Kapitel 12

Ausnahmen und Zusicherungen

12.1 Ausnahmebehandlung

- Ausnahmen oder Fehler werden in Prozeduren häufig mit Rückgabewert -1 signalisiert
- Fehler oder irreguläre Situation behandeln macht den Kontrollfluss u.U. sehr kompliziert!
- Explizite Ausnahmebehandlung: Eine Ausnahme (*exception*) ist eine Art Signal, das das Auftreten eines Fehlers oder einer irregulären Situation kennzeichnet.
- In Java werden Ausnahmen mit der **throw**-Anweisung ausgelöst.
- Ausnahmen können abgefangen und behandelt werden: *Exception Handler*
- Ausnahmebehandlungsroutinen werden mit der **try – catch – finally** Anweisung geschrieben

> The paradigm for using exceptions is the *try–catch–finally* sequence: you *try* something; if that something throws an exception, you *catch* the exception; and *finally* you clean up from either the normal code path or the exception code path, whichever actually happened.
> *The Java Programming Language, 3rd Edition, K. Arnold, J. Gosling, D. Holmes, Addison Wesley 2001, S. 28*

Programm 12.1, S. 290, zeigt ein erstes Beispiel.

Programm 12.1: Ausnahmebehandlung mit try – catch (*FactorialExcept.java*)

```java
/**
 * Berechnung der Fakultaet einer Zahl, uebergeben als Argument
 *  Aufruffehler werden mit try/catch behandelt
 */

public class FactorialExcept {

  public static void main( String[] args ) {

    // Versuche die Fakultaet zu berechnen --
    // was schief geht, wird in den nachfolgenden catch-Anweisungen behandelt

    try {
          int n = Integer.parseInt(args[0]);
          System.out.println(n +"! = " + fac(n));
    }
    // kein Argument angegeben, args[0] ist undefiniert
    catch (ArrayIndexOutOfBoundsException e) {
      System.err.println("Argument muss angegeben werden");
      System.err.println("Usage: java FactorialExcept <zahl>");
      System.exit(1);
    }

    // Argument ist da, aber keine Zahl
    // wird von Integer.parseInt() ausgeloest
    catch (NumberFormatException e) {
      System.err.println("Argument muss eine Integer sein");
      System.exit(2);
    }

    // Argument ist kleiner als 0
    // wird von fac() ausgeloest
    catch ( IllegalArgumentException e) {
      // von der Methode fac() gesendete Nachricht anzeigen
      System.err.println("Ungueltiges Argument: " + e.getMessage());
      System.exit(3);
    }
    finally {
      System.err.println("Zum guten Schluss!");
      // System.exit(4);
    }
    System.err.println("Zum allerletzten Schluss!" );
    System.exit(0);
  }

  public static double fac(int k) {
     if ( k < 0 ) throw new IllegalArgumentException("Parameter muss >= 0 sein");
     double f = 1.0;
     while ( k > 1 ) { f *= k; k--;}
     return f;
  }
}
```

Erläuterungen zu Programm 12.1 (S. 290):

- Zeile 14:

  Wir versuchen aus einem String eine Zahl zu machen – die Wrapper-Klasse *Integer* hat eine Methode **parseInt()** ; das kann auch schief gehen, weswegen wir diesen Aufruf in einem **try**-Block absichern!

- Zeile 15:

  Da wir nun das Thema “Ausnahmenbehandlung” angehen und dies im Kontext mit Ein-/Ausgabe wichtig ist, verlassen wir die Ulmer “Spezialklassen” und kehren zum Java-Standard zurück – **System.out** ist ein automatisch erzeugtes Objekt, das eine Verbindung zum Bildschirm (**stdout**) repräsentiert und eine Methode **println()** zur Verfügung stellt – mehr dazu im letzten Abschnitt dieses Kapitels.

- Zeile 19:

  Fehler gehören an die Diagnoseausgabe (**stderr**) – mit dem automatisch erzeugten Objekt **System.err** können wir das bewerkstelligen – auch dazu mehr am Ende dieses Kapitels

- Die folgenden drei **catch**-Blöcke gehen vom speziellen zum allgemeinen Ausnahmefall: kein Argument angegeben, falsches Argument angegeben, unzulässiges Argument angegeben

  Ausnahmen sind Objekte einer entsprechenden Ausnahmenklasse – in der *catch*-Anweisung muss ein solches deklariert werden!

  Zeile 18: Objekt e der Ausnahmeklasse $ArrayIndexOutOfBoundsException$

- Zeile 35:

  Die Ausnahmeklassen haben natürlich alle einen Konstruktor, mit dem ein Ausnahmeobjekt erzeugt werden kann. Dieser hat typischerweise ein String-Argument, welches andererseits mit einer Methode **getMessage()** abgefragt werden kann (Rückgabewert).

