Datenströme in Java

Arten von Strömen

Ein- und Ausgabeoperationen sind in Java im Wesentlichen über Datenströme realisiert. Dabei werden zwei Arten von Datenströmen unterschieden; byteorientierte Ströme sind von den abstrakten Klassen InputStream bzw. OutputStream abgeleitet, zeichenorientierte Ströme sind von den abstrakten Klassen Reader bzw. Writer abgeleitet.

Physikalische Ströme

Für die typischen Ein- und Ausgabesituationen sind von den oben genannten Klassen Unterklassen abgeleitet, die die jeweilige Anforderung implementieren.

	Byteströme		**Zeichenströme**	
Aufgabe	**Eingabe**	**Ausgabe**	**Eingabe**	**Ausgabe**
Dateibearbeitung	FileInputStream	FileOutputStream	FileReader	FileWriter
Datenfeld	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader StringReader	CharArrayWriter
Verbindung	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Netzwerk	socket. getInputStream()	socket. getOutputStream ()		
Konvertierung			InputStreamReader	OutputStreamWriter

Spezielle Ströme

Die grundlegenden Klassen stellen Methoden zur byte- bzw. zeichenweisen Ein- bzw. Ausgabe zur Verfügung (*read* bzw. *write*). Um zusätzliche Fähigkeiten bereit zu stellen, wird das so genannte Ausschmückungsmuster (adornment pattern) verwendet. Bei diesem Entwurfsmuster werden neue Eigenschaften in Unterklassen bereit gestellt, die ein Objekt der Basisklasse als Attribut im Konstruktor mit bekommen und auf diesem operieren können. Dadurch können die Ausschmückungen beliebig kombiniert werden. Typische Ausschmückungen sind das gepufferte Bearbeiten der Ströme (reduziert die echten Ein- bzw. Ausgabeoperationen) und das bereit Stellen von Methoden für weitere Datentypen. Insbesondere stellt BufferedReader eine Methode *readLine* zum Lesen ganzer Zeilen zur Verfügung.

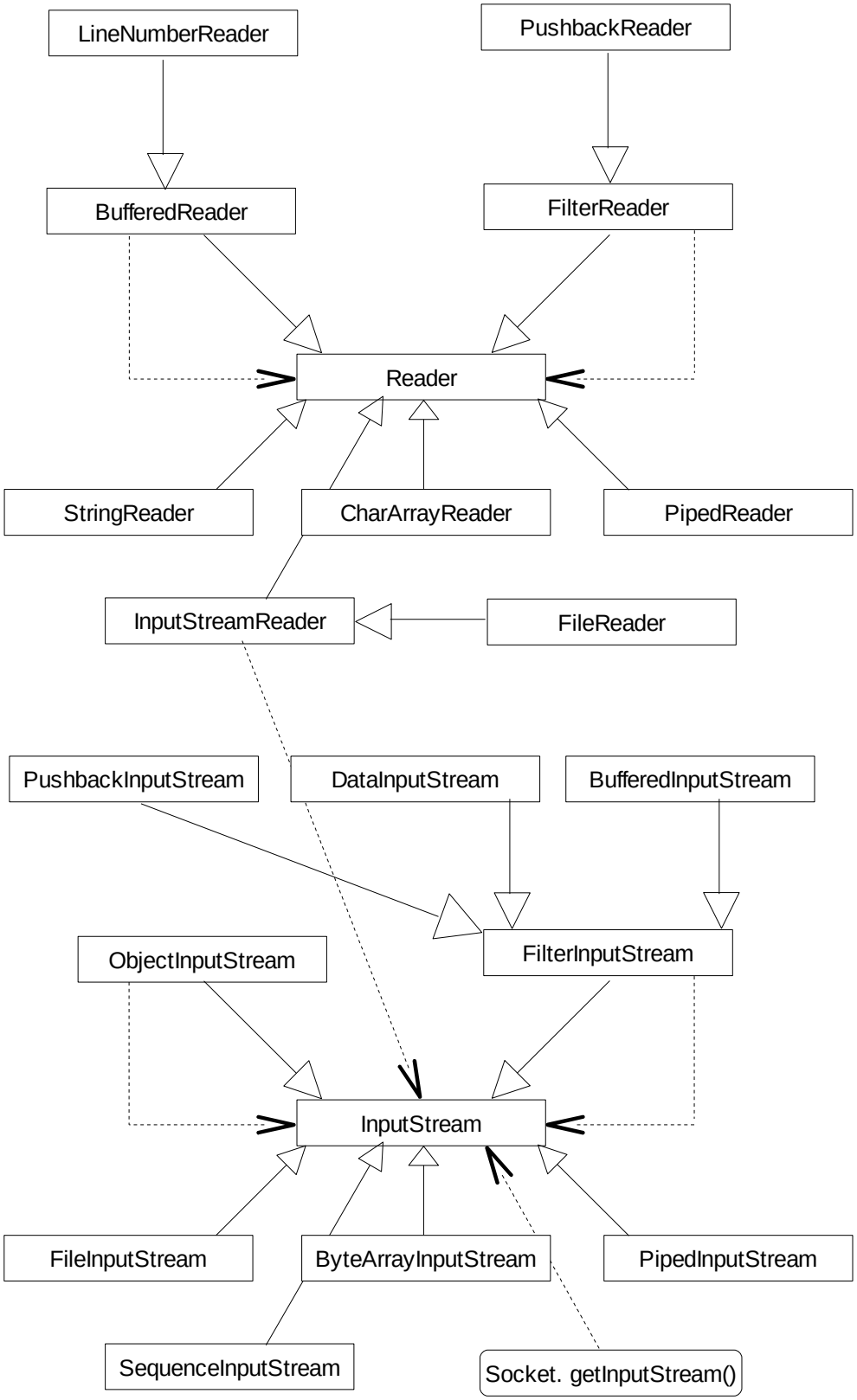
Eine wichtige Rolle bilden die „Print“-Ströme (PrintStream bzw.PrintWriter). Sie bieten Methoden an, um Werte von Standarddatentypen „lesbar“ darzustellen. Dabei können sowohl Standardformate als auch selbst definierte Formate verwendet werden. Auch die Ausgabe von Zeilenwechseln wird von diesen Strömen zur Verfügung gestellt.

