I/O-Serialisierung und Multithreading

PROG 2: Einführung in die Programmierung für Wirtschaftsinformatiker

---

Dr.-Ing. Steffen Helke
Technische Universität Berlin
Fachgebiet Softwaretechnik

27. Mai 2013

Übersicht

- Wiederholung: Ein- und Ausgabestreams
- Serialisierung
- Komprimierung
- Threads in Java

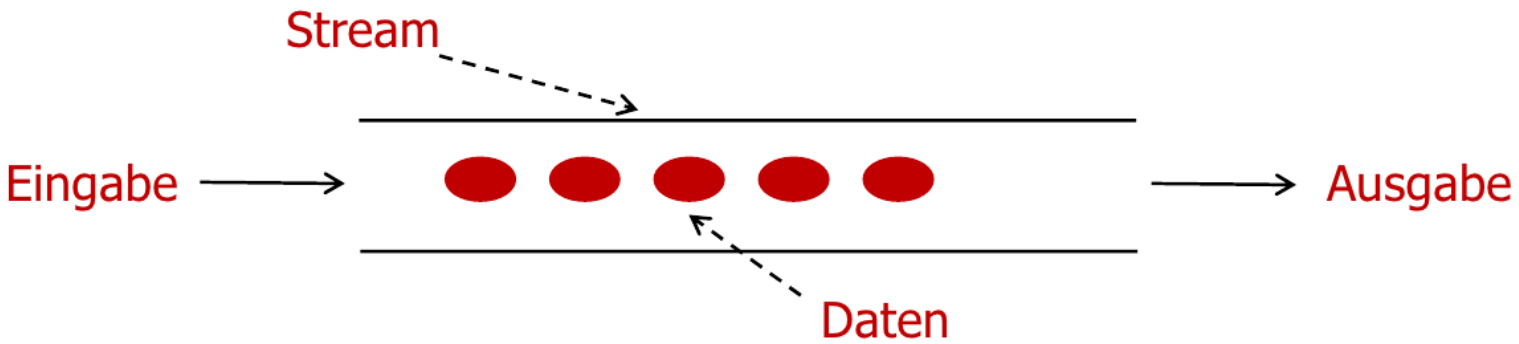
Teil III der Vorlesung PROG 2

Ein- und Ausgabestreams

Fortgeschrittene Techniken

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Streams in Java



Varianten

1. Byte-basierte Streams
   - Daten werden als Bytes gelesen/geschrieben
   - geeignet *für binäre Dateien*
2. Charakter-basierte Streams
   - Daten werden als alpahnumerische Zeichen (16 Bit Unicode-Zeichen) gelesen/geschrieben
   - geeignet *für textuelle Dateien*

