Grundlagen der Informatik für Ingenieure I

13 Java-E/A-System

13.1. E/A-System-Überblick
13.2 Dateisystem - Betriebssystemsicht
13.3 Java-I/O-System
13.4 I/O-Klassenhierachie
13.5 InputStream class; Reader class
13.5.1 read() method
13.5.2 skip() method
13.5.3 available() und ready() methods
13.5.4 close() method

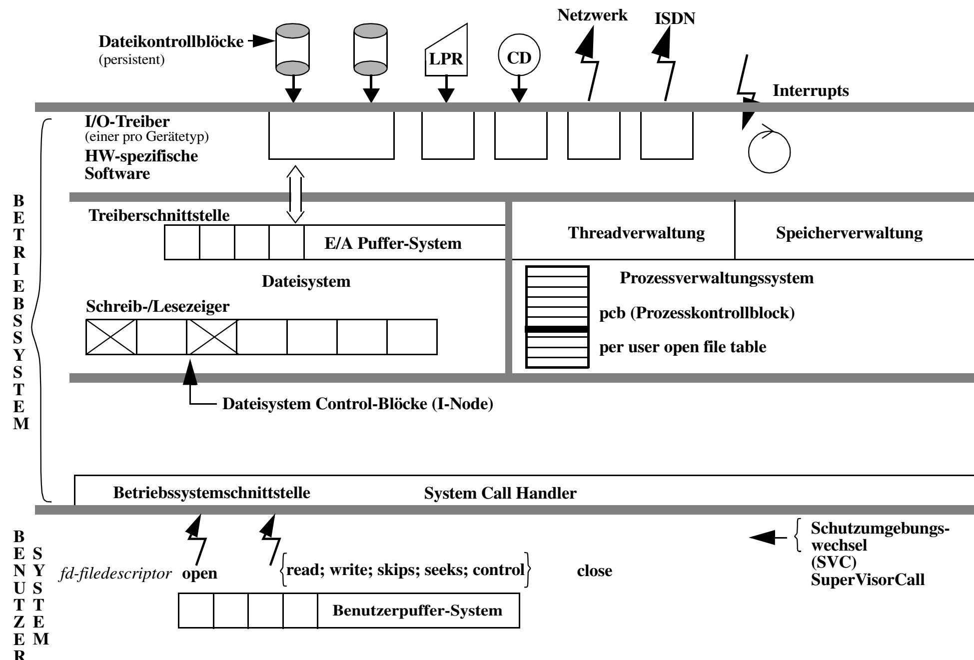
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13 Java-E/A-System

13.6 InputStream Subklassen
13.7 Reader Subklassen
13.8 File class
13.9 Beispiel: Lesen einer Datei
13.10 Output Stream class; Writer class
13.10.1 write() method
13.10.2 flush() und close() methods
13.11 OutputStream Subklassen
13.12 Writer Subklasse
13.13 Formatierte Ein-/Ausgabe
13.13.1 DataInputInterface
13.13.2 DataOutput Interface
13.13.3 Formatierte I/O - Ein Beispiel
13.13.4 Print Stream
13.14 Object Input/Output
13.15 Random Access Files
13.16 Anhang: Lesen einer Datei “by Line”

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.1 E/A-System - Überblick



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.1 E/A-System - Überblick

- Im E/A-System sind sämtliche E/A-Geräte zusammengefasst.
  Man unterscheidet aufgrund der Hardwareeigenschaften:
  - zeichenorientierte Geräte (wie: Bildschirm, Keyboard, Drucker...) und
  - blockorientierte Geräte (wie: Platten, Magnetbänder, CD-ROMs...)
- Beim Betriebssystem UNIX sind die physikalischen E/A-Geräte in das Dateisystem integriert:
  - Das Schreiben auf den Drucker bedeutet nichts anderes als eine Dateiausgabe z. B. auf die Datei “/dev/lp1”.
  - Das direkte Schreiben auf eine Platte bedeutet z. B. die Ausgabe auf die Datei “/dev/disk1”.
- Im Gegensatz zur E/A auf ein Peripheriegerät wird bei der E/A auf eine Datei bei UNIX von der E/A auf reguläre Dateien gesprochen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.2 Dateisystem - Betriebssystemsicht

