Fortgeschrittene Programmierung mit Java

Dr. E.Gutknecht

Vorlesungs-Skriptum Sommersemester 2002

Inhaltsverzeichnis

1 Rekursion **5**
- 1.1 Grundlagen der Rekursion . . . 5
- 1.2 Beispiele . . . 7
- 1.3 Übungen . . . 15

2 Suchen und Sortieren **17**
- 2.1 Schleifeninvarianten . . . 18
- 2.2 Lineare Suche . . . 20
- 2.3 Binäre Suche . . . 20
- 2.4 Elementare Sortier-Algorithmen . . . 24
- 2.5 Der Quicksort-Algorithmus . . . 27
- 2.6 Sortierzeiten . . . 31
- 2.7 Mischen (Merge) . . . 33
- 2.8 Gruppenverarbeitungen . . . 34
- 2.9 Übungen . . . 36

3 Dynamische Datenstrukturen **41**
- 3.1 Verkettete Listen . . . 43
- 3.2 Binäre Bäume . . . 50
- 3.3 Übungen . . . 58

4 Vererbung und Interfaces **63**
- 4.1 Klassenerweiterungen . . . 63
- 4.2 Polymorphismus . . . 70
- 4.3 Allgemeine Klassen für verkettete Listen . . . 72
- 4.4 Abstrakte Klassen und Methoden . . . 75
- 4.5 Die Klasse ‘Object’ . . . 76
- 4.6 Interfaces . . . 79
- 4.7 Eine allgemeine Klasse für binäre Bäume . . . 85
- 4.8 Interface für vergleichbare Objekte . . . 87

4.9 Übungen 90

5 Exception-Handling 91
5.1 Das Konzept der Exceptions 91
5.2 Exception-Handlers 94

II Klassische Anwendungen 97

1 Mandelbrot-Fraktale 99

2 Der Lorenz-Attraktor 109

3 Das 8-Damen Problem 113

4 Die Türme von Hanoi 115

5 Das Raytracing-Verfahren 117
5.1 Grundprinzip 117
5.2 Die Klasse Vek3d 119
5.3 Durchstosspunkte 120
5.4 Raytracing Bild des Mondes 123
5.5 Bild einer spiegelnden Kugel 127
5.6 Boden mit Schatten 130

6 Kürzeste Wege 131

Index 135

Kapitel 1

Rekursion

Wissen ist Macht.

– F.Bacon

Rekursion ist eine Programmiertechnik, mit der viele Probleme ausserordentlich elegant gelöst werden können. Es geht dabei um Methoden, die sich selber aufrufen zur Lösung einer reduzierten Version des ursprünglichen Problems.

1.1 Grundlagen der Rekursion

Eine Methode heisst *rekursiv*, wenn sie mindestens einen Aufruf von sich selbst enthält.

Natürlich muss dieser Aufruf von sich selbst mit einer Bedingung verknüpft sein, die nach endlich vielen Schritten nicht mehr zutrifft.

Wie wir sehen werden, können viele Probleme mittels Rekursion anstelle von Schleifen gelöst werden. Lösungen, die keine Rekursion verwenden, nennt man auch *iterative* Lösungen (Iteration = Wiederholung).

Der Call Stack

Der Call Stack ist ein spezieller Speicherbereich zur Abhandlung von allen Methoden-Aufrufen. Er ist wichtig für das Verständnis von rekursiven Methoden.

Bei jedem Aufruf einer Methode wird im Call Stack ein sogenanntes *Call Frame* für den Aufruf angelegt, in welchem die folgenden Daten abgespeichert werden:

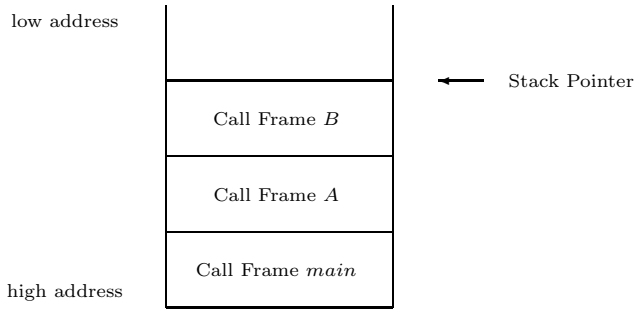
Call Frame

- Parameter der Methode
- lokale Variablen der Methode
- Rücksprung-Adresse

Als Beispiel betrachten wir eine Folge von drei verschachtelten Methoden-Aufrufen:

$$main \quad \rightarrow \quad \text{Methode } A \quad \rightarrow \quad \text{Methode } B$$

Während dem Ablauf der Methode B befinden sich drei Call Frames auf dem Stack:



Die Call Frames werden wie auf einem Teller-Stapel aufgestapelt, daher kommt der Name Stack (= Stapel). Dieses Prinzip garantiert eine effiziente Speicherverwaltung für Methoden-Aufrufe: Wenn eine Methode endet, wird das oberste Frame auf dem Stack gelöscht. Dazu muss nur der Stack Pointer modifiziert werden.

Der Call Stack wird von der höchsten Speicheradresse her gegen die tieferen Adressen aufgefüllt, was jedoch ohne weitere Bedeutung ist.

Folgerung:

Wenn eine Methode sich selber aufruft, wird für den zweiten Aufruf ein neues Frame angelegt, mit einem neuen Satz von lokalen Variablen und Parametern.

Dies ist die Grundlage der Rekursions-Technik, die jetzt an konkreten Beispielen mit aufsteigendem Schwierigkeitsgrad weiter eingeführt wird. Die ersten 3 Beispiele dienen nur der Erklärung der Rekursions-Technik, sie könnten genau so gut ohne Rekursion realisiert werden.

▶ *Merke:*

Wenn ein Problem ohne Rekursion gelöst werden kann, sollte auf Rekursion verzichtet werden, weil ohne Rekursion der Stack weniger belastet wird.

1.2 Beispiele

1.2.1 Die Fakultäts-Funktion

Die *Fakultät* einer positiven ganzen Zahl n ist die Zahl $n!$ ('n Fakultät'), definiert durch

$$n! = 1 \cdot 2 \cdot \ \ldots \ \cdot n$$

Rekursive Definition:

$$1! = 1 \quad \text{und} \quad n! = n \cdot (n-1)! \quad \text{für } n > 1$$

Rekursive Funktion:

```
static int fak(int n)
{
  if ( n <= 1 )
    return 1;
  else
    return n * fak(n−1);                    // rekursiver Aufruf
}
```

Erklärungen:

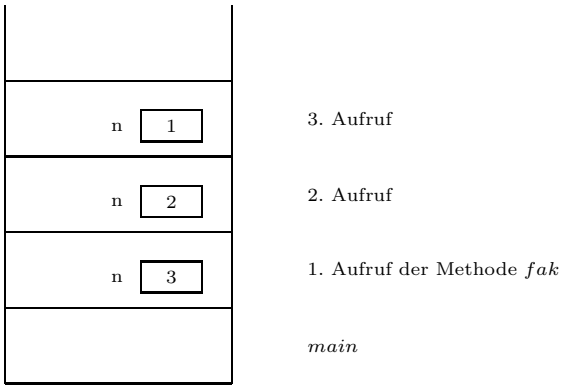
Bei der Auswertung des Ausdrucks '`n * fak(n−1)`' wird die laufende Methode unterbrochen und mit Parameter $n-1$ neu aufgerufen. Die Funktionsweise wird mit einem konkreten Aufruf-Beispiel klarer:

```
int resultat = fak(3);
```

Aufruf-Kette und Rückgabe-Werte:

fak(3)	$\rightarrow$	fak(2)	$\rightarrow$	fak(1)
$3 \cdot 2$	$\leftarrow$	$2 \cdot 1$	$\leftarrow$	1

Call Stack:



Der dritte rekursive Aufruf erhält als Parameter $n = 1$. Dadurch verzweigt das 'if' in den ersten Fall, der 1 als Resultat zurückgibt und die Funktion beendet.

Dann geht es bei der Rücksprung-Adresse weiter, d.h. bei der Berechnung des Produktes 'n * fak(n−1)' mit dem nächsten Frame ($n = 2$). Das Produkt erhält den Wert $2 \cdot 1$, dann erfolgt wieder ein 'return'. So geht es zurück, bis alle Aufrufe beendet sind.

1.2.2 Eine rekursive Verarbeitung

Gegeben ist die folgende rekursive Methode:

```
static void verarb()
{ int wert;
  InOut.print("Wert eingeben (0=Ende): ");
  wert = InOut.getInt();
  if ( wert > 0 )
  { verarb();
    InOut.println(wert);
  }
}
```

Was macht die Methode, wenn sie in der 'main'-Methode einer Applikation aufgerufen wird ?

```
static public void main(String[ ] args)
{ verarb();
}
```

Zeichnen Sie den Call Stack mit den Frames und der jeweiligen lokalen Variablen *wert*. Überprüfen Sie Ihre Antwort mit dem Computer.

1.2.3 Die Fibonacci-Funktion

Die Fibonacci-Folge ist die Zahlenfolge

$$a_0 = 1, \quad a_1 = 1, \quad a_2 = 2, \quad a_3 = 3, \quad a_4 = 5, \quad a_5 = 8, \quad \ldots$$

Jedes Glied, ab dem dritten, ist die Summe der beiden vorangehenden.

Rekursive Definition:

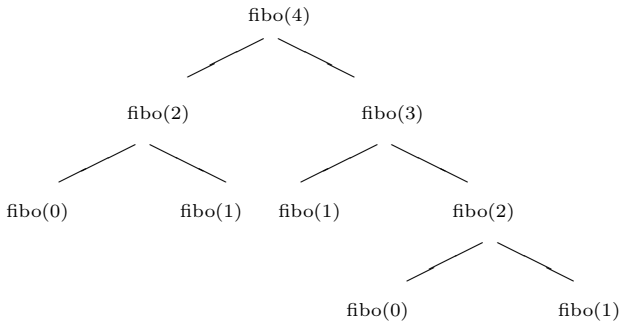
$$a_0 = 1, \quad a_1 = 1, \qquad \text{und} \qquad a_n = a_{n-2} + a_{n-1} \quad \text{für} \quad n >= 2$$

Rekursive Funktion:

Die folgende Funktion 'fibo' berechnet zu einem gegebenen n das Glied a_n der Fibonacci-Folge.

```
static int fibo(int n)
{ if ( n <= 1 )
    return 1;
  else
    return fibo(n−2) + fibo(n−1);
}
```

Die Funktion sieht sehr kompakt aus, sie führt jedoch schon für kleine n zu einem grossen Aufwand, z.B. sieht der Ablauf für $n = 4$ schematisch folgendermassen aus:



Dieses Beispiel ist ein Missbrauch der Rekursion, eine iterative Lösung ist viel weniger aufwendig.

1.2.4 Dezimal- und Dualziffern

Die Zehner-Ziffern einer ganzen Zahl $n \geq 0$, z.B. $n = 123$, können leicht bestimmt werden: Die tiefste Ziffer (3) ist der Rest von n modulo 10. Die weiteren Ziffern ergeben sich analog aus dem reduzierten Wert $n/10$ (ganzzahlige Division ohne Rest).

Wenn der Algorithmus mit einer Schleife implementiert wird, müssen die Ziffern zwischengespeichert werden, damit sie am Schluss in der richtigen Reihenfolge ausgegeben werden können. Rekursiv geht es eleganter.

Entwickeln Sie eine rekursive Methode

```
static void ziffern(int n)
```

welche die Zehnerziffern von n in der richtigen Reihenfolge auf den Bildschirm ausgibt. Die Lösung ist einfach: Die Methode ruft sich rekursiv auf für $n/10$ (wenn $n > 0$ ist), dann gibt sie die tiefste Dualziffer von n aus.

Analog erhält man die Dualziffern, wenn man 10 ersetzt durch 2, d.h. Reste modulo 2 berechnet und jeweils durch 2 dividiert. Z.B. für $n = 12$:

```
12 modulo 2 :  0      (tiefste Dualziffer)
 6 modulo 2 :  0
 3 module 2 :  1
 1 modulo 2 :  1
```

Dualziffern: 1100

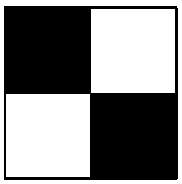
Probe: $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 0 \cdot 1 = 12$

Entwickeln Sie eine rekursive Methode, welche die Dualziffern in der richtigen Reihenfolge ausgibt.

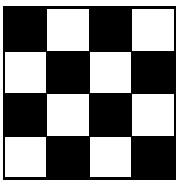
△ **Uebung:** Lösen Sie die Aufgabe 1.3.1

1.2.5 Schachbrett-Muster

Wir betrachten eine Folge von Schachbrett-Mustern:



Stufe 1



2

. . .

Die Folge kann rekursiv definiert werden: Das Muster der Stufe n besteht aus 4 verkleinerten Mustern der Stufe $n-1$. Entwickeln Sie eine rekursive Methode, welche das Muster der Stufe n und Grösse w zeichnet. Ansatz:

```
void zeichneMuster(Graphics g, int n,                // Stufe n
                   int left, int top, int w)         // Groesse w
{ if ( n > 1 )
  { zeichneMuster(g, n−1, left, top, w/2);
    zeichneMuster(g, n−1, left+w/2, top, w/2 );
    zeichneMuster(g, n−1, left, top+w/2, w/2 );
    zeichneMuster(g, n−1, left+w/2, top+w/2, w/2 );
  }
  else
  {
    zeichne das Grundmuster (4 Quadrate, Kante w/2)
  }
}
```

△ **Uebung:** Lösen Sie die Aufgabe 1.3.2

1.2.6 Permutationen

Bei diesem Problem sind wir echt auf Rekursion angewiesen. Eine iterative Lösung läuft darauf hinaus, dass man ebenfalls mit einem (selber verwalteten) Stack arbeiten muss.

Gegeben sind n Elemente, z.B. 3 Character 'a', 'b' und 'c'. Es sollen alle Permutationen, d.h. Anordnungen der Elemente auf den Bildschirm ausgegeben werden.

Wir bilden alle Permutationen mit ‘a’ an erster Stelle, dann alle mit ‘b’ an erster Stelle und schliesslich alle mit ‘c’ an erster Stelle:

a b c
a c b
b a c
b c a
c b a
c a b

Die Lösung kann rekursiv formuliert werden:

1. Halte das erste Element fest und bilde alle Permutationen der übrigen $n-1$.
2. Vertausche das erste Element mit dem zweiten, dann halte wieder das erste Element fest und permutiere die übrigen.

...

n. Vertausche das erste Element mit dem n-ten, dann halte wieder das erste Element fest und permutiere die übrigen.

Dabei ist zu beachten, dass nach jeder der n Teilaufgaben die Elemente wieder in der ursprünglichen Reihenfolge vorliegen müssen.

Realisierung

Die gegebenen Character werden in einem globalen Character-Array gespeichert und dann direkt in diesem Array permutiert:

```
static char[ ] c = { 'a', 'b', 'c' }
```

Wir setzen eine rekursive Methode an, welche die Permutationen produzieren soll:

```
static void permutiere(int iStart)
```

Der Parameter *iStart* gibt an, mit welchem Element im Array c die Methode beginnen soll. Mit den Elementen 0 bis $iStart - 1$ hat die Methode nichts zu tun.

Die Methode löst in einer Schleife die oben aufgeführten Teilaufgaben. Dabei ruft sie sich auch rekursiv auf.

Versuchen Sie die Methode selber zu entwickeln, bevor sie die folgende Lösung anschauen.

Vollständige Lösung:

```
// ———————— Permutationen ————————————————————————————————
public class Permut
{
  static char[ ] c = { 'a', 'b', 'c'} ;

  static void swap(int i, int j)                    // Vertauschung c[i] und c[j]
  { char tmp = c[i];
    c[i] = c[j];
    c[j] = tmp;
  }

  static void permutiere(int iStart)
  { if ( iStart == c.length−1 )
      System.out.println( new String(c) );          // Zeile ausgeben
    else
      for ( int i=iStart; i < c.length; i++)
      { swap(iStart, i);
        permutiere(iStart+1);
        swap(iStart, i);                            // Rueckstellung
      }
    }

    static public void main(String[] args)
    {
      permutiere(0);
    }
  }
```

△ **Uebung:** Lösen Sie die Aufgabe 1.3.3

1.2.7 Rekursive Auflistung eines Directories

Ein weiteres typisches Beispiel zum Einsatz von Rekursion ist die Auflistung eines Directories mit allen Unterverzeichnissen.

Für jedes Unterverzeichnis ist die ursprüngliche Aufgabe wieder auszuführen, wofür sich Rekursion aufdrängt.

Die Klasse 'File'

Dia Java Library stellt eine Klasse 'File' (Package 'java.io') zur Verfügung, für diverse Aktionen mit Files und Directories:

- Abklärung, ob ein File mit einem bestimmten Namen existiert
- Abfrage aller File-Namen, die in einem Directory katalogisiert sind

Konstruktoren und Methoden:

- File(String path)

  Erzeugt ein Objekt der Klasse, für das File mit dem angegebenen Namen. Der Name kann als vollständiger oder relativer Pfad angegeben werden. Dabei sind die Konventionen des Betriebssystems zu berücksichtigen:

  ```
  File f = new File("c:\\examples\\Main.java");
  File g = new File("MyFirst.java");
  ```

  Man beachte, dass das Zeichen '\' in Java als Escape-Sequenz '\\' (wie '\n' usw.) geschrieben werden muss.

- File(String dirPath, String name)

  Zweiter Konstruktor mit separater Angabe des Directory-Pfades des Files.

- boolean exists()

  Prüft, ob das File tatsächlich existiert.

- String getAbsolutePath()

  Erzeugt den vollständigen Pfad des Files

- boolean isDirectory()

  Bestimmt, ob das File ein Directory ist

- String[ ] list()

  Erstellt eine Liste der Files eines Directories (String-Array).

Beispiel:

Auflistung eines Directories (ohne Unterverzeichnisse)

```
import java.io.*;
public class ListDirectory
{

  static void list(String dirName)
  { File dir = new File(dirName);
    String[ ] names = dir.list();
    for (int i=0; i < names.length; i++)
      System.out.println(names[i]);
  }
```

```
  static public void main(String[ ] args)
  {
    list(args[0]);
  }
}
```

Erweitern Sie das Programm so, dass es auch Unterverzeichnisse auflistet. Dazu muss für jedes Element des Arrays 'names' ein File-Objekt erzeugt werden, mit dem Konstruktor:

```
File(String dirPath, String name)
```

wobei `dirPath=dir.getAbsolutePath()`. Dann wird geprüft, ob das File ein Directory ist (Methode 'isDirectory'), und gegebenenfalls wird dieses Unterverzeichnis mittels Rekursion verarbeitet.

1.3 Übungen

1.3.1 Grösster gemeinsamer Teiler

Entwickeln Sie eine rekursive Methode

```
int ggt(int a, int b)
```

zur Berechnung des grössten gemeinsamen Teilers unter Verwendung der Tatsache, dass der ggT zweier Zahlen a und b gleich dem ggT von $a - b$ und b ist, wenn $a > b$.

1.3.2 Das Sierpinski-Dreieck

Wir betrachten eine Folge von Dreiecksmustern

Stufe 1 2 3

Erstellen Sie eine rekursive Methode 'zeichneFigur', die die Figur einer bestimmten Stufe zu vorgegebenen Ecken A, B und C zeichnet:

```
void zeichneFigur(Graphics g, int stufe, int xA, int yA,
                                         int xB, int yB,
                                         int xC, int yC)
```

Man beachte, dass die Figur der Stufe n die Zusammensetzung von 3 Figuren der Stufe $n-1$ ist, wobei neben den gegebenen Punkten die Seitenmittelpunkte als Ecken hinzukommen.

1.3.3 Würfel-Kombinationen

Entwickeln Sie eine rekursive Methode, welche alle möglichen Augenkombinationen von n Würfeln ausgibt, z.B. für $n = 3$:

1 1 1
1 1 2
1 1 3
...

1.3.4 Das Münzenproblem

Gegeben sind n Münzen mit beliebigen ganzzahligen Werten

$$w_1,\ w_2,\ \ldots\ w_n$$

(Die Werte müssen nicht alle verschieden sein.)

Erstellen Sie eine rekursive Methode, welche zu einem beliebig vorgegebenen Betrag s eine Auswahl der Münzen sucht, deren Summe gleich s ist. Wenn es eine Lösung gibt, soll die Methode die betreffenden Münzen auf den Bildschirm ausgeben und als Resultat *true* zurückgeben, sonst gibt sie *false* als Resultat zurück.

Kapitel 2

Suchen und Sortieren

Suchen und Sortieren sind wichtige Grundverarbeitungen der Informatik. Als typisches Beispiel betrachten eine Suche im Bestand der Postleitzahlen der Schweizerischen Post, den wir auch für unsere Übungen verwenden wollen:

```
...
5116 AG Schinznach Bad
5200 AG Brugg AG
5200 AG Koenigsfelden
5200 AG Brugg AG 1
5200 AG Brugg AG 3
5200 AG Lauffohr
5200 AG Brugg AG Kaserne
5201 AG Brugg AG
5210 AG Windisch
5212 AG Hausen b. Brugg
5213 AG Villnachern
5222 AG Umiken
...
```

Es sind ca. 8000 Datensätze. Wir nehmen an, die Daten seien in zwei Arrays eingelesen:

```
int[ ] plz;
String[ ] ortDaten;
```

Bei einer Abfrage mit gegebener Postleitzahl wird im Array 'plz' nach dieser gesucht. Mit dem zugehörigen Index können dann die Ortdaten aus dem zweiten Array gelesen werden.

Wir können uns im folgenden daher auf 'int'-Arrays beschränken. Verallgemeinerungen der Algorithmen auf andere Arrays sind offensichtlich.

Übersicht:

Wir führen in diesem Kapitel die folgenden Algorithmen ein:

- Suchen
  - Lineare Suche
  - Binäre Suche in einem sortierten Array
- Sortieren

  Für das Sortieren eines Arrays gibt es viele Algorithmen. Wir führen zwei elementare und den bekanntesten fortgeschrittenen Algorithmus ein:
  - Sortieren durch direktes Auswählen (Select-Sort)
  - Bubble-Sort
  - Der Quicksort-Algorithmus von C.A. Hoare

Für die Erklärung und die Verifikation der Algorithmen führen wir die von E. Dijkstra entwickelte Methode der *Schleifeninvarianten* ein.

2.1 Schleifeninvarianten

Der Formulismus der Schleifeninvarianten von E. Dijkstra ist ein wichtiges Hilfsmittel für den Entwurf und die Verifikation von Algorithmen.

Wir führen den Formulismus am Beispiel eines schnellen Algorithmus zur Berechnung einer Potenz a^n für ganzzahlige $n \geq 0$ ein.

Algorithmus:

$$2^{10} \;=\; 4^5 \;=\; 4 \cdot 4^4 \;=\; 4 \cdot 16^2 \;=\; 4 \cdot 256^1 \;=\; 1024$$

Bei jedem Schritt wird eine Fallunterscheidung durchgeführt. Wenn der Exponent gerade ist, wird die Basis quadriert und der Exponent halbiert, sonst wird ein Faktor abgespalten und der Exponent um 1 reduziert.

```
static double berechnePotenz(double a, int n)
{ double faktor = 1;
  while ( n > 0 )
    if ( n % 2 == 0 )                    // n gerade ?
    { a *= a;
      n /= 2;
    }
    else
    { faktor *= a;
      n--;
    }
  return faktor;
}
```

Das Verfahren beruht darauf, dass das Produkt $faktor \cdot a^n$ während der Schleife konstant bleibt. Da am Anfang der Faktor gleich 1 gesetzt wird, ist das Produkt am Anfang und damit permanent gleich der gesuchten Potenz.

Definition

Eine *Schleifeninvariante* ist eine Aussage über Variablen eines Programmes, die am Anfang einer Schleife, sowie nach jedem Durchlauf der Schleife erfüllt ist.

Eine Aussage, die beim Beginn einer Schleife erfüllt ist, nennt man auch eine *Vorbedingung* (Precondition) für die Schleife.