Lesen und Schreiben von Dateien

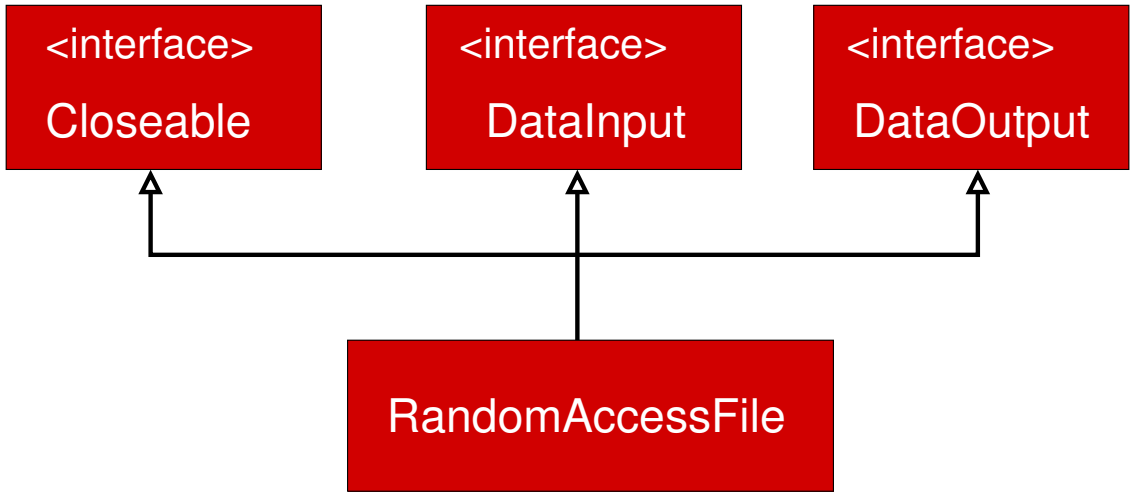
Herausforderung

- Auswahl der richtigen Werkzeuge
- *mehr als 20 Dateizugriffsklassen* verfügbar

Unterscheidung

1. Wahlfreier Zugriff (*random access*): Dynamische Bearbeitung an gewählten Positionen möglich
2. Sequentieller Zugriff (*stream access*): Elemente als Zeichenfolge, Bearbeitung erfolgt streng nacheinander

1. Wahlfreier Zugriff (Random Access)




```
// Konstruktoren
RandomAccessFile (String name, String mode)
RandomAccessFile (File file, String mode)
// aktuelle Position
long getFilePointer()
// Position setzen
void seek (long pos)
// Dateilänge
long length()
void setLength (long len)
```

Wahlfreier Zugriff (Random Access)

Mode beschreibt Zugriffsmodus

- r = nur lesen, rw = lesen und schreiben
- rws = synchronisiertes Lesen und Schreiben (Daten und Metadaten, wie z.B. letzter Dateizugriff)
- rwd = synchronisiertes Lesen und Schreiben (Daten ohne Metadaten)
- rwd und rws = sofortiges Schreiben auf die Platte

Einordnung

- Datei wird als großes Array von Zeichen betrachtet
- kann nur auf echten Daten verwendet werden, nicht auf Arrays oder Pipes
- kein Filterkonzept, wie bei Streams

Elementare Interfaces zum Lesen/Schreiben

- elementare Sicht auf Datei: Folge von Bytes
- abstraktere Sicht: Folge von primitiven Elementen

interface DataInput

```
void read(byte[] b)
void readFully(byte[] b)
...
boolean readBoolean()
byte readByte()
char readChar()
short readShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
String readline()
int skipBytes (int n)
String readUTF()
...
```

interface DataOutput

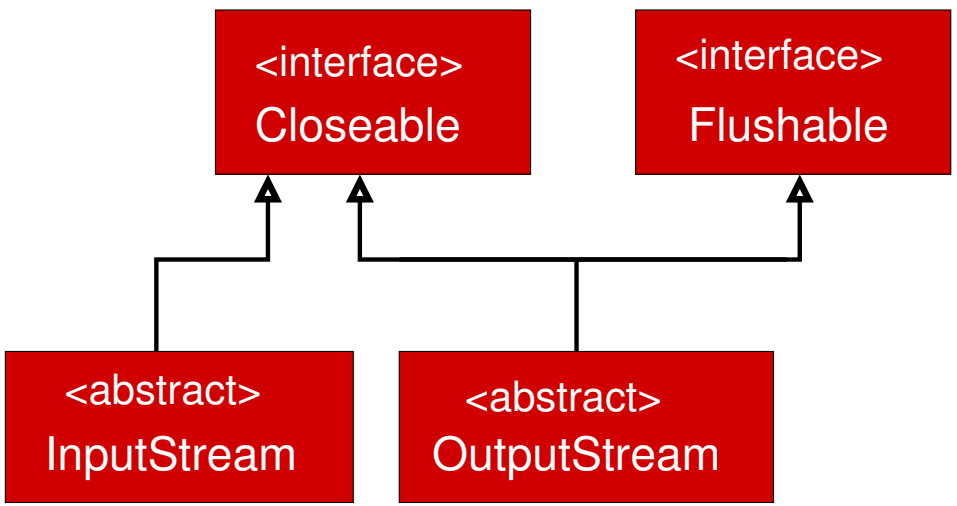
```
void write(byte[] b)
void write(int b)
...
void writeBoolean(boolean val)
void writeByte(int val)
void writeChar(int val)
void writeShort(int val)
void writeInt(int val)
void writeLong(long val)
void writeFloat(float val)
void writeDouble(float val)
void writeByte(String s)
void writeChars(String s)
void writeUTF(String s)
...
```

