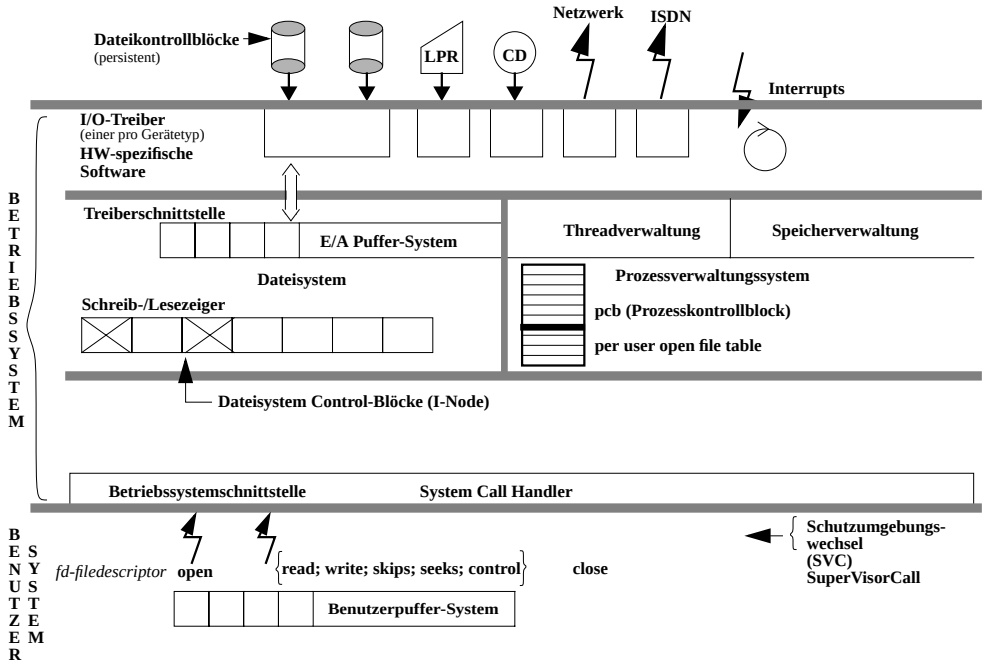
Grundlagen der Informatik für Ingenieure I

13 Java-E/A-System

13 Java-E/A-System

13.1 E/A-System - Überblick



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...)
  - 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 die Ausgabe z. B. 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.

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 Datenstrukturen 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.

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.

13.3 Java-I/O -System

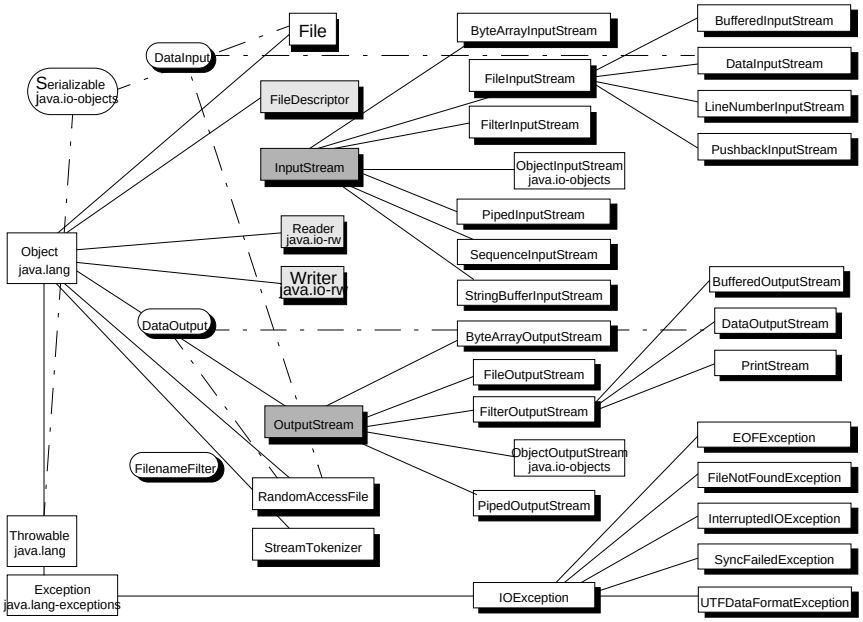
- Das Java-I/O-System ist ***Streams***-basiert.
  - *Streams* sind (vom Betriebssystem uninterpretierte) Datenströme von einer Datenquelle zu einer Datensenke.
    Man unterscheidet
    - *Inputstreams* und
    - *Outputstreams*.
  - Die physikalischen Ein-/Ausgabegeräte sind in das Unix Filesystem integriert.
  - Zusammen mit dem Streamskonzept 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.

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.
  - Im Rahmen dieser Vorlesung wird die Eingabe ausführlicher behandelt.
  - Die Ausgaben sind dann jeweils analog zu gestalten.
- Die **abstrakten** Basisklassen sind
  - *InputStream class,*
  - *Reader class,*
  - *OutputStreams class,*
  - *Writer class,*

