

Filtern von Dateinamen

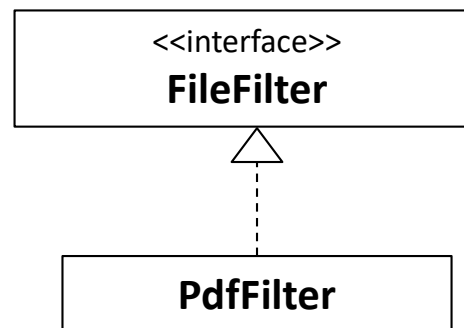
- Alle Einträge in ein Verzeichnis können wir über die Methode `File[] listFiles()` der Klasse `File` ermitteln
- Wir können die Datei- und Verzeichnisnamen nun filtern, indem wir die überladene Methode `File[] listFiles(FileFilter filter)` verwenden
 - Es werden nur Datei-/Verzeichnisnamen geliefert, die ein bestimmtes Filterkriterium erfüllen (z.B. Dateiendung `.pdf`)
 - Die Filterung wird von einem Objekt vom Typ `FileFilter` vorgenommen
- Das Interface `java.io.FileFilter` enthält nur eine Methode

```
public interface FileFilter {  
    boolean accept(File pathname);  
}
```

`FileFilter.java`

true, wenn pathname das Filterkriterium erfüllt, sonst false

- Eine Klasse PdfFilter kann nun die Schnittstelle FileFilter implementieren



- Dann kann z.B. in der Methode traverse() der Klasse FileTree (siehe oben) ein entsprechendes Filterobjekt genutzt werden

```
File[] list = dir.listFiles(new PdfFilter());
```

ersetzt ursprünglichen Aufruf von listFiles()

- Aufgabe 27: Implementieren Sie die Klasse PdfFilter! Hinweis: Verwenden Sie die Methoden toLowerCase() und endsWith() der Klasse String.

RandomAccessFile

- Wahlfreier Zugriff
 - Relativ zum Anfang kann das *i*-te Byte einer Datei angesteuert werden, ohne dass die dazwischen liegenden Bytes gelesen werden müssen
 - Basis für indizierte und gestreute Speicherung
- Die Klasse `RandomAccessFile` ermöglicht den wahlfreien Zugriff in Java
- Es kann ein Positionszeiger über die Bytes einer Datei navigiert werden
 - Der Positionszeiger gibt die (virtuelle) Position des Schreib-/Lesekopfs wieder
 - Der Positionszeiger gibt an, welche Byteposition als nächstes gelesen, bzw. geändert werden kann

- Die Klasse `RandomAccessFile` bietet u.a. die folgenden Methoden

Methode	Wirkung
<code>void seek(long pos)</code>	Positionszeiger auf Byteposition <code>pos</code> setzen (bezogen auf Dateianfang)
<code>int skipBytes(int n)</code>	Von der aktuellen Position wird Positionszeiger um <code>n</code> Bytes verschoben
<code>int read()</code>	liefert das gelesene Byte oder -1 bei Dateiende
<code>void write(int b)</code>	schreibt Byte an die aktuelle Position
<code>long getFilePointer()</code>	Position vom Zeiger
<code>long length()</code>	Länge der Datei

- Die Klasse besitzt die folgenden Konstruktoren
 - `RandomAccessFile(File file, String mode)`
 - `RandomAccessFile(String name, String mode)`
- Das Argument `mode` spezifiziert die Nutzung der Datei
 - `r` : Nur lesen (*read only*)
 - `rw` : Lesen und schreiben (*reading and writing*)
- Das Erzeugen einer Instanz öffnet eine Datei (reserviert die Datei beim Betriebssystem)
- Eine geöffnete Datei muss mit der Methode `close()` wieder geschlossen werden
 - Evtl. vorhandener Puffer-Inhalt wird geschrieben
 - Datei wird freigegeben
- Aufgabe 28: Schreiben Sie eine Methode `static void copy(File from, File to)`, die eine Datei kopiert.

Achtung: kann IOException werfen

Ergänzung zum Thema Exception

- Eine reservierte (geöffnete) Ressource (z.B. eine Datei) muss nach der Nutzung wieder freigegeben (geschlossen) werden
- Die Nutzung von Ressourcen ist in der Regel kritisch
 - Es können Ausnahmen geworfen werden (z.B. `IOException`)
- Das Freigeben eine Ressource erfolgt daher in der Regel in einem `finally`-Block
 - Auch das Freigeben einer Ressource kann evtl. wieder zu einer Ausnahme führen
- Ab Java 7 stellt die `try-with-resources`-Anweisung eine elegantere Möglichkeit bereit, Ressourcen wieder freizugeben

- Es wurde das Interface `AutoCloseable` eingeführt
 - Enthält nur die Methode `void close()`
- Für alle Objekte vom Typ `AutoCloseable`, die im *try-with-resources*-Teil deklariert wurden, wird garantiert die Methode `close()` vor dem Verlassen des `try`-Blocks aufgerufen

- Einfachste Form der *try-with-resources*-Anweisung
 - Resource ist vom Typ `AutoCloseable`

```
try (Resource res = ...){  
    // Nutzung von res  
}
```

- Eine *try-with-resources*-Anweisung kann auch `catch`-Blöcke und einen `finally`-Block besitzen
 - Diese werden nach dem Aufruf von `close()` ausgeführt

- Annahme: Bei der Nutzung einer Ressource in einer *try-with-resources*-Anweisung tritt ein Ausnahme auf. Der anschließende Aufruf von `close()` führt wieder zu einer Ausnahme.
 - Das Werfen der Ausnahme in `close()` wird dann unterdrückt
 - Das in `close()` erzeugte Ausnahmeobjekt wird dann der Ausgangsausnahme als *Suppressed-Exception* hinzugefügt

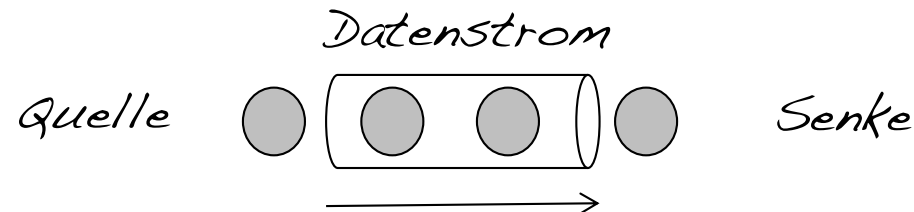
*Abweichend von einer Exception
im finally-Block*

Aufgabe 28b: Implementieren Sie eine Methode `static void cat(File quelle)`! Diese Methode gibt den Inhalt einer Textdatei (ASCII) auf der Konsole aus. Verwenden Sie ein `RandomAccessFile` und eine *try-with-resources*-Anweisung.