Beispiel: Kopieren eines Datei-Abschnittes

```
  // Variablen definieren, Dateinamen einlesen
  input = new RandomAccessFile(originalFile,"r");
  output = new RandomAccessFile(outFile,"rw");
  ...
  input.seek(21);
  for (i=0; i<10; i++) {
   byte b = input.readByte();
   output.writeByte(b);
  }
  output.seek(0);
  for (i=0; i<output.length(); i++) {
   byte e = output.readByte();
   System.out.printf("%c",e);
  }
  input.close();
  output.close();
}
```

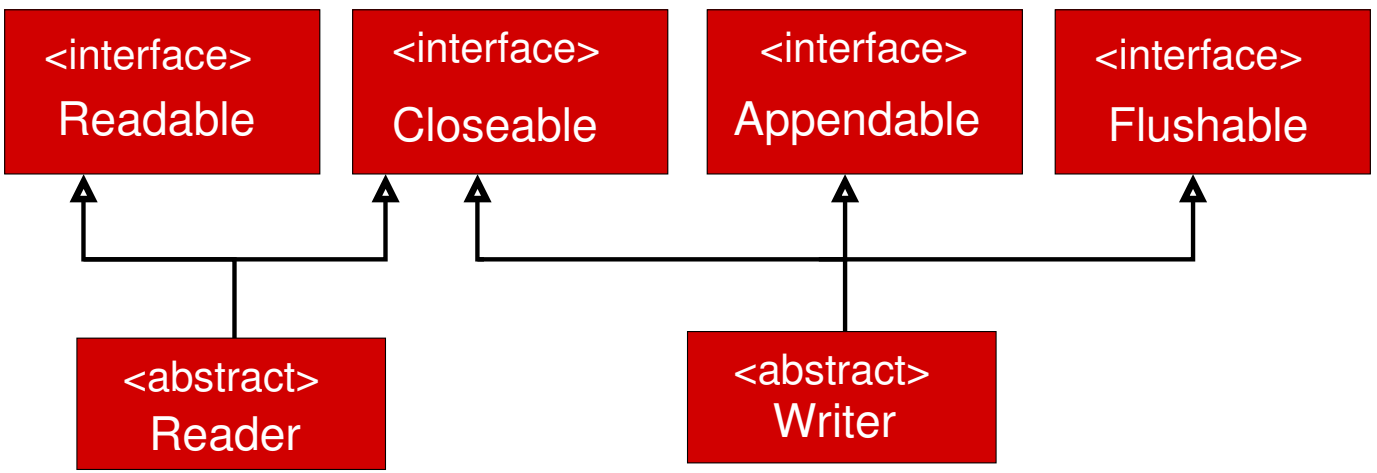
2. Sequentieller Zugriff (Bytes)

Basisklasse für	Bytes (auch Byte-Arrays)	Character (auch Character-Arrays)
Eingabe	InputStream	Reader
Ausgabe	OutputStream	Writer



Sequentieller Zugriff (Characters)

Basisklasse für	Bytes (auch Byte-Arrays)	Character (auch Character-Arrays)
Eingabe	InputStream	Reader
Ausgabe	OutputStream	Writer



