Informatik IIb

Objektorientierte Programmierung mit Java

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Wintersemester 2010/11

Vorlesung 4. Streams (Datenströme)

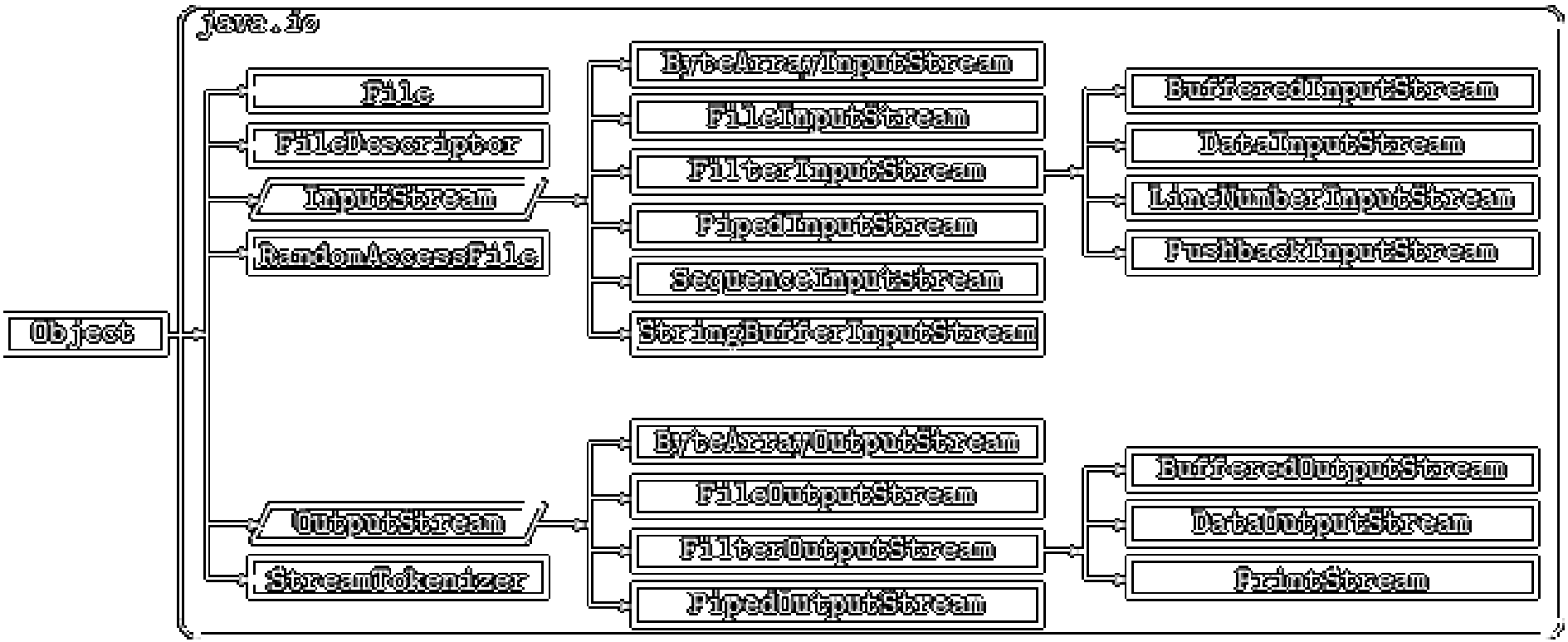
Lernziele

- Kenntnis von Stream-Mechanismen
- Kenntnis von Tastatureingaben
- Kenntnis des Lesens und Schreibens von Dateien

Java-Streams

- In Java werden wie in C++ für die Ein- und Ausgabe Streams benutzt, wobei es aber einige stärker spezialisierte Klassen als in C++ gibt.
- Die wesentlichen Klassen für Streams und Exceptions finden sich im Package java.io.
- In C++ gibt es die abstrakte Basisklasse ios, die die gemeinsame Funktionalität aller Streams zur Verfügung stellt und wesentliche Schnittstellen vorschreibt.
- In Java wird schon von vornherein zwischen Ein- und Ausgabe-Streams unterschieden.
- Es gibt zwei abstrakte Basisklassen InputStream und OutputStream, die nicht mit istream und ostream von C++ verwechselt werden dürfen.

java.io


java.io
File
FileDescriptor
InputStream
RandomAccessFile
Objects
ByteArrayInputStream
FileInputStream
FilterInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
OutputStream
StreamTokenizer
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
PipedOutputStream
BufferedOutputStream
DataOutputStream
PrintStream

Reader und Writer

- Die Entsprechungen für cin, cout und cerr in C++ sind System.in, System.out und System.err aus den Klassen BufferedInputStream bzw. PrintStream.
- Für die meisten angegebenen Stream-Klassen gibt es in JAVA 1.1 Versionen namens Reader bzw. Writer. Sie sind auf die Ein-/Ausgabe von Zeichen (statt Bytes) spezialisiert.
- Fast alle Operationen werfen eine IOException, so dass Blöcke mit Ein-/Ausgabe-Aufrufen immer in ein entsprechendes try/catch eingeschlossen werden müssen.

Eingabe von der Tastatur

- Eingabe von der Tastatur ist nicht so elegant wie in C++.
- Benutzung des Eingabe-Streams System.in.
- System.in liest zeichenweise von der Tastatur.
- Eingelesene Zeichenketten müssen in primitive Datentypen konvertiert werden.
- Konvertierung erfolgt mit Instanzen von InputStreamReader und BufferedReader.
- BufferedReader wird verwendet zum Zeilenweisen lesen mit der Methode readLine().
- Konvertierung in einen primitiven Datentyp erfolgt mit der Methode parse<Typ>, die für jeden Typ existiert.

Eingabe von der Tastatur

```
1  import java.io.*;
2
3  public class SimpleInput {
4    public static void main(String[] args)
5    throws IOException
6    {
7      int a, b, c;
8      BufferedReader din = new BufferedReader(
9                new InputStreamReader(System.in));
10
11     System.out.println("Bitte Zahl a eingeben: ");
12     a = Integer.parseInt(din.readLine());
13     System.out.println("Bitte Zahl b eingeben: ");
14     b = Integer.parseInt(din.readLine());
15     c = a + b;
16     System.out.println("a+b="+c);
17   }
18 }
```

Die Klasse Reader

- Für stream-orientierte Eingabe steht die abstrakte Klasse java.io.Reader zur Verfügung.
- Methoden:

```
public Reader()

public void close()
public void mark(int readAheadlimit)
public boolean markSupported()

public int read()
public int read(char[] cbuf)
public int read(char[] cbuf, int off, int len)

public long skip(long n)
public boolean ready()
public void reset()
```

Methoden der Klasse Reader

- Die parameterlose Variante read() liest das nächste Zeichen aus dem Eingabestrom und liefert es als int, dessen Wert im Bereich von 0 bis 65535 liegt.
- Ein Rückgabewert von -1 zeigt das Ende des Eingabestroms an.
- Die beiden anderen Varianten von read übertragen eine Reihe von Zeichen in das als Parameter übergebene Array und liefern die Anzahl der tatsächlich gelesenen Zeichen als Rückgabewert.
- Die Methode ready liefert true, falls der nächste Aufruf von read erfolgen kann, ohne daß die Eingabe blockt.
- Die Methode close schließt den Eingabestrom.

Methoden der Klasse Reader

- Die Methoden mark und reset ermöglichen, eine bestimmte Position innerhalb des Eingabe-Streams zu markieren und zu einem späteren Zeitpunkt wieder anzuspringen.
- Das funktioniert nur, wenn das Markieren unterstützt wird.
- Test auf Unterstützung des Markierens mit der Methode markSupported.
- mark merkt sich die aktuelle Leseposition und spezifiziert, wie viele Bytes anschließend maximal gelesen werden können, bevor die Markierung ungültig wird.
- reset setzt den Lesezeiger an die markierte Stelle zurück.
- Die Methode skip überspringt n Zeichen, der Rückgabewert von skip gibt die tatsächliche Anzahl an.

Abgeleitete Klassen von Reader

Klasse	Bedeutung
InputStreamReader	Basisklasse für alle Reader, die einen Byte-Stream in einen Character-Stream umwandeln.
FileReader	Konkrete Ableitung von InputStreamReader zum Einlesen aus einer Datei.
FilterReader	Abstrakte Basisklasse für die Konstruktion von Eingabefiltern.
PushbackReader	Eingabefilter mit der Möglichkeit, Zeichen zurückzugeben.
BufferedReader	Reader zur Eingabepufferung und zum Lesen von kompletten Zeilen.
LineNumberReader	Ableitung aus BufferedReader mit der Fähigkeit, Zeilen zu zählen.
StringReader	Reader zum Einlesen von Zeichen aus einem String.
CharArrayReader	Reader zum Einlesen von Zeichen aus einem Zeichen-Array.
PipedReader	Reader zum Einlesen von Zeichen aus einem PipedWriter.

Die Klasse FileReader

- Die aus InputStreamReader abgeleitete Klasse FileReader, ermöglicht die Eingabe aus einer Datei.
- Sie implementiert die abstrakten Eigenschaften von Reader und bietet zusätzliche Konstruktoren, die es erlauben, eine Datei zu öffnen.
- Bei Übergabe von fileName wird die so benannte Datei zum Lesen geöffnet.
- Die beiden anderen Konstruktoren erwarten ein File-Objekt, das eine zu öffnende Datei spezifiziert, oder ein FileDescriptor-Objekt, das eine bereits geöffnete Datei angibt.

Die Klasse FileReader (Forts.)

```
public FileReader(String fileName)
  throws FileNotFoundException

public FileReader(File file)
  throws FileNotFoundException

public FileReader(FileDescriptor fd)
```

Lesen aus Dateien

```
1  import java.io.*;
2  public class FileRead
3  {
4    public static void main(String[] args)
5    {
6      FileReader f;
7      int c;
8      try {
9        f = new FileReader("c:\\config.sys");
10       while ((c = f.read()) != -1) {
11           System.out.print((char)c);
12       }
13       f.close();
14     } catch (IOException e) {
15       System.out.println("Fehler beim Lesen der Datei");
16     }
17   }
18 }
```

Lesen aus Datei mit Pufferung

```
1  import java.io.*;
2  public class BufferedFileRead
3  {
4    public static void main(String[] args)
5    {
6      BufferedReader f;
7      String line;
8      try {
9        f = new BufferedReader(
10           new FileReader("c:\\config.sys"));
11       while ((line = f.readLine()) != null) {
12         System.out.println(line);
13       }
14       f.close();
15     } catch (IOException e) {
16       System.out.println("Fehler beim Lesen der Datei");
17     }
18   }
19 }
```

Ein weiteres Beispiel

```
import java.io.*;

public class LeseDatei {

  public static void main(String[] argv) {

    String nameEingabedatei;
    String zeile;
    File eingabedatei;
    FileReader fr;
    BufferedReader br;
```

Ein weiteres Beispiel (Forts.)

```
try {
  nameEingabedatei = argv[0];
  System.out.println("Inhalt␣von␣"
                     +nameEingabedatei);
  eingabedatei = new File(nameEingabedatei);
  fr = new FileReader(eingabedatei);
  br = new BufferedReader(fr);
  zeile=br.readLine();
  while ( zeile != null ) {
    System.out.println(zeile);
    zeile=br.readLine();
  }
  br.close();
}
```

Ein weiteres Beispiel (Forts.)

```
    catch (ArrayIndexOutOfBoundsException aioobe) {
      System.out.println("Aufruf␣mit:"+
                         "java␣LeseDatei␣eingabedatei"
    }
    catch (FileNotFoundException fnfe) {
      System.out.println("Habe␣gefangen:␣"+fnfe);
    }
    catch (IOException ioe) {
      System.out.println("Habe␣gefangen:␣"+ioe);
    }

  } // main

} // public class LeseDatei
```

Die Klasse Writer

- Für stream-orientierte Ausgabe steht die abstrakte Klasse java.io.Writer zur Verfügung.
- Methoden:

```
protected Writer()

public void close()
public void flush()

public void write(int c)
public void write(char[] cbuf)
abstract public void write(char[] cbuf,
                           int off, int len)
public void write(String str)
public void write(String str, int off, int len)
```

Methoden der Klasse Writer

- Der parameterlose Konstruktor öffnet den Ausgabestrom und und bereitet ihn für einen nachfolgenden Aufruf von write vor.
- close schließt den Ausgabestrom.
- write(int c) schreibt den Parameter c als Byte in den Ausgabestrom.
- Daneben gibt es weitere Varianten, die ein Array von Bytes oder ein String-Objekt als Parameter erwarten und dieses durch wiederholten Aufruf der primitiven write-Methode ausgeben.
- Mit flush werden eventuell vorhandene Puffer geleert und die darin enthaltenen Daten an das Ausgabegerät weitergegeben.

Abgeleitete Klassen von Writer

- Die abstrakte Basisklasse kann Writer nicht instanziert werden.
- Es gibt eine Reihe konkreter aus Writer abgeleiteter Klassen für die Verbindung zu einem konkreten Ausgabegerät oder für Filterfunktionen.

Abgeleitete Klassen von Writer

Klasse	Bedeutung
OutputStreamWriter	Basisklasse für alle Writer, die einen Character-Stream in einen Byte-Stream umwandeln
FileWriter	Konkrete Ableitung von OutputStreamWriter zur Ausgabe in eine Datei
FilterWriter	Abstrakte Basisklasse für die Konstruktion von Ausgabefiltern
PrintWriter	Ausgabe aller Basistypen im Textformat
BufferedWriter	Writer zur Ausgabepufferung
StringWriter	Writer zur Ausgabe in einen String
CharArrayWriter	Writer zur Ausgabe in ein Zeichen-Array
PipedWriter	Writer zur Ausgabe in einen PipedReader

Die Klasse FileWriter

- Die aus OutputStreamWriter abgeleitete Klasse FileWriter, ermöglicht die Ausgabe in eine Datei.
- Sie implementiert die abstrakten Eigenschaften von Writer und bietet zusätzliche Konstruktoren, die es erlauben, eine Datei zu öffnen.
- Bei Übergabe von fileName wird die so benannte Datei, falls sie existiert, zum Schreiben geöffnet und der bisherige Inhalt gelöscht.
- Falls die Datei mit dem Namen fileName nicht existiert, wird sie erzeugt.
- Sukzessive Aufrufe von write schreiben weitere Bytes in diese Datei.
- Wird zusätzlich der Parameter append mit dem Wert true an den Konstruktor übergeben, so werden die Ausgabezeichen an die Datei angehängt, falls sie bereits existiert.

Die Klasse FileWriter (Forts.)

```
public FileWriter(String fileName)
  throws IOException

public FileWriter(String fileName,
                  boolean append)
  throws IOException

public FileWriter(File file)
  throws IOException

public FileWriter(FileDescriptor fd)
```

Schreiben in Datei mit Pufferung

```
1  import java.io.*;
2  public class BufferedFileWrite
3  {
4    public static void main(String[] args)
5    {
6      BufferedWriter f;
7      String s;
8      try {
9        f = new BufferedWriter(
10           new FileWriter("buffer.txt"));
11       for (int i = 1; i <= 10000; ++i) {
12         s = "Dies ist die " + i + ". Zeile";
13         f.write(s);
14         f.newLine();
15       }
16       f.close();
17     } catch (IOException e) {
18       System.out.println("Fehler beim Erstellen der Datei");
19     }
20   }
21 }
```

Benutzung von FilterWriter

- Überlagerung der abstrakten Klasse FilterWriter.
- FilterWriter besitzt ein internes Writer-Objekt out, das bei der Instanzierung an den Konstruktor übergeben wird.
- FilterWriter überlagert drei der vier write-Methoden, um die Ausgabe auf out umzuleiten.
- Die vierte write-Methode (write(String)) wird nicht überlagert, sondern ruft in Writer die Variante write(String, int, int) auf.

Vorgehen für eigene Filterklasse

- Die Klasse wird aus FilterWriter abgeleitet.
- Im Konstruktor wird der Superklassen-Konstruktor aufgerufen, um out zu initialisieren.
- Die drei write-Methoden werden separat überlagert. Dabei wird jeweils vor der Übergabe der Ausgabezeichen an die Superklassenmethode die eigene Filterfunktion ausgeführt.
- Anschließend kann die neue Filterklasse wie gewohnt in einer Kette von geschachtelten Writer-Objekten verwendet werden.

Beispiel: Zeichen in Großschrift schreiben

```
import java.io.*;

class UpCaseWriter
extends FilterWriter
{
  public UpCaseWriter(Writer out)
  {
    super(out);
  }

  public void write(int c)
  throws IOException
  {
    super.write(Character.toUpperCase((char)c));
  }
```

Beispiel: Zeichen in Großschrift schreiben (Forts.)

```
public void write(char[] cbuf, int off, int len)
  throws IOException
  {
    for (int i = 0; i < len; ++i) {
      write(cbuf[off + i]);
    }
  }

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

Beispiel: Zeichen in Großschrift schreiben (Forts.)

```
public class UpperCaseFilterWriter
{
    public static void main(String[] args)
    {
        PrintWriter f;
        String s = "und dieser String auch";

        try {
            f = new PrintWriter(
                new UpCaseWriter(
                new FileWriter("upcase.txt")));
            //Aufruf von au"sen
            f.println(
                "Diese Zeile wird gross geschrieben");
            //Test von write(int)
            f.write('a');
            f.println();
```

Beispiel: Zeichen in Großschrift schreiben (Forts.)

```
            //Test von write(String)
            f.write(s);
            f.println();
            //Test von write(String, int, int)
            f.write(s,0,17);
            f.println();
            //Test von write(char[], int, int)
            f.write(s.toCharArray(),0,10);
            f.println();
            //---
            f.close();
        } catch (IOException e) {
            System.out.println(
                "Fehler beim Erstellen der Datei");
        }
    }
}
```

Die Klasse PrintWriter

- Die Klasse java.io.PrintWriter ermöglicht die Ausgabe primitiver Datentypen in textueller Form.
- Konstruktoren

```
public PrintWriter(Writer out)

public PrintWriter(Writer out, boolean autoflush)
```

- Der erste Konstruktor instanziert ein PrintWriter-Objekt durch Übergabe eines Writer-Objekts, auf das die Ausgabe umgeleitet werden soll.
- Beim zweiten Konstruktor gibt zusätzlich der Parameter autoflush an, ob nach der Ausgabe einer Zeilenschaltung automatisch die Methode flush aufgerufen werden soll.
- Ausgabe primitiver Datentypen geschieht mit überladenen Methoden print und println.

Beispiel: Textuelle Ausgabe

```
import java.io.*;

public class BufferedPrintWriter
{
  public static void main(String[] args)
  {
    PrintWriter f;
    double sum = 0.0;
    int nenner;
    try {
      f = new PrintWriter(
          new BufferedWriter(
          new FileWriter("Summe.txt")));
```

Beispiel: Textuelle Ausgabe (Forts.)

```
        for (nenner = 1; nenner <= 9000; nenner *= 2) {
          sum += 1.0 / nenner;
          f.print("Summand: 1/");
          f.print(nenner);
          f.print("   Summe: ");
          f.println(sum);
        }
        f.close();
      } catch (IOException e) {
        System.out.println(
            "Fehler beim Erstellen der Datei");
      }
    }
}
```

Ein weiteres Beispiel

Erzeugen einer HTML-Datei

```
public class SchreibeDatei {

  static void schreibeKopf(BufferedWriter bw, String newline,
                           String code)
  throws IOException {
    bw.write(""+newline);

    bw.write(" < HEAD > "+newline);
    bw.write(" < TITLE > "+newline);
    bw.write("Listing von "+code+" "+newline);
    bw.write(" < /TITLE > "+newline);
    bw.write(" < /HEAD > "+newline);

    bw.write(" < BODY > "+newline);
    bw.write(" "+newline);
    bw.write(" < HR > "+newline);
    bw.write(" "+newline);
```

Ein weiteres Beispiel (Forts.)

```
    bw.write("␣<␣TABLE␣BORDER=0␣CELLSPACING=0␣CELLPADDING=10␣"
             +"WIDTH=400␣BGCOLOR=WHITE␣>␣"+newline);
    bw.write("␣<␣TR␣>␣"+newline);
    bw.write("␣<␣TD␣WIDTH=150␣BGCOLOR=LIGHTGREY␣>␣"
             +"Java␣Quellcode␣<␣/TD␣>␣"+newline);
    bw.write("␣<␣TD␣WIDTH=250␣BGCOLOR=ORANGE␣>␣␣<␣B␣>␣␣<␣TT␣>␣"
             +code+".java␣<␣/TT␣>␣␣<␣/B␣>␣␣<␣/TD␣>␣"+newline);
    bw.write("␣<␣/TR␣>␣"+newline);
    bw.write("␣<␣/TABLE␣>␣"+newline);
    bw.write("␣"+newline);

    bw.write("␣<␣TABLE␣BORDER␣WIDTH=600␣BGCOLOR=LIGHTGREY␣>␣"
             +newline);
    bw.write("␣<␣TR␣>␣"+newline);
    bw.write("␣<␣TD␣>␣"+newline);
    bw.write("␣<␣PRE␣>␣"+newline);
  } // static void schreibeKopf
```

Ein weiteres Beispiel (Forts.)

