14.3.1 Ausgabe Byte-Streams

- Basis der Ausgabe-Streams ist die abstrakte Klasse OutputStream
- Sie stellt folgende Methoden zur Verfügung:
  - abtract public void write(int b)
    - Gibt lediglich die unteren 8 Bit aus und ignoriert alle übrigen.
    - Dies ist die einzige abstrakte Methode in der Klasse.
  - public void write(byte[] b)
    - Schreibt das Array in den Stream.
  - public void write(byte[] b, int offs, int len)
    - Schreibt den Teil des Arrays zwischen offs und offs+len-1 in den Stream.
  - public void flush()
    - Schreibt die gepufferten Daten auf das Ausgabegerät und leert alle Puffer.
  - public void close()
    - Schließt den Stream (und leert den Puffer). Danach sind keine weiteren Operationen erlaubt.
- **All diese Methoden können eine IO Exception werfen**

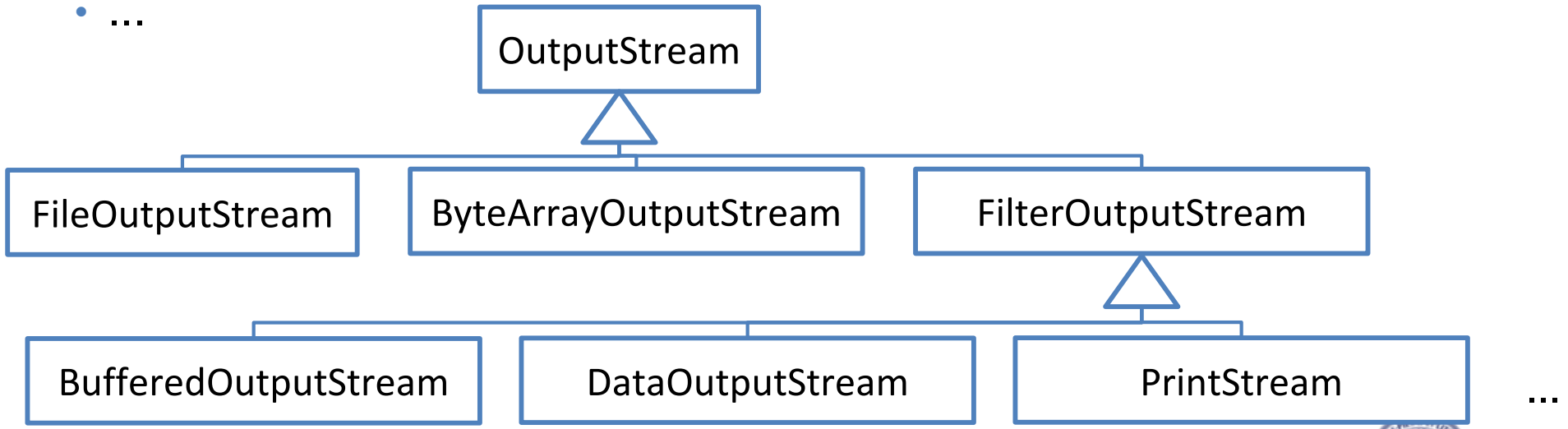
So kann das Argument als ein nicht-Vorzeichen-behafteter Wert angesehen werden.

Verschachtelung von Ausgabeströmen

- Was gibt es für Unterschiede zwischen Ausgabeströmen?
  - Schreiben in Datei
  - Schreiben in Byte-Array
  - Schreiben von rohen Byte-Folgen
  - Schreiben von primitiven Datentypen
  - Pufferung der Daten
  - ...

Verschiedene Ziele („Senken“)

Verschiedene Hilfsfunktionen („Filter“)



Verschachtelung von Ausgabeströmen

- Senken Ausgabeströme
  - Können eigenständig erzeugt und verwendet werden
  - Konstruktoren benötigen nur Angabe über Ziel des Bytestroms (z.B. Dateiname)
- Filter Ausgabeströme
  - Benötigen einen anderen Ausgabestrom, dem Hilfsfunktionalität hinzugefügt werden soll
  - Konstruktoren benötigen diesen Ausgabestrom
    - Alle Schreiboperationen werden dann intern an diesen Stream delegiert.

