Hexdump

```
import java.io.*;

public class Hexdump
{

private final static int BREITE=16;

static public void main(String args[])
{
    File datei;    // Die Datei die ausgegeben werden soll
    int inByte;    // Gerade gelesenes Byte
    int pos=0;     // Position in der Zeile
    long count=0;  // Bisher ausgegebene Bytes
    // Buffer für die ASCII-Repräsentation
    StringBuffer ascii=new StringBuffer(BREITE+1);
    // +1 um sicherzugehen, daß der Buffer nicht vergößert wird
    if(args.length!=1)
    {
        System.err.println("Benutzung: java Hexdump Dateiname");
        System.exit(1);
    }
```

```
    // Datei prüfen
    datei=new File(args[0]);
    if(!datei.exists() || !datei.canRead() || !datei.isFile())
    {
        System.err.println("Benutzung: java Hexdump Dateiname");
        System.exit(1);
    }
    try
    {
        // Inputstream öffnen, hier byteweise lesen
        BufferedInputStream in=new BufferedInputStream(
                                                new FileInputStream(datei));
        System.out.println("Hexdump of "+datei.getCanonicalPath()+":\n");
        while((inByte=in.read())>=0)
        {
            if(pos==0)
            {   // Zeilenbeginn: Offset bis hierher ausgeben
                String out=Long.toHexString(count).toUpperCase();
                for(int i=0;i<6-out.length();i++)
                    System.out.print("0");
                System.out.print(out+": ");
            }
            // toHexString liefert keine führenden Nullen, daher
            // bei einstellig händisch erweitern
            if(inByte<16)
                System.out.print("0");
```

```
            // Bytewert in Hex ausgeben
            System.out.print(Long.toHexString(inByte).toUpperCase());
            // Weiterzählen
            pos++;
            count++;
            // Ascii-Repräsentation anhängen ('.' wenn nicht-druckbar)
            if(inByte>=32 && inByte<=127)
                ascii.append((char)inByte);
            else
                ascii.append(".");
            if(pos==BREITE)
            {   // Ende einer Zeile, Trennung und Ascii-Repr. ausgeben
                System.out.print("   ");
                System.out.println(ascii.toString());
                ascii=new StringBuffer(BREITE+1);
                pos=0;
            }
        }
    }
    catch(FileNotFoundException e)
    {   // Sollte eigentlich nicht auftreten:
        // AUSSER Datei ist bereits von anderem Programm geöffnet
        System.err.println("Fehler beim Öffnen der Datei: "+e.toString());
        System.exit(1);
    }
```

```
    catch(IOException e)
    {
        System.err.println("Fehler beim Lesen der Datei: "+e.toString());
        System.exit(1);
    }
    // Rest der letzten Zeile ausgeben (sofern vorhanden)
    if(pos>0)
    {
        while(pos<BREITE)
        {
            System.out.print("  ");
            pos++;
        }
        System.out.print("   ");
        System.out.println(ascii.toString());
    }
}

}
```

Testausdruck:

```
Hexdump of C:\Java\lva\Dateien\Hexdump\Hexdump.cdb:

000000: 534A20436F6D70696C6174696F6E2044   SJ Compilation D
000010: 422076312E30320D0A1A000104001C43   B v1.02........C
000020: 3A5C4A6176615C6C76615C4461746569   :\Java\lva\Datei
000030: 656E5C48657864756D705C0000010028   en\Hexdump\....(
000040: 433A5C4A6176615C6C76615C44617465   C:\Java\lva\Date
000050: 69656E5C48657864756D705C48657864   ien\Hexdump\Hexd
000060: 756D702E6A617661000001002943A5C   ump.java....)C:\
000070: 4A6176615C6C76615C4461746569656E   Java\lva\Dateien
000080: 5C48657864756D705C48657864756D70   \Hexdump\Hexdump
000090: 2E636C6173730000094C48657864756D   .class...LHexdum
0000A0: 703B00                             p;.
```

