

Inhalt

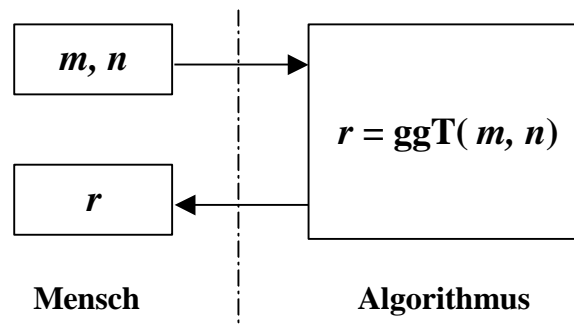
14	Dateiverwaltung, das Stream-Konzept.....	14-2
14.1	Datenströme	14-2
14.1.1	Datenströme in Java, Paket <code>java.io.*</code>	14-3
14.1.2	Standarddatenströme	14-4
14.2	Klasse <code>java.io.File</code>	14-7
14.3	Textdateien	14-9
14.3.1	Ungepufferte Reader- und Writer Klassen.....	14-10
14.3.2	Gepufferte Reader- und Writer Klassen.....	14-11
14.3.3	Beispiel „TextDatei“	14-14
14.4	Daten- und Objektdateien.....	14-19
14.4.1	Klassen <code>InputStream</code> und <code>OutputStream</code>	14-19
14.4.2	Dateien für Elementardatentypen	14-22
14.4.3	Dateien für Elementar- und Referenzdatentypen	14-25
14.4.4	Beispiel „Obst“	14-28
14.5	Übersicht über häufig verwendete Datenströme	14-33
14.6	Weitere wichtige Klassen.....	14-35

14 Dateiverwaltung, das Stream-Konzept

Ein- und Ausgaben werden über sogenannte **Datenströme** realisiert. Durch die *Komplexität* solcher Routinen stoßen insbesondere Anfänger auf große Schwierigkeiten. Deshalb haben wir für Java in einem Paket **Tools** in der Klasse **IO** Klassenmethoden zur Eingabe von Daten über Tastatur zur Verfügung gestellt. Jetzt ist es an der Zeit, diesen Mechanismus genauer zu erklären und allgemein auf das Konzept der Datenein- und -ausgabe einzugehen.

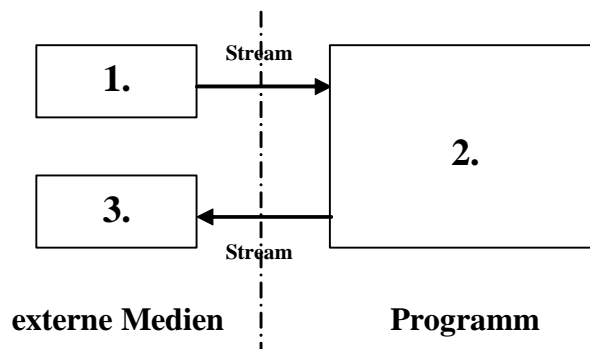
14.1 Datenströme

Dem euklidischen Algorithmus werden die beiden Startwerte m und n übergeben, nach seiner Ausführung liefert er das Ergebnis r zurück:



Jedes informationsverarbeitende System arbeitet, wie schon erwähnt, nach dem **EVA-Prinzip**:

1. **E**ingabe von *Informationen*
2. **V**erarbeitung von *Informationen* entsprechend vorgegebener Arbeitsanweisungen
3. **A**usgabe von *Informationen*

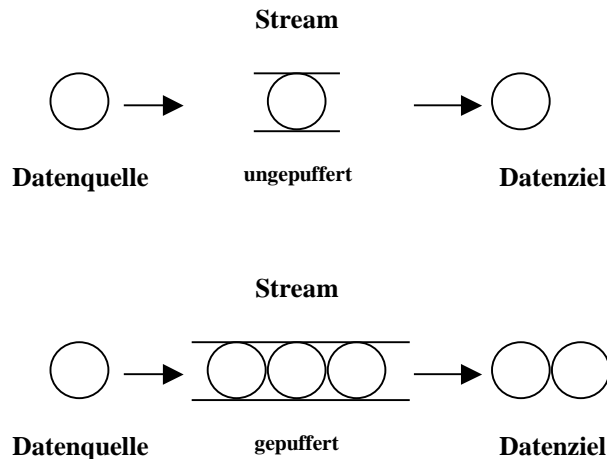


In den bisherigen Beispielen wurden meist benötigte **Daten** (*Informationen*) über eine *Tastatur* eingelesen und durch ein Programm manipulierte Daten in einem *Fenster*, der *Konsole*, ausgegeben. Nach dem Beenden des Programms gehen die Daten verloren.

Zu den *Standardaufgaben* eines Programms gehört es, auch auf *extern* vorhandene **Eingabedaten** zuzugreifen und **Ausgabedaten** *extern* abzuspeichern, d.h. der Zugriff auf *externe Medien* (Tastatur, Maus; Monitor, Drucker; Festplatte, ...) muss realisiert werden.

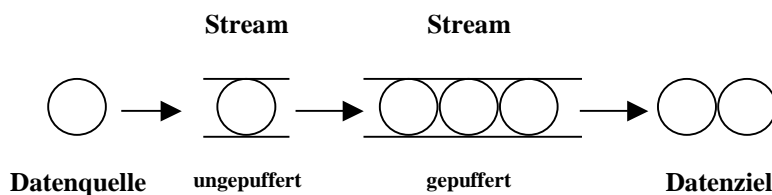
Die Kommunikation mit den externen Medien erfordert *Ein- und Ausgabemechanismen*. Die *Schnittstellen* zwischen einem Programm und den Ein- oder Ausgabemedien bilden sogenannte **Datenströme (Streams)**.

Ein Datenstrom nimmt Daten in Empfang und leitet sie weiter. Das *Weiterleiten* kann immer *nur in eine Richtung* erfolgen. Man unterscheidet zwischen **Eingabedatenströme** zum Lesen von Daten und **Ausgabedatenströme** zum Schreiben von Daten. Beide arbeiten nach dem **FIFO-Prinzip (First In First Out)**.



Datenströme transportieren **Dateneinheiten**. Sie können eine Dateneinheit (**ungepuffert**) oder mehrere Dateneinheiten (**gepuffert**) aufnehmen und weiter befördern. Durch gepufferte Zugriffe werden größere Sequenzen von Dateneinheiten zusammengefasst, was die *Performance bei Lese- und Schreibzugriffen* deutlich verbessert. Nachteilig kann sich die *Pufferung* auf die *Datensicherheit* auswirken.

Mehrere Datenströme können *hintereinandergeschaltet* werden. In diesem Zusammenhang ist der Begriff **Pipe, Pipeline** bzw. **Filter-Stream** geläufig.



14.1.1 Datenströme in Java, Paket `java.io.*`

Aus der Sicht der *Programme* wird unterschieden:

- **Eingabedatenströme** zum Einlesen von Daten
- **Ausgabedatenströme** zum Ausgeben von Daten

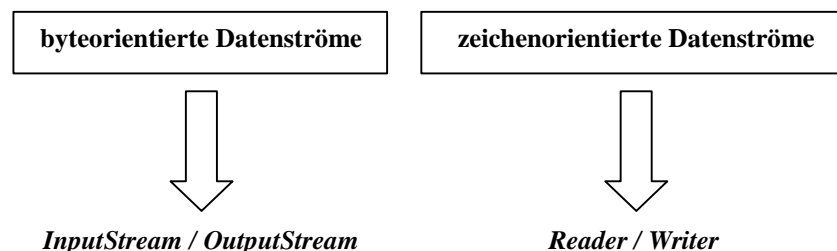
Aus der Sicht der *transportierten Dateneinheiten* wird unterschieden:

- **Byteströme** (8-Bit-Dateneinheiten, Datentype `byte`)
- **Zeichenströme** (16-Bit-Dateneinheiten, Datentype `char`)

Daraus ergeben sich vier Funktionalitäten, die im Paket `java.io.*` durch vier Hierarchien von Klassen abgedeckt werden:

- **InputStream** ist die *abstrakte* Oberklasse für alle *byteorientierten* Klassen zum Einlesen von Daten.
- **OutputStream** ist die *abstrakte* Oberklasse für alle *byteorientierten* Klassen zum Schreiben von Daten.
- **Reader** ist die *abstrakte* Oberklasse für alle *zeichenorientierten* Klassen zum Einlesen von Daten.
- **Writer** ist die *abstrakte* Oberklasse für alle *zeichenorientierten* Klassen zum Schreiben von Daten.