      → Typisches Muster der Programmierung
  - Schachtelung kann beliebig lang sein

Beispiele

ByteArrayOutputStream kann direkt erzeugt werden.

```
public static void main(String[] args) throws IOException {
    ByteArrayOutputStream byteOS = new ByteArrayOutputStream();
    BufferedOutputStream bufferedOS = new BufferedOutputStream(byteOS, 16);
    DataOutputStream dataOS = new DataOutputStream(bufferedOS);
}
```

dataOS: write-Methoden für Standard-Datentypen

bufferedOS: Zwischenspeicherung von Daten

byteOS: Speicherung des Datenstroms in byte[]

Beispiele

ByteArrayOutputStream kann direkt erzeugt werden.

```
public static void main(String[] args) throws IOException {
    ByteArrayOutputStream byteOS = new ByteArrayOutputStream();
    BufferedOutputStream bufferedOS = new BufferedOutputStream(byteOS, 16);
    DataOutputStream dataOS = new DataOutputStream(bufferedOS);

    dataOS.writeInt(4);
    dataOS.writeInt(3);
    dataOS.writeInt(2);
    dataOS.writeInt(1);
    System.out.println(byteOS.toByteArray().length);
    dataOS.writeInt(0);
    System.out.println(byteOS.toByteArray().length);

}
```

writeInt-Methode wird durch DataOutputStream hinzugefügt.

Ergibt „0“, da durch den BufferedOutputStream Daten noch zwischengespeichert werden.

Ergibt „16“, da der Puffer inzwischen voll war und die Daten (Bytes) an byteOS durchgeschrieben wurden.

FileOutputStream

- Erweiterung von OutputStream
  - Schreiben der Daten in eine Datei
- Konstruktoren
  - public FileOutputStream(String name) throws FileNotFoundException
  - public FileOutputStream(String name, boolean append) throws FileNotFoundException
  - public FileOutputStream() throws IOException
- FileNotFoundException wird „geworfen“, wenn
  - die Datei nicht existiert
  - die Datei existiert, aber die Datei ist ein Verzeichnis.
  - die Datei existiert, aber nicht erzeugt oder geöffnet werden kann.

Beispiel: FileOutputStream

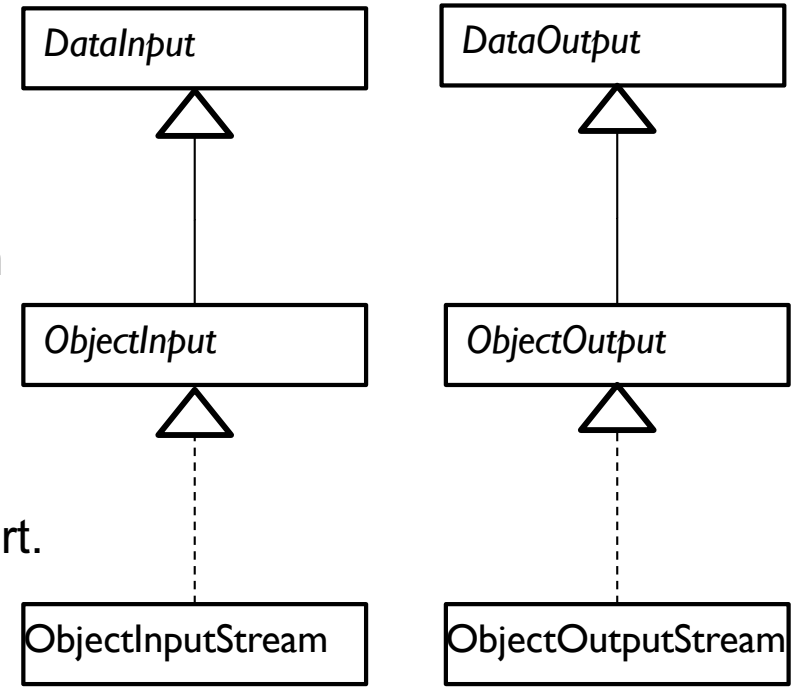
```
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutStreamTest {
    public static void main(String[] args) throws IOException {
        FileOutputStream out = null;
        try {
            out = new FileOutputStream(args[0], true);
            for (int i = 0; i < 256; ++i)
                out.write(i);
        }
        catch (IOException e) {
            e.printStackTrace();
            System.exit(1);      // Beendet das Programm
        }
        finally {
            out.close();             // Schließen der Datei als letzte Aktion
        }
    }
}
```