GrepReader

```
public class GrepReader extends BufferedReader
{
    String muster; // String, nach dem gesucht wird

    public GrepReader(Reader in, String muster)
    {
        super(in);
        this.muster=muster;
        // Warum nicht this.patter=new String(pattern); ?!?!!!!
    }
```

```
public String readLine() throws IOException
{
    String line;
    // Zeile einlesen, bis Dateiende oder Zeile mit Muster gefunden
    do
    {
        line=super.readLine();
    } while (line!=null && line.indexOf(muster)==-1);
    return line;
}

public static void main(String[] args)
{
    // Parameter prüfen
    if(args.length!=2)
    {
        System.err.println("Benutzung: java GrepReader
                            Dateiname Muster");
        System.exit(1);
    }
    File datei=new File(args[0]);
    if(!datei.exists() || !datei.canRead() || !datei.isFile())
    {
        System.err.println("Benutzung: java GrepReader Dateiname");
        System.exit(1);
    }
```

```
        // Neuen GrepReader erzeugen und entsprechende Reader übergeben
        try
        {
            GrepReader grep=new GrepReader(new FileReader(datei),args[1]);
            String line;
            while((line=grep.readLine())!=null)
                System.out.println(line);
            grep.close();
        }
        catch(FileNotFoundException e)
        {   // Sollte eigentlich nicht auftreten:
            // AUSSER Datei ist bereits von anderem Programm geöffnet
            System.err.println("Fehler beim Öffnen der Datei: "+
                               e.toString());
            System.exit(1);
        }
        catch(IOException e)
        {
            System.err.println("Fehler beim Lesen der Datei: "+
                              e.toString());
            System.exit(1);
        }
    }
}
```

LinenumberWriter

```
public class LinenumberWriter extends BufferedWriter
{
    private long currentLinenumber=1;   // Current number of line
    private int increment=1;    // Increment between two consecutive lines
    private boolean afterLF=true;   // If we are after a LF
    private String lineSeparator="\r\n";  // Separartor between two lines
    private int posInSep=0; // Next char in lineSeparator to be matched

public LinenumberWriter(Writer out)
{
    super(out);
}

public LinenumberWriter(Writer out,long startNumber,int increment)
{
    this(out);
    currentLinenumber=startNumber;
    this.increment=increment;
}
```

```
public void write(int c) throws IOException
{
    // Check first if immediately after a LF -> Write new linenumber
    if(afterLF)
    {
        afterLF=false;  // Must be BEFORE write, else endless recursion!
        write(""+currentLinenumber+": ");
    }
    super.write(c); // Write the character
    // This algorithm will only work, if no character appears twice
    // in lineSeparator! E.g.: Separator=RORX, Text=RORORX; At the
    // second O the posInSep will be set to 0, but it should be 2!
    if(c==lineSeparator.charAt(posInSep))
    {   // Check for separator
        posInSep++;
        if(posInSep==lineSeparator.length())
        {
            afterLF=true;
            currentLinenumber+=increment;
            posInSep=0;
        }
    }
    else    // Wrong character, reset to start
        posInSep=0;

}
```

```
public void write(char[] cbuf,int off,int len) throws IOException
{
    for(int i=off;i<cbuf.length && i<off+len;i++)
        write(cbuf[i]);
}

public void write(String s,int off,int len) throws IOException
{
    write(s.toCharArray(),off,len);
}

public long getCurrentLinenumber()
{
    return currentLinenumber;
}
```

```
static public void main(String args[])
{
    LinenumberWriter lw=new LinenumberWriter(
                                        new PrintWriter(System.out),10,10);
    try
    {
        lw.write('t');  lw.write('e');  lw.write('\n'); lw.write('s');
        lw.write('\r'); lw.write('t');  lw.write('\r'); lw.write('\n');
        lw.write('a');  lw.write('\r'); lw.write('\n');
        lw.write("String ohne LF ");
        lw.write("String mit\r\nLF ");
        lw.write("String mit LF am Ende\r\n");
        lw.write("Current linenumber: "+lw.getCurrentLinenumber()+"\r\n");
        lw.flush(); // Don't close, this would end the program!
        lw=new LinenumberWriter(new PrintWriter(System.out));
        lw.write("First line\r\nSecond line\r\nThird line\r\n
                  Fourth line\r\n");
        lw.close();
    }
    catch(IOException e)
    {
        System.err.println(e.toString());
    }
}
}
```

Testoutput:

```
10: te
s t
20: a
30: String ohne LF String mit
40: LF String mit LF am Ende
50: Current linenumber: 50

1: First line
2: Second line
3: Third line
4: Fourth line
```

Wann soll ein Filter ein In- wann ein OutputStream sein?