Byteorientierte Klassen transportieren Daten in Form von 8-bit-Einheiten und arbeiten mit dem Datentyp **byte**. *Zeichenorientierte* Klassen befördern Daten in Form von 16-bit-Einheiten, also mit dem Datentyp **char** bzw. Unicode-Zeichen.



Alle von diesen *abstrakten* Basisklassen abgeleiteten Klassen haben deren Namen als Endung.

14.1.2 Standarddatenströme

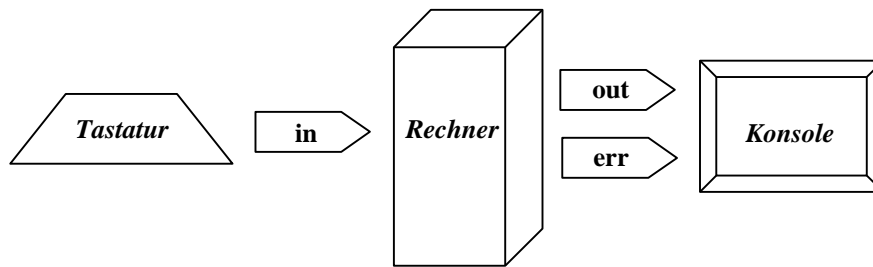
Die Standarddatenströme `in`, `out` und `err` sind *Klassenkonstanten* der Klasse `java.lang.System` und Objekte der Klassen des Pakets `java.io.*`.

- **System.in** ist der *Standardeingabedatenstrom*, ein Objekt der *byteorientierten* Klasse **InputStream**, und üblicherweise mit der *Tastatur* verbunden.

```
public static final InputStream in
```
- **System.out** ist der *Standardausgabedatenstrom*, ein Objekt der *byteorientierten* Klasse **PrintStream**, welche von **OutputStream** abgeleitet ist und üblicherweise mit der *Konsole* verbunden.

```
public static final PrintStream out
```
- **System.err** ist der *Standardfehlerausgabedatenstrom*, ein Objekt der *byteorientierten* Klasse **PrintStream**, welche von **OutputStream** abgeleitet ist und üblicherweise auch mit der *Konsole* verbunden.

```
public static final PrintStream err
```



Standardausgabe

Über den Objektverweis **System.out** wurden in früheren Beispielen bereits die Methoden **print** und **println** aufgerufen: **System.out.println("Hallo Welt!"**). Dies sind Methoden der Klasse **PrintStream**. Da auch **err** Objekt dieser Klasse ist, lassen sich diese Methoden auch zur Standardfehlerausgabe verwenden: **System.err.println("Fehler!"**).

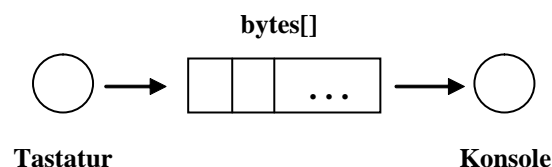
Die Klasse **PrintStream** erbt außerdem eine sehr einfache Ausgabemethode **write** der Klasse **OutputStream**.

Standardeingabe

Die Klasse **InputStream** stellt eine Methode **read** zur Verfügung, welche ein oder mehrere Bytes einliest.

Beispiel zur Verwendung der Methoden **read** und **write**

Von der Tastatur wird ein *Tastencode* über den Standardeingabedatenstrom **System.in** in das Bytefeld **bytes** *unkonvertiert* eingelesen. Ist das Feld gefüllt oder der Datenstrom beendet (**Enter**), so werden die eingelesenen Bytes aus dem Feld über den Standardausgabedatenstrom **System.out** an die Konsole geleitet.



Standard.java

```

// Standard.java
import java.io.*;

/**
 * Programm liest maximal 10 Bytes von der Tastatur
 * und gibt diese auf der Konsole aus
 */
public class Standard
{
    public static void main( String[] args)
    {

```

MM 2004

// read, write

```
byte bytes[] = new byte[10];

try
{
    System.out.println
    ( "Maximal 10 Zeichen eingeben:");          // Eingabe
    int bytesAnzahl = System.in.read( bytes);

    System.out.print
    ( "Es wurden " + bytesAnzahl + "B eingelesen: ");
    System.out.write( bytes);                  // Ausgabe
    System.out.println();
}
catch( Exception e)    // Fehlerbehandlung allgemein
{
    System.err.println( "Fehler: " + e);
}

System.out.println( "Fertig!");
}
```

14.2 Klasse `java.io.File`

Sehr häufig werden Verbindungen mit Dateien benötigt. In Java repräsentiert die Klasse **File** Dateien und Verzeichnisse. **In deren Objekten werden die Dateieigenschaften verwaltet, nicht ihre Inhalte.**

Konstruktor

- **File(String),**
ordnet einer Datei mit dem angegebenen Namen ein Objekt der Klasse **File** zu.

Das neu erzeugte Objekt stellt Methoden zur Verfügung, mit denen

- Informationen über Dateien oder Verzeichnisse abgefragt, wie deren Existenz, Zugriffsrechte, Pfad, Termin der letzten Änderung, ...
- Dateien umbenannt oder gelöscht,
- Verzeichnisse angelegt, umbenannt oder gelöscht werden können.

Methoden (Auswahl) der Klasse `java.io.File`

Name	Parameteranzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>canRead</code>	0		boolean	true , falls Leserecht
<code>canWrite</code>	0		boolean	true , falls Schreibrecht
<code>createNewFile</code>	0		boolean	legt neue Datei an
<code>delete</code>	0		boolean	löscht Datei
<code>exists</code>	0		boolean	true , falls Existenz
<code>getAbsolutePath</code>	0		String	liefert absoluten Pfad
<code>getName</code>	0		String	liefert Dateinamen
<code>isDirectory</code>	0		boolean	true , falls Verzeichnis
<code>isFile</code>	0		boolean	true , falls Datei
<code>length</code>	0		long	liefert Größe der Datei
<code>list</code>	0		String[]	liefert Verzeichnishierarchie, falls Objekt ein Verzeichnis ist, sonst null
<code>mkdir</code>	0		boolean	legt Verzeichnis an
<code>renameTo</code>	1	File	boolean	benennt Objekt um

FileInfo.java

```
// FileInfo.java                                MM 2004
import java.io.*;                                // File

/**
 * Programm ermittelt Datei/Verzeichnis-Eigenschaften
 * Aufruf: java FileInfo <Verzeichnis>|<Datei>
 */
public class FileInfo
{
    public static void main( String[] args)
    {
```

```
try
{
    // File-Objekt erzeugen
    File datei = new File( args[ 0] );
    // Datei -Eigenschaften abfragen
    if( datei.exists() )
    {
        if( datei.isFile() )
        {
            System.out.println
            ( "Datei " + args[ 0] + ": Datei");

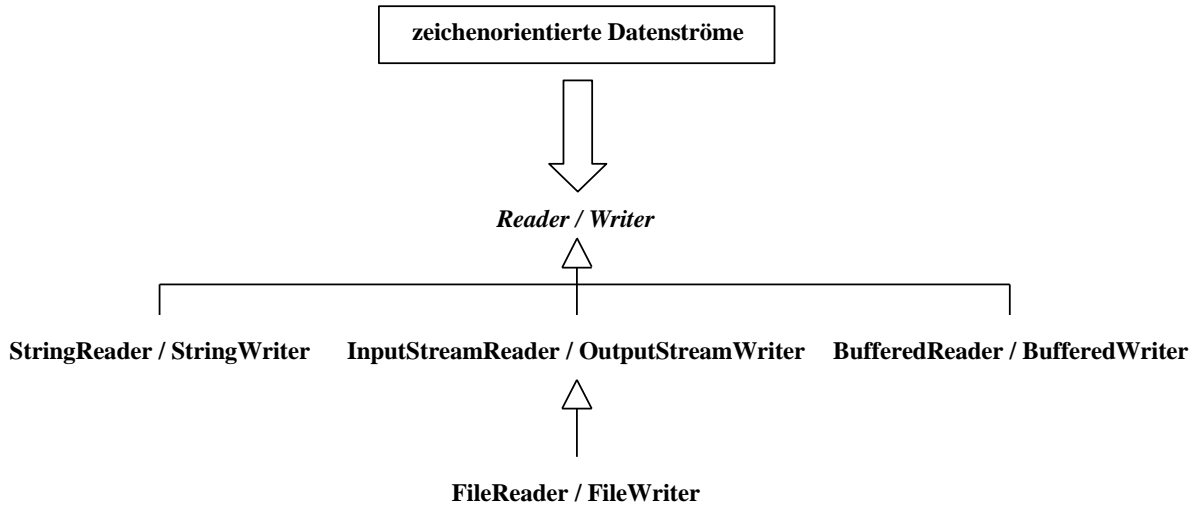
            System.out.println
            ( "kompletter Pfad: " + datei.getAbsolutePath() );
            System.out.println
            ( "Datei:          " + datei.getName() );
            System.out.println
            ( "Leserecht:      " + datei.canRead() );
            System.out.println
            ( "Schreibrecht:   " + datei.canWrite() );
        }
        // Verzeichnis-Eigenschaften abfragen
        if( datei.isDirectory() )
        {
            System.out.println
            ( "Datei " + args[ 0] + ": Verzeichnis");

            String[] dateien = datei.list();
            for( int i = 0; i < dateien.length; i++)
                System.out.println( dateien[ i] );
        }
    }
    else
        // Datei/Verzeichnis existiert nicht
        System.out.println
        ( "Datei " + args[ 0] + " nicht gefunden.");
}
// Befehl nicht vollstaendig eingegeben
catch( ArrayIndexOutOfBoundsException e )
{
    System.err.println
    ( "Aufruf: java FileInfo <Verzeichnis>|<Datei>");
}
catch( Exception e ) // Fehlerbehandlung allgemein
{
    System.err.println( "Fehler: " + e );
}
System.out.println( "Fertig!");
}
```

