Java I/O

- Input / Output stream library
- Regelt I/O über verschiedene Kanäle
  - Filesystem
  - Console
  - Netzwerk
  - Intern
- Führt notwendige Umwandlungen/Konvertierungen aus

Furchtbar viele Klassen

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter
ByteArrayInputStream
ByteArrayOutputStream
CharArrayReader
CharArrayWriter
DataInputStream
DataOutputStream
File
FileDescriptor
FileInputStream
FileOutputStream
FilePermission
FileReader
FileWriter
FilterInputStream
FilterOutputStream
FilterReader
FilterWriter
InputStream
InputStreamReader
LineNumberInputStream
LineNumberReader
ObjectInputStream
ObjectInputStream.GetField
ObjectOutputStream
ObjectOutputStream.PutField
ObjectStreamClass
ObjectStreamField
OutputStream
OutputStreamWriter
PipedInputStream
PipedOutputStream
PipedReader
PipedWriter
PrintStream
PrintWriter
PushbackInputStream
PushbackReader
RandomAccessFile
Reader
SequenceInputStream
SerializablePermission
StreamTokenizer
StringBufferInputStream
StringReader
StringWriter
Writer

Hierarchie der Klassen von java.io

1. zwei abstrakte Klassen
    1. InputStream
    2. OutputStream
2. Streams, die Bytes transportieren, z.B.:
    1. FileInputStream / FileOutputStream
    2. ByteArrayInputStream / ByteArrayOutputStream
3. FilterStreams, Streams mit Verarbeitung Konvertierungen, Serialisierung/Deserialisierung von Objekten, Pufferung z.B.:
    1. DataInputStream
    2. PrintStream
    3. BufferedInputStream

Die Klassen InputStream/OutputStream

- Diese Klassen stellen die elementare Funktionalität bereit.
- Quelle/ Senke der Informationsübertragung sind noch völlig offen.
- Sie finden immer Anwendung, wenn verschiedene Datenkanäle genutzt werden sollen.
- Hinter einem InputStream kann sich ein FileInputStream, NetzwerkStream oder ...InputStream verbergen.

InputSteam – die wichtigsten Funktionen

int	**available()** **Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.**
void	**close()** **Closes this input stream and releases any system resources associated with the stream.**
abstract int	**read()** **Reads the next byte of data from the input stream.**
int	**read(byte[] b)** **Reads some number of bytes from the input stream and stores them into the buffer array b.**
int	**read(byte[] b, int off, int len)** **Reads up to len bytes of data from the input stream into an array of bytes.**

- InputStream und OutputStream sind abstrakte Klassen.
- Sie lassen sich nicht instanzieren.
- Oft dienen Sie der Definition einer Referenzvariablen oder als Returntyp einer Funktion, die dann auf ein Objekt eines bytestransportierenden Streams verweist.
- So können die Quelle/Senke der Daten, ob die Daten vom lokalen Filesystem oder über das Netzwerk kommen, noch offen bleiben.

Bytestreams

- Sie sind abgeleitet von InputStream bzw. OutputStream.
- Diese Streams transportieren lediglich Bytes auf unterschiedlichen Kommunikationswegen.
- Beispielsweise liest ein FileInputStream Bytes aus einer Datei im lokalen Filesystem.
- Alle Bytestreams stellen die Funktionalität von InputStream bzw. OutputStream zur Verfügung.

Beispiele für Bytestreams

Input	Output
ByteArrayInputStream	ByteArrayOutputStream
FileInputStream	FileOutputStream
PipedInputStream	PipedOutputStrteam
StringBufferInputStream	
Streams über URL/URLConnection	
Streams über Sockets	

Die Klasse File

- Klasse zur Ermittlung der Eigenschaften von Dateien
- Diese Klasse dient nicht dem Datentransport und ergänzt die Klassen FileInput-/FileOutputStream
  - long length();
  - boolean isDirectory();
  - boolean isFile();
  - boolean canRead();
  - String[] listFiles();
  - String[] list();

Beispiel File und FileInputStream

```
import java.io.*;

class ListIt
{
  public static void main (String [] args)
                                  throws Exception
  {
    File f=new File(args[0]);

    if (!f.exists()|| !f.canRead())
    {
      System.out.println("Can't read "+f);
      return;
    }
    if (f.isDirectory())
    {
      String[] files=f.list();
      for(int i=0;i<files.length;i++)
        System.out.println(files[i]);
    }
    else
    try
    {
      FileInputStream fis=new FileInputStream(f);
      byte[] data=new byte[(int)f.length()];
      fis.read(data);
      System.out.write(data);
    }
    catch (FileNotFoundException e)
      {System.out.println("File Disappeared");}
  }
}
```