Im Potenz-Algorithmus ist die Aussage

$$gesuchte\ Potenz = faktor \cdot a^n$$

eine Schleifeninvariante. Am Ende der Schleife hat man zusätzlich

$$n = 0$$

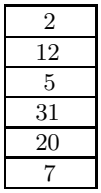
Die beiden Aussagen zusammen ergeben (wegen $a^o = 1$) :

$$gesuchte\ Potenz = faktor.$$

Damit ist die Korrektheit des Algorithmus bewiesen.

2.2 Lineare Suche

Bei einer linearen Suche in einem Array werden die Elemente der Reihe nach (sequentiell) durchlaufen, bis der Suchwert gefunden wurde, oder alle Elemente geprüft worden sind.



Suchwert: 31

Java-Methode:

```
int search(int[ ] a, int suchwert)          // Lineare Suche
{  for (int i=0; i < a.length; i++ )
      if ( a[i] == suchwert )
         return i;                          // gesuchte Position
   return −1;                               // not found
}
```

Wenn der Suchwert mehrfach vorkommt, liefert die Methode den Index des ersten Vorkommens, wenn er nicht vorkommt, ist das Resultat der Methode gleich -1.

△ **Uebung:** Lösen Sie die Aufgabe 2.9.1

2.3 Binäre Suche

Das binäre Suchen ist ein effizientes Verfahren für *sortierte* Arrays. Es funktioniert nach dem folgenden Prinzip:

- Der Wert des Elementes in der Mitte des Arrays wird mit dem Suchwert verglichen.
- Wenn er grösser als der Suchwert ist, wird die Suche im vorderen Teil des Arrays fortgesetzt, wenn er kleiner ist, im hinteren Teil.

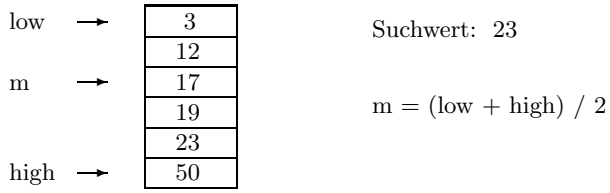
So wird bei jedem Schritt der Suchbereich halbiert. Nach diesem Prinzip sucht man auch manuell in einem sortierten Bestand, z.B. im Telefon-Buch.

Das Datenfeld, nach dem der Array sortiert ist, nennt man den Schlüssel oder *Key*. Im folgenden spielt es eine Rolle, ob die Keys im Array eindeutig sind oder nicht. Von eindeutigen Keys spricht man, wenn keiner der Werte mehr als einmal vorkommt. Andernfalls spricht man von mehrdeutigen Keys.

Key

A) Binäre Suche bei eindeutigen Keys

Wir führen Indizes *low* und *high* zur Begrenzung des Suchbereiches ein, sowie einen Index m für das Element in der Mitte:



Entwickeln Sie eine Methode 'binsearch', die als Resultat den Index des gesuchten Elementes liefert. Wenn das gesuchte Element nicht vorkommt, ist das Resultat gleich -1.

Ansatz:

```
int binsearch(int[ ] a, int suchwert)        // Array a muss sortiert sein
{ int low = 0;                               // untere Grenze des Suchbereichs
  int high = a.length−1;                     // obere Grenze des Suchbereichs
  int m;
  while ( low <= high )
  {
    Vergleiche das Element in der Mitte mit dem Suchwert.
    Wenn es ungleich dem Suchwert ist, reduziere die
    Grenze low bzw. high.
  }
  return −1;                                 // not found
}
```

Vollständige Lösung:

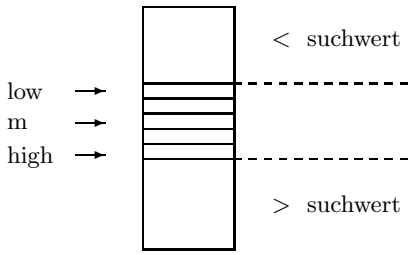
```
int binsearch(int[ ] a, int suchwert)      // Array a muss sortiert sein
{  int low = 0;                            // untere Grenze des Suchbereichs
   int high = a.length−1;                  // obere Grenze des Suchbereichs
   int m;
   while ( low <= high )
   {  m = (low + high) / 2;                // mittlerer Index
      if ( a[m] > suchwert )
         high = m−1;                       // weiter im vorderen Teil
      else if ( a[m] < suchwert )
         low = m+1;                        // weiter im hinteren Teil
      else
         return m;                         // gesuchte Position
   }
   return −1;                              // not found
}
```

Schleifeninvariante

Die zugrunde liegende Schleifeninvariante lautet:

a) für alle $i < low$ gilt $a[i] < suchwert$ und
b) für alle $j > high$ gilt $a[j] > suchwert$,

d.h. graphisch dargestellt:



Wenn die Schleife zuende läuft (d.h. der 'return' nie zum Einsatz kommt), ist am Schluss $low > high$. Daraus folgt mit der Schleifeninvariante sofort, dass alle Werte des Arrays kleiner bzw. grösser als der Suchwert sind, d.h. der Suchwert nicht vorkommt.

B) Binäre Suche bei mehrdeutigen Keys

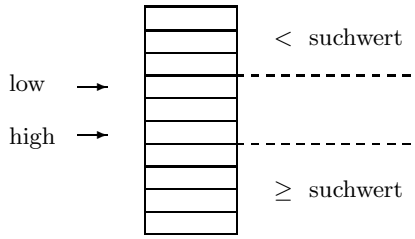
Im Falle von mehrdeutigen Keys soll das Resultat einer Suche der Index des *ersten* Elementes mit dem gesuchten Wert sein.

Dies kann mit einer kleinen Modifikation der obigen Methode 'binsearch' erreicht werden: wenn ein Element mit dem gewünschten Key gefunden wird, endet die Suche nicht, sondern läuft im vorderen Teil des Arrays weiter (Suche nach weiteren Vorkommen).

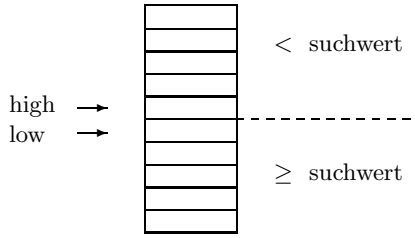
Im obigen Code kann also beim Vergleich in der Schleife der Fall der Gleichheit zum vorangehenden Fall hinzugenommen werden:

```
int binsearch2(int[ ] a, int suchwert)      // Array a muss sortiert sein
{ int low = 0;                              // untere Grenze des Suchbereichs
  int high = a.length−1;                    // obere Grenze des Suchbereichs
  int m;
  while ( low <= high )
  { m = (low + high) / 2;                   // mittlerer Index
    if ( a[m] >= suchwert )
      high = m−1;                           // weiter im vorderen Teil
    else
      low = m+1;                            // weiter im hinteren Teil
  }
  return low;                               // gesuchte Position
}
```

Schleifeninvariante:



Am Ende der Schleife ist $low > high$, d.h. zusammen mit der Schleifeninvariante hat man:



Resultat:

Der Index *low* zeigt am Ende auf das erste Element grösser oder gleich dem Suchwert, d.h. es sind die folgenden Fälle möglich:

a) Wenn der Suchwert im Array vorkommt, zeigt *low* auf das erste Vorkommen des Suchwertes.

b) Wenn der Suchwert nicht vorkommt, zeigt *low* entweder auf das erste Element grösser als der Suchwert oder ist gleich $a.length$, wenn alle Elemente echt kleiner als der Suchwert sind.

Der Fall b) kann ebenfalls von Bedeutung sein, wenn man die Einfügeposition für ein Element sucht.

△ **Uebung:** Lösen Sie die Aufgabe 2.9.2

2.4 Elementare Sortier-Algorithmen

Die elementaren Sortier-Algorithmen eignen sich für kleinere Arrays (bis ca. 1000 Elemente), da ihr Aufwand quadratisch mit der Anzahl Elemente wächst, d.h. bei einer Verdoppelung der Anzahl Elemente steigt die Sortierzeit auf das Vierfache.

Die bekanntesten elementaren Sortier-Algorithmen sind *Sortieren durch direktes Auswählen* und der *Bubble Sort*.

Sortieren durch direktes Auswählen (Select-Sort)

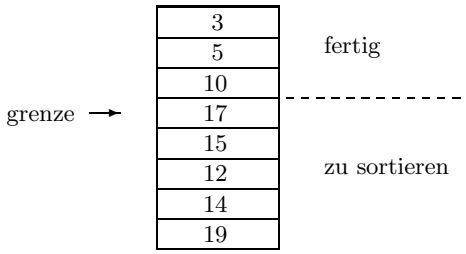
Dieses Verfahren funktioniert nach dem folgenden einfachen Prinzip, welches auch beim manuellen Sortieren irgendwelcher Daten angewandt werden kann:

1. Bestimme das kleinste Element des Arrays und vertausche es mit dem ersten Element des Arrays.

2. Weiter, nach dem gleichen Prinzip, mit dem reduzierten Array ohne das erste Element.

   Abbruch: wenn der noch zu bearbeitende Array nur aus einem Element besteht.

Wir führen eine Variable *grenze* ein, die immer angibt, ab welchem Element der Array noch zu sortieren ist:



Java-Methode:

```
static void selectSort( int[ ] a)
{ int min, pos;
  for ( int grenze = 0; grenze < a.length−1; grenze++ )
  { // ———— Minimum ab a[grenze] bestimmen ——
    min = a[grenze];  pos = grenze;
    for ( int i=grenze+1; i < a.length; i++ )
      if ( a[i] < min )
      { min = a[i];  pos = i;
      }
    // ———— Vertauschung a[grenze] und a[pos] ——
    a[pos] = a[grenze];
    a[grenze] = min;
  }
}
```

Man beachte, dass beim letzten Durchgang der Hauptschleife *grenze* gleich $a.length - 2$ ist. Dies bedeutet, dass der zu bearbeitende Array noch zwei Elemente hat. Wenn nur noch ein Element zu sortieren ist, muss nichts mehr gemacht werden.

△ **Uebung:** Lösen Sie die Aufgabe 2.9.3

Der Bubble-Sort Algorithmus

Beim Bubble-Sort Algorithmus werden sukzessive Sortierfehler behoben, bis der ganze Array sortiert ist.

Algorithmus:

1. Der Array wird sequentiell durchlaufen. Dabei werden je zwei aufeinanderfolgende Elemente verglichen. Wenn ihre Reihenfolge nicht stimmt, werden sie vertauscht.

   So wandern grosse Elemente sukzessive gegen das Ende des Arrays, sie steigen auf wie Luftblasen (Bubbles) im Wasser.

   Am Ende des Durchlaufs befindet sich das grösste Element des Arrays sicher an der höchsten Stelle des Arrays.

2. Weiter, nach dem gleichen Prinzip, mit dem reduzierten Array ohne das *letzte* Element.

Abbruch:

Wenn bei einem Durchlauf keine Sortierfehler festgestellt werden.

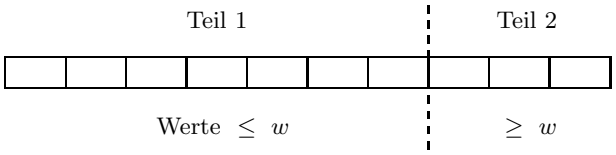
△ **Uebung:** Lösen Sie die Aufgabe 2.9.4

Der Bubble-Sort ist i.a. langsamer als der Select-Sort, da er zuviele Vertauschungen macht. Er kann schneller sein, wenn der Array schon vorsortiert ist, d.h. nur wenige Sortierfehler aufweist.

2.5 Der Quicksort-Algorithmus

Der Quicksort-Algorithmus von C.A. Hoare (1962) ist einer der schnellsten Sortier-Algorithmen. Er funktioniert nach einem genial einfachen Grundprinzip:

1. Wähle einen beliebigen Wert w aus dem Array und erzeuge mit geeigneten Vertauschungen eine Unterteilung des Arrays in zwei Teile, sodass alle Werte des ersten Teils $\leq w$ sind und die des zweiten Teils $\geq w$.



   Eine solche Unterteilung erlaubt, die beiden Teile unabhängig voneinander weiter zu sortieren, da kein Element vom linken Teil in den rechten oder umgekehrt verschoben werden muss.

2. Sortiere den linken Teil separat nach demselben Prinzip

3. Sortiere den rechten Teil separat nach demselben Prinzip

Die Schritte 2 und 3 können mittels Rekursion ausgeführt werden, sodass sich die Entwicklung des Algorithmus im wesentlichen auf den Punkt 1 beschränkt.

Gemäss der Definition der beiden Teile der Unterteilung dürfen Elemente mit Wert gleich w im linken *oder* im rechten Teil vorkommen, da die Bedingungen '$\leq w$' und '$\geq w$' gewählt wurden. Dies erleichtert den Algorithmus für die Unterteilung und genügt für separate Weiterverarbeitung der beiden Teile. Vom Prinzip her könnte man bei einem der beiden Teile eine echte Ungleichung verwenden.

Ansatz für eine rekursive Methode 'qSort'

Die Methode erhält als Parameter neben dem Array a zwei Indizes $left$ und $right$ eines Teilarrays, den sie bearbeiten soll. Dadurch kann sie sich mit Teilarrays rekursiv aufrufen.

```
static void qSort(int a[ ], int left, int right)
{
  if ( left >= right )  return;                    // nichts zu tun
  - Wähle einen Wert w für die Unterteilung
  - Erzeuge eine Unterteilung
  - Wenn der linke Teil der Unterteilung mindestens
    zwei Elemente hat, rufe qSort für den linken Teil auf.
  - Wenn der rechte Teil der Unterteilung mindestens
    zwei Elemente hat, rufe qSort für den rechten Teil auf.
}
```

Wahl des Wertes w

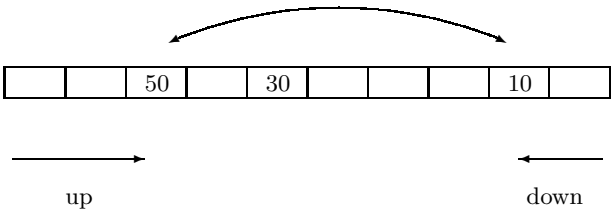
Für die Unterteilung im ersten Schritt kann grundsätzlich ein beliebiger Wert des Arrays gewählt werden. Günstig sind Werte, für welche die beiden Teile der Unterteilung möglichst gleich gross werden.

Man verzichtet aber auf solche Kriterien und wählt einfach den Wert des Elementes in der Mitte des Arrays.

Erzeugung einer Unterteilung:

Mit dem Algorithmus zur Erzeugung einer Unterteilung steht und fällt das ganze Verfahren. Tatsächlich geht dies sehr effizient:

- Der Array wird von links durchsucht (up), bis ein Element $\geq w$ gefunden wird, und von rechts (down), bis man auf ein Element $\leq w$ stösst.[1]
- Die beiden Elemente werden vertauscht, und der Prozess wird fortgesetzt, bis man sich beim Durchsuchen trifft (up > down).



[1] Die Relationen '$\leq$' und '$\geq$' anstelle von '$<$' und '$>$' garantieren das Anhalten der up- bzw. down-Schleife, spätestens beim Element mit Wert w

Das Element in der Mitte, dessen Wert (30) als Unterteilungswert w gewählt wurde, kann dabei ebenfalls verschoben werden. Der Unterteilungswert w bleibt aber für die Unterteilung fest.

Bei der Vertauschung der beiden Elemente werden mit einem Schlag zwei Fehler korrigiert. Dies ist ein wesentlicher Grund für die Effizienz des Algorithmus.

Versuchen Sie, die Unterteilung in die oben angesetzte Methode 'qSort' einzubauen, bevor sie die folgende Lösung studieren.

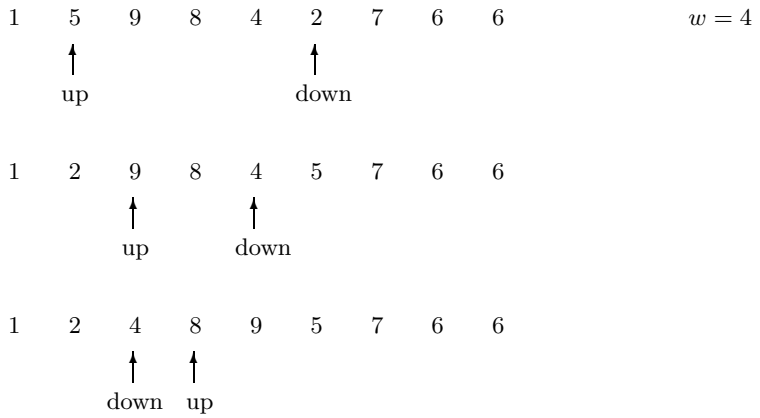
Vollständige Methode 'qSort'

```
// ———————— Quick−Sort (C.A.Hoare) ————————
static void qSort(int a[ ], int left, int right)
{ int tmp;
  if ( left >= right ) return;                  // nichts zu tun
  int m = (left+right) / 2;                     // mitlerer Index
  int w = a[m];                                 // Unterteilungswert
  int up = left, down = right;
  while ( up <= down )                          // Vorsortierung
  { while( a[up] < w) up++;
    while( a[down] > w) down−−;
    if ( up <= down )
    { tmp = a[up];                              // Vertauschung a[up] und a[down]
      a[up] = a[down];
      a[down] = tmp;
      up++; down−−;
    }
  }
  if ( down > left ) qSort(a, left, down);      // weiter mit linkem Teil
  if ( up < right ) qSort(a, up, right);        // weiter mit rechtem Teil
}
```

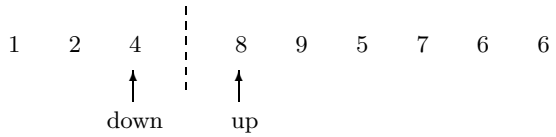
Man beachte, dass in den up- und down-Schleifen die 'while'-Bedingungen `a[up] < w` bzw. `a[down] > w` anstelle der oben eingeführten Abbruchbedingungen `a[up] >= w` bzw. `a[down] <= w` erscheinen.

Beispiel zur Erzeugung einer Unterteilung:

Die folgenden Zahlenreihen stellen den zu sortierenden Array dar, und zwar immer direkt nach den up/down-Schleifen, vor dem 'if' für die Vertauschung.



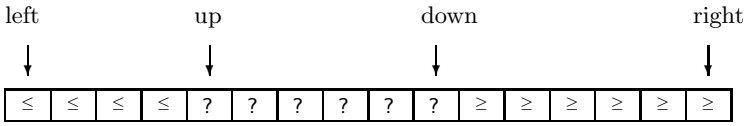
In der letzten Zeile ist *up* also vor der Vertauschung bereits grösser als *down*. Wegen der 'if'-Abfrage erfolgt daher keine Vertauschung mehr, was wichtig ist, da die Vorsortierung fertig ist mit den folgenden beiden Teilen:



Schleifeninvariante:

(1) Für alle $i < up$ ist $a[i] \leq w$ und

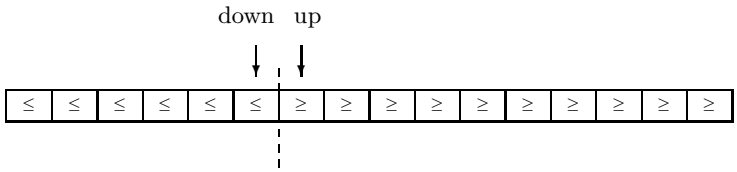
(2) Für alle $j > down$ ist $a[j] \geq w$



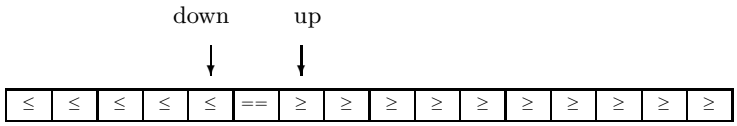
Diese Bedingungen sind am Anfang sicher erfüllt, weil es keine Elemente links von *up* bzw. rechts von *down* gibt.

Bei der Vertauschung von $a[up]$ und $a[down]$ werden gleich zwei Elemente richtig gestellt, folglich können *up* erhöht und *down* vermindert werden, unter Beibehaltung der Bedingungen (1) und (2).

Im Endzustand ist $up > down$. Zusammen mit der Schleifeninvariante ergibt dies die folgenden beiden möglichen Situationen:



oder:



Das Element zwischen *down* und *up* ist sowohl $\geq w$ wie auch $\leq w$, also gleich w. Es muss bei der weiteren Sortierung nicht mehr verschobem werden, d.h. man kann auch in diesem Fall mit den beiden Teilen links von *down* bzw. rechts von *up* weiterfahren.

△ **Uebung:** Lösen Sie die Aufgabe 2.9.5

2.6 Sortierzeiten

In der folgenden Tabelle sind gemessene Laufzeiten der verschiedenen Sortier-Algorithmen auf einem Intel Pentium 100 MHz für Arrays mit ganzen Zufallszahlen aufgeführt.

	Sortierzeit [Sekunden]		
Anzahl Elemente	Select Sort	Bubble Sort	Quicksort
2000	0.1	0.2	< 0.1
4000	0.5	0.8	< 0.1
8000	2.1	3.33	< 0.1
16000	8.4	17.2	< 0.1
32000	36.2	60.2	< 0.1
64000	152.1	265.8	0.1
128000	?	?	0.3
256000	?	?	0.7
512000	?	?	1.5
1024000	?	?	3.3

Die angegebenen Zeiten sind Mittelwerte einiger Läufe.

Auswertung:

- Der Unterschied der elementaren Verfahren zum Quicksort für grosse Arrays ist verblüffend. Die schlechten Zeiten des Bubble Sort sind ebenfalls unerwartet.
- Das Problem der elementaren Verfahren ist, dass ihr Aufwand proportional zum *Quadrat* der Anzahl Elemente ist (wegen den verschachtelten Schleifen).

  Dies bedeutet, dass bei einer Verdoppelung des Arrays die Sortierzeit auf das Vierfache steigt, was die Tabelle gut bestätigt.
- Man kann zeigen, dass der Aufwand des Quicksort proportional zu

  $$n \cdot \log(n) \quad \text{mit} \quad n = \text{Anzahl Elemente}$$

  ist. Weil die Logarithmus-Funktion sehr langsam wächst, z.B.

  $$\log(10000) = 4, \quad \log(100000) = 5,$$

  bedeutet dies im wesentlichen ein lineares Wachstum mit n. Dies geht ebenfalls aus der Tabelle gut hervor: Bei Verdoppelung des Arrays steigt die Zeit nur auf wenig mehr als das Doppelte.

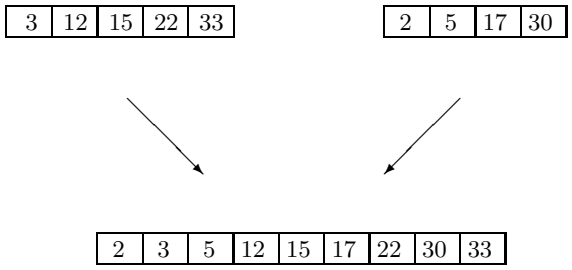
Schlussfolgerungen:

Für kleine Arrays ist der Select Sort geeignet, für grosse ist der Quicksort wesentlich schneller. Der Bubble Sort bringt gegenüber dem Select Sort nur dann Vorteile, wenn der zu sortierende Array schon fast sortiert ist.

2.7 Mischen (Merge)

Beim Mischen werden 2 oder mehrere sortierte Arrays (oder Files) zu einem einzigen zusammengefügt, der wiederum sortiert ist.

Beispiel:



Verfahren:

- Die Elemente der beiden Arrays werden mit zwei Indizes sequentiell durchlaufen.
- Dabei werden die beiden momentanen Elemente verglichen. Das kleinere wird übertragen, dann wird der betreffende Index erhöht.

Merge-Methode

In der Methode muss der Fall berücksichtigt werden, dass einer der beiden Arrays schon fertig verarbeitet ist, der andere noch nicht:

```
int[ ]  merge(int[ ] a, int[ ] b)
{ int[ ] c = new int[a.length+b.length];
  int ia = 0, ib = 0, ic = 0;
  while ( ia < a.length  || ib < b.length )
  { if ( ia == a.length )                    // a fertig verarbeitet ?
    { c[ic] = b[ib];                         // b uebertragen
      ib++;
    }
    else if ( ib == b.length )               // b fertig ?
    { c[ic] = a[ia];                         // a uebertragen
      ia++;
    }
    else if ( a[ia] <= b[ib] )
    { c[ic] = a[ia];                         // a uebertragen
      ia++;
    }
    else
    { c[ic] = b[ib];                         // b uebertragen
      ib++;
    }
    ic++;
  }
}
```

2.8 Gruppenverarbeitungen

In einem sortierten Datenbestand mit nicht eindeutigem Key, können Datensätze mit gleichen Keys als logische Gruppen betrachtet werden. Wegen der Sortierung folgen die Datensätze einer Gruppe direkt aufeinander.