14.3 Textdateien

Die *abstrakten* Klassen `java.io.Reader` und `java.io.Writer` legen die Methoden für die Eingabe und die Ausgabe in *Textdateien* fest.

Überblick über zeichenorientierte Datenströme



Methoden (Auswahl) der Klasse `java.io.Reader`

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>read</code>	0		<code>int</code>	liefert das nächste Zeichen
<code>read</code>	1	<code>char[]</code>	<code>int</code>	füllt das Feld mit den nächsten Zeichen auf und liefert die Anzahl zurück
<code>read</code>	3	<code>char[], int, int</code>	<code>int</code>	füllt das Feld vom angegebenen Index mit der angegebenen Anzahl von Zeichen auf, liefert die tatsächliche gelesene Anzahl zurück
<code>close</code>	0		<code>void</code>	schließt den Datenstrom

Methoden der Klasse `java.io.Writer`

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>write</code>	1	<code>int</code>	<code>void</code>	schreibt das angegebene Zeichen
<code>write</code>	1	<code>char[]</code>	<code>void</code>	schreibt die angegebenen Zeichen
<code>write</code>	3	<code>char[], int, int</code>	<code>void</code>	schreibt vom angegebenen Index an die angegebene Anzahl von Zeichen
<code>write</code>	1	<code>String</code>	<code>void</code>	schreibt den angegebene String
<code>write</code>	3	<code>String, int, int</code>	<code>void</code>	schreibt vom angegebenen Index an die angegebene Anzahl von Zeichen des String
<code>close</code>	0		<code>void</code>	schließt den Datenstrom
<code>flush</code>	0		<code>void</code>	leert den Datenstrom, indem noch enthaltene Zeichen gelöscht werden

14.3.1 Ungepufferte Reader- und Writer Klassen

Von den Klassen **Reader** und **Writer** sind die Klassen **InputStreamReader** und **OutputStreamWriter**, von diesen wiederum sind die Klassen **FileReader** und **FileWriter** abgeleitet.

Konstruktoren

- **InputStreamReader(InputStream)**,
erzeugt aus einem Byte-Eingabestrom einen Zeichen-Eingabestrom.
- **OutputStreamWriter(OutputStream)**,
erzeugt aus einem Zeichen-Ausgabestrom einen Byte-Ausgabestrom.
- **FileReader(File)**,
erzeugt zu einer Datei einen Zeichen-Eingabestrom.
- **FileWriter(File)**,
erzeugt zu einer Datei einen Zeichen-Ausgabestrom, der Inhalt wird überschrieben.
- **FileWriter(File, boolean)**,
erzeugt zu einer Datei einen Zeichen-Ausgabestrom,
false der Inhalt wird überschrieben,
true alle Zeichen werden an den schon bestehenden Inhalt angehängt.
- **StringReader(String)**,
erzeugt aus einem String einen Zeichen-Eingabestrom.
- **StringWriter()**,
erzeugt zu einem dynamisch wachsenden Stringpuffer einen Zeichen-Ausgabestrom.

TextKopie.java

```
// TextKopie.java                                     MM 2004
import java.io.*;                                     // Reader, Writer

/**
 * Ungepufferte Kopiermethode fuer Texte.
 * Aufruf: java TextKopie <Quelldatei> <Zieldatei>
 */
public class TextKopie
{
/**
 * Kopiert zeichenweise, ungepuffert.
 */
    public void kopieren( Reader in, Writer out)
        throws Exception
    {
        int zeichen;

        // Lesen und Schreiben eines Zeichens,
        // -1 Datenstromende (^D, ^Z)
        while(( zeichen = in.read()) != -1)
            out.write( zeichen);
    }

/**
 * Testprogramm.
 */
}
```

```
*/
public static void main( String[] args)
{
    try
    {
        File file;
        TextKopie text = new TextKopie();
            // Eingabedatenstrom erzeugen
        file = new File( args[ 0]);
        Reader in = new FileReader( file);
            // Ausgabedatenstrom erzeugen
        file = new File( args[ 1]);
        Writer out = new FileWriter( file);

        text.kopieren( in, out);           // Kopieren

        in.close();           // Schliessen der Dateien
        out.close();
    }
    // Befehl nicht vollstaendig eingegeben
    catch( ArrayIndexOutOfBoundsException e)
    {
        System.err.print( "Aufruf: ");
        System.err.println
        ( "java TextKopie <Quelldatei> <Zieldatei>");
    }
    // Textdatei existiert nicht
    catch( FileNotFoundException e)
    {
        System.err.println
        ( "Die Datei >" + args[ 0] + "< existiert nicht!");
    }
    // Fehlerbehandlung allgemein
    catch( Exception e)
    {
        System.err.println( "Fehler: " + e);
    }
    System.out.println( "Fertig!");
}
}
```

In diesem Beispiel wird eine Datei *zeichenweise* kopiert: Ein Zeichen der Originaldatei wird gelesen und anschließend in die Kopiedatei gespeichert, erst dann wird das nächste Zeichen behandelt.

14.3.2 Gepufferte Reader- und Writer Klassen

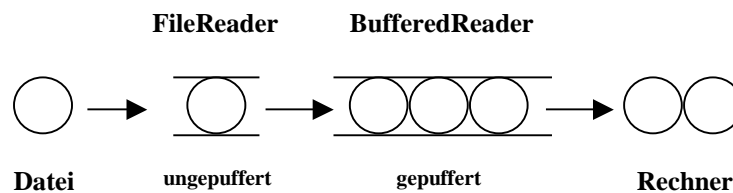
Sind *sehr viele Zeichen* von einer Datenquelle zu lesen bzw. in ein Datenziel zu schreiben, so ist ein externer Zugriff auf jedes Zeichen einzeln und dessen anschließenden Verarbeitung *uneffektiv*. Man setzt **Datenpuffer** ein. Diese befinden sich im Hauptspeicher und werden in größeren Abständen weiterverarbeitet.

In Java ermöglichen die Klassen **BufferedReader** und **BufferedWriter** die Pufferung mehrerer Dateneinheiten. Diese Klassen müssen mit *ungepufferten Datenströmen* verkettet werden:

Konstruktoren

- **BufferedReader(Reader)**,
erzeugt einen gepufferten Zeichen-Eingabestrom.
- **BufferedWriter(Writer)**,
erzeugt einen gepufferten Zeichen-Ausgabestrom.

