java.io

Ziel

- Verstehen der unterschiedlichen I / O Möglichkeiten
- Anwenden der Java I/ O Klassen

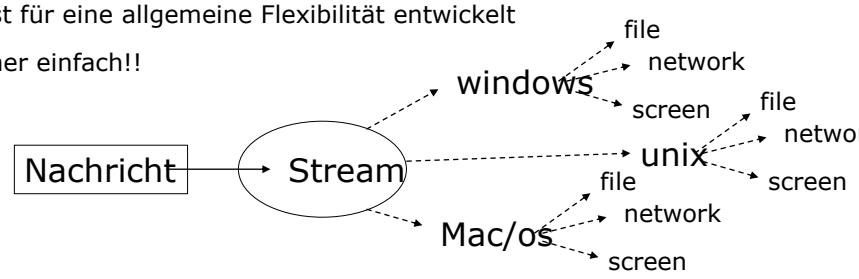
Ressourcen

- Java Tutorial
- Java API Dokumentation

Java API Prinzip

Java IO-API ist für eine allgemeine Flexibilität entwickelt

--> nicht immer einfach!!



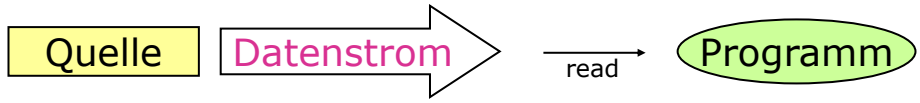
flexibel

- unterschiedliche Ein-/ Ausgabequellen
  - Files, Konsole, Netzwerkverbindungen, etc....
- unterschiedliche Kodierung
  - binär, Zeichen (Unicode), ...
- unterschiedlicher Zugriff
  - wortweise, zeilenweise, sequenziell, 'random-access', gepuffert ,...

streams als allgm. Konzept

Ein *stream* ist ein Objekt zum Transfer von Daten (Information).

Input stream: Transfer der Daten von der Quelle in das Programm.



Output stream: Transfer der Daten vom Programm zur Ziel.

Exceptions

I/ O erzeugt viele Ausnahmen (java.io.IOException):

- device error,
- file doesn't exist,
- file is protected,
- end of file.

IOExceptions: sind nicht Teil der RuntimeExceptions und müssen deshalb immer behandelt werden.
--> Compilerbeschwerde!
Manche Fehler sind schwer zu testen (device error).

OutputStreams

Ein **OutputStream** ist ein Objekt welches weiss, wie Bytes an einen Ziel-**OutputStream** zu senden sind.

FileOutputStream: Ein **FileOutputStream** ist ein Objekt welches weiss, wie Bytes in ein File geschrieben werden.

Methode in **OutputStream** :

void write(int b) throws IOException

Erzeugen eines **FileOutputStream** Objekts.

new FileOutputStream(String name)

new FileOutputStream(String name, boolean append)

Beide können throw **FileNotFoundException** erzeugen

Ein einfaches Beispiel (byte-stream)

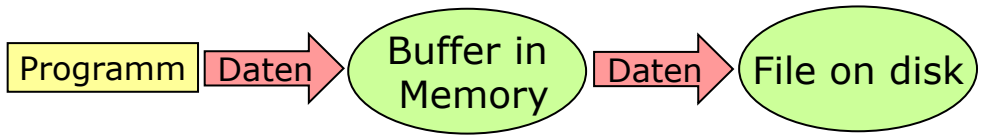
Schreiben einer kurzen Nachricht in ein File!

```
OutputStream outStream = null;      // keep compiler happy
try {
        outStream = new FileOutputStream(" OutputTest. txt");
}
catch (IOException e) {
        System. out. println(" Error in opening output file");
        System. exit( 1);
}                                   // end try/ catch
                                    // write to the output stream
try {                               // write individual bytes
        outStream. write( 'h'); outStream. write( 'e');
        outStream. write( 'l');  outStream. write( 'l');
        outStream. write( 'o');
}
catch (IOException e) {
        System. out. println(" Error in writing to output file");
        System. exit( 1);
}                                   // end try/ catch!
```

IO_1.java

Output Buffering

Beim Ausführen des Programms (letzte Folie) ist nicht sicher, dass das File alle Daten enthält – typischer Fehler!!
Grund: output buffering



Beim Schreiben eines Files muss das File mit ***close*** geschlossen werden.

