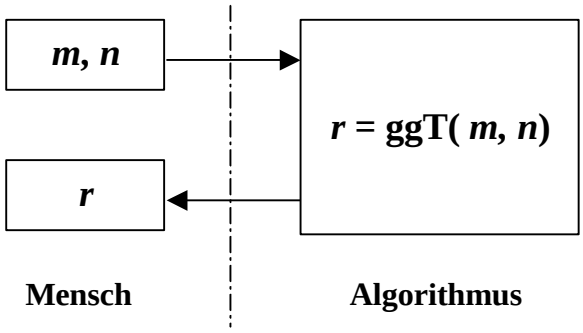
Inhalt

14Dateiverwaltung, das Stream-Konzept

Ein- und Ausgaben werden über sogenannte **Datenströme** realisiert. Durch die *Komplexität* solcher Routinen stoßen insbesondere Anfänger auf große Schwierigkeiten, das Konzept zu verstehen. Deshalb wurden für Java in einem Paket **`Tools`** in einer 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.

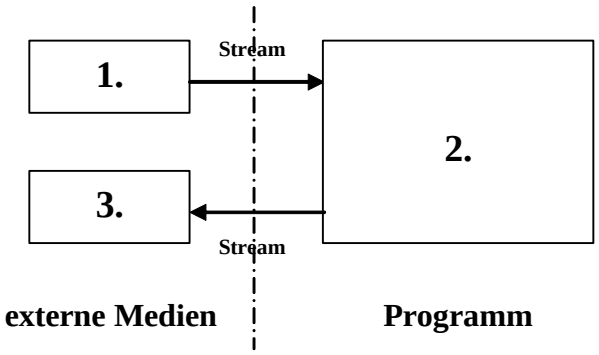
14.1 Datenströme

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**:

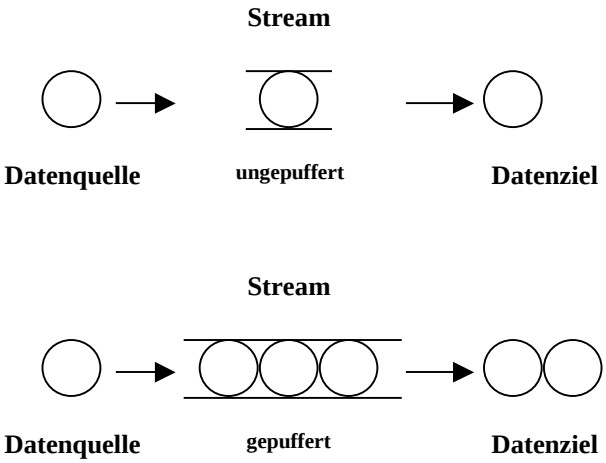
1. **E** ingabe von *Informationen*
2. **V** erarbeitung von *Informationen* entsprechend vorgegebener Arbeitsanweisungen
3. **A** usgabe von *Informationen*



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

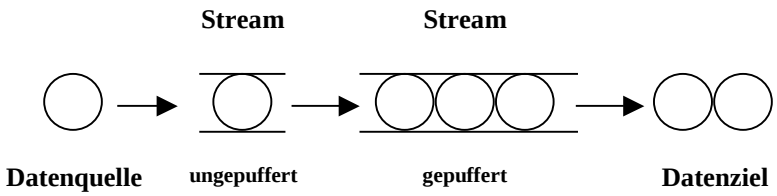
Zu den *Standardaufgaben* eines Programms gehört es, auch auf *extern* vorhandene **Eingabedaten** zuzugreifen und **Ausgabedaten** *extern* abzuspeichern, d.h. der Zugriff auf *externe Medien* (Tastatur, Maus; Monitor, Drucker; Festplatte, ...) muss realisiert werden.

Die Kommunikation mit 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 weiterbefördern. 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.*`

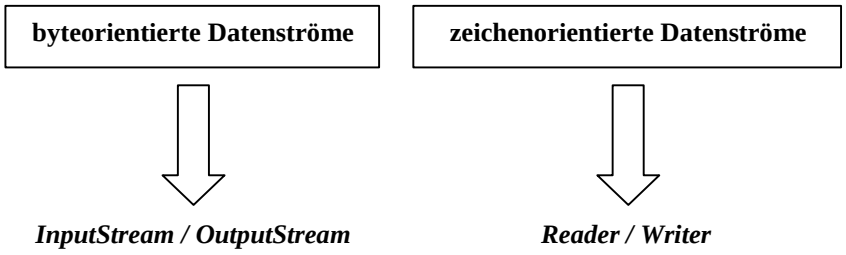
Aus der Sicht der *Programme* wird unterschieden:
- **Eingabedatenströme** zum Lesen von Daten
- **Ausgabedatenströme** zum Schreiben von Daten

Aus der Sicht der *transportierten Dateneinheiten* wird unterschieden:
- **Byteströme** (8-Bit-Dateneiheiten, 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 Lesen 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 Lesen 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.