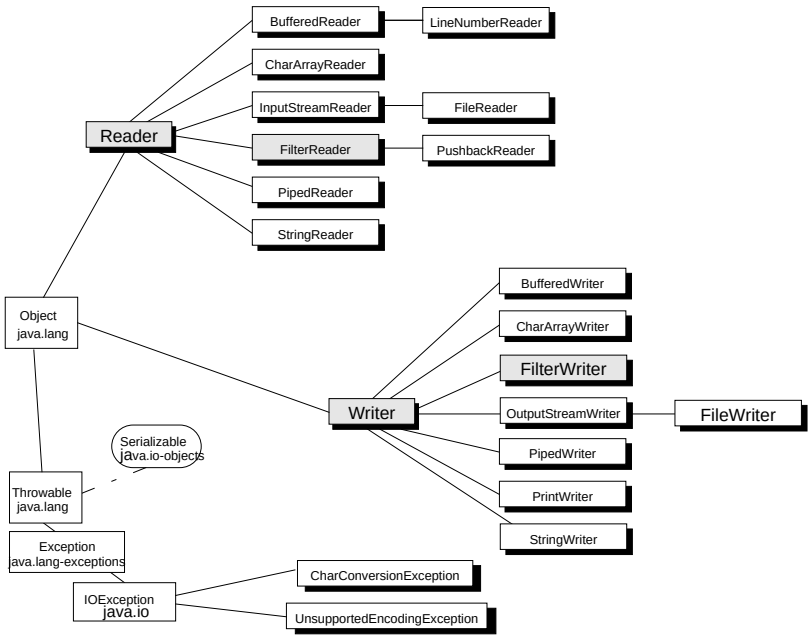
  wobei Lesen und Schreiben immer aus der Sicht des aufrufenden Programms zu sehen ist.

13.4 I/O-Klassenhierachie



13.4 I/O-Klassenhierachie

13.4 I/O-Klassenhierachie



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 **Characterstrom** 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 Erzeugen eines *Stream*-Objekts verborgen.
  - *read()*
  - *skip()*
  - *available() und ready(); mark und reset()*
  - *close()*

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 empfiehlt sich also für E/A-Vorgänge eigene *threads* vorzusehen.

1 read() method

- Beispiel: Blockweises Lesen

```
final int BLOCKSIZE = 1024;
FileInputStream 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 vorhandene Bytes gelesen.
  - Als Returnparameter wird die Anzahl der tatsächlich gelesenen Bytes zurückgegeben.

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.read( 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 );
```

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 gotless than expected” );
            }
}
System.out.println( “Skip: No more complete blocks” );

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


3 available() und ready() methods

- Mit diesen Methoden kann man abfragen, ob die gewünschte Anzahl von Bytes *(available())* oder ob Zeichen *(ready())* im *Stream* 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 charcters available” );
```

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();
```


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

13.6 *InputStream* - Subklassen

- Subklassen der *InputStream class(cont)*:
  - die abstrakte Klasse *FilterInputStream* mit den Subklassen
    - *BufferedInputStream*
    - *DataInputStream*
    - *LineNumberInputStream*
    - *PushBackInputStream*

13.7 *Reader* - Subklassen

- 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*

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!*

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" );
```

13.9 Beispiel: Lesen einer Datei

```
if (fd1.exists()){
    System.out.println( "The File exists already!" );
}   else {
    System.out.println( "The File doesn'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" );
}
```


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 );
}
}
}
```

13.9 Beispiel: Lesen einer Datei

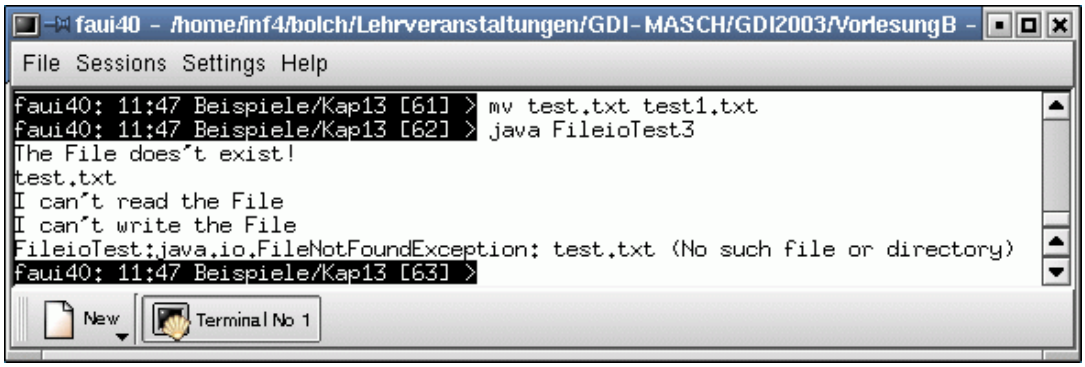
◆ Ergebnis:

- Datei: test.txt

```
dies ist eine textdatei
```

13.9 Beispiel: Lesen einer Datei

◆ Ergebnis ohne die testdatei:



13.10 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 *Streams*-Objekts also der *new*-Opereator.
  - *write()*
  - *flush()*
  - *close()*

- Analog zu der *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 *OutputStreams* und *Writer* funktional gleich sind, beschränken sich die folgenden Beispiele auf die *OutputStreams*.

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);
```

1 write() method

- oder als Einzelzeichenausgabe:

```
final int BLOCKSIZE = 1024;
OutputStream streamOut = new FileOutputStream
                              ( “pathname of the file” );

byte[] byteBuffer = new byte[BLOCKSIZE];
int i;
...
fillInDataIntoTheBuffer( byteBuffer );
i = 0;

while ( ( i != byteBuffer.length )
    streamOut.write( byteBuffer[i] );
    i = i + 1
}
....
```


2 flush() und close() methods

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

- Für die *close() method* gilt das im Abschnitt 13.5.4 gesagte.

13.11 *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*.

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*

13.12 *Writer* - Subklassen

- Subklassen der *Writer class*:
  - *OutputStreamWriter*
  - *FileWriter*
    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*

13.13 Anhang: 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 );
    }
```

13.13 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;

      System.out.println( "\n Inhalt der Datei: \n" );

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


13.13 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();
    }
}
```

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

- Ergebnis: