

Datenverarbeitung

Teil des Moduls 5CS-DPDL-20

Prof. Dr. Deweß

Thema 9



(extern) gespeicherte Daten

- Serialisierung/Deserialisierung
- Input/Output-Streams
- Umgang mit Dateien und Verzeichnissen
- Ausblick auf die Nutzung von Daten aus Datenbanken

Serialisierung und Deserialisierung



Wie kommen Objekte wie
ich in einen Stream?

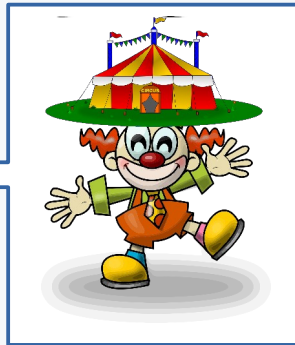
Und wie komme ich da
wieder heraus?

(ohne „kaputt“ zu gehen)

Serialisierung und Deserialisierung

Serialisierung in Java

Konvertierung eines Objektes
im aktuellen Zustand in einen
(Java-)Byte-Stream.

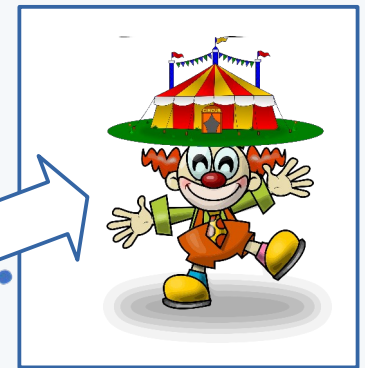


hier könnte der aktuelle
Zustand nun zwischen-
gespeichert werden



Deserialisierung in Java

Konvertierung eines (Java-)Byte-
Streams zurück in ein Objekt in
einem bestimmten Zustand



Serialisierung und Deserialisierung

Ein Beispiel

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

import de.baleipzig.classes.Circus;
import de.baleipzig.classes.Clown;

public class IOTestClass {
    public static void main(String[] args) {
        // Wir brauchen ein Objekt
        Clown myClown = new Clown("Kasper", 1, new Circus("JavaFun", 1));

        // Serialisierung und Output in eine Datei folgt gleich ...
        // Input aus einer Datei und Deserialisierung folgt danach ...
        // Sonstiges hilft anschließend aus ...
    }
}
```

Aufgabe 1a – Vorbereitung

Legen Sie bei sich eine entsprechende Testklasse an und sorgen Sie durch für Sie passende Importanweisungen dafür, dass Ihr „Clown“ aus der letzten Vorlesung darin benutzbar ist.

Serialisierung und Deserialisierung

Ein Beispiel

```
// ... ihre passenden Importanweisungen
public class IOTestClass {
    public static void main(String[] args) {
        // ... ihr Clown
        // Serialisierung und Datei-Output
        try {
            FileOutputStream fileOut = new FileOutputStream("clown.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(myClown);
            out.close();
            fileOut.close();
            System.out.println("Clown wurde erfolgreich serialisiert.");
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Input aus einer Datei und Deserialisierung folgt danach ...
        // Sonstiges hilft anschließend aus ...
    }
}
```

Aufgabe 1b

Implementieren Sie in Ihrer Testklasse den Teil zur Serialisierung. Ergänzen Sie dafür bei „Clown“ als implementierte Schnittstelle „Serializable“.

Serialisierung und Deserialisierung

Ein Beispiel

```
// Input aus Datei und Deserialisierung
try {
    FileInputStream fileIn = new FileInputStream("clown.ser");
    ObjectInputStream in = new ObjectInputStream(fileIn);
    Clown deserializedClown = (Clown) in.readObject();
    in.close();
    fileIn.close();
    System.out.println("Clown " + deserializedClown.getName()
        + " wurde erfolgreich deserialisiert.");
    System.out.println("Daten: " + deserializedClown);
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

Aufgabe 1c

Implementieren Sie in Ihrer Testklasse den Teil zur Deserialisierung.

Aufgabe 1d

Wenn Ihr Clown nun erfolgreich serialisiert, in eine Datei geschrieben, aus dieser gelesen und deserialisiert werden konnte, dann müsste es irgendwo bei Ihnen im Dateisystem die Datei „clown.ser“ geben. **Wo ist diese Datei?**

Serialisierung und Deserialisierung

Lösung 1d

Der Clown ist in Ihrem Arbeitsverzeichnis gelandet.

Wo sich dieses befindet, können Sie sich mit folgendem Quellcode in Ihrer Testklasse anzeigen lassen:

```
// Sonstiges hilft anschließend aus  
Path dir = Paths.get(System.getProperty("user.dir"));  
System.out.println("Arbeitsverzeichnis: " + dir);
```




I/O-Streams:

Schnittstelle zwischen Anwendung und „sonstwas“

Serialisierung/Deserialisierung von Objekten wird nicht in isolierten Streams, sondern im Rahmen von I/O-Streams betrachtet, wobei Streams allgemein eine Sequenz von Daten repräsentieren.

I/O-Streams

- sind ein Konzept zur Vereinfachung von Input-/Output-Operationen (Eingabe-/Ausgabe-Operationen)
- Input-Streams: repräsentieren eine Eingangsquelle
- Output-Streams: repräsentieren eine Ausgangssenke
- können verschiedene Arten von Quellen und Senken repräsentieren, z.B.:
 - Dateien
 - Ein-/Ausgabegeräte
 - andere Programme, ...
- einige transferieren nur Daten, andere können auch Daten bearbeiten

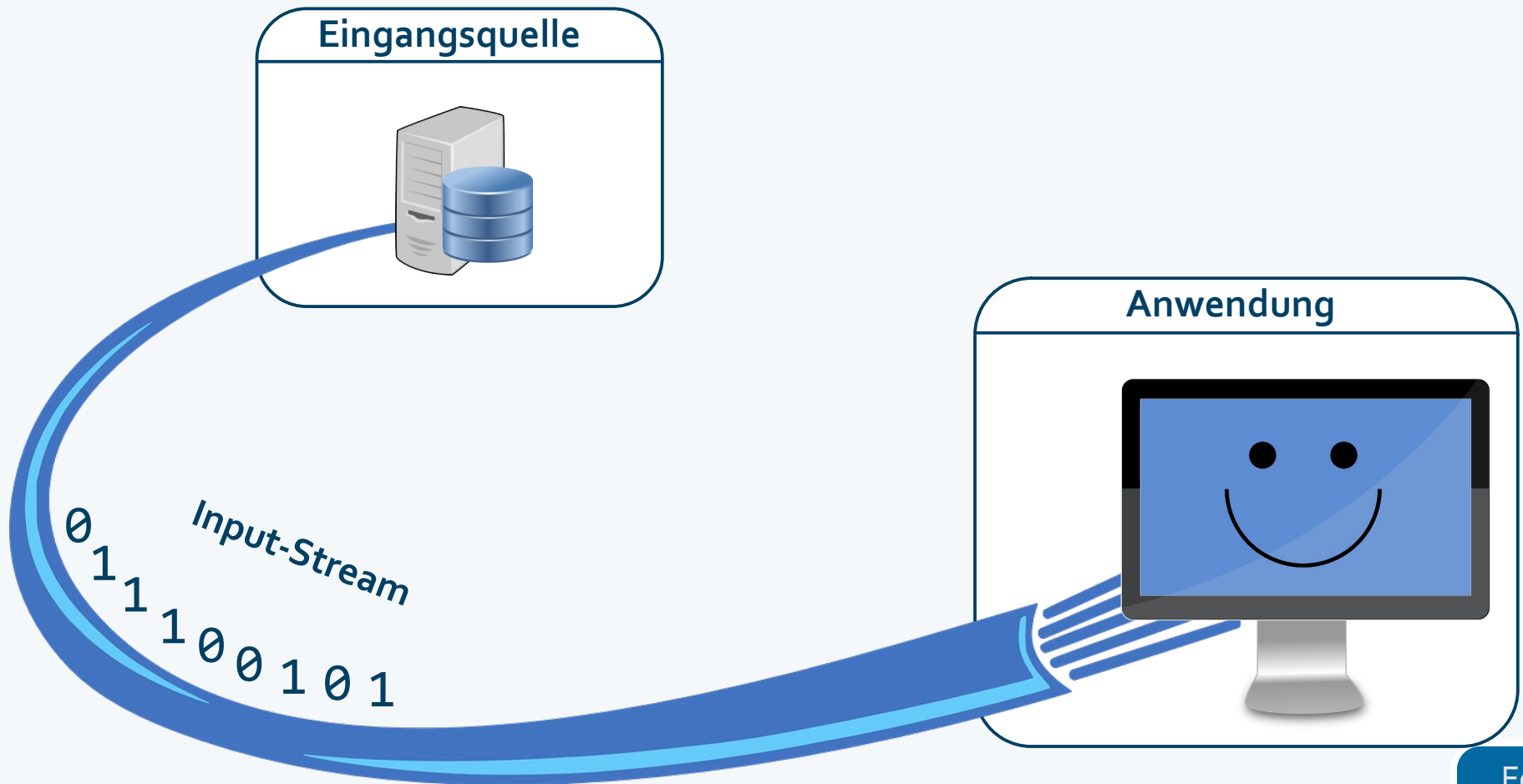


(extern) gespeicherte Daten

- Serialisierung/Deserialisierung
- **Input/Output-Streams**
- Umgang mit Dateien und Verzeichnissen
- Ausblick auf die Nutzung von Daten aus Datenbanken

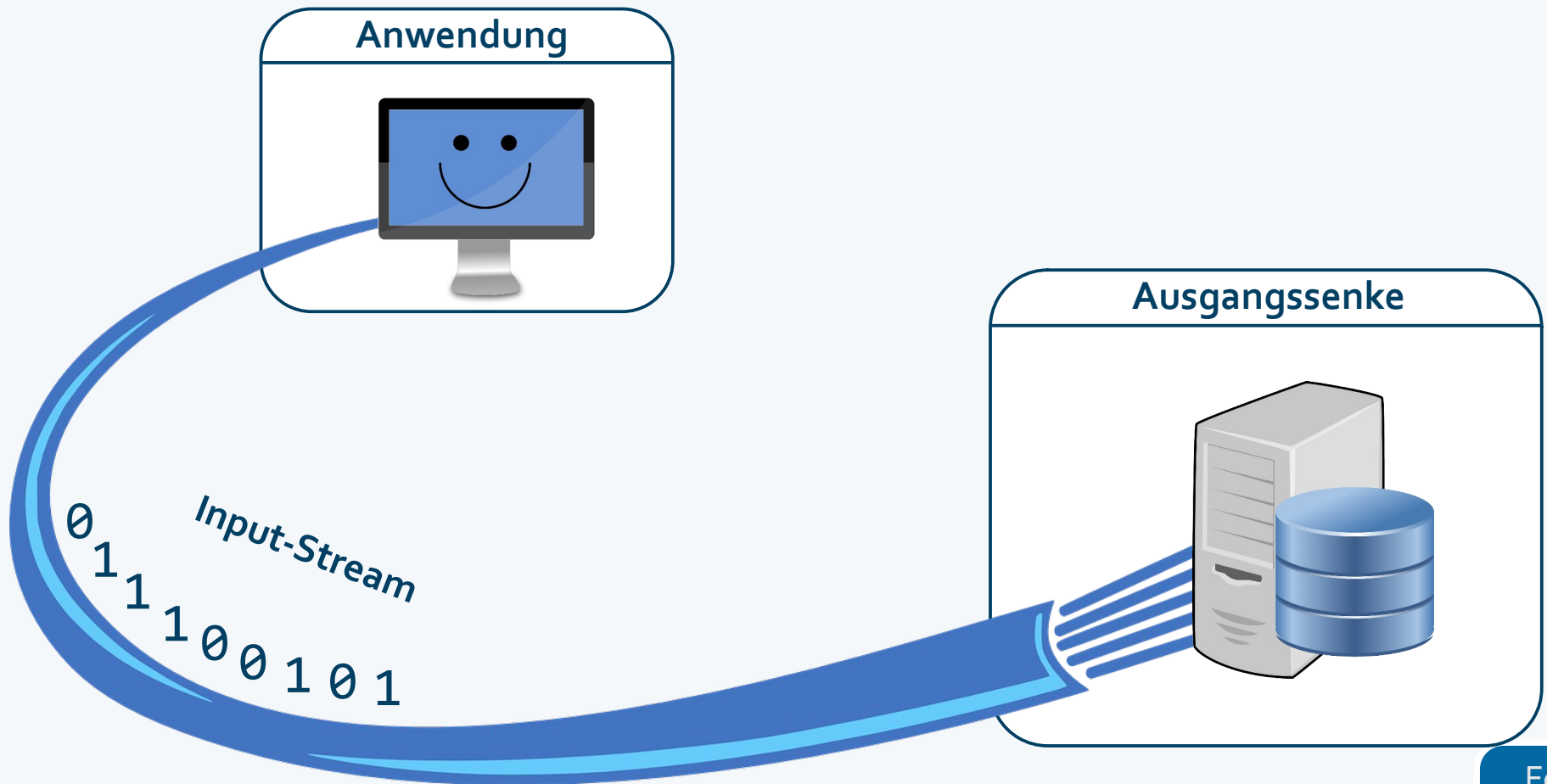
I/O-Streams: Input-Streams

Daten werden – eine Einheit nach der nächsten – von einer Eingangsquelle gelesen:



I/O-Streams: Output-Streams

Daten werden – eine Einheit nach der nächsten – auf eine Ausgangssenke geschrieben.



