File("test.txt");
        try (BufferedWriter bw =
            new BufferedWriter(new FileWriter(f, true))) {
            bw.write("Ich bin die erste Zeile die angehaengt wird");
            bw.newLine();
            bw.write("Ich bin die zweite Zeile die angehaengt wird");
            bw.newLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Öffnen des
BufferedWriter
mit Standard-IO

„append“-Flag

Wenn die Datei noch nicht
existiert wird sie ebenso
automatisch angelegt!

Datei schreiben mit BufferedWriter (4)

Komplette Datei überschreiben (NIO)

```
import java.io.BufferedWriter; import java.io.IOException;
import java.nio.file.Files; import java.nio.file.Path;
import java.nio.file.Paths;

public class BufferedWriterExampleNIO {

    public static void main(String[] args) {
        Path p = Paths.get("test.txt");
        try (BufferedWriter bw = Files.newBufferedWriter(p)) {
            bw.write("Ich bin die erste Zeile");
            bw.newLine();
            bw.write("Ich bin die zweite Zeile");
            bw.newLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Öffnen des
BufferedWriter
mit NIO

Wenn die Datei noch nicht
existiert wird sie automatisch
angelegt!

Datei schreiben mit BufferedWriter (3)

Inhalt an Datei anhängen (NIO)

```
import java.io.BufferedWriter; import java.io.IOException;
import java.nio.file.Files; import java.nio.file.Path;
import java.nio.file.Paths; import java.nio.file.StandardOpenOption;

public class BufferedWriterExampleAppendNIO {

    public static void main(String[] args) {
        Path p = Paths.get("test.txt");
        try (BufferedWriter bw =
            Files.newBufferedWriter(p, StandardOpenOption.APPEND)) {
            bw.write("Ich bin die erste Zeile die angehaengt wird");
            bw.newLine();
            bw.write("Ich bin die zweite Zeile die angehaengt wird");
            bw.newLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Öffnen des
BufferedWriter
mit NIO
„append“-Option

Wenn die Datei noch nicht
existiert wird sie ebenso
automatisch angelegt!


Datei schreiben mit PrintWriter

```
import java.io.File;  
import java.io.IOException;  
import java.io.PrintWriter;
```

```
public class PrintWriterExampleIO {
```

```
    public static void main(String[] args) {  
        try (PrintWriter pw = new PrintWriter("test.txt")) {  
            pw.println("Ich bin die erste Zeile");  
            pw.println("Ich bin die zweite Zeile");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Sonderfall: es kann direkt der Dateiname angegeben werden.



Datei wird komplett überschrieben

Historischer Vergleich: saubere Ressourcenbehandlung

```
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

 **So nicht mehr
verwenden!**

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

 Ab Java 1.7