```
static void schreibeFuss(BufferedWriter bw, String newline)
throws IOException {
  bw.write("␣</␣PRE␣>␣"+newline);
  bw.write("␣</␣TD␣>␣"+newline);
  bw.write("␣</␣TR␣>␣"+newline);
  bw.write("␣</␣TABLE␣>␣"+newline);
  bw.write("␣"+newline);
  bw.write("␣<␣HR␣>␣"+newline);
  bw.write("␣"+newline);
  bw.write("␣</␣BODY␣>␣"+newline);
  bw.write("␣</␣HTML␣>␣"+newline);
} // static void schreibeFuss
```

Ein weiteres Beispiel (Forts.)

```
public static void main(String[] argv) {

  String nameAusgabedatei;
  String zeile, newline;
  File ausgabedatei;
  FileWriter fw;
  BufferedWriter bw;

  newline = System.getProperty("line.separator");
  try {
    nameAusgabedatei = argv[0];
    ausgabedatei = new File(nameAusgabedatei+".html");
    fw = new FileWriter(ausgabedatei);
    bw = new BufferedWriter(fw);
    schreibeKopf(bw,newline,nameAusgabedatei);
    schreibeFuss(bw,newline);
    bw.close();
  }
```

Ein weiteres Beispiel (Forts.)

```
    catch (ArrayIndexOutOfBoundsException aioobe) {
      System.out.println("Aufruf mit: java SchreibeDatei name");
      System.out.println("erzeugt eine Datei name.html");
    }
    catch (IOException ioe) {
      System.out.println("Habe gefangen: "+ioe);
    }

  } // main

} // public class SchreibeDatei
```

Eigenschaften und Manipulation von Dateien

- Für das Lesen und Schreiben von Dateien sind deren Eigenschaften von Interesse.
- Eine Datei muss existieren, bevor sie gelesen werden kann.
- Das Auffinden einer Datei im Verzeichnisbaum kann erforderlich sein.
- Für das Lesen und Schreiben sind bestimmte Berechtigungen erforderlich.
- Der Inhalt eines Verzeichnisses soll angegeben werden.

Eigenschaften einer Datei ausgeben

```
import java.io.*;

public class FileTest {

  public static void main(String[] argv) {

    String fileSep = System.getProperty("file.separator");
    System.out.println("fileSep = "+fileSep);
    File datei = new File("bsp.txt");
    System.out.println(datei.exists());
    System.out.println(datei.getName());
    System.out.println(datei.getPath());
    System.out.println(datei.getParent());
    System.out.println(datei.getAbsolutePath());
    System.out.println(datei.isAbsolute());
    System.out.println(datei.canRead());
    System.out.println(datei.length());
    System.out.println(datei.lastModified());
  }
} // public class FileTest
```

Dateiverzeichnis auflisten

```
import java.io.*;
public class FileDir {
  public static void main(String[] argv) {
    String fileSep = System.getProperty("file.separator");
    File verzeichnis = new File(".."+fileSep+"..");
    System.out.println(verzeichnis.exists());
    System.out.println(verzeichnis.getName());
    System.out.println(verzeichnis.getPath());
    if (verzeichnis.exists()) {
      if (!verzeichnis.isFile()) {
        System.out.println();
        System.out.println("Listing aller Dateien");
        String[] dateien = verzeichnis.list();
        for (int i=0; i < dateien.length; i++) {
          System.out.println(dateien[i]);
        }
      }
    }
  }
} // public class FileDir
```

Zusammenfassung

- Ein- uns Ausgaben erfolgt in Java durch Streams.
- Eingaben über die Tastatur ist etwas kompliziert.
- Streams können verschachtelt werden.
- Lesen und Schreiben von Dateien erfolgt mit eigenen Klassen.
- Ausnahmen müssen abgefangen werden.
- Mit Ausgabefiltern lassen sich Dateiinhalte manipulieren.