Datenströme (Streams)

– Datenstrom (Stream)

- Lineare Folge (Sequenz) von Bytes/Zeichen
- Sequenz kann von beliebiger Länge sein
- Zugriff erfolgt rein sequentiell (FIFO) und unidirektional
- Verbindet eine Datenquelle mit einer Datensenke



– Idee

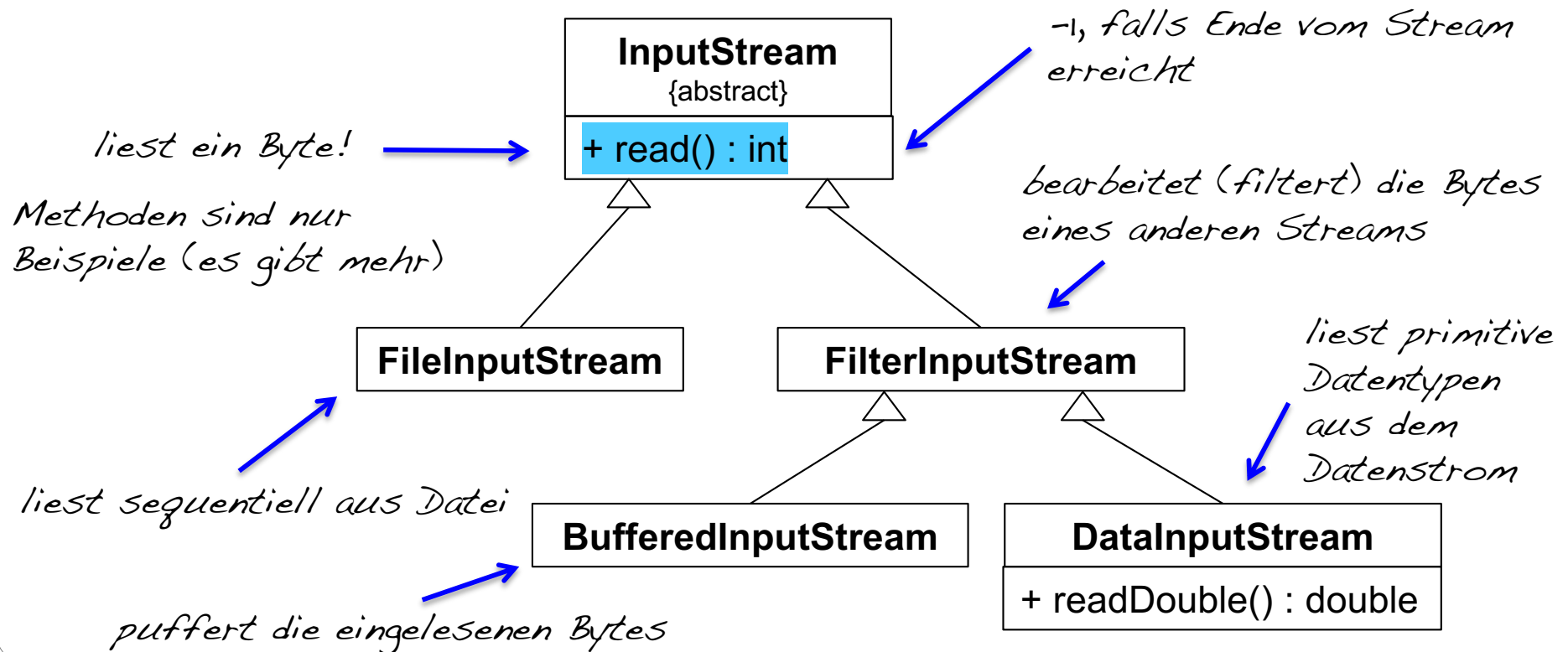
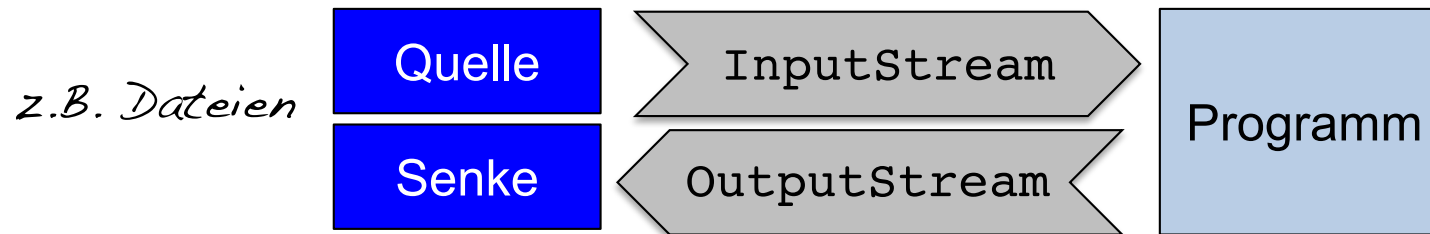
- Über einen Datenstrom findet eine Endkopplung von Datenquelle und Datensenke statt
 - Datenquelle schreibt die Daten in einen (logischen) Datenstrom, unabhängig von einer konkreten Senke
 - Datensenke liest Daten aus einem (logischen) Datenstrom, unabhängig von einer konkreten Quelle