I/O-Streams: I/O-Stream-Arten

- „Byte Stream“
 - „Character Stream“
- } grundlegende Streamunterscheidung
- „Data Stream“
 - „Object Stream“
- } Streamunterscheidung auf höherer Ebene
- „Buffered Streams“ → Unterstützung einer Zwischenspeicherung (gepufferte Streams)
 - „Standard Streams“ → spezielle Streams als Eigenschaft vieler Betriebssysteme
 - **zur Unterstützung von Streams:**
 - Scanner API (Objekte vom Typ „Scanner“): Eingaben in einzelne Token zerlegen
 - Formatting API (Methoden von „PrintStream“ und „PrintWriter“, z.B. „print“, „printf“, „println“, „format“): Daten gut lesbar formatiert zusammenfassen/präsentieren

I/O-Streams: „Byte Stream“ und „Character Stream“

„Byte Stream“ (InputStream, OutputStream)

- Stream zur Ein- bzw. Ausgabe von 8-Bit-Bytes
- Daten werden als Sequenz derartiger Bytes behandelt
- Verwendung hauptsächlich zur Übertragung binärer Nicht-Text-Daten (Bilder-, Audio-, Video-dateien) ohne Kodierungsschema
- funktionieren auf niedrigem, primitivem Level und sind vor allem wichtig, weil alle anderen Streams darauf basieren
- **Beispiel:** „FileInputStream“ und „FileOutputStream“

„Character Stream“ (Reader, Writer)

- Stream zur Ein- bzw. Ausgabe von Zeichenketten mit Kodierungsschema (ASCII, UNICODE, ...)
- Daten werden als Sequenz von Unicode-Zeichen behandelt, die automatisch passend zum lokalen Zeichensatz umgewandelt werden
- Verwendung hauptsächlich für die Übertragung von Textdaten
- **Beispiel:** „FileReader“ und „FileWriter“

benutzen auf unterster Ebene aber meist einen „Byte Stream“, „FileReader“ nutzt z.B. „FileInputStream“

I/O-Streams: Tür auf, Tür zu!

Wenn man einen Stream öffnet, ist es wichtig, diesen auch zu schließen!

- geöffnete, aber nicht geschlossene Streams können schwerwiegende Fehler (Ressourcenlecks und auch damit verbundene Sicherheitsrisiken) auslösen
- deshalb sollte man am besten eine „try-with-resources“-Anweisung nutzen (alte Variante: in einem try-Block selbst geöffnete Streams in einem finally-Block schließen, damit auch im Fehlerfall geöffnete Streams geschlossen werden)
- möglicher Fehlerfall: man möchte mehrere Streams öffnen, wobei das Öffnen eines Streams fehlschlägt

Aufgabe 2 (erst alte Variante vor Java SE 7, um Verbesserungspotential zu sehen)

Wandeln Sie den Quellcode zur Serialisierung aus Aufgabe 1 so ab, dass:

- die Referenzvariablen für beide Streams vor dem try-Block deklariert und mit „null“ initialisiert werden,
- die beiden Streams, sofern das Öffnen jeweils fehlerfrei funktioniert hat (nicht mehr Referenz auf „null“), sicherheitshalber beide in einem finally-Block geschlossen werden und
- werfen Sie möglicherweise dort auftretende IOExceptions weiter.

I/O-Streams: Tür auf, Tür zu!

Lösung 2

```
public static void main(String[] args) throws IOException {  
    // ...  
    FileOutputStream fileOut = null;  
    ObjectOutputStream out = null;  
    try {  
        fileOut = new FileOutputStream("clown.ser");  
        out = new ObjectOutputStream(fileOut);  
        // ...  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        if (out != null) {  
            out.close();  
        }  
        if (fileOut != null) {  
            fileOut.close();  
        }  
    }  
    // Variante für Eingabeströme (Input-Streams) folgt gleich ...  
}
```


I/O-Streams: Tür auf, Tür zu!

Lösung 2

```
public static void main(String[] args) throws IOException {  
    // ...  
    FileOutputStream fileOut = null;  
    ObjectOutputStream out = null;  
    try {  
        fileOut = new FileOutputStream("clown.ser");  
        out = new ObjectOutputStream(fileOut);  
        // ...  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        if (out != null) {  
            out.close();  
        }  
        if (fileOut != null) {  
            fileOut.close();  
        }  
    }  
    // Variante für Eingabeströme (Input-Streams) folgt gleich ...  
}
```

jetzt könnte hier aber auch noch ein Problem auftreten und eine Ausnahme geworfen werden ...

... und dann haben wir ggf. hier ein ernstes Problem und ein Leak

I/O-Streams: Tür auf, Tür zu!

„Try-with-Ressources“-Anweisung

- erlaubt eine „try-Anweisung“ mit Angabe einer Ressource, wobei die Ressource ein Objekt ist, welches nach Benutzung geschlossen werden muss
- Objekt muss entsprechend „java.lang.AutoCloseable“ (evtl. auch indirekt z.B. über „java.io.Closeable“) implementieren
- jede in einer „try-with-Ressources“-Anweisung dort als Ressource geöffnete Ressource wird geschlossen
- ggf. werden dann weder „catch“- noch „finally“-Block benötigt
- Syntax:
statt `try { ... hier stehen die Ressourcenanweisungen und der Rest ... }`
`try(... hier stehen die Anweisungen mit den Ressourcen ...) { ... Rest ... }`

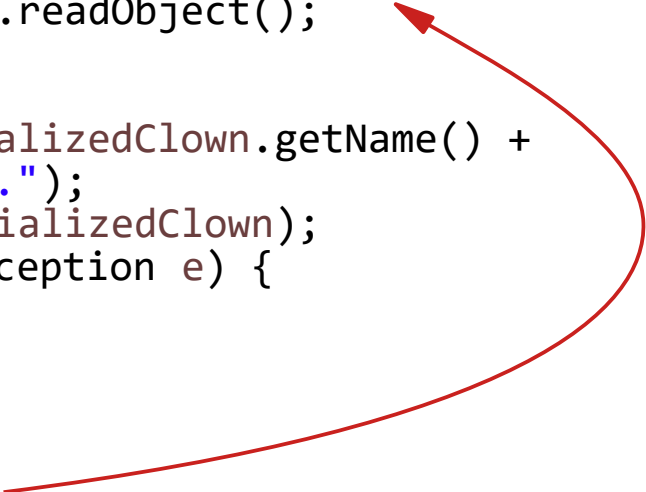
Aufgabe 3 (neue Variante seit Java SE 7)

Wandeln Sie den Quellcode zur Deserialisierung aus Aufgabe 1 so ab, dass die Ressourcen durch eine „try-with-Ressources“-Anweisung abgesichert sind.

I/O-Streams: Tür auf, Tür zu!

Lösung 3

```
public static void main(String[] args) throws IOException {  
    // ... und alte, unzureichende Schutzvariante  
    // Deserialisierung mit neuem, besserem Schutz  
    try (FileInputStream fileIn = new FileInputStream("clown.ser");  
        ObjectInputStream in = new ObjectInputStream(fileIn)) {  
        Clown deserializedClown = (Clown) in.readObject();  
        in.close();  
        fileIn.close();  
        System.out.println("Clown " + deserializedClown.getName() +  
            " wurde erfolgreich deserialisiert.");  
        System.out.println("Daten: " + deserializedClown);  
    } catch (IOException | ClassNotFoundException e) {  
        e.printStackTrace();  
    }  
    // ...  
}
```



Anmerkung:

Ressourcen werden dort geöffnet und dort auftretende Ausnahmen werden, wenn auch im try-Block Aufnahmen auftreten, unterdrückt; an diese käme man dann nur über die „Throwable.getSuppressed“-Methode der weitergeworfenen Ausnahme vom try-Block heran

I/O-Streams: „Character Stream“ und Tür auf, Tür zu!

Aufgabe 4a

- deklarieren und initialisieren Sie (mit einem Text ihrer Wahl) in der „main“-Methode ihrer Testklasse eine Variable vom Typ „String“
- benutzen Sie eine „try-with-Resources“-Anweisung, legen Sie als Ressource eine neue Referenz vom Typ „FileWriter“ fest und initialisieren Sie diese mit „`new FileWriter("FileWriterTest.txt")`“
- nutzen Sie die Methode „write“ aus der von „FileWriter“ indirekt erweiterten Klasse „java.io.Writer“, um den obigen String in die Datei „FileWriterTest.txt“ zu schreiben

Aufgabe 4b

Wir wollen nun die Datei „FileWriterTest.txt“ noch ergänzen.

Wiederholen Sie im Prinzip Aufgabe 4a, aber benutzen Sie zum Initialisieren der Ressource stattdessen einen Konstruktor aus „FileWriter“, der es Ihnen ermöglicht, „FileWriterTest.txt“ zu erweitern anstatt zu überschreiben.

Die für beide Aufgaben notwendige Dokumentation finden Sie unter:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/io/Writer.html> bzw.

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/io/FileWriter.html>

I/O-Streams: Tür auf, Tür zu!

Lösung 4a und 4b

```
public static void main(String[] args) throws IOException {  
    // ...  
  
    // Character Stream  
    String fileWriterText = "Mehr als ein Clown. ";  
  
    // Loesung 4a  
    try (FileWriter fileWriter = new FileWriter("FileWriterTest.txt")) {  
        fileWriter.write(fileWriterText );  
    }  
  
    // Loesung 4b  
    try (FileWriter fileWriter = new FileWriter("FileWriterTest.txt", true)) {  
        fileWriter.write(fileWriterText );  
    }  
}
```

I/O-Streams: „Wrapper“ (Streams einbetten)

- man kann (bzw. muss manchmal) Streams in weitere Streams einbetten („Wrapping“)
- **Sinn:** Hinzufügen weiterer Funktionalitäten



← statt jeden Spaghetti einzeln zu kochen, bietet es sich an, ganz funktional die Spaghetti vor dem Kochen zu „wrappen“ und so die Wartezeit bis zum fertigen Mahl signifikant zu verkürzen

- **übliche Szenarien:**
 - ungepufferten Stream in gepufferten Stream einpacken, z.B. „java.io.BufferedReader“ um einen „java.io.Reader“ nutzen
 - Hinzufügen von Komprimierungs- und Verschlüsselungsfunktionen zu einem unkomprimierten bzw. unverschlüsseltem Stream, z.B. „java.util.zip.GZIPInputStream“ um einen „java.io.InputStream“ nutzen
 - Hinzufügen von weiteren Funktionen (insb. auch Filter-Funktionen)

I/O-Streams: Einige „Wrapper“

Spezielle Byte-Streams - „Data Stream“ und „Object Stream“

„Data Stream“ (DataInput, DataOutput)

- Schnittstellen für einen Stream zur binären Ein- bzw. Ausgabe von Werten primitiver Datentypen und von String-Werten **von bzw. zu einem „Byte-Stream“**
- arbeiten nur als „Wrapper“ und benötigen daher einen eingebetteten Stream
- **Beispiele für implementierende Klassen:** „DataInputStream“ und „DataOutputStream“

„Object Stream“ (ObjectInput, ObjectOutputStream)

- Sub-Schnittstellen von „DataInput“ und „DataOutput“
- analog zu den „Data Streams“ Schnittstellen zur binären Ein- bzw. Ausgabe von Objekten
- Objekte müssen **serialisierbar** sein (Klasse muss Markierungsschnittstelle „Serializable“ implementieren) *[meist einfach erfüllbar]*
- **implementierende Klassen:** „ObjectInputStream“ und „ObjectOutputStream“



I/O-Streams: Einige „Wrapper“

Übung: „Data Stream“

Aufgabe 5a

Wir wollen primitive Daten schreiben. Vervollständigen Sie nachfolgenden Quellcode, damit die Gipfelnamen und Höhen in eine Datei geschrieben werden.

```
// Write Data Stream
int[] height = { 2962, 2874, 2750 };
String[] peak = {
    "Zugspitze",
    "Schneefernkopf",
    "Mittlere Wetter Spitze"
};

try (DataOutputStream dataOutputStream =
    new DataOutputStream(new FileOutputStream("peak.data"))) {
    for (int i = 0; i < height.length; i++) {
        dataOutputStream.writeInt( );
        dataOutputStream.writeUTF( );
    }
}
```


I/O-Streams: Einige „Wrapper“

Übung: „Data Stream“

Lösung 5a

```
// Write Data Stream
int[] height = { 2962, 2874, 2750 };
String[] peak = {
    "Zugspitze",
    "Schneefernkopf",
    "Mittlere Wetter Spitze"
};

try (DataOutputStream dataOutputStream =
    new DataOutputStream(new FileOutputStream("peak.data"))) {
    for (int i = 0; i < height.length; i++) {
        dataOutputStream.writeInt( height[i] );
        dataOutputStream.writeUTF( peak[i] );
    }
}
```

I/O-Streams: Einige „Wrapper“

Übung: „Data Stream“

Aufgabe 5b

Wir wollen primitive Daten lesen. Vervollständigen Sie nachfolgenden Quellcode, damit die eingelesenen Gipfelnamen und Höhen angezeigt werden.

```
// Read Data Stream
int readHeight;
String readPeak;

try (DataInputStream dataInputStream =
    new DataInputStream(new FileInputStream("peak.data"))) {
    while (true) {
        readHeight = dataInputStream.[REDACTED]
        readPeak = dataInputStream.[REDACTED]
        System.out.println(readPeak + ": " + readHeight);
    }
} catch (EOFException e) {
}
```

Wie wird eigentlich die „unendliche“ Schleife zum Einlesen beendet?