Dies leert den Puffer und gibt die Systemressourcen wieder frei.

OutputStream Methode:

void close() throws IOException

Einfaches Beispiel verbessert

```
try {
        outStream. close();
}
catch (IOException e) {
        System. out. println(" error in closing output file");
}                                       // end try/ catch
```

Das File OutputTest.txt existiert nun und enthält immer die Zeile:

'hello'

Wie kann ich nun Text in eine Datei schreiben ?

Text Dateien

Ein **FileOutputStream** schreibt einzelne Bytes.
Sehr flexibel: Mit Bytes kann man alles darstellen.
Nicht sehr angenehm um Text zu handhaben!

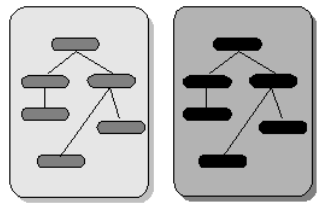
Textdateien: enthalten ASCII (oder Unicode) Zeichen.
Konzeptuell in Zeilen separiert. " newline characters (**'\ n'**)"
Beispiele: **.txt** , **.java** , **.html .**
Betriebssysteme haben spezifische Konventionen bzgl. Textdateien

Binärdateien: enthalten Daten binär kodiert.
Beispiele: **.class, .exe, .pdf, .doc**
(werden später erläutert).

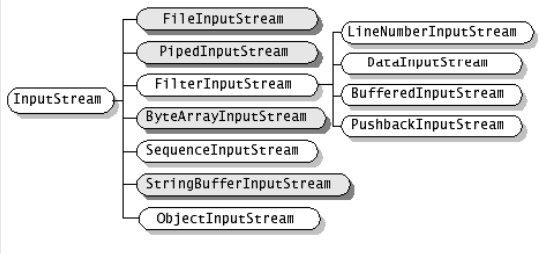
Character & Byte Streams

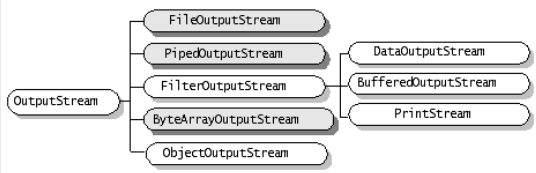
Die *stream*-Oberklasse erzeugt zwei vom Aufbau sehr ähnliche Klassenhierarchien!

Character Streams und Byte Streams

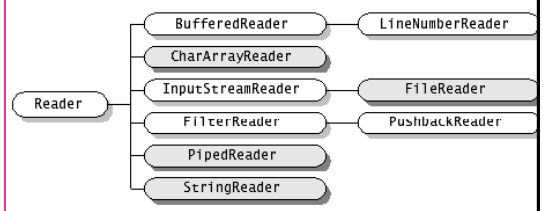


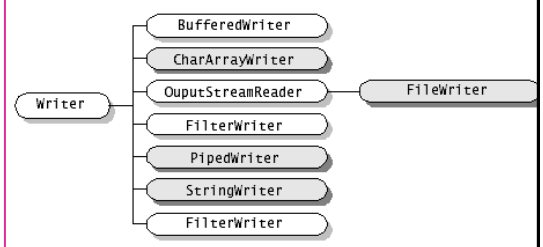
Byte Streams





Character Streams





Writer

Ein "Writer" schreibt Zeichen (char) in ein Textfile.

public class **PrintWriter** *extends* **Writer**

Klasse **PrintWriter** : schreibt formatierten Text in einen *stream*.
Die Methoden sind ähnlich zu:

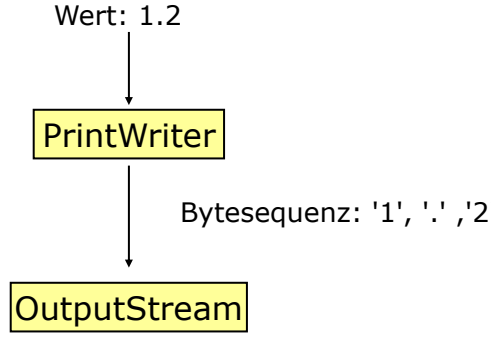
print
println

Überladen um jegliche Datentypen zu schreiben.
ebenso eine **close** Methode.
Keine der Methoden wirft Ausnahmen.