Beispiel:

In unserem File mit Postleitzahlen und Ort-Daten, sortiert nach Postleitzahlen, bilden die Datensätze Gruppen mit gleichen Postleitzahlen, z.B.

5200 AG Brugg AG
5200 AG Koenigsfelden
5200 AG Brugg AG 1
5200 AG Brugg AG 3

5200 AG Lauffohr
5200 AG Brugg AG Kaserne

Eine sequentielle Verarbeitung, die eine solche Gruppenstruktur verwendet, nennt man eine *Gruppenverarbeitung*.

Beispiel einer Gruppenverarbeitung:

Es soll für jede Gruppe die Anzahl Elemente bestimmt und auf den Bildschirm ausgegeben werden.

Es genügt, den Algorithmus für einen sortierten Array a von ganzen Zahlen zu entwickeln. Er kann problemlos für andere Datenstrukturen modifiziert werden.

Wir wählen einen Ansatz, der die Gruppenstruktur wiederspiegelt:

```
while ( ! endOfData )
{ Start Gruppe
  Elemente der Gruppe verarbeiten
  Abschluss der Gruppe
}
```

Vollständige Methode:

```
void gruppenVerarb(int[ ] a)                          // Elemente der Gruppen zählen
{ int i, counter;
  int current;
  i = 0;
  while ( i < a.length )
  { current = a[i];                                   // Start Gruppe
    counter = 0;
    while ( i < a.length && current == a[i] )         // Elemente verarbeiten
    { counter++;
      i++;
    }
    System.out.println(current + " " + counter);      // Abschluss der Gruppe
  }
}
```

Wenn die Daten von einem File eingelesen werden, muss darauf geachtet werden, dass das erste Element *vor* der Schleife eingelesen wird. Dies entspricht der Anweisung $i = 0$ im Array-Beispiel.

Dieses *Vorauslesen* ('read ahead') wurde von Jackson, dem Erfinder der Strukturierten Programmierung, eingeführt.

Das folgende Beispiel liest das File 'plzort.txt' mit Postleitzahlen und Ort-Daten von Übung 2.9.1 und bestimmt für jede Postleitzahl die Anzahl Datensätze:

```
// ___________ Gruppenverarbeitung (PLZ−File) ____________________________
import java.io.*;
public class PLZGruppen
{
  static public void main(String[ ] args)
              throws FileNotFoundException, IOException
  { int counter;
    String plz;
    FileReader fr = new FileReader("plzort.txt");
    BufferedReader br = new BufferedReader(fr);

    String input = br.readLine();                    // read ahead
    while ( input != null )
    { plz = input.substring(0, 4);
      counter = 0;
      while ( input != null &&
              plz.equals(input.substring(0, 4)))
      { counter++;
        input = br.readLine();                       // read next
      }
      System.out.println(plz + " " + counter);
    }
    br.close();
  }
}
```

△ **Uebung:** Lösen Sie die Aufgabe 2.9.6

2.9 Übungen

2.9.1 Ort-Abfrage mit PLZ

Gegeben ist ein Text-File 'plzort.txt' mit den Postleitzahlen und Ortdaten der Schweizerischen Post[2] Erstellen Sie ein Applet 'PlzOrt', welches das File in Arrays einliest und dann zu einer eingegebenen Postleitzahl die Ortdaten ausgibt.

Die Postleitzahl wird in einem TextField eingegeben.

[2] ftp://loki.cs.fh-aargau.ch/pub/java/plzort.txt

Zeilenformat des Files:

Postleitzahl (4 Stellen), Kanton (2 Stellen), Ort-Bezeichnung (variable Laenge) :

```
1000VDLausanne
1000VDLosanna
1000VDLes Monts−de−Pully
1000VDLausanne 2
1000VDLausanne 3
1000VDLausanne 4
```

Das File enthält knapp 8000 Zeilen. Lesen Sie die Postleitzahlen in einen separaten 'int'-Array ein.

2.9.2 PLZ-Abfrage mit Binärer Suche

Ersetzen Sie in der PLZ-Abfrage (Übung 2.9.1) die lineare Suche durch eine binäre. Das File 'plzort.txt' ist sortiert nach den Postleitzahlen.

2.9.3 Sort-Test

Erstellen Sie eine Applikation 'SortTest', welche einen Array mit n Elementen erzeugt und mit ganzen Zufallszahlen im Bereich 1 .. max initialisiert. Die Werte n und max sind Eingabewerte des Programmes.

Anschliessend wird der Array mittels Select-Sort sortiert. Dabei wird die benötige Sortierzeit gemessen (mit einer Stop-Uhr der Klasse 'Clocks' von früher, die unten aufgelistet ist).

Durchführung mit den Werten:

$$n = 4000,\ 8000,\ 16000,\ 32000,\ 64000 \quad \text{und} \quad max = n\ .$$

Verifizieren Sie die Behauptung, dass die Zeit bei einer Verdoppelung von n auf das Vierfache steigt.
(Zur Überprüfung der Sortierung sollte der sortierte Array auf den Bildschirm ausgegeben werden.)

```
public class Clocks                              // Stop−Uhren
{  private long startZeit;
   private long speicher = 0;
   private boolean gestartet = false;

   // _________ Methoden _____________________
```

```
  public void start()
  { if ( ! gestartet)
    { startZeit = System.currentTimeMillis();
      gestartet = true;
    }
  }
  public void stop()
  { if ( gestartet )
    { speicher += System.currentTimeMillis() − startZeit;
      gestartet = false;
    }
  }
  public void reset()
  { speicher = 0;
    gestartet = false;
  }

  public double display()[2pt]                    // Anzeige in Sekunden
  { if ( gestartet )
      return (System.currentTimeMillis() −
                startZeit + speicher) ∗ 0.001;
    else
      return speicher ∗ 0.001;
  }
}
```

2.9.4 Bubble-Sort

Erstellen Sie eine Methode 'bubbleSort' für den Bubble-Sort Algorithmus und bauen Sie diese in das SortTest-Programm ein (Übung 2.9.3).

2.9.5 Sort-Demo

Erstellen Sie ein Applet 'SortDemo' zur Veranschaulichung von Sort-Algorithmen mittels graphischer Darstellung:

- Das Applet erzeugt einen Array a mit n Elementen, z.B. $n = 8000$, und initialisiert ihn mit den Werten $1 .. n$.
- Der Array wird auf dem Bildschirm dargestellt, indem für jedes Element i ein Punkt mit Koordinaten

  $$x = i \quad \text{und} \quad y = a[i]$$

  gezeichnet wird (Koordinaten-Umrechnung für Bildschirm mittels 'Graph2d', x$Min = 1$, $xMax = n$, $yMin = 0$, $yMax = n$. Da die

Bildschirmauflösung kleiner als 8000 x 8000 ist, fallen einige Punkte aufeinander, was nicht stört).

Der Anfangszustand des Arrays erscheint als Gerade.

- Mit einem Button 'unSort' wird die Sortierung mittels genügend vielen zufälligen Vertauschungen zerstört. Jede Vertauschung wird auf dem Bildschirm nachgeführt.
- Mit einem Button 'sort' wird der Array mittels Quicksort sortiert. Dabei wird wieder jede Vertauschung auf dem Bildschirm nachgeführt.

Hinweise:

a) Führen Sie alle Vertauschungen mit einer Methode

```
void swap(Graphics g, Graph2d graph2d, int[ ] a, int i, int j)
```

durch. Die Methode zeichnet die beiden Elemente i und j in der Hintergrundfarbe, dann vertauscht sie diese im Array, und zeichnet die neuen Punkte in der Vordergrundfarbe.

b) In der 'paint'-Methode wird der ganze Array gezeichnet.

c) In der Methode 'actionPerformed' wird das Graphics-Objekt mittels 'getGraphics' geholt und an die aufgerufenen Methoden übergeben, sodass diese die Veränderungen direkt anzeigen können (Methode 'swap').

Testen Sie das Applet auch mit dem Select-Sort. Für den Bubble-Sort müsste die Anzahl Elemente verkleinert werden, sonst braucht er wegen den vielen Vertauschungen zuviel Zeit.

2.9.6 Konto-Verarbeitung

Gegeben sind ein Array mit Kontodaten und ein Array mit Buchungen (Ein-/Auszahlungen) für die Konten.

Element-Typen der Arrays:

```
class Konto
{ int nr;                                   // Konto−Nr.
  String name, vName;
  int saldo;                                // Saldo in Rappen
}

class Buchung
{ int kontoNr;                              // Konto−Nr.
  int betrag;                               // Betrag in Rappen
}
```

Buchungen mit positivem Betrag sind Einzahlungen, solche mit negativem Auszahlungen.

Aufgabe:

Erstellen Sie eine Methode

```
void verbuchung( Konto[ ] konto, Buchung[ ] buchung )
```

welche die Saldi der Konten aufgrund der zugehörigen Buchungen nachführt.

- Die gegebenen Arrays sind sortiert nach Konto-Nummern.
- Im Konten-Array sind die Konto-Nummern eindeutig, im Array der Buchungen kommen i.a. mehrere Buchungen zu einem Konto vor (Gruppenstruktur).
- Buchungen mit Konto-Nummern, die im Konten-Array nicht vorkommen, sind als fehlerhafte Datensätze auf den Bildschirm auszugeben.
- Die Arrays dürfen nur einmal sequentiell verarbeitet werden, sodass die Methode bei Bedarf leicht auf File-Input umgestellt werden könnte.

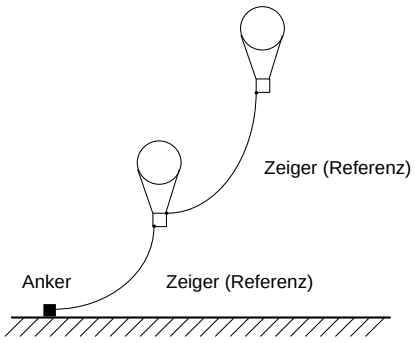
Kapitel 3

Dynamische Datenstrukturen

Dynamische Datenstrukturen bestehen wie Arrays aus mehreren Elementen eines bestimmten Typs. Sie unterscheiden sich von Arrays dadurch, dass ihre Elemente im Speicher nicht direkt hintereinander liegen müssen, denn sie sind mittels *Zeigern* (Pointers) verknüpft.

Ein Zeiger ist eine Variable, die eine Speicheradresse enthält, d.h. in Java eine Referenzvariable.

Anschaulicher Vergleich von Dynamischen Datenstrukturen mit verketteten Luftballons[1]:



Dynamische Datenstrukturen bringen gegenüber Arrays Vorteile, wenn Elemente in einer bestimmten Reihenfolge zwischen bestehende Elemen-

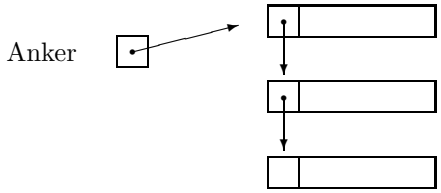
[1] nach J. Ludewig, 'Einführung in die Informatik'

te eingefügt werden sollen, sowie bei der Entfernung von Elementen. Bei diesen Operationen müssen lediglich Zeiger mit neuen Werten versehen werden, ohne dass andere Elemente verschoben werden müssen.

Die wichtigsten Arten von Dynamischen Datenstrukturen sind *verkettete Listen* und *binäre Bäume*.

Verkettete Listen

Jedes Element einer verketteten Liste enthält neben den eigentlichen Daten einen Zeiger (Referenz) *next* auf das nachfolgende. Beim letzten Element der Liste ist diese Referenz gleich *null*.



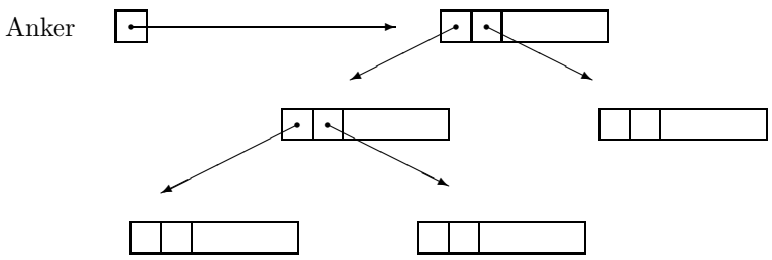
Vergleich mit Arrays

- In einer verketteten Liste können Elemente nach Bedarf hinzugefügt oder entfernt werden, auch zwischen schon bestehenden Elementen. Dazu müssen lediglich Zeiger umgehängt werden.

- Ein Direktzugriff auf das i-te Element einer verketteten Liste mit einem Index ist jedoch nicht möglich. Die Elemente können nur sequentiell (der Reihe nach) gelesen werden.

Binäre Bäume

Jedes Element eines binären Baumes enthält zwei Zeiger $left$ und $right$ auf weitere Elemente. So entsteht eine Verzweigungsstruktur, die als umgekehrter Baum interpretiert werden kann.



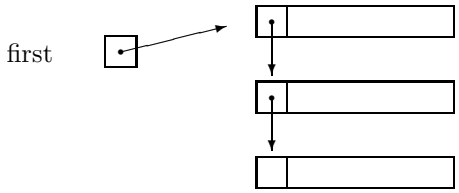
3.1 Verkettete Listen

Wir führen die Erzeugung und Verwendung von verketteten Listen an typischen Beispielen ein.

3.1.1 Einlesen eines Textfiles in eine verkettete Liste

Gegeben sei ein Textfile, welches in den Speicher eingelesen werden soll. Da im voraus nicht bekannt ist, wieviele Zeilen das File enthält eignet sich ein Array von Strings nicht als Datenstruktur.

Wir verwenden daher eine verkettete Liste, welche für jede Zeile des Files ein Element enthält:



Klasse für die Elemente der Liste:

```
class Zeile
{ Zeile next;                    // Zeiger auf naechstes Element
  String daten;                  // Datenzeile
}
```

Diese Definition hat rekursiven Charakter, da 'next' notwendigerweise eine Referenz auf ein gleichartiges Objekt 'Zeile' ist.

Die Klasse 'Liste'

Wir führen eine weitere Klasse für Listen ein. Ein Objekt der Klasse repräsentiert eine verkettete Liste. Es enthält als wichtigste Datenkomponente eine Referenz 'first' auf das erste Element der Liste.

Für die Einfügung neuer Elemente am Ende einer Liste, führen wir eine zusätzliche Datenkomponente 'last' ein, die immer auf das letzte Element der Liste zeigt.

```
class Liste
{ Zeile first;                          // Zeiger auf erstes Element
  Zeile last;                           // Zeiger auf letztes Element
  // -- Methoden, siehe unten --
}
```

Methoden der Klasse 'Liste'

- Einfügung eines Elementes

  Zur Einfügung eines Elementes führen wir eine Methode 'insert' ein. Die Methode erhält ein Element (Klasse 'Zeile') und fügt es am Ende der Liste ein, indem sie die Zeiger entsprechend umhängt. Dabei wird der Fall der leeren Liste speziell behandelt.

```
public void insert(Zeile zeile)
{ if ( first == null )
     first = zeile;                     // Einfuegung als erstes Element
  else
     last.next = zeile;                 // Einfuegung am Ende
  last = zeile;
}
```

- Bildschirm-Ausgabe

  Als typisches Beispiel einer sequentiellen Verarbeitung der Elemente einer verketteten Liste betrachten wir eine Methode 'print', welche die Daten der Elemente auf den Bildschirm ausgibt.

  Die Methode verwendet einen Hilfszeiger 'tmp', mit dem die Elemente in einer Schleife durchlaufen werden, analog zu einem laufenden Index für einen Array.

```
public void print()
{ Zeile tmp = first;                    // laufender Zeiger
  while ( tmp != null )
  { System.out.println(tmp.daten);      // Ausgabe
    tmp = tmp.next;                     // Zeiger nachführen
  }
}
```

Hauptprogramm

In einer Applikation wird jetzt eine Liste erzeugt, in welche die Zeilen eines Textfiles eingelesen werden.

```
static public void main(String[ ] args)
          throws FileNotFoundException, IOException
{ Liste liste = new Liste();
  Zeile zeile;
  String input;
  String fName = "test.txt";                // File−Name
  FileReader f = new FileReader(fName);     // File oeffnen
  BufferedReader br =
             new BufferedReader(f);
  while ( (input = br.readLine()) != null ) // Zeilen einlesen
  { zeile = new Zeile();                    // neues Element
    zeile.daten = input;
    liste.insert(zeile);                    // Einfuegung in Liste
  }
  br.close();
}
```

▶ *Merke:*

Für jedes Element der Liste muss ein neues Objekt 'Zeile' erzeugt werden, da die Methode 'insert' nur die Verknüpfungen nachführt.

3.1.2 Beispiel einer sortierten Liste

Als typisches Beispiel einer sortierten verketteten Liste betrachten wir eine Rangliste für ein Skirennen, die bei jeder Zielankunft eines Fahrers aktualisiert wird.

Die Einfügung eines Fahrers erfolgt an der richtigen Stelle aufgrund seiner Fahrzeit.

Klasse für die Elemente der Liste:

```
class Fahrer                                  // Fahrer−Daten
{ Fahrer next;                                // Zeiger auf naechstes Element
  double zeit;                                // Fahrzeit
  int startNr;
  String name;
}
```

Klasse für Ranglisten:

Wir führen wieder die Datenkomponente 'first' ein, den Zeiger auf das erste Element. Ein Zeiger auf das letzte Element wird hier nicht benötigt, da die Elemente nicht am Ende, sondern gemäss ihren Zeiten eingefügt werden.

```
class Liste
{ Fahrer first;                               // Zeiger auf erstes Element
  // −− Methoden, siehe unten −−
}
```

Methode zur Einfügung eines Fahrers

Die folgende Methode 'insert' der Klasse 'Liste' fügt ein Fahrer-Element in die Liste ein. Die Einfügeposition wird aufgrund der Zeit des Fahrers bestimmt.

```
public void insert(Fahrer fahrer)             // Einfuegung in Sortierfolge
{
  Fahrer tmp, prev;
  - Der Zeiger tmp wird auf das erste Element der Liste gesetzt,
    dessen Zeit grösser als die des einzufügenden Fahrers ist
    und der Zeiger prev auf das vorangehende Element.
  - Das neue Element wird zwischen prev und tmp in die Liste eingefügt.
}
```

Versuchen Sie die Methode zu vervollständigen, bevor Sie die vollständige Lösung anschauen.

```
public void insert(Fahrer fahrer)              // Einfuegung in Sortierfolge
{ Fahrer tmp=first;
  Fahrer prev=null;
  while ( tmp != null && tmp.zeit <= fahrer.zeit )
  { prev = tmp;
    tmp = tmp.next;
  }
  fahrer.next = tmp;
  if ( prev == null )
    first = fahrer;                            // Einfuegung an erster Stelle
  else
    prev.next = fahrer;                        // Einfuegung nach prev
}
```

3.1.3 Queues

Eine *Queue* (Warteschlange) ist eine verkettete Liste, in welche neue Elemente immer am Ende eingefügt werden, und bei der immer das erste Element zuerst verarbeitet wird.

Verarbeitungs-Prinzip:

FIFO: 'First in first out'. Wer zuerst gekommen ist, wird zuerst bedient.

Standard-Methoden:

enqueue	Einfügung eines Elementes am Ende der Liste
dequeue	erstes Element der Liste holen und aus der Liste entfernen

Java-Klassen:

```
public class QueueEl                           // Elemente
{ QueueEl next;
  int data;                                    // Daten (nach Bedarf)
}
```

```
public class Queue                          // Queues
{
  QueueEl first, last;                      // Referenzen

  public void enqueue(QueueEl e)
  { e.next = null;
    if ( first == null )                    // Queue leer ?
      first = e;
    else
      last.next = e;                        // Einfuegung am Ende
    last = e;
  }

  public QueueEl dequeue()
  { QueueEl tmp = first;
    if ( first != null )                    // Entfernung aus Queue
    { first = first.next;
      tmp.next = null;
    }
    if ( first == null )                    // letztes Element entfernt ?
      last = null;
    return tmp;
  }
}
```

△ **Uebung:** Lösen Sie die Aufgabe 3.3.1

3.1.4 Stacks

Ein *Stack* (Stapel) ist eine verkettete Liste, die wie ein Tellerstapel verwendet wird: neue Elemente werden vorne eingefügt, bei der Verarbeitung wird wieder (wie bei Queues) das vorderste Element, d.h. das zuletzt eingefügte, genommen.

Verarbeitungs-Prinzip:

LIFO: 'Last in first out'. Das zuletzt eingefügte Element wird als erstes verarbeitet.

Standard-Methoden:

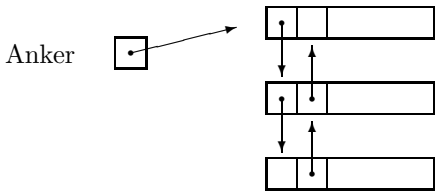
push	Einfügung eines Elementes in den Stack
pop	Hole das erste Element und entferne es vom Stack

Beispiel eines Stacks: Der Call Stack für Methoden-Aufrufe.

△ **Uebung:** Lösen Sie die Aufgabe 3.3.2

3.1.5 Doppelt verkettete Listen

In einer doppelt verketteten Liste enthält jedes Element neben der 'next'-Referenz eine weitere, 'prev', die auf das vorangehende Element verweist:



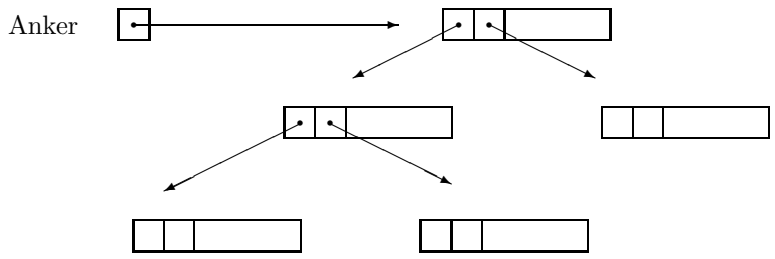
Die Rückwärtspfeile sind aus optischen Gründen vertikal dargestellt. Tatsächlich zeigen die Referenzen natürlich auf die Anfangsadressen der Elemente.

Zweck der Rückwärtsverkettung:

- Rückwärtslesen der Liste
- Einfache Entfernung von Elementen

3.2 Binäre Bäume

Jedes Element eines binären Baumes enthält neben den eigentlichen Daten zwei Zeiger $left$ und $right$, die auf weitere Elemente zeigen.



Knoten Die Elemente eines binären Baumes nennt man auch *Knoten* (Nodes), das oberste Element heisst Wurzel (Root).

Blatt Ein Knoten, der keine Nachfolger hat ($left$ und $right$ beide gleich $null$), wird auch als *Blatt* (Leaf) bezeichnet.

Rekursive Definition

Ein Baum hat als Datenstruktur rekursiven Charakter:

Ein binärer Baum ist entweder leer, oder er besteht aus einer Wurzel mit einem linken und einem rechten Teilbaum (die leer sein können).

Dies deutet darauf hin, dass bei den Verarbeitungen von binären Bäumen auch Rekursion zum Einsatz kommt.

Suchbäume

Binäre Bäume sind prädestiniert für Suchvorgänge nach dem Prinzip des binären Suchens. Dazu muss man aber bei der Suche nach einem bestimmten Wert bei jedem Knoten feststellen können, ob die Suche in den linken oder in den rechten Teilbaum verzweigen soll.

Dies ist möglich, wenn der Baum sortiert ist, gemäss der folgenden Definition.