- Aus (UNIX-) Betriebssystemsicht wird eine Datei als ein Strom von Daten (Bytes) ohne jede Struktur betrachtet:
  - die Daten werden nicht interpretiert.
- Die Strukturen auf Daten werden durch die Programme geprägt, die diese Dateien benutzen.
  - So erwartet z. B. ein Compiler oder ein Lader “gewisse” Formate und Formatinformationen.
  - Entsprechend prägen auch Java-Programm-Systeme Dateien Strukturen auf.
- Zwischen Arbeitsspeicher- und Plattenspeichersystemen gibt es erhebliche Verarbeitungszeit - Differenzen.
  - Das Betriebssystem versucht durch “vorausschauende” Maßnahmen die tatsächliche Verarbeitungsdauer dem Benutzer gegenüber durch interne Parallelverarbeitung zu verbergen.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.2 Dateisystem - Betriebssystemsicht

- Auf der Hardwareebene werden alle Datensätze und Datenblöcke vom Betriebssystem auf Plattenblöcke fester Größe abgebildet.
- Diese bilden die Transfereinheiten zu den Peripheriegeräten.
- Bei der Plattenperipherie ist die Größe von 8 kByte (z. Zt.) sehr verbreitet.
- Die effizienteste Transfereinheit ist abhängig von der Plattentechnologie und den internen Plattenpuffersystemen in der Platteneinheit selbst.
- moderne Systeme arbeiten mit var. Blockgrößen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.3 Java-I/O -System

- Das Java-I/O-System ist ***Streams***-basiert.
  - ***Streams*** sind (vom Betriebssystem uninterpretierte) Datenströme von einer Datenquelle zu einer Datensenke. Man unterscheidet
    - ***Inputstreams*** und
    - ***Outputstreams***.
  - Die physikalischen Ein-/Ausgabegeräte sind in das Unix Filesystem integriert.
  - Zusammen mit dem ***Streams***-Konzept hat man so ein einheitliches Datentransportkonzept geschaffen, dass sowohl auf physikalische Geräte als auch auf Dateien angewendet werden kann.
  - Da diesen E/A-Systemen auch immer Puffersysteme unterlegt sind, handelt es sich, abstrakt gesehen, bei ***Streams*** um Datenströme zwischen Puffern.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.3 Java-I/O -System

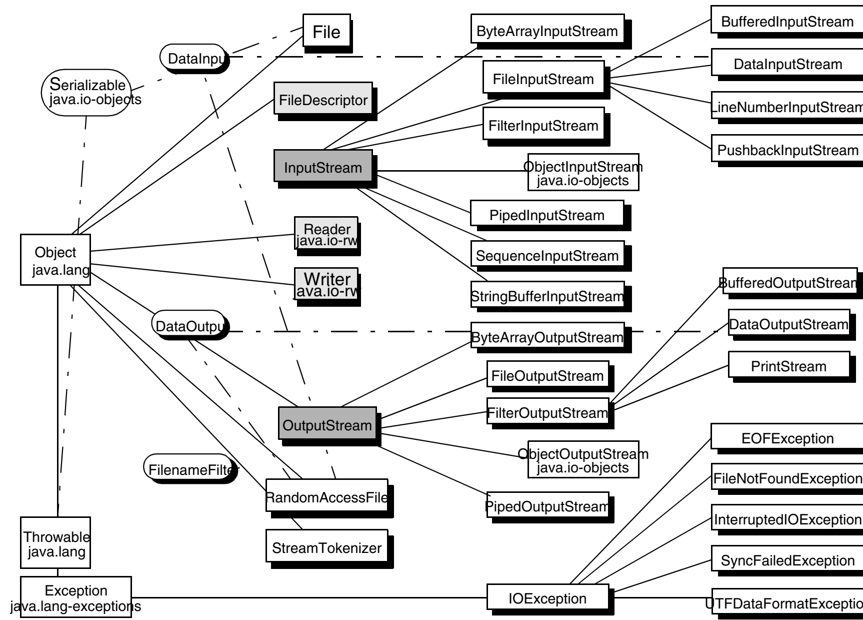
- In Java wird eine Hierachie von I/O-Mechanismen angeboten.
  - Dabei wird versucht das System symmetrisch hinsichtlich *Input* und *Output* zu gestalten.
  - Deshalb wird im Rahmen dieser Vorlesung die Eingabe ausführlicher behandelt.
  - Die Ausgaben sind dann jeweils analog zu gestalten.

- Die **abstrakten** Basisklassen sind:
  - *InputStream class,*
  - *Reader class,*
  - *OutputStreams class,*
  - *Writer class,*

  wobei Lesen und Schreiben immer aus der Sicht des aufrufenden Programms zu sehen ist.

