Skript
Informatik B

Objektorientierte Programmierung in Java

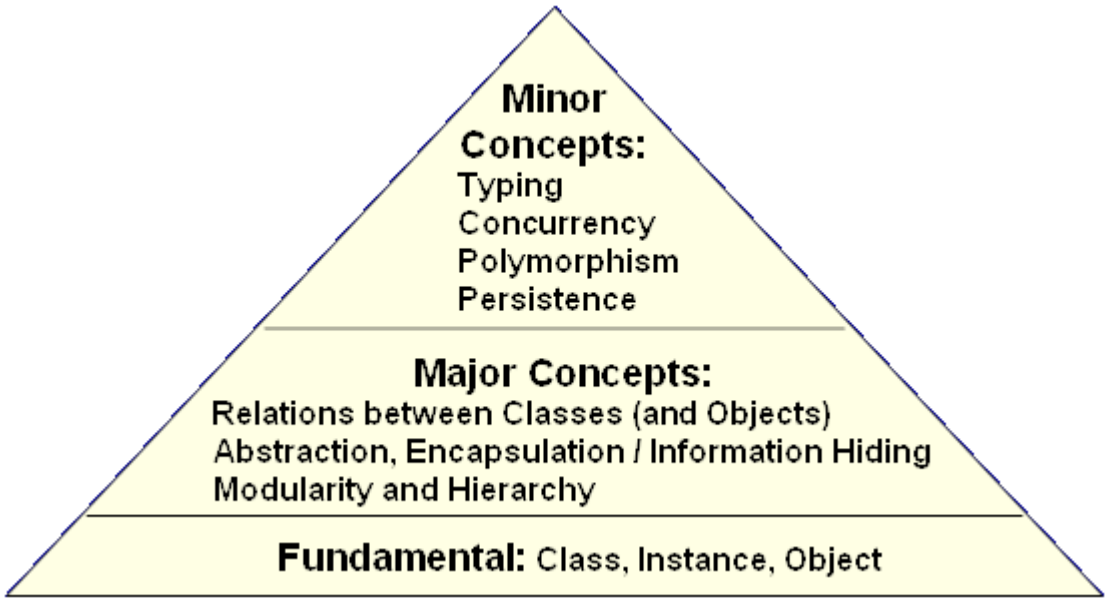
Sommersemester 2011

- Teil 6 -

Inhalt

… wird schrittweise erweitert

Kapitel 8:
Persistenz



[Boo94]

8.1 Grundlagen der Persistenz

Objekte werden normalerweise mit dem Programmende automatisch zerstört.

Ziel:
- Objekte und ihre Beziehungen sollen über den aktuellen Programmlauf hinaus aufbewahrt werden.
- Langfristige (persistente) Speicherung von Klassen und Objekten mit ihren Strukturen, Zuständen, Beziehungen auf einem nicht flüchtigen Medium (z.B. Festplatte).
- Es ist möglich, aus den langfristig gespeicherten Daten wieder einen analogen Arbeitsspeicherzustand wie vor der Speicherung herzustellen.

Persistente vs. transiente Objekte:
- Persistent (lat.): anhaltend
  In der Programmierung: Das Objekt überdauert sein Programm.
- Transient (lat.): vorübergehend
  In der Programmierung: Das Objekt ist flüchtig und überlebt das Programm nicht.

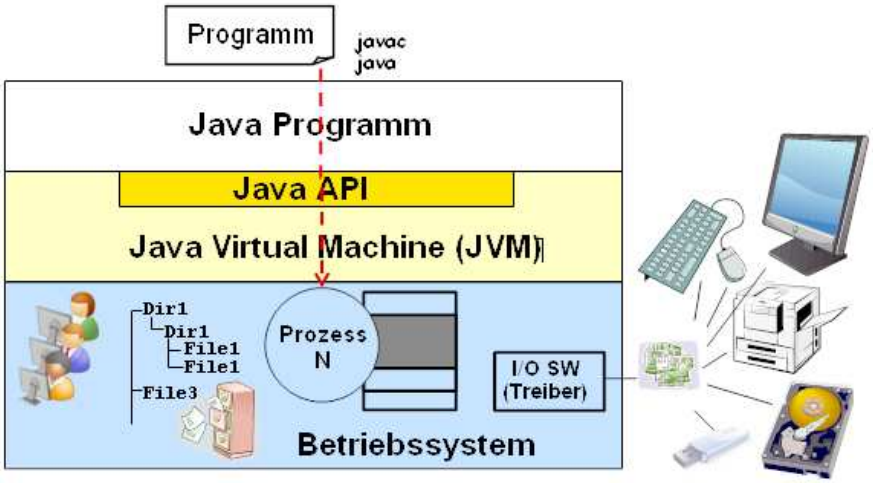
Passive vs. aktive Objekte:
- Passives Objekt: Das Objekt ist im laufenden Programm vorhanden, aber nur in seiner gespeicherten Form.
- Aktives Objekt: Das Objekt ist als laufende Instanz im Programm (und im Hauptspeicher) vorhanden.

Langfristige Speicherung mit:
- Dateien: Nutzung des Dateisystems (verwaltet vom Betriebssystem).
- Datenbanken bzw. Datenbankmanagementsystemen: Spezielles Systemprogramm, das die persistente Speicherung unterstützt (umfangreiche Datenmengen, viele Benutzer).

Hilfsmittel auf Programmierebene:
- Dateien und Verzeichnisse (engl. Files, Directories)
- Ein-/Ausgabeströme (engl. Streams)
- Serialisierung
- Datenbankanbindung (in Form einer API = Application Programming Interface)

Diese Hilfsmittel sind nicht nur zum Speichern von Objekten geeignet.

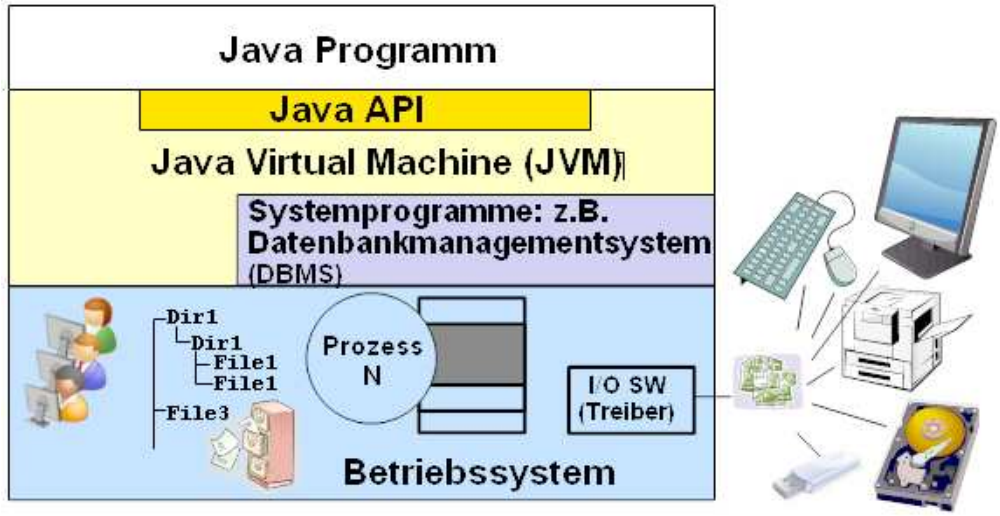
Zur Erreichung von Persistenz ist eine enge Zusammenarbeit mit dem Betriebssystem erforderlich.



Das Betriebssystem hat folgende Aufgaben:
- Benutzerverwaltung und Zugriffsrechteverwaltung,
- Prozessverwaltung und Interprozesskommunikation,
- Dateiverwaltung,
- Speicherverwaltung,
- I/O Steuerung (I/O = Input/Output = Eingabe/Ausgabe).

System in Schichten: Abstraktion von der Plattform
Die (sehr häufig zu findende) Einteilung des Gesamtsystems in Schichten haben wir für Java bereits vorn gesehen.
Sogenannte Systemprogramme laufen direkt über dem Betriebssystem.
Systemprogramme sind z.B. der Editor vi oder auch Datenbankmanagementsysteme.
Bei den Datenbankmanagementsystemen gibt es verschiedene Arten. Die verbreitetsten Datenbankmanagementsysteme sind relational. Alle Daten (d.h. im OO-Programm auch die Objekte) werden in ihnen in Form von Tabellen gespeichert. Diese können mit einer eigenen (bequemen) Sprache SQL (Structured Query Language) nach verschiedenen Kriterien ausgelesen bzw. verwaltet werden. Neben relationalen Datenbankmanagementsystemen existieren unter anderem auch objektorientierte. In der OO-Welt entfällt damit eine Umwandlung von Objekt zu Tabelle. Die Objekte können darin direkt abgelegt werden.

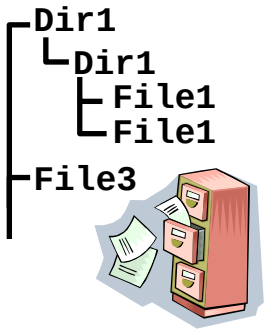
Begriff *Plattform* (engl. Platform): Eine Plattform bezeichnet „alles darunterliegende“. In der Regel (z.B. aus Sicht der Java-Programmiererin) ist das Betriebssystem eine Plattform (gegebenenfalls auch noch weitere, darüber liegende Schichten).



Schwierigkeit: Plattformunabhängigkeit
Java tut sich mit I/O und anderen systemnahen Aufgaben schwer, weil grundsätzlich das Konzept der Abstraktion verfolgt wird. Abstraktion ist aber bei solchen systemnahen Aufgaben nicht mehr vollständig durchzuhalten. Stattdessen werden teils Plattformdetails bis in die oberen Schichten wichtig (z.B. Rechteverwaltung).
Aus diesem Grund ändert sich an diesen Teilen von Java von Version zu Version häufiger etwas. Auch für Java 7 wird es hier wieder einige Erweiterungen geben.
Systemnahen Aufgaben sind oft besser bei systemnahen Programmiersprachen (z.B. C) aufgehoben.

8.2 Datei-/Verzeichnisverwaltung

Es gibt eine Vielzahl von Aufgaben für die Verwaltung von Dateien und Verzeichnissen:



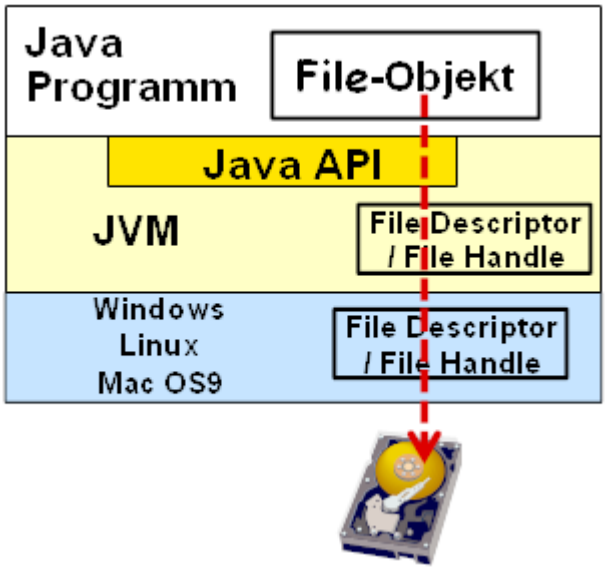
- Dateien und Verzeichnisse anlegen, löschen, umbenennen, verschieben, filtern/suchen/auflisten,
- Datei-/Verzeichnis-Attribute/Eigenschaften abfragen und setzen (z.B. Rechte),
- Dateien und Verzeichnisse auslesen, schreiben.