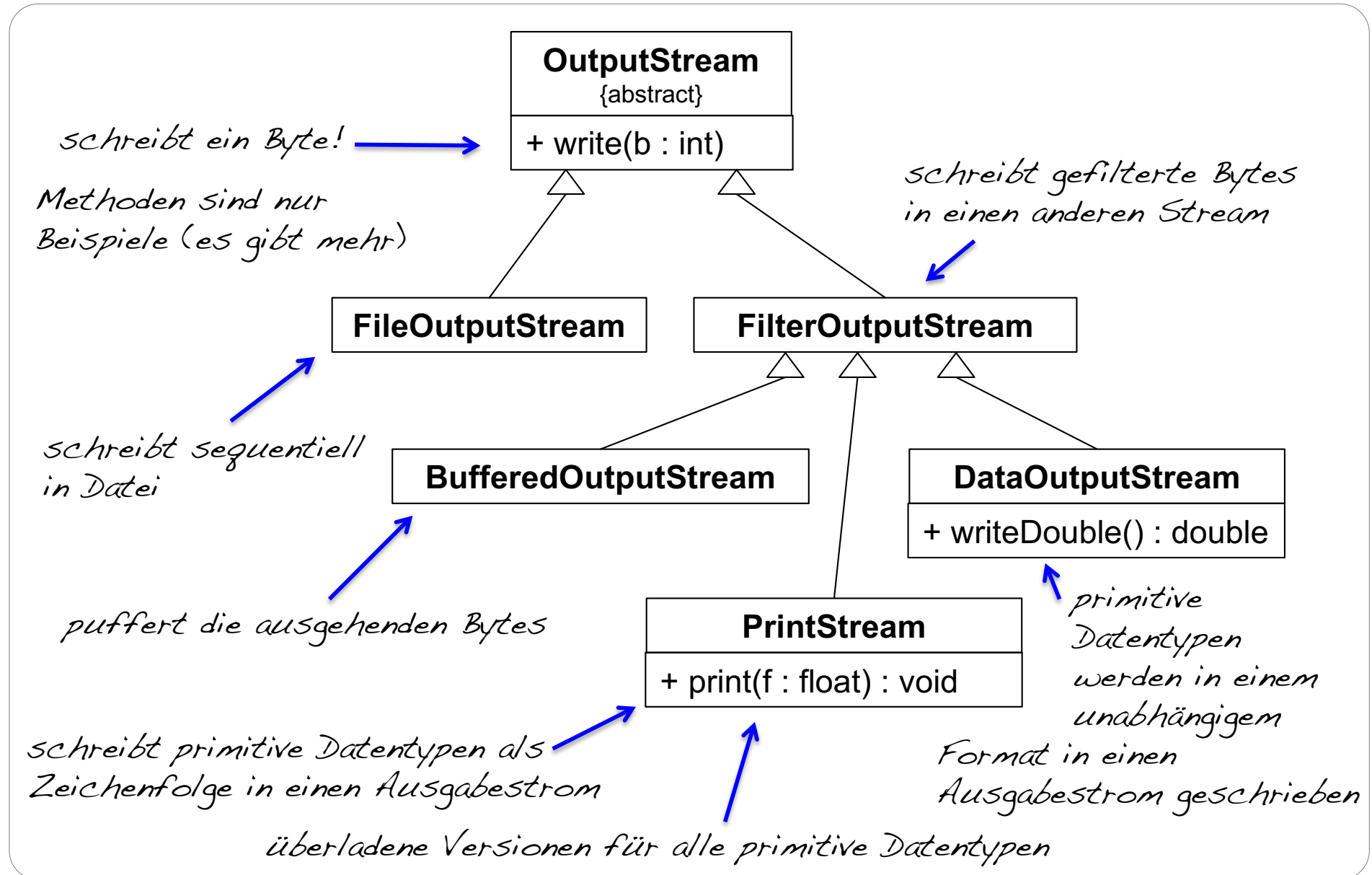
- Analogie: Fernseher (Senke) mit einem HDMI-Anschluss. Durch Umstecken des HDMI-Kabels (Stream) kann ein Receiver (Quelle) einfach durch einen DVD-Player (Quelle) ausgetauscht werden
- Ziel: Flexible Kombination von Streams
 - Austausch der Quelle
 - Austausch der Senke
 - Streams mit Zusatznutzen (z.B. Puffer, Filter)
 - Verketteten von Streams (z.B. Zusammenfassen von Dateien)
- Die folgenden Punkte sind bei der Verwendung von Streams in Java zu beachten
 - Datentypen (primitive- und Referenztypen)
 - Datenformate (Bytes vs Unicode)
 - Datenquellen und –senken mit unterschiedlichen Typen
 - Vor- und Nachbearbeitung (Zusatznutzen)

- Die Stream-Klassen werden in Java nach den folgenden Kriterien unterschieden
 - Datenflussrichtung
 - Eingabe
 - Ausgabe
 - Datentyp
 - byte
 - char
- Aus den möglichen Kombinationen von Datenflussrichtung und Datentyp ergeben sich vier Stream-Hauptklassen

Richtung	Datentyp	Stream-Klasse
In	byte	InputStream
Out	byte	OutputStream
In	char	Reader
Out	char	Writer

Byte-Streams





– Beispiel: Kopieren von Dateien

```
public class Utility {  
  
    public static void copy(File from, File to) {  
        try (InputStream in = new FileInputStream(from);  
            OutputStream out = new FileOutputStream(to)) {  
  
            int c;  
            while ((c = in.read()) != -1) {  
                out.write(c);  
            }  
        } catch (FileNotFoundException e) {  
            // TODO Ausnahmebehandlung  
        } catch (IOException e) {  
            // TODO Ausnahmebehandlung  
        }  
    }  
}
```

*in.close(); und out.close();
werden automatisch aufgerufen*


erzwingt ein flush()

Achtung: auch close() kann eine Exception werfen

Kombination von Streams (Bsp: Puffer)

- Blockorientierte Geräte (z.B. Festplatten) schreiben und lesen die Daten in Datenblöcken
- Um eine bessere Performance zu erlangen, ist es daher sinnvoll, die Daten in einem Stream solange zu puffern, bis ein Datenblock komplett gefüllt werden kann
- Ansatz: Ein `BufferedOutputStream` wird mit einem `FileOutputStream` kombiniert

```
FileOutputStream out = new FileOutputStream(to);  
BufferedOutputStream buffer = new BufferedOutputStream(out);
```

Verkettung von Streams über den Konstruktor 

- Alles, was wir in den Stream `buffer` schreiben, wird gepuffert an den Stream `out` weitergeleitet
- Aufgabe 29: Ergänzen Sie die Methode `copy` um einen Puffer!