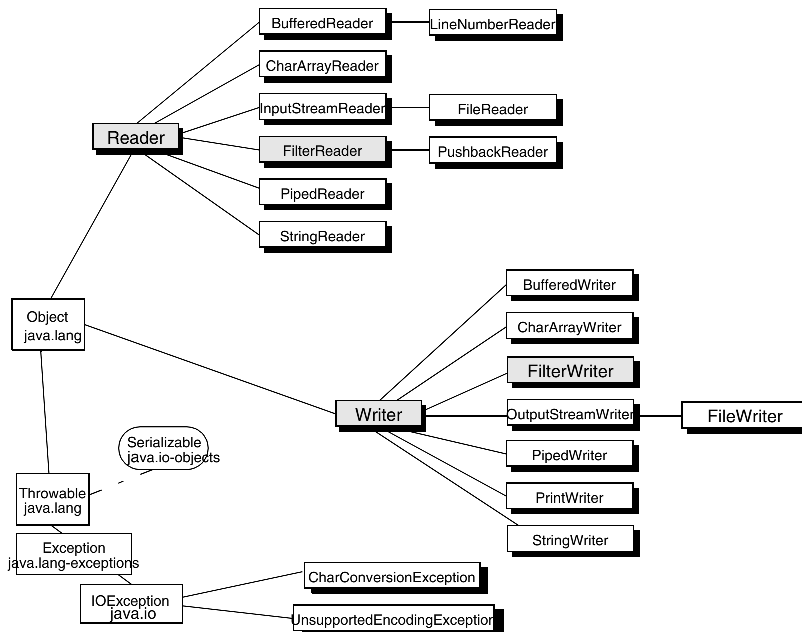
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.4 I/O-Klassenhierachie



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.4 I/O-Klassenhierachie



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.5 InputStream class; Reader class

- Die **abstrakten** *Input classes*:
  - ***InputStream class*** definiert die Mechanismen, die notwendig sind, einen **Bytestrom** einer Datenquelle *(source)* zu lesen.
  - ***Reader class*** definiert die Mechanismen die notwendig sind, einen ***Character*-Strom** einer Datenquelle *(source)* zu lesen.
  - Der einzige Unterschied zwischen diesen Klassen ist die verschiedenen Interpretation der Elemente der Datenströme.

- Folgende Methoden werden von diesen Klassen bereitgestellt:
  - *open()* gibt es im klassischen Sinne nicht!!!
    open() ist im Instantiieren eines *Stream*-Objekts verborgen.
  - *read()*
  - *skip()*
  - *available() und ready(); mark und reset()*
  - *close()*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 read() method

- Es gibt 3 Varianten der ***read() method***:

```
int read( byteBuffer )

int read( byteBuffer, fromIndex, nbOfElements )

int read()
```

  - Diese Methoden sind blockierend:
    - d.h. die Kontrolle wird erst dann an den Aufrufer zurückgegeben, wenn der angeforderte Eingabestrom vollständig zur Verfügung steht.
  - Es empfiehl sich allgemein für E/A-Vorgänge eigenen *threads* vorzusehen, wenn Parallelarbeit möglich ist.
    (Im Rahmen dieser Vorlesung nicht notwendig!)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 read() method

◆ Beispiel: Blockweises Lesen

```
final int BLOCKSIZE = 1024;
InputStream streamIn = new FileInputStream( "pathname of
                                             the file" );
byte[] byteBuffer = new byte[BLOCKSIZE];

if ( streamIn.read( byteBuffer ) != byteBuffer.length )
    System.out.println( " I got less than expected" );
...
```

- Die *read() method* liest Daten von der Quelle bis der Puffer gefüllt ist.
- Ist die Quelle nicht in der Lage die dafür notwendigen Daten bereitzustellen - z. B. weil vorher das Ende der Datei erreicht wurde - werden nur die tatsächlich vorhandenen Bytes gelesen.
- Als Returnparameter wird die Anzahl der tatsächlich gelesenen Bytes zurückgegeben.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 read() method

◆ Das gleiche Beispiel mit der ***Reader class***

```
final int BLOCKSIZE = 1024;
Reader readerIn = new FileReader ( "pathname of the file" );
char[] charBuffer = new char[BLOCKSIZE];

if ( readerIn(charBuffer ) != charBuffer.length
    System.out.println( " I got less than expected" );
...
```

◆ Die zweite Variante der ***read() method*** zum Lesen von Teilbereichen:

```
int read ( byteBuffer, fromIndex, nbOfElements )

z. B.
streamIn.read( byteBuffer, 100, 300 );
readerIn.read( charBuffer, 100, 300 );
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 read() method

◆ Lesen von Einzelbytes bzw. Einzelzeichen:

```
final int EOF = -1;
final int BLOCKSIZE = 1024;
InputStream streamIn = new FileInputStream( "pathname of the file" );
byte[] byteBuffer = new byte[BLOCKSIZE];
int i, dataRead, data;
i = 0;

while ( (data = streamIN.read() ) != EOF ) {
    byteBuffer[i] = (byte) data;  // casting; weil "int read()"
    i = i + 1;
}
dataRead = i;
...
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 skip() method

◆ Mit der *skip() method* kann man eine angegebene Anzahl von Zeichen bzw. Bytes überspringen.