Zur Umsetzung muss der Compiler und das Laufzeitsystem (in Java: JVM) mit der Plattform eng zusammenarbeiten. Die Plattformunabhängigkeit ist dadurch teilweise nicht machbar.

8.2.1 Dateien und Verzeichnisse in Java: File

Wir haben bereits zu Beginn gesehen, dass große Systeme häufig in Schichten eingeteilt sind. Daraus folgt: Plattformabhängige Konzepte (z.B. Dateien) müssen in jeder Schicht mit einer entsprechenden Verwaltungsstruktur repräsentiert werden.
Sprachentwickler bemühen sich, das jeweilige Konzept mit einer Repräsentation in den höheren Schichten so weit wie möglich plattformunabhängig zu gestalten.

Umsetzung in Java:
Java bietet eine Menge von Klassen an (Paket java.io), die für I/O-Aktivitäten (z.B. Umgang mit Dateinamen) verwendet werden können.



- File-Objekt für plattformunabhängige Dateioperationen
- File-Objekt repräsentiert eine Datei oder ein Verzeichnis im Dateisystem (Verweis auf einen absoluten oder relativen Pfadnamen)
- Ein File-Objekt wird auf eine Datenstruktur in der JVM abgebildet. Diese Datenstruktur wird wiederum auf eine im Betriebssystem abgebildet. Auf Betriebssystemebene heißt die Datenstruktur *Datei Deskriptor* bzw. *File Deskriptor* (engl. File Descriptor oder File Handle). Das entspricht einer Referenz auf die Datei.

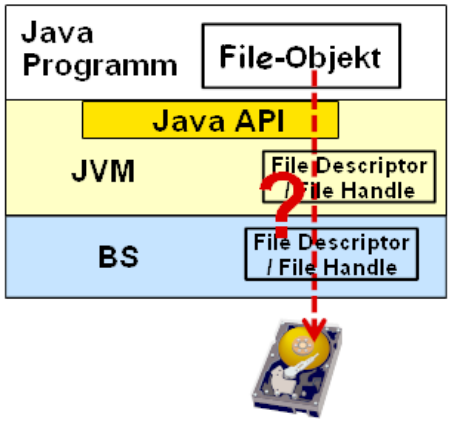
Erzeugen eines File-Objekts: aus einem Dateinamen

- File(String pathname)
- File(String parent, String child)
- File(URI uri)

Achtung: Plattformabhängigkeiten
Beispiele für das Problem der Plattformvarianzen für ein Konzept:

- Rechteverwaltung (Die Betriebssysteme Windows, Linux, Mac OS haben alle unterschiedliche Ansätze, wie die Rechteverwaltung aussieht).
- Pfadtrenner im Dateinamen
  Windows: Backslash ("\")
  Linux/Unix: Slash ("/")
  Abhilfe: Die Klasse File speichert in der Klassenvariable separatorChar den für die jeweilige Plattform richtigen Pfadtrenner.
- Darstellung des Wurzelverzeichnisses
  Windows: Laufwerk:Backslash (z.B. "Z:\")
  Linux/Unix: Slash ("/")
  Abhilfe: Die Klasse File gibt mit der Klassenmethode File.listRoots() die Wurzelverzeichnisse aus.

File-Objekte sind "immutable", stehen unveränderlich nur für eine Datei.
Ein File-Objekt muss nicht unbedingt eine existierende Datei oder ein existierendes Verzeichnis repräsentieren.
Ändert sich der Dateiname, ist das File-Objekt ungültig.
Auf das File-Objekt kann zwar noch zugegriffen werden, die Anfragen liefern aber unsinnige Ergebnisse (d.h. es wird bei einem ungültigen Zugriff insbesondere auch keine Exception geworfen!).



Anfragen an das File-Objekt:

- exists(): Test, ob die Datei oder das Verzeichnis tatsächlich im Dateisystem vorhanden ist.
- isFile(): Test, ob das File-Objekt eine Datei repräsentiert.
- isDirectory(): Test, ob das File-Objekt ein Verzeichnis repräsentiert.

8.2.2 Anlegen, Löschen, Umbenennen, Auflisten mit File

Im Folgenden sind exemplarisch einige wichtige Operationen aufgeführt.
Die offizielle Java API liefert die in der aktuellen Java-Version gültigen Details zu allen möglichen Operationen.

1) Neue, leere Datei anlegen (mit dem im File-Objekt gespeicherten Namen oder einem temporären Namen):

- boolean createNewFile() throws IOException
  Legt eine neue, leere Datei mit dem im File-Objekt gespeicherten Namen an, insofern eine Datei mit diesem Namen noch nicht existiert.
- static File createTempFile(String prefix, String suffix) throws IOException
  Legt eine neue Datei im temporären Verzeichnis an
  Der Dateiname setzt sich aus einem benutzerdefinierten Präfix, einer Zufallsfolge und einem Suffix zusammen.
- static File createTempFile(String prefix, String suffix, File directory) throws IOException
  Legt eine neue Datei im gewünschten Verzeichnis an.

Der Dateiname setzt sich aus einem benutzerdefinierten Präfix, einer Zufallsfolge und einem Suffix zusammen.

2) Dateien und leere Verzeichnisse löschen:

- boolean delete()
  Löscht die Datei oder das leere Verzeichnis.
- void deleteOnExit()
  Löscht die Datei/das Verzeichnis, wenn die virtuelle Maschine korrekt beendet wird. Einmal vorgeschlagen, kann das Löschen nicht mehr rückgängig gemacht werden. Falls die JVM fehlerhaft beendet wird (z.B. Stromausfall), kann die Datei möglicherweise noch vorhanden sein, was insbesondere für temporäre Dateien lästig ist.

3) Verzeichnisse anlegen und auflisten:
Ein Verzeichnis kann reine Dateien oder selbst auch wieder Unterverzeichnisse besitzen.
mkdir(), mkdirs(), list(), listFiles()

4) Umbenennen und Verschieben:
renameTo(File d):Umbenennung der Datei in den Namen, der durch das File-Objekt d gegeben ist.
Verschieben wird durch Umbenennen realisiert (mit Einschränkung: Verschieben ist nur auf demselben Datenträger möglich).
Achtung: File-Objekte sind "immutable" (ändert sich der Dateiname, ist das File-Objekt ungültig bzw. weitere Zugriffe sind ungültig).

5) Rechte zum Lesen, Schreiben, Ausführen:

boolean canRead(), canWrite(), canExecute()
boolean setReadOnly(), setWritable ...

6) Letztes Änderungsdatum (gemessen in Millisekunden relativ zur Referenzzeit: 01.01.1970 00:00:00 UTC):
long lastModified(), boolean setLastModified(long time)

7) Länge der Datei in Bytes: long length()

8) Ermittlung des freien Plattenspeichers:

getFreeSpace(), getUsableSpace(), getTotalSpace()

Bemerkung:

- Ein Programmierer sollte nicht versuchen, eine API auswendig zu lernen (Schade um die Zeit!).
  Sinnvoll ist es, sich einige zentrale Klassen oder Operationen zu merken und vor allem die wichtigen grundsätzlichen Möglichkeiten. Wer die Möglichkeiten kennt, hat bei der Lösungssuche für Programmierprobleme Vorteile.
  Bei Bedarf schaut der Programmierer in der aktuellen API die Details nach, da sich APIs auch immer wieder ändern.
- Für I/O-Operationen müssen Exceptions beachtet und durch einen entsprechenden try-catch-Block aufgefangen werden.

8.3 I/O-Streams

- Strom (engl. Stream): Ein Stream ist eine geordnete Folge von Daten, die eine Quelle und eine Senke haben. In der Programmierung bilden sie eine Schnittstelle eines Programms nach außen zur Ein-/Ausgabe von Daten. Sie sind ein Hilfsmittel zum Bewegen von Daten.
- Einsatz: Streams werden beispielsweise für den Zugriff auf Dateiinhalte benötigt.

- Streams sind in der Regel unidirektional (d.h. die Daten werden nur in eine Richtung übertragen: Ein oder Aus).
- Ein Stream fungiert als eine Art Pipeline nach dem Prinzip einer Warteschlange.
- Streams sind nicht gebunden an einen speziellen Typ für das Ein-/Ausgabeobjekt:

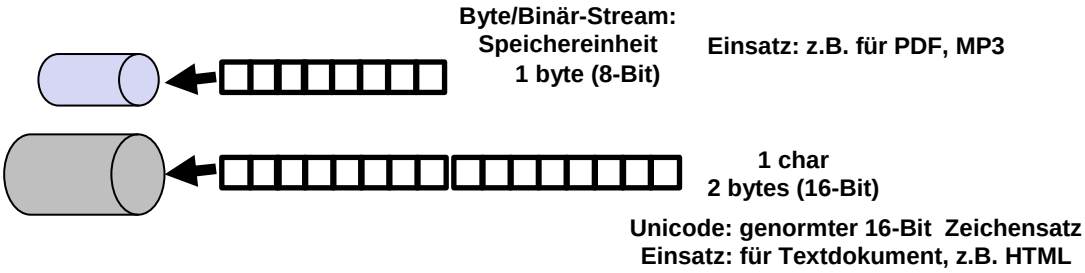
  ⇒ Einheitliche Modellierung der Datenübermittlung unabhängig von der Art der Quelle oder Senke (z.B. Hauptspeicher, Internet, Drucker).
  ⇒ Beliebige Schachtelung/Verkettung möglich, s. unten, Abschnitt „Verknüpfen von Streams“.

- In Java: Paket `java.io` mit über 30 Klassen für die Ein-/Ausgabeverarbeitung (Klassen für Streams und Hilfsklassen)
- Nutzung durch den Entwickler: Objekt der benötigten Stream-Klasse erzeugen und vordefinierte Operationen der Streams für die Ein-/Ausgabe (entsprechend API) nutzen.

Bemerkung:
Streams sind ein allgemeines I/O-Konzept. Es findet sich also nicht nur in Java. Die Macher des Betriebssystems Unix haben Streams „erfunden“. Die Idee war die Schaffung eines abstrakten Geräts (engl. Device), so dass nicht mehr jedesmal beim Programmieren die Hardware-Details der Devices als Wissen notwendig sind. Das Betriebssystem kapselt und versteckt die Ansteuerungsdetails der Hardware-Geräte vor dem Programmierer. Der Treiber des jeweiligen Geräts wird mit den Standardoperationen (read, write) angesprochen und der Treiber setzt diese dann auf das spezifische Gerät um. Aus Sicht der Programmierenden werden alle I/O-Operationen einheitlich wie ein I/O auf eine Datei gesehen.
Ein Treiber ist ein Codestück, das eine spezielle Hardware ansteuern kann.

8.3.1 Arten von Streams in Java

- Eingabe-Stream (Einlesen aus Datenquelle) versus Ausgabe-Stream (Ausgabe in Datenspeicher), engl. Input-Stream, Output-Stream
  Bemerkung: Was Ein- und was Ausgabe ist, ist eine Frage des Standpunkts.
- Byte/Binär-Streams versus Unicode/Zeichen-Streams



Ohne Zeichen-Streams müsste man von Hand die 2 Bytes in zwei einzelne Bytes transformieren und beim Herauslesen darauf achten, wie die beiden Bytes eingeschrieben wurden, d.h. man müsste eine manuelle Transformation von einem größeren Typen in zwei kleinere machen und umgekehrt. Das ist fehleranfällig.
Durch die Verwendung des geeigneten Streams (Unicode-Stream) bekommt die Programmiererin von der Konvertierung Unicode nach Byte nichts mit.

Bemerkung:
Während Ein-/Ausgabe-Streams ein allgemeines Konzept sind, ist die Unterscheidung in Binär- und Zeichenstrom Java-spezifisch. Andere Sprachen unterscheiden hier eventuell anders bzw. bieten andere grundlegende, benutzerfreundliche Streams an.