Beispiel: Man will codierte Daten von der Festplatte einlesen (z. B. Lesen einer komprimierten Datei = FilterInputStream), aber auch komprimierte Daten vom Hauptspeicher auf die Festplatte schreiben (z. B. eine vom Netzwerk empfangene komprimierte Datei entpackt speichern =FilterOutputStream).

- Einfachste Lösung: Keinen Stream produzieren, sondern eine Klassen, die ein Bytearray in ein Bytearray decodiert. Diese kann dann universell eingesetzt werden. Nachteil: Sehr unschön und unpraktisch.
- Einfache Lösung: Zwei Stream produzieren, sowohl einen DecodeInputStream als auch einen DecodeOutputStream. Nachteil: Doppelte Arbeit, Synchronisationsprobleme bei Ausbesserungen in einer Klasse, unschön.
- Elegante Lösung: Sich für einen Stream entscheiden (was am öftesten benötigt wird) und den anderen über einen universell einsetzbaren ReverseStream damit emulieren (ein beliebiger InputStream wird dadurch umhüllt und stellt die selbe Funktionalität als OutputStream zur Verfügung). Nachteil: ReverseStream ist etwas kompliziert zu programmieren (aber recht kurz!), dafür aber für ALLE Streams einsetzbar.

Richtlinie:

Stream die dekodieren, sind meistens eine Subklasse von InputStream.
Streams die kodieren sind meistens eine Subklasse von OutputStream

Base64InputStream

```
public class Base64InputStream extends FilterInputStream
{
    static private byte pad=61;//'=';
    static private int BADBYTE=-1;
    static private int b[] = new int[256];
    static
    {
        for(int i=0;i<256;i++)
            b[i]=BADBYTE;
        b['A'] = 0;  b['B'] = 1;  b['C'] = 2;  b['D'] = 3;  b['E'] = 4;
.....
        b['8'] = 60; b['9'] = 61; b['+'] = 62; b['/'] = 63;
        // The pad byte doesn't have an encoding mapping, but
        // it's not an automatic error.
        b[pad]=-2;
    }
    /* Buffer for decoded bytes that haven't been read */
    int buf[]=new int[3];
    int buffered=0;
    /* Buffer for clusters of encoded bytes */
    byte ebuf[]=new byte[4];
    boolean textfile;
```

```
public Base64InputStream(InputStream in)
{
    this(in, false);
}

public Base64InputStream(InputStream in,boolean textfile)
{
    // To make life easier, we slip a WSStripInputStream in just ahead
    // of us, so that we don't have to worry about whitespace bytes.
    super(new WSStripInputStream(in));
    this.textfile=textfile;
}

public synchronized int read() throws IOException
{
    if (buffered==0)
    {
        fill_buffer();
    }
    int b=buf[--buffered];
    if(textfile && b=='\r' && peek()=='\n')
    {   // Skip \r in textfiles
        b=read();
    }
    return b;
}
```

```
public synchronized int peek() throws IOException
{
    if(buffered==0)
    {
        fill_buffer();
    }
    return buf[buffered-1];
}

public int read(byte b[],int off,int len) throws IOException
{
    int i=off; int c=0;
    while(i<(off+len) && c>=0)
    {
        c=read();
        if(c>=0)
            b[i++]=(byte)c;
    }
    return i-off;
}
```

```
public long skip(long n) throws IOException
{
    // Can't just read n bytes from 'in' and throw them away, because
    // n bytes from 'in' will result in roughly (4n/3) bytes from
    // 'this', and we can't even calculate the exact number easily,
    // because of the potential of running into the padding at the
    // end of the encoding.  It's  easier to just read from 'this'
    // and throw those bytes away, even though it's less efficient.
    int i=0; int c=0;
    while(i<n && c>=0)
    {
        c=read();
        if(c>=0)
            i++;
    }
    return i;
}

protected void fill_buffer() throws IOException
{
    if(buffered!=0)
    {  // Just for safety ...
        return;
    }
    int l=in.read(ebuf);    // Fill in input buffer
    int numbytes=3;         // Number of decoded bytes
```

```
        if(l==0)
        {   // Must've reached EOF last time ...
            // Fill buffer with EOF indicators for read() to return.
            for(int i=0;i<buf.length;i++)
            {
                buf[i]=-1;
                buffered++;
            }
            return;
        }
        if(l<ebuf.length)
        {
            throw new EOFException();
        }
        // Check for bad bytes
        for(int i=0;i<ebuf.length;i++)
        {
            if(b[ebuf[i]]==BADBYTE)
            {
                throw new BadFormatException("Base64: invalid byte "+
                                             (char)ebuf[i]);
            }
```

```
            // While we're at it, take notice of padding
            if(ebuf[i]==pad)
            {
                if(i<2)
                {
                    throw new BadFormatException("Base64: padding
                                                  starts too soon");
                }
                numbytes=i-1;
            }
        }
        // Now do the decoding
        // Will only work for 3 bytes at most!
        int shiftUpper=4;
        int shiftLower=2;
        for(int i=0;i<numbytes;i++)
        {
            buf[(numbytes-1)-i]=
               ((b[ebuf[i]]<<shiftLower)&0xff)|(b[ebuf[i+1]]>>shiftUpper);
            buffered++;
            shiftUpper-=2;
            shiftLower+=2;
        }
    }
}
```