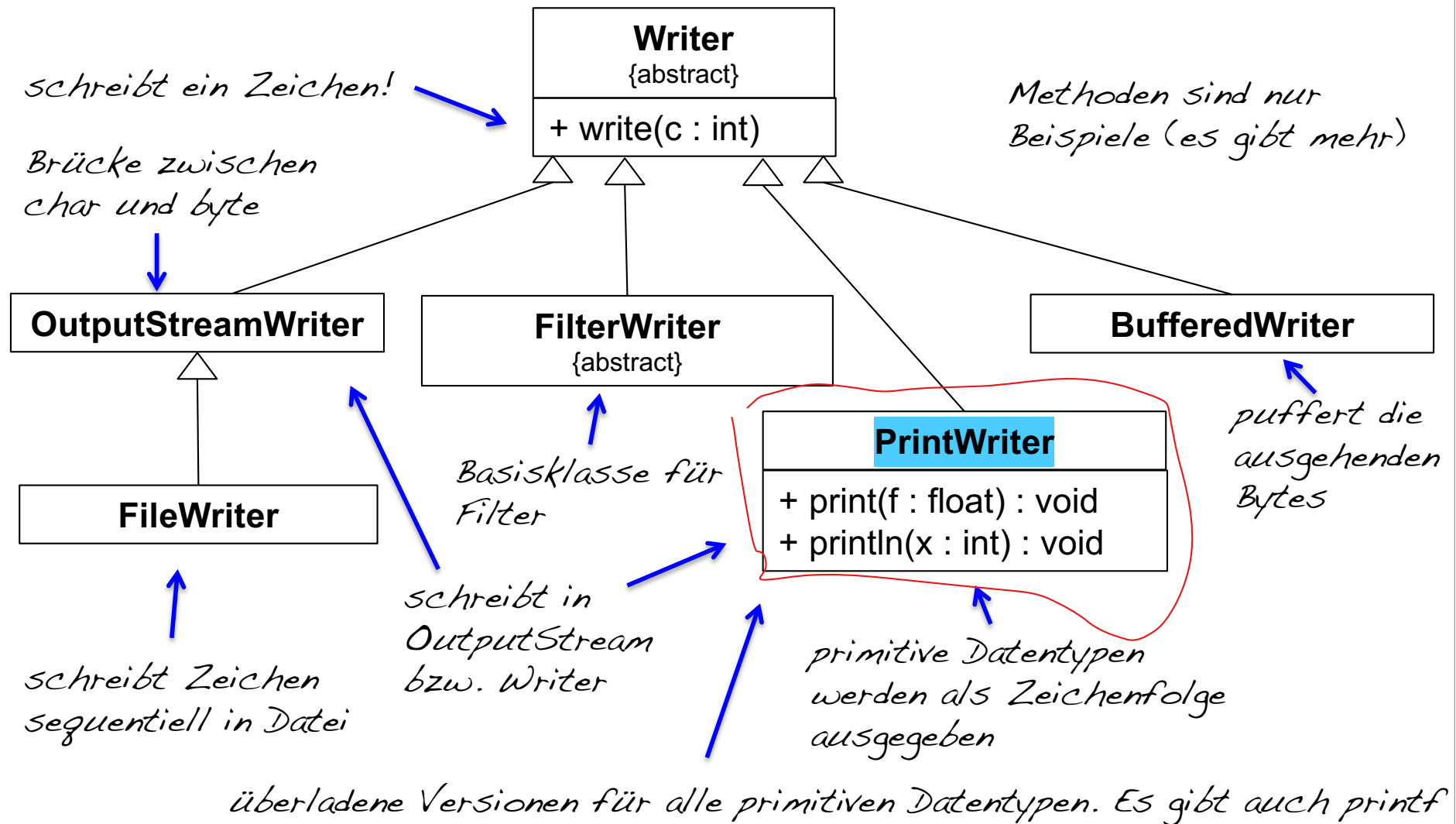
– Bekannte Streams

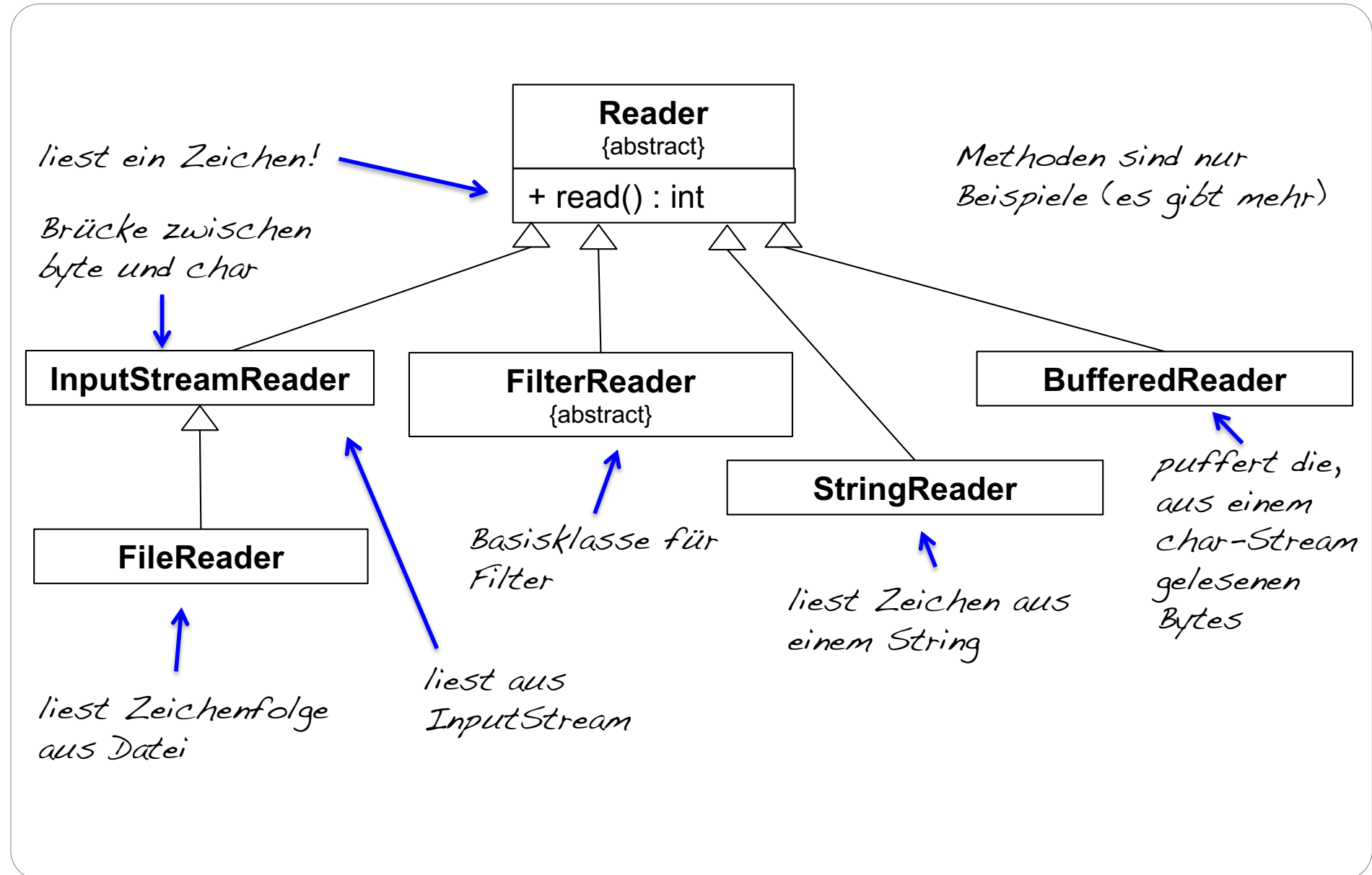
- In der Klasse `System` gibt es die folgenden Klassenattribute

```
public static final PrintStream out;    // Standardausgabe  
public static final InputStream in;    // Standardeingabe
```

- Mit `System.out.printf` wird also über einen speziellen `OutputStream` auf den Bildschirm geschrieben
-
- Wenn wir mit Zeichen arbeiten müssen, dann ist die Verwendung von `Byte-Stream` umständlich
 - Java verwendet die `Unicode-Codierung`
 - Eine Variable vom Typ `char` besteht aus 2 Bytes
 - Für die Verarbeitung von Zeichen gibt es daher spezielle `Char-Streams`

Char-Streams






– Beispiel: Eine universelle Begrüßung

```
public class Utility {  
  
    public static void sayHello(OutputStream s) throws IOException {  
        String t = "Hello World";  
        OutputStreamWriter sw = new OutputStreamWriter(s);  
  
        sw.write(t.toCharArray()); ← überladene Version von write  
        sw.flush(); ← Leeren des Datenstroms wird erzwungen  
    }  
}
```

- Aufgabe 30: Schreiben Sie ein Hauptprogramm. Rufen Sie die Methode `sayHello` zweimal auf. Die Ausgabe soll einmal auf die Konsole, und einmal in eine Datei geschrieben werden.

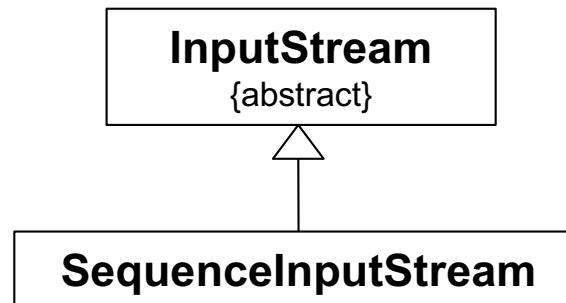
- Aufgabe 31: Schreiben Sie eine statische Methode `void schreiben(Angestellter a, File f)!`
Die Daten des Angestellten `a` sollen in die Datei `f` geschrieben werden. Verwenden Sie dabei das folgende Ausgabeformat:

```
Name: Obermeier  
Gehalt: 10000.00
```