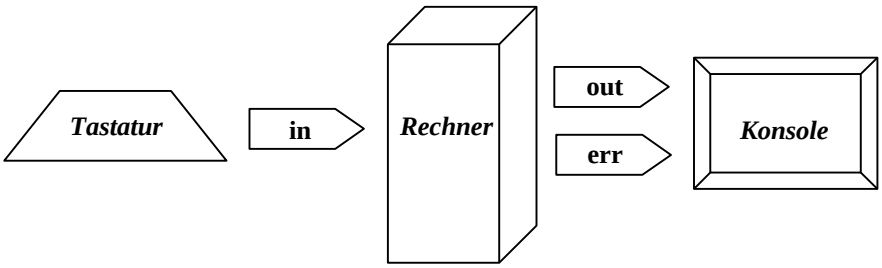
14.1.2 Standarddatenströme

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 *Standard**eingabe**datenstrom*, ein Objekt der *byteorientierten* Klasse **`InputStream`**, und üblicherweise mit der *Tastatur* verbunden.
  ```
  public static final InputStream in;
  ```

- **`System.out`** ist der *Standard**ausgabe**datenstrom*, ein Objekt der *byteorientierten* Klasse **`PrintStream`**, welche von **`OutputStream`** abgeleitet und üblicherweise mit der *Konsole* verbunden ist.
  ```
  public static final PrintStream out;
  ```

- **`System.err`** ist der *Standard**fehlerausgabe**datenstrom*, ebenfalls ein Objekt der *byteorientierten* Klasse **`PrintStream`**, welche üblicherweise auch mit der *Konsole* verbunden ist.
  ```
  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!")`**.

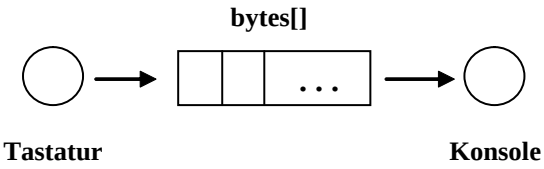
Die Klasse **`PrintStream`** erbt außerdem eine sehr einfache Ausgabemethode **`write`** der Klasse **`OutputStream`**.

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 für jede Taste ein *Tastencode* (bestehend aus 1 oder 2 Byte) über den Standardeingabedatenstrom **`System.in`** in ein Bytefeld **`bytes`** eingelesen. Ist das Feld gefüllt oder der Datenstrom beendet (**`RETURN`**: `\n`, 2 Byte), so werden die eingelesenen Bytes aus dem Feld über den Standardausgabedatenstrom **`System.out`** auf der *Konsole* ausgegeben. Fehlerausschriften werden über den Standardfehlerausgabedatenstrom **`System.err`** an die Konsole geleitet.



Standard.java

```
// Standard.java                                        MM 2014
import java.io.*;                                  // read, write

/**
 * Programm liest maximal 10 Bytes von der Tastatur
 * und gibt diese auf der Konsole aus
 */
public class Standard
{
```

```
  public static void main( String[] args)
  {
    byte[] bytes = new byte[10];       // Feld zum Einlesen

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

      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 einem Objekt der Klasse `File` werden Dateieigenschaften verwaltet, *nicht* deren 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	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
`canRead`	0		`boolean`	**`true`**, falls Leserecht
`canWrite`	0		`boolean`	**`true`**, falls Schreibrecht
`createNewFile`	0		`boolean`	legt neue Datei an
`delete`	0		`boolean`	löscht Datei
`exists`	0		`boolean`	**`true`**, falls Existenz
`getAbsolutePath`	0		`String`	liefert absoluten Pfad
`getName`	0		`String`	liefert Dateinamen
`isDirectory`	0		`boolean`	**`true`**, falls Verzeichnis
`isFile`	0		`boolean`	**`true`**, falls Datei
`length`	0		`long`	liefert Größe der Datei
`list`	0		`String[]`	liefert Verzeichnishierarchie, falls Objekt ein Verzeichnis ist, sonst **`null`**
`mkdir`	0		`boolean`	legt neues Verzeichnis an
`renameTo`	1	`File`	`boolean`	benennt Datei um