Definition

Ein binärer Baum heisst *sortiert* in Bezug auf eine Datenkomponente 'key', wenn für *jeden* Knoten p des Baumes die folgenden beiden Bedingungen erfüllt sind:

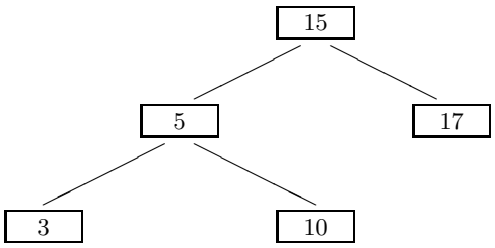
(a) Für jeden Knoten l des linken Teilbaums von p gilt:

$$l.key \leq p.key$$

(b) Für jeden Knoten r des rechten Teilbaums von p gilt:

$$r.key > p.key$$

Beispiel eines sortierten Baumes:



Man beachte, dass die Bedingungen (a) und (b) für *jeden* Knoten erfüllt sein müssen, nicht nur für die Wurzel. Einen sortierten Baum nennt man auch *Suchbaum*.

Klassen für Suchbäume mit Methoden für Grundoperationen:

```
public class Node
{ public Node left, right;          // Referenzen
  public int key;                   // Key
                                    // weitere Daten nach Bedarf
  public Node(int key)              // Konstruktor
  { this.key = key;
  }
}
```

```
public class Tree
{ public Node root;                          // Root−Referenz
  public void insert(Node node)              // Einfügung in Sortierfolge
  { siehe unten }
  public Node search(int key)                // Suche
  { siehe unten }
  public void printKeys(Node start)          // Bildschirmausgabe
  { siehe unten }
}
```

Bevor wir die Methoden entwickeln, betrachten wir ein Anwendungs-Beispiel:

Die folgende 'main'-Methode erstellt einen sortierten binären Baum mit positiven ganzen Zahlen (Eingabewerte). Anschliessend werden die Werte in Sortierfolge ausgegeben.

```
static public void main(String[ ] args)
{ Tree tree = new Tree();                    // Tree−Objekt
  Node node;
  int input;                                 // Eingabe−Wert
  InOut.print("Key eingeben (0=Ende): ");
  input = InOut.getInt();
  while ( input > 0 )
  { node = new Node(input);                  // Knoten erzeugen
    tree.insert(node);                       // Einfuegung in Tree
    InOut.print("Key eingeben (0=Ende): ");
    input = InOut.getInt();
  }
  tree.printKeys(tree.root);                 // Ausgabe
}
```

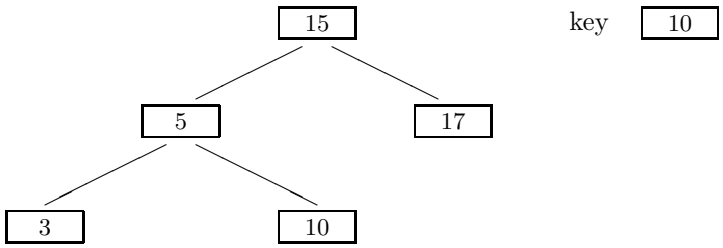
Die Methoden für die Grundoperationen werden im folgenden entwickelt, wobei wir mit der Suche beginnen wollen.

Suche in einem sortierten Baum

Die Suche in einem sortierten Baum nach einem bestimmten Wert 'key' erfolgt nach dem folgenden Prinzip:

(a) Wenn key == root.key
    return root.

(b) Wenn key < root.key
    suche weiter im linken Teilbaum
  sonst
    suche weiter im rechten Teilbaum



Die Search-Methode (Klasse Tree):

```
public Node search(int key)                    // Suche
{ Node p = root;
  while ( p != null && p.key != key )
    if ( key < p.key )
      p = p.left;                              // nach links
    else
      p = p.right;                             // nach rechts
  return p;
}
```

Einfügung in einen sortierten Baum

Die Einfügung eines neuen Knotens *node* erfolgt nach dem folgenden Prinzip:

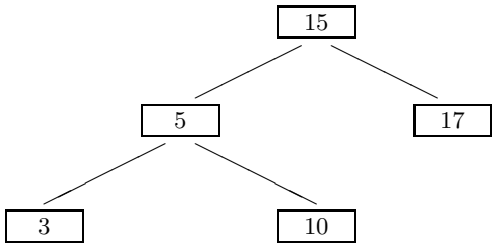
(a) Wenn der Baum leer ist, setze root = node.

(b) Andernfalls vergleiche node.key mit root.key :

  Wenn node.key $\leq$ root.key ist
    füge den Node in den linken Teilbaum ein,
  sonst
    füge den Node in den rechten Teilbaum ein.

Beispiel:

Der Node mit key = 12 soll in den folgenden Baum eingefügt werden:



Wir schauen auf jeder Stufe des Baumes, ob der Node in den linken oder in den rechten Teilbaum des betreffenden Knotens kommt.

Auf der untersten Stufe wird er schliesslich eingefügt. Der neue Node kommt in den rechten Teilbaum des Nodes '10'.

Bei diesem Verfahren werden neue Knoten also immer unter Blätter des Baumes eingefügt, wodurch an der bestehenden Struktur nichts verändert werden muss.

Die Insert-Methode (Klasse Tree):

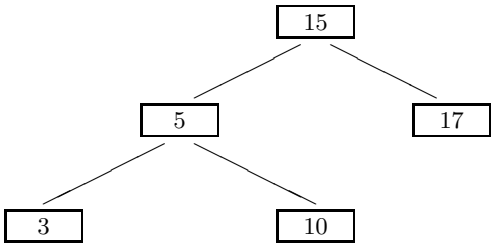
Die Bestimmung der Einfügeposition ist in eine separate Hilfsmethode 'searchLeaf' ausgelagert.

```
public void insert(Node node)                  // Einfügung
{ Node p;
  if ( root == null )
    root = node;
  else
  { p = searchLeaf(node.key);                  // Einfuegeposition
    if ( node.key <= p.key )
      p.left = node;                           // Einfuegung links
    else
      p.right = node;                          // Einfuegung rechts
  }
}
```

```
private Node searchLeaf(int key)              // Einfuegeposition bestimmen
{ Node p = root, prev = null;
  while ( p != null )
  { prev = p;
    if ( key <= p.key )
      p = p.left;                             // nach links
    else
      p = p.right;                            // nach rechts
  }
  return prev;
}
```

Traversierung

Bei einer Traversierung eines Baumes werden alle Knoten (in einer bestimmten Reihenfolge) besucht, wobei für jeden Knoten eine bestimmte Verarbeitung ausgeführt wird.



Beispiel:

Für jeden Knoten soll der *key* auf den Bildschirm ausgegeben werden.

Dies geht sehr einfach mittels Rekursion:

(a) Verarbeite die Wurzel
(b) Traversiere den linken Teilbaum der Wurzel
(c) Traversiere den rechten Teilbaum der Wurzel

Bei dieser Version stimmt jedoch in der Ausgabe die Sortierung nicht, da der Key der Wurzel *vor* den Keys des linken Teilbaums ausgegeben wird:

15 5 3 10 17

Preorder Man nennt dies eine *Preorder-Traversierung.*

Inorder-Traversierung:

Inorder Für eine sortierte Ausgabe ist eine *Inorder-Traversierung* erforderlich:

(a) Traversiere den linken Teilbaum der Wurzel
(b) Verarbeite die Wurzel
(c) Traversiere den rechten Teilbaum der Wurzel

Rekursive Print-Methode (Klasse Tree):

Die Methode benötigt als Parameter eine Referenz auf den Node, mit dem sie starten soll, damit sie sich selber rekursiv aufrufen kann.

```
void printKeys(Node start)
{ if ( start != null )
  { printKeys(start.left);                  // linker Teilbaum
    System.out.println(start.key);          // Ausgabe
    printKeys(start.right);                 // rechter Teilbaum
  }
}
```

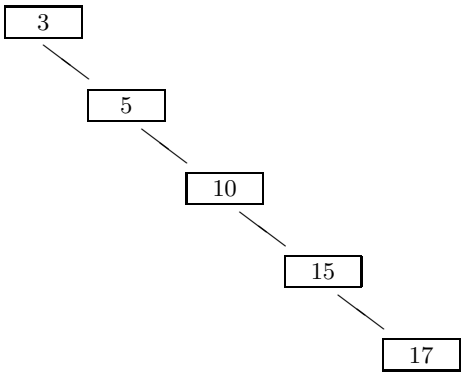
Postorder Bei einer *Postorder-Traversierung* erfolgt die Verarbeitung der Wurzel *nach* der Traversierung des linken und rechten Teilbaumes.

Performance beim Suchen

Der Aufwand bei der Suche nach einem Node in einem Suchbaum hängt von der Anzahl Stufen, die bei dem Suchvorgang durchlaufen werden ab. Auf jeder Stufe ist ein Vergleich erforderlich.

Wenn der Baum optimal organisiert ist, entspricht dieser Aufwand dem des binären Suchens in einem sortierten Array.

Ein Baum kann jedoch mehr oder weniger entartet sein, im Extremfall zu einer verkappten linearen Liste:



Dies ist ebenfalls ein sortierter Baum. Er entsteht in dieser Form, wenn die Einfügung der Knoten mit der Methode 'insert' in Sortierfolge erfolgt:

3 5 10 15 17

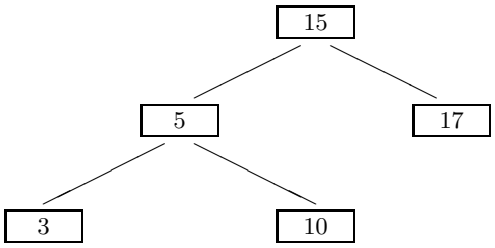
▶ *Merke:*

Bei der Erstellung eines Suchbaumes sollten die Knoten in einer zufälligen Key-Reihenfolge eingefügt werden, nicht in der Sortierfolge.

Höhe eines Baumes

Die Höhe eines Baumes ist die maximale Anzahl Knoten eines Pfades von der Wurzel zu einem Blatt.

Unser Standard-Baum hat Höhe 3 :



Die Höhe ist massgebend für die *maximale* Anzahl Knoten eines binären Baumes:

Höhe	maximale Anzahl Knoten
1	1
2	3
3	7
4	15
n	$2^n - 1$

Für $n = 10$ ergibt dies 1023 Knoten. Ein binärer Baum mit 1000 Knoten hat also eine Höhe im Bereich 10 .. 1000.

Rekursive Definition der Höhe

Die Höhe eines binären Baumes kann folgendermassen definiert werden:

(a) Die Höhe des leeren Baumes ist 0

(b) Die Höhe h eines nichtleeren binären Baumes ist gleich

1 + max(hLeft, hRight)

mit:

hLeft	Höhe des linken Teilbaums der Wurzel
hRight	Höhe des rechten Teilbaums der Wurzel

Diese Definition kann leicht in eine rekursive Methode zur Berechnung der Höhe umgesetzt werden (Übung 3.3.5).

3.3 Übungen

3.3.1 Umwandlung einer Liste in einen Array

Erstellen Sie eine Methode

```
int[ ] convert(Queue q)
```

die eine Queue von 'int'-Elementen in einen Array umwandelt.

3.3.2 Klassen für Stacks

Erstellen Sie Klassen für Stacks von 'int'-Elementen, analog zu den eingeführten Klassen für Queues.

3.3.3 Zufallszahlen

Die bekannteste Methode zur Erzeugung von reellen Zufallszahlen

$$x_1,\ x_2,\ x_3,\ \ldots$$

im Intervall $[0, 1]$ ist die lineare Kongruenzmethode (D.H. Lehner, 1948). Bei dieser wird eine Folge von *ganzen* Zahlen

$$u_1,\ u_2,\ u_3\ \ldots$$

berechnet, mit dem folgenden Algorithmus:

1. Wähle eine beliebige positive ganze Zahl u_o (Startwert).
2. Definiere

   $$u_i = (k\ \cdot\ u_{i-1})\ \ modulo\ \ m \qquad \text{für} \quad i = 1, 2, \ldots$$

   Dabei sind k und m geeignete Konstanten (siehe unten).

Die so berechneten Werte sind infolge der Restbildung alle kleiner als m. Die gewünschten *reellen* Zufallszahlen im Intervall $[0, 1]$ ergeben sich daraus mit der Formel:

$$x_i\ =\ u_i\ /\ m \qquad \text{('double'-Division).}$$

Da zu einem bestimmten Startwert u_o immer dieselbe Folge entsteht, nennt man so erzeugte Zufallszahlen auch Pseudozufallszahlen. Sie bewähren sich jedoch bei Simulationen hervorragend. Wenn nötig, kann man in Anwendungen den Startwert u_o zufällig wählen, z.B. aufgrund der momentaten Zeit.

Perioden:

Perioden sind für Zufallszahlen natürlich unerwünscht, sie sind bei diesem Verfahren jedoch unvermeidbar, da die Werte u_i im endlichen Bereich

$$0,\ 1,\ 2,\ \ldots\ m-1$$

liegen. Früher oder später wiederholt sich daher ein u, und dann führt das Bildungsgesetz zu einer Periode.

Wenn die 0 auftritt, sind alle nachfolgenden Zahlen ebenfalls 0, was nicht eintreten darf.

Im Idealfall werden alle Werte des Bereiches $1, 2, \ldots m-1$ in einer bestimmten Reihenfolge durchlaufen. Dann hat die Folge für jeden Startwert in diesem Bereich die Periodenlänge $m-1$. Dies nennt man eine *Vollperiode*.

Vollperiode

Aufgabe:

Erstellen Sie eine Applikation, welche zu gegebenen Werten k, m und u_o Glieder u_1, u_2, ... berechnet und in eine *sortierte* verkettete Liste speichert, bis ein Wert ein zweites Mal auftritt.

Ausgabe: Werte der Liste, Anzahl Werte (= Periodenlänge).

Vollperioden:

Vollperioden sind in der sortierten Ausgabe leicht erkennbar. Verifizieren Sie, dass die folgenden Konstanten zu Vollperioden führen:

k	m
5	23
2117	6143
3792	8191

Die folgenden Konstanten wurden von der Firma IBM für den Einsatz in der Praxis vorgeschlagen:

$$k = 7^5 = 16807, \qquad m = 2^{31} - 1 = 2147483647$$

Sie führen ebenfalls zu Vollperioden. Für die Berechnung der u_i wird der Datentyp *long* anstelle von *int* benötigt, da bei der Berechnung des Produktes $k \cdot u$ der maximale 'int'-Wert $2^{31}-1$ überschritten werden kann.

Damit erhalten wir einen vollwertigen Zufalls-Generator:

```
// ---------- Zufalls-Generator ----------------------------------------

public class RandomIBM
{  final long k = 16807;                    // 7**5
   final long m = 2147483647;               // 2**31 -1
```

```
  private long u;

  public void initialize(int startwert)
  {  u = startwert;
  }

  public double nextNumber()                    // reelle Zufallszahl in [0,1]
  {  u = ( k * u ) % m;
     return (double) u / m;
  }

  public int nextInt(int max)                   // ganze Zufallszahl in 0 .. max
  {  double x = nextNumber() * (max+1);
     int wert = (int) x;
     if ( wert == max + 1 )                     // Extremfall
        wert = max;
     return wert;
  }

}
```

Man beachte, dass die in der Methode 'nextInt' verwendete reelle Zufallszahl x im Intervall $[0, max + 1]$ erzeugt wird, damit nach dem Abschneiden der Dezimalstellen gleich verteilte Werte im Bereich 0 .. max entstehen.

Der Extremfall $x = max + 1$, der Wahrscheinlichkeit 0 hat, wird sicherheitshalber zum letzten Intervall hinzugenommen werden. Mathematisch (d.h. ohne Rundungsfehler) kann er bei dem verwendeten Algorithmus gar nicht vorkommen, da u maximal gleich $m - 1$ ist, d.h. u/m kleiner als 1 ist.

3.3.4 Suche nach gemeinsamen Elementen

Gegeben sind 3 sortierte verkettete Listen von ganzen Zahlen ≥ 0, z.B. Personen-Nummern von Teilnehmern von 3 Kursen.

Aufgabe:

Bestimmung aller Nummern, die in allen Listen enthalten sind (Durschschnitts-Menge).

Bedingung:

Jede Liste wird nur einmal durchgelesen.

Testdaten:

Wir erzeugen 3 sortierte verkettete Listen mit ganzzahligen Zufallszahlen im Bereich

0 .. 999

Frage:

Wieviele gemeinsame Werte sind zu erwarten, wenn jede Liste 150 Werte enthält ?

Die Antwort der Mathematik lautet: 2.7 im Mittel

Bei Tests sind also einige wenige gemeinsame Elemente zu erwarten. Für einen reproduzierbaren Test verwenden wir ganze Zufallszahlen im Bereich 0 .. 999, die mit der Methode 'nextInt' der Klasse 'RandomIBM' (siehe Aufgabe 3.3.3) erzeugt werden.

Startwert für den Zufallsgenerator: 1000

Resultat:

Gemeinsame Elemente: 7, 634, 731

3.3.5 Berechnung der Höhe eines Baumes

Erzeugen Sie einen sortierten binären Baum mit 1000 ganzen Zufallszahlen im Bereich 1 .. 10000 und bestimmen Sie dessen Höhe.

3.3.6 Erstellung einer Kopie eines binären Baumes

Erstellen Sie eine Methode

```
public Tree createCopy()
```

der Klasse 'Tree', welche eine vollständige Kopie eines Baumes erstellt und als Resultat zurückgibt. Hinweis: verwenden Sie eine Preorder-Traversierung.

Kapitel 4

Vererbung und Interfaces

Es kann nicht alles gesagt werden.
— *N.N.*

4.1 Klassenerweiterungen

Eine Klasse kann eine andere erweitern:

```
class B extends A
{ ...
}
```

Dadurch erbt die Klasse B alle Datenkomponenten und Methoden der Basis-Klasse A.

- In der Erweiterungsklasse können weitere Datenkomponenten und Methoden definiert werden.
- Methoden der Basis-Klasse können bei Bedarf in der Erweiterungsklasse überschrieben werden, d.h. sie werden mit dem gleichen Namen und den gleichen Parametern neu definiert und ersetzen die gleichnamigen Methoden der Basis-Klasse.

Wir kennen das Konzept schon von den Applets: jedes Applet ist eine Erweiterung eines Basis-Applets 'Applet'.

Wir führen die Details an einem konkreten Beispiel ein, welches von einer Basis-Klasse 'Line' für Strecken auf dem Bildschirm ausgeht.

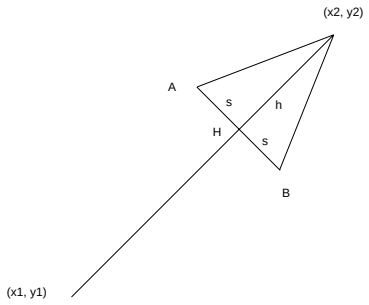
Basis-Klasse 'Line' (Strecken)

```
class Line                                  // Strecken
{  int x1, y1;
   int x2, y2;

   Line(int x1, int y1,                     // Anfangspunkt
        int x2, int y2)                     // Endpunkt
   {  this.x1 = x1;  this.y1 = y1;
      this.x2 = x2;  this.y2 = y2;
   }

   void zeichne(Graphics g)
   {  g.drawLine(x1, y1, x2, y2);
   }

   void translate(int dx, int dy)           // Verschiebung
   {  x1 += dx;  y1 += dy;
      x2 += dx;  y2 += dy;
   }
}
```

Erweiterungsklasse 'Arrow'

Wir führen jetzt eine Erweiterungsklasse 'Arrow' für Pfeile (Strecken mit einer Pfeilspitze) ein.

```
class Arrow extends Line
{  int s, h;      // Pfeilabmessungen

   Arrow(int x1, int x2,
         int y1, int y2,
         int s, int h)
   {  super(x1, y1, x2, y2);
      this.s = s;
      this.h = h;
   }
}
```



Das Schlüsselwort 'super'

Mit dem Schlüsselwort 'super' wird im Konstruktor der Erweiterungsklasse 'Arrow' der Konstruktor der Basis-Klasse 'Line' aufgerufen. Die Bezeichnung 'super' hängt mit dem Begriff 'Superklasse' zusammen, der unten eingeführt wird.

Ein solcher Aufruf eines Konstruktors der Basis-Klasse mittels 'super' ist obligatorisch, und zwar gleich als erstes Statement. So ist gewährleistet, dass in einem Konstruktor der Erweiterungsklasse immer ein Konstruktor der Basis-Klasse aufgerufen wird.

Der 'super'-Aufruf kann nur dann entfallen, wenn die Basis-Klasse einen Konstruktor ohne Parameter enthält, welcher dann automatisch aufgerufen wird.

Bemerkung:

Konstruktoren werden nicht vererbt, d.h. wenn eine Basis-Klasse einen Konstruktor mit Parametern hat muss in einer Erweiterungs-Klasse ein Konstruktor definiert werden, in welchem der ursprüngliche Konstruktor mittels 'super' aufgerufen wird (wie im oberen Beispiel).

Überschreiben von Methoden

Die Zeichen-Methode der Klasse 'Line' ist für die Klasse 'Arrow' nicht genügend, da sie nur eine Strecke ohne Pfeilspitze zeichnet. Sie muss in der Klasse 'Arrow' überschrieben werden. Die Punkt-Bezeichnungen in der Methode entsprechen der Figur auf Seite 64.

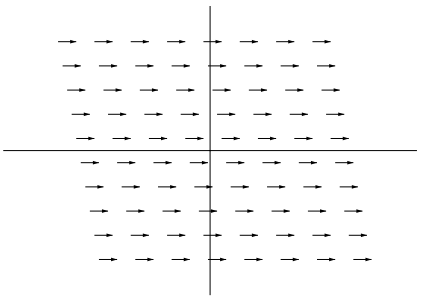
```
void zeichne(Graphics g)
{  double dx, dy, laenge;
   double n1, n2;
   int xH, yH;                                  // Hilfspunkt H
   super.zeichne(g);                            // Basis−Zeichnungsmethode (Strecke)
   // Pfeil−Dreieck zeichnen
   dx = x2−x1;  dy = y2−y1;
   laenge = Math.sqrt(dx*dx + dy*dy);
   dx /= laenge;  dy /= laenge;
   n1 = −dy;  n2 = dx;                          // Normalenrichtung
   xH = x2 − (int) (h * dx);                    // Hilfspunkt auf Pfeilachse
   yH = y2 − (int) (h * dy);
   zeichneDreieck(g,                            // Hilfsmethode
         x2, y2,                                // Spitze
         xH + (int)(s*n1), yH + (int)(s*n2),    // A
         xH − (int)(s*n1), yH − (int)(s*n2));   // B
}

private void zeichneDreieck(Graphics g,
            int x1, int y1, int x2, int y2, int x3, int y3)
{  int[ ] xEcken = { x1, x2, x3 } ;
   int[ ] yEcken = { y1, y2, y3 } ;
   g.fillPolygon( xEcken, yEcken, 3 );
}
```

In der Methode 'zeichne' kommt wieder das Schlüsselwort 'super' zum Einsatz. Mit diesem wird in einer Methode einer Erweiterungsklasse eine überschriebene Methode der Basis-Klasse aufgerufen:

```
super.zeichne(g);                    // Methode der Klasse Line aufrufen
```

Anwendungsbeispiel

Das folgende Applet verwendet die Klassen 'Line' und 'Arrow'. Es zeichnet ein Vektorfeld.



```
// ------------------- Vektor-Feld --------------------
import java.awt.*;
import java.applet.*;
public class VektorFeld extends Applet
{

  public void init()
  { setBackground(Color.blue);
    setForeground(Color.yellow);
  }

  public void paint(Graphics g)
  {
    Dimension wnd = getSize();              // Window-Groesse
    int len=30, s=3, h=10;                  // Pfeil-Abmessungen
    int dx = 2*len, dy = 40;                // Verschiebungen
    int e = dx/8;                           // Einrueckung
    int n1 = 10;                            // Anzahl Zeilen mit Vektoren
    int n2 = 8;                             // Anzahl Vektoren pro Zeile
    int hoehe = (n1-1) * dy;
    int breite = (n2-1) * (dx+e) + s;
    int xM = wnd.width/2;                   // Mittelpunkt
    int yM = wnd.height/2;
```

```
    int xStart = xM − breite/2;
    int yStart = yM − hoehe/2;
    Line xAchse = new Line(0, yM, wnd.width, yM);
    Line yAchse = new Line(xM, 0, xM, wnd.height);
    Arrow a = new Arrow(xStart, yStart, xStart+len, yStart, s, h);
    xAchse.zeichne(g);
    yAchse.zeichne(g);
    for (int i=0; i < n1; i++)
    { for (int j=0; j < n2; j++)
      { a.zeichne(g);                              // Pfeil zeichnen
        a.translate(dx, 0);
      }
      a.translate(−n2*dx, 0);
      a.translate(e, dy);
    }
  }
}
```

△ **Uebung:** Lösen Sie die Aufgabe 4.9.1

Super- und Subklassen

Eine Erweiterungsklasse nennt man auch *Subklasse* der Basis-Klasse. Dies erscheint auf den ersten Blick unangebracht, es wird jedoch natürlich, wenn man die zugehörigen Objekte betrachtet:

Durch die Einführung weiterer Datenkomponenten und Methoden in einer Erweiterungsklasse wird die Menge der Objekte eingeschränkt, d.h. man erhält eine Untermenge (Subklasse) von Objekten der Basis-Klasse.