8.3.2 Java Stream-Klassen: lesen, schreiben

Die Stream-Basisklassen haben weitgehend identische Operationen für das Einlesen von Daten und für die Manipulation der Verbindung zur Datenquelle.
Beim `InputStream` finden sich zusätzliche Operationen wie z.B. `skip(n)` oder `available()`.

`available()`: Gibt die Zahl der verfügbaren Zeichen zurück, die im Datenstrom sofort ohne Blockierung gelesen werden können.

	Bytes (oder Byte-Arrays)	Zeichen (char oder Zeichen-Arrays, Strings)
aus Quelle lesen	InputStream	Reader
in Senke schreiben	OutputStream	Writer

Speziell für Dateien existieren speziellere Subklassen der allgemeineren Stream-Klassen:

	Bytes (oder Byte-Arrays)	Zeichen (char oder Zeichen-Arrays, Strings)
aus Dateien lesen	FileInputStream	FileReader
in Dateien schreiben	FileOutputStream	FileWriter

FileWriter und FileReader erlauben eine zeichenbasierte Verarbeitung (z.B. Auslesen von Zeichen oder Strings).
Will man eine Datei kopieren sollte man keinen Reader verwenden, sondern byteweise arbeiten. Eine Zeichenrepräsentation der Dateiinhalte ist nicht relevant (der lesende Mensch ist nicht involviert).

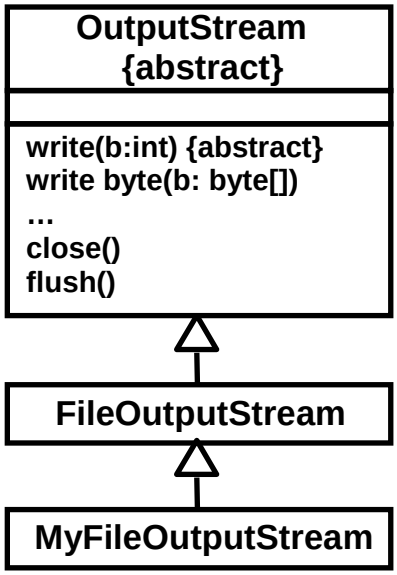
Blöcke:
Weil Dateien in Blöcken (bestehend z.B. aus 1024 Bytes) angeordnet sind, ist ein „Byte-für-Byte-Lesen“ (oder –Schreiben) nicht sehr effizient. Wird auf eine Datei über das Betriebssystem zugegriffen, wird immer auf einen ganzen Block der Datei zugegriffen (blockweises Lesen und Schreiben). Dadurch sind weniger Lese-/Schreibaufrufe an das externe Gerät (z.B. Festplatte) notwendig. Aus einem gelesenen Block werden die benötigten Daten selektiert (falls nicht der gesamte Block benötigt wird). Wie groß ein Block einer Datei ist, hängt vom Betriebssystem (und dessen Konfiguration) ab.
Die Streams sollten daher effizient so verwendet werden, dass sie die Daten in geeigneten Byte-Blockgrößen lesen und schreiben. Auf Betriebssystemebene ist ein solches blockweises Lesen und Schreiben schneller. Das Kopieren einer Datei Byte für Byte dauert demgegenüber relativ lang, da jedes Byte einzeln ausgelesen und geschrieben werden muss.

Bemerkung:
Die Blockgröße einer Datei ist plattformspezifisch. Durch Abstraktion sollte der Programmierer sich darum prinzipiell nicht kümmern müssen.
An der Blockgröße kann man wieder einmal gut erkennen, wie plattformspezifisches Wissen von Zeit zu Zeit eine fundiert ausgebildete Entwicklerin vom „Hobby-Programmierer“ unterscheidet. Ein Programmierer, der also behauptet „brauche ich nicht“ hat sich damit selbst der entsprechenden Kategorie zugeordnet.

Aufbau der Stream-Klassen:

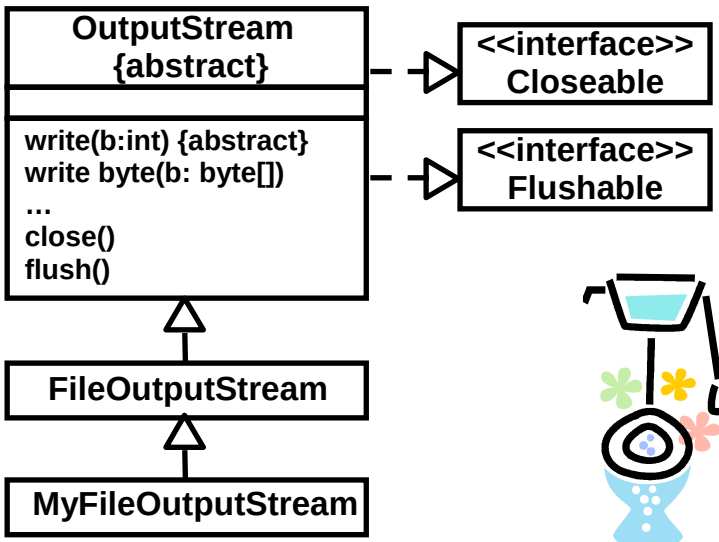
- Superklassen geben abstrakte und leere Methoden (z.B. read(), write()) vor.
- Unterklassen überschreiben diese abstrakten (und leeren) Methoden, da nur sie wissen, wie spezifisch für die Ressource (Quelle oder Senke) tatsächlich (und mit spezifischer Funktionalität) gelesen oder geschrieben oder z.B. die Ressource geschlossen wird. Die Unterklassen realisieren die spezifische Funktionalität des Stream-Typs.

  Beispiel: BufferedReader als Unterklasse von Reader hat zusätzlich die Operation readLine() (kann also eine ganze Zeile am Stück einlesen).
  BufferedWriter hat gegenüber der Superklasse Writer die Methode newLine() zusätzlich (newLine() kann einen Zeilenumbruch in den Ausgabe-Strom schreiben).



- Leere Methoden müssen nicht überschrieben werden.

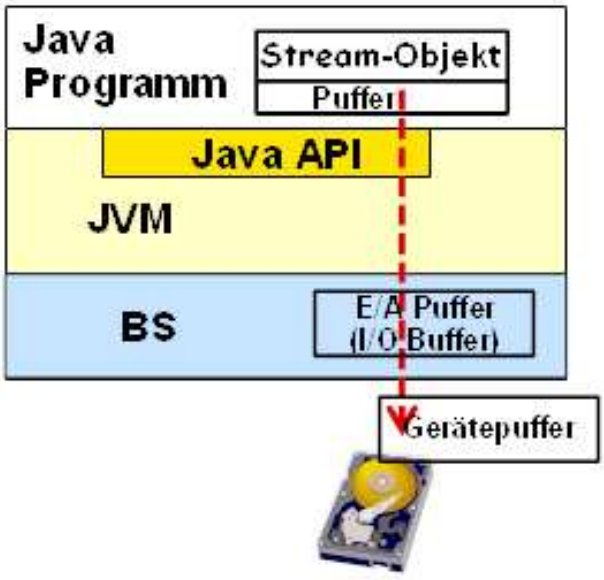
Man beachte: Die Superklassen geben leere Implementierungen vor, damit der Programmierer von Unterklassen nicht gezwungen ist, grundsätzlich immer die Methoden zu überschreiben, auch wenn sie gar nicht gebraucht werden. Streams abstrahieren von zugrundeliegenden realen Geräten. Manche Geräte können mit bestimmten Operationen nichts anfangen.



- Implementierung von Closeable, falls der Datenstrom geschlossen werden kann (void close() throws IOException). Beispiel: Alle Reader/Writer-Klassen.
- Implementierung von Flushable für schreibende, pufferende Stream-Klassen: flush() schreibt im Stream noch zwischengepufferte Daten in die Senke (leert den Puffer).

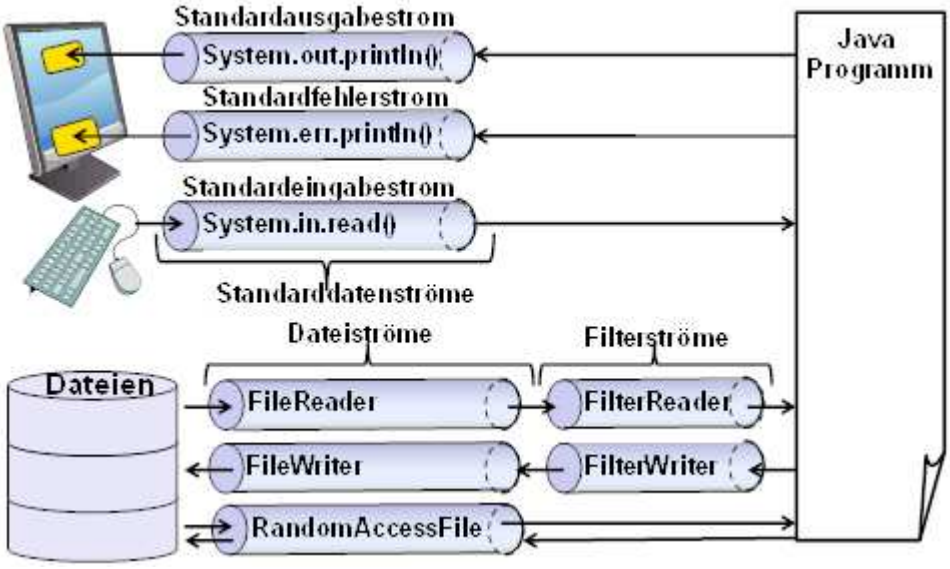
- Achtung: Jede Schicht besitzt einen Puffer! flush() leert nicht den Betriebs-systempuffer. Auf Betriebssystemebene gibt es eigene Befehle, den Puffer z.B. für Dateien zu Leeren, z.B. sync())

Wer also nichts vom Betriebssystempuffer weiß, riskiert den Datenverlust trotz Verwendung von flush() auf der Programmebene.



8.3.3 Standard I/O-Streams

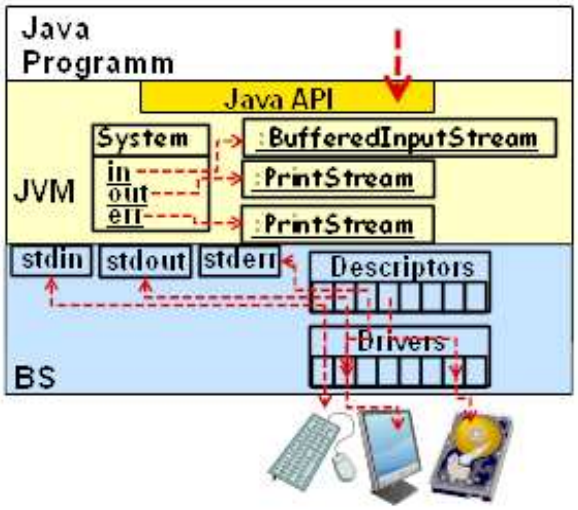
Neben den allgemeinen Streams, den Datei-Streams und den Filter-Streams, gibt es in Java die teils schon ganz zu Beginn verwendeten Standard I/O-Streams, die wir uns jetzt genauer anschauen.



Neben der „Erfindung“ von Streams haben Unix-Entwickler eine automatische Verbindung von Input und Output mit einem Default-Stream eingeführt. Dieses Prinzip findet sich auch auf Programmiersprachenebene (z.B. in Java) wieder.

Zusammenhänge und Strukturen:

- Beim Programmstart mit Java lädt die JVM die Klasse `System`, erzeugt automatisch die Standard-Stream-Objekte (z.B. `BufferedInputStream`), die die Schnittstelle zum Betriebssystem ansprechen können und füllt die Klassenvariablen (z.B. `in`) mit Referenzen auf diese Standard-Stream-Objekte.



- Die Standard-Streams sind immer geöffnet.
- Die Standard-Stream-Objekte können Daten von und zu den vom Betriebssystem bereitgestellten Schnittstellen weiterleiten:
  stdin: Standardeingabe,
  stdin: Standardausgabe,
  stderr: Fehlerausgabekanal.
- Wird eine solche Betriebssystemschnittstelle angesprochen, weiß diese, wie die Daten weitergeleitet werden müssen. Intern werden dazu Deskriptoren (engl. Descriptors) analog zu den File Deskriptoren verwendet. Im Allgemeinen zeigt der Deskriptor mit der Nummer 0 auf den Treiber, der das Eingabegerät ansteuert. Der Deskriptor 1 zeigt auf den Treiber, der das Standardsausgabegerät ansteuert und der Deskriptor 2 auf den Treiber, der das Standardfehlergerät ansteuert.
  Das Betriebssystem behandelt I/O einheitlich (wie für File Deskriptoren) und setzt die Standard I/O-Streams automatisch.

Theoretisch könnte eine Programmiererin die Schreibarbeit durch Einführung einer neuen Referenz folgendermaßen abkürzen:

```
final PrintStream o = System.out;
o.println("test");
```

Bemerkung: Wie bereits vorn bei beim Thema „Klasseneigenschaften“ gesehen, sollte eine solche Abkürzung vermieden werden und stattdessen im Sinne der besseren Lesbarkeit konsequent `System.out` verwendet werden.

Ströme umlenken:

Standard-Ströme können über Java mit Hilfe statischer Funktionen in der Klasse `System` umgelenkt werden:

```
setOut(PrintStream out)
setErr(PrintStream err)
setIn(InputStream in)
```

Zweck der Umlenkung an einem Beispiel: Daten sollen nicht von der Tastatur, sondern von einer Datei oder der Netzwerkverbindung empfangen werden.

Bemerkung:
Wird geschrieben, wird im Normalfall die Schreibreihenfolge eingehalten.
Da der Standard-Fehlerkanal und der Standard-Ausgabekanal getrennt zueinander sind, können sich die Ausgaben von diesen jedoch jeweils überholen.
Das gilt insbesondere, wenn Pufferung involviert ist (geschrieben wird erst, wenn ein Puffer voll ist).
Durch die Trennung von Fehler- und Ausgabekanal können beide Datenströme unabhängig zueinander ans jeweils passende Gerät umgeleitet werden (z.B. die Ausgabe an den Bildschirm und die Fehler an den Drucker als Fehlerprotokoll).

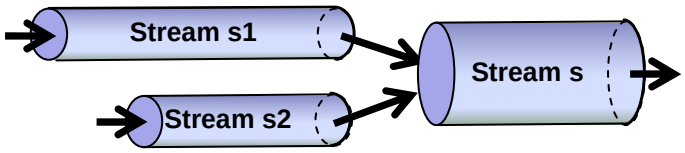
8.3.4 Verknüpfen von Streams

Streams können (z.B. in Java) nach drei Prinzipien verknüpft werden:

(1) Spezialisierung von Streams durch Vererbung (wie oben gesehen)
(2) Verketten von Streams
(3) Filtern/Schachteln: Beim Erzeugen des Streams wird ein (oder mehrere) andere(s) Stream-Objekt(e) als Quelle oder Senke mitgegeben

Zu (2): Verketten (keine "echten" Filter)

Verketten bedeutet das Zusammenfassen mehrerer Input-Stream-Objekte zu einem großen Eingabestrom, so dass sie wie ein einzelner Stream erscheinen. In Java kann dazu die Stream-Klasse `SequenceInputStream` eingesetzt werden.



Einsatz: Wir wollen aus mehreren Strömen lesen und es ist uns gleichgültig, was es jeweils für ein Strom ist und wo er startet.

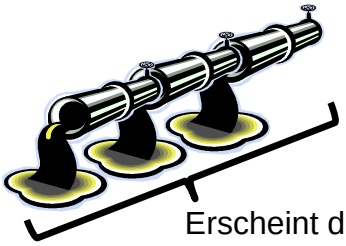
Beispiel in Java:

```
InputStream s1 = new FileInputStream(„datei1.txt");
InputStream s2 = new FileInputStream(„datei2.txt");
InputStream s = new SequenceInputStream(s1,s2);
```

Eine `read()`-Methode auf `s` liest im Beispiel erst die Daten aus `s1`, dann diejenigen von `s2`. Kommen keine Daten mehr von `s2`, aber wieder von `s1` können diese nicht mehr verarbeitet werden.

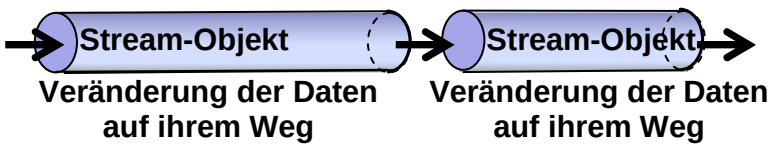
Zu (3): Schachteln bzw. Filtern

Filter sind Stream-Objekte, die selbst wieder Stream-Objekte verwalten.



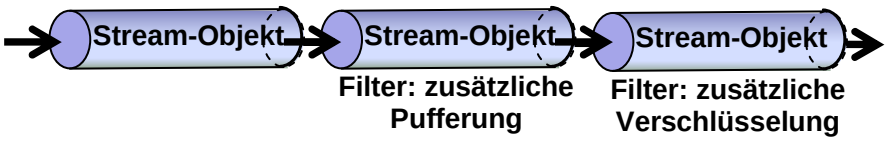
Erscheint dem Aufrufer als ein einziger Stream

Filter verändern die Daten und realisieren so bestimmte Zusatzfunktionen (geeignet zur Durchführung eigener Datenmanipulationen).



Streams können mit einem Samowar (russisch-türkische Teemaschine) verglichen werden: Es wird Wasser eingefüllt (Daten), welches auf seinem Weg durch eine Menge Teekonzentrat (Filter) eine Zusatzfunktion erhält.

Beispiel für den Einsatz von Filtern:



Auf dem Weg der Daten können wir einen Filter einsetzen, der beispielsweise die Zeichen der Nachricht zählt, komprimiert oder in den Lesestrom zurückschreibt.
Anstelle großer und mächtiger Streams wird durch das Konzept ein modularer Ansatz gemäß dem Verantwortlichkeitsprinzip verfolgt. Eine Stream-Klasse ist nur für eine Aufgabe zuständig. Eine spezifisch zugeschnittene Funktionalität kann dadurch erreicht werden, dass die richtigen Filtertypen wiederverwendet und wie in einem Baukastensystem kombiniert werden.
Die Verknüpfung von Filtern wird dadurch möglich, dass Streams von der konkreten und spezifischen Quelle und Senke abstrahieren. Daten können an eine Senke oder aber an einen anderen Stream weitergereicht werden. Dieser andere Stream wird auf abstrakter Ebene gleich betrachtet wie beispielsweise eine Datei als Senke.

Die Verknüpfung der Filter ist in Java (leider) nicht ganz orthogonal (d.h. nicht alle Streams können mit allen einfach so verknüpft werden. Die jeweiligen Konstruktoren zeigen, welche Verknüpfung erlaubt ist).

Spezielle Filter-Stream-Superklassen (auch für eigene Filterklassen) in der Java Standardbibliothek:

	Bytes (oder Byte-Arrays)	Zeichen (char oder Zeichen-Arrays,Strings)
lesen	FilterInputStream	FilterReader
schreiben	FilterOutputStream	FilterWriter

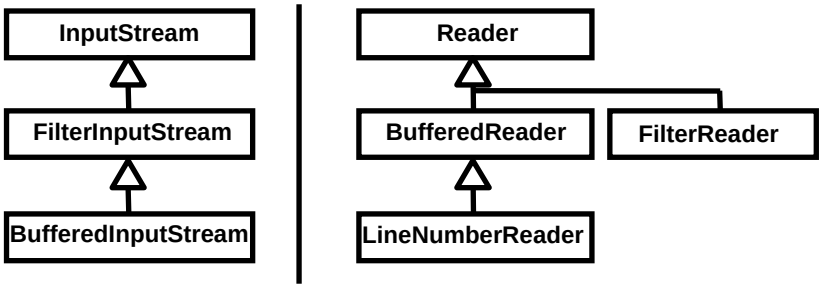
`FilterInputStream` und `FilterOutputStream` filtern Bytes der Ein- und Ausgabe. Sie besitzen Methoden zum Lesen bzw. Schreiben von Daten. Die Filter selbst (bzw. Subklassen dieser) sind für eine Veränderung der ausgelesenen Daten zuständig.
`FilterInputStream` und `FilterOutputStream` haben selbst keinen Filter implementiert. Dafür sind ihre Subklassen zuständig.

Beispiele für Filterklassen in Java:

- `GZIPInputStream` liest komprimierte Daten aus,
  `GZIPOutputStream` schreibt Daten im ZIP Format.
- `BufferedInputStream` und `BufferedOutputStream` sind Filter, die blockweise lesen/schreiben und steigern dadurch die IO-Geschwindigkeit.

Java-Beispiel: Filter zur Pufferung bei In-Streams (symmetrisch dazu für Out-Streams)

- Pufferung ist für die I/O-Geschwindigkeit wichtig.
- Blockweises Lesen/Schreiben ist effizienter als zeichenweises Lesen/Schreiben, da Dateien in Blöcken angeordnet sind und das Betriebssystem Daten ebenfalls blockweise verarbeitet.



Beispiel: Pufferung mit `BufferedReader` in Java

```
BufferedReader r =
    new BufferedReader(new FileReader(„SomeText.txt“));
```

- Dem Konstruktor der Klasse `BufferedReader` wird ein Objekt der Klasse `FileReader` übergeben.
- `read()` auf `BufferedReader` ruft eventuell zuerst `read()` des `FileReader`-Objekts, um den Puffer aufzufüllen.

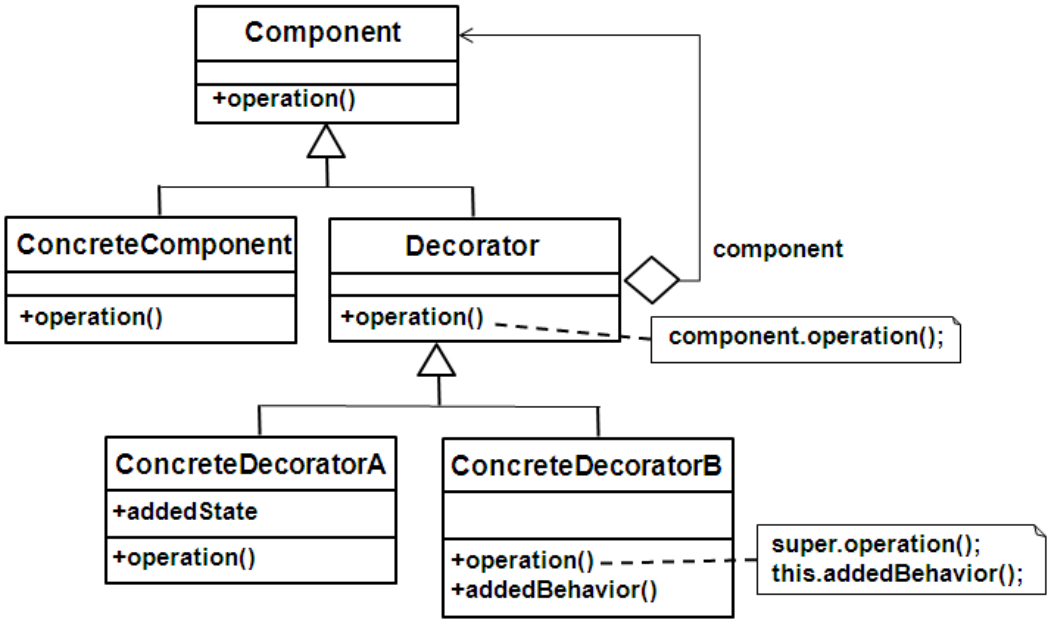
8.3.5 Exkurs: Entwurfsmuster Decorator

Die I/O-Klassen in Java können so angeordnet werden, dass sie beim Übertragen der Daten Funktionalität bzw. Verarbeitungsschritte hinzufügen. Das dabei kennengelernte Prinzip entspricht einem Entwurfsmuster, dem sogenannten *Decorator*.