- Zeile 47:

  Eine Ausnahme bei Methodenaufrufen besteht häufig darin, dass unzulässige Parameter übergeben werden – dies behandelt die Ausnahmeklasse **IllegalArgumentException**.

  hier: für negative ganze Zahlen erzeugen wir eine Ausnahme (mit **new** und **Konstruktoraufruf**), die wir an den Aufrufer unserer Methode $fac()$ “werfen”.

Ausführung von Programm 12.1, S. 290:

```
euclid$ java FactorialExcept
Argument muss angegeben werden
euclid$
euclid$ echo $?
1
euclid$ java FactorialExcept aaaaaa
Argument muss eine Integer sein
euclid$ echo $?
2
euclid$ java FactorialExcept -4
euclid$= 0 sein
euclid$ echo $?
3
euclid$ java FactorialExcept 4
4! = 24.0
Zum guten Schluss!
Zum allerletzten Schluss!
euclid$ echo $?
0
euclid$
```

Grundstruktur der try/catch/finally-Anweisung

```
try {
  /* Im Normalfall wird dieser Code-Block wie geschrieben
   * abgearbeitet; u.U. kann er aber eine Ausnahme ausloesen:
   * direkt ueber eine throw-Anweisung oder
   * indirekt durch Aufruf einer Methode, die eine Ausnahme ausloest
   */
}

catch ( FirstException e1 ) {
  /* Dieser Block enthaelt Anweisungen, die Ausnahmen (Ausnahmeobjekte)
   * des Typs >>FirstException<< oder einer davon abgeleiteten Klasse
   * behandeln. Anweisungen in diesem Block koennen unter dem Namen
   * >>e1<< auf dieses Ausnahmeobjekt zugreifen
   */
}

catch ( AnotherException e2 ) {
  /* Dieser Block enthaelt Anweisungen, die Ausnahmen (Ausnahmeobjekte)
   * des Typs >>AnotherException<< oder einer davon abgeleiteten Klasse
   * behandeln. Anweisungen in diesem Block koennen unter dem Namen
   * >>e2<< auf dieses Ausnahmeobjekt zugreifen
   */
}

finally {
  /* Dieser Block enthaelt Anweisungen, die nach dem Verlassen
   * des try-Blocks immer ausgefuehrt werden, egal wie der
   * try-Block verlassen wurde, ob wegen Erreichen des Blockendes,
   * ob wegen Erreichen einer break- / continue- oder return-Anweisung,
   * ob wegen einer Ausnahme, die mit einem catch-Block abgefangen wurde
   * oder ob wegen einer nicht abgefangenen Ausnahme;
   * nicht aber bei Aufruf von System.exit() im try-Block
}
```

Ausführung einer throw-Anweisung

- die normale Programmausführung wird unmittelbar angehalten
- der Interpreter sucht nach einer Ausnahmebehandlungsroutine, die diese Ausnahme abfangen (behandeln) kann
  - zunächst wird im umgebenden Code-Block gesucht
  - wird eine passende gefunden, so wird der Code-Block beendet und die Ausnahmebehandlung des Code-Blocks gestartet
  - nach deren Beendigung geht es mit der Ausführung nach deren Code weiter
- Wenn der umgebende Code-Block keine passende Ausnahmebehandlung hat, so wird im nächst höheren umgebenden Codeblock gesucht, ...!
- Wenn die Methode **keine** Ausnahmebehandlungsroutine enthält, die die mit *throw* ausgelöste Ausnahme behandeln kann:
  - die Methode wird abgebrochen und es geht zum Aufrufer zurück
  - jetzt wird hier nach einer passenden Ausnahmebehandlungsroutine gesucht
  - ggf. geht es im Aufrufstack entsprechend nach oben
  - wird die Ausnahme nirgendwo behandelt, so geht es irgendwann zur *main()*-Methode
  - wird sie auch hier nicht behandelt, so gibt der Java-Interpreter eine Fehlermeldung und einen Stack-Trace aus (der zeigt wo die Ausnahme ausgelöst wurde); danach Termination (siehe Programm 12.2, S. 295, als Modifikation von Programm 12.1, S. 290.