FileInfo.java

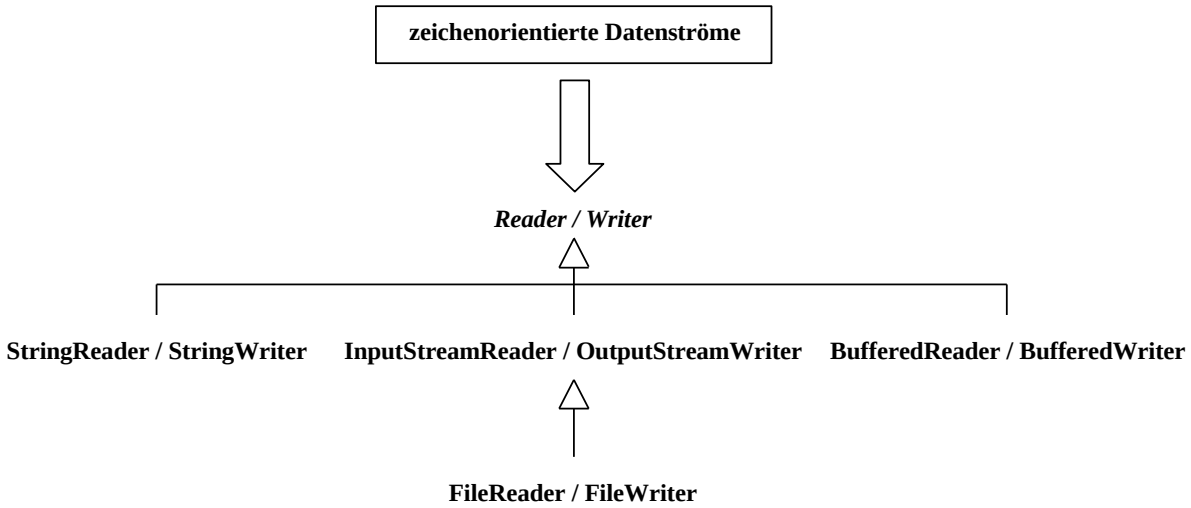
```
// FileInfo.java                                        MM 2014
import java.io.*;                                       // File

/**
 * 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]);

      if( datei.exists())   // Datei-Eigenschaften abfragen
      {
        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
    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
`read`	0		`int`	liefert das nächste Zeichen
`read`	1	`char[]`	`int`	füllt das Feld mit den nächsten Zeichen auf und liefert die Anzahl zurück
`read`	3	`char[], int, int`	`int`	füllt das Feld vom angegebenen Index mit der angegebenen Anzahl von Zeichen auf, liefert die tatsächliche gelesene Anzahl zurück
`close`	0		`void`	schließt den Datenstrom

Methoden der Klasse `java.io.Writer`

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
`write`	1	`int`	`void`	schreibt das angegebene Zeichen
`write`	1	`char[]`	`void`	schreibt die angegebenen Zeichen
`write`	3	`char[], int, int`	`void`	schreibt vom angegebenen Index an die angegebene Anzahl von Zeichen
`write`	1	`String`	`void`	schreibt den angegebene String
`write`	3	`String, int, int`	`void`	schreibt vom angegebenen Index an die angegebene Anzahl von Zeichen des Strings
`close`	0		`void`	schließt den Datenstrom
`flush`	0		`void`	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 2014
import java.io.*;                                 // Reader, Writer

/**
 * Ungepufferte Kopiermethode fuer Texte.
 * Aufruf: java TextKopie <Quelldatei> <Zieldatei>
 */
public class TextKopie
{
/**
 * Kopiert zeichenweise, ungepuffert.
 * @param in Quelle
 * @param out Ziel
 */
  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)
    {
      TextKopie text = new TextKopie();

      Reader in = null;
      Writer out = null;

      try
      {                                    // Eingabedatenstrom
        in = new FileReader( new File( args[ 0]));
                                           // Ausgabedatenstrom
        out = new FileWriter( new File( args[ 1]));

        text.kopieren( in, out);                  // Kopieren
      }
                                // Befehl nicht vollstaendig
      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!");
      }

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

      finally                              // Aufraeumarbeiten
      {
        try
        {                          // Schliessen der Textdateien
          if( in != null) in.close();
          if( out != null) out.close();
        }
        catch( IOException e)
        {
          System.err.println
          ( "Fehler beim Schließen der Dateien!" );
        }
      }

      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.

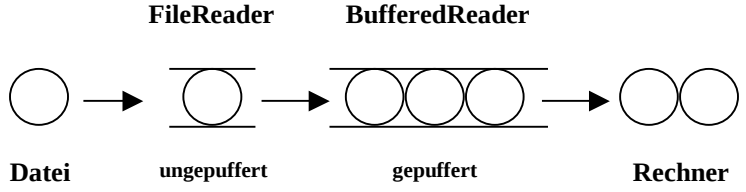
14.3.2 Gepufferte `Reader`- und `Writer`- Klassen

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 anschließende 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* Lesen 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
`readLine`	0		`String`	liefert ganze Textzeile aus dem Puffer, Zeilentrennzeichen: `'\n' '\r'`

Methode der Klasse `java.io.BufferedWriter`

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
`newLine`	0		`void`	schreibt Zeilenwechsel in den Puffer

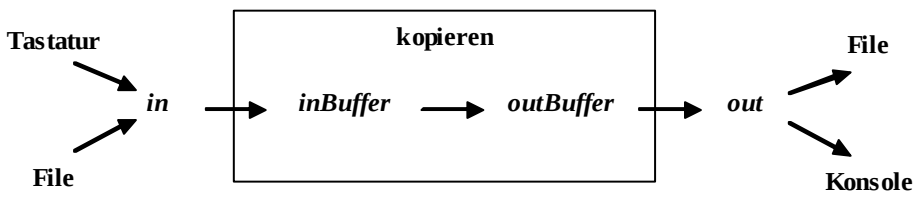
TextKopieMitPuffer.java

```
// TextKopieMitPuffer.java                                MM 2014
import java.io.*;                              // Reader, Writer

/**
 * Gepufferte Kopiermethode fuer Texte.
 * Aufruf:
 * java TextKopieMitPuffer <Quelldatei> <Zieldatei>
 */
public class TextKopieMitPuffer
{
/**
 * Kopiert zeilenweise, gepuffert.
 * @param inBuffer Quelle
 * @param outBuffer Ziel
 */
  public void kopieren
  ( BufferedReader inBuffer, BufferedWriter outBuffer)
    throws Exception
  {
    try
    {
      while( true)
      {                                  // Lesen und Schreiben
        outBuffer.write( inBuffer.readLine());
        outBuffer.newLine();                 // Zeilenwechsel
      }
    }

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

/**
 * Testprogramm.
 */
  public static void main( String[] args)
  {
    TextKopieMitPuffer text = new TextKopieMitPuffer();

    BufferedReader inBuffer = null;
    BufferedWriter outBuffer = null;

    try
    {
      inBuffer =             // Eingabedatenstrom, gepuffert
      new BufferedReader(new FileReader(new File(args[0])));

      outBuffer =            // Ausgabedatenstrom, gepuffert
      new BufferedWriter(new FileWriter(new File(args[1])));

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

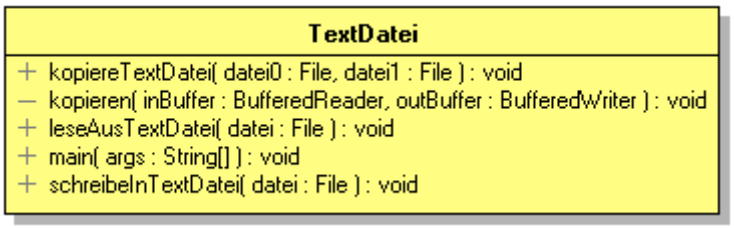
```
                                // Befehl nicht vollstaendig
      catch( ArrayIndexOutOfBoundsException e)
      {
        System.err.print( "Aufruf: java ");
        System.err.println
        ( "TextKopieMitPuffer <Quelldatei> <Zieldatei>");
      }
                                // Textdatei existiert nicht
      catch( FileNotFoundException e)
      {
        System.err.println
        ( "Die Datei >" + args[ 0] + "< existiert nicht!");
      }

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

      finally                              // Aufraeumarbeiten
      {
        try
        {                       // Schliessen der Textdateien
          if( inBuffer != null) inBuffer.close();
          if( outBuffer != null) outBuffer.close();
        }
        catch( IOException e)
        {
          System.err.println
          ( "Fehler beim Schließen der Dateien!" );
        }
      }

      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 *Standardausgabestrom* muss aber nach dem Kopieren *geleert* werden, da sonst evtl. noch Zeichen im Puffer verbleiben und die Schreibaktion nicht bis zum Ende ausgeführt wird. Das Schließen mittels **`close`** erledigt diese Aufräumarbeiten automatisch.



TextDatei.java

```
// TextDatei.java                                          MM 2014
import java.io.*;                              // Reader, Writer

/**
 * 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
{
/**
 * Kopiert zeilenweise, gepuffert.
 * @param inBuffer Quelle
 * @param outBuffer Ziel
 */
  private void kopieren
     ( BufferedReader inBuffer, BufferedWriter outBuffer)
     throws Exception
  {
     try
     {
       while( true)
       {                                  // Lesen und Schreiben
         outBuffer.write( inBuffer.readLine(););
         outBuffer.newLine();                 // Zeilenwechsel
       }
     }

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

/**
```

```
 * Liest Text von der Tastatur
 * und schreibt ihn in eine Datei.
 * @param datei Zieldatei
 */
  public void schreibeInTextDatei( File datei)
    throws Exception
  {
    BufferedReader inBuffer = null;
    BufferedWriter outBuffer = null;

    try
    {               // Tastatureingabedatenstrom, gepuffert
      inBuffer =
      new BufferedReader( new InputStreamReader( System.in));
                  // Ausgabedatenstrom auf datei, gepuffert
      outBuffer =
      new BufferedWriter( new FileWriter( datei));

      kopieren( inBuffer, outBuffer);           // Kopieren
    }

    catch( Exception e)       // Fehlerbehandlung allgemein
    {
      System.err.println( "Fehler beim Schreiben!");
      throw e;
    }

    finally                                 // Aufraeumarbeiten
    {
      try
      {                                 // Schliessen der Datei
        if( outBuffer != null) outBuffer.close();
      }
      catch( IOException e)
      {
        System.err.println
        ( "Fehler beim Schließen der Textdatei!" );
      }
    }
  }

/**
 * Kopiert eine Textdatei zeilenweise in eine Textdatei.
 * @param datei0 Quelldatei
 * @param datei1 Zieldatei
 */
  public void kopiereTextDatei( File datei0, File datei1)
    throws Exception
  {
    BufferedReader inBuffer = null;
    BufferedWriter outBuffer = null;
```

```
      try
      {            // Eingabedatenstrom von datei0, gepuffert
        inBuffer =
        new BufferedReader( new FileReader( datei0));
                   // Ausgabedatenstrom auf datei1, gepuffert
        outBuffer =
        new BufferedWriter( new FileWriter( datei1));

        kopieren( inBuffer, outBuffer);           // Kopieren
      }

      catch( Exception e)       // Fehlerbehandlung allgemein
      {
        System.err.println( "Fehler beim Schreiben!");
        throw e;
      }

      finally                                 // Aufraeumarbeiten
      {
        try
        {                              // Schliessen der Dateien
          if( inBuffer != null) inBuffer.close();
          if( outBuffer != null) outBuffer.close();
        }
        catch( IOException e)
        {
          System.err.println
          ( "Fehler beim Schließen der Textdateien!" );
        }
      }
    }

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

      try
      {             // Eingabedatenstrom von datei, gepuffert
        inBuffer =
        new BufferedReader( new FileReader( datei));
                  // Ausgabedatenstrom auf Konsole, gepuffert
        outBuffer =
        new BufferedWriter(new OutputStreamWriter(System.out));

        kopieren( inBuffer, outBuffer);           // Kopieren
      }
```

```
    catch( Exception e)       // Fehlerbehandlung allgemein
    {
      System.err.println( "Fehler beim Schreiben!");
      throw e;
    }

    finally                              // Aufraeumarbeiten
    {
      try
      {                          // Schliessen der Textdatei
        if( inBuffer != null) inBuffer.close();
                                 // Leeren des Ausgabepuffers
        if( outBuffer != null) outBuffer.flush();
      }
      catch( IOException e)
      {
        System.err.println
        ( "Fehler beim Schließen der Textdatei!" );
      }
    }
  }

/**
 * Testprogramm,
 * Tastatur --> Datei1 --> Datei2 --> Konsole.
 */
  public static void main( String[] args)
  {
    TextDatei text = new TextDatei();

    try
    {
      File datei0 = new File( args[ 0]);
      File datei1 = new File( args[ 1]);

// 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
      catch( ArrayIndexOutOfBoundsException e)
      {
        System.err.println
        ( "Auruf: java TextDatei <Quelldatei> <Zieldatei>");
      }

      catch( Exception e)       // Fehlerbehandlung allgemein
      {
        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, das *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. Java stellt Datenströme zur Nutzung von Dateien für andere Datentypen zur Verfügung.