Dateiname

also anhängen!

ObjectOutputStream

- **Besonderheiten der Klasse**
  - primitive Datentypen und komplette Objekte (inklusive aller referenzierten Objekte) können binär ausgegeben werden.
    - Die Objekte müssen jedoch die Schnittstelle Serializable implementieren.
  - Beispiel für einen **FilterOutputStream**
  - Die Schnittstelle DataOutput wird implementiert.
  - ObjectOutputStream ist die Basis für die Serialisierung in Java.
- Eine andere Möglichkeit Daten binär in einem Strom auszugeben, wird durch die Klassen DataOutputStream angeboten.
  - Diese Klasse implementiert ebenfalls DataOutput
  - Erlaubt aber nicht direkt das Serialisieren beliebiger Objekte

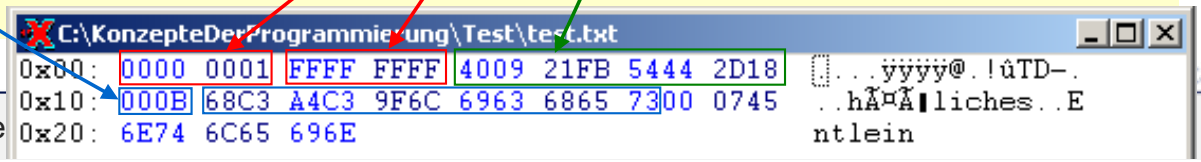


Beispiel

```
public void writeToFile(String fname) throws IOException {
// Der Einfachheit halber reichen wir die Exception weiter.
   Point p = new Point(1.4, 3.14);
   // Schreiben
   FileOutputStream fos = new FileOutputStream(fname,true);
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   oos.writeObject(p);
   oos.close();

   // Lesen – Erklärung noch später
   ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fname));
   Point q = (Point) ois.readObject();
   System.out.println("Ausgabe: " + q);
   ois.close();
}
```