Programm 12.2: Ausnahmen ohne Behandlung (*FactorialExcept1.java*)

```java
/**
 * Berechnung der Fakultaet einer Zahl -- diese wird als Argument uebergeben
 */

public class FactorialExcept1 {

 public static void main( String [] args ) {

  int n = Integer . parseInt ( args [0]);
  System.out . println (n +"! = " + fac (n ));
  System. exit (0);
 }

 public static double fac (int k) {
   if ( k < 0 )
     throw new IllegalArgumentException ("Parameter muss >= 0 sein");
   double fact = 1.0;
   while ( k > 1 ) { fact *= k; k--;}
   return fact ;
 }
}
```

Ausführung:

```
euclid$  java FactorialExcept1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsE
xception: 0
   at FactorialExcept1.main(FactorialExcept1.java:9)
euclid$  java FactorialExcept1 aaaa
Exception in thread "main" java.lang.NumberFormatException: For
input string: "aaaa"
   at java.lang.NumberFormatException.forInputString(NumberFormat
Exception.java:48)
   at java.lang.Integer.parseInt(Integer.java:446)
   at java.lang.Integer.parseInt(Integer.java:496)
   at FactorialExcept1.main(FactorialExcept1.java:9)
euclid$  java FactorialExcept1 -4
Exception in thread "main" java.lang.IllegalArgumentException:
Parameter muss >= 0 sein
        at FactorialExcept1.fac(FactorialExcept1.java:16)
        at FactorialExcept1.main(FactorialExcept1.java:10)
euclid$  java FactorialExcept1 4
4! = 24.0
euclid$
```

Ausnahmetypen:

- **Jede Ausnahme ist ein Objekt!**
- Sein Typ ist **java.lang.Throwable** oder eine daraus abgeleitete Klasse
- Standardmäßig abgeleitete Klassen:
  - **java.lang.Error** : meist nicht behebbare Fehler (**Error-Exceptions**
  - **java.lang.Exception**: weisen auf weniger schwerwiegende Zustände hin **Exceptions**
    Dazu gehören:
    java.io.EOFException (End-of-File)
    java.lang.ArrayIndexOutOfBoundsException (ein Programm versucht, über das Ende eines Arrays hinaus zu lesen / zu schreiben
- Ausnahmen als Objekt können Daten und Methoden, die auf diesen arbeiten, enthalten
  - Klasse **Throwable** und alle davon abgeleiteten Klassen enthalten String-Feld mit lesbarer Fehlermeldung zum Ausnahmezustand
  - wird eingetragen, wenn das Objekt erzeugt wird
  - kann mit der Methode **getMessage()** gelesen werden

Deklaration von Ausnahmen:

- Unterscheidungen:
  - *Error*-Ausnahmen vs. *Exception*-Ausnahmen (s.o)
  - während der Übersetzung **geprüfte** (*checked*), also per Deklaration anzugebende Ausnahmen, und **nicht geprüfte** (*unchecked*) Ausnahmen
- Es gilt:
  - Jedes Ausnahmeobjekt, dessen Klasse von *Error* abgeleitet ist, ist ungeprüft!
  - Jedes Ausnahmeobjekt, dessen Klasse von *Exception* abgeleitet ist, ist geprüft

    Ausnahme:
    es ist eine von *java.lang.RuntimeException* abgeleitete Klasse; in diesem Fall ist sie ungeprüft (*RuntimeException* ist von *Exception* abgeleitet)
- praktisch jede Methode kann ungeprüfte (unprüfbare) Ausnahmen auslösen (*OutOfMemoryError* oder *NullPointerException* bei Methoden, die Arrays oder Objekte verwenden (*null*-Argument))
- geprüfte Ausnahmen treten nur in wohldefinierten Situationen auf (z.B. beim Lesen aus Dateien: *FileNotFoundException* – man muss damit rechnen, dass die gesuchte Datei nicht gefunden werden kann)

Arbeiten mit geprüften / ungeprüften Ausnahmen:

- Beim Schreiben einer Methode, die eine geprüfte Methode auslöst, muss eine **throws**-Anweisung verwendet werden, um diese Ausnahme im Methodenkopf zu deklarieren; der Java-Compiler prüft, ob diese Ausnahmen im Methoden-Kopf deklariert wurden (daher "geprüfte" Ausnahmen).

  Beispiele:

  – siehe Programm 12.3, S. 298:
    `public static void main(String[] args) throws IOException{`
  – siehe Programm 12.1, S. 290:

```
public static double fac(int k) {
  if ( k < 0 )
    throw new IllegalArgumentException("Parameter muss >= 0 sein");
  double f = 1.0;
  while ( k > 1 ) { f *= k; k--; }
  return f;
```

  Hier wird eine Ausnahme des Typs *java.lang.IllegalArgumentException* ausgelöst – diese ist von RuntimeException abgeleitet, daher **ungeprüft** und deswegen muss sie nicht deklariert werden.
  Hätte aber auch deklariert werden können:

```
public static double fac(int k) throws IllegalArgumentException {
  if ( k < 0 )
    throw new IllegalArgumentException("Parameter muss >= 0 sein");
  double fact = 1.0;
  while ( k > 1 ) { fact *= k; k--; }
  return fact;
}
```

- Müssen "geprüfte" Ausnahmen deklariert werden? ↪ Ja!