Stromarten (Streams) in Java

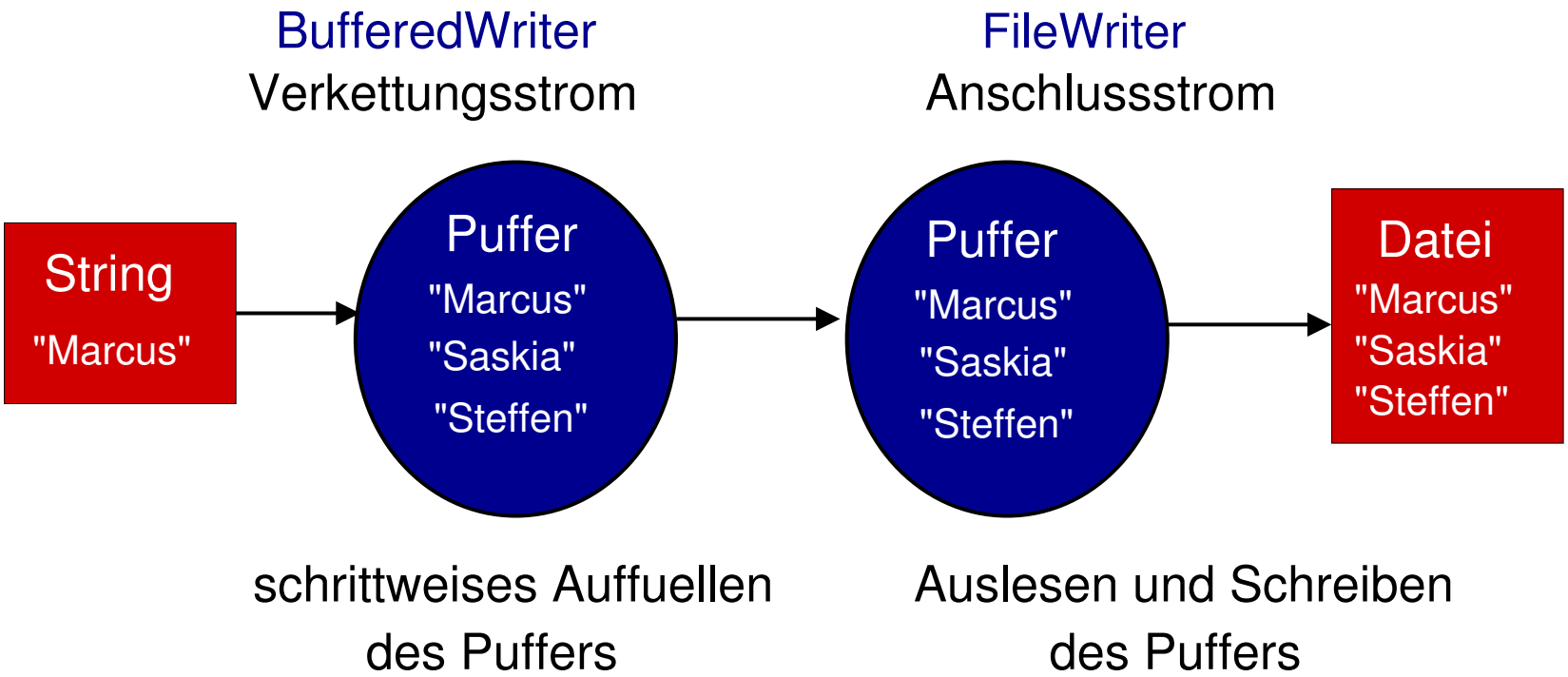
Anschlussstrom

- bildet Verbindung zu Dateien
- bietet *low-level* Methoden zum Lesen/Schreiben von Bytes

Verkettungsstrom

- bildet Verbindung zum Objekt
- bietet *high-level* Methoden zum Lesen/Schreiben
- muss mit anderen Strömen verkettet werden, z.B. *BufferedWriter* mit *FileWriter*

Beispiel: Puffern beim Speichern



Klassen für die Behandlung von Eingaben

Byte-Stream	Character-Stream	Funktionalität
InputStream	Reader	Basis (abstrakte Klassen)
BufferedInputStream	BufferedReader	Puffern der Eingaben
LineNumberInputStream	LineNumberReader	Ergänzung um Zeilennummern
ByteArrayInputStream	CharArrayReader	Lesen von Arrays
FilterInputStream	FilterReader	Filtern (abstrakte Klassen)
PushbackInputStream	PushbackReader	Rückgabe bereits gelesener Zeichen
PipedInputStream	PipedReader	Lesen von *Piped*-Ausgabestreams
SequenceInputStream		Verbinden mehrerer InputStreams
	InputStreamReader	Umwandlung von Byte zu Character
	StringReader	Lesen aus Strings

Klassen für die Behandlung von Ausgaben

Byte-Stream	Character-Stream	Funktionalität
OutputStream	Writer	Basis (abstrakte Klassen)
BufferedOutputStream	BufferedWriter	Ausgabe von Puffern
ByteArrayOutputStream	CharArrayWriter	Schreiben in Arrays
FileOutputStream	FileWriter	Schreiben in Dateien
PrintStream	PrintWriter	Konvertieren Primitive in Strings
PipedOutputStream	PipedWriter	Schreiben in *Pipes*
	StringWriter	Schreiben in Strings
	OutputStreamWriter	Umwandlung von Character zu Byte