◆ Man kann sich das so vorstellen, dass man einen Lesezeiger um eine bestimmte Zahl von Zeichen/Bytes weiter positioniert, ohne Zeichen/Bytes zu lesen:

```
// Lesen jedes ungeraden Blocks(Zählung beginnt bei 0!):

final int BLOCKSIZE = 1024;
long skipReturn; // Returnparameter von skip ist vom Typ "long"!

InputStream streamIn = new FileInputStream("pathname of the file");
byte[] byteBuffer = new byte[BLOCKSIZE];

while ( ( skipReturn = streamIn.skip( BLOCKSIZE ) != -1 ) {
         if (streamIn.read(byteBuffer) != byteBuffer.length {
            System.out.println( " Read: I got less than expected" );
         }
}
System.out.println("Skip: No more complete blocks");

// Kommen beide Meldungen, war Streamende bei read() erreicht,
// sonst bei skip!
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 available() und ready() methods

- Mit diesen Methoden kann man abfragen, ob die gewünschte Anzahl von Bytes *(available())* oder ob Zeichen *(ready())* im *Stream* - ohne *blocking*) zur Verfügung stehen.

- Es ist also kein Ersatz dafür, abzufragen, wie lang eine Datei ist.

```
if ( streamIn.available() < 1024 )
    System.out.println( " no more full blocks available" );

if (readerIn.ready() != true)
    System.out.println( " no more characters available" )
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 close() method

- Es ist auch in Java guter Programmierstil, nicht mehr benötigte Ressourcen explizit freizugeben.

- Hierzu dient die *close()-method*,

```
InputStream streamIn = new ...
Reader readerIn = new ...
...
// Daten lesen
...
streamIn.close();
readerIn.close();
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.6 *InputStream* - Subklassen

◆ In den Subklassen der ***InputStream class*** bzw. der ***Reader class*** sind die "Eingabe" - Methoden implementiert, soweit es nicht schon in den Superklassen geschehen ist.

◆ Da die ***InputStream class*** bzw. die ***Reader class*** abstrakte Klassen sind, können sie nicht instantiiert werden.

◆ Subklassen der ***InputStream*** *class*:

- ***FileInputStream***
  Lesen von einer Datei im Dateisystem
- ***PipedInputStream***
  Implementierung einer Pipe - zusammen mit PipedOutputStream
- ***ByteArrayInputStream***
  Lesen von Bytefeldern im Speicher
- ***SequenceInputStream***
  Mehrere Eingabeströme zu einem vereinigen
- ***StringBufferInputStream***
  Lesen aus einem Stringbuffer

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.6 *InputStream* - Subklassen

◆ Subklassen der ***InputStream class*** *(cont)*:

- die abstrakte Klasse ***FilterInputStream*** mit den Subklassen
  - *BufferedInputStream*
  - *DataInputStream*
  - *LineNumberInputStream*
  - *PushBackInputStream*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.7 *Reader* - Subklasse

◆ Subklassen der ***InputReader*** *class*:

- ***FileReader***
  Lesen von einer Datei im Dateisystem
- ***PipedReader***
  Implementierung einer Pipe - zusammen mit ***PipedWriter***
- ***CharArrayReader***
  Lesen von Bytefeldern im Speicher
- ***BufferedReader***
  gepufferte Eingabe
- ***StringReader***
  Lesen aus einem ***Stringbuffer***
- die abstrakte Klasse ***FilterReader*** mit der Subklasse
  – *PushBackReader*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.8 *File class*

◆ Die Konstruktoren der ***FileInputStream class*** bzw. ***FileReader class*** akzeptieren Filenamen oder File-Objekte als Parameter.

◆ Unsere bisherigen Beispiele haben von der ersten Möglichkeit Gebrauch gemacht.

◆ In der Regel benötigt man jedoch “Zugang” zum Filesystem selbst, um z. B. abfragen zu können,

- ob eine Datei bereits existiert,
- welches die Zugriffsrechte sind, sie ggf. zu löschen oder umzubenennen, usf.

◆ Es ist also sinnvoll, zunächst ein Objekt der ***File class*** zu instantiieren und dieses Objekt als Parameter dem ***InputFileStream*** (*oder FileReader*) bzw. ***OutputFileStream*** *(oder FileWriter)* zu übergeben.

◆ In der “Docu” finden Sie die API-Spezifikation der *File class!*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.9 Beispiel: Lesen einer Datei