I/O-Streams: Einige „Wrapper“

Übung: „Data Stream“

Lösung 5b

```
// Read Data Stream
int readHeight;
String readPeak;

try (DataInputStream dataInputStream =
    new DataInputStream(new FileInputStream("peak.data"))) {
    while (true) {
        readHeight = dataInputStream.readInt();
        readPeak = dataInputStream.readUTF();
        System.out.println(readPeak + ": " + readHeight);
    }
} catch (EOFException e) {
}
```

Implementierungen von „DataInput“ nutzen zum Erkennen des Dateiendes nicht irgendwelche Rückgabewerte, sondern nutzen aus, dass in dem Fall eine „EOFException“ geworfen wird.

Auf diese Weise wird die scheinbar „unendliche“ Schleife verlassen.



I/O-Streams: Einige „Wrapper“

Übung: „Object Stream“

Aufgabe 6

Wir wollen uns anschauen, was passiert, wenn man ein Objekt mehrmals über einen „Object Stream“ deserialisiert (erinnern Sie sich an „clone()“?).

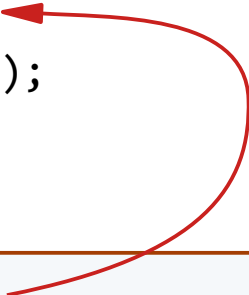
- Deserialisieren Sie den in Aufgabe 1 in Datei „clown.ser“ serialisierten „Clown“ analog zur mit Aufgabe 3 verbesserten Aufgabe 1c in eine neue Referenzvariable „doppelgangerClown“.
- Ändern Sie für „doppelgangerClown“ den Namen des zugehörigen „Circus“.
- Lassen Sie sich sowohl den Clown „deserializedClown“ als auch den Clown „doppelgangerClown“ wie üblich auf der Konsole anzeigen. Beachten Sie, dass Sie dafür am Quellcode zum „deserializedClown“ eine kleine Veränderung (z.B. Vorziehen der Deklaration und zunächst Initialisierung mit „null“) vornehmen müssen, damit dieser Clown an dieser Stelle noch verfügbar ist.

I/O-Streams: Einige „Wrapper“

Übung: „Object Stream“

Lösung 6

```
Clown deserializedClown = null;
// Deserialisierung
try (FileInputStream fileIn = new FileInputStream("clown.ser");
     ObjectInputStream in = new ObjectInputStream(fileIn)) {
    deserializedClown = (Clown) in.readObject();
    // ...
}
// ...
try (FileInputStream fileIn = new FileInputStream("clown.ser");
     ObjectInputStream in = new ObjectInputStream(fileIn)) {
    Clown doppelgangerClown = (Clown) in.readObject();
    doppelgangerClown.getCircus().setName("NewJavaFun");
    System.out.println("Daten: " + deserializedClown);
    System.out.println("Daten Kopie: " + doppelgangerClown);
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```



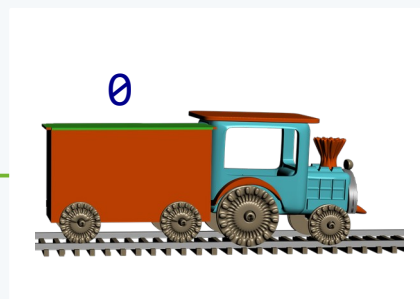
neue Deserialisierung liefert neues Objekt; Serialisierung und Deserialisierung bieten also eine Möglichkeit für „tiefe“ Kopien

I/O-Streams: „Buffered Streams“ als „Wrapper“

bisher: „ungepufferte“ Streams („raw streams“):

- jede einzelne Lese- oder Schreibanfrage (inklusive Anfrage für Festplattenzugriff, Netzwerkaktivität, ...) wird direkt vom darunterliegenden Betriebssystem bearbeitet
- sehr speichersparsam
- führt meist zu Ineffizienzen (eigentlich will man einen Satz auf die Festplatte schreiben und nicht tausendmal ein Zeichen)
- „InputStream“, „OutputStream“, „Reader“ und „Writer“ sind ungepufferte Streams

011
110
111



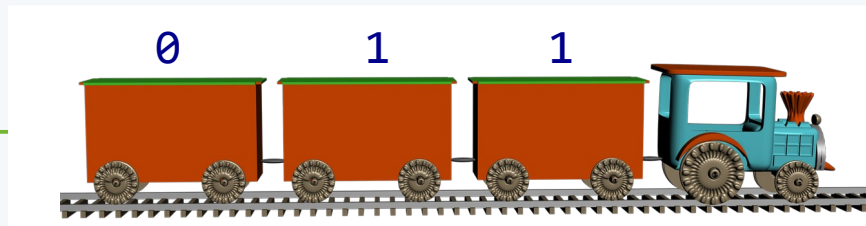
011
110
111

I/O-Streams: „Buffered Streams“ als „Wrapper“

„gepufferte“ Streams („buffered streams“)

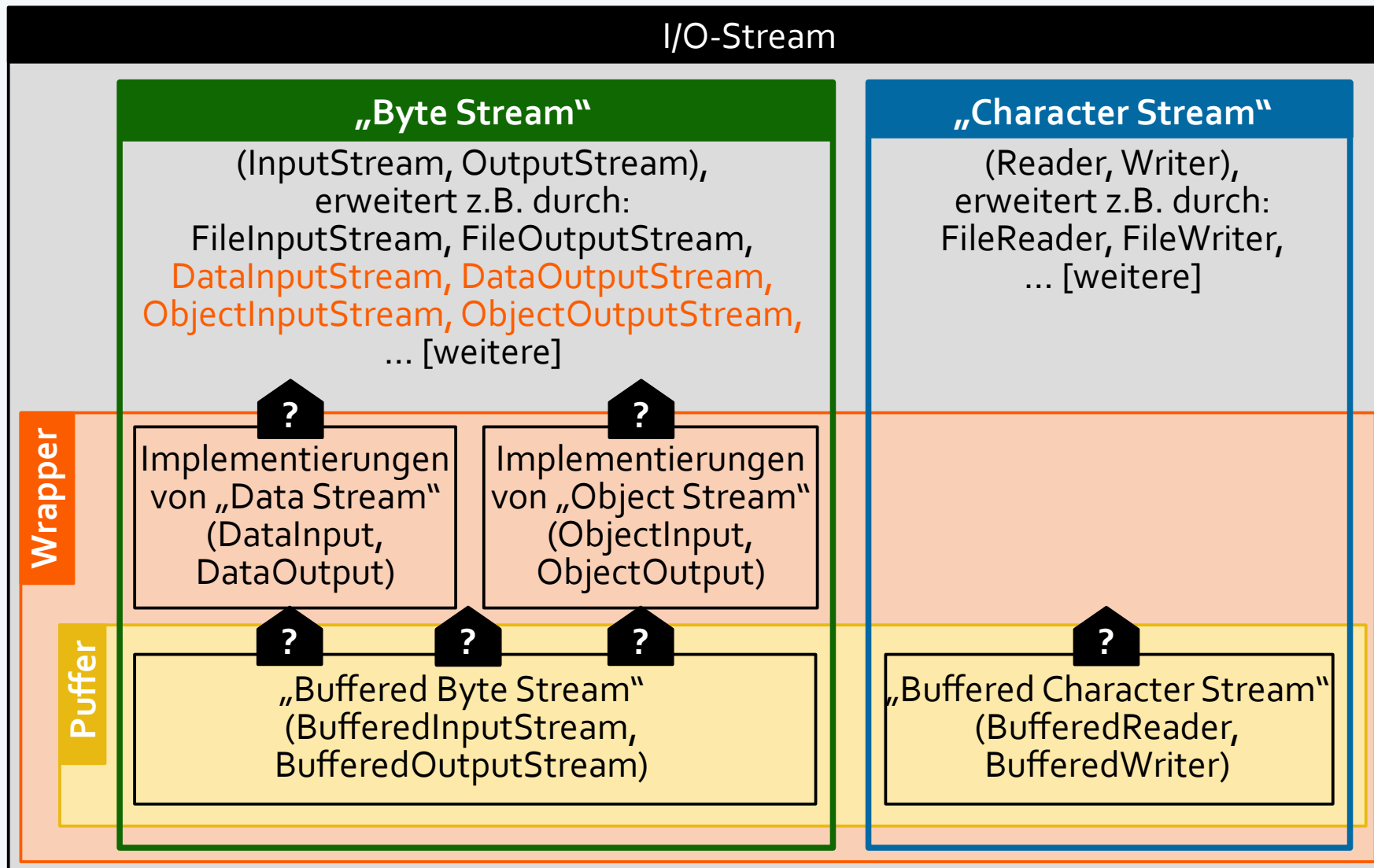
- fügen zwischen Datenquelle/-senke und Stream zusätzlichen Puffer zur Zwischenspeicherung von Daten hinzu
- damit werden Operationen wie Speicher- und Netzwerkzugriffe reduziert, da mehrere Daten auf einmal geschrieben/gelesen werden können:
(also z.B. zeilenweise Verarbeitung statt zeichenweise bei einer Textdatei)
 - Daten gehen beim Schreiben in den Puffer und werden erst von der zugrundeliegenden Schnittstelle geschrieben, wenn dieser voll ist
 - Daten werden beim Lesen aus dem Puffer gelesen und dieser wird über die zugrundeliegende Schnittstelle erst wieder aufgefüllt, wenn der Puffer leer ist
- „ungepufferte“ Streams können über „Wrapper“-Streams in „gepufferte“ umgewandelt werden, wobei es vier derartige Klassen gibt:
„BufferedInputStream“ und „BufferedOutputStream“ für „Byte Streams“ sowie „BufferedReader“ und „BufferedWriter“ für „Character Streams“

011
110
111



011
110
111

I/O-Streams: Was haben wir bis jetzt?



I/O-Streams: „Buffered Streams“ am Beispiel

Aufgabe 7

Prinzipiell könnten wir unsere Datei „FileWriterTest.txt“ aus Aufgabe 4 wie folgt mit einem Puffer („Buffered Stream“) dazwischen einlesen:

```
try (FileInputStream fileInputStream = new FileInputStream("FileWriterTest.txt");
    BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream)) {
    int r = bufferedInputStream.read();
    while (r != -1) {
        System.out.print((char)r);
        r = bufferedInputStream.read();
    }
} catch (IOException e) {
    // ... hier muesste man was tun; interessiert und aber in der Uebung jetzt nicht
}
```

beachte: „read“ liefert „int“, „readLine“ liefert etwas anderes

Wir haben aber beim Schreiben sinnvoller Weise (Textdatei) einen „Character Stream“ benutzt. Versuchen Sie dementsprechend, zum Lesen auch einen gepufferten „Character Stream“ zu benutzen. Nutzen Sie dann sinnvoller Weise statt „read“ auch gleich die Methode „readLine“ des gepufferten Streams.

Zusatz: Nutzen Sie die Methode „lines“ und verarbeiten Sie den so erhaltenen Stream.

I/O-Streams: „Buffered Streams“ am Beispiel

Lösung 7

```
try (FileReader fileReader = new FileReader("FileWriterTest.txt");
    BufferedReader bufferedReader = new BufferedReader(fileReader)) {
    String line = bufferedReader.readLine();
    while (line != null) {
        System.out.println(line);
        line = bufferedReader.readLine();
    }
} catch (IOException e) {
    // ... hier muesste man was tun
}
```

*beachte: „readLine“
liefert „String“*

Lösung Zusatzaufgabe:

```
try (FileReader fileReader = new FileReader("FileWriterTest.txt");
    BufferedReader bufferedReader = new BufferedReader(fileReader)) {
    System.out.println(bufferedReader.lines()
        .collect(Collectors.joining(System.lineSeparator())));
} catch (IOException e) {
    // ... hier muesste man was tun
}
```

I/O-Streams: „Buffered“ mal anders ... (nicht klausurrelevant)

Aufgabe 8

```
try (FileInputStream fileInputStream = new FileInputStream("blumen.png");
    FileOutputStream fileOutputStream = new FileOutputStream("bunte_blumen.jpg")) {
    BufferedImage image = ImageIO.read(fileInputStream);
    BufferedImage convertedImage = new BufferedImage(image.getWidth(),
    image.getHeight(), BufferedImage.TYPE_INT_RGB);
    convertedImage.createGraphics().drawImage(image, 0, 0, Color.WHITE, null);
    boolean canWrite = ImageIO.write(convertedImage, "jpg", fileOutputStream);
    if (!canWrite) {
        throw new IllegalStateException("Failed to write image.");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

„FileInputStream“ benötigt man nicht für Text-Dateien, aber Java „kann auch Bilder“. Versuchen Sie, das Bild „blumen.png“ in ein Bild „graue_blumen.jpg“ nur mit Grautönen zu konvertieren, indem Sie eine andere Konstante statt „TYPE_INT_RGB“ von „BufferedImage“ nutzen.