Beispiel:

Ein Objekt der Klasse 'Arrow' ist eine Strecke mit einem Pfeil, d.h. eine spezielle Strecke. Die Objekte der Klasse 'Arrow' bilden also eine Untermenge aller Strecken. Die Klasse 'Arrow' ist in diesem Sinne eine Subklasse von 'Line'.

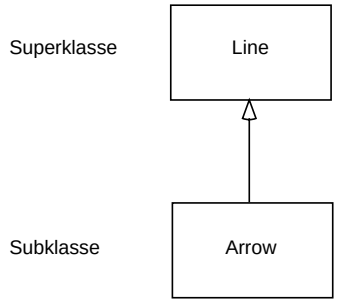
Die Basis-Klasse einer Erweiterungsklasse heisst bei dieser Terminologie *Superklasse*.

Also:

Erweiterungsklasse = Subklasse
Basis-Klasse = Superklasse

Graphische Darstellung (UML Diagramm):

Zur graphischen Darstellung der Beziehungen zwischen Klassen werden Vererbungsdiagramme im UML-Format (Unified Modeling Language) verwendet.



Der Pfeil von einer Subklasse zur Superklasse kann als 'is a' Beziehung gedeutet werden: jeder Pfeil ist auch eine Strecke.

Bemerkungen:

1. Mehrfach-Vererbung

   Java erlaubt (im Gegensatz zu C++) *keine* Mehrfach-Vererbung, d.h. eine Klasse kann nicht mehrere Klassen erweitern. In der 'extends'-Klausel kann also nur *eine* Klasse angegeben werden. Ersatz für Mehrfach-Vererbung sind Interfaces, siehe unten.

2. Mehrstufige Vererbung

   Der Prozess der Klassenerweiterungen kann hingegen über beliebig viele Stufen fortgesetzt werden: Wenn B eine Erweiterungsklasse von A ist, kann man eine weitere Erweiterungsklasse C von B bilden usw.

3. Unterscheidung der Beziehungen 'is a' und 'has a'

   Die Beziehung 'is a' für Subklassen muss unterschieden werden von 'has a' für Zusammensetzungen: ein Objekt kann aus einer oder mehreren Strecken zusammengesetzt sein, d.h. es hat als Datenkomponenten Objekte der Klasse 'Line'.

   Dazu werden keine Erweiterungsklassen (Subklassen) benötigt.

   Beispiel: Basis-Klasse 'String'.

Eine Klasse, z.B. für Personen, die als Datenkomponente einen String (Name) verwendet, ist deshalb keine Erweiterung der Klasse 'String'.

Zugriffs-Kontrolle

Bei der Definition von Datenkomponenten und Methoden einer Klasse können die folgenden Attribute zur Festlegung des Zugriffs angegeben werden:

Attribut	Bedeutung
private	Die Datenkomponente bzw. Methode kann nur von Methoden derselben Klasse verwendet werden.
public	Keine Einschränkung, Verwendung von allen Methoden beliebiger Klassen zugelassen
keine Angabe	Verwendung von allen Methoden desselben Packages zugelassen
protected	Verwendung von allen Methoden desselben Packages und von Methoden von Subklassen zugelassen.

4.2 Polymorphismus

Für Erweiterungsklassen gilt das folgende Prinzip: Eine Referenzvariable für ein Objekt der Basis-Klasse kann auch Objekte einer Erweiterungsklasse referenzieren:

```
Line line;                                  // Referenzvariable fuer Superklasse
Arrow arrow = new Arrow(...);
line = arrow;                               // zulaessige Zuweisung
```

Die Referenzvariable 'line' referenziert jetzt also ein Objekt der Klasse 'Arrow'. Dies hat Konsequenzen bei Methoden-Aufrufen.

Methoden-Aufrufe

Da die Referenzvariable 'line' jetzt ein Objekt 'Arrow' referenziert, stellt sich die Frage, was bei dem Statement

```
line.zeichne(g);
```

gezeichnet wird, eine Strecke oder ein Pfeil.

Antwort:

Es wird ein *Pfeil* gezeichnet, weil das von 'line' referenzierte Objekt tatsächlich ein Arrow ist.

Prinzip:

Beim Aufruf einer Methode eines Objektes p ist nicht der Datentyp der Referenzvariablen p massgebend für die Wahl der aufgerufenen Methode, sondern der Typ des referenzierten Objektes selber.

Polymorphismus

Diesen Sachverhalt nennt man *Polymorphismus* oder *Vielgestaltigkeit*, da ein fester Methoden-Aufruf verschiedene Wirkungen haben kann:

```
Line line;
Line achse = new Line(...);
Arrow arrow = new Arrow(...);
line = achse;
line.zeichne(g);                    // Strecke wird gezeichnet
line = arrow;
line.zeichne(g);                    // Pfeil wird gezeichnet
```

Abfrage des Datentyps eines Objektes

Im Zusammenhang mit Polymorphismus ist es in gewissen Situationen erforderlich, den Typ eines Objektes abzufragen. Dies erfolgt mit dem Operator 'instanceof': *instanceof*

```
if ( line instanceof Arrow )  ...
```

Auf der linken Seite des Operators steht eine Referenzvariable für ein Objekt und auf der rechten Seite ein Klassenname.

Explizite Konversionen von Referenzvariablen

Referenzvariablen können explizite konvertiert werden, mit der gleichen Syntax wie bei numerischen Konversionen, d.h. der neue Datentyp wird in runden Klammern vor den Ausdruck gesetzt:

```
Arrow a;
a = (Arrow) line;
```

▶ *Merke:*

Bei einer solchen Konversion wird zur Laufzeit geprüft, ob die betreffende Variable *line* auf ein Objekt des Zieltyps zeigt. Ist dies nicht der Fall, wird die Exception 'ClassCastException' ausgelöst und das Programm abgebrochen.

4.3 Allgemeine Klassen für verkettete Listen

Das Konzept der Klassenerweiterungen eignet sich zur Erstellung von allgemein verwendbaren Klassen für verkettete Listen, Trees usw.

Prinzip:

- Für die Elemente werden allgemeine Klassen definiert, welche nur die benötigten Referenzen, aber keine Daten enthalten.
- Für eine konkrete Anwendung wird eine Erweiterungsklasse für die Elemente mit den benötigten Datenkomponenten definiert.

Die Details führen wir an Beispielen ein.

Allgemeine Stacks

Wir definieren eine Basis-Klasse 'StackEl' für die Elemente von Stacks und eine Klasse 'Stacks' mit den Mehoden 'push' und 'pop'.

```
// ___________ Stack−Element ____________________
public class StackEl
{
  StackEl next;
}

// ___________ Stack ____________________________
public class Stack
{
  StackEl first;

  public void push(StackEl e)
  { e.next = first;
    first = e;
  }

  public StackEl pop()
  { StackEl temp = first;
    if ( first != null )
    { first = first.next;
      temp.next = null;
    }
    return temp;
  }
}
```

Anwendungs-Beispiel:

Das folgende Beispiel erzeugt einen Stack von reellen Zahlen. Dazu wird eine Erweiterungsklasse 'DoubleEl' definiert.

```
class DoubleEl extends StackEl
{  double data;
   public DoubleEl(double wert)                    // Konstruktor
   {  data = wert;
   }
}

public class StackTest
{
   static public void main(String[ ] args)
   {  Stack theStack = new Stack();
      DoubleEl e;
      theStack.push(new DoubleEl(3));
      theStack.push(new DoubleEl(1));
      theStack.push(new DoubleEl(5));
      e = (DoubleEl) theStack.pop();
      while ( e != null )
      {  System.out.println(e.data);               // Ausgabe
         e = (DoubleEl) theStack.pop();
      }
   }
}
```

Man beachte, dass die von der 'pop'-Methode erhaltene Referenz in eine 'DoubleEl'-Referenz konvertiert werden muss, da das Resultat von 'pop' eine Referenz des Typs 'StackEl' ist.

Allgemeine sortierte verkettete Listen

Bei sortierten verkettete Listen stellt sich ein neues Problem: die Listen-Elemente müssen vergleichbar sein ('kleiner oder gleich'). Zu diesem Zweck wird in der Basis-Klasse 'ListEl' eine leere Vergleichsmethode 'compareTo' eingeführt.

Diese muss in einer Anwendung der Klasse in der Erweiterungsklasse überschrieben werden durch eine richtige Vergleichsfunktion.

```
// ___________ Element fuer sortierte verkettete Liste
public class ListEl
{ ListEl next;
  public int compareTo(ListEl e) ;                    // Vergleichs-Methode
  { return 0;
  }
}

// ___________ Sortierte verkettete Liste ___________
public class List
{ ListEl first;
  public void insert(ListEl e)
  { ListEl p, prev;
    if ( first == null )
      first = e;
    else
    { p = first;
      prev = null;
      while ( p != null &&
                  p.compareTo(e) <= 0 )               //  p <= e ?
      { prev = p;
        p = p.next;
      }
      e.next = p;
      if ( prev == null )
        first = e;                                    // Einfuegung als erstes El.
      else
        prev.next = e;                                // Einfuegung nach prev
    }
  }
```

Anwendung der Klasse:

```
class DoubleEl extends ListEl                    // Erweiterungsklasse für Listen−Elemente
{  double key;
   public DoubleEl(double key)
   {  this.key = key;
   }
   public int compareTo(ListEl e)                // Vergleichsmethode
   {  DoubleEl ee = (DoubleEl) e;
      return key − ee.key;
   }
}

public class ListTest                            // Test−Applikation
{
   static public void main(String[ ] args)
   {  List list = new List();
      DoubleEl e;
      list.insert(new DoubleEl(3.4));
      list.insert(new DoubleEl(1.2));
      list.insert(new DoubleEl(5.8));
      e = (DoubleEl) list.first;
      while ( e != null )
      {  System.out.println(e.key);              // Ausgabe
         e = (DoubleEl) e.next;
      }
   }
}
```

Die Klasse hat einen konzeptuellen Mangel. Wenn man in einer Anwendung vergisst, die Vergleichsmethode zu überschreiben, wird die der Basis-Klasse verwendet, welche zu keiner Sortierung führt.

Dieser Mangel kann mit abstrakten Klassen behoben werden.

4.4 Abstrakte Klassen und Methoden

In einer Klasse kann eine Methode mit dem Attribut 'abstract' versehen werden. Dann hat sie in dieser Klasse keinen Code, sie ist nur spezifiziert, mit Name, Parametern und Resultat-Typ:

```
public abstract int compareTo(ListEl e);       // Spezifikation
```

Eine Klasse mit abstrakten Methoden heisst *abstrakte Klasse*. Sie muss ebenfalls mit dem Attribut 'abstract' definiert werden:

```
public abstract class ListEl
{ ListEl next;
  public abstract int compareTo(ListEl e);    // Spezifikation
}
```

Konsequenzen:

- Zu einer abstrakten Klasse können keine Objekte erzeugt werden, es muss zuerst eine Erweiterungsklasse definiert werden.
- Eine abstrakte Methode *muss* in einer Erweiterungsklasse vollständig definiert werden.

Damit ist der Mangel der Klasse für die Elemente einer sortierten verketteten Liste behoben. Für die Klasse 'List' und für die Anwendung ergeben sich keine Änderungen.

▶ *Merke:*

Eine abstrakte Klasse ist eine Klasse mit mindestens einer abstrakten Methode. Eine abstrakte Methode hat keinen Code, sie besteht nur aus einer Spezifikation. In der Erweiterungsklasse muss sie vollständig definiert werden.

4.5 Die Klasse 'Object'

Jede Klasse ist in Java automatisch eine Erweiterung einer obersten Superklasse, der Klasse 'Object'. Für diese können ebenfalls Referenzvariablen definiert werden:

```
Object obj;
```

Eine solche Referenzvariable kann beliebige Objekte referenzieren. Dies erlaubt einen anderen Ansatz für allgemeine Klassen für Stacks und Queues:

Stacks mit allgemeinen Objekt-Referenzen

```
public class StackEl2                          // Stack−Elemente
{ StackEl2 next;
  Object obj;
}

public class Stack2                            // Stacks
{ StackEl2 first;

  public void push(Object obj)
  { StackEl2 e = new StackEl2();
    e.obj = obj;
    e.next = first;
    first = e;
  }

  public Object pop()
  { StackEl2 temp = first;
    if ( first != null )
    { first = first.next;
      temp.next = null;
    }
    if ( temp != null )
      return temp.obj;
    else
      return null;
  }
}
```

Anwendungsbeispiel

Die folgende Applikation erstellt einen Stack mit String-Objekten. Man beachte, dass bei diesem Konzept keine Erweiterung der Klasse für die Stack-Elemente erforderlich ist.

```
public class StackTest2
{
  static public void main(String[] args)
  { String s;
    Stack2 theStack = new Stack2();
    theStack.push("Meier");
    theStack.push("Mueller");
    theStack.push("Huber");
    s = (String) theStack.pop();               // Ausgabe
    while ( s != null )
    { System.out.println(s);
      s = (String) theStack.pop();
    }
  }
}
```

Bei diesem Konzept sieht der Anwender nur die eigentlichen Daten der Stack-Elemente, die Referenzen sind eingekapselt in die Klasse StackEl2, mit welcher der Anwender direkt nichts zu tun hat.

Das Konzept setzt voraus, dass die Daten der Elemente in einem Objekt verpackt sind, auch wenn es nur 'int'- oder 'double'-Werte sind.

Wenn die Daten vergleichbar sein müssen, z.B. bei sortierten Listen oder Suchbäumen, werden für diesen Ansatz zusätzlich Interfaces benötigt.

△ **Uebung:** Lösen Sie die Aufgabe 4.9.2

4.6 Interfaces

Das Konzept der Interfaces ist eine geniale Erfindung der Entwickler von Java. Interfaces sind Spezifikationen von Schnittstellen für Objekte. Unter einer *Schnittstelle* eines Objektes versteht man einen Satz von aufrufbaren Methoden.

Ein Objekt, welches eine bestimmte Schnittstelle aufweist, kann überall verwendet werden, wo die betreffenden Methoden benötigt werden, z.B. als Parameter einer Methode.

Die Einführung von Interfaces eröffnet neue Möglichkeiten für die Definition von allgemein verwendbaren Methoden und Objekten.

Was ist ein Interface ?

Ein Interface enthält *Spezifikationen* von Methoden. Eine Spezifikation einer Methode besteht aus dem Namen der Methode, der Parameter-Liste und dem Resultat-Typ. Sie enthält jedoch keinen Code.

Wir kennen schon die Interfaces für die Event-Verarbeitung, z.B. das Interface 'KeyListener'. Dieses ist folgendermassen definiert:

```
public interface KeyListener
{ void KeyPressed(KeyEvent e);
  void KeyReleased(KeyEvent e);
  void KeyTyped(KeyEvent e);
}
```

Ein Applet, welches das Interface implementiert, verpflichtet sich, alle Methoden des Interfaces vollständig zu definieren. Es besitzt dann die Schnittstelle 'KeyListener', welche bei Keyboard-Events verwendet wird.

Wozu ist ein Interface ?

Technisch gesehen, definiert ein Interface einen *Referenztyp*, d.h. einen Datentyp für Referenzvariablen (wie eine Klasse).

Dieser Referenztyp kann wie ein Referenztyp einer Klasse verwendet werden, z.B. als Datentyp für einen Parameter einer Methode. Beim Aufruf dieser Methode kann dann ein beliebiges Objekt übergeben werden, welches das Interface implementiert.

So können Methoden realisiert werden, die sehr allgemeine Parameter zulassen. Wir führen die Details zur Definition und Verwendung eines Interfaces an einem Beispiel ein.

Ein Interface für Punkt-Transformationen

In der Ebene gibt es verschiedene Arten von Transformationen: Drehungen, Verschiebungen, Streckungen, Spiegelungen usw. Es sind Punkt-Transformationen, d.h. sie bilden einen beliebigen Punkt auf einen Bildpunkt ab.

A) Definition des Interfaces

Wir führen ein Interface 'PunktTransformation' ein mit der Spezifikation einer Methode 'bildPunkt' :

```
public interface PunktTransformation
{ Punkt2d bildPunkt(Punkt2d p);          // Bildpunkt von p
}
```

Die Methode 'bildPunkt' erhält als Parameter einen Punkt und liefert als Resultat einen neuen, den Bildpunkt. Wie dieser Punkt berechnet wird, ist im Moment offen gelassen.

Bemerkungen:

- Die in einem Interface spezifizierten Methoden erhalten automatisch das Attribut 'public'.
- Ein Interface wird (wie eine Klasse) im zugehörigen File '.java' abgespeichert und compiliert. Dabei entsteht ein '.class'-File mit dem Byte-Code des Interfaces.
- Das Interface ermöglicht die Definition von Referenzvariablen:

  ```
  PunktTransformation transf;          // Referenzvariable
  ```

  Die Referenzvariable 'transf' kann beliebige Objekte referenzieren, die das Interface 'PunktTransformation' implementieren (s. unten).

  Der 'new'-Operator zur Erzeugung eines Objektes steht für Interfaces *nicht* zur Verfügung.

B) Implementierung des Interfaces

Eine Klasse, die ein Interface implementiert verpflichtet sich, alle Methoden des Interfaces (vollständig) zu definieren. Wir betrachten zwei solche Klassen für Streckungen und Drehungen:

```
public class Streckung                          // Streckung in der Ebene
             implements PunktTransformation
{  double q;                                    // Streckungsfaktor
   public Streckung(double faktor)
   {  q = faktor;
   }
   public Punkt2d bildPunkt(Punkt2d p)          // Bild eines Punktes
   {  Punkt2d pp = new Punkt2d(q * p.x, q * p.y);
      return pp;
   }
}

public class Drehung                            // Drehung um den Nullpunkt
             implements PunktTransformation
{  double cos, sin;
   public Drehung(double winkel)                // Drehwinkel in Grad
   {  double phi = winkel * Math.PI / 180;      // Bogenmass
      cos = Math.cos(phi);
      sin = Math.sin(phi);
   }
   public Punkt2d bildPunkt(Punkt2d p)
   {  Punkt2d pp = new Punkt2d(cos*p.x - sin*p.y,
                               sin*p.x + cos*p.y);
      return pp;
   }
}
```

C) Verwendung des Interfaces

Der Name eines Interfaces kann (wie der Name einer Klasse) als Datentyp für Parameter einer Methode verwendet werden. Beim Aufruf der Methode kann ein beliebiges Objekt, welches das Interface implementiert, übergeben werden.

Wir betrachten eine Klasse 'Polygon' für Vielecke in der Ebene:

```
import java.awt.*;
public class Polygon                            // Vielecke in der Ebene
{  Punkt2d [ ] ecken;                           // Eckpunkt−Koordinaten
   public Polygon(int nEcken)
   {  ecken = new Punkt2d[nEcken];
   }
   public void transform
          (PunktTransformation transf)          // Interface als Parameter−Typ
   {  for (int i=0; i < ecken.length; i++)
         ecken[i] = transf.bildPunkt(ecken[i]);
   }
```

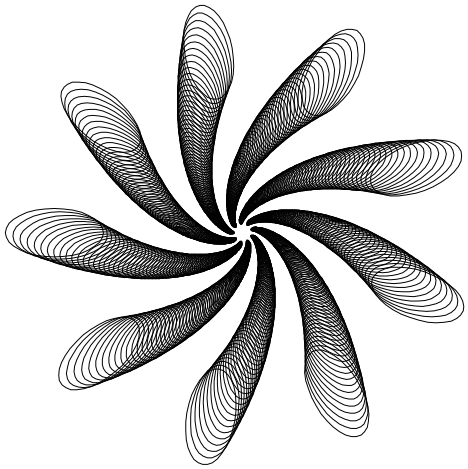
```
  public void zeichne(Graphics g, Graph2d graph2d)
  { int[ ] x = new int[ecken.length];
    int[ ] y = new int[ecken.length];
    for (int i=0; i < ecken.length; i++)
    { x[i] = graph2d.pixCol(ecken[i].x);
      y[i] = graph2d.pixLine(ecken[i].y);
    }
    g.drawPolygon(x, y, ecken.length);
  }
}
```

Die Methode 'transform' erhält als Parameter eine Punkt-Transformation, mit welcher die Eckpunkte des Polygons transformiert werden.

Beim Aufruf der Methode kann ein beliebiges Objekt übergeben werden, welches das Interface 'PunktTransformation' implementiert. So können beliebige Transformationen übergeben werden (Streckungen, Drehungen usw.)

Anwendungsbeispiel:

Wir betrachten ein Applet 'Ellipsen', welches mit der Klasse 'Polygon' die folgende Figur aus Ellipsen zeichnet:



Die Figur entsteht durch wiederholte Drehung und Streckung einer Ellipse. Die erste Ellipse ist die grosse Ellipse auf der negativen x-Achse.

Diese wird bei jedem Schritt um 40.1 Grad im Gegenuhrzeigersinn gedreht und mit dem Streckungsfaktor 0.9965 verkleinert.

```
import java.awt.*;
import java.applet.*;
public class Ellipsen extends Applet
{
  final double breite=20.0;
  final int nEllipsen = 800;
  final int nEcken = 32;
  final double a = 1.4;                          // grosse Halbachse
  final double b = 0.7;                          // kleine Halbachse
  final double xM = −5.4;                        // Start−Ellipse
  final double yM = 0;
  Streckung streckung = new Streckung(0.9965);
  Drehung drehung = new Drehung(40.1);

  Polygon berechneEllipse(double xM, double yM, // Ausgangs−Ellipse
              int nEcken, double a, double b)
  { Polygon ellipse = new Polygon(nEcken);
    double phi = 2 * Math.PI / nEcken;
    for (int i=0; i < nEcken; i++)
      ellipse.ecken[i] = new Punkt2d(xM + a * Math.cos(i*phi),
                                     yM + b * Math.sin(i*phi));
    return ellipse;
  }

  public void init()
  { setBackground(Color.blue);
    setForeground(Color.yellow);
  }

  public void paint(Graphics g)
  { double hoehe;
    Dimension wnd = getSize();
    Graph2d graph2d = new Graph2d(wnd.width, wnd.height);
    hoehe = breite * wnd.height / wnd.width;
    graph2d.setDisplayRange(−0.5*breite, 0.5*breite,
                            −0.5*hoehe, 0.5*hoehe);
    Polygon ellipse = berechneEllipse(xM, yM, nEcken, a, b);
    ellipse.zeichne(g, graph2d);
    for (int i=0; i < nEllipsen; i++)
    { ellipse.transform(drehung);                // drehen
      ellipse.transform(streckung);              // strecken
      ellipse.zeichne(g, graph2d);               // zeichnen
    }
  }
}
```

Interfaces und Klassenerweiterungen

Ein Interface ist im Prinzip eine abstrakte Klasse mit lauter abstrakten Methoden. Durch die Implementation eines Interfaces wird wie bei einer Klassenerweiterung eine Relation 'is a' definiert:

Wenn eine Klasse das Interface 'PunktTransformation' implementiert, ist jedes Objekt der Klasse eine PunktTransformation.

Der Unterschied zu Klassenerweiterungen ist der, dass mehrere Interfaces implementiert werden können (zusätzlich zu einem eventuellen 'extends'):

```
public class xxxx implements A, B, C
```

Interfaces für die Ereignisverarbeitung

Bei der Verarbeitung von Ereignissen hatten wir schon Kontakt mit Interfaces, u.a. mit dem Interface 'ActionListener' für Buttons.