	Byteströme	
Aufgabe	**Eingabe**	**Ausgabe**
Pufferung	BufferedInputStream	BufferedOutputStream
Standarddatentypen	DataInputStream	DataOutputStream
Objekte	ObjectInputStream	ObjectOutputStream
Lesen rückgängig machen	PushbackInputStream	
Lesbare Ausgabe		PrintStream

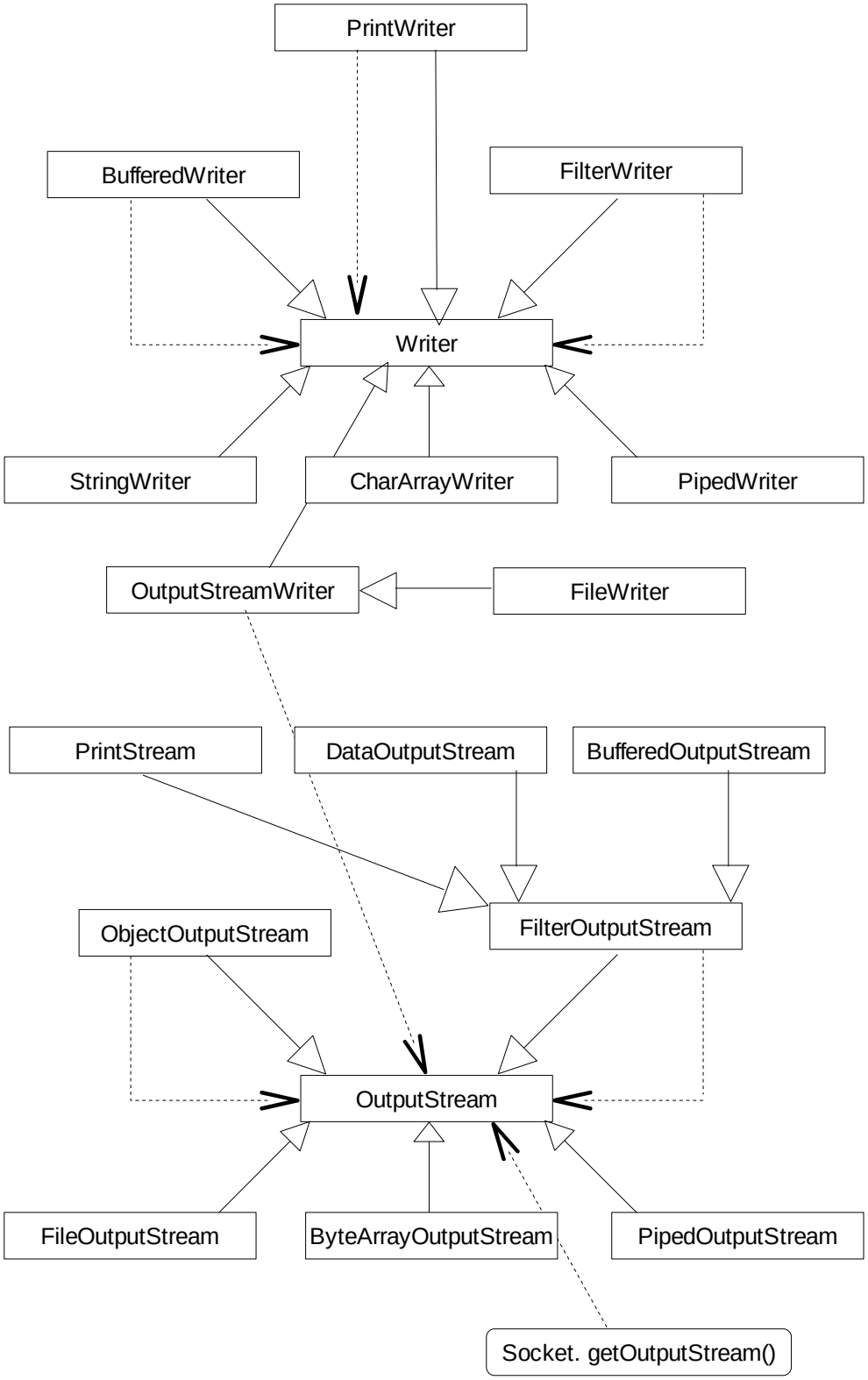
	Zeichenströme	
Aufgabe	**Eingabe**	**Ausgabe**
Pufferung	BufferedReader	BufferedWriter
Zeilenzählung	LinenumberReader	
Lesen rückgängig machen	PushbackReader	
Lesbare Ausgabe		PrintWriter