I/O-Streams: „Buffered“ mal anders ... (nicht klausurrelevant)

Lösung 8

```
try (FileInputStream fileInputStream = new FileInputStream("blumen.png");
    FileOutputStream fileOutputStream = new FileOutputStream("graue_blumen.jpg")) {
    BufferedImage image = ImageIO.read(fileInputStream);
    BufferedImage convertedImage = new BufferedImage(image.getWidth(),
        image.getHeight(), BufferedImage.TYPE_BYTE_GRAY);
    convertedImage.createGraphics().drawImage(image, 0, 0, Color.WHITE, null);
    boolean canWrite = ImageIO.write(convertedImage, "jpg", fileOutputStream);
    if (!canWrite) {
        throw new IllegalStateException("Failed to write image.");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

I/O-Streams: „Buffered Streams“ – „Flushing“

wenn Daten dringend gebraucht werden

(nicht klausurrelevant):

- man kann mit der Methode „flush“ den Puffer eines Streams vorzeitig verarbeiten, bevor dieser voll ist
- ist grundsätzlich auf jeden „Stream“ anwendbar, wirkt aber nur bei gepufferten Streams (*sonst gibt es auch nichts zu leeren*)
- ist erstens meist sowieso nicht nötig und
- zweitens meist nicht explizit nötig, da viele Klassen „autoflush“ unterstützen, so dass bei vielen Schlüsselereignisse automatisch ein „flush“ initiiert wird
- interessant vor allem bei PrintStream und Netzwerk-Streams (*die wir uns in der Vorlesung nicht anschauen*)



I/O-Streams: „Standard Streams“

Die meisten Betriebssysteme haben Standard-Streams, die z.B. standardmäßig Tastatureingaben lesen und Ausgaben auf den Bildschirm schreiben usw.

Java unterstützt drei Standard-Streams (Alternative dazu: „Console“-Objekt):

- für Eingaben (Input):
 - „Standard Input“: über „System.in“
 - für Ausgaben (Output):
 - „Standard Output“: über „System.out“
 - „Standard Error“: über „System.err“
- von Haus aus keine „Character Stream“-Eigenschaften; „Wrapping“ in „InputStreamReader“ möglich
- „PrintStream“-Objekte, die viele Eigenschaften von „Character Streams“ nachbilden

„Byte Streams“ (historisch bedingt)

Aufgabe 9

Sie kennen die Methode „println“ von „System.out“.

Versuchen sie, einen Ausgabe über den „Standard Error Stream“ zu generieren.

I/O-Streams: „Standard Streams“

Lösung 9

```
System.err.println("Wo ist der Fehler?");
```

Ausgabe:

Wo ist der Fehler?

I/O-Streams: Aufgabe 10 (fast) zum Abschluss

Ihnen sind die folgenden Quellcode-Teile gegeben, um eine dem nebenstehendem Aufbau entsprechende Datei namens „sportler.csv“ einzulesen.

```
try (
    // hier müsste man etwas tun
    ("sportler.csv")) {
    String line = bufferedReader.
    while (line != null) {
        Scanner s = new Scanner(line).useDelimiter(
        );
        System.out.println("Name: " + s.next() + ", Alter: "
            + s.next() + ", Sportart: " + s.next());
        line = bufferedReader.readLine();
    }
} catch (IOException e) {
    // hier müsste man etwas tun
}
```

sportler.csv

Name,Alter,Sportart
Andreas,20;Rudern
Berta,22;Fußball
Clemens,25;Tischtennis

Nutzen Sie einen „FileReader“, um „sportler.csv“ einzulesen. Umhüllen („Wrappen“) Sie den „FileReader“ mit einem „BufferedReader“. Nutzen Sie eine Methode von „BufferedReader“, um jeweils eine ganze Zeile („line“) der Datei einzulesen. Sorgen Sie dafür, dass der verwendete Scanner sowohl Kommas als auch Semikolons als Trennzeichen erkennt.

I/O-Streams: Lösung 10

```
try (BufferedReader bufferedReader
    = new BufferedReader(new FileReader("sportler.csv"))) {
    String line = bufferedReader.readLine();
    while (line != null) {
        Scanner s = new Scanner(line).useDelimiter(",|;");
        System.out.println("Name: " + s.next() + ", Alter: "
            + s.next() + ", Sportart: " + s.next());
        s.close();
        line = bufferedReader.readLine();
    }
} catch (IOException e) {
    // hier muesste man etwas tun
}
```

I/O-Streams: Aufgabe 11 zum Abschluss

Ihnen sind in der Datei „clowns.csv“ jeweils Daten von „Clowns“ gegeben, wobei in jeder Zeile der Datei die „Clown“-Daten wie folgt – jeweils getrennt durch Komma bzw. Doppelpunkt – gegeben sind:

Clownname, Lachfaktor; Zirkusname, Zirkus-ID

Ziel ist es, diese „Clowns“ als Objekte im Programm verfügbar zu machen.

- a) Deklarieren Sie zunächst eine Liste zur Aufnahme der „Clowns“ und initialisieren Sie diese mit einer neuen „ArrayList“.
- b) Nutzen Sie analog zu Aufgabe 10 einen durch einen „BufferedReader“ umhüllten „FileReader“ als I/O-Stream für die in der Datei enthaltenen Daten.
- c) Lesen Sie analog zu Aufgabe 10 die Daten der Datei zeilenweise ein.
- d) Nutzen Sie zum Verarbeiten der einzelnen Zeilen einen „Scanner“. Beachten Sie dabei die möglichen Trennzeichen zwischen den Daten.

Verarbeiten Sie die einzelnen Zeilen so, dass Sie jeweils die einzelnen für die Erzeugung eines neuen „Clowns“ benötigten Daten ermitteln und diese nutzen, um jeweils einen neuen „Clown“ zu erzeugen und fügen Sie diesen gleich Ihrer „Clown“-Liste hinzu.

- e) Lassen Sie sich zum Abschluss Ihre „Clown“-Liste anzeigen.

I/O-Streams: Lösung 11 zum Abschluss