Jetzt können wir den Mechanismus der Ereignis-Verarbeitung vollständig verstehen.

Wenn ein Applet Messages von einem Button erhalten will, muss es das Interface 'ActionListener' implementieren, welches die Spezifikation der Methode 'actionPerformed' enthält:

```
public interface ActionListener
{ void actionPerformed(ActionEvent e);
}
```

Die Implementation dieses Interfaces ist die Voraussetzung dafür, dass sich das Applet bei einem Button-Objekt als 'ActionListener' anmelden kann:

```
Button button = new Button("ok");
button.addActionListener(this);
```

Der Parameter der Methode 'addActionListener' hat den Datentyp 'ActionListener', d.h. die Methode erwartet als Parameter ein Objekt, welches das Interface 'ActionListener' implementiert.

Dadurch kann der Button bei einem Maus-Klick dem Applet eine Message senden, indem er die Methode 'actionPerformed' des Applets aufruft.

4.7 Eine allgemeine Klasse für binäre Bäume

Die folgende Klasse 'Tree' für sortierte binäre Bäume ist nach dem gleichen Prinzip wie die frühere Klasse für sortierte verkettete Listen aufgebaut, mit einer abstrakten Klasse für die Elemente.

Die Klasse enthält eine Methode 'traverse', welche einen Baum traversiert und für jeden Knoten eine bestimmte Verarbeitung durchführt. Die gewünschte Verarbeitung wird mit einem Interface übergeben.

```
// ___________ Interface fuer Methode 'traverse' ____
public interface NodeVerarbeitung
{  void verarbeite(Node node);
}

// ___________ Nodes for binary Trees _______________
public abstract class Node
{  public Node left, right;

   public abstract int compareTo(Node node);

}

// ___________ Binary Trees _____________________________
public class Tree
{  public Node root;                              // Root−Element

   public void insert(Node node)                  // Einfügung
   {  Node p;
      if ( root == null )
         root = node;
      else
      {  p = searchLeaf(node);                    // Einfuegeposition
         if ( node.compareTo(p) <= 0 )            // node.key <= p.key ?
            p.left = node;                        // Einfuegung links
         else
            p.right = node;                       // Einfuegung rechts
      }
   }

   private Node searchLeaf(Node node)             // Einfuegeposition bestimmen
   {  Node p = root, prev = null;
      while ( p != null )
      {  prev = p;
         if ( node.compareTo(p) <= 0 )            // node.key <= p.key ?
            p = p.left;                           // nach links
         else
            p = p.right;                          // nach rechts
      }
      return prev;
   }
```

```
  public Node search(Node node)                    // Suche
  { Node p = root;
    while ( p != null && p.compareTo(node) != 0 )
      if ( p.compareTo(node) >= 0 )                // p.key >= node.key ?
        p = p.left;                                // weiter im linken Teilbaum
      else
        p = p.right;                               // weiter im rechten Teilbaum
      return p;
  }

  public void traverse(NodeVerarbeitung nv)
  { traverseRekursiv(root, nv);
  }

  private void traverseRekursiv(Node root, NodeVerarbeitung nv)
  {
    if ( root != null )
    { traverseRekursiv(root.left, nv);
      nv.verarbeite(root);
      traverseRekursiv(root.right, nv);
    }
  }
}
```

Anwendungsprogramm:

```
// ___________ Binary-Tree Test _________________
class IntNode extends Node                         // Erweiterung des Standard-Nodes
{ int key;

  IntNode(int key)
  { this.key = key;
  }

  public int compareTo(Node node)
  { IntNode n = (IntNode) node;
    return key - n.key; }
  }

  class IntNodeVerarb
              implements NodeVerarbeitung
  {
    public void verarbeite(Node node)
    { IntNode n = (IntNode) node;
      System.out.println(n.key);
    }
  }
```

```
public class TreeTest
{
  static public void main(String[ ] args)
  {
    Tree tree = new Tree();
    IntNode n1 = new IntNode(3);
    IntNode n2 = new IntNode(1);
    IntNode n3 = new IntNode(5);
    IntNode n;
    tree.insert(n1);
    tree.insert(n2);
    tree.insert(n3);
    tree.traverse(new IntNodeVerarb());
    n = (IntNode) tree.search(new IntNode(5));
    if ( n != null )
      System.out.println("returned key: " + n.key);
    else
      System.out.println("not found");
    }
  }
```

4.8 Interface für vergleichbare Objekte

Wir führen ein Interface 'ComparableObj' für vergleichbare Objekte ein:

```
public interface ComparableObj
{ int compareTo(ComparableObj obj);
}
```

Mit diesem Interface können diverse Anwendungen realisiert werden, die allgemeine Objekte zulassen, welche eine Vergleichsfunktion zur Verfügung stellen. Ein typisches Beispiel einer solchen Anwendung ist das Sortieren eines Arrays von Objekten.

Eine Klasse zum Sortieren eines Arrays von Objekten

Die folgende Klasse ermöglicht die Sortierung eines beliebigen Arrays von Objekten, die das Interface 'ComparableObj' implementieren.

```
public class ArraySort
{
  static public void qSort( ComparableObj[ ] a )    // Quick−Sort
  { qSortRekursiv(a, 0, a.length−1);
  }

  static private void qSortRekursiv( ComparableObj[ ] a, int left, int right )
  { int up, down, m;
    ComparableObj w, tmp;
    if ( left >= right )
      return;                                       // nichts zu tun
    // −−−−−− Unterteilung −−−−−−−−
    m = (left+right) / 2;
    w = a[m];                                       // Wert fuer Vorsortierung
    up = left;  down = right;
    while ( up <= down )
    { while( a[up].compareTo(w) < 0)                // a[up] < w ?
        up++;
      while( a[down].compareTo(w) > 0)              // a[down] > w ?
        down−−;
      if( up <= down )
      { tmp = a[up];                                // Vertauschung a[up] und a[down]
        a[up] = a[down];
        a[down] = tmp;
        up++;  down−−;
      }
    }
  // −−−−−− linken und rechten Teil sortieren −−−−−−
  if ( down > left ) qSortRekursiv(a, left, down);
  if ( up < right ) qSortRekursiv(a, up, right);
  }
}
```

Anwendungsbeispiel:

Wir verwenden die Klasse zur Sortierung eines Arrays von Kalenderdaten.

```
class Datum implements ComparableObj              // Kalender−Daten
{
  int jahr, monat, tag;

  public Datum(int j, int m, int t)               // Konstruktor
  { jahr = j;
    monat = m;
    tag = t;
  }

  public int compareTo(ComparableObj obj)         // Vergleichsmethode
  { Datum d = (Datum) obj;
    if ( jahr < d.jahr ) return −1;
    else if ( jahr > d.jahr ) return 1;
```

```
    if ( monat < d.monat ) return −1;
    else if ( monat > d.monat ) return 1;
    return tag − d.tag;
  }

  void ausgabe()
  { System.out.println(tag + "." + monat + "." + jahr);
  }
}



public class SortTest
{
  static Datum[ ] dates = { new Datum(2000, 3, 12),
                            new Datum(1999, 8, 15),
                            new Datum(2000, 5, 22),
                            new Datum(2000, 5, 12) } ;
  static public void main( String[ ] args )
  {
    ArraySort.qSort(dates);                          // Sortierung
    for ( int i=0; i < dates.length; i++ )
      dates[i].ausgabe();
  }
}
```

Bemerkung:

In den Java-Versionen ab 1.2 ist ein solches Interface für vergleichbare Objekte integriert ('Comparable'), und die Standard-Klassen 'String' usw. implementieren dieses.

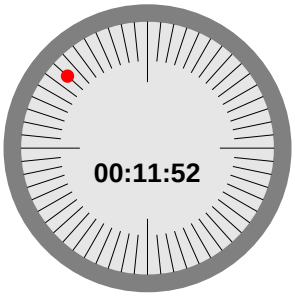
4.9 Übungen

4.9.1 Erweiterte Stop-Uhren

Erstellen Sie Erweiterungen der Klasse 'Clocks' (siehe Seite 37) für Stop-Uhren mit digitaler bzw. analoger Anzeige auf dem Bildschirm. Führen Sie für die Anzeige eine Methode 'show' ein:

```
void show(Graphics g)
```

Vorschlag für analoge Anzeige (Sekunden analog, vollständige Zeit digital):



4.9.2 Queues mit allgemeinen Objekt-Referenzen

Erstellen Sie eine Klasse 'Queue2' für Queues von allgemeinen Objekten, nach dem Konzept, bei dem die Elemente der Queues neben der Referenz 'next' eine Referenz für ein beliebiges Objekt (Klasse 'Object') enthalten.

Methoden der Klasse 'Queue2' :

```
void enqueue(Object obj)
Object dequeue()
boolean isEmpty()
```

Die Methode 'isEmpty' prüft, ob die Queue leer ist.

Testprogramm:

Erstellen Sie ein Testprogramm, welches 4 Strings in eine Queue speichert und wieder liest.

Kapitel 5

Exception-Handling

Whatever can go wrong, will go wrong.

— Murphy's Law

5.1 Das Konzept der Exceptions

Exceptions sind ein Hilfsmittel für eine gesicherte Behandlung von Fehler-Situationen, d.h. die Fehler können nicht unentdeckt bleiben. Wir kennen Exceptions des Runtime-Systems, z.B.

ArrayOutOfBoundsException.

Diese wird aktiviert, wenn ein Element eines Arrays mit einem ungültigen Index angesprochen wird. Dadurch wird das Programm mit einer entsprechenden Fehlermeldung abgebrochen (sofern kein Exception-Handler für die Exception vorhanden ist, siehe unten).

Exceptions können auch selber definiert und in Fehlersituationen ausgelöst werden.

Definition einer Exception

Eine Exception wird mit einem beliebigen Namen als Erweiterung der Klasse 'Exception' definiert:

```
public class LengthException extends Exception
{
  public LengthException()                    // Konstruktor
  { super("ungueltige Laenge");               // Basis−Konstruktor aufrufen
  }
}
```

Auslösung einer Exception

throw

Eine Exception wird mit dem Statement 'throw' ausgelöst. Dabei muss ein Objekt der betreffenden Exception-Klasse angegeben werden (ohne Klammern, da 'throw' ein Statement, keine Methode ist):

```
LengthException e;

if ( ... )
{ e = new LengthException();
  throw e ;
}
```

Wirkung des 'throw'-Statements:

Beim 'throw'-Statement wird der normale Programm-Ablauf unterbrochen, und es erfolgt eine der folgenden Aktionen:

- Wenn in der laufenden Methode ein Exception-Handler für die Exception vorhanden ist, verzweigt der Programmablauf zu diesem (mit einem 'goto'-Sprungbefehl). Die Definition von Exception-Handlers wird unten eingeführt.
- Liegt in der laufenden Methode kein Exception-Handler für die Exception vor, so wird die Methode beim 'throw'-Statement abgebrochen, und die Exception wird an die aufrufende Methode zur weiteren Behandlung weitergeleitet.

Wenn in einer Methode Exceptions auftreten können, die an die aufrufende Methode weitergeleitet werden, müssen diese im Kopf der Methode mittels 'throws' angegeben werden.

Beispiel:

Die folgende Methode berechnet das Skalarprodukt zweier Vektoren. Bei unterschiedlichen Längen der Vektoren wird eine Exception ausgelöst.

```
static double skalarProd                    // Skalar−Produkt
            ( double[ ] a, double[ ] b)
            throws LengthException          // Exception weiterleiten
{ double s = 0;
  if ( a.length != b.length )
    throw new LengthException();
  for (int i=0; i < a.length; i++)
    s += a[i]*b[i];
  return s;
}
```

Wenn die Vektoren a und b unterschiedliche Längen haben, wird die Exception 'LengthException' ausgelöst.

Dadurch wird die Methode 'skalarProd' nach dem 'throw'-Statement abgebrochen und die Exception an die aufrufende Methode weitergeleitet.

Die aufrufende Methode muss die Exception behandeln, oder ebenfalls weiterleiten, z.B.

```
static public void main( String[ ] args )
            throws LengthException
{  double[ ] a, b ;
   ...
   double s = skalarProd(a, b);
   ...
}
```

Wenn die Exception ausgelöst wird, wird sie an den Java Interpreter weitergeleitet, welcher das Programm abbricht.

Bemerkungen:

- Wenn in einer Methode eine Exception ausgelöst werden kann (direkt, oder durch Weiterleitung von einer aufgerufenen Methode), so muss die Exception behandelt oder mittels 'throws' im Kopf der Methode weitergeleitet werden.
- Wichtige Exceptions des Runtime-Systems, die praktisch überall auftreten können, wie 'ArrayOutOfBoundsException' usw., müssen nicht behandelt und auch nicht explizite weitergeleitet werden.

  Wenn diese Exceptions nicht behandelt werden, werden sie automatisch weitergeleitet.
- Wenn eine Methode vom Runtime-System aufgerufen wurde, und eine Exception weiterleitet, wird das Programm (genauer der momentane Thread) mit einer entsprechenden Fehlermeldung abgebrochen.

5.2 Exception-Handlers

Die Behandlung von Exceptions erfolgt mit einem 'try/catch'-Statement, welches das folgende Format aufweist:

```
try
{ ...
}
catch ( Exception1 e )
{ ...                                   // Behandlung Exception1
}
catch ( Exception2 e )
{ ...                                   // Behandlung Exception1
}
```

Funktionsweise:

Die Anweisungen in den Klammern nach 'try' werden ausgeführt (der sogenannte 'try'-Block). Dabei sind zwei Fälle zu unterscheiden:

(a) Wenn im 'try'-Block eine Exception auftritt (durch ein 'throw'-Statement oder infolge eines Methoden-Aufrufs), so wird der 'try'-Block abgebrochen und die 'catch'-Statements werden geprüft:

- Wenn ein 'catch'-Statement vorhanden ist, welches die Exception behandelt, werden die betreffenden Anweisungen ausgeführt, anschliessend läuft das Programm *nach* den 'catch'-Statements normal weiter.
- Wenn kein 'catch'-Statement für die Exception vorhanden ist, wird die momentane Methode abgebrochen und die Exception an die aufrufende Methode weitergeleitet.

(b) Wenn im 'try'-Block keine Exception auftritt, haben die 'catch'-Statements keine Bedeutung, der Programmablauf läuft nach den 'catch'-Statements normal weiter.

Beispiel:

Wenn beim Einlesen einer ganzen Zahl mit der Methode 'InOut.getInt' etwas nichtnumerisches eingegeben wird, wird die Exception 'NumberFormatException' des Runtime-Systems ausgelöst. Dies ist eine Exception, die nicht behandelt werden muss.

Das folgende Programm liest einen ganzzahligen Wert mit Exception-Handling ein. Im Falle einer ungültigen Eingabe kommt ein Exception-Handler zum Einsatz, der bewirkt, dass die Eingabe wiederholt werden kann.

```
// ___________ Input mit Exception−Handling _____
public class ReadInt
{
  static int input()                                  // Input mit Fehler−Handling
  { int wert = 0;
    boolean done = false;
    while ( !done )
    { try
      { wert = InOut.getInt();
        done = true;
      }
      catch ( NumberFormatException e )
      { InOut.println("ungueltige Eingabe");
        InOut.print("Wert: ");
      }
    }
    return wert;
  }
  static public void main(String[ ] args)
  { int n;
    InOut.println("ganzzahliger Wert: ");
    n = input();
    InOut.println("Eingabewert: " + n);
  }
}
```

Die 'finally'-Klausel

Nach den 'catch'-Zeilen eines 'try'-Statements kann noch eine 'finally'-Klausel angefügt werden.

```
try
{ ...
}
catch ( Exception1 e )
{ ...                                   // Behandlung Exception1
}
catch ( Exception2 e )
{ ...                                   // Behandlung Exception1
}
finally
{ ...
}
```

Die Statements der 'finally'-Klausel werden in jedem Fall ausgeführt, sowohl bei einem normalen Ende des 'try'-Blocks, als auch im Fall einer Exception. Wenn im Falle einer Exception ein 'catch'-Zweig zum Einsatz kommt wird dieser vor der 'finally'-Verarbeitung ausgeführt.

Die 'finally'-Klausel ist also für Aktionen die in jedem Falle durchgeführt werden müssen.

—— All's well that ends ——

Teil II

Klassische Anwendungen

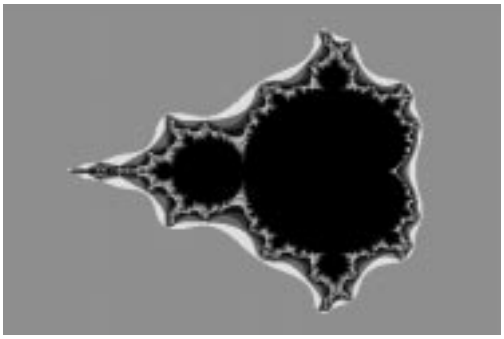
Anwendung 1

Mandelbrot-Fraktale

Die Mandelbrot-Menge ist das berühmteste Fraktal. Sie ist durch einen einfachen Algorithmus definiert und stellt eine der kompliziertesten geometrischen Figuren der Mathematik dar. Der Algorithmus wurde in den 70er Jahren vom Mathematiker Benoit Mandelbrot entwickelt.

Fraktale

Ein *Fraktal* ist eine geometrische Figur, die durch einen mathematischen Algorithmus definiert ist, der dazu führt, dass die Figur bei Vergrösserungen nicht einfacher wird, sondern immer neue Verzweigungen aufzeigt (wie Küstenlinien, Wolken, Gebirge usw.)

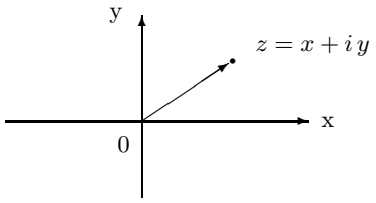


Fraktale Geometrie

Zur Beschreibung von fraktalen Formen hat Mandelbrot den Begriff der *fraktalen Geometrie* eingeführt, zur Unterscheidung von der klassischen Geometrie (Geraden, Kreise usw.).

Punkte und komplexe Zahlen

Zur Formulierung des Algorithmus von Mandelbrot ist es vorteilhaft, Punkte der Ebene als komplexe Zahlen aufzufassen. Einem Punkt $P(x, y)$ entspricht dabei die komplexe Zahl $z = x + i\,y$:



Dabei ist i, die imaginäre Einheit, d.h. der Punkt $(0, 1)$ auf der y-Achse. Die komplexe Schreibweise ermöglicht eine elegante Formulierung des Mandelbrot-Algorithmus unter Verwendung der Multiplikation für komplexe Zahlen. Alle Berechnungen mit dem Computer erfolgen jedoch mit den reellen Koordinaten x und y, sodass nur minimale (oder keine) Kenntnisse der komplexen Zahlen benötigt werden.

Abstand vom Nullpunkt

Den Abstand eines Punktes $z = x + i\,y$ vom Nullpunkt O nennt man den *Betrag* $|\,z\,|$ von z, also nach Pythagoras:

$$|\,z\,| = \sqrt{x^2 + y^2}$$

Orbits

Orbits sind die Gundlage für den Mandelbrot-Algorithmus. Ein Orbit ist eine unendliche Punktfolge in der Ebene:

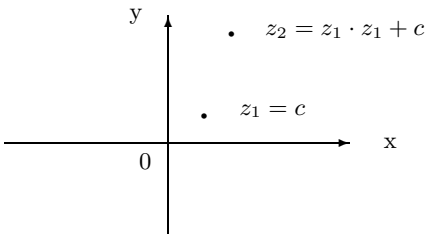
Sei c ein bliebiger Punkt der Ebene. Der *Orbit* mit Startpunkt c ist die Punktfolge $z_1, z_2, z_3 \ldots$ definiert durch

$$z_1 = c, \quad z_2 = z_1 \cdot z_1 + c, \quad z_3 = z_2 \cdot z_2 + c \quad \ldots$$

Das Bildungsgesetz der Folge ist also die einfache Formel:

$$\boxed{z' \;=\; z \cdot z + c}$$

Dabei steht z für den momentanen Punkt der Folge und z' für den nächsten.



Das Bildungsgesetz kann auch direkt mit den xy-Koordinaten der Punkte geschrieben werden:

$$\begin{aligned} x' &= x^2 - y^2 + c_1 \\ y' &= 2 \cdot x \cdot y + c_2 \end{aligned}$$

Diese Gleichungen folgen sofort aus der Definition der komplexen Multiplikation. Leser ohne Kenntnisse der komplexen Zahlen können diese Gleichungen als Definition des nächsten Punktes betrachten.

Beschränkte Orbits

Ein Orbit heisst beschränkt, wenn es eine Konstante r gibt, sodass

$$|z_i| \leq r \quad \text{für alle} \quad i = 1, 2, \ldots$$

d.h. alle (unendlich vielen) Punkte des Orbits liegen im Kreis $K_r(0)$ mit Radius r und Mittelpunkt O.

Definition der Mandelbrot-Menge

Die Mandelbrot-Menge M ist die Menge aller Punkte c der Ebene, für welche der zugehörige Orbit beschränkt ist.

Der Nullpunkt O gehört sicher zu M, da alle Punkte des zugehörigen Orbits gleich O sind. Damit ist M nicht leer.

Die Tatsache, dass diese einfache Definition eine der kompliziertesten Figuren ergibt, war für Mandelbrot eine grosse Überraschung.

Der Grund für die Komplexität der Mandelbrotmenge ist das *chaotische Verhalten* der Orbits: Die Eigenschaften eines Orbits zu einem

Startpunkt c sagen i.a. nichts aus über die Orbits in der Umgebung des Punktes c.

Für die Untersuchung von Orbits ist nach dem folgenden Satz der Kreis $K_2(0)$ mit Radius 2 um den Nullpunkt wichtig:

Satz

Ein Punkt c liegt genau dann in M, wenn alle (unendlich vielen) Punkte des Orbits im Kreis $K_2(0)$ liegen.

Sobald also *ein* Punkt eines Orbits den Kreis $K_2(0)$ verlässt, gehört der Startpunkt des Orbits nicht zu M. Daraus folgt insbesondere, dass M selber in $K_2(0)$ liegt.

Der Beweis des Satzes erfolgt mit elementaren algebraischen Rechnungen. Für mathematisch interessierte Leser ist er unten (Seite 107) aufgeführt.

Die Escape-Funktion

Die Grundlage aller Mandelbrot-Bilder ist die folgende Escape-Funktion:

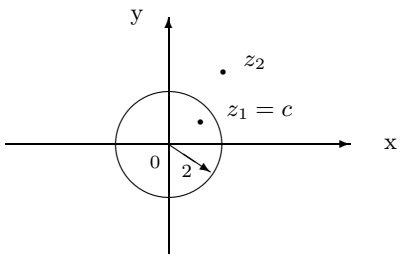
```
int esc(double c1, double c2, int n_max)
```

Die Funktion berechnet zu einem beliebigen Punkt (c_1, c_2) der Ebene eine positive ganze Zahl, die sogenannte *Escape-Number* des Punktes, nach dem folgenden Algorithmus:

Berechne die Punkte

$$z_1,\ z_2,\ ,z_3,\ ...$$

des Orbits mit Startpunkt c, bis der berechnete Punkt z_i einen Abstand grösser als 2 vom Nullpunkt hat, oder der Index i den Wert n_max überschritten hat. Das so bestimmte i ist die sogenannte *Escape-Number* des Punktes c.



Die Escape-Number ist also eine ganze Zahl im Bereich

$$1 \quad .. \quad n\_max + 1 \quad ,$$

die angibt, wie rasch der Orbit den Kreis $K_2(0)$ verlässt. Die Escape-Number $n\_max + 1$ bedeutet, dass alle berechneten Punkte des Orbits im Kreis $K_2(0)$ liegen. Ein solcher Punkt c ist folglich ein Kandidat für die Mandelbrot-Menge. Da wir nur endlich viele Punkte des Orbits berechnet haben, liegt der Punkt jedoch nicht mit Sicherheit in der Mandelbrot-Menge.