Teil III der Vorlesung PROG 2

Ein- und Ausgabestreams

Serialisierung

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Speichermöglichkeiten für Programmdaten

Textdateien

- kompatibel mit anderen Programmen
- Speicherung aller Daten als Zeichenfolge mit Delimitern getrennt (Komma, Semikolon)
- alternativ als XML-Daten (hierarchische Strukturierung)

Serialisierung

- kompatibel nur mit Java-Programmen
- *Flachklopfen* der Objekte eines Programms zum Speichern
- *Rekonstruktion* der flachen Daten zu Objekten beim Einlesen
- Java-Unterstützung durch *Interface Serializable*

Serialisierung von Java-Objekten

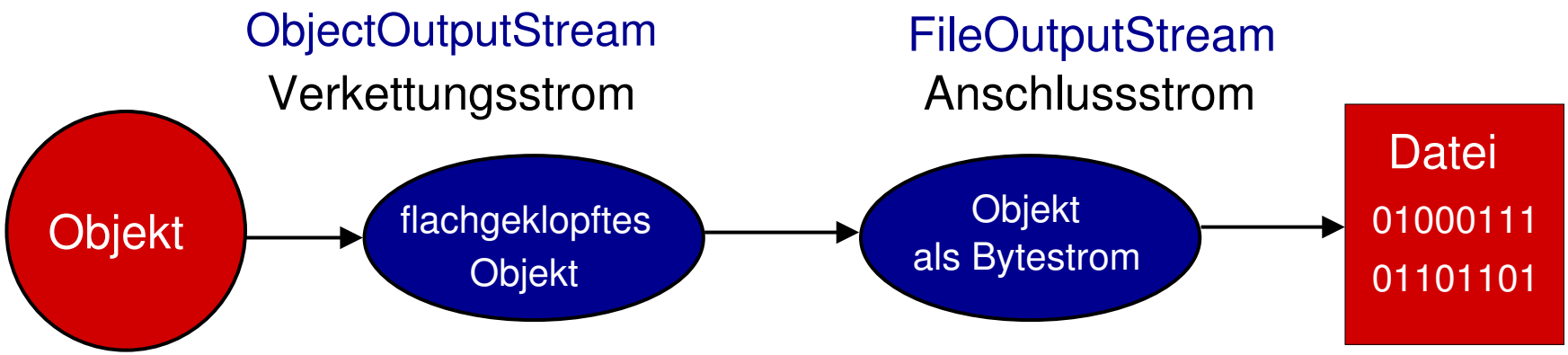
Vorteile einer Serialisierung

- bei Speicherung in Textdatei gehen Informationen über Datentypen verloren
- Lösung: Speichern von expliziten Metainformationen

Was wird von einem Objekt gespeichert?

- elementare Werte von (Instanz-)variablen
- Objekte, die über (Instanz-)variablen referenziert werden
- Teilinformationen über die Klasse des Objekts oder von Klassen referenzierter Objekte

Serialisierung: Konzeptueller Ablauf



Serialisierung mit einfachen Datentypen

```
// Anschlussstrom zu einer Datei
FileOutputStream fs = new FileOutputStream("Test.ser");

// Verkettungsstrom: vom Objekt zum Anschlussstrom
ObjectOutputStream os = new ObjectOutputStream(fs);

// Objekte Speichern
os.write(new Student ("Steffen"));
os.write(new Student ("Marcus"));

// Verkettungsstrom schliessen
// (inklusive Anschlussstrom)
os.close();
```

Serialisierung mit Objektreferenzen

```
// Objekte der Klasse Student sind serialisierbar
class Student implements Serializable {
 String name;
 int matrikel;

 Student (String n, int m) {
  name = n;
  matrikel = m;
 }
}

// Studentendatenbasis von Prog 2 ist serialisierbar
class Prog2 implements Serializable {
 Student[] studenten = new Student[100];
 ...
 void add (Student s) { ... }
}
```