Aufgabe des Decorator-Entwurfsmusters:

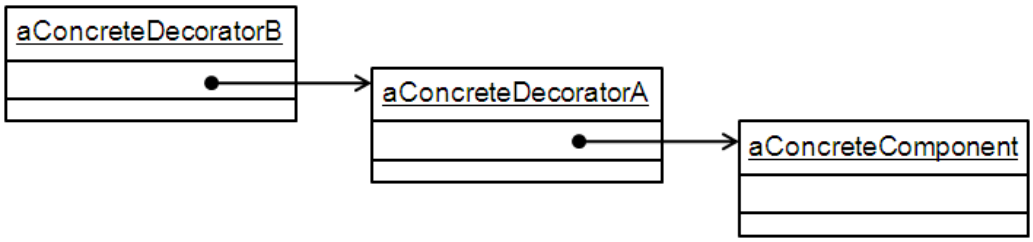
Ein Decorator-Entwurfsmuster fügt einer Kernkomponente (einem Objekt) dynamisch zusätzliche Dekoration (Funktionalität, Verantwortlichkeiten) durch Komposition und Delegation hinzu.
Das Entwurfsmuster Decorator bietet damit eine flexible Alternative zur Verfeinerung / Anpassung durch Vererbung.
Beispiel: Dekoration eines Fensters mit Rahmen, Scrollbar, Hintergrundsfarbe etc.

Klassendiagramm und Umsetzung des Decorator-Entwurfsmusters:



Ein Client nutzt die Schnittstelle (Methode `operation()`) eines Objekts vom Typ `Component`. Eine `Component` kann direkt verwendet werden oder wird von einem `Decorator` dynamisch umwickelt. Ein `Decorator` ist ebenfalls vom Typ `Component` und implementiert das gleiche Interface bzw. die gleiche abstrakte Klasse wie die `Component`, die der `Decorator` umwickelt.

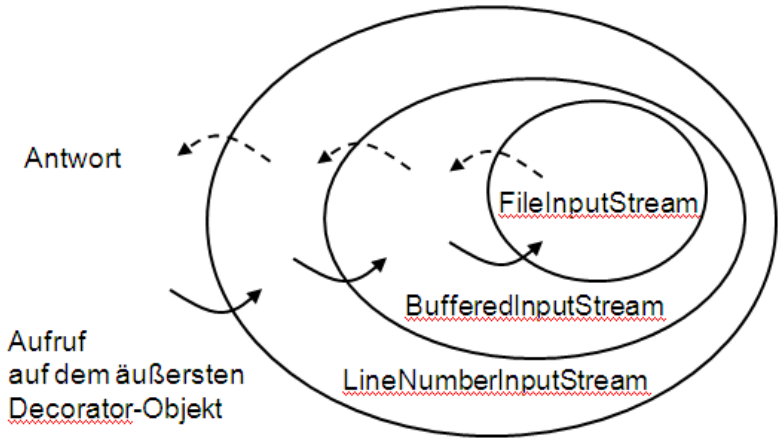
Eine `Component` ist damit entweder eine `ConcreteComponent` oder aber ein `Decorator` bzw. `ConcreteDecorator` (z.B. ein Objekt der Klasse `LineNumberInputStream`).
Der `Decorator` hat die Referenz auf eine andere `Component` (Aggregationsbeziehung).
Diese Referenz ist wieder entweder eine `ConcreteComponent` oder ein `Decorator` (z.B. `BufferedInputStream`) usw.
Prinzipiell können so beliebig viele `Decorator`s in beliebiger Reihenfolge und Anzahl zum Umwickeln einer `Component` eingesetzt werden.



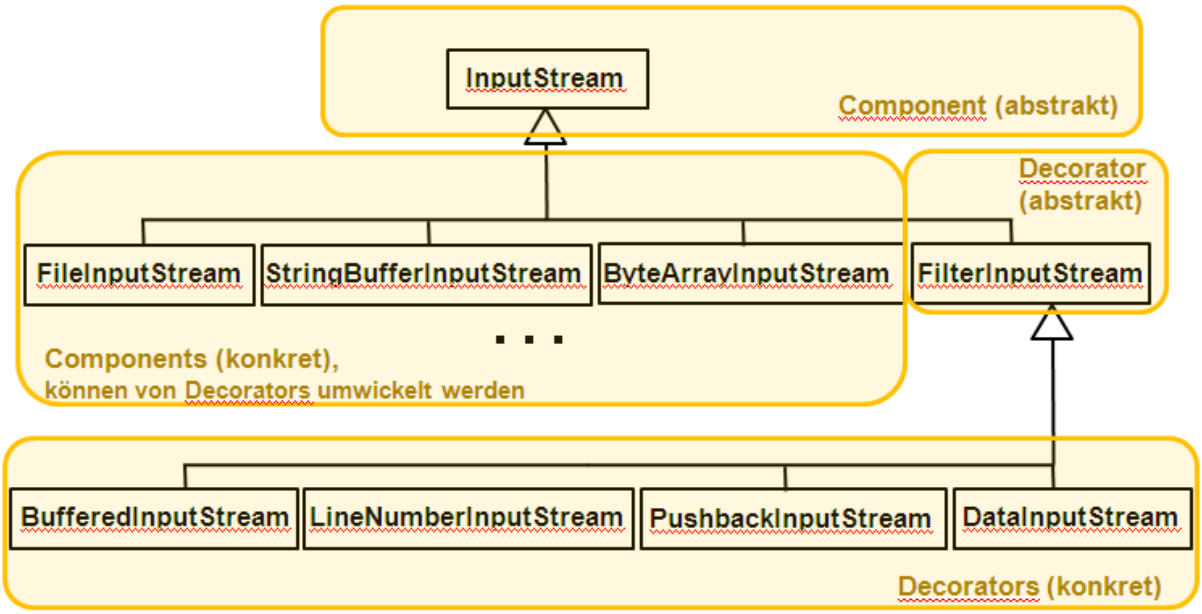
Jeder konkrete Decorator kann einen eigenen Zustand hinzufügen (addedState) oder zusätzliches Verhalten. Zusätzliches Verhalten wird durch die Methode addedBehavior() bzw. durch Erweiterung der überschreibenden Methode operation() erreicht. Eine Methode zur Verhaltenserweiterung (addedBehavior()) wird aus der Methode operation() heraus aufgerufen. Typischerweise wird eine schon in der Component existierende Operation vor oder nach der Abarbeitung erweitert (oder aber insgesamt ersetzt).
Falls kein spezielles Verhalten hinzugefügt werden soll, wird kein addedBehavior() eingesetzt und nur die Operation der Superklasse gerufen. Die Operation operation() der Superklasse ruft die Operation der referenzierten Komponente auf und reicht so die Verarbeitung in der Decorator-Kette weiter.
Die Namen der Klassen, Attribute und Operationen (z.B. operation()) unterscheiden sich je nach Anwendungsfall.

Beispiel für den Einsatz des Decorator-Entwurfsmusters: java.io

Jeder eingesetzte Decorator umwickelt den nächsten bis im Inneren schließlich eine konkrete Komponente eingesetzt wird. Jeder Decorator fügt der Komponente somit zusätzliche Funktionalität hinzu. In der Java-Bibliothek (I/O) fügt ein LineNumberInputStream beispielsweise die Fähigkeit hinzu, die Zeilennummern beim Einlesen der Daten zu zählen. Ein BufferedInputStream ist ein weiterer konkreter Decorator, der die Eingabe (zur Performance-Verbesserung) puffert. BufferedInputStream erweitert das Interface außerdem um die Methode readLine() für zeilenweises Einlesen. Eine konkrete Komponente ist beispielsweise FileInputStream.
Für einen Client ist diese Umwicklungsstruktur transparent. Tatsächlich nutzt er immer das äußerste Objekt (d.h. den äußersten Decorator), weiß aber tatsächlich prinzipiell nichts über die Dekoration(skette).



Die Java-Bibliothek verfolgt für die I/O-Klassen das Decorator-Entwurfsmuster, was sich auch so im Klassendiagramm weitgehend wiederfindet:



Bewertung des Entwurfsmusters Decorator:

Vorteile:

- Es kann Verhalten erweitert werden, ohne dass Vererbung genutzt werden muss. Die Vererbung wird nur eingesetzt, um Schnittstellen festzulegen und um `Component` und `Decorator` einen gemeinsamen Typ zu geben. Das Umwickeln mit einem `Decorator` ändert daher nicht den Typ einer `Component` und verhindert so Typprobleme.
  Statt abstrakter Superklassen könnten zu diesem Zweck auch Interfaces der Programmiersprache eingesetzt werden.
- Funktionalität kann dynamisch zur Laufzeit hinzugefügt und entfernt werden, ohne die Notwendigkeit andere, vorhandene Objekte (bzw. vorhandenen Code) zu modifizieren (Open-Closed-Principle: Offen für Erweiterungen, geschlossen für Modifikation). Bei Verwendung von Vererbung müßten demgegenüber Änderungen zur Übersetzungszeit festgelegt werden.

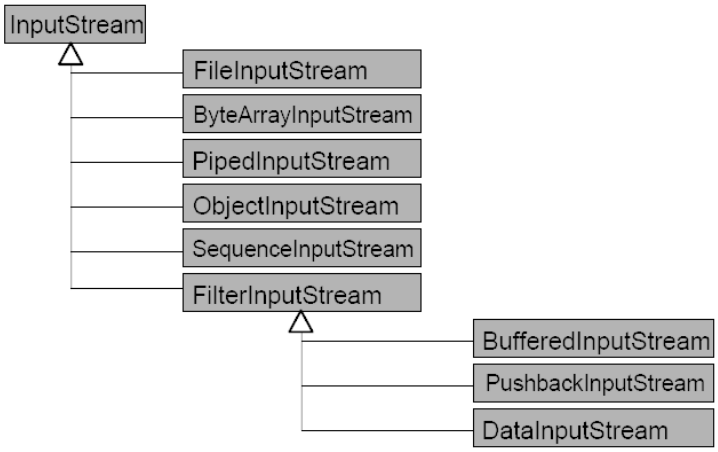
Nachteile:

- Es entstehen viele kleine Wrapper-Klassen.
- Der Entwurf enthält mehr Abstraktion und mehr Komplexität.
- Dadurch wird die Bibliothek oft schwer zu verstehen.