Zum Beispiel werden beim *gepufferten* Einlesen aus einer Datei zunächst die Daten *zeichenweise* von der Datei eingelesen und in einem *Puffer* abgelegt. Anschließend erfolgt die Weiterverarbeitung.



```
Reader in = new FileReader( file);
BufferedReader inBuffer = new BufferedReader( in);
```

oder

```
BufferedReader inBuffer =
    new BufferedReader( new FileReader( file));
```

Methode der Klasse `java.io.BufferedReader`

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>readLine</code>	0		String	liefert ganze Textzeile aus dem Buffer, Zeilentrennzeichen: '\n' '\r'

Methode der Klasse `java.io.BufferedWriter`

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>newLine</code>	0		void	schreibt Zeilenwechsel in den Buffer

TextKopieMitPuffer.java

```
// TextKopieMitPuffer.java
import java.io.*;
// Reader, Writer
MM 2004

/**
 * Gepufferte Kopiermethode fuer Texte.
```

```
* Aufruf: java TextKopieMitPuffer <Quelldatei> <Zieldatei>
*/
public class TextKopieMitPuffer
{
/**
* Kopiert zeilenweise, gepuffert.
*/
public void kopieren
( BufferedReader inBuffer, BufferedWriter outBuffer)
throws Exception
{
    String zeile;
                                // zeilenweises Lesen und Schreiben,
    try                                // null Datenstromende
    {
        while( true)
        {
            zeile = inBuffer.readLine();
            outBuffer.write( zeile);
            outBuffer.newLine();           // Zeilenwechsel
        }
    }
                                // Dateiende erreicht
    catch( NullPointerException e){}
}

/**
* Testprogramm.
*/
public static void main( String[] args)
{
    try
    {
        File file;
        TextKopieMitPuffer text =
            new TextKopieMitPuffer();
            // Eingabedatenstrom erzeugen, gepuffert
        file = new File( args[ 0]);
        Reader in = new FileReader( file);
        BufferedReader inBuffer =
            new BufferedReader( in);
            // Ausgabedatenstrom erzeugen, gepuffert
        file = new File( args[ 1]);
        Writer out = new FileWriter( file);
        BufferedWriter outBuffer =
            new BufferedWriter( out);

        text.kopieren( inBuffer, outBuffer); // Kopieren

        inBuffer.close();           // Schliessen der Dateien
        outBuffer.close();
    }
}
```

```

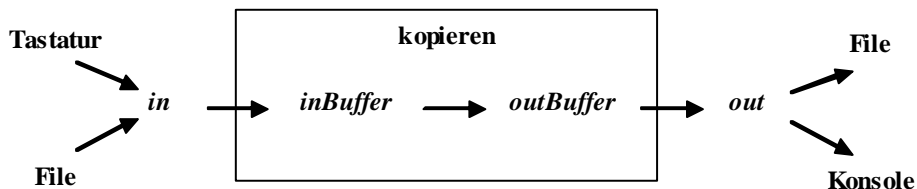
        // Befehl nicht vollstaendig eingegeben
    catch( ArrayIndexOutOfBoundsException e)
    {
        System.err.print( "Aufruf: java ");
        System.err.println
        ( "TextKopieMitPuffer <Quelldatei> <Zieldatei>");
    }

        // Textdatei existiert nicht
    catch( FileNotFoundException e)
    {
        System.err.println
        ( "Die Datei >" + args[ 0] + "< existiert nicht!");
    }
    catch( Exception e) // Fehlerbehandlung allgemein
    {
        System.err.println( "Fehler: " + e);
    }
    System.out.println( "Fertig!");
}
}

```

14.3.3 Beispiel „TextDatei“

Das folgende Beispiel liest Text von der Tastatur und schreibt diesen in eine Datei, Anschließend wird eine Kopie der Datei angelegt und auf der Konsole ausgegeben. Alle drei Aktionen verlaufen über die bereits oben angegebene Kopiermethode **kopieren**, nur die *Ein- und Ausgabe-Datenströme* variieren.



Zu beachten ist, dass man *Standarddatenströme* nach Beenden des Kopierens *nicht* schließen sollte, damit sie auch für die weitere Arbeit zur Verfügung stehen. Der *Standard-Ausgabestrom* muss aber nach dem Kopieren *geleert* werden, da sonst evtl. noch Zeichen im Puffer verbleiben und die Schreibaktion nicht bis zum Ende ausgeführt wird.

TextDatei	
◆	kopiereTextDatei(datei0 : File, datei1 : File) : void
🗑️	kopieren(inBuffer : BufferedReader, outBuffer : BufferedWriter) : void
◆	leseAusTextDatei(datei : File) : void
◆	main(args : String[]) : void
◆	schreibelnTextDatei(datei : File) : void

TextDatei.java

```
// TextDatei.java
import java.io.*;

/**
 * Gepuffertes Kopieren von Textdateien.
 * Aufruf: java TextDatei <Quelldatei> <Zieldatei>
 * Schreiben in Datei:      Tastatur --> Datei
 * Kopieren von/nach Datei: Datei --> Datei
 * Lesen aus Datei:        Datei --> Konsole
 */
public class TextDatei
{
    /**
     * Liest Text von der Tastatur
     * und schreibt ihn in eine Datei.
     * @param datei Zieldatei
     */
    public void schreibeInTextDatei( File datei)
        throws Exception
    {
        Reader in;
        Writer out;
        BufferedReader inBuffer;
        BufferedWriter outBuffer;

        // Tastatureingabedatenstrom, gepuffert
        in = new InputStreamReader( System.in);
        inBuffer = new BufferedReader( in);

        // Ausgabedatenstrom auf datei, gepuffert
        out = new FileWriter( datei);
        outBuffer = new BufferedWriter( out);

        // Schreiben
        kopieren( inBuffer, outBuffer);

        // Schliessen der Textdatei
        outBuffer.close();
    }

    /**
     * Kopiert eine Textdatei zeilenweise in eine Textdatei.
     * @param datei0 Quelldatei
     * @param datei1 Zieldatei
     */
    public void kopiereTextDatei
        ( File datei0, File datei1)
        throws Exception
    {
        Reader in;
```

```
    Writer out;
    BufferedReader inBuffer;
    BufferedWriter outBuffer;

// Eingabedatenstrom von datei0, gepuffert
    in = new FileReader( datei0);
    inBuffer = new BufferedReader( in);

// Ausgabedatenstrom auf datei1, gepuffert
    out = new FileWriter( datei1);
    outBuffer = new BufferedWriter( out);

// Kopieren
    kopieren( inBuffer, outBuffer);

// Schliessen der Textdateien
    inBuffer.close();
    outBuffer.close();
}

/**
 * Gibt Datei zeilenweise auf der Konsole aus.
 * @param datei Quelldatei
 */
public void leseAusTextDatei ( File datei)
    throws Exception
{
    Reader in;
    Writer out;
    BufferedReader inBuffer;
    BufferedWriter outBuffer;

// Eingabedatenstrom von datei, gepuffert
    in = new FileReader( datei);
    inBuffer = new BufferedReader( in);

// Ausgabedatenstrom auf Konsole, gepuffert
    out = new OutputStreamWriter( System.out);
    outBuffer = new BufferedWriter( out);

// Ausgeben der Kopie
    kopieren( inBuffer, outBuffer);

// Schliessen der Textdatei, Leeren des Ausgabepuffers
    inBuffer.close();
    outBuffer.flush();           // Leeren
}

/**
 * Kopiert zeilenweise, gepuffert,
```

```
* schliesst nach Beenden die Dateien.
* @param inBuffer Quelle
* @param outBuffer Ziel
*/
private void kopieren
( BufferedReader inBuffer, BufferedWriter outBuffer)
throws Exception
{
    String zeile;

// zeilenweises Lesen und Schreiben
    try
    {
        while( true)
        {
            zeile = inBuffer.readLine();
            outBuffer.write( zeile);
            outBuffer.newLine();           // Zeilenwechsel
        }
    }

// Dateiende erreicht
    catch( NullPointerException e){}
}

/**
 * Testprogramm
 */
public static void main( String[] args)
{
    try
    {
        File datei0 = new File( args[ 0]);
        File datei1 = new File( args[ 1]);
        TextDatei text = new TextDatei();

// Schreiben in Datei:  Tastatur    -->  Datei datei0
        System.out.println( "Schreiben des Textes");
        System.out.print( "Text eingeben, ");
        System.out.println
        ( "Abbruch: Strg+Z/Ctrl+Z bzw. Strg+D/Ctrl+D:");
        text.schreibeInTextDatei( datei0);
        System.out.println( "Schreiben abgeschlossen!\n");

// Kopie von/nach datei: Datei datei0 --> Datei datei1
        System.out.println( "Kopieren der Datei");
        text.kopiereTextDatei( datei0, datei1);
        System.out.println( "Kopieren abgeschlossen!\n");

// Lesen aus Datei:      Datei datei1 --> Konsole
        System.out.println( "Lesen der Kopie");
    }
}
```

```
        text.leseAusTextDatei( datei1);
        System.out.println( "Lesen abgeschlossen!\n");
    }

    // Befehl nicht vollstaendig eingegeben
    catch( ArrayIndexOutOfBoundsException e)
    {
        System.err.println
        ( "Aufruf: java TextDatei <Quelldatei> <Zieldatei>");
    }

    // Fehlerbehandlung allgemein
    catch( Exception e)
    {
        System.err.println( "Fehler: " + e);
    }

    System.out.println( "Fertig!");
}
}
```

14.4 Daten- und Objektdateien

Texte sind nur eine Form von Daten. Eine *Textdatei* ist normalerweise *durch Zeilen strukturiert*. Das Ende der zumeist unterschiedlich langen Zeilen ist durch ein besonderes Zeichen, dem *Zeilenendezeichen* '`\n`', gekennzeichnet. Eine *gepufferte Ein- und Ausgabe* beschränkt sich auf die *zeilenweise Ein- und Ausgabe*. Dadurch sind Textdateien für die Speicherung von nicht - **char** - Datentypen ungeeignet, vorausgesetzt, man möchte diese Werte nicht immer in **char** - Datentypen konvertieren und bei Gebrauch zurückkonvertieren. Deshalb stellt Java auch Klassen zur Nutzung von Dateien für alle anderen Datentypen zur Verfügung.