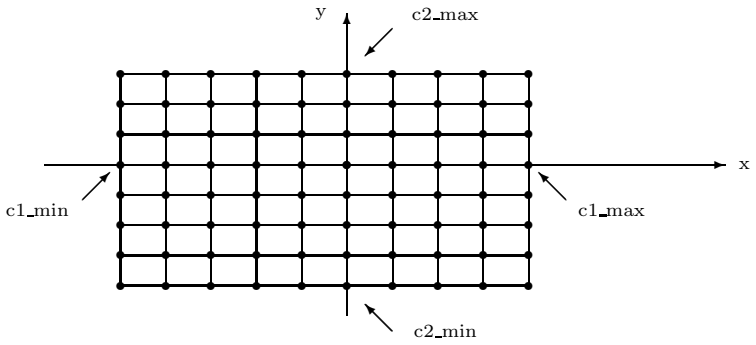
Die Escape-Funktion wird bei der Bilderzeugung für jeden Bildpunkt aufgerufen. Es ist daher darauf zu achten, dass sie keine unnötigen Rechnungen macht. Insbesondere wird die Wurzel-Funktion nicht benötigt, da beim Vergleich des Abstandes eines Punktes mit 2 das Quadrat des Abstandes mit 4 verglichen werden kann.

Der Bild-Algorithmus

Die berühmten Farbbilder von Mandelbrot entstehen nach dem folgenden Algorithmus:

1. Wähle einen rechteckförmigen Ausschnitt in der Ebene, der die Mandelbrot-Menge, oder einen Ausschnitt von ihr enthält.

   Das Rechteck wird durch seine Grenzen $c1\_min$, $c1\_max$ auf der x-Achse, bzw. $c2\_min$, $c2\_max$ auf der y-Achse vorgegeben und mit Gitterlinien überzogen:



cols	Anzahl vertikale Gitterlinien
lines	Anzahl horizontale Gitterlinien

Abstände der Gitterlinien:

$$\begin{aligned} dx &= (c1\_max - c1\_min) \;/\; (cols - 1) \\ dy &= (c2\_max - c2\_min) \;/\; (lines - 1) \end{aligned}$$

Jedem Gitterpunkt in Zeile i und Spalte j entspricht in natürlicher Weise ein Bildpunkt (Pixel) auf dem Bildschirm, mit $line = i$ und $col = j$.

2. Wähle eine Konstante n_max für die maximale Anzahl Iterationen in der Escape-Funktion.

3. Wähle eine Farbzuordnungs-Tabelle, die jeder möglichen Escape-Number

   $$1 \quad .. \quad n\_max + 1$$

   eine Farbe zuordnet. Dabei wird der höchsten Escape-Number die Farbe Schwarz zugeordnet, die Farben für die übrigen Escape--Numbers sind willkürlich.

   Dies ergibt für die Punkte der Mandelbrot-Menge die Farbe Schwarz, weil diese die Escape-Number $n\_max + 1$ haben.

4. Zeichnen der Punkte

   Das Gitter wird zeilenweise durchlaufen. Dabei werden für jeden Gitterpunkt (i, j) die folgenden Aktionen durchgeführt:

   - Berechne die Koordinaten $(c1, c2)$ des zugehörigen Punktes in der Ebene:

     $$\begin{aligned} c1 &= c1\_min \;+\; j \cdot dx \\ c2 &= c2\_max \;+\; i \cdot dy \end{aligned}$$

   - Berechne die Escape-Number $esc(c1, c2, n\_max)$ des Punktes.
   - Zeichne das zu dem Gitterpunkt gehörige Pixel (j, i) in der Farbe, die der Escape-Number zugeordnet ist.

Bild-Parameter:

(a) Konstanten: $cols = 480, \quad lines = 361, \quad n\_max = 40$

(b) Gitter-Daten:

Mittelpunkt-Koordinaten: $x_m = -0.7,\ y_m = 0$

Breite und Höhe: $breite = 3.5, \quad hoehe = breite \cdot lines/cols$

Dies ergibt die folgenden Grenzen:

$c1\_min = x_m - 0.5 \cdot breite, \quad c1\_max = x_m + 0.5 \cdot breite,$
$c2\_min = y_m - 0.5 \cdot hoehe, \quad c2\_max = y_m + 0.5 \cdot hoehe$

(c) Beispiel einer Farbzuordnungs-Tabelle

Escape-Number	Farbe	R	G	B
1 .. 5	gray	128	128	128
6	white	192	192	192
7	light_magenta	255	0	255
8	magenta	128	0	128
9	red	128	0	0
10	light_red	255	0	0
11 .. 12	yellow	255	255	0
13	light_green	0	255	0
14	green	0	128	0
15	light_cyan	0	255	255
16 .. 19	cyan	0	128	128
20 .. 23	light_blue	0	0	255
24 .. 29	blue	0	0	128
30 .. 40	brown	128	128	0
41	black	0	0	0

Die benötigten Farben werden am besten in einem Array von Colors definiert. Die Farbzuordnung ist nach dem Prinzip des Regenbogens aufgebaut:

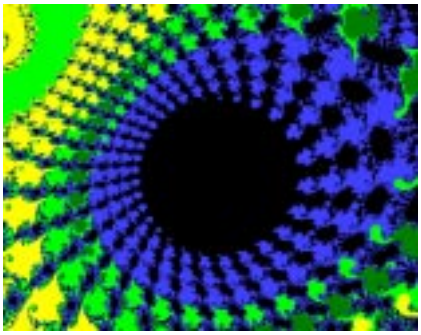
rot → gelb → grün → blau.

Die Punkte mit der höchsten Escape-Number $n\_max + 1$ erscheinen gemäss Farbtabelle schwarz. Dies sind sicher die Punkte der Mandelbrot-Menge, da diese gemäss Definition Escape-Number ∞ haben, aber (je nach Grösse von n_max) eventuell auch andere.

Das schwarze Gebiet stellt folglich nur ein Näherungsbild der Mandelbrot-Menge dar, welches umso genauer ist, je grösser n_max gewählt wird.

Ausschnitt-Vergrösserung

Die folgenden Parameter ergeben eine Ausschnitt-Vergrösserung in der Form eines Auges.



- Konstanten: $cols = 500$, $lines = 400$, $n\_max = 300$
- Gitter-Daten:

  Wie oben, aber mit Mittelpunkt-Koordinaten $x_m = -0.74691$, $y_m = 0.10725$ und $breite = 0.002$
- Farbzuordnung

Escape-Number	Farbe	R	G	B
1 .. 20	light_blue	0	0	255
21 .. 40	green	0	128	0
41 .. 60	light_green	0	255	0
61 .. 80	yellow	255	255	0
81 .. 100	yellow	255	255	0
101 .. 120	yellow	255	255	0
121 .. 140	yellow	255	255	0
141 .. 160	light_green	0	255	0
161 .. 180	green	0	128	0
181 .. 200	light_green	0	255	0
201 .. 220	light_blue	0	0	255
221 .. 240	light_blue	0	0	255
241 .. 260	blue	0	0	128
261 .. 280	blue	0	0	128
281 .. 300	blue	0	0	128
301	black	0	0	0

Wo befindet sich dieses Auge ?

Vergrössern Sie die angegebene Breite des Gitters (z.B. auf 0.02 oder 0.2), damit Sie sehen an welcher Stelle der Mandelbrot-Menge dieser Ausschnitt liegt.

Nachtrag: Beweis des Satzes

Wir führen hier noch den Beweis des folgenden Satzes, der auf Seite 102 eingeführt wurde.

Satz

Ein Punkt c liegt genau dann in M, wenn alle (unendlich vielen) Punkte des Orbits im Kreis $K_2(0)$ liegen.

Beweis:

Es genügt zu zeigen, dass jeder Orbit, der den Kreis $K_2(0)$ verlässt, gegen Unendlich konvergiert. Der Beweis benötigt eine Fallunterscheidung aufgrund der Lage des Startpunktes c :

a) c ausserhalb des Kreises $K_2(0)$, d.h. es gilt $|c| > 2$

In diesem Fall liegt bereits der Startpunkt ausserhalb $K_2(0)$, es ist also zu zeigen, dass der Orbit gegen Unendlich konvergiert. Wir setzen $q := |c| - 1$ und zeigen mittels vollständiger Induktion, dass

$$|z_k| \;\geq\; q^{k-1} \cdot |c| \qquad \text{für} \quad k = 1, 2, ...$$

Wegen $q > 1$ folgt daraus sofort, dass der Orbit gegen Unendlich konvergiert.

Für $k = 1$ ist nichts zu zeigen, da $z_1 = c$ ist.
Induktions-Schritt $k \to k + 1$:

$$\begin{aligned} |z_{k+1}| &= |z_k^2 + c| \\ &\geq |z_k|^2 - |c| \;=\; |z_k| \cdot \left(|z_k| - \frac{|c|}{|z_k|} \right) \end{aligned}$$

Dabei wurde die allgemeine Ungleichung $|a + b| \geq |a| - |b|$ verwendet, die aus der Dreiecksungleichung folgt. Mit der Induktionsvoraussetzung folgt:

$$|z_{k+1}| \;\geq\; q^{k-1} \cdot |c| \cdot (\, |c| - 1 \,) \;=\; q^k \cdot |c|$$

b) c liegt im Kreis $K_2(0)$, d.h. es gilt $|c| \leq 2$

Wir zeigen, dass der Orbit gegen Unendlich konvergiert, wenn er den Kreis $K_2(0)$ verlässt. Sei z_n ein Punkt des Orbits mit $|z_n| > 2$. Wir setzen

$$q := |z_n| - 1$$

und zeigen mittels vollständiger Induktion, dass

$$|z_{n+k}| \;\geq\; q^k \cdot |z_n| \qquad \text{für} \quad k = 0, 1, 2, ...$$

Wegen $q > 1$ folgt daraus sofort, dass der Orbit gegen Unendlich konvergiert.

Für $k = 0$ ist nichts zu zeigen.

Induktions-Schritt $k \to k + 1$:

$$\begin{aligned} |z_{n+k+1}| &= |z_{n+k}^2 + c| \\ &\geq |z_{n+k}|^2 - |c| \;=\; |z_{n+k}| \cdot \left(|z_{n+k}| - \frac{|c|}{|z_{n+k}|} \right) \end{aligned}$$

Mit der Induktionsvoraussetzung folgt:

$$|z_{n+k+1}| \;\geq\; q^k\,|z_n| \cdot \left(q^k\,|z_n| \;-\; \frac{|c|}{q^k\,|z_n|} \right)$$

Wegen $q > 2$, $|c| <= 2$ und $|z_n| > 2$ erhält man

$$|z_{n+k+1}| \;\geq\; q^k \cdot |z_n| \cdot (\,|z_n| - 1\,) \;=\; q^{k+1} \cdot |z_n| \qquad \text{q.e.d.}$$

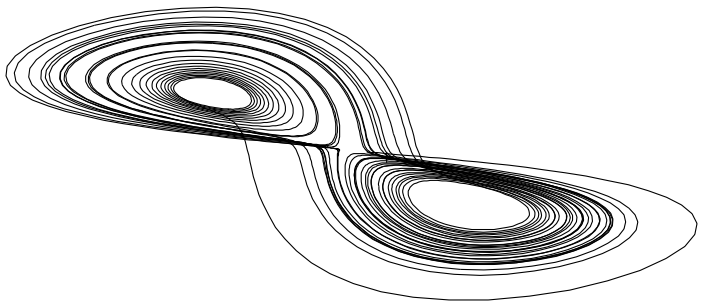
Anwendung 2

Der Lorenz-Attraktor

Der Lorenz-Attraktor wurde in den 60er Jahren von E. Lorenz im Zusammenhang mit langfristigen Wettervorhersagen entdeckt. Er stellt ein Gebiet im dreidimensionalen Raum dar, welches Teilchen anzieht, die sich nach einem bestimmten Gesetz bewegen.

Unabhängig vom Startpunkt bewegen sich diese Teilchen nach kurzer Zeit auf dem Attraktor.

Typische Bahn auf dem Attraktor:



Der Lorenz-Attraktor war der Ausgangspunkt der Chaos-Theorie.

Das Dynamische System von Lorenz

Grundlegend für die Bahnen der Teilchen sind die folgenden drei Funktionen von x, y und z :

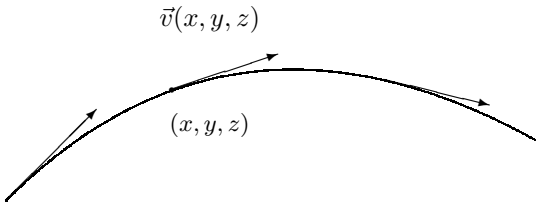
$$\begin{aligned} v_1(x,y,z) &= 10\,y - 10\,x \\ v_2(x,y,z) &= 28\,x - y - x \cdot z \\ v_3(x,y,z) &= x \cdot y - \tfrac{8}{3} \cdot z \end{aligned}$$

Zu jedem Punkt (x, y, z) kann man die drei Funktionswerte zu einem Vektor $\vec{v}(x, y, z)$ zusammenfassen. So erhält man zu jedem Punkt (x, y, z) einen Vektor $\vec{v}(x, y, z)$. Eine solche Zuordnung nennt man ein

Vektorfeld *Vektorfeld* oder ein *Dynamisches System*.

Eine anschauliches Beispiel eines Vektorfeldes ist das Geschwindigkeitsfeld einer Flüssigkeitsströmung: jedem Punkt der Strömung wird der Geschwindigkeitsvektor der fliessenden Teilchen in diesem Punkt zugeordnet.

Die Vektoren des Geschwindigkeitsfeldes sind tangential an die Bahnkurven der fliessenden Teilchen (Stromlinien).



Umgekehrt gibt es zu einem beliebig vorgegebenen Vektorfeld, z.B. dem Lorenz-Vektorfeld, eine eindeutig bestimmte Strömung, deren Geschwindigkeitsfeld gleich dem gegebenen Vektorfeld ist. Die Berechnung der Bahn eines Teilchens der Strömung ist einfach:

Man startet in einem beliebigen Punkt und schreitet in kleinen Schritten immer in der durch das Vektorfeld vorgegebenen Richtung voran. So entsteht ein Streckenzug, der für kleine Schritte eine gute Näherung der Bahnkurve des Teilchens darstellt.

Algorithmus für die Berechnung einer Bahn

Der folgende Algorithmus berechnet zu einem beliebigen Startpunkt die Bahnkurve im Raum und zeichnet sie mittels Normalprojektion auf die xy-Ebene.

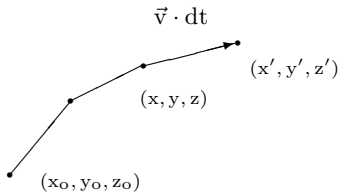
1. Wähle einen beliebigen Startpunkt (x_o, y_o, z_o), z.B. $(14, 20, 22)$ und einen Zeitschritt dt, z.B. $dt = 0.001$.

2. Setze $x = x_o$, $y = y_o$, $z = z_o$ (laufender Punkt)

3. Berechne den Vektor $\vec{v}$ des Vektorfeldes im Punkt (x, y, z) mit den Gleichungen von Lorenz.

4. Berechne den nächsten Punkt der Bahn durch Addition des Vektors $\vec{v} \cdot dt$ zum momentanen Punkt:

$$x' = x + v_1 \cdot dt$$
$$y' = y + v_2 \cdot dt$$
$$z' = z + v_3 \cdot dt$$



5. Zeichne die zurückgelegte Strecke von (x, y, z) nach (x', y', z') auf dem Bildschirm mittels Normalprojektion auf die $xy-$Ebene (z-Koordinaten weglassen).

6. Setze $x = x'$, $y = y'$, z= z' und gehe zu Schritt 3.

Anzahl Schritte: 40000

Koordinatenbereich für graphische Darstellung:

$$xMin = -40, \quad xMax = 40, \quad yMin = -30, \quad yMax = 30$$

In der am Anfang dargestellten Figur wurde die Bahnkurve nach der Projektion in die $xy-$Ebene um den Winkel $-0.4\,\pi$ (Bogenmass) im Uhrzeigersinn um den Nullpunkt gedreht.

Bemerkung:

Das verwendete Verfahren zur numerischen Berechnung von Bahnkurven stammt von Euler. Es gibt noch ein verbessertes Verfahren von Runge-Kutta, welches bei gleicher Schrittweite dt bessere Genauigkeit für die Bahnkurve liefert.

Der Attraktor

Die Bahnkurven des Dynamischen Systems von Lorenz haben interessante Eigenschaften:

- Für jeden Startpunkt (ausser (0,0,0)) haben die Bahnkurven nach einer vom Startpunkt abhängigen Anfangsphase immer die gleiche typische Form.

  Man nennt das Gebiet, in welchem die Bahnkurven verlaufen, den *Attraktor*.

Chaos

- Das Lorenz-System besitzt weiter die Eigenschaft des *deterministischen Chaos:* Zwei Bahnkurven, die sich im Anfangspunkt nur minimal unterscheiden, verlaufen zunächst praktisch gleich, plötzlich verzweigen sie jedoch und laufen total unterschiedlich auf dem Attraktor.

  Diesen Sachverhalt nennt man deterministisches Chaos: Zu einem gegebenen Anfangspunkt ist der Verlauf der Bahnkurve nach der Theorie der Dynamischen Systeme bis auf alle Zeiten exakt bestimmt (Determinismus), aber kleinste Abweichungen des Anfangspunktes wirken sich langfristig signifikant aus.

  Diese extreme Empfindlichkeit in Bezug auf die Anfangsbedingungen bedeutete für Lorenz die Unmöglichkeit einer langfristigen Wettervorhersage: minime Messungenauigkeiten in den Anfangsbedingungen verstärken sich langfristig zu massiven Fehlern.

  Lorenz hat dies mit dem *Schmetterlings-Effekt* ausgedrückt:

  Der Flügelschlag eines Schmetterlings kann einen Tornado auslösen oder verhindern.

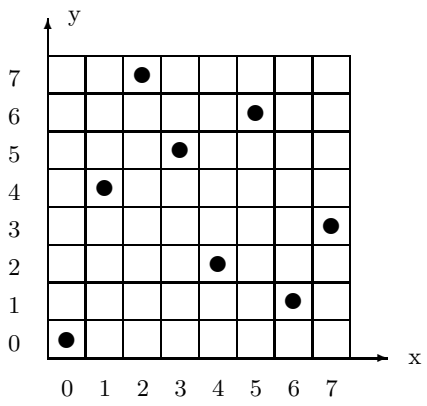
Anwendung 3

Das 8-Damen Problem

Das 8-Damen Problem besteht darin, auf einem Schachbrett mit 8×8 Feldern 8 Damen so aufzustellen, dass keine Dame eine andere bedroht.

Eine Dame bedroht die Zeile und die Spalte in der sie steht, sowie die beiden Diagonalen, die sich auf ihrem Feld kreuzen.

Beispiel einer Lösung:



Da eine Dame die ganze Spalte, in der sie sich befindet, bedroht, folgt sofort, dass bei der Konstruktion einer Lösung in jeder Spalte genau eine Dame zu setzen ist.

Algorithmus zur Konstruktion der ersten Lösung

Wir setzen die Damen spaltenweise, beginnend mit der ersten Spalte, wobei wir y[0] = 0 setzen. Beim Setzen einer Dame wird diese von unten nach oben verschoben, bis sie auf einem unbedrohten Feld steht.

Gibt es in der betreffenden Spalte kein unbedrohtes Feld, so befindet man sich in einer Sackgasse und muss für die Dame in der vorangehenden Spalte eine neue Position finden.

Bestimmung der weiteren Lösungen

Wenn man eine Lösung gefunden hat, wird sie ausgegeben, dann wird die letzte Dame vom Brett genommen, und die vorletzte beginnt vorzurücken. Etwas Überlegung zeigt, dass wir so alle Lösungen durchlaufen. (Anzahl Lösungen: 92)

Entwickeln Sie das Programm in zwei Schritten: (a) Ausgabe der ersten Lösung , (b) Ausgabe aller Lösungen. Die Lösungen werden in der Form von 8 Zahlen ausgegeben (ohne Graphik).

Das Programm kann iterativ oder rekursiv realisiert werden. Führen Sie einen Array

```
int[ ] y = new int[8];
```

ein, in dem die momentanen y-Positionen der Damen gespeichert werden. Zwei Damen i und j bedrohen sich gegenseitig, wenn mindestens eine der folgenden Bedingungen zutrifft:

(a) $y[i] = y[j]$ (gleiche Zeile) oder

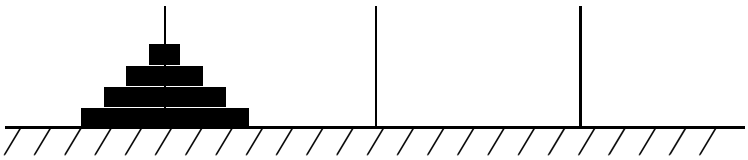
(b) $y[j] - y[i] = j - i$ (Diagonale mit positiver Steigung) oder

(c) $y[j] - y[i] = i - j$ (Diagonale mit negativer Steigung)

Anwendung 4

Die Türme von Hanoi

Bei dem Spiel der Türme von Hanoi sind 3 Stäbe gegeben, auf denen n Scheiben zu Türmen aufgestapelt werden können. Die Scheiben haben verschiedene Grössen und befinden sich am Anfang der Grösse nach geordnet auf dem ersten Stab:



- Die Aufgabe des Spiels besteht darin, den Turm schrittweise auf einen anderen Stab zu verschieben.
- Bei jedem Schritt wird die oberste Scheibe eines Stabes auf einen anderen Stab verlegt.
- Dabei darf nie eine grössere Scheibe auf einer kleineren liegen.

Der Schlüssel für ein Lösungsverfahren des Problems ist die folgende rekursive Betrachtungsweise, bei der das Problem für n Scheiben zurückgeführt wird auf $n-1$ Scheiben:

Die Verschiebung des Turmes mit n Scheiben von Stab 1 nach 2 erfolgt in 3 Schritten:

1. Verschiebe den Turm, bestehend aus den obersten $n-1$ Scheiben von 1 nach 3. Nach der Verschiebung ist Stab 2 leer.
2. Verschiebe die unterste Scheibe von 1 nach 2.
3. Verschiebe die $n-1$ Scheiben von 3 nach 2.

Man beachte, dass in den Schritten 1 und 3 die nicht behandelte Scheibe nicht stört, da sie die grösste ist.
Realisieren Sie das Programm als Applikation ohne Graphik, so dass es Verschiebebefehle ausgibt:

```
Verschiebung von 1 nach 2
Verschiebung von 1 nach 3
...
```

Definieren Sie dazu eine rekursive Methode:

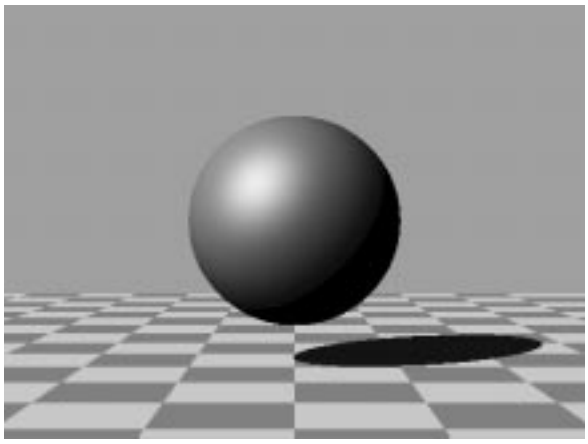
```
static void move(int nScheiben, int von, int nach)
```

Anwendung 5

Das Raytracing-Verfahren

5.1 Grundprinzip

Mit dem Raytracing-Verfahren (Strahlen-Verfolgung) können realistische Bilder von beleuchteten Gegenständen erzeugt werden. Als Beispiel betrachten wir eine Kugel, die von einer Lichtquelle beleuchtet wird.