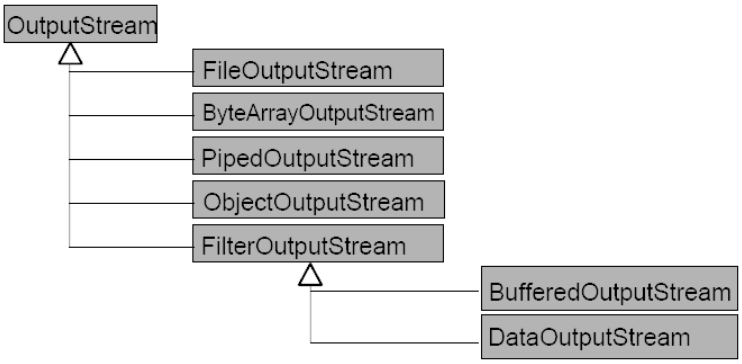
Fazit: Decorator ist ein wichtiges und sehr leistungsfähiges Entwurfsmuster. Aufgrund der genannten Nachteile sollte es aber nur dort eingesetzt werden, wo die Flexibilität für Änderungen wirklich benötigt wird.

8.3.6 I/O-Klassenhierarchie in Java (Auszug)

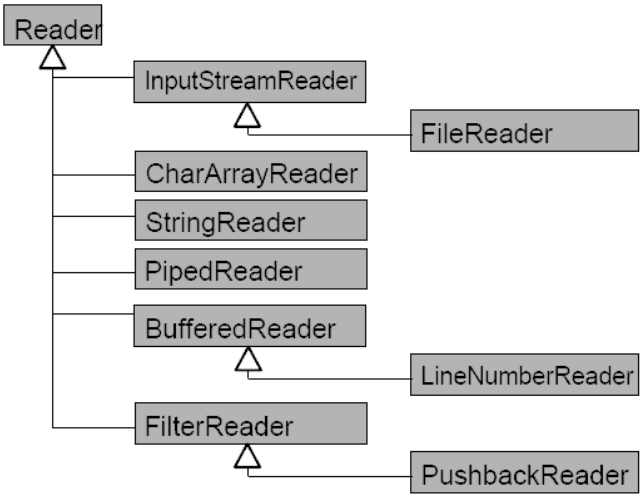
Diese Aufstellung dient nur der Übersicht über die Zusammenhänge und Möglichkeiten. Zur Programmierung muss die Java API mit ihren aktuellen Spezifikationen verwendet werden.



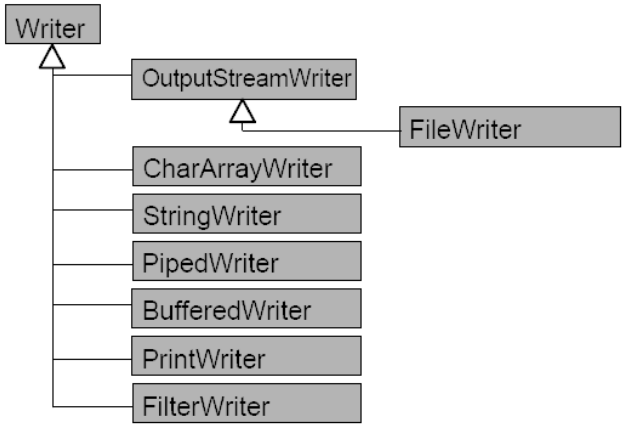
Klasse	Beschreibung
InputStream	Abstrakte Klasse für die byte-orientierte Eingabe
FileInputStream	Eingabe von Datei
ByteArrayInputStream	Eingabe aus Byte-Array
PipedInputStream	Eingabe von Pipe (Nachrichtenwarteschlange zur Kommunikation zwischen Threads)
ObjectInputStream	Eingabe serialisierter Objekte
SequenceInputStream	Verkettung mehrerer Eingabe-Streams zu einem Stream
FilterInputStream	Abstrakte Klasse für die gefilterte Eingabe
BufferedInputStream	Eingabe mit Pufferung
PushbackInputStream	Eingabe mit der Möglichkeit zur Rückstellung von gelesenen Bytes
DataInputStream	Eingabe von primitiven Datentypen in portabler Weise



Klasse	Beschreibung
OutputStream	Abstrakte Klasse für die byte-orientierte Ausgabe
FileOutputStream	Ausgabe in eine Datei
ByteArrayOutputStream	Ausgabe in ein Byte-Array
PipedOutputStream	Ausgabe in eine Pipe (Nachrichtenwarteschlange zur Kommunikation zwischen Threads)
ObjectOutputStream	Ausgabe serialisierter Objekte
FilterOutputStream	Abstrakte Klasse für die gefilterte Ausgabe
BufferedOutputStream	Ausgabe mit Pufferung
DataOutputStream	Ausgabe von primitiven Datentypen in portabler Weise



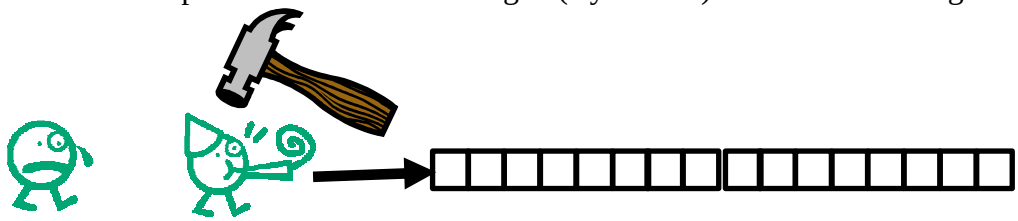
Klasse	Beschreibung
Reader	Abstrakte Klasse für die zeichen-orientierte Eingabe
InputStreamReader	Klasse für das Interpretieren eines Byte-Streams als Zeichen-Stream
FileReader	Eingabe von Datei
CharArrayReader	Eingabe aus Zeichen-Array
StringReader	Eingabe aus Zeichenkette
PipedReader	Eingabe von Pipe (Nachrichtenwarteschlage zur Kommunikation zwischen Threads)
BufferedReader	Eingabe mit Pufferung und Eingabe ganzer Zeilen
LineNumberReader	Eingabe mit der Fähigkeit, Zeilen zu zählen
FilterReader	Abstrakte Klasse für gefilterte Eingabe
PushbackReader	Eingabe mit der Möglichkeit zur Zurückstellung von Zeichen



Klasse	Beschreibung
Writer	Abstrakte Klasse für die zeichen-orientierte Ausgabe
OutputStreamWriter	Klasse für die Umformung eines Zeichen-Streams in einen Byte-Stream
FileWriter	Ausgabe in eine Datei
CharArrayWriter	Ausgabe in ein Zeichen-Array
StringWriter	Ausgabe in eine Zeichenkette
PipedWriter	Ausgabe in eine Pipe (Nachrichtenwarteschlage zur Kommunikation zwischen Threads)
BufferedWriter	Ausgabe mit Pufferung
PrintWriter	Ausgabe aller Datentypen im Textformat
FilterWriter	Abstrakte Klasse für die gefilterte Ausgabe

8.4. Serialisierung

Ziel: Wir möchten ein aktives Objekt (im Hauptspeicher einer Anwendung zugeordnet) in ein transportables bzw. speicherbares Format bringen (Bytestrom) = Serialisierung



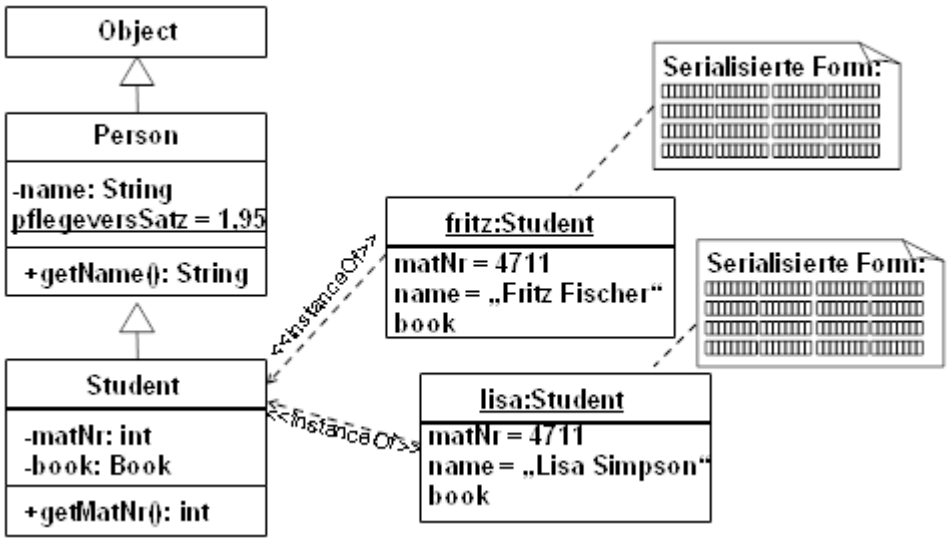
Einsatzbeispiele: (Persistente = dauerhafte) Speicherung in einer Datei, Transport über eine Netzwerkverbindung

Deserialisierung: Serialisiertes Objekt rekonstruieren

Grundsätzlich muss überlegt werden, welche Informationen eines Objekts den ganz spezifischen Objektzustand zu einem bestimmten Zeitpunkt widerspiegeln. Nur diese Informationen müssen wir durch eine Serialisierung speichern. Die übrige Information (z.B. die Klassenvorgaben) könnten wir zwar ebenfalls in eine persistente Form bringen. Das wäre aber Verschwendung.

Bei der Serialisierung/Deserialisierung möchten wir alle mit dem Objekt verbundenen Variablen berücksichtigen.

Beispiel:



Problemfälle beim Serialisieren:

- Instanzvariablen, die Referenzen auf andere Instanzen enthalten (rekursive Serialisierung!)
- Vererbung, geerbte Instanzvariablen
- Klassenvariablen
- Daten, die nicht serialisiert werden sollen (z.B. sensible Daten)
- Änderung und Versionierung

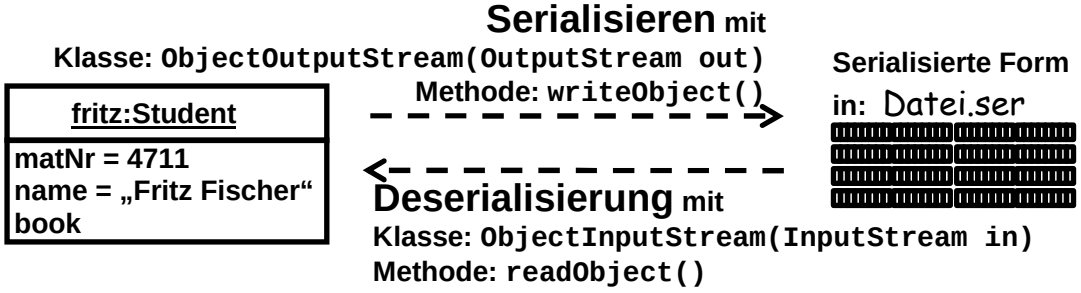
Auch die Beziehungen müssen erhalten bleiben. Die meisten Beziehungen bilden sich allerdings auf der Programmiersprachenebene auf Instanzvariablen ab.

8.4.1 Standardserialisierung in Java (Paket java.io)

Umwandlung in einen Bytestrom / Binärformat mit Java Object Serialization (JOS).

Serialisierung:

- Läuft den Zustand und die Objektverweise rekursiv ab und schreibt in einen OutputStream.
- Konvention: Eine Datei, die ein serialisiertes Objekt speichert hat die Endung .ser:



Deserialisierung:

- Findet den Typ des serialisierten Objekts, baut daraus das Zielobjekt auf (keine Instanziierung, kein Konstruktoraufruf!), stellt den Objektzustand wieder her (d.h. sorgt für die richtige Variablenbelegung).
- Die zum Objekt gehörende Klasse muss vorhanden sein. Anderenfalls erhalten wir in Java eine ClassNotFoundException.
- Wenn nötig: Rekursive Rekonstruktion (d.h. auch Objekte, auf die verwiesen wird).

Beispiel einer Serialisierung in Java:

```
FileOutputStream fos =
      new FileOutputStream("Datei.ser");

ObjectOutputStream o =
      new ObjectOutputStream(fos);

o.writeObject(new Integer(10));
```

In Java wird die Serialisierung bzw. Deserialisierung mit Hilfe von Streams realisiert.
Ein FileOutputStream wird im Beispiel benötigt, um das serialisierte Objekt in eine Datei zu schreiben (im Beispiel oben in eine Datei mit Namen Datei.ser). Ein ObjectOutputStream ist für die tatsächliche Serialisierung zuständig. Ein Objekt wird durch eine Nachricht writeObject() an das ObjectOutputStream-Objekt serialisiert. Im Beispiel wird auf diese Weise ein Integer-Objekt serialisiert.
Die Deserialisierung läuft symmetrisch in umgekehrter Richtung. Bei der Deserialisierung ist zu beachten, dass der Rückgabewert der Methode readObject() vom Typ Object ist. Er muss daher durch Typkonvertierung in den tatsächlichen Typ (oder in den einer seiner Superklassen) angepasst werden.

Korrekter Umgang mit zyklischen Referenzen:

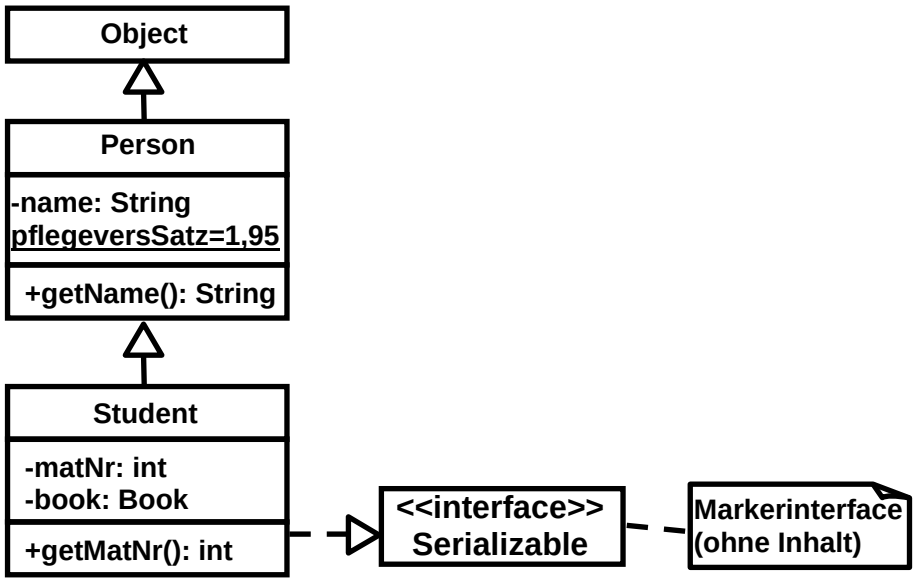
- Ein Objekt wird auch dann nur einmal serialisiert bzw. deserialisiert, wenn in einer Objektstruktur mehr als eine Variable darauf verweist. In der rekursiven Abarbeitung eines Objekts mit seinen Verweisen erkennt das System solche zyklischen Referenzen anhand von Objekt Handles (eindeutige Verweise bzw. eindeutige Kennzeichnungen).
- Im Fall von serialisierten zyklischen Referenzen stellt das System sicher, dass nach der Deserialisierung auf das eine Objekt verwiesen wird.
- Geeignetes Hilfsmittel: Hash-Tabelle in `ObjectOutputStream` (verzeichnet alle bereits serialisierten Objekte).

Serialisierbarkeit:

- Nur Objekte, deren Klasse das Markerinterface `Serializable` implementieren, können serialisiert werden. Implementiert eine Klasse das Markerinterface, so ist sie automatisch in der Lage, alle Instanzvariablen zu serialisieren.
- Sehr viele Klassen der Java API implementieren das Interface; einige Klassen können jedoch nicht serialisiert werden.
  Eine Serialisierung ist nicht für alle Klassen sinnvoll.
  Beispiele: Informationen, die ihren Wert nach einer Deserialisierung verloren haben (z.B. ein mit einer Datei verbundener `BufferedWriter`), sicherheitskritische Informationen, leicht wieder erzeugbare Informationen, offene Dateien oder Netzwerkverbindungen

Rekursive Serialisierung (Verfolgen von Objektreferenzen):

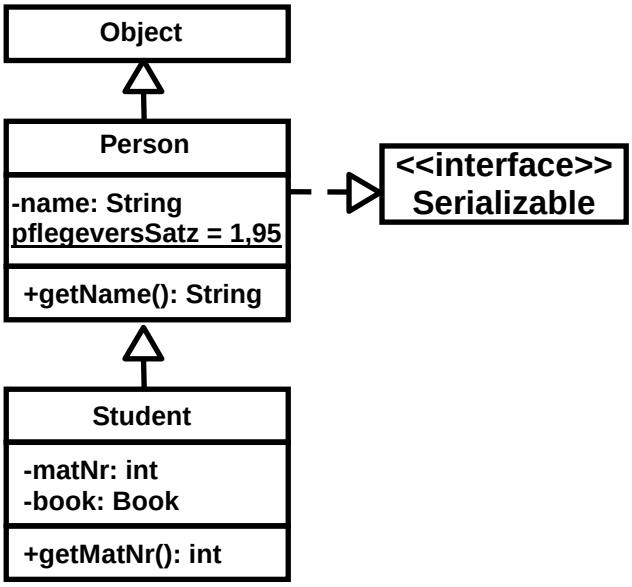
Implementiert im Beispiel das `Book`-Objekt nicht ebenfalls das `Serializable` Interface wird eine `NotSerializableException` geworfen, wenn wir versuchen, das `Book`-Objekt oder das `Student`-Objekt zu serialisieren. Die Klasse referenziert dann mit `Book` ein nicht serialisierbares Objekt.



Serialisierung und Vererbung:

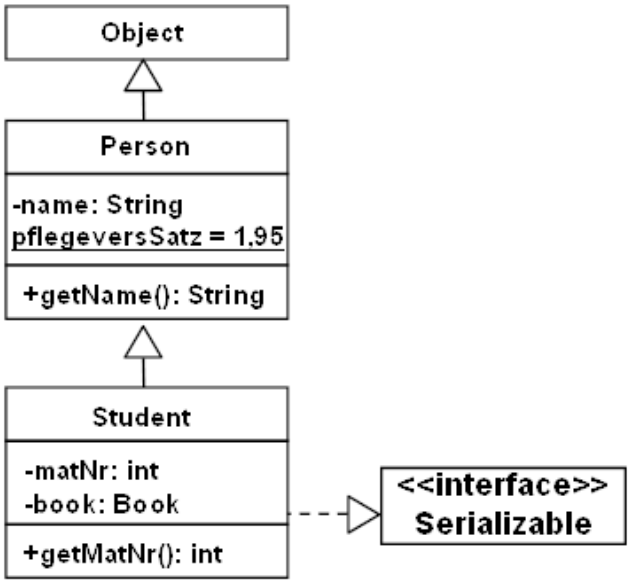
- Implementiert eine Klasse das Interface Serializable, sind auch alle Unterklassen serialisierbar.
- Umgekehrt gilt das nicht.

Beispiel 1:
Person ist serialisierbar, da Serializable implementiert wird. Student ist serialisierbar, da die Eigenschaft von Person geerbt wird. Bei einer Serialisierung von Student wird auch die Superklasse Person serialisiert.



Beispiel 2:

- Die Superklasse Person implementiert in diesem Beispiel nicht das Markerinterface Serializable (auch Object implementiert es nicht).
- Soll ein Student-Objekt serialisiert werden, werden daher die von Person geerbten Elemente nicht serialisiert.
- Deserialisierung: Java ruft den parameterlosen Konstruktor zur sinnvollen Belegung der geerbten, nicht serialisierten Variablen.
- Gibt es keinen solchen Konstruktor, ist eine Deserialisierung nicht möglich.



Object und Serializable:

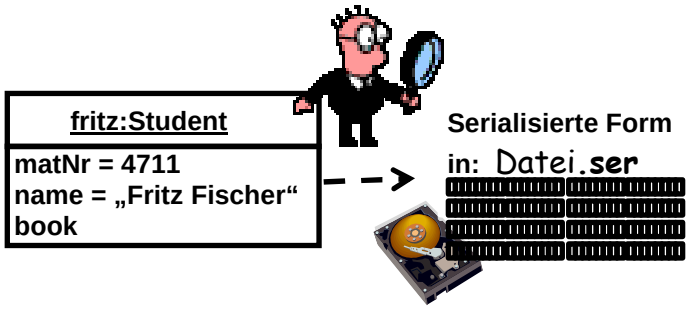
Würde Object Serializable implementieren, wären damit alle Klassen Serializable. Das wollen wir nicht: Viele (API) Klassen sind zwar serialisierbar (d.h. also sie implementieren Serializable), aber nicht alle. Würde Object das Interface Serializable implementieren, müssten wir für alle nicht serialisierbaren Klassen die geerbte Eigenschaft (Serialisierbarkeit) wieder zurücknehmen (z.B. durch die Implementierung eines Interface NotSerializable ?). Das würde dem Substitutionsprinzip widersprechen.

Klassenvariablen:

- Klassenvariablen werden nicht serialisiert (im Beispiel: `pflegeversSatz`).
- Klassenvariablen gehören nicht einem spezifischen Objekt. Eine Serialisierung bezieht sich aber immer auf ein Objekt.

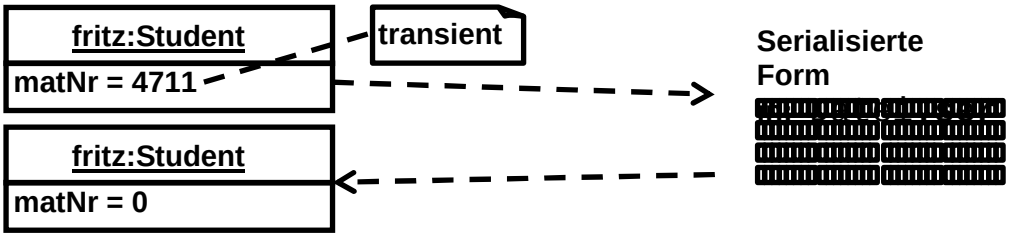
Explizite Ausnahmen von der Serialisierung (z.B. bei geheimen und sensiblen Daten)

Sensible (auch private) Daten sind durch die Serialisierung lesbar und manipulierbar. Durch die Serialisierung liegen diese Daten z.B. auf der Festplatte und es lassen sich die internen Belegungen ablesen und ändern.



Umsetzung von Ausnahmen in Java:

- Einzelne Instanzvariablen können von der Serialisierung ausgenommen werden. In Java werden diese als *transient* deklariert. Wird eine Variable mit diesem Schlüsselwort versehen, wird sie bei der Serialisierung nicht berücksichtigt.
- Beispiel: `transient private int matNr;`
- Einsatz:
  - sensible Daten (z.B. Passwörter)
  - Instanzvariable, die auf ein nicht serialisierbares Objekt verweist
  - Zustände, die nur temporär sind (z.B. bei Streams)
- Deserialisierung: Den mit dem Schlüsselwort `transient` versehenen Variablen wird bei der Deserialisierung jeweils der typspezifische Default-Wert zugewiesen (kein Konstruktoraufruf!).