14.4.1 Klassen `InputStream` und `OutputStream`

Die *abstrakten* Klassen `InputStream` und `OutputStream` legen Methoden für *byteorientierte* Eingaben aus Dateien und Ausgaben in Dateien für *Elementar- und Referenzdatentypen* fest.

Methoden (Auswahl) der Klasse `java.io.InputStream`

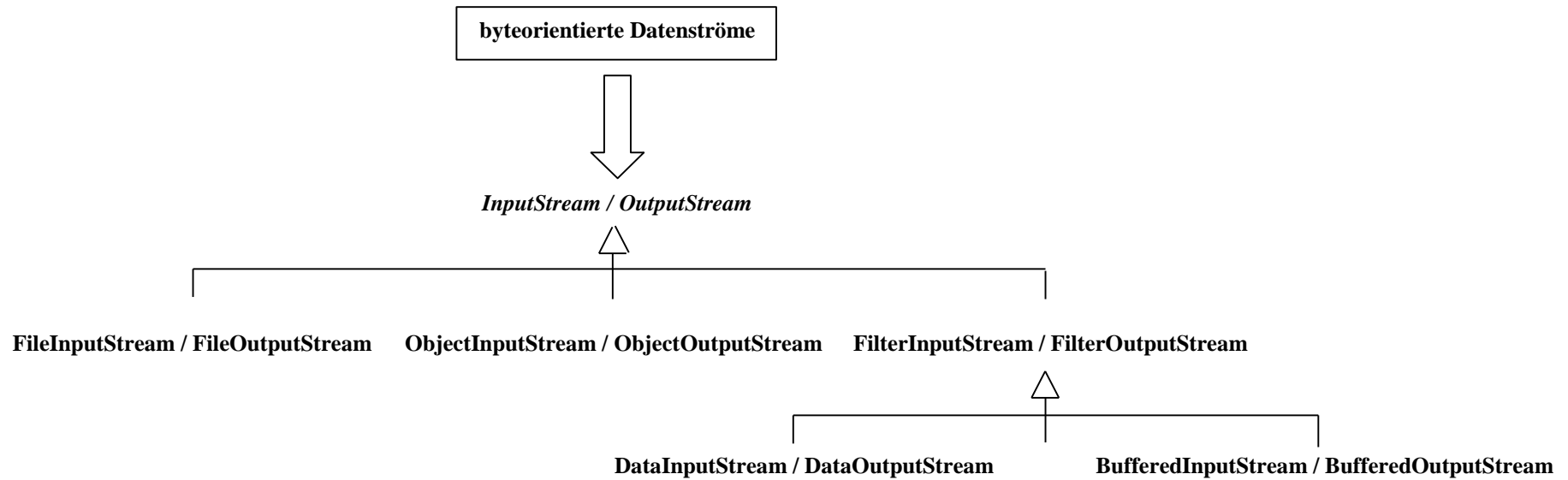
Name	Parameteranzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>read</code>	0		<code>int</code>	liefert das nächste Byte als <code>int</code> -Wert
<code>read</code>	1	<code>byte[]</code>	<code>int</code>	füllt das Feld mit den nächsten Bytes auf und liefert die Anzahl zurück
<code>read</code>	3	<code>byte[], int, int</code>	<code>int</code>	füllt das Feld vom angegebenen Index mit der angegebenen Anzahl von Bytes auf und liefert die tatsächliche gelesene Anzahl zurück
<code>close</code>	0		<code>void</code>	schließt den Datenstrom

Methoden der Klasse `java.io.OutputStream`

Name	Parameteranzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>write</code>	1	<code>int</code>	<code>void</code>	schreibt das angegebene Byte
<code>write</code>	1	<code>byte[]</code>	<code>void</code>	schreibt die angegebenen Bytes
<code>write</code>	3	<code>byte[], int, int</code>	<code>void</code>	schreibt vom angegebenen Index an die angegebene Anzahl von Bytes
<code>close</code>	0		<code>void</code>	schließt den Datenstrom
<code>flush</code>	0		<code>void</code>	leert den Buffer, indem alle noch enthaltenen Bytes abgearbeitet werden

Klassen für *byteorientierte* Datenströme sind analog den Klassen der *zeichenorientierten* Ströme aufgebaut.

Überblick über zeichenorientierte Datenströme



Die Klassen **FileInputStream** und **FileOutputStream** stellen Methoden zur Verfügung, welche einzelne Bytes in Dateien schreiben bzw. daraus lesen. Sie werden von den Klassen **InputStream** und **OutputStream** abgeleitet.

Konstruktoren

- **FileInputStream(File)**,
erzeugt zu einer Datei einen Byte-Eingabestrom.
- **FileOutputStream(File)**,
erzeugt zu einer Datei einen Byte-Ausgabestrom, der Inhalt wird überschrieben.
- **FileOutputStream(File, boolean)**,
erzeugt zu einer Datei einen Byte-Ausgabestrom,
false der Inhalt wird überschrieben,
true alle Zeichen werden an den schon bestehenden Inhalt angehängt.

Die Klassen **DataInputStream** und **DataOutputStream** definieren Methoden zum Lesen bzw. Schreiben von *Elementardatentypen*. Mit dem entsprechenden **FileInputStream** bzw. **FileOutputStream** verbunden, lassen sich diese Daten in bzw. aus Dateien transportieren. Die Klassen **DataInputStream**, **DataOutputStream** sind von den Klassen **FilterInputStream** und **FilterOutputStream** abgeleitet.

Konstruktoren

- **DataInputStream(InputStream)**,
erzeugt aus einem Byte-Eingabestrom einen Eingabestrom für *Elementardatentypen*.
- **DataOutputStream(OutputStream)**,
erzeugt aus einem Ausgabestrom für *Elementardatentypen* einen Byte-Ausgabestrom.

Die Klassen **ObjectInputStream** und **ObjectOutputStream** definieren Methoden zum Lesen bzw. Schreiben von *Elementar-* bzw. *Referenzdatentypen*. Mit dem entsprechenden **FileInputStream** bzw. **FileOutputStream** verbunden, lassen sich diese Daten in bzw. aus Dateien transportieren. Die Klassen **ObjektInputStream**, **ObjektOutputStream** sind direkt von den Klassen **InputStream** und **OutputStream** abgeleitet.

Konstruktoren

- **ObjectInputStream(InputStream)**,
erzeugt aus einem Byte-Eingabestrom einen Eingabestrom für *Elementar-* bzw. *Referenzdatentypen*.
- **ObjectOutputStream(OutputStream)**,
erzeugt aus einem Ausgabestrom für *Elementar-* bzw. *Referenzdatentypen* einen Byte-Ausgabestrom.

Auch für byteorientierte Datenströme gibt es mit den Klassen **BufferedInputStream** und **BufferedOutputStream** die Möglichkeit, mehrere Dateneinheiten zu *puffern*.

Konstruktoren

- **BufferedInputStream(InputStream)**,
erzeugt einen gepufferten Byte-Eingabestrom.

- **BufferedOutputStream(OutputStream)**, erzeugt einen gepufferten Byte-Ausgabestrom.

14.4.2 Dateien für Elementardatentypen

In *byteorientierten* Dateien liegen *elementare* Daten in einer direkt vom Programm lesbaren Form vor, d.h. so wie sie auch im Speicher des Rechners angeordnet sind. Sie können deshalb direkt (ohne Konvertierung) verarbeitet werden.