```
import java.io.*;

public class  FileioTest3 {

    public static void  main( String args[] ) {

/* Liest Bytes von der Datei "test.txt" bis zu einer Groesse von
  max 1024 Bytes und gibt die Anzahl tatsächlich gelesener Bytes
  aus. Längere Dateien verursachen IndexOutOfBoundsExceptions!
*/

    final int BLOCKSIZE = 1024;
    byte[] ByteBuffer = new byte[BLOCKSIZE];
    int i;
    int data;
    String path;

    File fd1 = new File( "test.txt" );
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.9 Beispiel: Lesen einer Datei

```
    if ( fd1.exists() ) {
        System.out.println( "The File exists already!" );
    }   else {
        System.out.println( "The File does't exist!" );
    }

    System.out.println( fd1.getPath() );

    if (fd1.canRead()) {
        System.out.println( "I can read the File" );
    } else {
        System.out.println( "I can't read the File" );
    }
    if ( fd1.canWrite() ) {
        System.out.println( "I can write the File" );
    } else {
        System.out.println( "I can't write the File" );
    }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.9 Beispiel: Lesen einer Datei

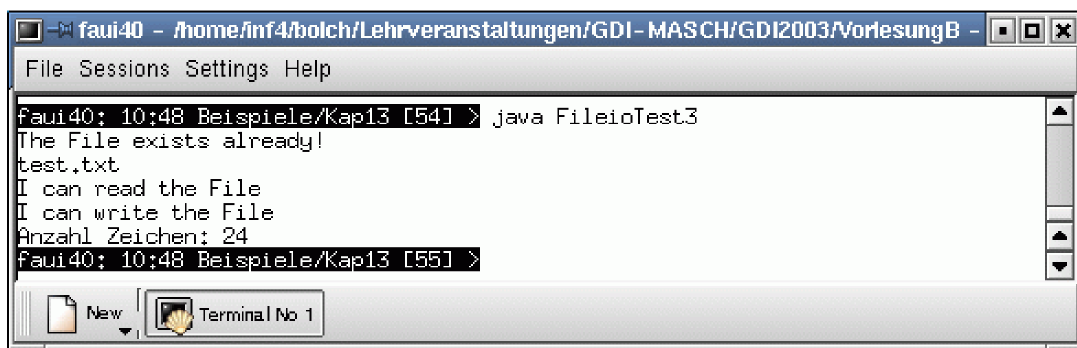
```
    i = 0;
    try {
      InputStream streamIn = new FileInputStream( fd1 );
      while ( ( data = streamIn.read() ) != -1 ) {
        byteBuffer[i] = (byte) data;
        i = i + 1;
      }
      System.out.println( "Anzahl Zeichen: "+i );

    }
    catch ( FileNotFoundException e ) {
        System.out.println( "FileioTest:" + e );
    }
    catch ( IOException e ) {
        System.out.println( "FileioTest:" + e );
    }
    }
  }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.9 Beispiel: Lesen einer Datei

- Ergebnis:
  - Datei: test.txt

    `dies ist eine Textdatei`



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.9 Beispiel: Lesen einer Datei

◆ Ergebnis ohne die Testdatei:

```
faui40 - /home/inf4/bolch/Lehrveranstaltungen/GDI-MASCH/GDI2003/VorlesungB -
File  Sessions  Settings  Help
faui40: 11:47 Beispiele/Kap13 [61] > mv test.txt test1.txt
faui40: 11:47 Beispiele/Kap13 [62] > java FileioTest3
The File does't exist!
test.txt
I can't read the File
I can't write the File
FileioTest:java.io.FileNotFoundException: test.txt (No such file or directory)
faui40: 11:47 Beispiele/Kap13 [63] >
New   Terminal No 1
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.9 *OutputStream class; Writer class*

◆ In den *abstrakten* Klassen ***OutputStream*** und ***Writer*** sind die zu ***InputStream*** und ***Reader*** “inversen” Methoden definiert, soweit sinnvoll:

- ***open()*** gibt es im klassischen Sinne nicht; ***open()*** ist das Erzeugen eines ***Stream***-Objekts also der ***new***-Operator.
- *write()*
- *flush()*
- *close()*

◆ Analog zur ***read() method*** der ***InputStream class*** bzw. der ***Reader class*** ist die ***write() method*** der ***OutputStream class*** und der ***Writer class*** blockierend.

◆ Da die Methoden in ***OutputStream*** und ***Writer*** funktional gleich sind, beschränken sich die folgenden Beispiele auf ***OutputStream***.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 write() method

◆ Analog zur ***read() method*** gibt es auch **drei** Varianten der ***write() method:***

- Ausgabe eines Puffers:

```
final int BLOCKSIZE = 1024;

OutputStream streamOut = new FileOutputStream
                            ( "pathname of the file" );

byte[] byteBuffer = new byte[BLOCKSIZE];
    ....
    fillInDataIntoTheBuffer( byteBuffer );
    streamOut.write( byteBuffer );
....
```

- oder Ausgabe von Teilen des Puffers:

```
    streamOut.write( byteBuffer, 100, 300 );
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 write() method

- oder als Einzelzeichenausgabe:

```
final int BLOCKSIZE = 1024;

OutputStream streamOut = new FileOutputStream
                            ( "pathname of the file" );

byte[] byteBuffer = new byte[blocksize];
...
fillInDataIntoTheBuffer( byteBuffer );

for (int i = 0; i < byteBuffer.length i++) {
    streamOut.write( byteBuffer[i] );
}
....
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 *flush() und close() methods*

- Mit dem Aufruf der ***flush() method*** erreicht man, dass eventuell noch im Systempuffern vorhandene Daten auf das eigentliche Ziel des Datenstromes hinausgeschrieben werden.

- Es ist auch in Java guter Programmierstil, nicht mehr benötigte Ressourcen explizit freizugeben.

- Hierzu dient die ***close()-method*** (siehe Abschnitt: *InputStream class; Reader class*)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.10 *OutputStream* - Subklassen

- In den Subklassen der *OutputStream class* bzw. der *Writer class* sind die “Ausgabe” - Methoden implementiert, soweit es nicht schon in den Superklassen geschehen ist.

- Da die *OutputStream class* bzw. die *Writer class* abstrakte Klassen sind, können sie nicht instantiiert werden.

- Die “eigentlichen” *Output-Streams* sind Subklassen der *OutputStream class* und *Writer class*.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.11 *OutputStream* - Subklassen

◆ Subklassen der *OutputStream class*:

- *FileOutputStream*
  Schreiben in eine Datei des Dateisystems.
- *PipedOutputStream*
  Implementierung einer Pipe - zusammen mit *PipedInputStream*
- *ByteArrayOutputStream*
  Schreiben in Bytefelder im Speicher.
- die abstrakte Klasse *FilterOutputStream* mit den Subklassen
  - *BufferedOutputStream*
  - *DataOutputStream*
  - *PrintStream*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.12 *Writer* - Subklassen

◆ Subklassen der *Writer class*:

- *OutputStreamWriter*
- *File Writer*
  Schreiben in eine Datei des Dateisystems.
- *PipedWriter*
  Implementierung einer Pipe - zusammen mit PipedReader
- *CharArrayWriter*
  Schreiben in Characterfelder im Speicher
- *BufferedWriter*
  gepufferte Ausgabe
- *StringWriter*
  Schreiben in einen *Stringbuffer*
- *PrintWriter*
- *FilterWriter*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.13 Formatierte Ein-/Ausgabe

❏ Bisher haben wir die gelesenen Daten entweder gar nicht *(Byte)* oder nur als *Characters* interpretiert. Meistens ist aber sehr wohl bekannt von welchem Typ die Daten in einem *Stream* (z. B. aus einer Datei) sind. Es ist also sinnvoll, sie gleich als **formatierte** Daten zu interpretieren.
Der Realisierung dienen die *DataInputStream class* und *DataOutputStream class* mit dem
- *DataInput Interface* und dem
- *DataOutput Interface*.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 DataInput Interface

```
void      readFully(byte[] bbuffer)
                              throws IOException
void      readFully(byte[] bbuffer, int offset, int length
                              throws IOException

int       skipBytes(int n)    throws IOException

boolean   readBoolean()       throws IOException
byte      readByte()          throws IOException
int       readUnsignedByte    throws IOException
short     readShort()         throws IOException
int       readUnsignedShort() throws IOException
char      readChar()          throws IOException
int       readInt()           throws IOException
long      readLong()          throws IOException
float     readFloat()         throws IOException
double    readDouble()        throws IOException