BufferedReader

```
public class BufferTest {

static public void main(String args[]) {
    int argIndex=2;
    boolean bufferInput=false;  // Buffer input?
    boolean bufferOutput=false; // Buffer output?
    // Read parameters
    if(args.length<2) {
        System.out.println("Too few parameters.\nUsage: java BufferTest
                           sourcefile destinationfile [-i] [-o]");
        System.exit(0);
    }
    while(argIndex<args.length) {
        if(args[argIndex].equalsIgnoreCase("-i"))
            bufferInput=true;
        else if(args[argIndex].equalsIgnoreCase("-o"))
            bufferOutput=true;
        else {
            System.out.println("Unknown argument: "+args[argIndex]+
                               ".\nUsage: java BufferTest sourcefile
                                destinationfile [-i] [-o]");
            System.exit(0);
        }
        argIndex++;
    }
```

```
File src=new File(args[0]);
if(!src.exists() || !src.isFile() || !src.canRead())
{
    System.out.println("Source file doesn't exist or cannot be read.\n
       Usage: java BufferTest sourcefile destinationfile [-i] [-o]");
    System.exit(0);
}
File dest=new File(args[1]);
if(dest.exists())
{   // File exists -> check if it can be written to
    if(!dest.isFile() || !dest.canWrite())
    {
        System.out.println("Destination file cannot be written.\n
        Usage: java BufferTest sourcefile destinationfile [-i] [-o]");
        System.exit(0);
    }
}
else
{   // Check if parent directory can be written to
    String parent=dest.getParent();
    // If no parent, look in current directory
    if(parent==null)
        parent=System.getProperty("user.dir");
    File dir=new File(parent);
```

```
        if(!dir.exists() || !dir.isDirectory() || !dir.canWrite())
        {
            System.out.println("Cannot write to destination directory.\n
            Usage: java BufferTest sourcefile destinationfile [-i] [-o]");
            System.exit(0);
        }
    }
    InputStream in=null;
    OutputStream out=null;
    try
    {
        // Open streams (with or without buffer)
        InputStream fSrc=new FileInputStream(src);
        OutputStream fDest=new FileOutputStream(dest);
        if(bufferInput)
            in=new BufferedInputStream(fSrc);
        else
            in=fSrc;
        if(bufferOutput)
            out=new BufferedOutputStream(fDest);
        else
            out=fDest;
        int buffer;
```

```
        // Copy single byte only till EOF
        long start=System.currentTimeMillis();  // Start timming
        while((buffer=in.read())!=-1)
            out.write(buffer);
        long end=System.currentTimeMillis();   // End timing
        System.out.println("Elapsed time in seconds: "+(end-start)/1000);
    }
    catch(IOException e)
    {
        System.out.println("Error during copying: "+e.toString());
    }
    finally
    {   // Close in any case
        try
        {
            if(in!=null)
                in.close();
            if(out!=null)
                out.close();
        }
        catch(IOException e) {   // Nothing can be done here }
    }

}

}
```