Methoden (Auswahl) der Klasse `java.io.DataInputStream`

Name	Parameteranzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>readTtt</code>	0		<code>ttt</code>	liefert einen Wert des Datentyps <code>ttt</code>

Methoden (Auswahl) der Klasse `java.io.DataOutputStream`

Name	Parameteranzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>writeTtt</code>	1	<code>ttt</code>	<code>void</code>	schreibt den Wert des Datentyps <code>ttt</code>

`ttt` stehen für die Datentypen `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short` und `Ttt` entsprechend mit Großbuchstaben beginnend.

Das folgende Beispiel schreibt `double`-Werte in eine Datei und liest diese wieder heraus.

DoubleDatei
<ul style="list-style-type: none"> ◆ <code>leseAusDoubleDatei(datei : File) : void</code> ◆ <code>main(args : String[]) : void</code> ◆ <code>schreibeInDoubleDatei(datei : File) : void</code>

DoubleDatei.java

```
// DoubleDatei.java
```

MM 2004

```
import java.io.*;
```

```
/**
```

```
 * Programm schreibt und liest Double-Werte
```

```
 * in bzw. aus einer Datei.
```

```
 */
```

```
public class DoubleDatei
```

```
{
```

```
/**
```

```
 * Liest Double-Werte von der Tastatur
```

```
* und schreibt sie in eine Datei.
*/
public void schreibeInDoubleDatei( File datei)
throws Exception
{
    // Tastatureingabedatenstrom erzeugen, gepuffert
    Reader in = new InputStreamReader( System.in);
    BufferedReader inBuffer = new BufferedReader( in);

    // Ausgabedatenstrom auf Datei datei erzeugen
    OutputStream out = new FileOutputStream( datei);
    DataOutputStream outDatei =
        new DataOutputStream( out);

    // Eingabe eines Double-Wertes von Tastatur
    String zeile = inBuffer.readLine();

    while( !zeile.equals( ""))
    {
        // Schreiben des Double-Wertes in die Datei
        try
        {
            double zahl = Double.parseDouble( zeile);
            outDatei.writeDouble( zahl);
        }

        // Fehler beim Zahlenformat
        catch( NumberFormatException e)
        {
            System.out.println
                ( "Nur Double-Werte eingeben!");
        }

        // Eingabe eines Double-Werts von Tastatur
        zeile = inBuffer.readLine();
    }

    outDatei.close(); // Schliessen der Datendatei
}

/**
 * Liest Double-Werte aus einer Datei
 * und schreibt sie auf der Konsole aus.
 */
public void leseAusDoubleDatei( File datei)
throws Exception
{
    // Eingabedatenstrom auf Datei datei erzeugen
    InputStream in = new FileInputStream( datei);
    DataInputStream inDatei = new DataInputStream( in);
```

```
try
{
    while( true)
    {
        // Lesen eines Double-Wertes aus der Datei
        double zahl = inDatei.readDouble();

        // Schreiben eines Double-Wertes auf der Konsole
        System.out.println( "" + zahl);
    }
}

// Fehler beim Zahlenformat
catch( NumberFormatException e)
{
    System.out.println
    ( "Auf der Datei befinden sich Nicht-Double-Werte!");
}

// Dateiende erreicht, Schliessen der Datei
catch ( EOFException e)
{
    inDatei.close();
}
}

/**
 * Testprogramm.
 */
public static void main( String[] args)
{
    try
    {
        File datei = new File( args[ 0]);
        DoubleDatei daten = new DoubleDatei();

        System.out.print( "Double-Werte schreiben ");
        System.out.println( "(Abbruch mit ENTER): ");
        daten.schreibeInDoubleDatei( datei);

        System.out.println( "Double-Werte lesen: ");
        daten leseAusDoubleDatei( datei);
    }

    // Befehl nicht vollstaendig eingegeben
    catch( ArrayIndexOutOfBoundsException e)
    {
        System.out.println
        ( "Aufruf: java DoubleDatei <Zieldatei>");
    }
}
```

```

    catch( Exception e)      // Fehlerbehandlung allgemein
    {
        System.out.println( "Fehler: " + e);
    }
    System.out.println( "Fertig!");
}
}

```

```

C:\Monika_neu\Java\JavaVorl\WS_05\Programme\Streams\Double>java DoubleDatei
Aufruf: java DoubleDatei <Zieldatei>
Fertig!

C:\Monika_neu\Java\JavaVorl\WS_05\Programme\Streams\Double>java DoubleDatei In.dat
Double-Werte schreiben (Abbruch mit ENTER):
3
1234.5
t
Nur Double-Werte eingeben?
.01
e-3
Nur Double-Werte eingeben?
1e-3

Double-Werte lesen:
3.0
1234.5
0.01
0.0010
Fertig!

C:\Monika_neu\Java\JavaVorl\WS_05\Programme\Streams\Double>

```

Die Werte in der Datendatei sind entsprechend ihrer internen Speicherdarstellung abgelegt:

In.dat

```
@       @ J      ?,,zG@ {?PbM     
```

14.4.3 Dateien für Elementar- und Referenzdatentypen

Komplexe Daten müssen für den Transport in eine Datei **serialisiert** bzw. beim Transport aus einer Datei **deserialisiert** werden. Beim **Serialisieren** werden die Objekte in einen *byteorientierten* Datenstrom zerlegt. Dieser kann dann in eine Datei gespeichert werden. Umgekehrt wird beim **Deserialisieren** das Objekt aus dem *byteorientierten* Datenstrom wieder zusammengesetzt.

Dazu stellen die Klassen **ObjectInputStream** und **ObjectOutputStream** die bereits bekannten Methoden der Klassen **DataInputStream** und **DataOutputStream** und neue spezielle Methoden zur Verfügung.

Methoden (Auswahl) der Klasse `java.io.ObjectInputStream`

Name	Parameteranzahl	Parameter-typ	Ergebnis-typ	Beschreibung
<code>readObject</code>	0		<code>Object</code>	deserialisiert ein Objekt
<code>readTtt</code>	0		<code>ttt</code>	liefert einen Wert des Datentyps <code>ttt</code>

Methoden (Auswahl) der Klasse `java.io.ObjectOutputStream`

Name	Parameteranzahl	Parameter-typ	Ergebnis-typ	Beschreibung
<code>writeObject</code>	1	<code>Object</code>	<code>void</code>	serialisiert ein Objekt
<code>writeTtt</code>	1	<code>ttt</code>	<code>void</code>	schreibt den Wert des Datentyps <code>ttt</code>

Objekte können nur *serialisiert* werden, wenn sie das Interface `java.io.Serializable` implementieren. **Dieses Interface hat weder Methoden noch Attribute.** Deshalb genügt es, den Kopf der Objektklasse mit `implements Serializable` zu ergänzen.

Beispiel

In einer generischen Klasse `ObjektDatei<E>` werden Methoden zum Lesen und Schreiben von Objektdateien zusammengefasst. Der Objekttyp `E` wird noch nicht festgelegt.

ObjektDatei<E>
◆ kopiereObjektDatei(datei0 : File, datei1 : File) : void
◆ leseAusObjektDatei(datei : File, col : Collection<E>) : void
◆ listeObjekte(col : Collection<E>) : void
◆ schreibeInObjektDatei(datei : File, obj : E) : void
◆ schreibeInObjektDatei(datei : File, col : Collection<E>) : void