String    readLine()          throws IOException
String    readUTF()           throws IOException
```

❏ Die ersten zwei Methoden entsprechen den *read() methods* von *InputStream*, die dritte Methode der *skip() method*.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 DataOutput Interface

```
void write(int i)                       throws IOException;
void write(byte[] buffer)               throws IOException;
void write(byte[] buffer, int offset, int length)
                                        throws IOException;

void writeBoolean(boolean b)            throws IOException;
void writeByte(int i)                   throws IOException;
void writeShort(int i)                  throws IOException;
void writeChar(int i)                   throws IOException;
void writeInt(int i)                    throws IOException;
void writeLong(long l)                  throws IOException;
void writeFloat(float f)                throws IOException;
void writeDouble(double d)              throws IOException;

void writeBytes(String s)               throws IOException;
void writeChars(String s)               throws IOException;
void writeUTF(String s)                 throws IOException;
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Formatierte I/O - Ein Beispiel

```
import java.io.*;
public class DataIOTest {
    public static void main(String[] args) throws IOException{

// write the data out

    DataOutputStream out = new DataOutputStream(new
                              FileOutputStream("invoice1.txt"));
    double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
    int[] units = { 12, 8, 13, 29, 50 };
    String[] descs = { "Java T-shirt",
         "Java Mug",
         "Duke Juggling Dolls",
         "Java Pin",
         "Java Key Chain" };

        for (int i = 0; i < prices.length; i ++) {
            out.writeDouble(prices[i]);
            out.writeChar('\t');
            out.writeInt(units[i]);
            out.writeChar('\t');
            out.writeChars(descs[i]);
            out.writeChar('\n');
        }
    out.close();
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Formatierte I/O - Ein Beispiel

```
// read it in again

    DataInputStream in = new DataInputStream(new
    FileInputStream("invoice1.txt"));
    double price;
    int unit;
    StringBuffer desc;
    double total = 0.0;
    try {
      while (true) {
            price = in.readDouble();
            in.readChar();        // throws out the tab
            unit = in.readInt();
            in.readChar();        // throws out the tab
            char chr;
            desc = new StringBuffer(20);
            char lineSep =
            System.getProperty("line.separator").charAt(0);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Formatierte I/O - Ein Beispiel

```
            while ((chr = in.readChar()) != lineSep)
                desc.append(chr);
                System.out.println("You've ordered " +
                        unit + " units of " +
                        desc + " at $" + price);
                total = total + unit * price;
            }
        } catch (EOFException e) {
        }
        System.out.println("For a TOTAL of: $" + total);
        in.close();
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 PrintStream

```
public void   write(int byteOrChar); // byte (PrintStream), char (PrintWriter)
public void   write(byte[] buffer, int offset, int length)      // PrintStream
public void   write(char[] buffer, int offset, int length)      // PrintWriter
public void   write(String string);                             // PrintWriter
public void   write(String string, int offset, int length)      // PrintWriter
Sowohl in PrintWriter und in PrintStream:
public void   flush()
public void   close()

public void   print(Object o)
public void   print(String s)
public void   print(char[] buffer)
public void   print(char c)
public void   print(int i)
public void   print(long l)
public void   print(floatf)
public void   print(double d)
public void   print(boolean b)
public void   println(Object o)
public void   println(String s)
public void   println(char[] buffer);
public void   println(char c)
public void   println(int i)
public void   println(long l)
public void   println(float f)
public void   println(double d)
public void   println(boolean b)
public void   println()            // output a blank line
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.14 Object Input/Output

- ❏ Das *ObjectInput Interface* und das *ObjectOutput Interface* sind Erweiterungen des *DataInput Interfaces* bzw. *DataOutput Interfaces*.
  - ◆ Bisher haben wir nur mit primitiven Datentypen, *Arrays* und *Strings* gearbeitet, nicht jedoch mit komplexeren Datenstrukturen wie verketteten Listen, Bäumen, Warteschlangen, Stacks, etc.
    In objektorientierten Systemen sind dies, wie Sie bereits aus Kapitel 12 wissen, nicht nur “verzeigerte” Listen, sondern Objekte mit Referenzbeziehungen untereinander.
    Auch solche Strukturen möchte man persistent z. B. auf Dateien ablegen können, um irgendwann mit genau diesem Zustand in der Bearbeitung fortzufahren.
  - ◆ Seit Java 1.1 ist das Konzept der *Serialisation* in Java eingeführt.
    Für alle Klassen, die diese Eigenschaft besitzen, ist es möglich, Instanzen dieser Klasse in Streams zu bearbeiten, also auch auf Dateien abzuspeichern. Diesem Zweck dienen die
    - *ObjectInputStream class* und die
    - *ObjectOutputStream* class.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.15 Random Access Files

❏ Bisher haben wir beim Zugriff auf Dateien nur die sequentielle Methode des Zugriff auf Dateien nutzen können.
Häufig möchte man jedoch auch wahlfrei *(random)* zugreifen können, also genau auf eine bestimmte Position einer Datei zugreifen. Hierfür stellt Java die Klasse *RandomAccessFile* bereit.