Zeichenkonvertierung

Zeichenorientierte Ströme nutzen die byteorientierten Ströme. Die Abbildung der Unicode-kodierten Zeichen auf Bytefolgen erfolgt über angebbare, standardisierte Zeichensätze in den Klassen InputStreamReader bzw. OutputStreamWriter. Bei allen impliziten Verwendungen wie in den Klassen FileReader bzw. FileWriter wird automatisch der systemtypische Zeichsatz für die Konvertierung zu Grunde gelegt. Bei Netzwerkverbindungen sollte man hier systemunabhängige Darstellungen wie UTF-8 oder UTF-16 verwenden.



Übersicht über die Eingabeströme



Übersicht über die Ausgabeströme

Typische Beispiele
Im folgenden wird die Verwendung der Ströme für typische Ein- und Ausgabeaufgaben einschließlich der Datenübertragung im Netz exemplarisch dargestellt. Wo angebracht, geben die Beispiele auch Alternativen mit eventuellen Bedeutungsunterschieden an.
Die Beispiele zeigen jeweils Ein- und Ausgabe zusammen auf.

Binäre Datenspeicherung auf Datei
Diese klassische Art der Datenspeicherung ist heute nicht mehr allgemein gebräuchlich, da Änderungen und Ergänzungen des Datenformat sehr aufwendig sind und für alle älteren Versionen eigene Eingabemethoden benötigt werden. Allerdings ist sie immer noch die kompakteste und schnellste Art der Datenspeicherung; für feste Formate wie bei Bildern, Audio- oder Videoinformation wird grundsätzlich binäre Speicherung verwendet. Zu einer Zeit ist auf einer Datei nur Lesen oder nur Schreiben möglich.

```
DataOutputStream out;
    out = new DataOutputStream (new FileOutputStream (<Dateiangabe>));
    out. writeInt (<Ganzzahlwert>);
    out. writeDouble (<Gleitkommazahlwert>);
    out. close ();

DataInputStream in;
    in = new DataInputStream (new FileInputStream (<Dateiangabe>));
    i = in. readInt ();
    d = in. readDouble ();
    in. close ();
```

Anmerkung:
Bei der oben genannten Form wird für jeden Methodenaufruf eine Dateioperation angestoßen. Effizienter ist die Verwendung gepufferter Ein- bzw. Ausgabe. Der Aufruf der Methode *flush* leert zuerst den Pufferspeicher und anschließen eventuell vorhandene Puffer der Dateiverwaltung.

```
    out = new DataOutputStream (new BufferedOutputStream (new FileOutputStream
           (<Dateiangabe>)));
    in = new DataInputStream (new BufferedInputStream (new FileInputStream
           (<Dateiangabe>)));
```