ObjektDatei.java

```
// ObjektDatei.java
```

MM 2007

```
import java.io.*;
import java.util.*;

/**
 * Verwalten von Objektdateien.
 * Lesen aus Datei:           Datei           --> Collection,
 * Schreiben in Datei:       Collection       --> Datei,
 * Anhaengen an Datei:      Objekt          --> Datei,
 * Kopieren von/nach Datei:  Datei           --> Datei,
 * Ausgabe aus Collection:   Collection      --> Konsole
 */
public class ObjektDatei<E>
{
/**
```

```
* Lesen aus einer Objektdatei.
* @param datei Quelldatei
* @param col Ziel, Collection von Objekten
*/
public void leseAusObjektDatei
    ( File datei, Collection<E> col)
    throws Exception
{
    FileInputStream in =
        new FileInputStream( datei);
    ObjectInputStream inDatei =
        new ObjectInputStream( in);

    try
    {
        while( true) col.add( (E)inDatei.readObject());
    }
    catch( Exception e)
    {
        inDatei.close();
    }
}

/**
* Schreiben in eine Objektdatei.
* @param datei Zieldatei
* @param col Quelle, Collection von Objekten
*/
public void schreibeInObjektDatei
    ( File datei, Collection<E> col)
    throws Exception
{
    FileOutputStream out =
        new FileOutputStream( datei);
    ObjectOutputStream outDatei =
        new ObjectOutputStream( out);

    Iterator<E> it = col.iterator();
    while( it.hasNext())
        outDatei.writeObject( it.next());

    outDatei.flush();
    outDatei.close();
}

/**
* Schreiben in Objektdatei, anhaengend.
* @param datei Zieldatei
* @param obj Quelle, Objekt
*/
public void schreibeInObjektDatei
    ( File datei, E obj)
```

```

    throws Exception
    {
        Collection<E> col = new Vector<E>();
        if( datei.exists())
            leseAusObjektDatei( datei, col);
        col.add( obj);
        schreibeInObjektDatei( datei, col);
    }

/**
 * Kopiert eine Objektdatei objektweise
 * in eine Objektdatei.
 * @param datei0 Quelldatei
 * @param datei1 Zieldatei
 */
public void kopiereObjektDatei
    ( File datei0, File datei1)
    throws Exception
    {
        Vector<E> col = new Vector<E>();

        leseAusObjektDatei( datei0, col);
        schreibeInObjektDatei( datei1, col);
    }

/**
 * Ausgabe einer Objektliste.
 * @param col Collection von Objekten
 */
public void listeObjekte( Collection<E> col)
    {
        Iterator<E> it = col.iterator(); int i = 0;
        while( it.hasNext())
            System.out.println( "" + i++ + " " + it.next());
    }
}

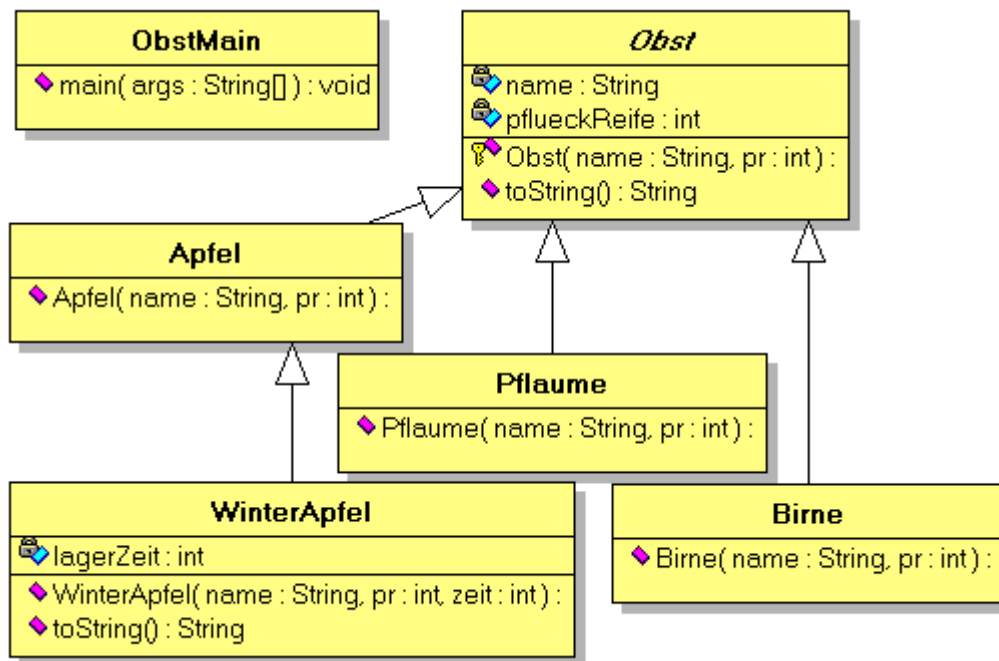
```

14.4.4 Beispiel „Obst“

Es folgt ein Anwendungsprogramm zur Verwaltung von Obstsorten unter Verwendung der Klasse **ObjektDatei<Obst>**:

Apfel, Birne, Pflaume und speziell Winterapfel sind jeweils als Klasse festgelegt. Eine übergeordnete *abstrakte* Klasse **Obst** fasst die Gemeinsamkeiten zusammen. Diese Klassenhierarchie ist so aufgebaut, dass sie zu jeder Zeit durch neue Obstsorten erweitert werden kann.

Dazu werden unter dem Oberbegriff Obst der Name **name** und die Pflückzeit **pflueckReife** von Äpfel, Birnen und Pflaumen aufgenommen. Bei Winteräpfeln wird noch die Lagerzeit **lagerZeit** registriert. Die Daten sollen in einer Datei abgespeichert werden. In einer Testversion werden Obstdaten in eine Datei gespeichert und anschließend wieder gelesen.

**Obst.java**

```

// Obst.java
import java.io.*;

/**
 * Klasse (abstrakt) Obst verwaltet fuer verschiedene
 * Obstarten, deren Namen und ihre Pflueckreife.
 */
public abstract class Obst implements Serializable
{
    private String name;
    private int pflueckReife; // Monat

    protected Obst( String name, int pr)
    {
        this.name = name;
        pflueckReife = pr;
    }

    public String toString()
    {
        return "" + name +
            "\t - " + getClass().getName() +
            "\t - PR (Monat): " + pflueckReife;
    }
}

/**
 * Birne.
 */
class Birne extends Obst

```

```
{
    public Birne( String name, int pr)
    {
        super( name, pr);
    }
}

/**
 * Pflaume.
 */
class Pflaume extends Obst
{
    public Pflaume( String name, int pr)
    {
        super( name, pr);
    }
}

/**
 * Apfel.
 */
class Apfel extends Obst
{
    public Apfel( String name, int pr)
    {
        super( name, pr);
    }
}

/**
 * Winterapfel, hat eine Lagerzeit.
 */
class WinterApfel extends Apfel
{
    private int lagerZeit;                // in Monaten

    public WinterApfel( String name, int pr, int zeit)
    {
        super( name, pr);
        lagerZeit = zeit;
    }

    public String toString()
    {
        return super.toString()
            + "\t - LZ(Monate): " + lagerZeit;
    }
}
```

In einem Testprogramm werden verschiedene Obstsorten¹ angelegt, in eine Datei ausgelagert und wieder gelesen. Alle Dateien werden abschließend gelöscht.

ObstMain.java

```
// ObstMain.java                                MM 2007
import java.io.*;
import java.util.*;

/**
 * Testprogramm fuer die Klassen Obst und ObjektDatei.
 */
public class ObstMain
{
    public static void main( String[] args)
    {
        try
        {
            // Aepfel
            Obst apfel =
                new Apfel( "Weisser KlarApfel      ", 7);
            Obst winterApfell1 =
                new WinterApfel( "Roter Boskoop      ", 10, 5);
            Obst winterApfel2 =
                new WinterApfel( "Kaiser Wilhelm    ", 10, 5);
            Obst winterApfel3 =
                new WinterApfel( "Roter Jonathan    ", 9, 7);
            // Birnen
            Obst birnel1 =
                new Birne( "Clapps Liebling    ", 8);
            Obst birne2 =
                new Birne( "Gelbe Wiilliams Christ", 8);
            Obst birne3 =
                new Birne( "Gute Luise          ", 10);
            // Pflaumen
            Obst pflaumel1 =
                new Pflaume( "Hauszwetsche      ", 9);
            Obst pflaume2 =
                new Pflaume( "Koenigin Viktoria  ", 8);
            Obst pflaume3 =
                new Pflaume( "Ontariopflaume    ", 7);

            /* ----- */
            // Schreibe in Datei
            File datei = new File( "Obst.obj");
            ObjektDatei<Obst> obstDatei = new ObjektDatei<Obst>();

            System.out.println( "Schreiben der Datei");
            obstDatei.schreibeInObjektDatei( datei, apfel);
            obstDatei.schreibeInObjektDatei
                ( datei, winterApfell1);
        }
    }
}
```