- Konstruktoren

```
RandomAccessFile(“filename”, Zugriffsmodus)
```

wobei Zugriffsmodus “r”; “w” oder “rw” sein kann.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.15 Random Access Files

❏ Für die Positionierung steht das Konzept eines Dateizeigers zur Verfügung, eine Variable, die die Position in der Datei angibt:

- Nach dem Öffnen der Datei *(new)* ist der Zeiger auf den Anfang der Datei positioniert.
- Nach *write()* und *read()* steht der Zeiger vor dem nächsten noch nicht gelesenen bzw. nicht überschriebenen Datum.
- Mit *skipBytes(int)* wird der Zeiger um eine Anzahl Bytes nach vorne verschoben
- Mit *seek(int)* wird auf eine absolute Position in der Datei positioniert
- Mit *getFilePointer()* erhält man die aktuelle Position in der Datei als Returnparameter.
-

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.16Anhang: Lesen einer Datei "by Line"

◆ Beispiel:

```
import java.io.*;

public class  ReadFileByLine {

   final int MAX = 200;
   private String[] lines = new String[MAX];

   public void readLines( File fd ) {

   //  Liesst Zeilen (= Strings) von einer Datei "fd"
   //  in ein Strings - Feld"

      int i = 0;

      try {
          BufferedReader streamIn = new BufferedReader( new FileReader(fd)
);

          while( ( ( lines[i] = streamIn.readLine() ) != null )
                  && (i < MAX ) ) i++;
          System.out.println( "Anzahl Lines: " + i );
      }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.16 Anhang: Lesen einer Datei *"by Line"*

```
      catch ( FileNotFoundException e ) {
          System.out.println( "FileioTest:" + e );
      }
      catch ( IOException e ) {
          System.out.println( "FileioTest:" + e );
      }
   }
   void printStringBuffer(){

      int i = 0;

      while( ( lines[i] != null) && (i < MAX  ) ) {
        System.out.println( lines[i] );
        i++;
      }
   }

}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.16 Anhang: Lesen einer Datei *"by Line"*

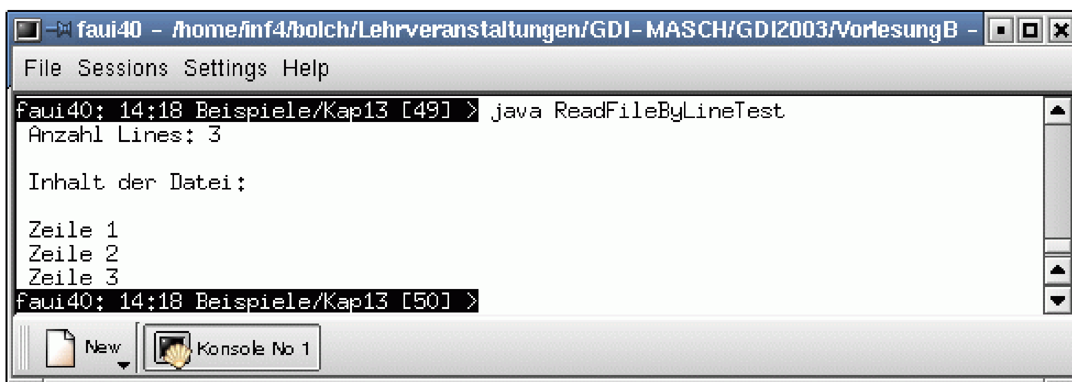
◆ Testprogramm:

```
import java.io.*;

class ReadFileByLineTest {

    public static void main( String args[] )
        throws java.io.IOException {

        File fd1 = new File( "test.txt" );
        ReadFileByLine eingabe = new ReadFileByLine();

        eingabe.readLines( fd1 );
        eingabe.printStringBuffer();
    }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

13.16 Anhang: Lesen einer Datei *"by Line"*

◆ Ergebnis:



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.