Erzeugen eines PrintWriter Objekts

Ein PrintWriter-Objekt nutzt ein OutputStream-Objekt.

Die Ausgabe wird formatiert - welche Zeichen (char) müssen geschrieben werden - und sendet die Zeichen dann an den OutputStream.



Erzeugen eines PrintWriter Objekts

Um einen PrintWriter zu erzeugen,
muss zuerst ein OutputStream erzeugt werden.

```
OutputStream outStream = null;      // keep compiler happy
                                    // create the output stream
try {
        outStream = new FileOutputStream( fileName);
}
catch (IOException e) {
        System. out. println(" Error in opening output file");
        System. exit( 1);
}                                   // end try/ catch
                                    // create PrintWriter to write
                                    //                 formatted output

PrintWriter writer = new PrintWriter( outStream);
```

Erzeugen eines PrintWriter Objekts (3)

Kürzere Version.

```
PrintWriter writer = null;
try {
        writer = new PrintWriter( new FileOutputStream(" outputfile. txt"));
}
catch (IOException e) {
        System. out. println(" Error in opening output file");
        System. exit( 1);
}
```

Nutzung eines PrintWriter Objekts

```
                                        // write a string
writer. println(" hello, world!");
                                        // write some other types
writer. print(" the year is ");
writer. print( 2012);
writer. print(", the current temperature is ");
writer. println( 10.4);
writer. print(" and my boolean variable is ");
boolean b = true;
writer. println( b);
writer. close()
```

IO_2.java

In der Datei steht:

hello, world!
the year is 2012, the current temperature is 10.4
and my boolean variable is true

Standard Output

Schreiben in den "standard output" (Bildschirm):

System.out.print
System.out.println

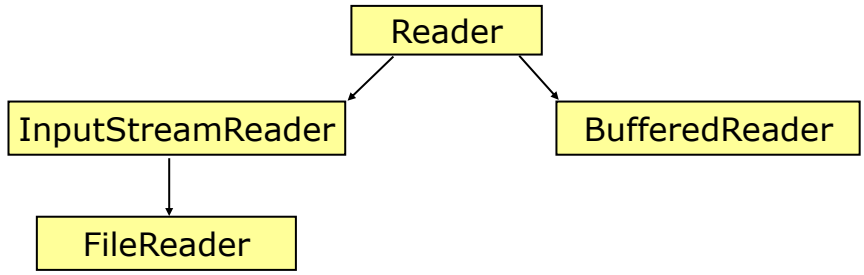
System ist eine Klasse im API.

System.out ist *static* in System.

Der Datentyp von **System.out** ist **PrintStream**.
PrintStream ist ähnlich zu **PrintWriter.**

Text Input

Lesen von einer Textdatei.



Ein **FileReader** liest ein Zeichen von einer Textdatei.

Konstruktor:

FileReader(String name)
throws FileNotFoundException

Anwenden von FileReader

Ein "Reader" hat die Methode :

int read () throws IOException

Liest einzelnes Zeichen.

' returns -1 ' am Ende der Datei.

Wird gewöhnlich selten direkt verwendet !!

BufferedReader

Ein "BufferedReader" kann ganze Zeilen auf einmal lesen,
- angenehmer.

Erzeugen eines BufferedReader aus einem einfacheren Reader:

new BufferedReader(new Reader("name"));

hat die Methode

String readLine() throws IOException .

Der von *readLine* zurückgegebene ' String ' enthält nicht das end-of-line Zeichen.
Am Ende der Datei wird null zurückgegeben (null != "").

Beispiel zu BufferedReader

```
String filename = "sample. txt";
try {
        BufferedReader reader = new BufferedReader( new FileReader( filename ));

        String inputLine;                       // a line of input from the file
        while (true) {                          // exit with break
                inputLine = reader. readLine();
                if (inputLine == null) break;
                System. out. println( inputLine);
        }                                       // end while
        reader. close();
}
catch (FileNotFoundException e) {
        System. out. println(" Error: " + filename + " not found");
}
catch (IOException e) {
        System. out. println(" I/ O error while reading");
}
```

IO_3.java

Andere Datentypen lesen?