  – eine (eigene) Methode ruft eine andere Methode auf – diese könnte prinzipiell eine geprüfte Ausnahme auslösen
  – die (eigene) Methode **muss** dann diese Ausnahme entweder mit einer eigenen Routine behandeln oder diese mit *throws* deklarieren, um zu zeigen, dass die (eigene) Methode die Ausnahme weiterreichen kann.

- Woher weiss man, ob eine Methode, die man aufruft, eine Ausnahme auslösen kann?

  ↪ Methodenkopf anschauen
  ↪ den Compiler "fragen" (siehe nachfolgendes Beispiel).

Im Programm 12.3 (S. 298) lesen wir mit der Java-Standardbibliothek (mehr dazu im letzten Abschnitt dieses Kapitels) von *stdin*, in gleicher Weise wie man auch aus Dateien liest. Beim Lesen aus Dateien sind grundsätzlich Ausnahmen zu erwarten:

Programm 12.3: Ohne throws-Deklaration (*Factorial2.java*)

```
1  /**
2   * Integer aus Stdin
3   * Fakultaet iterativ berechnen
4   */
5
6  import java.io.*; // Alle Klassen im Paket java.io importieren
7
8  public class Factorial2 {
9
10   public static void main(String[] args) throws IOException {
11
12     BufferedReader in =
13         new BufferedReader(new InputStreamReader(System.in));
14     for (;;) {
15       System.out.print("Gib Zahl: ");
16       String line = in.readLine();
17       if( ( line == null ) || (line.equals("quit") ) )
18         break;
19       try {
20         int n = Integer.parseInt(line);
21         System.out.println(n + "! = " + factorial(n));
22       }
23       catch(Exception e) {
24         System.out.println("Fehler in Eingabe <<" + e.getMessage() + ">>"); }
25     }
26   }
27
28   public static double factorial(int k) {
29     if ( k < 0 ) return 0.0;
30     double fact = 1.0;
31     while ( k > 1 ) {
32       fact *= k;
33       k--;
34     }
35     return fact;
36   }
37 }
```

Erläuterungen:

- Zeile 12-13:

  Aus dem Objekt *System.in* (Verbindung zi *stdin*) erzeugen wir ein *InputStreamReader*-Objekt, aus diesem wiederum ein *BufferedReader*-Objekt.

- Zeile 16:

  BufferedReader-Objekte kennen eine Methode *readline()*, mit der eine Zeile (*String*-Objekt) gelesen werden kann.

Lässt man in Programm 12.3 (S. 298) im Methodenkopf von *main()* die **throws-**Deklaration weg, so meldet der Java-Compiler:

```
Factorial2.java:16: unreported exception java.io.IOException;
must be caught or declared to be thrown
     String line = in.readLine();
                         ^
1 error
```

Noch ein Beispiel: Berechnung großer Zahlen (**java.math.BigInteger** und **java.util**

Programm 12.4: Große Fakultäten (*FacBig.java*)

```java
// Berechnung der Fakultaeten von grossen Zahlen
import java.math.BigInteger;
import java.util.*;

public class FacBig {
 private static ArrayList table = new ArrayList();
 // ArrayList ist dynamisches Array
 static { // erstes Element mit 0!=1 initialisieren
       table.add(BigInteger.valueOf(1));
      }
 public static BigInteger fac(int k) {
   if ( k < 0 )
     throw new IllegalArgumentException("Parameter muss >= 0 sein");
   for (int size = table.size(); size <= k; size++) {
      BigInteger lastFac = (BigInteger) table.get(size-1);
      BigInteger nextFac = lastFac.multiply(BigInteger.valueOf(size));
      table.add(nextFac);
   }
   return (BigInteger) table.get(k);
 }

 public static void main( String[] args ) {
  try { int n = Integer.parseInt(args[0]);
         for (int i = 1; i <= n; i++)
           System.out.println(i +"! = " + fac(i));
       }
  // kein Argument angegeben, args[0] ist undefiniert
  catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Usage: java FactorialExcept <zahl>");
    System.exit(1);
  }
  catch ( IllegalArgumentException e) {
    System.out.println("Ungueltiges Argument: " + e.getMessage());
    System.exit(2);
  }
  System.exit(0);
 }
}
```

Erläuterungen:

- Zeile 2:

  Die Klasse **java.math.BigInteger** enthält Methoden zum Arbeiten mit sehr großen ganzen Zahlen.

- Zeile 6:

  Die Klasse **ArrayList** implementiert ein Listen-Interface

- Zeilen 8-10:

  mit der Methode *add()* der Klasse *ArrayList* wird eine erste große Integer-Zahl (Wert von 1) hinzugefügt!