```
List<Clown> clowns = new ArrayList<>();
try (BufferedReader bufferedReader =
    new BufferedReader(new FileReader("clowns.csv"))) {
    String line = bufferedReader.readLine();
    while (line != null) {
        Scanner s = new Scanner(line).useDelimiter(",|:");
        clowns.add(new Clown(s.next(), Integer.parseInt(s.next()),
            new Circus(s.next(), Integer.parseInt(s.next()))));
        s.close();
        line = bufferedReader.readLine();
    }
} catch (IOException e) {
    // hier muesste man etwas tun
}
System.out.println(clowns);
```

In der Praxis müsste man natürlich noch mehr Fehlerquellen ausschließen!

Einschub - Speicherbereiche:

Bedeutung für I/O-Streams, Serialisierung, Deserialisierung

Szenarien beim Austausch von Objekten zwischen verschiedenen Anwendungen, Prozessen und Systemen

- Möchte man alle Objektdaten speichern oder austauschen?
- Gibt es eventuell Daten, die nur intern benötigt werden und für andere Anwendungen nicht relevant sind?
- Gibt es vielleicht sogar temporäre Daten, die überhaupt nicht gespeichert werden müssen?

Weitere Überlegungen dazu:

- Kann man Speicherplatz sparen, wenn nur relevante Daten serialisiert werden?
- Ist die Datenübertragung effizienter, wenn nur relevante Daten serialisiert werden?
- Könnte aus auch aus Sicherheitsgründen relevant sein, nicht alle Daten zu serialisieren und ggf. übers Internet zu übertragen?

Einschub - Speicherbereiche:

transiente vs. permanente Speicher

Transienter Speicher (flüchtiger Speicher)

- bezeichnet einen flüchtigen Speicherbereich, der nur für kurze Zeit existiert

Permanenter Speicher (dauerhafter Speicher)

- bezeichnet einen permanenten Speicherbereich, in dem Daten langfristig gespeichert werden können

Einschub - Speicherbereiche:

transiente vs. permanente Speicher

Transienter Speicher (flüchtiger Speicher)

- bezeichnet einen flüchtigen Speicherbereich, der nur für kurze Zeit existiert

Permanenter Speicher (dauerhafter Speicher)

- bezeichnet einen permanenten Speicherbereich, in dem Daten langfristig gespeichert werden können

- bei der Serialisierung ist in Java dafür das Schlüsselwort „transient“ wichtig
- mit „transient“ markierte Objekteigenschaften werden bei der Standard-Serialisierung nicht serialisiert und können dann entsprechend bei der Deserialisierung nicht aus dem serialisierten Objektzustand wieder hergestellt werden
- Anwendungsfälle: abgeleitete Eigenschaften; Eigenschaftswerte, die den Objektzustand nicht repräsentieren; nicht-serialisierbare Eigenschaften (z.B. bei Referenzen auf Objekte); sensible Daten, die nicht über ein Netzwerk gesendet werden sollen
- Bsp: `private transient String fullName`; `private transient Image thumbnailImage`;

Einschub - Speicherbereiche:

transiente vs. permanente Speicher

Transienter Speicher (flüchtiger Speicher)

- bezeichnet einen flüchtigen Speicherbereich, der nur für kurze Zeit existiert

Permanenter Speicher (dauerhafter Speicher)

- bezeichnet einen permanenten Speicherbereich, in dem Daten langfristig gespeichert werden können

- umfasst insbesondere
 - Dateien (einfache Text-Dateien bis hin zu Objekt-Dateien serialisierter Objekte),
 - Datenbanken (insbesondere bei großen Datenmengen und komplexen Datenstrukturen) und
 - Netzwerk-Sockets (Java bietet APIs, um Netzwerk-Sockets zu erstellen und so Daten über das Netzwerk zu senden; Sockets sind dabei Endpunkte im über ein Netzwerk laufenden Kommunikationsfluss zwischen Programmen)



(extern) gespeicherte Daten

- Serialisierung/
Deserialisierung
- Input/Output-Streams
- Umgang mit Dateien
und Verzeichnissen
- Ausblick auf die Nutzung
von Daten aus
Datenbanken

Umgang mit Dateien und Verzeichnissen

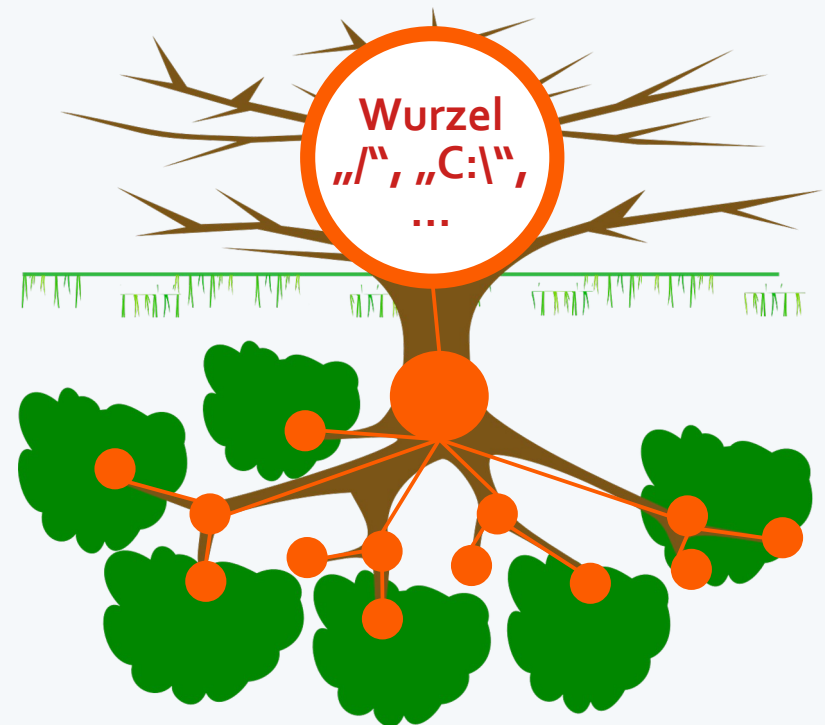
Wie sieht die Organisation dafür aus?

Verschiedene Betriebssysteme, verschiedene Dateisysteme ...

Damit man überhaupt mit Dateien auf einem Speichermedium arbeiten kann, muss man zunächst auf diesem Speichermedium eine Ablagestruktur festlegen, die das Speichern, Wiederfinden, Lesen, Verändern und Löschen von Dateien ermöglicht.

Diese Ablagestruktur wird durch ein „Dateisystem“ zur Verfügung gestellt, welches man durch entsprechendes „Formatieren“ auf ein Speichermedium bekommt.

Die heute üblichen Dateisysteme sind hierarchisch in Form eines Dateibaumes strukturiert.



Umgang mit Dateien und Verzeichnissen

Wie sieht die Organisation dafür aus?

Beispiele für Dateisysteme

- FAT32 (FAT: „File Allocation Table“): sehr kompatibel; gute Hardwareunterstützung (daher häufig auf mobilen Datenträgern zu finden); allerdings Dateigröße evtl. praxisrelevant begrenzt und keine Verschlüsselungs- und Komprimierungsfunktionen
- exFAT: Erweiterung von FAT32 für größere Dateien
- NTFS („New Technology File System“): Verbesserung von FAT; Datenverschlüsselung und -komprimierung wird unterstützt; größere Dateien sind möglich (häufig auf Windowsfestplatten)
- HFS+ („Hierarchical File System Plus“): Dateisystem auf (älteren) Apple-Computern
- APFS („Apple File System“): modernes Dateisystem auf Apple-Computern; optimiert für den Einsatz auf SSDs; interessante Zusatzfunktionen (z.B. Schutz bei Systemabstürzen)
- ext4 („fourth extended filesystem“): modernes Dateisystem auf Linux-Computern

Umgang mit Dateien und Verzeichnissen

Wie sieht die Organisation dafür aus?

Dateisysteme: Funktionsebenen

- Struktur:
Wie sind Dateien und Verzeichnisse (organisieren Dateien und geben diesen eine hierarchische Struktur) organisiert? Gibt es überhaupt Verzeichnisse oder befinden sich alle Dateien „flach“ auf einer Ebene?
- Speicherplatzverwaltung:
Wie wird freier und belegter Speicherplatz verwaltet? Wie wird Speicherplatz für Dateien und Verzeichnisse zugewiesen?
- Zugriffskontrolle:
Gibt es eine Zugriffskontrolle? Wer kann auf was zugreifen? Wie komplex ist das Rechtemanagement?
- Dateisystemtreiber:
Wie gestaltet sich die Kommunikation zwischen dem Betriebssystem und dem Dateisystem? Hat das Betriebssystem überhaupt die nötigen Treiber für den Umgang mit einem bestimmten Dateisystem?
- Dienstfunktionen:
Gibt es Dienstprogramme zur Formatierung, zur Datensicherung und zur Wiederherstellung? Welche weiteren Dienstprogramme gibt es?

Umgang mit Dateien und Verzeichnissen

Wie sieht die Organisation dafür aus?

Dateisysteme: „Datei“-Typen

ausführbare Dateien

- können vom Betriebssystem (also abhängig davon) oder einer Laufzeitumgebung (oder abhängig davon) ausgeführt werden
- Programme, Skripte, Binärdateien mit Anweisungen

nicht-ausführbare Dateien

- enthalten keine direkt verarbeitbaren Anweisungen (also wieder betriebssystem-abhängig)
- z.B. Textdateien, Bilder, Videos, Datenbankdateien

Verzeichnisse

- Dateien, die andere Dateien und Verzeichnisse enthalten