Datenspeicherung in einer Datei in lesbarer Darstellung
Diese Form ist leichter zu warten als binäre Form, insbesondere, wenn die Elemente zusammen mit ihrer Bedeutung beschrieben sind, wie das z. B. in XML-Dateien der Fall ist. Zahlwerte müssen aus der Zeichendarstellung wiederhergestellt werden.

```
PrintWriter out;
    out = new PrintWriter (new FileWriter (<Dateiangabe>));
    out. println (<Ganzzahliger Datenwert, evtl. mit Formatangabe>);
    out. close ();

BufferedReader in;
    in = new BufferedReader (new FileReader (<Dateiangabe>));
    zeile = in. readLine ();
    try
    {
        i = Integer. parseInt (zeile);
    }
    catch (Exception e)
    {
        i = 0;
    }
    in. close ();
```

Anmerkung 1:

Auch hier kann durch Puffern die Effizienz deutlich verbessert werden:

```
    out = new PrintWriter (new BufferedWriter (new FileWriter (<Dateiangabe>)));
```

Anmerkung 2:

```
    out = new PrintWriter (new FileWriter (<Dateiangabe>));
    in = new BufferedReader (new FileReader (<Dateiangabe>));
```

sind Abkürzungen für

```
    out = new PrintWriter (new OutputStreamWriter (new FileOutputStream
(<Dateiangabe>)));
    in = new BufferedReader (new InputStreamReader (new FileInputStream
(<Dateiangabe>)));
```

Das bedeutet, dass zur Zeichenkodierung die systemabhängige Standardkodierung verwendet wird. Falls sich die Daten nicht auf den 7-bit-ASCII-Zeichensatz beschränken, sollte eine systemunabhängige Kodierung explizit eingestellt werden, z. B.

```
    out = new PrintWriter (new OutputStreamWriter (new FileOutputStream (<Dateiangabe>),
"UTF-8"));
    in = new BufferedReader (new InputStreamReader (new FileInputStream (<Dateiangabe>),
"UTF-8"));
```

Übertragung von lesbaren Daten im Netz in normierter Darstellung

Die Daten werden über die *print*-Methode des Ausgabestroms erzeugt. Gelesen wird zeilenweise. Für die Übertragung wird als Zeichenkodierung UTF-8 festgelegt. Die Verbindung kann in beide Richtungen parallel genutzt werden. Gegebenenfalls muss mit einem Aufruf von *out. flush ()* garantiert werden, dass die Daten umgehend übertragen werden. Alternativ kann mit einem optionalen zweiten Parameter beim Erzeugen des Printwriter-Objekts angegeben werden, dass beim Ausgeben eines Zeilenwechsels (also insbesondere bei jedem Aufruf von *println*) der Ausgabestrom automatisch übetragen wird.

```
PrintWriter out;
    out = new PrintWriter (new OutputStreamWriter (sock. getOutputStream (), "UTF-8"));
    out. print (<Primitiver Datenwert, evtl. mit Formatangabe>);
    out. println (<Primitiver Datenwert, evtl. mit Formatangabe>);
    out. close ();

BufferedReader in;
    in = new BufferedReader (new InputStreamReader (sock. getInputStream (), "UTF-8"));
    zeile = in. readLine ();
    in. close ();
```

Anmerkung:

Der Konstruktor der Klasse PrintWriter kann auch direkt mit einem Ausgabestrom versorgt werden:

```
    out = new PrintWriter (sock. getOutputStream ());
```

Das ist eine Abkürzung für

```
    out = new PrintWriter (new OutputStreamWriter (sock. getOutputStream ()));
```

, d. h. Es wird die systemabhängige Standardkonvertierung verwendet.

Übertragung von Objekten im Netz

Voraussetzung ist jeweils ein existierendes und verbundenes Objekt sock der Klasse Socket. Das Objekt kann als Ergebnis eines Aufrufs von *accept* auf der Serverseite oder einer Instanzierung (mit gleichzeitiger Verbindungsherstellung) auf der Klientenseite entstehen. Die Verbindung kann in beide Richtungen parallel genutzt werden. Gegebenenfalls muss mit einem Aufruf von *out. flush ()* garantiert werden, dass die Daten umgehend übertragen werden.

```
ObjectOutputStream out;
    out = new ObjectOutputStream (sock. getOutputStream ());
    out. writeObject (<beliebiges Objekt>);
    out. close ();

ObjectInputStream in;
```

```
in = new ObjectInputStream (sock. getInputStream ());
meinObjekt = (MeineKlasse) in. readObject ();
in. close ();
```