Hinweis: Kombinieren Sie einen `FileWriter` mit einem `PrintWriter`.

SequenceInputStream



- Daten können aus mehreren Streams hintereinander gelesen werden

```
public SequenceInputStream(Enumeration e) ← mehrere Streams
public SequenceInputStream(InputStream s1, InputStream s2)
```

↑
zwei Streams

FilterWriter

- Um einen eigenen `FilterWriter` zu erstellen, sind die folgenden Aktivitäten durchzuführen

- Neue Klasse aus ~~FilterWriter~~ ableiten
- Konstruktor der Superklasse aufrufen, um den Ausgabestrom zu initialisieren
- Die drei `write`-Methoden (siehe API) überschreiben
- Aus der Methode `write(int c)` wird die `write`-Methode der Superklasse aufgerufen

- Aufgabe 32: Schreiben Sie eine Klasse `UpperCaseWriter`. Dieser *Writer* soll alle Zeichen in Großschrift umwandeln. Hinweis:

Verwenden Sie die Methoden

`char Character.toUpperCase(char c)`

und

`char[] toCharArray()` der Klasse `String`

*Testen Sie die Klasse
zusammen mit einem
PrintWriter und einem
FileWriter*


Serialisierung

- Objekte existieren (lokal) im Hauptspeicher (Heap)
- Es gibt Gründe, warum Objekte nicht nur im lokalen Hauptspeicher liegen sollten
 - Persistenz:
 - Objekte sollen auch nach dem Prozessende existieren
 - Die Objekte können z.B. nach einem Neustart wieder eingelesen werden
 - Verteilte Verarbeitung
 - Objekte sollen über das Netzwerk verschickt werden
 - Sie können dann auf entfernten Rechnern aktiv werden
- Serialisierung: Wandelt den Zustand eines Objekts und zusätzliche Statusinformationen in eine Folge von Bytes

– Was wird von einem Objekt serialisiert?