- dienen zur Organisation und Strukturierung von Dateien

Pseudo-Dateien

- spezielle Betriebssystem-Dateien mit Informationen über Systemzustände, Prozesse, Geräte
- ggf. nur mit speziellen Werkzeugen und Berechtigungen erreichbar
- z.B. \$Volume, \$LogFile, /proc, /dev/printer

Umgang mit Dateien und Verzeichnissen

Wie sieht die Organisation dafür aus?

Dateien – Daten (Meta-Daten) zu Dateien

- beschreiben die Eigenschaften einer Datei oder eines Verzeichnisses, die vom Betriebssystem oder dem Dateisystem bereitgestellt werden
- können meist von anderen Programmen zum Organisieren und beim Durchsuchen von Dateien verwendet werden
- manche Dateisysteme erlauben zusätzlich die Festlegung weiterer, benutzerdefinierter Eigenschaften, die speziell für die Bedürfnisse einer bestimmten Anwendung oder Organisation geeignet sind.
- **übliche Daten:**
 - Name: Name der Datei oder des Verzeichnisses
 - Größe: Größe der Datei (in Bytes)
 - Erstelldatum: Datum, an dem die Datei erstellt wurde
 - Änderungsdatum: Datum, an dem die Datei zuletzt geändert wurde
 - Zugriffsdatum: Datum, an dem die Datei zuletzt geöffnet oder gelesen wurde
 - Zugriffsrechte: Berechtigungen, die angeben, wer auf die Datei zugreifen darf
 - Dateityp: Typ der Datei (z.B. Text, Bild, Video, Audio)

Umgang mit Dateien und Verzeichnissen

Moderne Möglichkeiten mit Java

„java.io“ vs. „java.nio“

„java.io“ (klassisch)

„java.nio“ (modern)

- **„Stream“-orientiert:**
 - Daten werden Byte-für-Byte bzw. mit Puffer Zeile-für-Zeile, aber immer **Eins-Unwiederbringlich-Nach-Dem-Anderen von einem Stream/Puffer-Stream** verarbeitet
 - ist oft langsam, da synchron/blockierend gearbeitet wird
 - bietet durch den „Stream“-Ansatz **einfache, gute Filtermöglichkeiten**
- viele Klassen zum Lesen und Schreiben von Daten mittels I/O-Streams
- Klasse „java.io.File“ hat aber einige Schwachstellen (zu wenig geworfene informative Ausnahmen; inkonsistente Umbenennung von Dateien auf verschiedenen Plattformen; schwache Unterstützung von Meta-Daten; Skalierungsprobleme; Probleme bei symbolischen Ringverweisen)

Umgang mit Dateien und Verzeichnissen

Moderne Möglichkeiten mit Java

„java.io“ vs. „java.nio“

„java.io“ (klassisch)

„java.nio“ (modern)

- **„Block/Puffer“-orientiert:**
 - Daten werden nicht aus einem „Stream“, sondern immer aus einem/in einen Speicherblock (Puffer) verarbeitet (sind also per se länger greifbar)
 - statt der „Streams“ zur Verbindung der Anwendung zum restlichen System gibt es dann Kanäle („Channels“), die den Speicherblock mit dem restl. System verbinden
 - erlaubt eine asynchrone, nicht-blockierende Arbeitsweise
- enthält neue, verbesserte und erweiterte Schnittstellen zur Arbeit mit Dateien (inklusive Arbeit mit symbolischen Links, Meta-Daten usw.) sowie
- neue bzw. verbesserte Schnittstellen zur Unterstützung von Multi-Casts und asynchroner Ein- und Ausgabe von Daten
- in der Regel leistungsfähiger und flexibler als „java.io“, aber etwas komplexer
- wird vornehmlich bei Dateisystem-Arbeiten (kopieren, suchen, ...) als Ersatz für die schwache „java.io.File“, bei Serveranwendungen, bei denen eine schnelle, asynchrone Arbeitsweise nötig ist, sowie falls keine „Stream“-Verarbeitung sinnvoll ist, genutzt

Umgang mit Dateien und Verzeichnissen

Moderne Möglichkeiten mit Java

„java.io“ vs. „java.nio“: Wie kann man „Datei-Arbeit“ modernisieren?

„java.io“
(klassisch)

„java.nio“
(modern)

I/O-Streams sind prinzipiell meist brauchbar (auch wenn sie „java.io.File“ benutzen), aber: reine Datei-Arbeit (suchen, organisieren, ...) mit „java.io.File“ sollte man modernisieren

„java.io.File“ (Klasse)

„java.nio.file“ (Paket)

- „Paths“: für Umgang mit Dateipfaden
- „Files“: für Datei-Operationen (kopieren, umbenennen, ...)
- „FileSystem“: für Dateisystem-Informationen
- ... (viele weitere Klassen)

„java.nio.file.attribute“ (Paket)

...

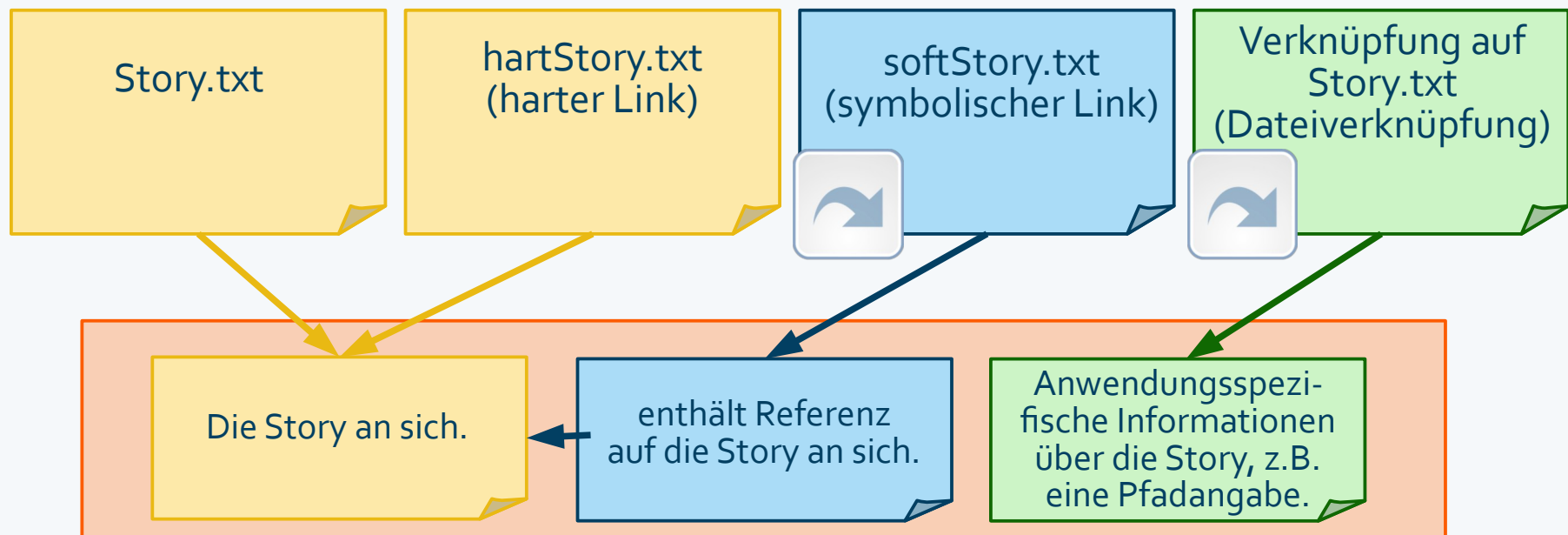
Tipps zur Modernisierung (inkl. Abbildung alter auf neue Möglichkeiten):
<https://docs.oracle.com/javase/tutorial/essential/io/legacy.html>

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Erste Schritte

Dateien: Verweise

- Harte Links [hard links] und dazu ähnlich noch Abzweigungspunkte [junctions]
- Symbolische Links [soft links]
- Verknüpfungen [short cuts]



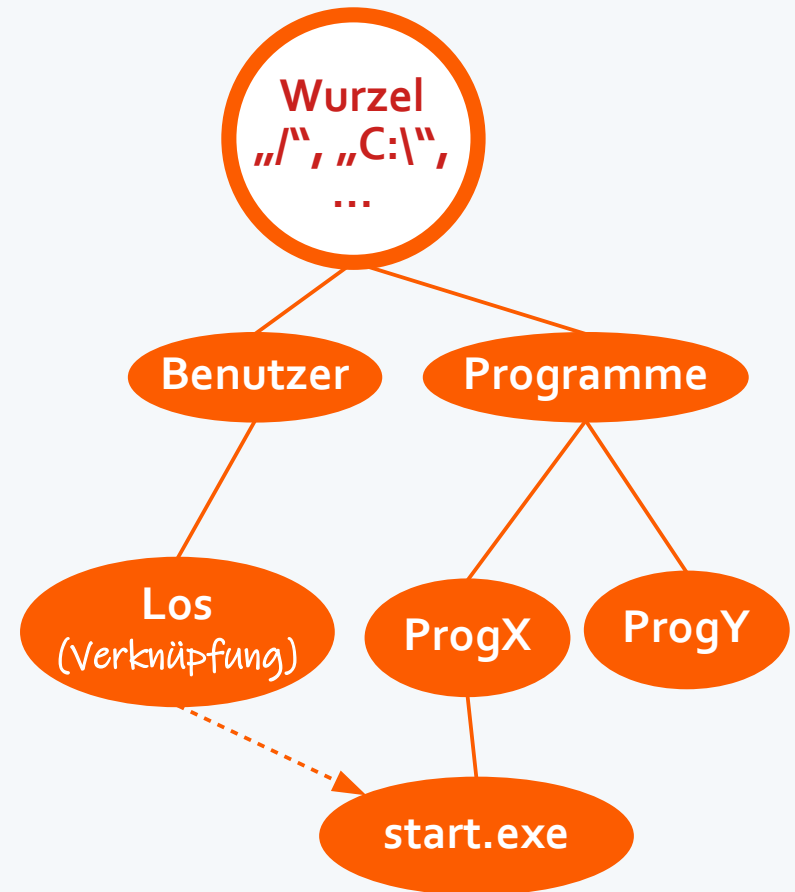
Links erstellt man in Windows mit „mklink“, sofern man die nötigen Rechte dafür hat. Einfache Dateiverknüpfungen sind das, was der „normale“ Nutzer tut.

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Erste Schritte

Dateipfade

- Dateien werden über ihren Pfad von der Wurzel im Dateisystem aus identifiziert
- Trennzeichen (z.B. „/“ oder „\“) für die Pfadbestandteile (Knoten) ist systemabhängig
- Beispiele:
C:\Programme\ProgX\start.exe (Windows)
/Programme/ProgX/start.exe (Linux)



Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Erste Schritte

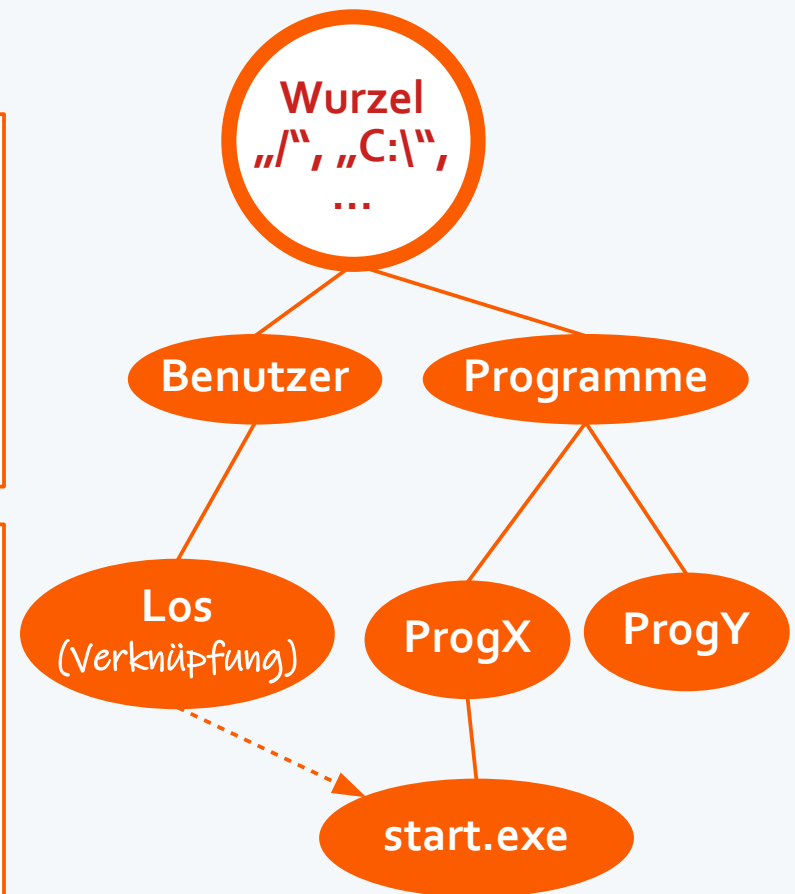
Dateipfade – absolute und relative

absoluter Pfad

- kompletter Pfad von der Wurzel einschließlich aller Pfadbestandteile
- z.B.: C:\Programme\ProgX\start.exe

relativer Pfad

- Pfadbestandteile ab einem bestimmten Knoten
- benötigt weitere Information (Lage des Startknotens) zur Dateilokalisierung
- z.B.: ProgX\start.exe

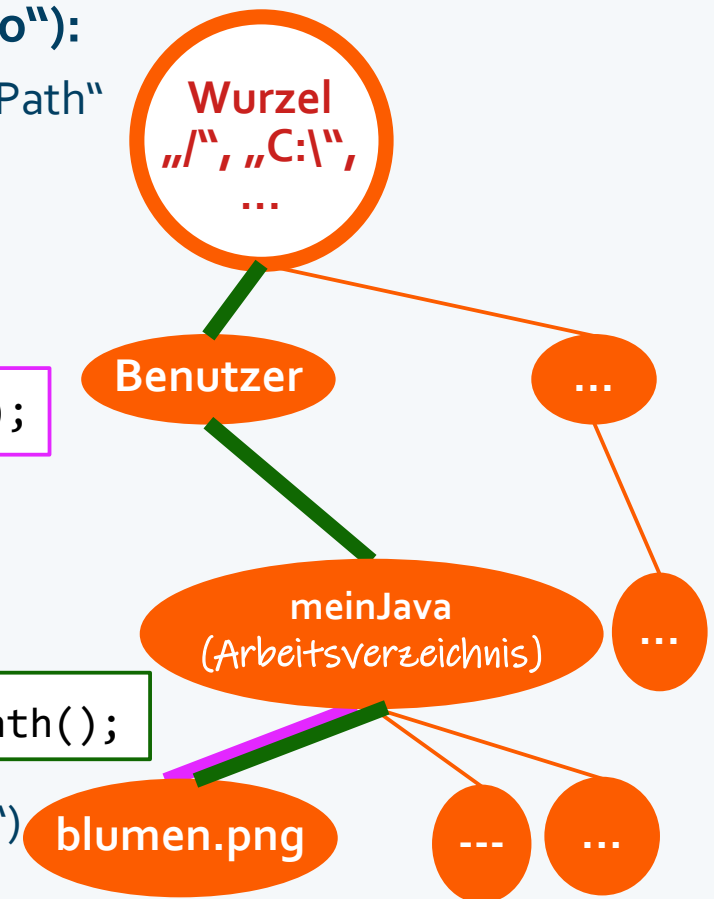


Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Erste Schritte

Dateipfade (nicht unbedingt Dateien) mit Java (mit „nio“):
zur Repräsentation von Dateipfaden dient Schnittstelle „Path“
aus „java.nio.file“:

- Pfaderzeugung:
 - mit Methode „of()“
 - z.B. `Path filePath = Path.of("blumen.png");`
- Pfadumwandlung:
 - mit Methoden „toUri“, „toAbsolutePath“ oder „toRealPath“ (nur bei existierenden Dateien)
 - z.B. `Path fullPath = filePath.toAbsolutePath();`
- weitere Methoden, um Pfade zu verbinden („resolve“) oder Pfade zwischen Pfaden zu ermitteln



Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Erste Schritte

Informationen über Dateipfade:

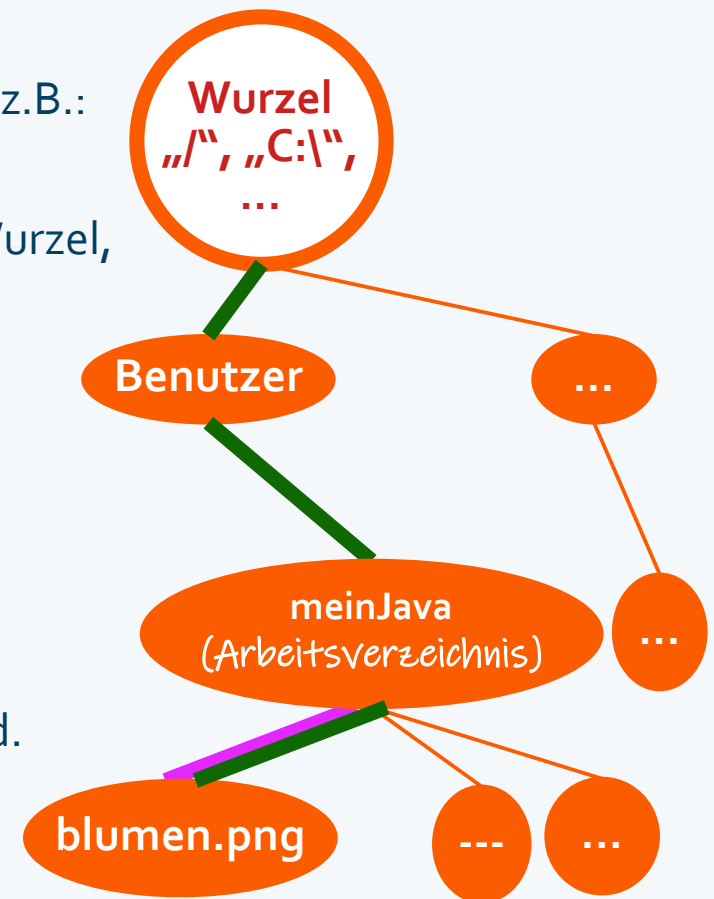
Erhält man viele über Methoden der Schnittstelle „Path“, z.B.:

- `getFileName()`: Name des letzten Pfadelementes
- `getName(x)`: Name des x. Pfadelementes nach der Wurzel, wobei die Zählung bei Null beginnt
- `getNameCount()`: Anzahl der Pfadelemente
- `getParent()`: Pfad ohne letztes Element
- `getRoot()`: Wurzel des Pfades

Aufgabe 12

Erstellen Sie einen relativen Pfad zu einer Datei in Ihrem Arbeitsverzeichnis. Ermitteln Sie dazu den absoluten Pfad.

Nutzen Sie für beide Pfadangaben mindestens drei Methoden aus der Schnittstelle „Path“, um weitere Informationen über die Pfade zu erhalten.



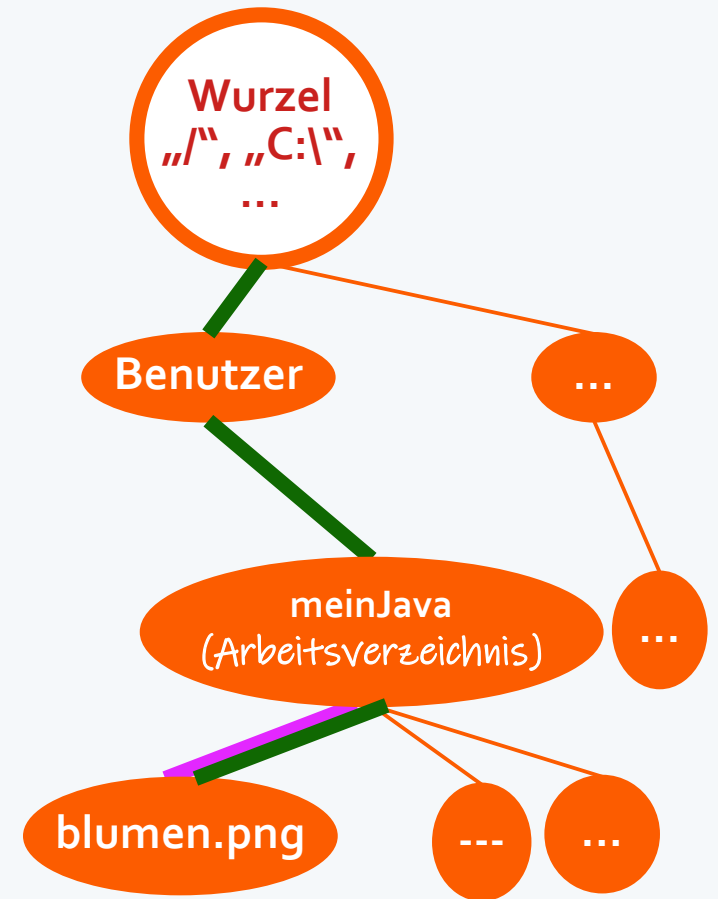
Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Erste Schritte

Lösung 12:

```
Path filePath = Path.of("blumen.png");
System.out.println(filePath.toString());
System.out.println(filePath.getParent());
System.out.println(filePath.getRoot());

Path fullPath = filePath.toAbsolutePath();
System.out.println(fullPath);
System.out.println(fullPath.getName(1));
System.out.println(fullPath.getParent());
System.out.println(fullPath.getRoot());
```



Umgang mit Dateien und Verzeichnissen

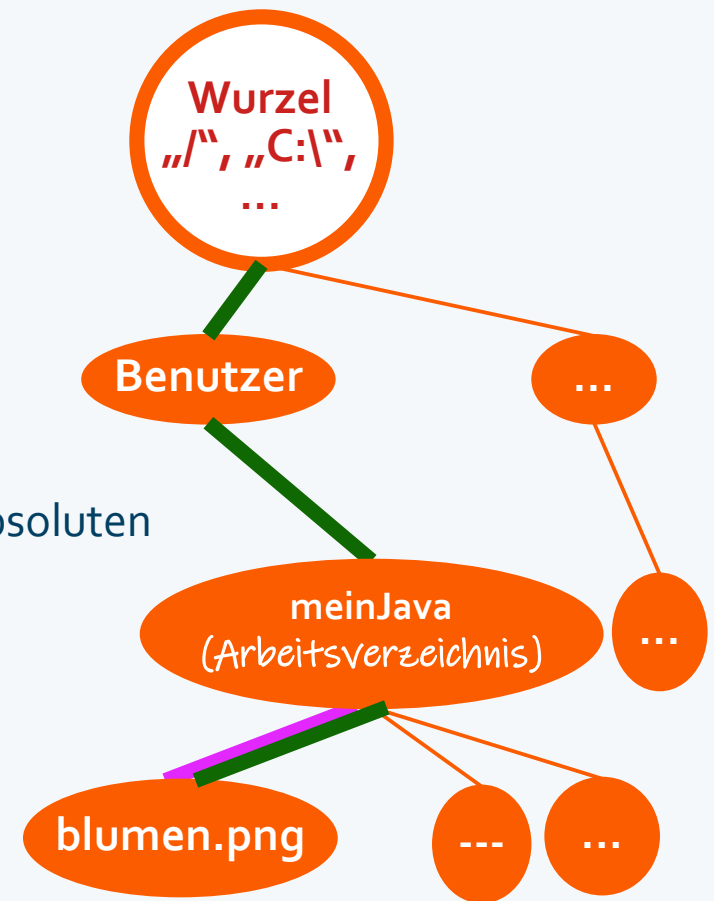
„java.nio.file“ – Erste Schritte

Informationen über Dateipfade:

- relativ und absoluter Pfad haben auch intern einen verschieden hohen Informationsgehalt
- zum Vergleich zweier Pfade kann man wie üblich die Methode „equals“ benutzen

Aufgabe 13

Vergleichen Sie ihren relativen und den dazugehörigen absoluten Pfad. Was stellen Sie fest?



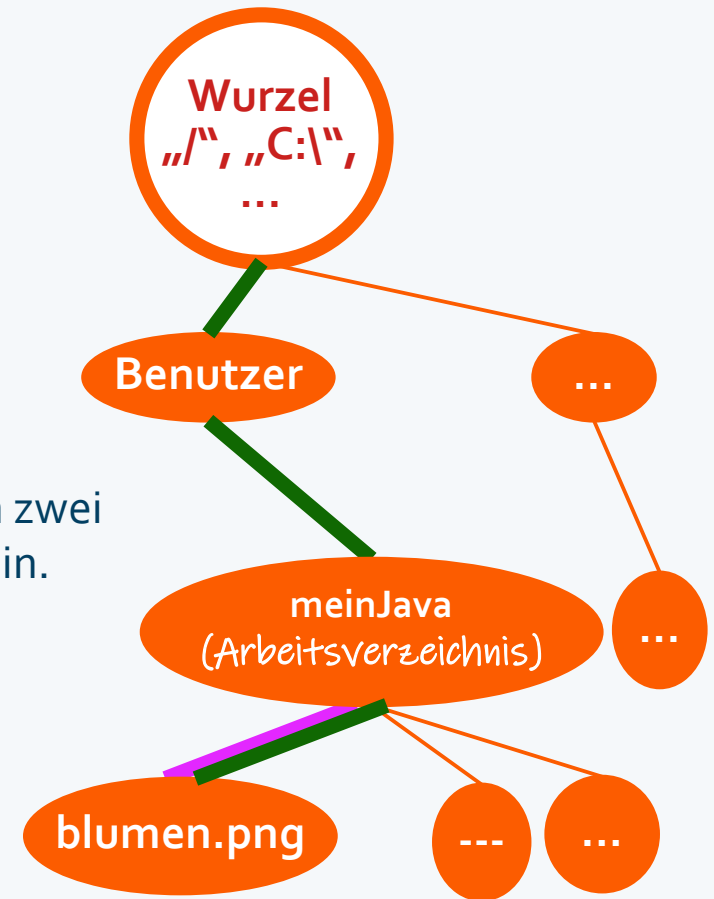
Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Erste Schritte

Lösung 13

```
if (filePath.equals(fullPath)) {  
    System.out.println("same path ");  
    //but it is not the same ...  
}
```

Der Pfadvergleich bezieht sich auf die Pfade an sich. Auch zwei Pfade, die zur selben Datei führen, müssen nicht gleich sein.



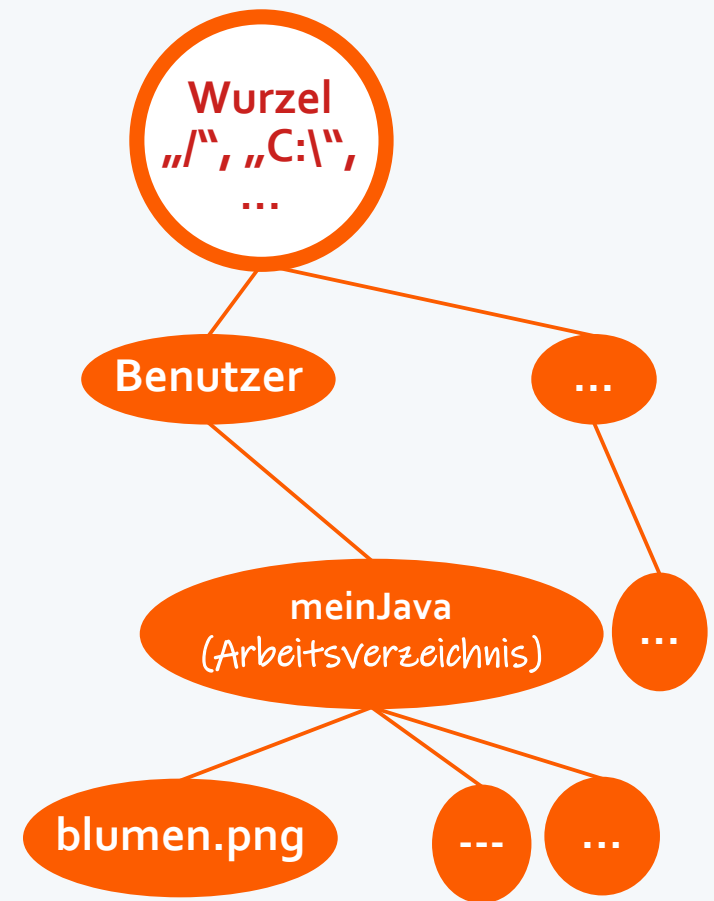
Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“)

Dateien mit „java.nio.file“

Einstiegspunkt für den Umgang mit Dateien ist die Klasse „Files“ („java.nio.file.Files“) mit vielen diesbezüglichen Methoden zum:

- Dateityp ermitteln
- Vergleich, ob verschiedene Pfade zur gleichen Datei führen
- Dateien erstellen, kopieren und löschen
- Verzeichnisse erstellen
- Dateien im Dateisystem verschieben
- Dateien einlesen und schreiben (wir haben uns dazu schon Methoden aus „java.io“ angeschaut)
- Umgang mit Dateiattributen
- ... (viele mehr)



Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Aufgabe 14 – Dateityp ermitteln

Finden Sie mit der Methode „isRegularFile“ für einen Ihrer Pfade aus Aufgabe 12 heraus, ob dieser anscheinend (also ohne Überprüfung des eigentlichen Dateiinhaltes) zu einer regulären Datei gehört.

Anmerkung:

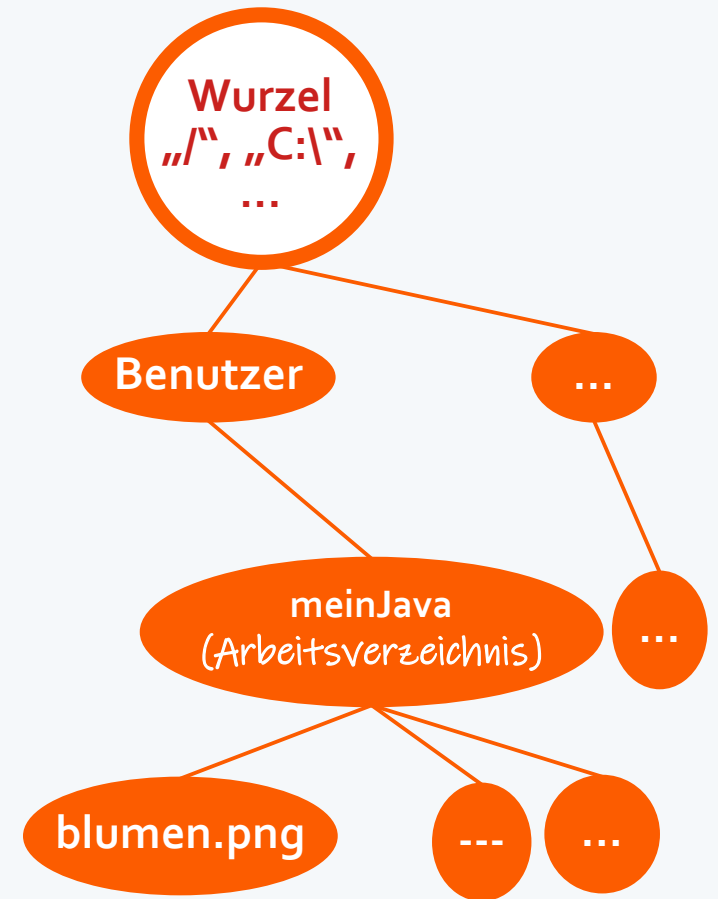
Eine „reguläre“ Datei ist vereinfacht gesagt einfach eine Datei, die irgendwelche Text- oder Binärdaten enthält.

Keine „regulären“ Dateien sind zum Beispiel Verzeichnisse oder Pseudo-Dateien.

Unerreichbare Dateien können nicht als „reguläre“ Dateien erkannt werden.

Dokumentation:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/Files.html>

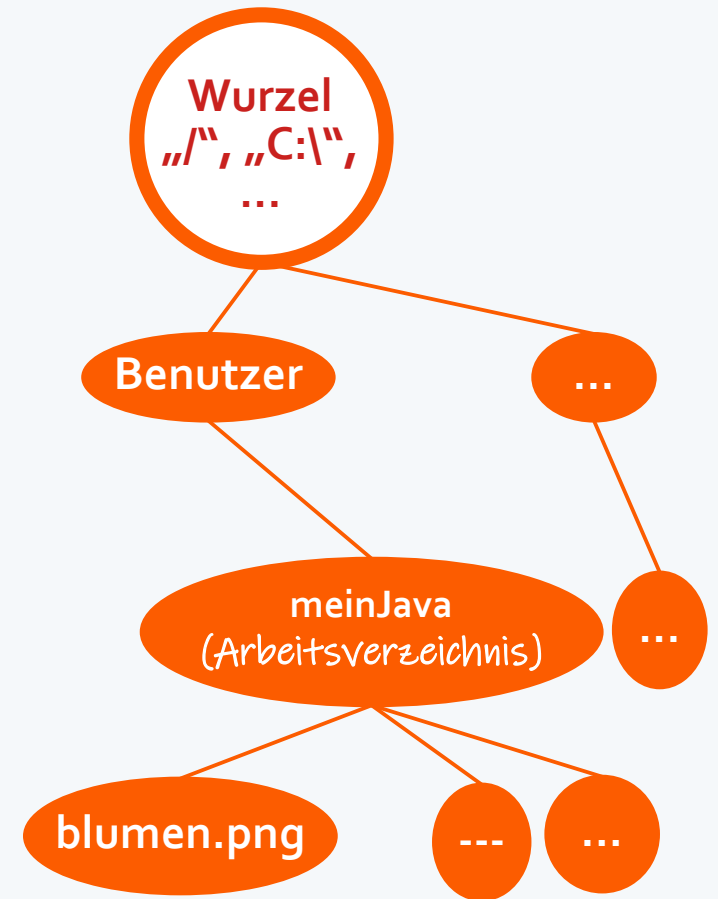


Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Lösung 14 – Dateityp ermitteln