Speicherung der gesamten Datenbasis

```
public class SerializableProg2 {
 public static void main(String[] arg) {
 // neues Veranstaltungsobjekt für Prog 2
 Prog2 ss12 = new Prog2();
 ss12.add(new Student("Steffen",129539));
 ...
 try {
  FileOutputSream fs = new FileOutputStream("test.ser");
  ObjectOutputStream os = new ObjectOutputStream(fs);
  os.writeObject(ss12);
  os.close();
  } catch (IOException e) { ... }
 }
}
```

Beispiel: Wo ist der Fehler?

```
public class Student {
 String name; int matrikel;
 Student (String n, int m) {
  name = n;
  matrikel = m; }

public class Prog2 implements Serializable {
 Student[] studenten = new Student[100];
 ... }

public class SerializableProg2 {
 ...
 Prog2 ss12 = new Prog2();
 ss12.add(new Student("Steffen",129539));

 os.writeObject(ss12);
}
```

Wenn Serializable vergessen wurde ...

```
public class Student { // Student nicht serialisierbar
 String name; int matrikel;
 Student (String n, int m) {
  name = n;
  matrikel = m; }

public class Prog2 implements Serializable {
 Student[] studenten = new Student[100];
 ... }

public class SerializableProg2 {
 ...
 Prog2 ss12 = new Prog2();
 ss12.add(new Student("Steffen",129539));

 os.writeObject(ss12); // NotSerializableException
}                      // wird geworfen
```

Ausnahmen: Variablen nicht serialisieren

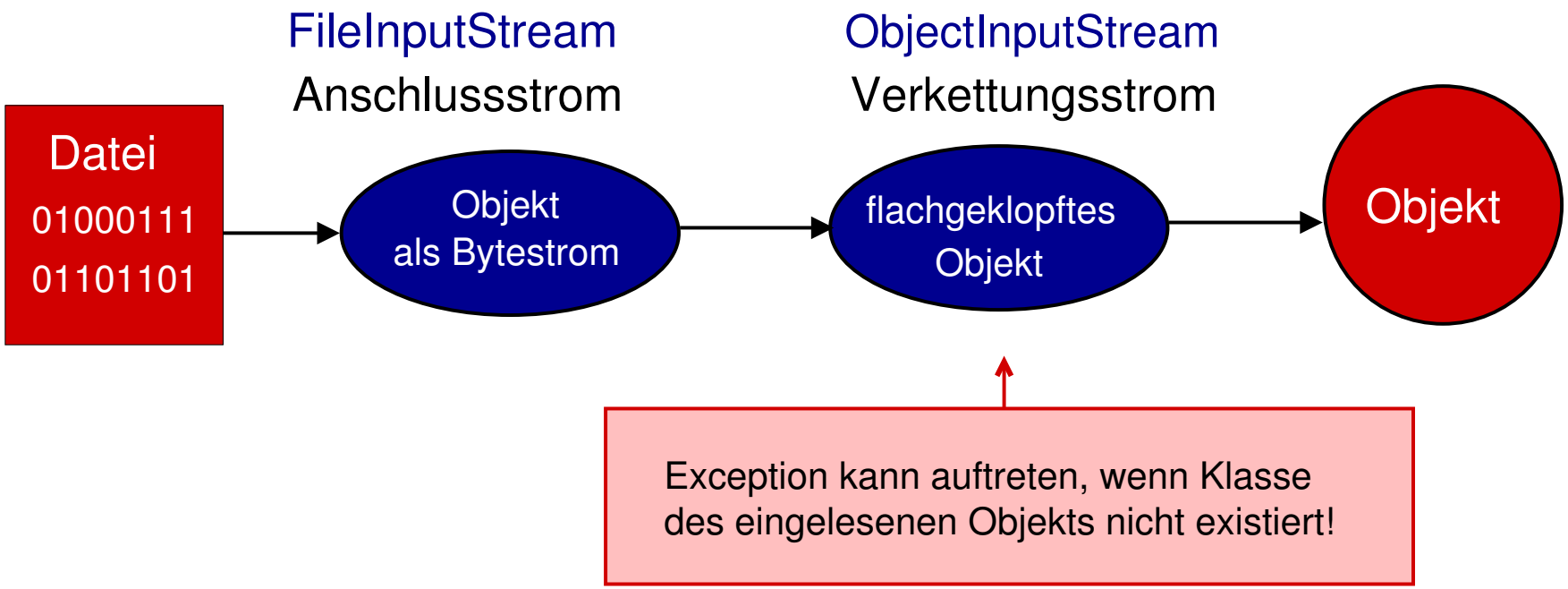
```
public class Student implements Serializable{
 String name; int matrikel;
 transient int note; // Note nicht serialisieren
 Student (String n, int m) {
  name = n;
  matrikel = m; }

public class Prog2 implements Serializable {
 Student[] studenten = new Student[100];
 ... }

public class SerializableProg2 {
 ...
 Prog2 ss12 = new Prog2();
 ss12.add(new Student("Steffen",129539));

 os.writeObject(ss12); // keine Exception mehr
}
```