"BufferedReader" kann *char* oder *string* lesen.

Was ist mit *int, double, boolean, etc...*

Es gibt keine direkten Methoden:

ein *string* ist in die anderen Typen zu konvertieren**.**

string nach int

int i = Integer.parseInt(s)

string nach double

double d = Double.parseDouble(s)

Es ist jedoch notwendig zu wissen was man aus der Datei erwartet.

----> NumberFormatException

Beispiel

Angenommen wir erwarten in der Eingabedatei .

int,
double,
boolean

(jeweils ein separate Zeile)

Die jeweilige Methode zum parsen wird eine **NumberFormatException** auswerfen falls **int** und/oder **double** nicht entsprechend formatiert sind.

Beispiel IO_4.java (2)

```
try {
        BufferedReader reader = new BufferedReader(...);
                                                    // read the int
        String inputLine = reader. readLine();
        if (inputLine == null)
                throw new NumberFormatException();
        int theInt = Integer. parseInt( inputLine);
        System. out. println(" the integer is: " + theInt);
```

Beispiel (3)

```
                                        // read the double
        inputLine = reader. readLine();
        if (inputLine == null)
                throw new NumberFormatException();
        double theDouble = Double. parseDouble( inputLine);
        System. out. println(" the double is: " + theDouble);
```

Beispiel (4)

```
                                        // read the boolean
        inputLine = reader. readLine();
        if (inputLine == null)
                throw new NumberFormatException();
        boolean theBoolean;
        if (inputLine. equalsIgnoreCase("true"))
        theBoolean = true;
        else if (inputLine. equalsIgnoreCase("false"))
                theBoolean = false;
            else
                throw new NumberFormatException();
        System. out. println(" the boolean is: " + theBoolean);
        reader. close();
    }
```

Beispiel abschliessen

```
                                        // read the boolean

catch (NumberFormatException e) {
        System. out. println(" illegal file format");
}
catch (FileNotFoundException e) {
        System. out. println(" Error: file " + filename + " not found");
}
catch (IOException e) {
        System. out. println(" I/ O error while "+ "reading from file");
}
```

IO_4.java

java.io.File

- Plattformunabhänige Definition von Datei und Verzeichnis Namen.
- Hat Methoden um Files und Directories abzufragen.
- Constants to represent platform separators:
  directory
    UNIX mit / und DOS \
  path
    UNIX mit : und DOS mit ;

Erzeugen neuer File-Objekte :

```
File myFile
myFile = new File("test.dat");
myFile = new File("/", "test.dat");
File myDir = new File("/");
myFile = new File(myDir, "test.dat");
```

java.io.File

String getName();
String getPath();
String getAbsolutePath();
String getParent();
boolean renameTo(File);
long lastModified();
long length();
boolean delete();

boolean exists();
boolean canWrite();
boolean canRead();
boolean isFile();
boolean isDirectory();
boolean mkdir();
String[] list();

Standard Input

Lesen vom "standard input" (Tastatur)

System.in ist *static* in **System**

Der Datentyp von **System.in** ist **InputStream**.
InputStream

System.in.read liest einzelne Bytes

Lesen von Tastatur

```
import java.io.*;
    static String readln(){
            InputStreamReader isr = new InputStreamReader ( System.in );
            BufferedReader br = new BufferedReader ( isr );
            String s = null;
            try {
                    s = br.readLine();
            }
            catch ( IOException ioe ) {
                    System.err.println(ioe);
            }
            return s;
    }
```

Nutzen von Exceptions

IO_5.java

```
String filename = "";
BufferedReader kbd_reader;
kbd_reader = new BufferedReader(new InputStreamReader(System.in));
BufferedReader file_reader = null ;
while (file_reader== null) {
        try {
                System. out. print(" enter input file name: ");
                filename = kbd_reader.readLine();
                file_reader = new BufferedReader( new FileReader( filename));
        }
        catch (FileNotFoundException e) {
                System. out. println( filename + " does not exist.");
                System. out. println(" please try again");
        }
        catch (IOException e) {
                System. out. println(" Problems when reading from Keyboard");
        }
} // end while // now read from reader...
```

Binärdateien (siehe auch DataIODemo.java)

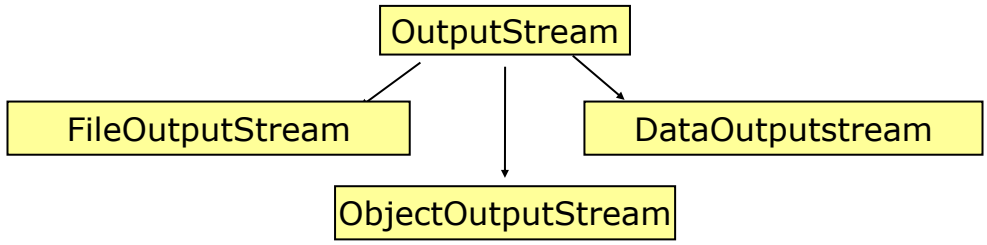
Zur Erinnerung: *Textdateien*: enthalten ASCII (oder Unicode) Zeichen.

Binärdateien: enthalten Bytes mit binär Werten, nicht zur Interpretation als Zeichen gedacht.
Beispiele: **.class, .exe, .pdf, .doc .**

Eine Binärdatei ist ohne zugrundeliegendes Dateiformat bedeutungslos.
Das Dateiformat stellt eine Interpretation der Daten bereit.
Dateiformat = Abmachung wie eine Datei zu lesen ist.

DataOutputStream

DataOutputstream zum Schreiben binärer Daten.



Zur Erinnerung: Ein **FileOutputstream** schreibt einzelne Bytes.

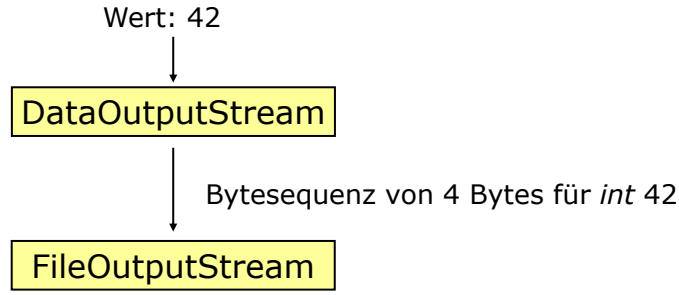
DataOutputStream übersetzt Java Datentypen in Byterepräsentation!

DataOutputStream anwenden

Erzeugen eines DataOutputStreams aus einem einfachen OutputStream!

DataOutputStream (OutputStream out)

z.B. **DataOutputStream** binary_stream =
new **DataOutputStream** (
new **FileOutputStream** (filename));



DataOutputStream Methoden

void writeInt(int)

void writeDouble(double)

void writeChar(char)

void flush()

.....

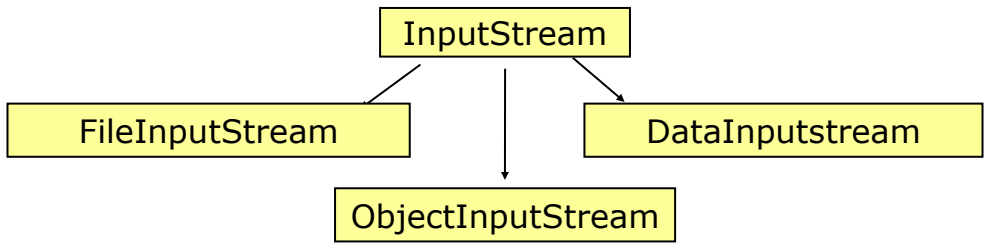
void writeUTF(String)

Binäre Repräsentation eines Strings

Binäre Repräsentationen sind Maschinen unabhängig!

Binary Input

DataInputstream zum lesen binärer Daten.



InputStream: abstrakte Klasse zum Lesen einzelner Bytes.
FileInputStream: einfache Klasse zum Lesen einzelner Bytes aus Dateien.

DataInputStream übersetzt Byterepräsentation nach Java Datentypen!

DataInputStream Methoden

```
DataInputStream binary_stream =
                    new DataInputStream (
                            new FileInputStream (filename ) );
```

```
int readInt()             throws IOException
double readDouble() throws IOException
...
.......
String  readUTF()      throws IOException
```