```
if (Files.isRegularFile(filePath)) {  
    System.out.println("is regular file");  
    // path is regular file  
}
```



Dokumentation:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/Files.html>

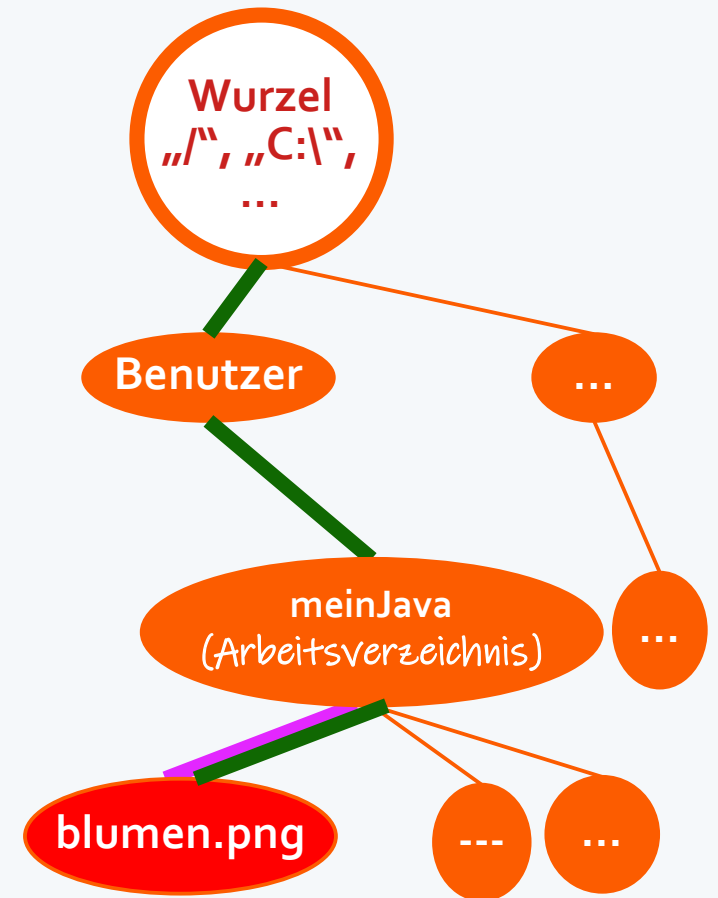
Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Aufgabe 15 – Pfadvergleich

Wir haben in Aufgabe 13 zwei Pfade verglichen.

Finden Sie mit der Methode „isSameFile“ heraus, ob beide Pfade zur gleichen Datei führen.



Dokumentation:

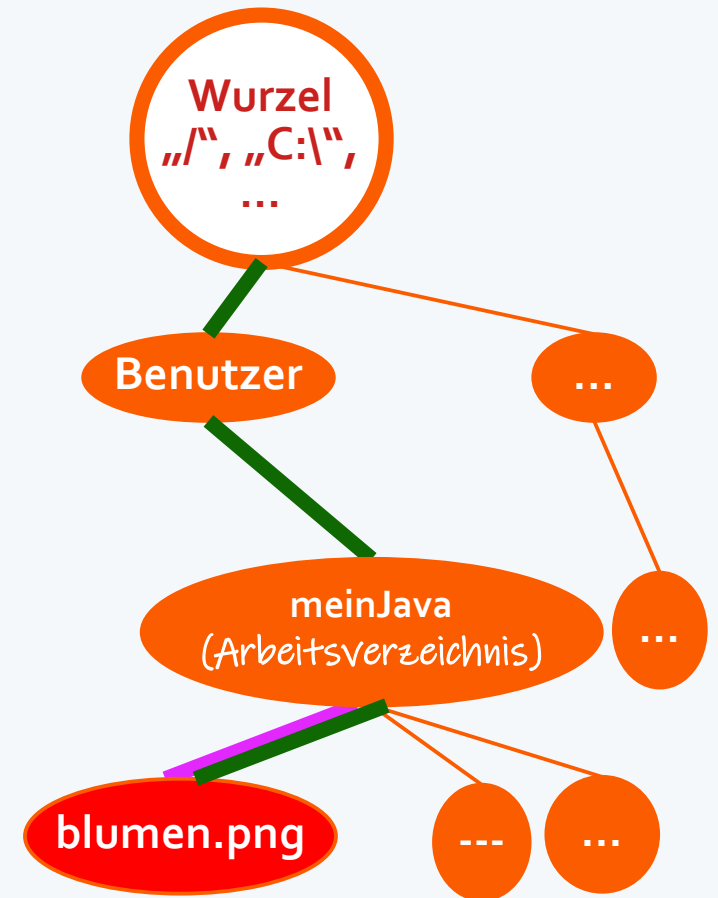
<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/Files.html>

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Lösung 15 – Pfadvergleich

```
if (Files.isSameFile(filePath, fullPath)) {  
    System.out.println("same file");  
}
```



Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Aufgabe 16 – Dateien kopieren und löschen

Die Datei, die sich hinter Ihren in der vorigen Aufgabe verwendeten Pfaden verbirgt, soll testweise kopiert und anschließend soll die Kopie gelöscht werden.

Deklarieren und initialisieren Sie dafür zunächst eine Variable „copyPath“ vom Typ „Path“, um einen Pfad zur (als Datei noch nicht existierenden) Kopie zu erstellen (Achtung: Noch nicht benutzten Dateiname verwenden!)

Falls Ihr bisheriger Pfad durch die Variable „filePath“ repräsentiert wird, lässt sich eine Kopie zum Beispiel wie folgt erreichen:

```
Files.copy(filePath, copyPath,  
    StandardCopyOption.COPY_ATTRIBUTES, StandardCopyOption.REPLACE_EXISTING);
```

Nutzen Sie anschließend eine Methode von „Files“, um die Kopie wieder zu löschen.

Dokumentation:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/Files.html>

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Lösung 16 – Dateien kopieren und löschen

```
Path filePath = Path.of("blumen.png");  
Path copyPath = Path.of("copiedBlumen.png");  
Files.copy(filePath, copyPath,  
    StandardCopyOption.COPY_ATTRIBUTES, StandardCopyOption.REPLACE_EXISTING);  
try {  
    Files.delete(copyPath);  
} catch (NoSuchFileException x) {  
    // tu was  
} catch (DirectoryNotEmptyException x) {  
    // tu was  
} catch (IOException x) {  
    // tu was  
}
```

← existiert

← oder:
„deleteIfExists“

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Aufgabe 17 – Verzeichnis erstellen und Datei verschieben

- Erstellen Sie einen neuen Pfad für ein zu erstellendes Verzeichnis (z.B. „test“).
- Nutzen Sie eine Methode von „Files“, um tatsächlich das Verzeichnis zu erstellen.
- Wiederholen Sie Ihren Kopiervorgang aus der letzten Aufgabe, um eine Datei zu erzeugen, die verschoben werden soll.
- Erstellen Sie einen Zielpfad, auf den die in c) erstellte Kopie verschoben werden soll. Nutzen Sie dafür Ihren Pfad aus a), die Methode „resolve“, um diesen Pfad zu erweitern und die Methode „getFileName“ im Zusammenhang mit dem Pfad ihrer Kopie, um von diesem nur den Dateinamen zu ermitteln (nur dieses Pfadelement soll mit „resolve“ an das Zielverzeichnis angefügt werden).
- Nutzen Sie die Methode „move“ mit der Option „StandardCopyOption.REPLACE_EXISTING“, um die Kopie aus c) entsprechend dem Pfad aus d) zu verschieben.

Dokumentation:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/Files.html>

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Lösung 17 – Verzeichnis erstellen und Datei verschieben

```
Path newDir = Path.of("test");  
// oder: Files.createDirectory(newDir);  
Files.createDirectories(newDir);  
  
Files.copy(filePath, copyPath,  
    StandardCopyOption.COPY_ATTRIBUTES, StandardCopyOption.REPLACE_EXISTING);  
Path targetPath = newDir.resolve(copyPath.getFileName());  
Files.move(copyPath, targetPath, StandardCopyOption.REPLACE_EXISTING);
```

Dokumentation:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/Files.html>

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Aufgabe 18 – Umgang mit Dateiattributen

Lesen Sie die Dokumentation zur Schnittstelle „BasicFileAttributes“.

Erzeugen Sie für einen Pfad Ihrer Wahl „BasicFileAttributes“ und nutzen Sie dann mindestens zwei Methoden der Klasse, um sich die davon gelieferten Datei-Attribute anzeigen zu lassen.

Dokumentation:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/attribute/BasicFileAttributes.html>

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Lösung 18 – Umgang mit Dateiattributen

```
BasicFileAttributes attr = Files.readAttributes(filePath,  
    BasicFileAttributes.class);  
  
System.out.println("creationTime: " + attr.creationTime());  
System.out.println("lastAccessTime: " + attr.lastAccessTime());  
  
System.out.println("lastModifiedTime: " + attr.lastModifiedTime());  
System.out.println("isDirectory: " + attr.isDirectory());  
System.out.println("isOther: " + attr.isOther());  
System.out.println("isRegularFile: " + attr.isRegularFile());  
System.out.println("isSymbolicLink: " + attr.isSymbolicLink());  
System.out.println("size: " + attr.size());
```

Dokumentation:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/attribute/BasicFileAttributes.html>

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Datei einlesen – ein Beispiel (es gibt viele Wege, auch Methoden, die einen bekannten `InputStream` oder einen bekannten `BufferedReader` liefern ...)

(nicht klausurrelevant – wir haben „java.io“ angeschaut und Sie sollen nicht in der Klausur doppelt lernen müssen ;-))

```
try {
    Path fileWriterPath = Path.of("FileWriterTest.txt");
    List<String> lines = Files.readAllLines(fileWriterPath);
    for (String line : lines) {
        System.out.println(line);
    }
} catch (Exception e) {
    System.out.println("An error occurred: " + e.getMessage());
}
```

Dokumentation:

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/nio/file/Files.html>

Umgang mit Dateien und Verzeichnissen

„java.nio.file“ – Dateien („Files“) am Beispiel

Sonstiges *(auch nicht klausurrelevant)*

Das Paket „java.nio.file“ bietet noch viele weitere, interessante Fähigkeiten, z.B.:

- Dateibäume durchwandern (Stichwort: „FileVisitor“)
- nach Dateien suchen (Stichwort: „PathMatcher“, „regex“ bzw. „glob“ [ist so ähnlich])
- nichtsequentieller („random access“) Dateizugriff (nicht ab Beginn, sondern ab einer späteren Stelle in einer Datei lesen oder schreiben)
- „WatchService“, um Veränderungen im Dateisystem zu registrieren (z.B. Erstellen neuer Dateien in einem Verzeichnis, Dateiumbenennung, Dateispeicherung durch andere Programme)
- ermitteln des im Dateisystem verwendeten Trennzeichens und aller Dateispeicher (Stichwort: Laufwerke, Volumen)



(extern) gespeicherte Daten

- Serialisierung/Deserialisierung
- Input/Output-Streams
- Umgang mit Dateien und Verzeichnissen
- Ausblick auf die Nutzung von Daten aus Datenbanken

Ausblick auf die Nutzung von Daten aus Datenbanken

übliche Nutzungsvarianten

- durch Nutzung der „Java Database Connectivity (**JDBC**) API“:
Diese Datenbankschnittstelle ermöglicht prinzipiell universellen Datenzugriff auf jede Datenquelle, von relationalen Datenbanken bis hin zu Tabellenkalkulationen und Flat Files. Praktisch wird sie aber hauptsächlich für den Zugriff auf relationale Datenbanken verwendet.
- durch Nutzung eines die „Jakarta Persistence API (**JPA**)“ implementierenden Persistenzframeworks wie
Apache OpenJPA, **Hibernate** oder EclipseLink (Referenzimplementierung)
Die „Jakarta Persistence API“ ist eine Schnittstelle für Java-Anwendungen, die insbesondere dazu dient, Java-Objekte dauerhaft (persistent) in einer relationalen Datenbank zu speichern und darauf zuzugreifen.

Ausblick auf die Nutzung von Daten aus Datenbanken: „Java Database Connectivity (JDBC) API“

grundsätzliches Vorgehen

1. JDBC-Treiber laden: Für die Kommunikation zwischen Java und einer Datenbank muss eine Implementierung der JDBC API in Form eines JDBC-Treibers benutzt werden. Die meisten Datenbank-Hersteller bieten einen entsprechenden JDBC-Treiber für ihre spezielle Datenbank an. Dieser muss heruntergeladen und in das eigene Java-Projekt eingebunden werden.

Für MySQL finden Sie beispielsweise den Treiber und Informationen dazu unter:
<https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-overview.html>

Ausblick auf die Nutzung von Daten aus Datenbanken: „Java Database Connectivity (JDBC) API“

grundsätzliches Vorgehen

2. Verbindung zur Datenbank herstellen: Mit der „DriverManager“-Klasse kann unter Angabe der URL der zu verbindenden Datenbank, des Benutzernamens und des Passwortes eine Verbindung zur gewünschten Datenbank hergestellt werden.

```
String url = ...;
String username = ...;
String password = ...;
try (Connection con = DriverManager
    .getConnection(url, username, password)) {
    // hier die über „con“ angesprochene Verbindung nutzen
}
```

Ausblick auf die Nutzung von Daten aus Datenbanken: „Java Database Connectivity (JDBC) API“

grundsätzliches Vorgehen

3. SQL-Abfragen generieren (Stichworte: „Statement“, „PreparedStatement“, „CallableStatement“ über die im zweiten Schritt erstellte „Connection“) und ausführen (Stichworte: „executeQuery()“, „executeUpdate()“, „execute()“).

Zugriff auf zurückgegebene Daten zum Beispiel über ein „ResultSet“-Objekt.

```
try (Statement stmt = con.createStatement()) {  
    // hier „stmt“ nutzen, z.B.:  
    String selectSql = "SELECT * FROM students";  
    try (ResultSet result = stmt.executeQuery(selectSql)) {  
        // hier „result“ nutzen, z.B.:  
        while (result.next()) {  
            int id = result.getInt("id");  
            String name = result.getString("name");  
            // ...  
        }  
    }  
}
```

Ausblick auf die Nutzung von Daten aus Datenbanken: „Jakarta Persistence API (JPA)“

- abstrahiert zugrunde liegende Datenbank und bietet objektorientierte Sichtweise auf Datenbank-Entitäten
- mittels JPA-Annotationen (z.B. @Entity, @Id, @Column usw.) können Java-Objekte als Entitäten gekennzeichnet werden, wodurch gleichzeitig die Abbildung von Java-Objekten auf Datenbanktabellen und umgekehrt definiert wird
- verwendet „EntityManager“, um die Entitäten in der Datenbank zu verwalten und die Datenbank-Operationen durchzuführen (z.B. Datenbankabfragen, Einfügen, Aktualisieren oder Löschen von Daten)
- ist prinzipiell plattformunabhängig und ermöglicht die Arbeit mit verschiedenartigen Datenbanken (z.B. Oracle, MySQL, PostgreSQL usw.), ohne dass Entwickler SQL oder spezielle Datenbank-APIs für jede Datenbank beherrschen müssen
- benötigt wird für die Verwendung allerdings eine Implementierung von JPA; eine der bekanntesten und weit verbreitete Implementierung ist Hibernate, siehe: <https://hibernate.org/>

Vielen Dank für Ihre
Aufmerksamkeit.