- Zeilen 11-20:

  Berechnung von k-Fakultät als große Integer.

  Mit der Methode *size()* erhalten wir die Anzahl der Elemente in unserer *ArrayList* – es werden die weiteren Fakultäten *size* bis k berechnet und im Array abgelegt!

finally-Block:

- wird ausgeführt, auch wenn der try-Block mit break, continue oder return verlassen wird
- erst wird ein catch-Block (sofern passend) angesprungen und dann der finally-Block
- gibt es keinen lokalen catch-Block, so wird der finally-Block ausgeführt, **bevor** die Kontrolle an den nächsten umgebenden catch-Block, der die Ausnahme behandeln kann, weitergegeben wird
- wird im finally-Block die Kontrolle mit einer return, break, continue, oder throw-Anweisung abgegeben oder wird eine Methode aufgerufen, die eine Ausnahme auslöst, so wird die ausstehende Kontrollabgabe verworfen und die neue verarbeitet, d.h. z.B.:
  - in einem finally-Block wird eine Ausnahme ausgelöst, so ersetzt diese Ausnahme eine evt. gerade in Verarbeitung befindliche Ausnahme
  - enthält der finally-Block eine return-Anweisung, dann kehrt die Methode normal zurück, auch wenn eine Ausnahme ausgelöst aber noch nicht behandelt wurde.

Beispiel:

Programm 12.5: try–finally (*Continue.java*)

```
// Continue in Schleifen

public class Continue{

  public static void main(String[] args) {

    for(int k = 0; k <= 9; k++){
      if (k % 2 == 0) continue;
      System.out.println("k = " + k);
    }
    int k = 0;
    while (k <= 9 ) {
      try {
        if (k % 2 == 0) continue;
        System.out.println("k = " + k);
      }
      finally {
        k++;
      }
    }
  }
}
```

12.2 Assertions

Ab Version 1.4!

- Zusicherungen sind Bedingungen, die an beliebigen Stellen einen korrekten Programmzustand beschreiben sollen: Schlüsselwort **assert**.
- Zusicherungen müssen beim Übersetzen aktiviert werden, ansonsten sind sie nichts anderes als Kommentar!
- Ist eine Zusicherung aktiviert und schlägt sie fehl, so wird eine Ausnahme erzeugt (**java.lang.AssertionError**)
- Die assert-Anweisung kann ein Argument (eine Bedingung) oder zwei Argumente (zusätzlich: Fehlernachricht an den Konstruktor *AssertionError()*) haben

Programm 12.6 (S. 302) zeigt ein Beispiel.

Programm 12.6: Assertion – Zusicherung (*MyAssert.java*)

```java
// Assertions

import java.io.*; // Alle Klassen im Paket java.io importieren

public class MyAssert {

  public static void main(String[] args) throws IOException{
    BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    for (;;) {
      System.out.print("Gib Zahl: ");
      String line = in.readLine();
      if( ( line == null ) || ( line.equals("quit") ) )
        break;
      try {
        int n = Integer.parseInt(line);
        System.out.println(n + "! = " + factorial(n));
      }
      catch(Exception e) {
        System.out.println("Fehler in Eingabe <<" + e.getMessage() + ">>"); }
    }
  }

  public static double factorial(int k) {
    assert (k >= 0) : "got " + ">>" + k + "<<";
    double fact = 1.0;
    while ( k > 1 ) { fact *= k; k--;}
    return fact;
  }
}
```

Übersetzung und Ausführung:

```
thales$ javac -source 1.4 MyAssert.java
thales$ java -ea:MyAssert MyAssert
Gib Zahl: 9
9! = 362880.0
Gib Zahl: -1
Exception in thread "main" java.lang.AssertionError: -1
        at MyAssert.factorial(MyAssert.java:30)
        at MyAssert.main(MyAssert.java:22)
thales$
thales$ java -ea:MyAssert
# Diagnose-Meldungen, u.a.
#  -ea[:<packagename>...|:<classname>]
thales$
thales$ java -ea MyAssert
Gib Zahl: -1
Exception in thread "main" java.lang.AssertionError: -1
        at MyAssert.factorial(MyAssert.java:30)
        at MyAssert.main(MyAssert.java:22)
thales$
```

- **Übersetzen von Zusicherungen:**

  ```
  javac -source 1.4 MyAssert.java
  ```

- Aktivieren von Zusicherungen
  - Zusicherungen sind zunächst deaktiviert (*disabled*) – Performance!
  - Anm.: (Gute) Zusicherungen sind auch guter Kommentar!
  - Die Aktivierung von Zusicherungen in allen Klassen außer Systemklassen erfolgt mit der Option **-ea**
  - Sollen auch die Assertions in den Systemklassen aktiviert werden: Option **-esa**
  - Deaktivieren von Zusicherungen: Option **-da**

12.3 File-IO

Die grundlegende Abstraktion zum **File-IO** sind **Streams**. Streams sind abstrakte Konzepte, mit welchen man Ein-und Ausgabekanäle modelliert. Man unterscheidet zwischen Output Streams (Ausgabeströmen), in welche man Bytes schreiben, und Input Streams (Eingabeströmen), von welchen man Bytes lesen kann.