14.4.1 Klassen `InputStream` und `OutputStream`

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

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

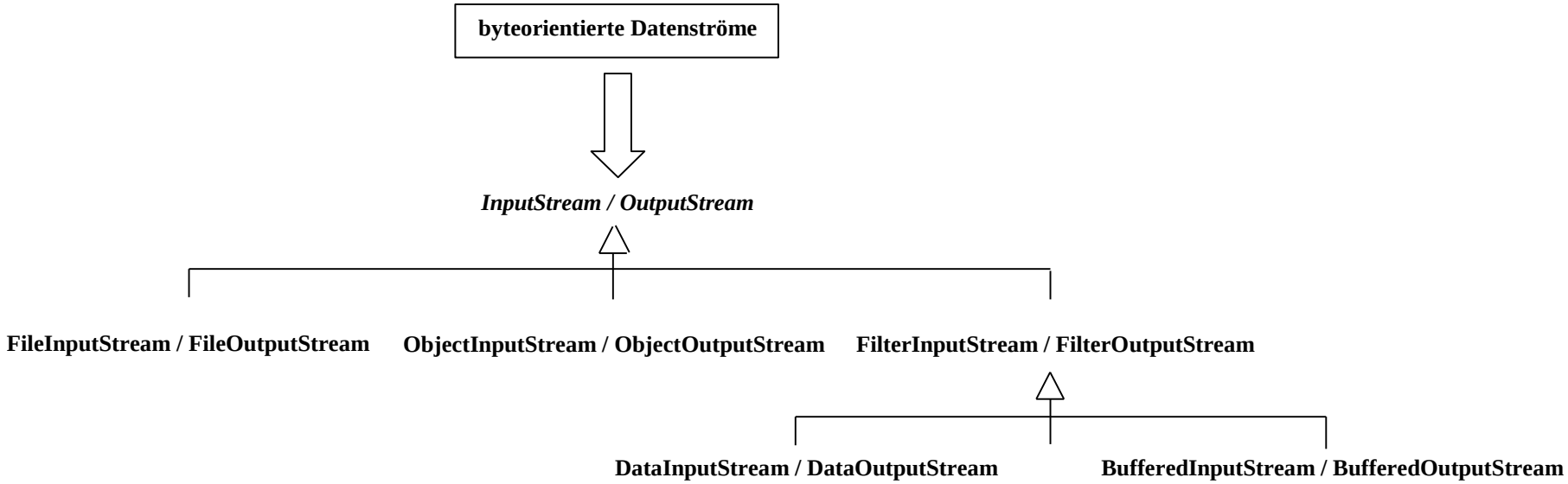
Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
`read`	0		`int`	liefert das nächste Byte als **`int`**-Wert
`read`	1	`byte[]`	`int`	füllt das Feld mit den nächsten Bytes auf und liefert die Anzahl zurück
`read`	3	`byte[], int, int`	`int`	füllt das Feld vom angegebenen Index mit der angegebenen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

Methoden der Klasse `java.io.OutputStream`

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

Klassen für *byteorientierte* Datenströme sind ähnlich 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. aus Dateien 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. Durch diese binäre Speicherung wird eine deutlich verringerte Dateigröße erreicht.

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

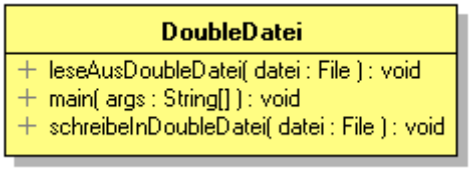
Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
`read`*`Ttt`*	0		***`ttt`***	liefert einen Wert des Datentyps ***`ttt`***

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

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
`write`*`Ttt`*	1	***`ttt`***	`void`	schreibt den Wert des Datentyps ***`ttt`***

`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.