Deserialisierung: Konzeptueller Ablauf



Beispiel: Deserialisierung in Java

```
// Anschlussstrom zu einer Datei
FileInputStream fs = new FileInputStream("Test.ser");

// Verkettungsstrom: vom Objekt zum Anschlussstrom
ObjectInputStream os = new ObjectInputStream(fs);

// Objekte in der gleichen Reihenfolgen
// wie beim Schreiben einlesen
Object o1 = os.readObject();
Object o2 = os.readObject();

// Objekte casten
Student student1 = (Student) o1;
Student student2 = (Student) o2;

os.close();
```

Serialisierung bei Vererbung

```
public class Student {
 String name; int matrikel;
 Student() {} // Konstruktor ohne Argument ist nötig
 Student(String n, int m) {
  name = n; matrikel = m; }
}
public class StudentProg2 extends Student
                          implements Serializable {
  int gesamtNote;
  StudentProg2(String n, int m, int note) {
   super(n,m); gesamtNote = note}}
...
 StudentProg2 s1 = new StudentProg2("Steffen",129539,1);
 os.writeObject(s1);
 s1New = (StudentProg2) is.readObject();
}
```

Welche Belegung haben die Instanzvariablen von s1New?

name = null, matrikel = 0, gesamtNote = 1

Regeln für Serialisierung mit Vererbung

(1) Unterklassen von serialisierbaren Klassen sind serialisierbar

(2) Serialisierung in (1) kann durch neue Definition von *readObject* und *writeObject* (explizites Werfen der Exception *NotSerializableException*) verhindert werden

(3) private Instanzvariablen einer *nicht* serialisierbaren Oberklasse einer serialisierbaren Klasse werden nicht serialisiert

(4) Deserialisierung in (3) ist nur möglich, wenn die nicht serialisierbare Oberklasse über einen Konstruktor ohne Argumente verfügt

(5) Wiederherstellen öffentlicher Instanzvariablen der Oberklasse in (3) ist nur bei neuer Definition der Methoden *readObject* und *writeObject* möglich

Teil III der Vorlesung PROG 2

Ein- und Ausgabestreams

Datenkomprimierung

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Datenkomprimierung

Einordnung

- Reduktion des Datenvolumens und Beschleunigung der Gesamtlaufzeit
- Java kann Zip und GZip-Komprimierung (LZW-Algorithmus)

```
byte[] buf = new byte[4096];
FileOutputStream fs = FileOutputStream(outFile);
ZipOutputStream zs = new ZipOutputStream(fs);

FileInputStream in = new FileInputStream(inFile);
zs.putNextEntry(new ZipEntry(inFile));
int len;
 while ((len = in.read(buf)) > 0)
        zs.write(buf, 0, len);

in.close();
zs.close();
```

Unterschiede zwischen Zip und GZip/GunZip

Zip-Archive

- *unterstützt Archivierung*, d.h. bündelt in einem Archiv mehrere Dateien
- jede Datei wird separat komprimiert
  $\Rightarrow$ Dateien sind immer einzeln zugreifbar

GZip/GunZip

- *unterstützt keine Archivierung*, d.h. nur zum Komprimieren einzelner Dateien
- wird oft mit *tar* verwendet, ein separates Softwareprogramm zur Archivierung (ohne Komprimierung)
- das gesamte *tar*-Archiv wird dann als eine Datei komprimiert
  $\Rightarrow$ kein Dateizugriff ohne Dekomprimierung des Archivs