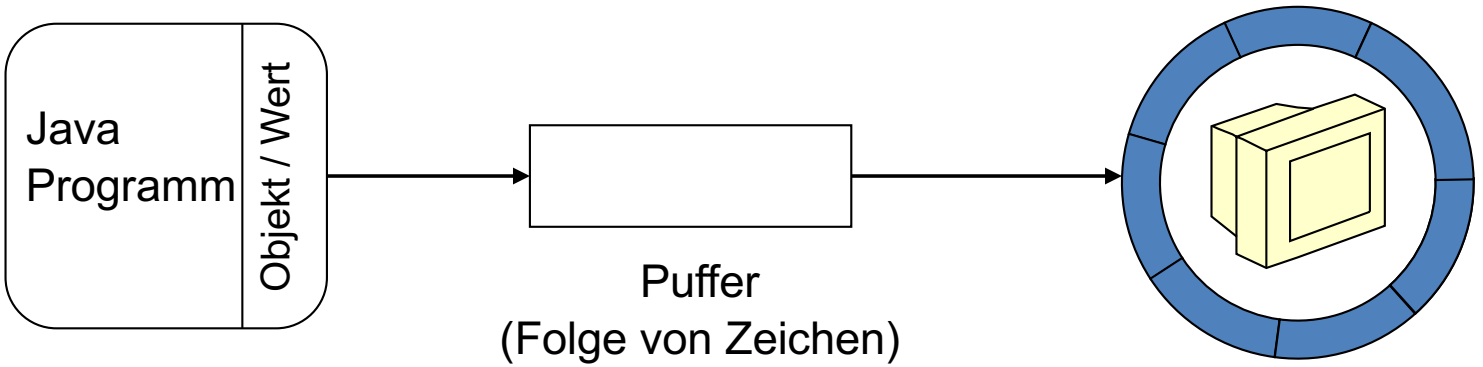
Ausgabe im Binärformat

```
import java.io.*;

public class DataOutputStreamTest {
  public static void main(String[] args) {
    try {
      DataOutputStream out = new DataOutputStream(
                             new BufferedOutputStream(
                             new FileOutputStream("test.txt")));
      out.writeInt(1);
      out.writeInt(-1);
      out.writeDouble(Math.PI);
      out.writeUTF("häßliches");
      out.writeUTF("Entlein");
      out.close();
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



Ausgabe mit der Klasse PrintStream

- Die Klasse **System** bietet eine Referenzvariable **out** mit einem Objekt der Klasse **PrintStream** an, das einen zeichenorientierten Bildschirm modelliert.



- Auswahl von Methoden aus PrintStream
  - print (<Datentyp> x); // x wird in den Puffer geschrieben
  - flush(); // Der Puffer wird geleert.
  - println(<Datentyp> x); // entspricht: print(x); print('\n');

Ausgabe mit der Klasse PrintStream

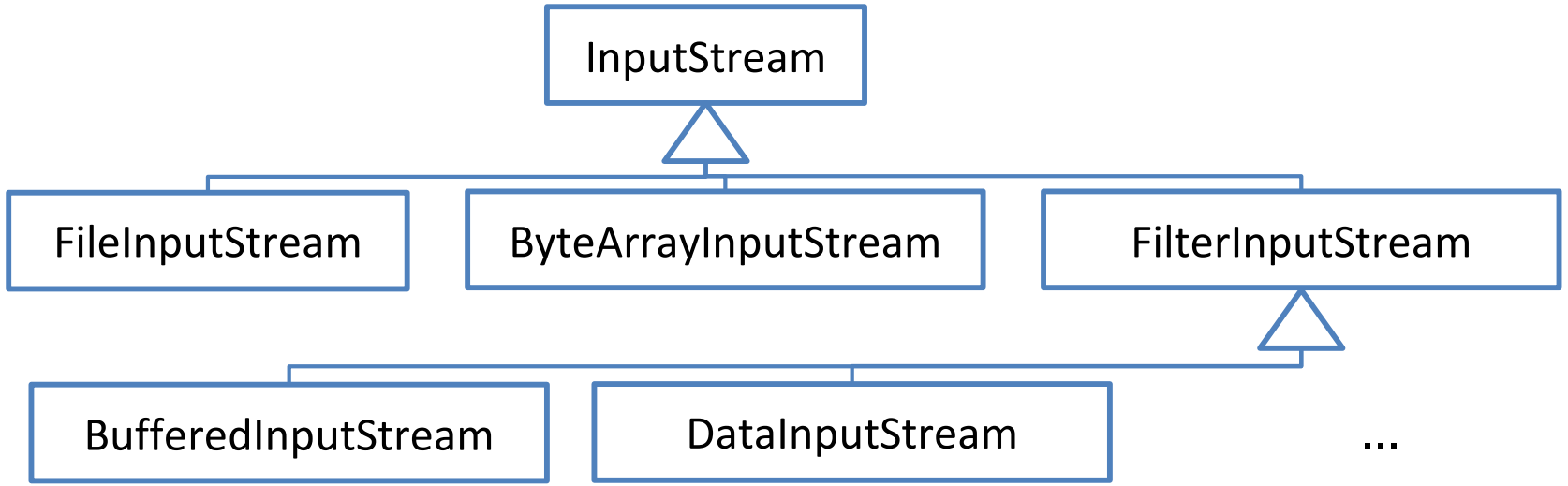
- PrintStream ist ein **FilterOutputStream**
  - Bietet Konstruktor PrintStream(OutputStream)
  - Fügt so dem zugrunde liegenden Ausgabestrom die Möglichkeit hinzu eine Menschen-lesbare (Text-basierte) Darstellung von Werten zu erzeugen
- PrintStream bietet zusätzlich Konstruktoren, um den Datenstrom direkt in eine Datei zu schreiben
  - PrintStream(File datei), PrintStream(String dateName)
  - Intern wird dann ein FileOutputStream erzeugt, an den der Datenstrom weitergegeben wird.

14.3.2 Eingabe Byte-Streams

- Basis der Eingabe-Streams ist die abstrakte Klasse InputStream.
- Sie stellt unter anderem folgende Methoden zur Verfügung:
  - public abstract int read() throws IOException
    - Diese Methode muss in den abgeleiteten Klassen implementiert werden.
  - public int read(byte[] b) throws IOException
    - Liest die Daten aus dem Stream in das Array
  - public int read(byte[] b, int off, int len) throws IOException
    - Liest len-off+1 Bytes aus der Eingabe und überträgt sie in das Array.
  - public void close() throws IOException
    - Schließt den Stream. Danach sind keine weiteren Operationen erlaubt.
  - public void reset() throws IOException
    - Setzt die Lesemarke auf den Anfang des Streams.

Verschachtelung von Eingabeströmen

- Analog zu Ausgabeströmen gibt es:
  - Eingabeströme, die direkt Daten einlesen („Quellen“)
  - Filter-Eingabeströme (FilterInputStream), die Funktionalität hinzufügen