Übernehmen Sie den Quelltext und übergeben Sie beim Aufruf einen Dateinnamen oder einen Verzeichnisnamen.
Beipiele:

```
java ListIt ListIt.java
Java ListIt ..
```

Lesen von Daten bekannter Länge

```
FileInputStream fis=new FileInputStream(f);

int len=(int)f.length();

System.out.println("Length: "+len);
byte[] data=new byte[len];

// Lesen der Daten bei (sehr) kleinen Dateien
// mit einer read-Operation
// fis.read(data,0,(int)f.length());

int lenr=0;
// Stream lesen, wenn die Laenge bekannt ist
while (lenr<len) {lenr+=fis.read(data,lenr,len);

System.out.println(new String(data,0,len));}
```

Anmerkung:
Das Lesen mit nur einer Read-Operation geht nur bei kleinen Datenbeständen Und schneller Daten-übertragung

Der allgemeine Fall wird mit einer Schleife, wie abgebildet realisiert.
Bauen Sie das in ListIt ein.

Erläuterungen dazu

- Kleine Dateien können mit einem einzigen Aufruf von read gelesen werden.
- Bei größeren Dateien oder langsamen Datenströmen liest read so viel Daten, wie gerade verfügbar sind, das können auch 0 Bytes sein.
- Die Anzahl gelesener Bytes wird als Returnwert zurückgegeben (-1 bei EOF)
- Mit Hilfe dieses Wertes (im Beispiel lenr) wird ein offset zum Befüllen des Speicherbereiches berechnet.
- Auf diese Weise wird das Bytearray sukzessive mit den Daten gefüllt.

ByteArrayOutputStream

```
HexDump(InputStream fis)
{
 try
 {
   // Lesen eines endlichen Streams unbekannter Laenge
   // Lesen der Daten in Datenpuffer (buf)
   // Schreiben der Daten in einen ByteArrayOutputStream,
   // der mit den ankommenden Daten waechst

   ByteArrayOutputStream bos=new ByteArrayOutputStream(1024);
   byte buf[]=new byte[1024];
   int lenr;

   while ((lenr=fis.read(buf))>-1) bos.write(buf,0,lenr);
   // data ist ein Bytearray als Instanzvariable
   data=bos.toByteArray();
 }catch(Exception e){System.out.println(e);}
}
```

Initiale Größe 1024,
Wird automatisch vergrößert

Hier wird aus dem Stream ein Array für die weitere Verarbeitung

Eräuterungen dazu

- Insbesondere bei Streams über das Netz ist die Datenlänge oft unbekannt.
- Als Ziel der Übertragung wird etwas benötigt, das mit den ankommenden Daten (unendlich) wächst.
- Das Konzept dafür sind Streams.
- Der ByteArrayOutputStream ist ein gewöhnlicher Stream, bei dem das Medium zur Aufnahme der Daten ein Bytearray im Ram ist, das sukzessive bedarfsgerecht vergrößert wird.
- Stellen Sie sich einen Luftballon vor, der sich mit der Menge der eingeblasenen Luft vergrößert.
- Nach dem Ende der Datenübertragung kann der ByteArrayOutputstream in ein ByteArray (hier data) umgewandelt werden, um die Daten zu verarbeiten.

FilterStreams

- Die FilterStreams bilden die dritte Schicht der IO-Klasslibrary.
- Sie erhalten bei ihrer Instanzierung grundsätzlich einen anderen Stream, oft einen Bytes transportierenden Stream als Instanzierungsparameter.
- Sie führen eine Vorverarbeitung, bzw. Aufbereitung der zu transportierenden Daten durch.
- Typische Aufgaben bestehen in Konvertierungen oder im Packen/Entpacken von Daten. (b.Bsp.:zip)

FilterStreams

Filterstreams	
DataInputStream	DataOutputStream
BufferedInputStream	BufferedOutputStream
ObjectInputStream	ObjectOutputStream
GZIPInputStream	GZIPOutputStream
ZipInputStream	ZipOutputStream
	PrintStream

Die Klasse DataInputStream

- Gehört zur Gruppe der FilterStreams.
- Liest Daten der primitiven Datentypen aus einem Stream.
- Typische Funktionen sind readInt, readLong, readDouble usw.

```
int readInt();
```

- readInt liest 4 Bytes und gibt diese als int-Wert zurück
- Der Eingabestrom muss natürlich entsprechene Bytefolgen enthalten.