Einen Stream kann man sich als (unendliche) Folge von Bytes vorstellen:

- Ausgabe eines Textes auf einen Bildschirm kann als die Folge der Bytes angesehen werden, aus denen der Text besteht.
- Beim Lesen einer Datei ist der Stream einfach die Folge der Bytes, aus denen die Datei besteht.
- Beim Versenden eines Bildes über ein Netzwerk besteht der Stream aus den Bytes, aus denen das Bild zusammengesetzt ist

Das Stream-Konzept ist stark abstrahierend von der Frage, woher oder wohin die Bytes strömen oder was sie eigentlich darstellen.

In Java sind die abstrakten Klassen **java.io.OutputStream** und **java.io.InputStream** definiert, welche Aus- bzw. Eingabe-Ströme repräsentieren. Diese Klassen sind abstrakt, weil das Konzept der strömenden Bytes ziemlich fundamental ist und nicht von der Quelle (File, Netzwerk, Tastatur ...) oder der Senke (File, Netzwerk, Bildschirm, ...) abhängt.

Einige Methoden dieser Klassen:

InputStream

`public void close();`	Schliessen des Eingabestroms und Freigeben von damit verbundenen Systemressourcen
`public abstract int read();`	Lesen eines Bytes. Rückgabewert: Das gelesene Byte, oder -1 wenn das Ende des Stroms erreicht wurde. Diese Methode wird in jeder abgeleiteten Klasse implementiert.
`public int read(byte[] byteArray);`	Lesen aller verfügbaren Bytes des Eingabestroms in ein Byte-Array. Rückgabewert: Die Anzahl gelesener Bytes, oder -1 wenn das Ende des Stroms erreicht wurde. Diese Methode wird in jeder abgeleiteten Klasse implementiert.
`public int skip(long numBytes);`	Überspringt die nächsten `numBytes` Bytes.

OutputStream

`public void close();`	wie InputStream
`public abstract void write(byte b);`	Schreiben eines einzelnen Bytes. Diese Methode wird in jeder abgeleiteten Klasse implementiert
`public void write(byte[] byteArray);`	Schreiben eines byte-Arrays
`public void flush();`	Leert Puffer, bewirkt also, dass ggf. gepufferte Bytes geschrieben / geschickt werden.

Für konkrete Datenquellen sind auch "konkrete" Klassen implementiert:

- **java.io.FileInputStream** für das Lesen von Dateien
- **java.io.ByteArrayInputStream** um Daten aus einem Array im Arbeitsspeicher so zu lesen, wie wenn sie aus einer Datei oder über das Netz kämen
- **java.io.FileOutputStream** für das Schreiben einer Datei
- **java.io.ByteArrayOutputStream** für das Schreiben in ein Array

Reader und Writer

Diese verleihen einem Eingabe- / Ausgabestrom eine gewisse Struktur.

Reader:

Wenn man zum Beispiel eine Textdatei lesen will, so ist man eher daran interessiert, ganze Zeilen als Strings einlesen zu können als ein Array von Bytes. Von einem *Reader*-Objekt kann man einzelne Chars, Arrays von Chars oder Strings beziehen. Viele von *Reader* abgeleitete Klassen sorgen dafür, dass ein Strom von Bytes in Unicode Zeichen umgewandelt wird.

Jeder *Reader* besitzt einen **InputStream**; die meisten *Reader* erwarten einen *InputStream* als Parameter für ihren Konstruktor. Dieser *InputStream* ist der eigentliche Strom, auf dem die Bytes hereinkommen, bevor sie vom *Reader* in Zeichen (Characters) umgewandelt werden.

Eine der praktischsten *Reader* ist die Klasse **BufferedReader**, welche unter anderem eine Methode `String readLine()` besitzt. Diese Methode liest die Bytes ihres Eingabestroms, setzt sie zu *Chars* zusammen und fügt sie nacheinander in einen String. Sobald ein Zeilenumbruch kommt, ist das Sammeln der Bytes vorbei und der String wird (ohne das *newline*) zurückgegeben. Um einen *BufferedReader* zu erzeugen, muss man dem Konstruktor ein anderes *Reader*-Objekt übergeben.