Ergebnisse:

Ohne Buffer:	183 s
Input-Buffer:	120 s
Output-Buffer:	101 s
Input- und Output-Buffer:	40 s

Wann buffert man?

Buffern sollte man überall dort, wo Lese- oder Schreiboperationen kostenintensiv sind: Fetplatte, Netzwerk, …
Man sollte es jedoch damit nicht übertreiben:

- Buffert man sehr viel, leidet die Performance, weil viele unnötige Operationen durchgeführt werden
- Die Bufferung sollte eher dem Benutzer überlassen werden. Er kann über eine Verkettung von Streams jederzeit zusätzliche Buffer einfügen; ein fix einprogrammierter Buffer kann jedoch nie entfernt werden. Dies ist insbesondere bei Filtern problematisch: Vielleicht befindet sich vor uns ein LinenumberInputReader, der bei extensiver Bufferung dann andere Zeilennummern anzeigt, als vom nächsten Stream in der Kette gerade wirklich zurückgeliefert werden (z. B. bei zusätzlichen Fehlerinformationen).

Richtlinie:

Buffern sie immer soviel, wie Sie für die augenblickliche Operation gerade brauchen.
Beispiel Base64InputStream:

- Bei byteweisem Lesen jeweils 4 Inputbytes buffern, die 3 Outputbytes ergeben
- Wird jedoch ein größeres Array eingelesen (read(byte[],off,len)), so sollte vorher ausgerechnet werden, wieviele Bytes man benötigt und diese dann gleich auf einmal eingelesen werden.

Spezielles zur Serialisierung von Objekten

- Nicht-serialisierbare Superklassen müssen einen Parameterlosen und zugreifbaren Konstruktor zur Verfügung stellen, ansonsten können Subklassen NICHT serialisiert werden.
- Das Schlüsselwort *transient*: Als transient markierte Variablen werden bei der Serialisierung ausgelassen. Dies bedeutet, daß sie selbst angelegt werden müssen (oder mit spezifischen readObject/writeObject-Methoden zu speichern sind).
- Das Schlüsselwort *static*: Statische Variablen werden nicht serialisiert. Sie sollten aber auch in read-/writeObject nicht serialisiert werden, da sie sonst für jedes Objekt einmal im OutputStream gespeichert werden: u. U. eine Vervielfachung der Datenmenge. Sie sind mit extra-Methoden nur einmal zu speichern. Ihr Einsatz in serialisierbaren Klassen sollte daher SEHR gut überlegt werden!
- Bei der De-Serialisierung wird von der (serialiserbaren, sonst siehe oben) Klasse KEIN Konstruktor aufgerufen und auch KEINE Variableninitialisierungen (bei der Deklaration angegeben) vorgenommen. Dies ist wichtig bei transient oder static Variablen sowie bei händischer Serialisierung (Bsp.: Siehe Variable inventar im Beispiel "Person").
- Bei der Serialisierung brauchen Sie sich keine Gedanken darüber zu machen, ob Sie Objekte mehrfach speichern: Es wird automatisch nur einmal gespeichert und bei späteren Verwendungen nur mehr eine Referenz auf diesen Speicherort abgelegt. Dies ist aber kein Freibrief für völlig gedankenlose Speicherung!
- Bei der Serialisierung werden ALLE von dem zu serialisierenden Objekt aus erreichbaren Objekte mit-serialisiert. Dies kann zu Problemen führen, wenn irgendwo eine nicht-serialisierbare Klasse enthalten ist, oder z. B. bei GUI-Anwendungen: Fenster sind zwar serialisierbar, doch macht dies nur selten Sinn und sie benötigen auch sehr viel Platz. In diesem Fall sollten sie als transient markiert werden.