- Klassenname (voll qualifiziert)
- Attribute (Voraussetzung: nichtstatisch, nichttransient)
- Hashwert (Signatur)

elementare Datentypen
Referenztypen




– Was wird nicht serialisiert?

- statische Attribute
- transiente Attribute
- Bytecode

– Deserialisierung

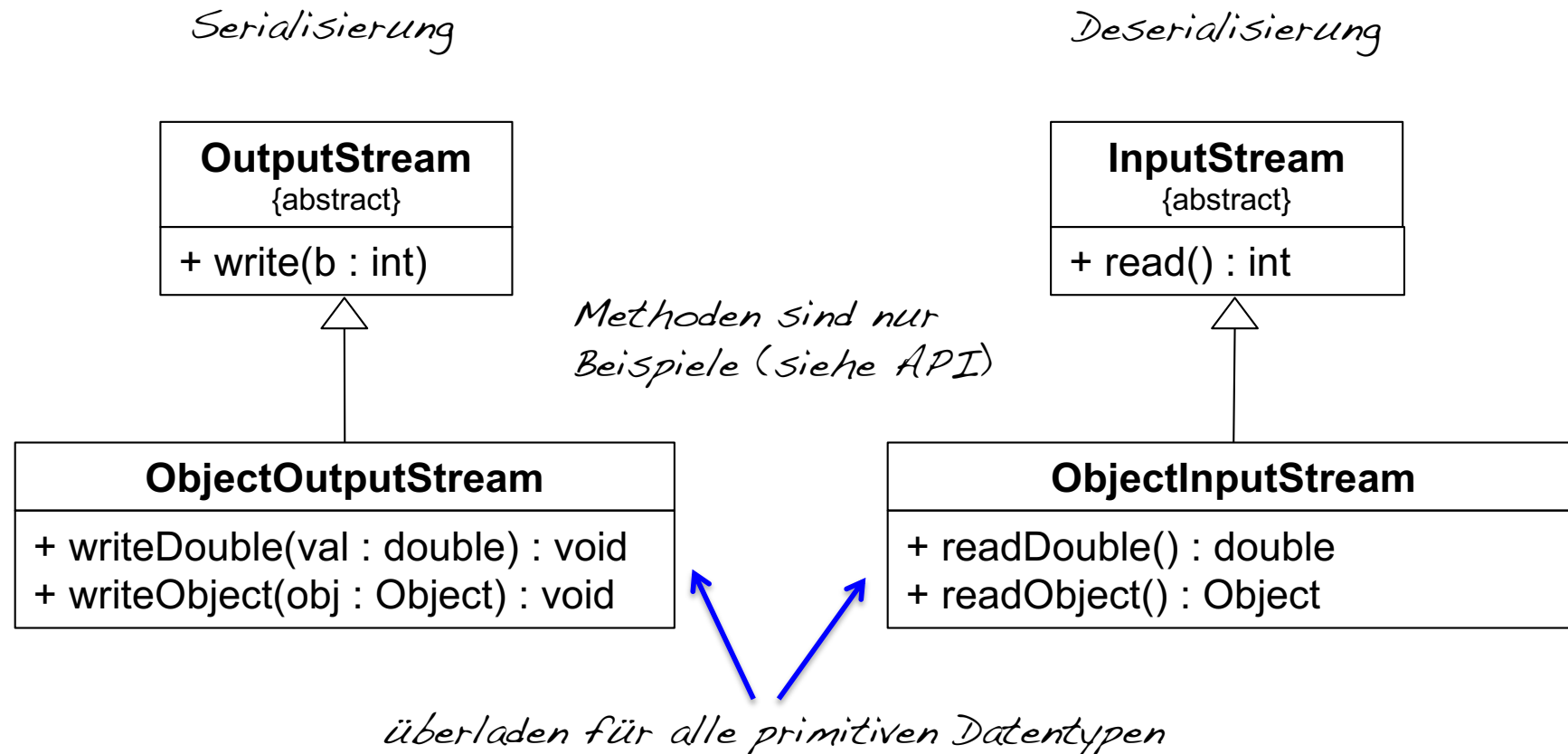
- Umkehrung der Serialisierung
- Folge von Bytes wird in eine Objektinstanz gewandelt

Bytecode der Klasse muss vorhanden sein



- Einige Attribute können nicht sinnvoll serialisiert werden
 - 1) (temporäre) Ressourcen, z.B. Netzwerk- oder Datenbankverbindungen
 - 2) abgeleitete (berechnete) Attribute
- Attribute, die mit dem Schlüsselwort `transient` definiert sind, werden nicht serialisiert
- Aufgabe 32-2: Warum werden statische Attribute nicht serialisiert?

– Die Serialisierung und Deserialisierung erfolgt über Datenströme



- Die Methode writeDouble serialisiert nur ein Attributwert
- Die Methode writeObject serialisiert ein „komplettes“ Objekt

- Mit der Methode `writeObject` der Klasse `ObjectOutputStream` können nur Objekte serialisiert werden, deren Klasse die Schnittstelle `java.io.Serializable` implementiert
 - Die Schnittstelle `Serializable` enthält keine Methoden (die Schnittstelle dient nur als Markierung)
- Aufgabe 33: Welche Bestandteile der folgenden Klasse werden von der Methode `writeObject` serialisiert?

```
public class A implements Serializable{  
    private String name; ✓  
    private transient RandomAccessFile raf; ✗  
    protected static int id; ✗  
  
    public A(String name, RandomAccessFile raf){  
        this.name = name;  
        this.raf = raf;  
        id = 0;  
    }  
}
```

– Beispiel: Serialisierung einer Instanz der Klasse A

```
A a = new A("Testklasse", new RandomAccessFile(
    new File("test.txt"), "rw"));

File sfile = new File("/Users/dwiesmann/IO/a.ser"); ← Zieldatei

try (FileOutputStream fos = new FileOutputStream(sfile);
    ObjectOutputStream oos = new ObjectOutputStream(fos)) {

    oos.writeObject(a);

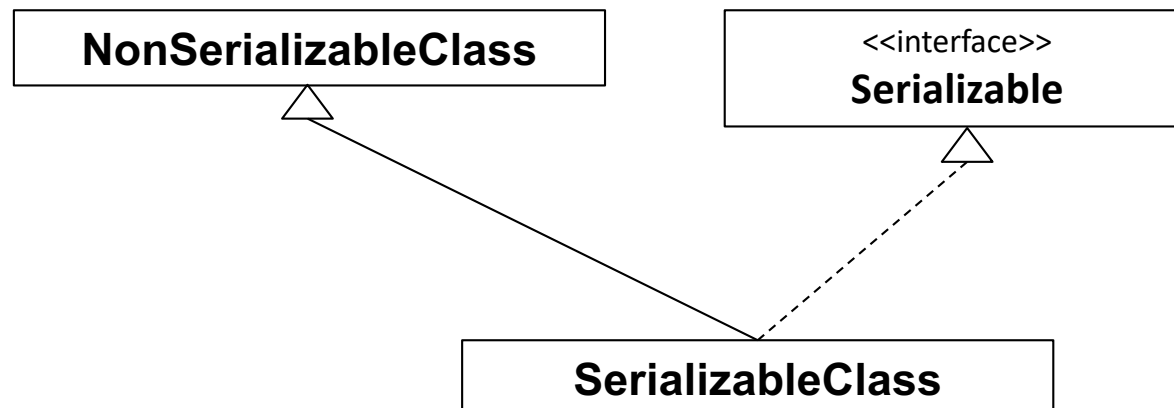
} catch (IOException e) {
    System.out.println("Fehler bei der Serialisierung");
    // TODO Ausnahmebehandlung
}
```

Serialisierung (points to `oos.writeObject(a);`)

*Verkettung des
ObjectOutputStream
mit einem
FileOutputStream* (points to the constructor of `ObjectOutputStream`)

Deserialisierung

- Die Deserialisierung erfolgt in der gleichen Reihenfolge wie die Serialisierung
- Bei der Deserialisierung wird der Konstruktor des erzeugten Objekts nicht aufgerufen
 - Ausnahme: Für eine nicht-serialisierbare Oberklasse wird der Standardkonstruktor aufgerufen



– Beispiel: Deserialisierung einer Instanz der Klasse A

```
File sfile = new File("/Users/dwiesmann/IO/a.ser");  
A a = null;
```

*↑
Datei mit der
serialisierten Instanz*

```
try (FileInputStream fis = new FileInputStream(sfile);  
    ObjectInputStream ois = new ObjectInputStream(fis)) {
```

```
    a = (A) ois.readObject();
```

*↑
Deserialisierung*

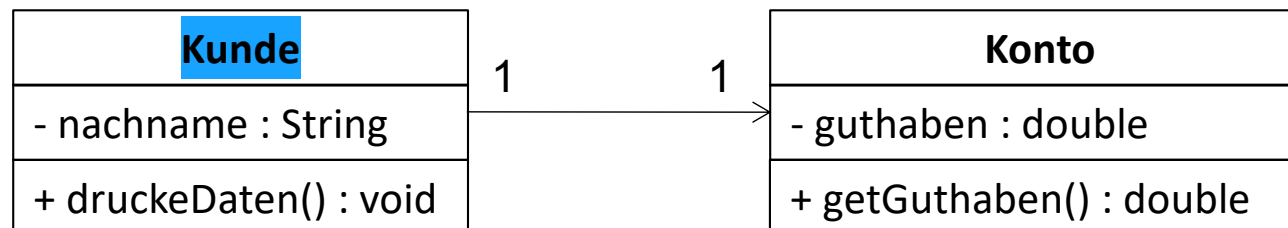
*↑
Verkettung des
ObjectInputStream
mit einem
FileInputStream*

```
    } catch (IOException e) {  
        System.out.println("IO-Fehler bei der Deserialisierung");  
        // TODO Ausnahmebehandlung  
    } catch (ClassNotFoundException e) {  
        System.out.println("Fehler: class-Datei nicht gefunden");  
        // TODO Ausnahmebehandlung  
    }  
}
```

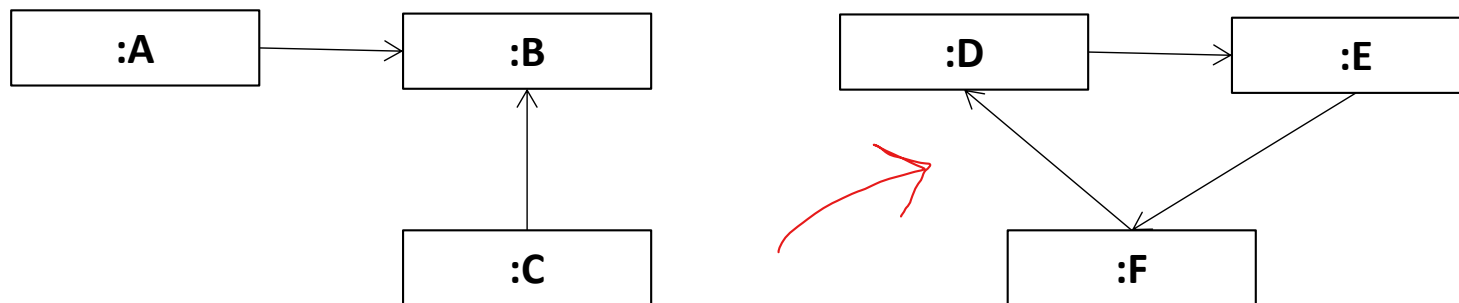
*↑
A.class muss vorliegen und zur
serialisierten Instanz passen*

Serialisierung von Referenztypen

- Ein serialisierbares Attribut kann eine Referenz auf eine andere Objektinstanz sein
 - Das referenzierte Objekt wird dann automatisch serialisiert (Klasse muss `Serializable` implementieren)
 - Die automatische Serialisierung und Deserialisierung kann sich (rekursiv) über beliebig viele Stufen erstrecken
- Aufgabe 34: Erzeugen Sie einen Kunden mit einem Konto. Serialisieren Sie den Kunden. In einem weiteren Programm deserialisieren Sie die Kundeninstanz und lassen sich die Kontoinformationen ausgeben.



- Hinweis: Die Containerklassen `ArrayList` und `LinkedList` implementieren die Schnittstelle `Serializable`
- Die Serialisierung und Deserialisierung von Referenztypen ist eine komplexe Aufgabe
- Aufgabe 35: Welches Verhalten darf bei einer Serialisierung in den folgenden beiden Fällen nicht auftreten?



Objektdiagramme

- Um jede Objektinstanz nur einmal zu serialisieren, wird vom Serialisierungsmechanismus intern eine Hash-Tabelle aufgebaut
 - Jede bereits serialisierte Instanz wird dort mit einer entsprechenden Referenz vermerkt
 - Da Referenzen in der Hashtabelle gespeichert sind, kann der Garbage-Collector Objektinstanzen nicht löschen, obwohl sie fachlich evtl. gar nicht mehr referenziert werden
 - Dies kann zu Speicherplatzproblemen führen
 - Änderungen an bereits serialisierten Objekten werden zudem nicht gespeichert, solange die Objekte noch in der Hashtabelle gespeichert sind
 - Lösung: Die Methode `reset()` der Klasse `ObjectOutputStream` löscht die interne Hashtabelle

Wichtig

Versionsnummern für Klassen

– Situation

- Eine Klasse `Bestellung` enthält ein Attribut `private String datum;`
- Es werden Instanzen der Klasse `Bestellung` serialisiert
- Danach wird in der Klasse `Bestellung` der Typ des Attributs `datum` von `String` auf `Calendar` geändert
- Die vorher serialisierten Instanzen sollen nun wieder deserialisiert werden

Fehler: Bytefolge nicht mehr konsistent zur class-Datei

- Wenn serialisierte Instanzen nicht mehr konsistent zu der aktuellen `class`-Datei sind, ist eine erfolgreiche Deserialisierung nicht gewährleistet

„Versionsnummer“

- Für jede Klasse wird daher ein Hashwert (Signatur) berechnet
 - Der Hashwert der Klasse wird serialisiert
 - Bei der Deserialisierung müssen die Hashwerte übereinstimmen

- Falls die Versionsnummer der Klasse nicht zur Versionsnummer der serialisierten Instanz passt, wird eine `InvalidClassException` geworfen
- Für die Berechnung der Versionsnummer wird ein Hashwert über die Klasse gebildet
- Dabei werden die folgenden Daten zu einem Zahlenwert vom Typ `long` verrechnet
 - Klassenname
 - Methodensignaturen
 - Attribute
 - Implementierte Schnittstellen
- Eine serialisierbare Klasse wird automatisch um die folgende Konstante ergänzt

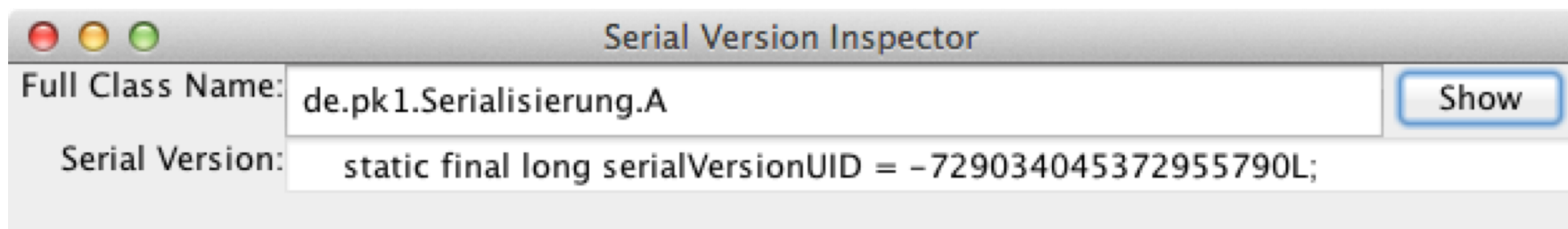
```
static final long serialVersionUID;
```

- Mit dem Werkzeug serialver kann die serialVersionUID einer Klasse berechnet werden

```
> serialver de.pk1.Serialisierung.A
de.pk1.Serialisierung.A:    static final long
serialVersionUID = -729034045372955790L;
>
```


- Analog kann mit der Option -show ein interaktives Fenster geöffnet werden

```
> serialver -show
```



- **Nachteil einer automatischen Berechnung der Versionsnummer:**
 - Das Hinzufügen einer Methode ändert z.B. die Versionsnummer der Klasse
 - Eine Deserialisierung von Instanzen ist dann nicht mehr möglich, obwohl die zusätzliche Methode die Deserialisierung eigentlich nicht beeinträchtigt
- **Lösung:** Die `serialVersionUID` kann manuell vergeben werden

```
public class A implements Serializable{  
    private String name;  
    private transient RandomAccessFile raf;  
    protected static int id;  
  
    private static final long serialVersionUID =  
                                                -729034045372955790L;  
    //  
}
```

 *es erfolgt keine automatische Berechnung mehr*

- Bei den folgenden Modifikationen muss die `serialVersionUID` in der Regel nicht geändert werden
 - Hinzufügen und Entfernen von Methoden
 - Entfernen vom Attributen
 - Hinzufügen von Attributen (sind nach der Deserialisierung nicht initialisiert)

- Folgende Modifikationen erfordern dagegen eine Änderung der `serialVersionUID`
 - Umbenennen von Attributen
 - Definition von Attributen als `transient` (oder umgekehrt)
 - Definition von Attributen als `static` (oder umgekehrt)