```
import java.io.FileInputStream;
import java.io.FileOutputStream;

public class FileCopy {
  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("java FileCopy inputfile outputfile");
      System.exit(1);
    }
    try {
      FileInputStream in = new FileInputStream(args[0]);
      FileOutputStream out = new FileOutputStream(args[1]);
      byte[] buf = new byte[4096];
      int len;
      while ((len = in.read(buf)) > 0)
          out.write(buf, 0, len);
      out.close();
      in.close();
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Ist der Aufruf korrekt?

beide Ströme initialisieren

je 4096 Byte lesen und schreiben (bis auf den letzten Zugriff)

SequenceInputStream

- Ein SequenceInputStream dient dazu, zwei oder mehr InputStreams so miteinander zu verbinden, dass die Daten nacheinander aus den einzelnen Streams gelesen werden.
- Die beteiligten Streams können direkt an den Konstruktor übergeben werden:
  - public SequenceInputStream(InputStream s1, InputStream s2)
- Darüber hinaus wird es in der Klasse ermöglicht, dieses Prinzip auf beliebig viele InputStreams anzuwenden.

ObjectInputStream und DataInput

- Analog zu ObjectOutputStream gibt es eine Klasse ObjectInputStream, mit der die geschriebenen Daten eingelesen werden können.
- Beispiel für einen **FilterInputStream**
- ObjectInputStream implementiert das Interface DataInput, das folgende Methoden definiert:
  - …
  - long readLong() throws IOException
  - double readDouble() throws IOException
  - Object readObject() throws IOException
  - …
  - String readUTF() throws IOException

Beispiel

- Der folgende Konstruktor der Klasse Point liest die Daten aus der Datei und initialisiert damit das Objekt.

```
public Point(String fname) throws IOException {
      FileInputStream fis = new FileInputStream(fname);
      ObjectInputStream ois = new ObjectInputStream(fis);

      Point p = (Point) ois.readObject();
      // Klasse Point muss Serializable implementieren!

      ois.close();
  }
```

Zusammenfassung

- Einführung in das Paket java.io
  - Verwaltung von Dateien
    - Klasse File
  - Wahlfreier Zugriff auf Dateien
    - RandomAccessFile
  - Sequentieller Zugriff mit Datenströmen
- Datenströme unterscheiden zwischen
  - Byte-Datenströme
    - Klassen InputStream und OutputStream
  - Character-Datenströme
    - Reader und Writer
    - Funktionalität zum Lesen und Schreiben von Textdateien