Suchen Sie die Klasse **`java.io.DataInputsStream`** in der Java API-Dokumentation (https://docs.oracle.com/javase/8/docs/api/) und studieren Sie diese.

DataOutputStream

```
class DataIOStream
{
 public static void main(String args[])throws Exception
 {
  FileOutputStream fout=new FileOutputStream("d.bin");
  DataOutputStream dout=new DataOutputStream(fout);
  for (String s:args)
    dout.writeLong(Long.parseLong(s));
  dout.close();
 }
}
```

Zeichenketten aus der Kommandozeile werden nach long konvertiert und in die Datei geschrieben (8 Bytes pro Zahl, big endian)

```
-> java DataIOStream 12 48 65535 7
-> hexdump -C data.bin
00000000 00 00 00 00 00 00 00 0c  00 00 00 00 00 00 00 30
00000010 00 00 00 00 00 00 ff ff  00 00 00 00 00 00 00 07
```


Java speichert in der Byteorder big endian!! - x86: Little endian

```
class DataIOStream
{
  public static void main(String args[])throws Exception
  {
    if (args.length==0)
        {System.err.println("usage: java DataIOStream <num> <num> ...");
         System.exit(-1)}
    FileOutputStream fout=new FileOutputStream("data.bin");
    DataOutputStream dout=new DataOutputStream(fout);
    for (String s:args)
      dout.writeLong(Long.parseLong(s));
    dout.close();
    FileInputStream fis=new FileInputStream("data.bin");
    DataInputStream di=new DataInputStream(fis);
    long l;
    try
    {
       while(true)
       {
          l=di.readLong();
          System.out.println(l);
       }
    }
    catch (EOFException e)
       {System.out.println("EOF reached");}
  }
}
```

Ergänzen Sie den Quelltext um notwendige import-Zeilen
Compilieren Sie das Programm, führen Sie es aus (Kommandozeilenargumente nicht vergessen!).
Betrachten Sie die entstandene Datei mit einem Hexdumpviewer.

Reader und Writer

- Diese Klassen dienen der Ein-/Ausgabe von Text
- Berücksichtigen in besonderer Weise Internationalisierung
- Zeichen werden in Java in 2 Bytes (Unicode) verarbeitet.
- Reader und Writer sind wieder, wie Input-/OutputStream abstrakte Klassen, bei denen Quelle und Ziel noch offen ist.
- Sie verarbeiten chars anstelle von bytes

Reader und Writer

Reader	Writer
CharArrayReader	CharArrayWriter
InputStreamReader	OutputStreamWriter
PipedReader	PipedWriter
BufferedReader	BufferedWriter
FileReader	FileWriter
	PrintWriter
LineNumerReader	

Reader in ListIt

```
String ret=““;
try
{
//  FileInputStream fis=new FileInputStream(f);
//  InputStreamReader r=new InputStreamReader(fis);
  FileReader         r=new FileReader(f);
  char[] data=new char[(int)f.length()];
   // Lesen kleiner Dateien
  r.read(data,0,(int)f.length());
  ret=new String(data);
}
catch (Exception e)
{
  System.out.println("File Disappeared");
}
```

BufferedReader

Buffered Reader ist die beste Wahl, Text zeilenweise zu verarbeiten.

int	**read()** **Read a single character.**
int	**read(char[] cbuf, int off, int len)** **Read characters into a portion of an array.**
String	**readLine()** **Read a line of text.**

Quelltext in ListIt.java sinnvoll einsetzen, Compilieren und ausprobieren.

```
try
{
  FileInputStream fis=new FileInputStream(f);
  InputStreamReader isr=new InputStreamReader(fis);
  BufferedReader br=new BufferedReader(isr);

  String line;
  while ((line=br.readLine())!=null)
    System.out.println(line);
}
catch (FileNotFoundException e)
  {System.out.println("File Disappeared");}
```

Exceptionhandling ist bei IO-Operationen in der Regel unumgänglich.
Die Erzeugung des des BufferedReaders wird oft ineinander geschachtelt:

```
BufferedReader br=new BufferedReader(new InputStreamReader(new FileInputStream(f)));
```

Bei Verwendung eines FileReaders gelingt das Ganze etwas kürzer:

```
BufferedReader br=new BufferedReader(new FileReader(f));
```