Writer:

In vielen Fällen ist es ziemlich umständlich, eine gewünschte Ausgabe in einen Strom von Bytes zu verwandeln. Ein *Writer*-Objekt ist in der Lage *Chars* und *Strings* in *Bytes* umzuwandeln (was z.B. bei Unicode Zeichen sehr nützlich ist).

Beim Schreiben vom Textdateien werden Zeilentrenner je nach Betriebssystem unterschiedlich (`\n` oder auch `\r\n`) repräsentiert. Ein *Writer* für eine Textdatei fügt automatisch den richtigen Zeilentrenner ein.

Jeder *Writer* besitzt einen *OutputStream*; die meisten *Writer* erwarten einen *OutputStream* als Parameter für ihren Konstruktor. Dieser *OutputStream* ist der eigentliche Strom, auf dem die Bytes hinausgeschrieben werden, die vom *Writer* aus den *Chars* und *Strings* erzeugt worden sind.

Einer der praktischsten *Writer* ist die Klasse **PrintWriter**, welche für jeden primitiven Datentyp eine entsprechende **println()**-Methode besitzt. Diese konvertieren den Parameter in einen String; dieser wird in eine Folge von Bytes umgewandelt und zusätzlich wird noch ein Zeilentrenner angefügt.

Um einen *PrintWriter* zu erzeugen, muss man dem Konstruktor ein anderes Writer-Objekt übergeben.

Beispiele:

Wir hatten bisher mit der speziell präparierten Klasse *Read* aus dem Paket *IOulm* gearbeitet. Diese basiert natürlich auch auf dem oben skizzierten Konzept.

Beim Lesen von Zeichen-Strömen geht man normalerweise so vor, dass man zunächst eine Instanz eines *InputStream* (d.h. einer von *InputStream* abgeleiteten Klasse) erzeugt. Danach erzeugt man eine Instanz einer von *Reader* abgeleiteten Klasse. Diesen *Reader* benutzt man schliesslich für das Erstellen eines *BufferedReader*.

Programm 12.7: Lesen von stdin und schreiben nach stdout (*Read1.java*)

```java
1  import java.io.*;
2  
3  public class Read1{
4  
5    public static void main(String[] args) {
6           String inline;
7           InputStreamReader instream = new InputStreamReader(System.in);
8           BufferedReader buff = new BufferedReader(instream);
9           try {
10              while ( (inline = buff.readLine()) != null)
11                      System.out.println(inline);
12  
13              }
14              catch(Exception e) {
15                      System.err.println(e.getMessage());
16                      System.exit(1);
17              }
18   }
19 }
```

Erläuterungen zu Programm 12.7, S. 306:

- Das Objekt **System.in** ist ein *InputStream*, die Objekte **System.out** und **System.err** und sind beide *OutputStreams*, die wie man es von Unix her kennt, standardmäßig vom System bereits geöffnet und vorbereitet sind.
- Aus dem *InputStream* machen wir jetzt einen *InputStreamReader* (ein *InputStreamReader* konvertiert Bytes zu Chars): Zeile 7
- Um jetzt auf angenehme Weise Zeilen von stdin lesen zu können, erzeugen wir einen BufferedReader: Zeile 8
- Um die nächste Zeile zu lesen, verwenden wir die *readLine()*-Methode: Zeile 10

  NB: Die Methode *readLine()* blockiert, bis die Zeile mit Return abgeschlossen wird. Erst dann wird der String mit den eingegebenen Zeichen gefüllt. Der null-Referenz wird bei Ende der Eingabe geliefert.
- Die Zeilen 7 und 8 lassen sich auch zusammenfassen zu

  ```java
  BufferedReader buff = new BufferedReader(new InputStreamReader(System.in));
  ```
- Ausgabeseitig ist nichts anzumerken!

Lesen aus einer Textdatei:

Programm 12.8: Lesen aus einer Textdatei (*ReadFile.java*)

```java
import java.io.*;

public class ReadFile {

  public static void main(String[] args) {
      String inline, infile;
      FileInputStream fstream = null;
      if (args.length == 0){
            System.err.println("usage: java ReadFile filename"); System.exit(1);
      }
      infile = args[0];
      try { fstream = new FileInputStream(infile);
      } catch (FileNotFoundException fnfe) {
           System.err.println(fnfe.getMessage()); System.exit(2);
      }

      BufferedReader buff = new BufferedReader(new InputStreamReader(fstream));
      try {
          while ((inline = buff.readLine()) != null) System.out.println(inline);
      } catch(Exception e) {
          System.err.println(e.getMessage()); System.exit(1);
      }
  }
}
```

Ausführung:

```
euclid$ java ReadFile
usage: java ReadFile filename
euclid$ java ReadFile xxx
xxx (No such file or directory)
euclid$ java ReadFile einText
Read1.class
Read1.java
einText
euclid$ cat einText
Read1.class
Read1.java
einText
euclid$ java ReadFile einText1
einText1 (Permission denied)
euclid$ ll einText1
--w-r--r--    1 swg      users          31 2006-04-18 19:56 einText1
euclid$
```

Für das Lesen von Dateien gibt es die von InputStreamReader abgeleitete Klasse **FileReader**, die sich direkt über einen Dateinamens erzeugen lässt – Programm 12.9, S. 308.