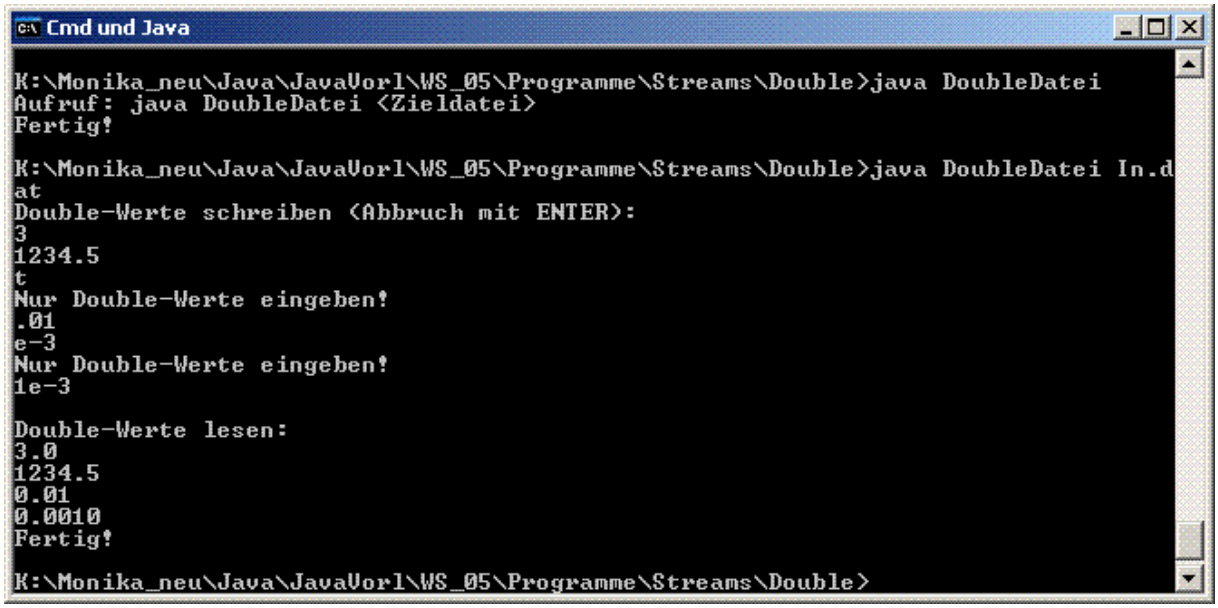
DoubleDatei.java

```
// DoubleDatei.java                                        MM 2014
import java.io.*;                  // InputStream, OutputStream

/**
 * Programm schreibt und liest Double-Werte
 * in bzw. aus einer Datei.
 */
public class DoubleDatei
{
/**
 * Liest Double-Werte aus einer Datei
 * und schreibt sie auf der Konsole aus.
 * @param datei Dateiname
 */
  public void leseAusDoubleDatei( File datei)
    throws Exception
  {
    DataInputStream inDatei = null;
```

```
      try
      {
        inDatei =              // Eingabedatenstrom auf datei
        new DataInputStream(new FileInputStream( datei));

        while( true)
        {
                                // Lesen eines Double-Wertes
          double zahl = inDatei.readDouble();

                // Schreiben eines Double-Wertes auf Konsole
          System.out.println( "" + zahl);
        }
      }
                                 // Fehler beim Zahlenformat
      catch( NumberFormatException e)
      {
        System.err.println
        ( "Auf der Datei befinden sich Nicht-Double-Werte!");
      }
                  // Dateiende erreicht, Schliessen der Datei
      catch ( EOFException e){}

      catch( Exception e)       // Fehlerbehandlung allgemein
      {
        System.err.println( "Fehler beim Schreiben!");
        throw e;
      }

      finally                                 // Aufraeumarbeiten
      {
        try
        {                       // Schliessen der Dateidatei
          if( inDatei != null) inDatei.close();
        }
        catch( IOException e)
        {
          System.err.println
          ( "Fehler beim Schließen der Datendatei!" );
        }
      }
    }

  /**
   * Liest Double-Werte von der Tastatur
   * und schreibt sie in eine Datei.
   * @param datei Dateiname
   */
    public void schreibeInDoubleDatei( File datei)
      throws Exception
    {
```

```
      DataOutputStream outDatei = null;

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

        outDatei =             // Ausgabedatenstrom auf datei
        new DataOutputStream( new FileOutputStream( datei));

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

        while( !zeile.equals( ""))
        {
          try
          {       // Schreiben des Double-Wertes in die Datei
            double zahl = Double.parseDouble( zeile);
            outDatei.writeDouble( zahl);
          }
                                  // Fehler beim Zahlenformat
          catch( NumberFormatException e)
          {
            System.err.println( "Nur Double-Werte eingeben!");
          }
                   // Eingabe eines Double-Werts von Tastatur
          zeile = inBuffer.readLine();
        }
      }

      catch( Exception e)       // Fehlerbehandlung allgemein
      {
        System.err.println( "Fehler beim Schreiben!");
        throw e;
      }

      finally                                 // Aufraeumarbeiten
      {
        try
        {                          // Schliessen der Dateidatei
          if( outDatei != null) outDatei.close();
        }
        catch( IOException e)
        {
          System.err.println
          ( "Fehler beim Schließen der Datendatei!" );
        }
      }
    }

  /**
   * Testprogramm.
```

```
  */
   public static void main( String[] args)
   {
     DoubleDatei daten = new DoubleDatei();

     try
     {
       File datei = new File( args[ 0]);

       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
     catch( ArrayIndexOutOfBoundsException e)
     {
       System.err.println
       ( "Aufruf: java DoubleDatei <Zieldatei>");
     }
     catch( Exception e)        // Fehlerbehandlung allgemein
     {
       System.err.println( "Fehler: " + e);
     }

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



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

In.dat

```
@□        @"J       ?„záG®□{?PbMÒñ©ü
```

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`

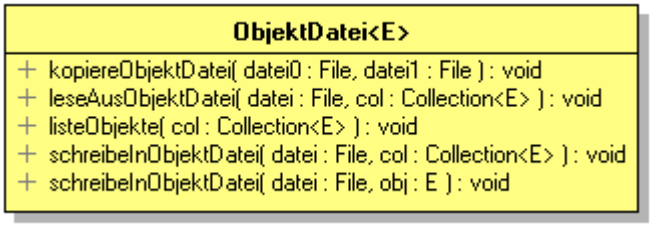
Name	Parameter-anzahl	Parameter-typ	Ergebnis-typ	Beschreibung
`readObject`	0		**`Object`**	deserialisiert ein Objekt
`read`*`Ttt`*	0		***`ttt`***	liefert einen Wert des Datentyps ***`ttt`***

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

Name	Parameter-anzahl	Parameter-typ	Ergebnis-typ	Beschreibung
`writeObject`	1	**`Object`**	`void`	serialisiert ein Objekt
`write`*`Ttt`*	1	***`ttt`***	`void`	schreibt den Wert des Datentyps ***`ttt`***

Objekte können nur *serialisiert* werden, wenn sie das Interface **`java.io.Serializable`** implementieren. *Dieses Interface hat weder Methoden noch Attribute.* Es ist ein sogenanntes **marker**-Interface, welches dazu dient, serialisierbare Klassen zu kennzeichnen. Deshalb genügt es, den Kopf der Objektklasse mit **`implements Serializable`** zu ergänzen.
Für Elementardatentypen verwendet man entsprechende **Wrapper**-Klassen. Diese sind generell serialisiert.

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



ObjektDatei.java

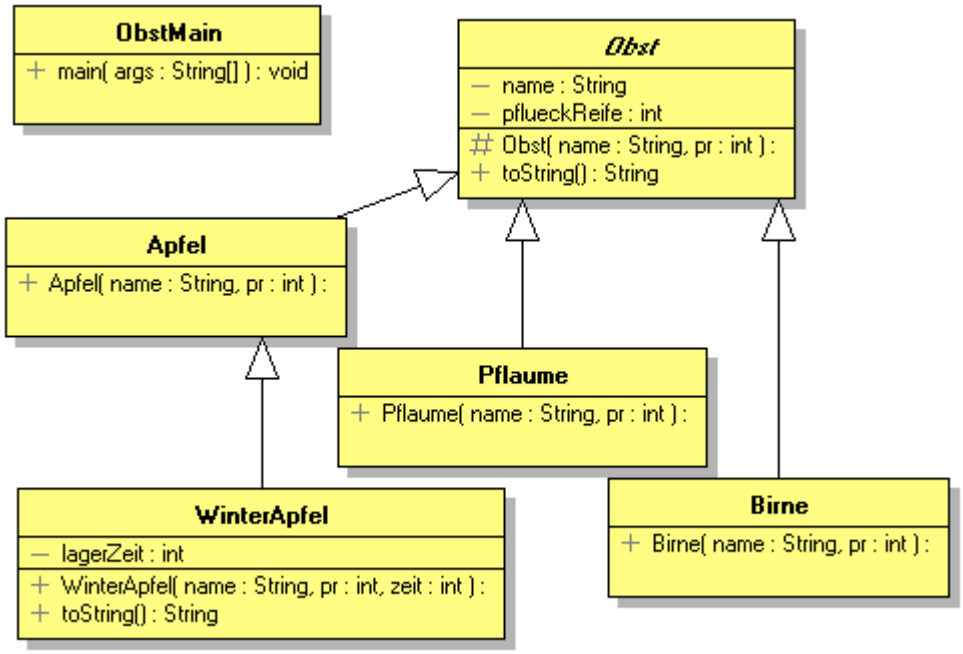
```
// ObjektDatei.java                                    MM 2014
import java.io.*;              // Inputstream, OutputStream
import java.util.*;                    // Collection, Vector