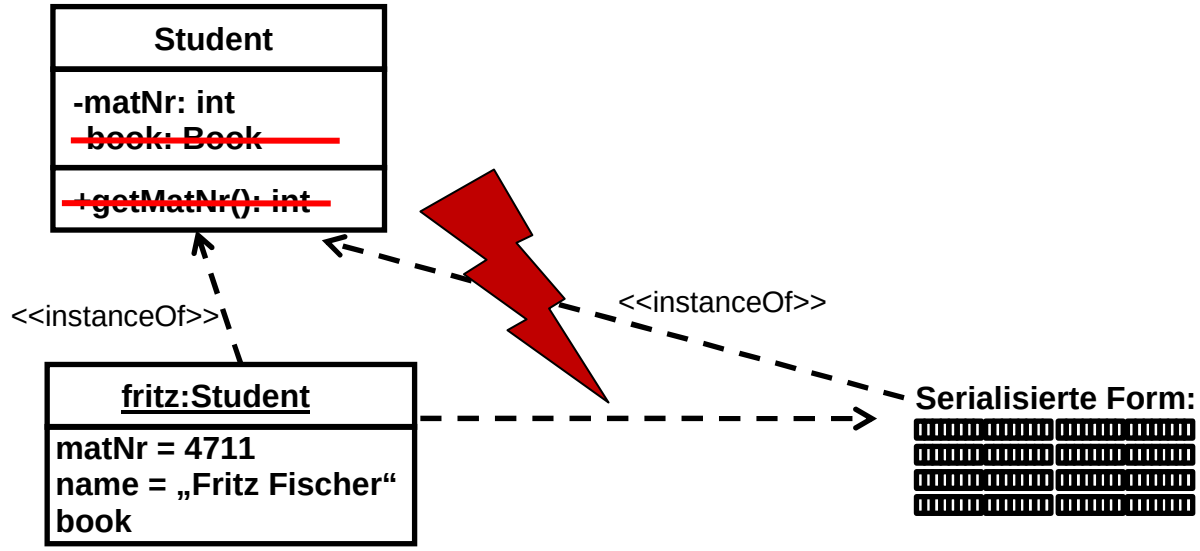


Alternative zu `transient`:

- Alternative Festlegung, welche Instanzvariablen serialisiert werden sollen bzw. können mit Hilfe der Klassenvariable `serialPersistentField`.
- Die Variable muss folgendermaßen definiert werden:
  `private static final ObjectStreamField[] serialPersistentFields`
- Die einzelnen Einträge des Arrays sind die Variablen, die serialisiert werden sollen.

Versionierung und Versionskontrolle:

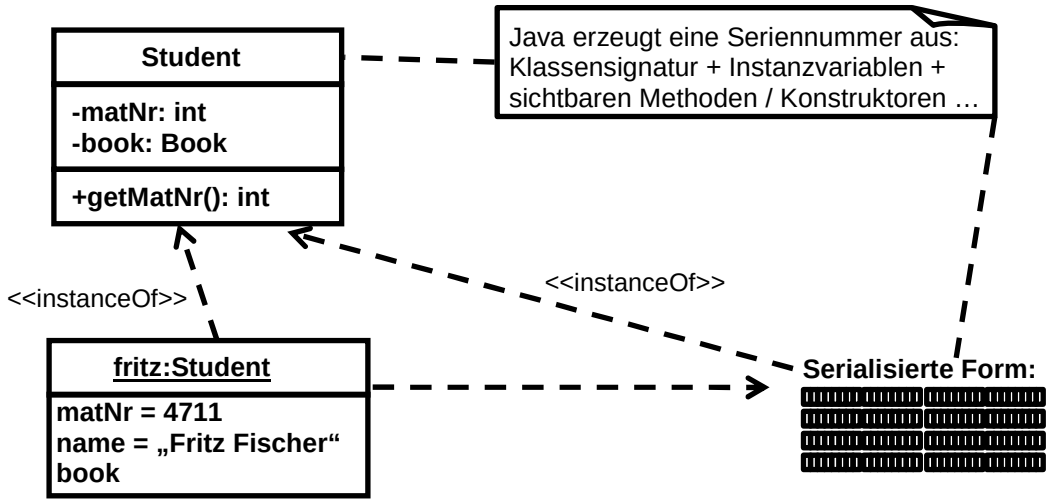
- Eine neue Version unseres Programms bzw. unseres Objekts entsteht, wenn wir es ändern (z.B. verbessern). Auch der Klassen-Code zu serialisierten Objekten kann durch Weiterentwicklung verändert werden.
- Problem der Änderung im Kontext der Serialisierung: Die Implementierung und das serialisierte Objekt laufen auf getrennten Wegen auseinander.
  Die Klassenimplementierung passt nach der Änderung eventuell nicht mehr zum serialisierten Objekt
  ⇒ Inkompatibilität zwischen Quellcode und dessen Serialisierungen.



Liegen die serialisierten Objekte in persistenter, serialisierter Form vor, sind sie vom eigentlichen Java-Code abgekapselt. Es kann vorkommen, dass sich der Java-Code ändert (im Beispiel entfernen wir nach der Serialisierung das Attribut `book` und eine Methode) und anschließend die persistent gespeicherten Objekte nicht mehr zur Implementierung passen.

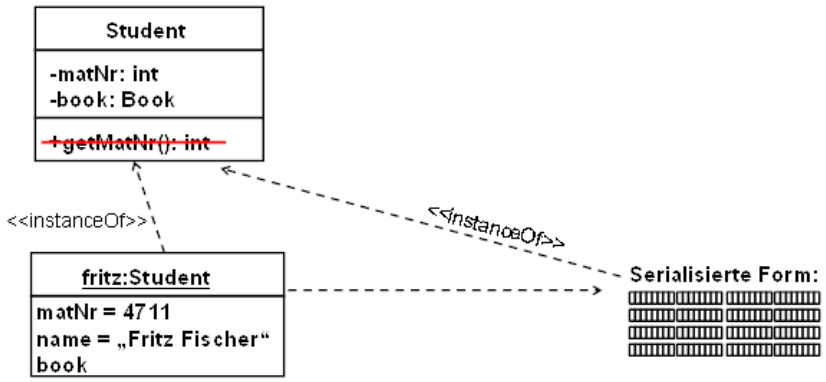
Abhilfe gegen das Inkompatibilitätsproblem: Versionsnummer (eingebauter Java Versionsschutz)

- Zu einer Klasse kann eine Versionsnummer erzeugt werden: serialVersionUID (SUID).
- Die Versionsnummer wird nach einem Hash-Verfahren automatisch aus dem Namen bzw. der Signatur der Klasse, den implementierten Interfaces, den sichtbaren Methoden bzw. Konstruktoren (mit Bezeichner, Modifikator, Rückgabetyp, Parametern aber ohne Methodenrümpfe) und den Instanzvariablen errechnet.
- Beim Aufruf der Methode `writeObject()` (Serialisierung eines Objekts) wird diese Versionsnummer automatisch aus der Klasse berechnet und in den Ausgabestrom geschrieben.
- Klassenänderung ⇒ Versionsnummer ändert sich (d.h. die Versionsnummer vom serialisierten Objekt und der aktuellen, geänderten Klassenimplementierung weichen voneinander ab).
- Ändert sich die Klasse in einem Bestandteil der Versionsnummer ⇒ Deserialisierung nicht mehr möglich (die serialisierten Daten dürfen wegen Inkompatibilität nicht mehr verwendet werden; die aktuelle Versionsnummer wird mit derjenigen im serialisierten Objekt verglichen und führt gegebenenfalls zur einer `InvalidClassException`).



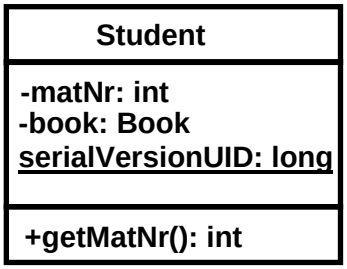
Eine kleine Änderung an der Klasse kann damit alle davon serialisierten Objekte unbrauchbar machen. Obwohl die automatische Versionskontrolle ein sinnvoller Mechanismus ist, um die Konsistenz zwischen Quellcode und serialisierten Objekten sicherzustellen, wären manchmal (je nach Änderung an der Klasse) die Serialisierungen trotzdem noch kompatibel.

Beispiel: Die Entfernung der sichtbaren Methode `getMatNr()` würde in unserem Fall zu keiner Inkonsistenz zwischen Klasse und serialisierten Objekten führen.



Abhilfe: Versionskontrolle durch eine eigene Versionsnummer (in der Verantwortung der Programmiererin) statt automatische Versionsnummernvergabe.

- Eigene Versionsnummer durch Anlegen der Klassenkonstante:
  `static final long serialVersionUID = …;`
- Die Erkennung und das Festhalten der Inkompatibilität geschieht in Eigenverantwortung durch den Programmierer.
- Oracle/Sun empfiehlt bei serialisierbaren Klassen generell, eine eigene serialVersionUID anzulegen, da sich die automatische Berechnung auch bei sich nicht ändernden Klassen je nach Compiler und JVM ändern könnte.



In der manuellen Versionsnummernvergabe muss der Programmierer gut überlegen, wann er eine neue Nummer vergeben sollte.

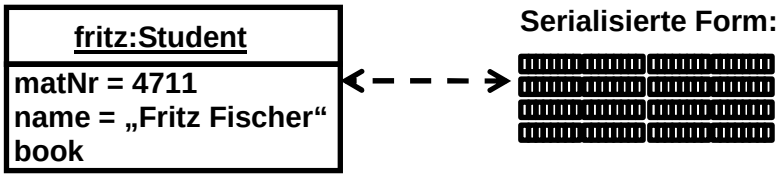
- Änderungen, die teils keine Probleme für serialisierte Objekte darstellen: Hinzufügen / Entfernen von Methoden, Hinzufügen von Instanzvariablen, Klassen im Vererbungsbaum

einfügen/entfernen, Änderung der Sichtbarkeit von Instanzvariablen, Änderung einer Instanzvariable von transient nach „non-transient“.
Beim Hinzufügen neuer Variablen ist zu beachten, dass diese nach der Deserialisierung nicht initialisiert sind. Sie erhalten Default-Werte.

- Beispiele für gegebenenfalls problematische Änderungen (die folglich in der Regel zu einer neuen Versionsnummer führen sollten): Entfernen von Instanzvariablen, Umbenennung von Instanzvariablen, Änderung des Typs von Instanzvariablen, Änderung zu static oder zu transient, Änderung des Klassennamens, Verschieben einer Klasse in der Vererbungshierarchie, Änderung der Klasse zu „non-serializable“.

8.4.2 Alternativen zur Serialisierung in Java Programmen

(1) Serialisierung in XML (JavaBeans Persistence)
(2) JAXB API (Abbildung der Objektstruktur auf XML-Dokumente), JAXB ist Teil von Java seit Java 6
(3) Zusätzliche Frameworks (z.B. Hibernate zur Abbildung in eine Datenbank)
(4) Handarbeit (eigenes Format)



Serialisierung mit XML:

- Die Serialisierung mit einem `ObjectOutputStream` hat zur Folge, dass die Objekte in einem für den Menschen nicht lesbaren, sehr speziellen Format vorliegen.
- Will man eine textuelle Repräsentation eines Objektes oder das serialisierte Objekt jenseits von Java verwenden, bietet sich die Serialisierung in ein XML-Format an.
- Dazu muss lediglich anstelle eines `ObjectOutputStream`-Objektes ein Objekt der Klasse `java.beans.XMLEncoder` verwendet werden.
- Zum Auslesen persistent gespeicherter Objekte dient die Klasse `java.beans.XMLDecoder`.
- Soll eine Klasse mit einem `XMLEncoder` serialisiert werden, muss sie die Struktur einer JavaBean aufweisen, d.h.:
  - Es muss ein Default Constructor vorhanden sein.
  - Für alle zu serialisierenden Attribute müssen get- und set-Methoden vorhanden sein.
  - Die Klasse muss public deklariert sein.