¹ Daten aus <http://pflanzenboerse-online.de>

```
    obstDatei.schreibeInObjektDatei
        ( datei, winterApfel2);
    obstDatei.schreibeInObjektDatei
        ( datei, winterApfel3);
    obstDatei.schreibeInObjektDatei( datei, birne1);
    obstDatei.schreibeInObjektDatei( datei, birne2);
    obstDatei.schreibeInObjektDatei( datei, pflaume1);
    obstDatei.schreibeInObjektDatei( datei, pflaume2);

    obstDatei.schreibeInObjektDatei( datei, birne3);
    obstDatei.schreibeInObjektDatei( datei, pflaume3);
    System.out.println( "Schreiben abgeschlossen!\n");

/* ----- */
// Kopiere in Datei
File dateiKopie = new File( "ObstKopie.obj");

System.out.println( "Kopieren der Datei");
obstDatei.kopiereObjektDatei( datei, dateiKopie);
System.out.println( "Kopieren abgeschlossen!\n");

/* ----- */
// Lese aus Datei

Vector obst = new Vector();

System.out.println( "Lesen der Kopie");
obst.clear();
obstDatei leseAusObjektDatei( dateiKopie, obst);
obstDatei listeObjekte( obst);
System.out.println( "Lesen abgeschlossen!\n");

/* ----- */
// Zugriff auf Lagerzeit von Winteraepfeln
System.out.println( "Winteraepfel");
for( int i = 0; i < obst.size(); i++)
{
    if( obst.get( i ).getClass().getName()
        == "WinterApfel")
        System.out.println( (WinterApfel)obst.get(i));
}

/* ----- */
// Loeschen der Obstdateien

datei.delete();
dateiKopie.delete();
}
catch( Exception e)
{
    System.out.println( "Fehler: " + e);
}
}
}
```

14.5 Übersicht über häufig verwendete Datenströme

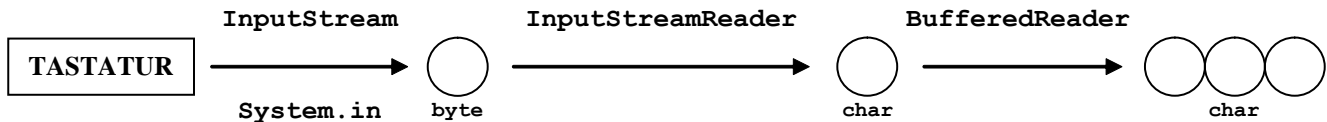
Textdaten (gepuffert)

Methoden

```
read(); readLine(); write(); newLine();
```

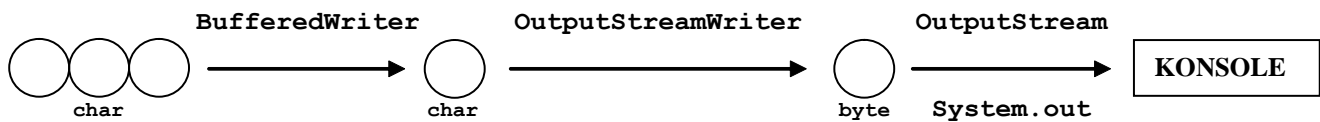
Lesen von Tastatur

```
BufferedReader inBuffer =  
    new BufferedReader( new InputStreamReader( System.in));
```



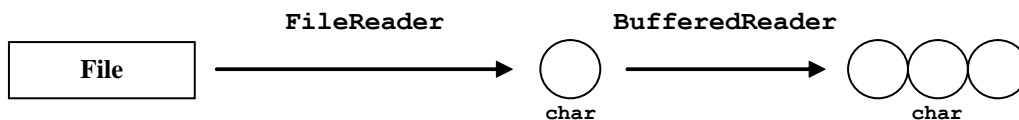
Schreiben auf Konsole

```
BufferedWriter outBuffer =  
    new BufferedWriter( new OutputStreamWriter( System.out));
```



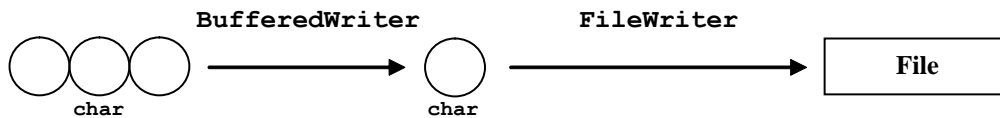
Lesen von Datei

```
BufferedReader inBuffer =  
    new BufferedReader( new FileReader( File datei));
```



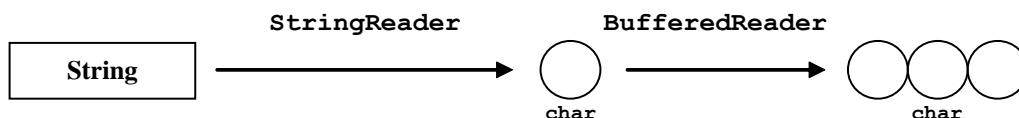
Schreiben auf Datei

```
BufferedWriter outBuffer =  
    new BufferedWriter( new FileWriter( File datei));
```



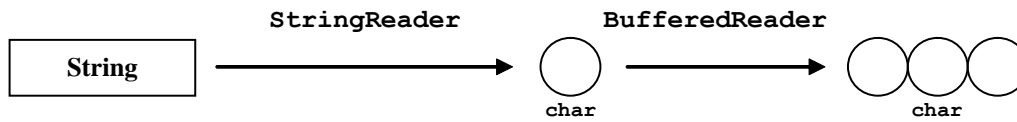
Lesen von String

```
BufferedReader inBuffer =  
    new BufferedReader( new StringReader( String s));
```



Schreiben auf StringBuffer

```
BufferedWriter outBuffer = new BufferedWriter( new StringWriter());
```



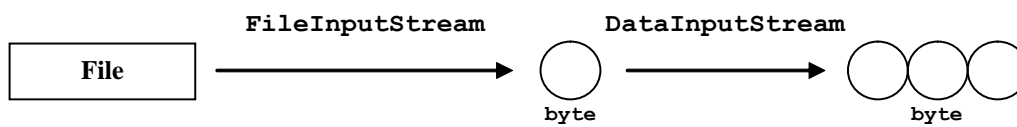
Elementardatentypen (ungepuffert)

Methoden

```
readTtt(); writeTtt(); (Ttt: Elementardatentyp ttt)
```

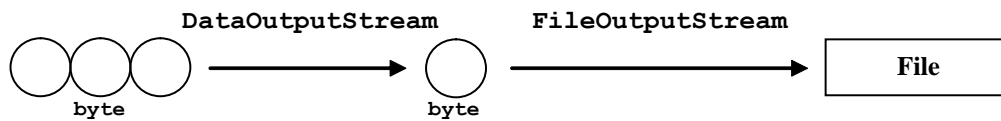
Lesen von Datei

```
DataInputStream inDatei =
    new DataInputStream( new FileInputStream( File datei));
```



Schreiben auf Datei

```
DataOutputStream outDatei =
    new DataOutputStream(new FileOutputStream( File datei));
```



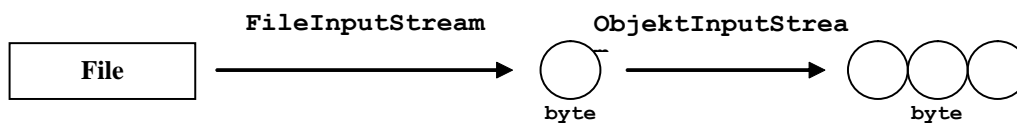
Referenzdatentypen (ungepuffert)

Methoden

```
readObject(); writeObject();
readTtt(); writeTtt(); (Ttt: Elementatdatentyp ttt)
```

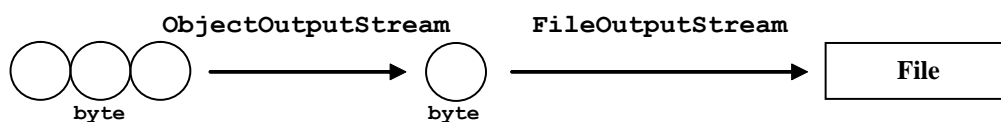
Lesen von Datei

```
ObjectInputStream inDatei =
    new ObjectInputStream( new FileInputStream( File datei));
```



Schreiben auf Datei

```
ObjectOutputStream outDatei =
    new ObjectOutputStream( new FileOutputStream( File datei));
```



14.6 Weitere wichtige Klassen

java.io.PrintStream

java.io.PrintWriter

- Klasse **java.io.PrintStream**, von der Klasse **OutputStream** abgeleitet
- Klasse **java.io. PrintWriter**, von der Klasse **Writer** abgeleitet
- Methoden `print` und `println`

java.io.RandomAccessFile

- Dateizugriffe zum gleichzeitigen Lesen und Schreiben für *Elementardatentypen*
- Methoden gleichnamig denen der Klassen `DataInputStream` und `DataOutputStream`

java.io.StreamTokenizer

- Zerlegung eines *zeichenorientierten* Datenstroms in einzelne Tokens, Leerzeichen dient als Trenner
- `StreamTokenizer` dienen der Analyse von Texten

java.util.ZipInputStream, java.util.ZipOutputStream bzw.

java.util.GZIPInputStream, java.util.GZIPOutputStream

- Verwalten von Daten im Zip- bzw. GZip-Format