/**
 * 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
  {
    ObjectInputStream inDatei = null;

    try
    {
      inDatei =
      new ObjectInputStream( new FileInputStream( datei));

      while( true) col.add( (E)inDatei.readObject());
    }
              // Dateiende erreicht, Schliessen der Datei
    catch ( EOFException e){}

    catch( Exception e)       // Fehlerbehandlung allgemein
    {
      System.err.println( "Fehler beim Schreiben!");
      throw e;
    }

    finally                                // Aufraeumarbeiten
    {
      try
      {                       // Schliessen der Objektdatei
        if( inDatei != null) inDatei.close();
      }
      catch( IOException e)
      {
        System.err.println
        ( "Fehler beim Schließen der Objektdatei!");
```

```
        }
      }
    }

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

      try
      {
        outDatei =
        new ObjectOutputStream( new FileOutputStream( datei));

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

      catch( Exception e)       // Fehlerbehandlung allgemein
      {
        System.err.println( "Fehler beim Schreiben!");
        throw e;
      }

      finally                                // Aufraeumarbeiten
      {
        try
        {                          // Schliessen der Dateidatei
          if( outDatei != null) outDatei.close();
        }
        catch( IOException e)
        {
          System.err.println
          ( "Fehler beim Schließen der Datendatei!" );
        }
      }
    }

  /**
   * 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
    {
      Collection<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)
    {
      int i = 0;
      for( E obj: col)
        System.out.println( "" + i++ + " " + obj);
    }
  }
```

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 Äpfeln, 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                                              MM 2014
import java.io.*;                                  // Serializable

/**
 * 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 Obstsorten[1] angelegt, in eine Datei ausgelagert und wieder gelesen. Alle Dateien werden abschließend gelöscht.

ObstMain.java

```
// ObstMain.java                                          MM 2014
import java.io.*;                                          // File
import java.util.*;                                       // Vector

/**
 * Testprogramm fuer die Klassen Obst und ObjektDatei.
 */
public class ObstMain
{
  public static void main( String[] args)
  {
/* ------------------------------------------------------ */
                                                    // Aepfel
      Obst apfel =
      new Apfel( "Weisser KlarApfel    ", 7);
      Obst winterApfel1 =
      new WinterApfel( "Roter Boskoop        ", 10, 5);
      Obst winterApfel2 =
      new WinterApfel( "Kaiser Wilhelm       ", 10, 5);
      Obst winterApfel3 =
      new WinterApfel( "Roter Jonathan       ", 9, 7);

                                                    // Birnen
      Obst birne1 =
      new Birne( "Clapps Liebling      ", 8);
      Obst birne2 =
      new Birne( "Gelbe Wiiliams Christ", 8);
      Obst birne3 =
      new Birne( "Gute Luise           ", 10);

                                                 // Pflaumen
      Obst pflaume1 =
      new Pflaume( "Hauszwetsche         ", 9);
      Obst pflaume2 =
      new Pflaume( "Koenigin Viktoria    ", 8);
      Obst pflaume3 =
      new Pflaume( "Ontariopflaume       ", 7);

/* ------------------------------------------------------ */
                                                 // Obstdatei
    ObjektDatei<Obst> obstDatei = new ObjektDatei<Obst>();

    File datei = new File( "Obst.obj");
    File dateiKopie = new File( "ObstKopie.obj");

    try
```

[1] Daten aus http://pflanzenboerse-online.de

```
    {
      System.out.println( "Schreiben der Datei");

      try
      {
                                        // Schreibe in Datei
        obstDatei.schreibeInObjektDatei( datei, apfel);
        obstDatei.schreibeInObjektDatei
                                ( datei, winterApfel1);
        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);
      }

      catch( Exception e)     // Fehlerbehandlung allgemein
      {
        System.err.println( "Fehler beim Schreiben! + e");
        return;
      }

      System.out.println( "Schreiben abgeschlossen!\n");

/* ------------------------------------------------------ */
                                        // Kopiere in Datei
      System.out.println( "Kopieren der Datei");

      try
      {
        obstDatei.kopiereObjektDatei( datei, dateiKopie);
      }

      catch( Exception e)     // Fehlerbehandlung allgemein
      {
        System.err.println( "Fehler beim Kopieren! + e");
        return;
      }

      System.out.println( "Kopieren abgeschlossen!\n");

/* ------------------------------------------------------ */
                                        // Lese aus Datei
      Vector obst = new Vector();
      obst.clear();
```