Teil IV der Vorlesung PROG 2

Multithreading

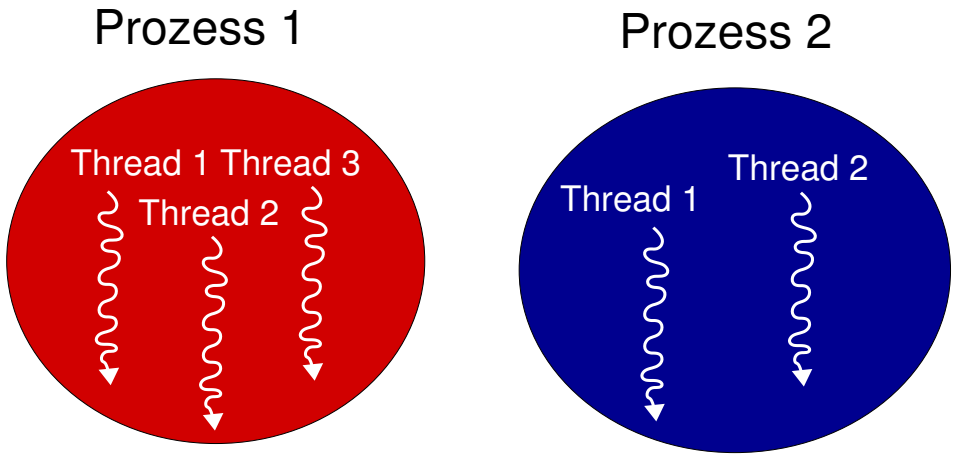
Motivation

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Nebenläufige Programme mit Threads

Definition Threads

- auch leichtgewichtige Prozesse
- mehrere parallel ablaufende Aktivitäten innerhalb eines Prozesses



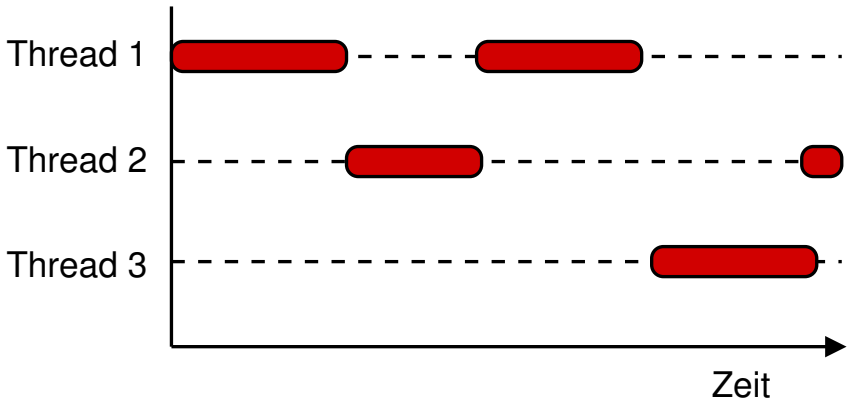
Technische Besonderheiten

- jeder Thread realisiert einen sequentiellen Kontrollfluss, d.h. besitzt seinen eigenen Programmzähler
- im Gegensatz zu Prozessen gemeinsame Ressourcen, d.h. Threads besitzen gemeinsamen Adressraum

Prozessorbelegung für Threads

Realisierung

- nebenläufige Programme führen mehrere Rechenschritte gleichzeitig (simultan) durch
- Abarbeitung erfolgt verschränkt, oft auf einem Prozessor



Vorteile von Threads

- schnelles Umschalten zwischen Anwendungen auf unterschiedlichen Threads
- gemeinsame Nutzung von Ressourcen ist einfach

Probleme beim Einsatz von Threads

Nachteile

- Programmausführung zeigt *oft nicht-deterministisches Verhalten* (schwierig beim Test nebenläufiger Programme)
- *Konflikte beim Schreiben gemeinsamer Ressourcen*, wie z.B. beim Zugriff auf gemeinsame globale Variable oder eine Datei

Lösung

- Einsatz verschiedener Synchronisationsmechanismen, um zumindest teilweise deterministisches Systemverhalten zu erzwingen
- nur kontrollierten Zugriff auf gemeinsame Variablen erlauben, z.B. mit Hilfe von Semaphoren