Programm 12.9: Lesen aus einer Textdatei – vereinfacht (*ReadFile1.java*)

```
1 import java.io.*;
2
3 public class ReadFile1 {
4
5   public static void main(String[] args) {
6       String inline, infile;
7       FileReader fr = null;
8       if (args.length == 0){
9           System.err.println("usage: java ReadFile filename"); System.exit(1);
10      }
11      infile = args[0];
12      try { fr = new FileReader(infile);
13      } catch (FileNotFoundException fnfe) {
14          System.err.println(fnfe.getMessage()); System.exit(2);
15      }
16      BufferedReader buff = new BufferedReader(fr);
17      try {
18          while ((inline = buff.readLine()) != null) System.out.println(inline);
19      }
20      catch(Exception e) {
21          System.err.println(e.getMessage()); System.exit(1);
22      }
23   }
24 }
```

Erläuterungen:

- Da ein *FileReader* ein *InputStreamReader* ist, können wir direkt einen *BufferedReader* herstellen – Zeile 16.

Beim Schreiben von Zeichen-Strömen (Char) geht man normalerweise so vor, dass man zunächst eine Instanz eines **FileOutputStream** (einer von *OutputStream* abgeleiteten Klasse) erzeugt. Danach erzeugt man eine Instanz der Klasse **OutputStreamWriter**, einer von **Writer** abgeleiteten Klasse. Diesen *Writer* benutzt man schliesslich für das Erstellen eines **PrintWriter**s.

Dies kann über die Klasse **FileWriter**, abgeleitet von **OutputStreamWriter**, die sich direkt mit Hilfe eines Dateinamens erzeugen lässt, weider abkürzen (siehe Programm **??**, S. **??**.

Programm 12.10: Lesen aus einer und Schreiben in eine Textdatei (*ReadWriteFile.java*)

```java
import java.io.*;

public class ReadWriteFile {

  public static void main(String[] args) {
      String inline, infile, outfile;
      FileReader fr = null; FileWriter fw = null;
      if (args.length != 2){
            System.err.println("usage: java ReadWriteFile infile outfile");
            System.exit(1);
      }
      infile = args[0]; outfile = args[1];
      try { fr = new FileReader(infile);
      } catch (IOException fnfe) {
           System.err.println(fnfe.getMessage()); System.exit(2);
      }
      BufferedReader buff = new BufferedReader(fr);

      try {
        fw = new FileWriter(outfile);
      } catch (IOException fnfe) {
           System.err.println(fnfe.getMessage()); System.exit(3);
      }
      PrintWriter pw = new PrintWriter(fw);

      int i = 0;
      try {
          while ( (inline = buff.readLine()) != null) {
              i++; pw.println("" + i + ": " + inline);
          }
      }
      catch(Exception e) {
          System.err.println(e.getMessage()); System.exit(1);
      } finally {
          pw.close();
      }
  }
}
```

```
euclid$  ls *Text
einText
euclid$  cat einText
Read1.class
Read1.java
einText
euclid$  java ReadWriteFile
usage: java ReadWriteFile infile outfile
euclid$  java ReadWriteFile einText
usage: java ReadWriteFile infile outfile
euclid$  java ReadWriteFile einText ausText
euclid$  cat ausText
1: Read1.class
2: Read1.java
3: einText
euclid$  chmod -w ausText
euclid$  java ReadWriteFile einText ausText
ausText (Permission denied)
euclid$  chmod +w ausText
euclid$  java ReadWriteFile einText ausText
euclid$  cat ausText
1: Read1.class
2: Read1.java
3: einText
euclid$
```

Anm.: Damit sind auch **System.in**, **System.out** und **System.err** klar

- *System.in* ist eine Instanz eines **InputStream** (automatisch erzeigt)
- *System.out* ist ein (automatisch erzeugtes) **PrintStream**-Objekt und besitzt deshalb die diversen Methoden für die Ausgabe von primitiven Datentypen wie z.B. die verschiedenen verb|println()|-Varianten
- für *System.err* gilt dasselbe

Die Klasse **FileWriter** besitzt noch einen Konstruktor, mit dem man eine Datei zum Schreiben im Sinne von hinten Anfügen öffnen kann:

```
FileWriter(File file, boolean append)
          Constructs a FileWriter object given a File object.
```