```
      System.out.println( "Lesen der Kopie");

      try
      {
        obstDatei.leseAusObjektDatei( dateiKopie, obst);
        obstDatei.listeObjekte( obst);
      }

      catch( Exception e)     // Fehlerbehandlung allgemein
      {
        System.err.println( "Fehler beim Lesen! + e");
        return;
      }

      System.out.println( "Lesen abgeschlossen!\n");

/* ------------------------------------------------------ */
                // Zugiff auf Lagerzeit von Winteraepfeln
      System.out.println( "Winteraepfel");

      for( int i = 0; i < obst.size(); i++)
      {
        if(obst.get(i).getClass().getName().equals("WinterApfel"))
          System.out.println((WinterApfel)obst.get(i));
      }
    }

    catch( Exception e)       // Fehlerbehandlung allgemein
    {
      System.err.println( e);
    }

/* ------------------------------------------------------ */
    finally                                 // Aufraeumarbeiten
    {                           // Loeschen der Obstdateien
      datei.delete();
      dateiKopie.delete();
    }
  }
}
```

14.5 Übersicht über häufig verwendete Datenströme

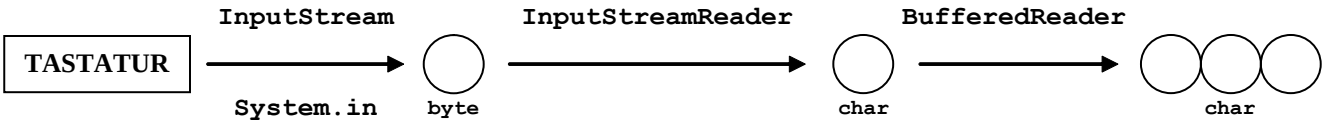
Textdaten (gepuffert)

Methoden

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

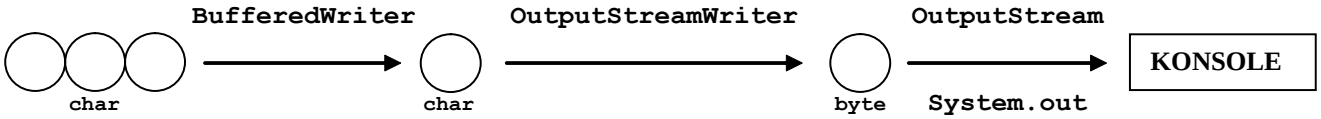
Lesen von Tastatur

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



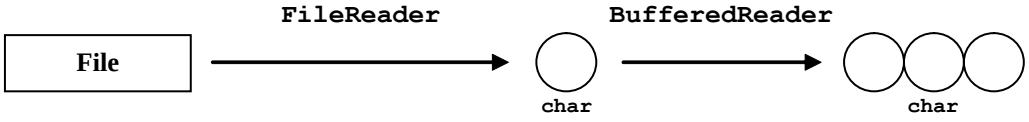
Schreiben auf Konsole

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



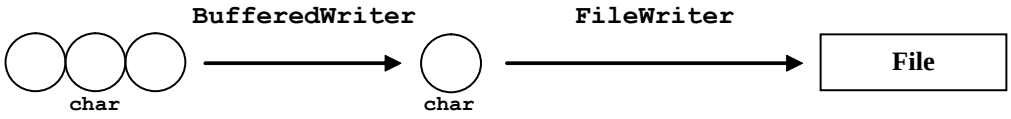
Lesen von Datei

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



Schreiben auf Datei

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



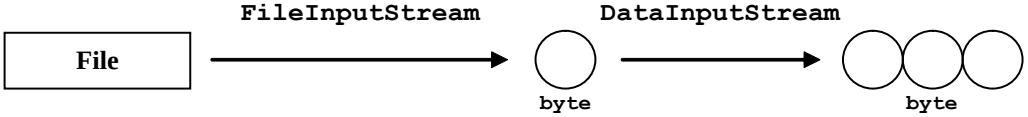
Elementardatentypen (ungepuffert)

Methoden

`read`*`Ttt`*`(); write`*`Ttt`*`();` (*`Ttt`* : Elementardatentyp *`ttt`*)

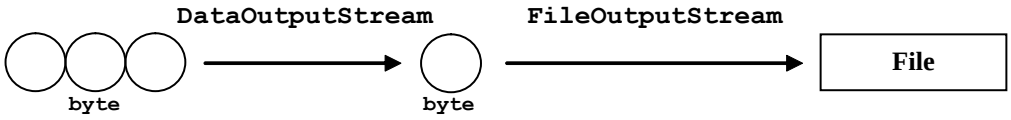
Lesen von Datei

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



Schreiben auf Datei

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



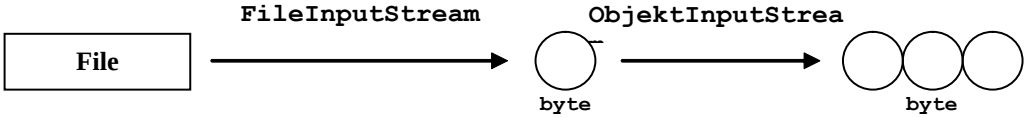
Referenzdatentypen (ungepuffert)

Methoden

```
readObject(); writeObject();
```
`read`*`Ttt`*`(); write`*`Ttt`*`();` (*`Ttt`* : Elementatdatentyp *`ttt`*)

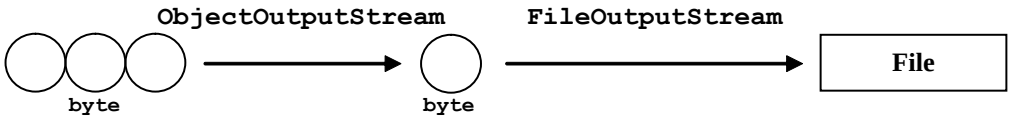
Lesen von Datei

```
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