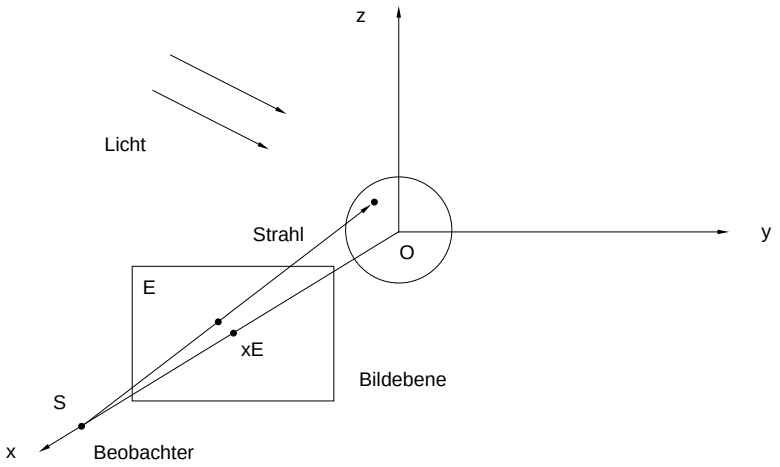


Bei der wirklichen Betrachtung eines Gegenstandes empfängt der Beobachter (oder die Kamera) Lichtstrahlen von dem Objekt. Das Raytracing-Verfahren funktioniert umgekehrt: Vom Beobachter aus werden virtuelle, rückwärts laufende Lichtstrahlen gegen das Objekt ausgesandt. Daher kommt der Name *Strahlen-Verfolgung*.

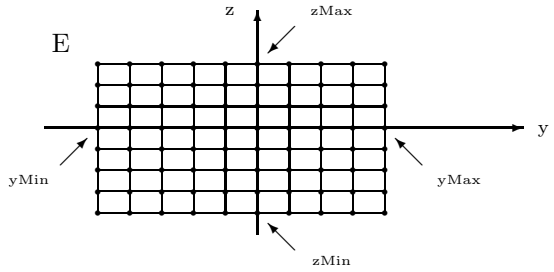
Der Raytracing-Algorithmus

Ein Raytracing-Bild wird in den folgenden drei Schritten zeilenweise Punkt für Punkt berechnet und gezeichnet:

1. Wahl einer Beobachter-Position S und einer Bildebene E parallel zur yz-Ebene, vor dem Beobachter.



2. Festlegung eines Gitters in E mit der Auflösung des Bildschirm-Windows, sodass jedem Gitterpunkt ein Pixel auf dem Bildschirm entspricht.



cols Anzahl vertikale Gitterlinien
lines Anzahl horizontale Gitterlinien

3. Die Gitterpunkte werden (zeilenweise) durchlaufen. Für jeden Gitterpunkt P wird von S aus ein Strahl durch P ausgesandt.

   - Wenn der Strahl die Kugel durchstösst, wird im ersten Durchstosspunkt die Helligkeit berechnet, und das zum Gitterpunkt gehörige Pixel mit der entsprechenden Graustufe gezeichnet.
   - Wenn ein Strahl die Kugel nicht trifft, heisst dies, dass er in den Raum entschwindet, folglich wird das Pixel mit einer Hintergrund-Farbe gezeichnet.

5.2 Die Klasse Vek3d

Wir führen eine Klasse für Vektoren im Raum ein:

```
public class Vek3d
{
  public double x, y, z;                    // Komponenten
}
```

Die Klasse kann auch für Punkte verwendet werden, indem man Punkte mit ihren Ortsvektoren identifiziert.

Methoden:

```
public Vek3d()                              // Konstruktor
{ }

public Vek3d (double x, double y, double z) // Konstruktor
{ this.x = x;  this.y = y;  this.z = z;
}

public void setVal(double x, double y, double z)  // Komponenten setzen
{ this.x = x;  this.y = y;  this.z = z;
}

public void setVal(Vek3d from)              // Komponenten kopieren
{ x = from.x;  y = from.y;  z = from.z;
}

public double norm()                        // Laenge des Vektors
{ return Math.sqrt(x*x+y*y+z*z);
}
```

Vektor-Operationen

Die Grundoperationen für Vektoren werden als statische Methoden realisiert. Es sind Prozeduren, keine Funktionen, weil dadurch nicht bei jedem Aufruf ein neues Objekt für das Resultat erzeugt werden muss.

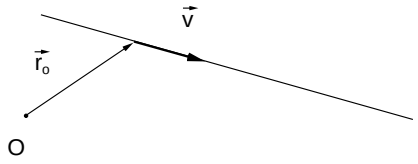
```
public static void add(Vek3d a, Vek3d b, Vek3d c)      // c = a + b
{   c.setVal(a.x + b.x, a.y + b.y, a.z + b.z);
}

public static void sub(Vek3d a, Vek3d b, Vek3d c)      // c = a − b
{   c.setVal(a.x − b.x, a.y − b.y, a.z − b.z);
}

public static void mult(double t, Vek3d a, Vek3d c)    // c = t * a
{   c.setVal(t * a.x, t * a.y, t * a.z);
}

public static double skalarProd(Vek3d a, Vek3d b)      // Skalar−Produkt
{  return  a.x * b.x + a.y * b.y + a.z * b.z;
}
```

5.3 Durchstosspunkte

Das A und O des Raytracing-Verfahrens sind Durchstosspunkte von Strahlen (Geraden) durch Objekte. Wir benötigen Durchtosspunkte von Geraden durch Kugeln.

Geraden

Eine Gerade wird beschrieben durch den Ortsvektor $\vec{r}_o$ eines beliebigen Punktes der Geraden und einen Richtungsvektor $\vec{v}$.



Der laufende Punkt der Geraden wird beschrieben durch

$$\vec{r} = \vec{r}_o + t \cdot \vec{v}$$

wobei t eine beliebige reelle Zahl ist. Die Gleichung kann auch in Komponenten geschrieben werden:

$$\begin{aligned} x &= x_o + t \cdot v_x \\ y &= y_o + t \cdot v_y \\ z &= z_o + t \cdot v_z \end{aligned}$$

wobei:

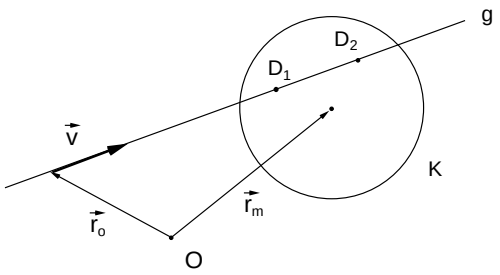
$$\vec{r} = (x, y, z), \quad \vec{r}_o = (x_o, y_o, z_o) \quad \text{und} \quad \vec{v} = (v_x, v_y, v_z)$$

Java-Klasse für Geraden:

```
public class Gerade
{  public Vek3d r0 = new Vek3d();              // Punkt auf der Geraden
   public Vek3d v = new Vek3d();               // Richtungsvektor

   public Gerade()                             // Konstruktor
   {   }

   public Gerade(Vek3d r0, Vek3d v)            // Konstruktor
   {  this.r0.setVal(r0);
      this.v.setVal(v);
   }
}
```

Durchstosspunkte mit einer Kugel

Wir berechnen jetzt die Durchstosspunkte der Geraden g mit einer Kugel K, welche gegeben sei durch den Mittelpunkt $\vec{r}_m = (x_m, y_m, z_m)$ und den Radius R.



Die gesuchten Durchstosspunkte sind diejenigen Punkte (x, y, z) auf der Geraden g, die den Abstand R vom Mittelpunkt der Kugel haben, d.h. mit der Abstands-Formel für Punkte:

$$(x - x_m)^2 + (y - y_m)^2 + (z - z_m)^2 = R^2 \ .$$

Mit den Formeln für die Koordinaten des laufenden Punktes von g folgt:

$$(x_o + t\,v_x - x_m)^2 + (y_o + t\,v_y - y_m)^2 + (z_o + t\,v_z - z_m)^2 = R^2 \ .$$

Mit der Abkürzung $\vec{u} = \vec{r}_o - \vec{r}_m$, d.h.

$$u_x = x_o - x_m, \quad u_y = y_o - y_m, \quad u_z = z_o - z_m$$

erhält man

$$(u_x + t\,v_x)^2 + (u_y + t\,v_y)^2 + (u_z + t\,v_z)^2 \ = \ R^2 \ .$$

Dies ist eine quadratische Gleichung

$$a\,t^2 + b\,t + c = 0$$

mit

$$\begin{aligned} a &= v_x^2 + v_y^2 + v_z^2 \ = \ \vec{v} \cdot \vec{v} \\ b &= 2\,(u_x v_x + u_y v_y + u_z v_z) \ = \ 2\,\vec{u} \cdot \vec{v} \\ c &= u_x^2 + u_y^2 + u_z^2 - R^2 \ = \ \vec{u} \cdot \vec{u} - R^2 \end{aligned}$$

Resultat:

Die Ortsvektoren der gesuchten Durchstosspunkte sind gegeben durch:

$$\vec{d}_{1,2} = \vec{r}_o + t_{1,2} \cdot \vec{v}$$

wobei t_1 und t_2 die Lösungen der quadratischen Gleichung sind. Wenn diese keine (reelle) Lösungen hat, durchstösst die Gerade die Kugel nicht.

△ **Uebung:**

Führen Sie eine Klasse 'Kugel' ein, analog zur Klasse 'Gerade', und berechnen Sie die Durchstosspunkte der Geraden g:

$$\vec{r}_o = (10, 4, -2), \quad \vec{v} = (1, 2, 1)$$

mit der Kugel:

$$\vec{r}_m = (22, 26, 10), \quad \text{Radius } 4$$

Lösungen: $(22.897, 29.794, 10.897)$, $(19.770, 23.540, 7.770)$

5.4 Raytracing Bild des Mondes

Das Raytracing-Verfahren eignet sich zur Darstellung des Mondes als Kugel, die von der Sonne beleuchtet wird. Je nach Position der Sonne entstehen dabei die bekannten Formen von der Mondsichel bis zum Vollmond.

Berechnung der Helligkeit in einem Punkt

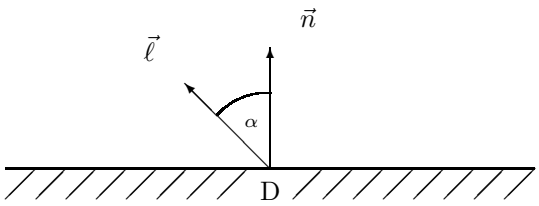
Infolge der rauhen Oberfläche des Mondes wird das einfallende Licht diffus (allseitig) reflektiert, wie bei einer rauhen Wand (im Gegensatz zu einer glatten, spiegelnden Fläche).

Für diffuse Reflexion kann die Helligkeit I in einem Punkt D auf der Oberfläche des Objektes mit dem *Gesetz von Lambert* berechnet werden:

Gesetz von Lambert

$$I = c \cdot \cos(\alpha)$$

Dabei ist c eine Konstante (maximale Helligkeit) und α ist der Einfallswinkel des Lichtes zur Normalen:



$\vec{\ell}$ Richtung zur Lichtquelle
$\vec{n}$ Flächennormale (Senkrechte zur Kugeloberfläche)

Die Formel ist einleuchtend: sie ergibt maximale Helligkeit für senkrechten Einfall des Lichtes ($\alpha = 0$). Mit wachsendem α nimmt die Helligkeit ab, bis auf 0 bei parallelem Einfall ($\alpha = 90^o$).

Für Winkel $\alpha > 90^o$ wird I negativ. Dies ist als 0 zu interpretieren, da in diesem Fall der Punkt D nicht beleuchtet wird.

Für die Berechnung von $\cos(\alpha)$ wird das Skalarprodukt von Vektoren verwendet:

$$\cos(\alpha) = \frac{\vec{n} \cdot \vec{\ell}}{\| \vec{n} \| \cdot \| \vec{\ell} \|}$$

Dabei bezeichnet $\| \vec{a} \|$ die Länge eines Vektors $\vec{a}$. Bei diffuser Reflexion spielt also die Blickrichtung des Beobachters für die Helligkeit keine Rolle, sondern nur die Einfallsrichtung des Lichtes in Bezug auf die Normale.

Dies ist nicht selbstverständlich, entspricht aber unserer Erfahrung (eine Wand erscheint aus allen Richtungen gleich hell).

Konfiguration

Hintergrund-Farbe: schwarz

$\vec{r}_m = (0, 0, 0)$	Kugel-Mittelpunkt
R = 1	Kugel-Radius
$\vec{s} = (10, 0, 0)$	Beobachter
xE = 8.0	x-Koordinate Bildebene
$\vec{\ell} = (-1.2, 4, -2.8)$	Richtung zur Lichtquelle (Sonne)

Parameter des Gitters in der Bildebene:

cols = 640	Anzahl vertikale Gitterlinien
lines = 480	Anzahl horizontale Gitterlinien
breite = 1.0	Breite des Gitters
hoehe = breite * lines / cols	Hoehe
dy = breite / (cols-1)	horizontaler Abstand der Gitterlinien
dz = hoehe / (lines-1)	vertikaler Abstand der Gitterlinien
yMin = - 0.5 * breite	y-Koord. linker Rand des Gitters
zMax = 0.5 * hoehe	z-Koord. oberer Rand

Bildberechnung

Die Gitterpunkte werden zeilenweise durchlaufen. Für jeden Gitterpunkt werden die folgenden Aktionen durchgeführt:

1. Berechnung der Koordinaten des Gitterpunktes $\vec{p}$:

   $$\begin{aligned} x &= xE \\ y &= yMin + i * dy \\ z &= zMax - j * dz \end{aligned}$$

   Dabei ist i der Spalten- und j der Zeilenindex des Gitterpunktes.

2. Festlegung des Strahls g vom Beobachter $\vec{s}$ aus durch den Gitterpunkt $\vec{p}$, d.h. $\vec{r}_o = \vec{s}$, Richtungsvektor $\vec{v} = \vec{p} - \vec{s}$.

3. Berechnung der Durchstosspunkte des Strahls g mit der Kugel.

4. Wenn keine Durchstosspunkte existieren, ist für den Gitterpunkt nichts mehr zu tun, das zugehörige Pixel bleibt schwarz.

5. Andernfalls wird die Helligkeit im näher liegenden Durchstosspunkt $\vec{d}$ mit dem Gesetz von Lambert berechnet. Dabei kann die Konstante $c = 1$ gewählt werden:

   $$helligkeit = \frac{\vec{n} \cdot \vec{\ell}}{\| \vec{n} \| \cdot \| \vec{\ell} \|}$$

   Die Berechnung eines Normalenvektors $\vec{n}$ im Durchstosspunkt $\vec{d}$ ist für die Kugel einfach:

   $$\vec{n} = \vec{d} - \vec{r}_m$$

Wenn das Skalarprodukt negativ ausfällt, ist der Punkt nicht beleuchtet, das zugehörige Pixel bleibt schwarz, sonst wird es mit der entsprechenden Graustufe gezeichnet, d.h. mit den RGB-Werten

$$r = g = b = helligkeit \cdot 255 \qquad \text{(ganzzahlig)}$$

Zum Zeichnen in der berechneten Graustufe muss ein Objekt der Klasse 'Color' erzeugt werden.

Bemerkung:

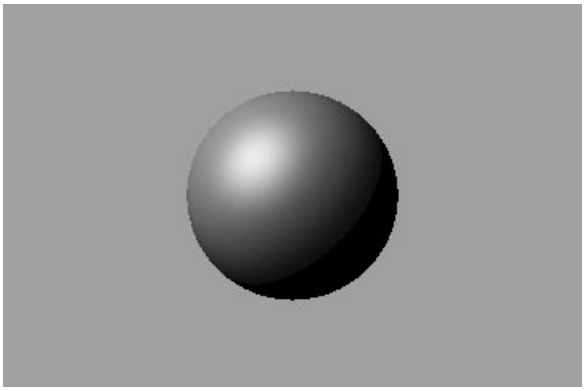
Die Video-Karte muss auf mindestens 65000 Farben (High Color) eingestellt sein, sonst erscheinen Stufen, da zuwenig verschiedene Graustufen dargestellt werden.

Blauer Hintergrund:

Eine schöne Variation ergibt sich, wenn der Hintergrund blau gezeichnet wird und für den Mond Farbstufen verwendet werden, die von weiss in blau (statt in schwarz) laufen. Dazu werden nur die Rot- und die Grün-Anteile reduziert.

5.5 Bild einer spiegelnden Kugel

Das Raytracing-Applet, welches den Mond darstellt, kann leicht modifiziert werden, sodass es eine glänzende Metallkugel darstellt:



Erforderliche Aenderungen:

- Konfiguration:

  Hintergrund-Farbe: blau

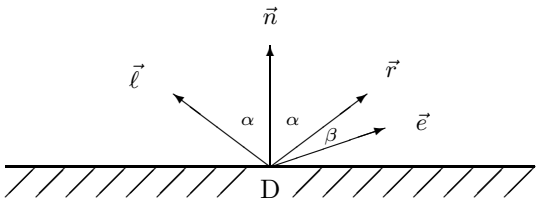
  Richtung zur Lichtquelle: $\vec{\ell} = (1, -1, 1)$

- Helligkeitsberechnung:

  Zusätzlich zur diffusen Reflexion des Lichtes wird eine spiegelnde Reflexion berücksichtigt, die im folgenden eingeführt wird.

Spiegelnde Oberflächen

Bei glattpolierten Oberflächen tritt neben der diffusen Reflexion des Lichtes die spiegelnde Reflexion auf. Diese ist richtungsspezifisch mit Maximum in Richtung des gespiegelten Strahls $\vec{r}$:



$\vec{n}$	Flächennormale
$\vec{\ell}$	Richtung zur Lichtquelle (toLight)
$\vec{r}$	gespiegelter Strahl (Einfallswinkel = Ausfallwinkel)
$\vec{e}$	Richtung zum Beobachter (toEye)

Der Einfallswinkel α des Lichtes ist massgebend für den diffusen Anteil der Reflexion. Der Winkel β gibt die Abweichung der Beobachtungsrichtung von der Richtung des gespiegelten Strahls an. Er bestimmt die Intensität der spiegelnden Reflexion in Beobachtungsrichtung (Maximum für $\beta = 0$).

Eine geeignete Formel für die totale Helligkeit I in einem Punkt D ist eine Summe der diffusen und der spiegelnden Reflexion, mit zwei Gewichtsfaktoren k_d und k_s. Die spiegelnde Reflexion wird als Funktion von $\cos(\beta)$ angesetzt:

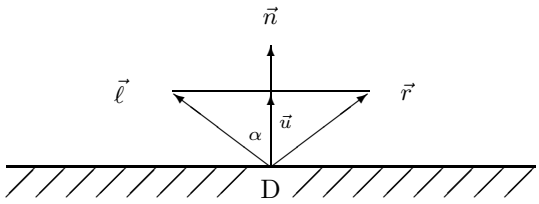
$$I = k_d \cdot \cos(\alpha) + k_s \cdot (\cos(\beta))^p \tag{5.1}$$

mit:

$k_d = 0.6$	Anteil diffuse Reflexion
$k_s = 0.4$	Anteil spiegelnde Reflexion
$p = 4$	Exponent für Abnahme der spiegelnden Reflexion

Der Exponent p bestimmt, wie stark die spiegelnde Reflexion richtungsspezifisch ist. Ein grosses p führt zu einem scharfen Spiegelfleck der Lichtquelle auf der Kugel.

Die Berechnung des gespiegelten Strahls ergibt sich aus der folgenden Figur:



$$\vec{u} \;=\; \cos(\alpha) \cdot \frac{\vec{n}}{\|n\|}$$

$$\vec{\ell} + \vec{r} \;=\; 2 \cdot \vec{u} \qquad \text{(Ergänzung zu Rhombus)}$$

also

$$\vec{r} \;=\; 2 \cdot \vec{u} - \vec{\ell}$$

Die cos-Werte werden wieder mit dem Skalarprodukt berechnet:

$$\cos(\alpha) = \frac{\vec{n} \cdot \vec{\ell}}{\| \vec{n} \| \cdot \| \vec{\ell} \|} \qquad \cos(\beta) = \frac{\vec{r} \cdot \vec{e}}{\| \vec{r} \| \cdot \| \vec{e} \|}$$

Der Vektor $\vec{e}$ ist der Vektor vom momentanen Punkt $\vec{d}$ der Fläche (Durchstosspunkt) zum Beobachter $\vec{s}$, d.h. $\vec{e} = \vec{s} - \vec{d}$.

Es empfiehlt sich, vor den obigen Rechnungen die Vektoren $\vec{\ell}$, $\vec{n}$ und $\vec{e}$ zu normieren. Dann fallen die Nenner in den Formeln weg (der Vektor $\vec{r}$ hat dieselbe Länge wie $\vec{\ell}$, ist also automatisch auch normiert).

Die Methode 'lightIntens'

Erstellen Sie eine Funktion 'lightIntens', welche die Helligkeit in einem Punkt D der Fläche aufgrund der Richtungen $\vec{\ell}$, $\vec{n}$ und $\vec{e}$ berechnet:

```
double lightIntens(Vek3d toLight,        // Richtung zur Lichtquelle
                   Vek3d n,              // Flaechennormale in D
                   Vek3d toEye)          // Richtung zum Beobachter
```

Das Resultat der Funktion ist die berechnete Helligkeit gemäss Gleichung 5.1, als reelle Zahl im Intervall $[0, 1]$. Wenn der Punkt P der Fläche nicht beleuchtet wird, ist das Resultat der Funktion negativ.

5.6 Boden mit Schatten

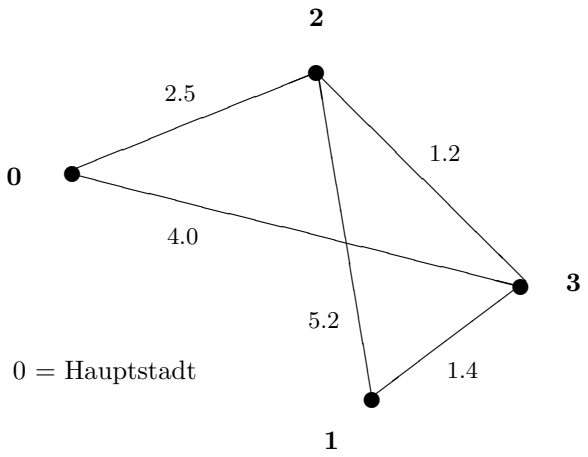
Mit wenig Aufwand kann im Applet 'MetallKugel' der im Bild auf Seite 117 dargestellte Boden mit dem Schatten der Kugel dargestellt werden. Dazu ist folgendermassen vorzugehen:

- Für jeden Strahl, der die Kugel nicht trifft, wird untersucht, ob er auf den Boden (z.B. auf der Höhe $z = -1.4 \cdot Kugelradius$) auftrifft.
- Für einen solchen Lichtstrahl wird vom Auftreffpunkt D auf dem Boden ein zweiter Lichtstrahl gegen die Lichtquelle ausgesandt. Wenn dieser Lichtstrahl die Kugel durchstösst, liegt D im Schatten der Kugel.

Anwendung 6

Kürzeste Wege

Gegeben ist ein Netz von Städten, die mit Wegen gegebener Längen verbunden sind. Dabei muss nicht jede Stadt mit jeder verbunden sein. Die Städte sind identifiziert mit Nummern 0, 1, ...



Für das Problem der Bestimmung der kürzesten Wege von der Hauptstadt zu den anderen Städten gibt es eine elegante Methode ohne Mathematik, mittels *Simulation*. Sie wurde von N. Wirth publiziert:

One day, the Chinese Emperor issued the order that the minimal distance to each of his empire's villages from the capital was to be determined. The method to be followed was the following: Large groups of

scouts were to march out into the country, notably at constant speed, in each direction, i.e. on each emanating path. Once they would encounter the next village, the group would split up, each subgroup proceeding on an outgoing path. One man would march back to report the time it took to reach the village, another would remain to report to groups arriving later that he had been first.

N.Wirth: Schemes for Multiprogramming and their Implementation in Modula-2 (ETH Technical Report 1984)

Daten-Modell

Das Städte-Netz wird in einem Array von Städten gespeichert. Dabei ist die Nummer einer Stadt gleich dem Index im Array. Zu jeder Stadt werden die folgenden Daten gespeichert:

- Array mit den wegführenden Wegen (Ziel, Länge)
- Variable 'besucht' die angibt, ob bei der Simulation schon ein Läufer in der Stadt angekommen ist.
- Variable 'herkunft' mit der Nummer der Stadt, von welcher der erste Läufer gekommen ist (für Rückverfolgung des Weges bei der Ausgabe).

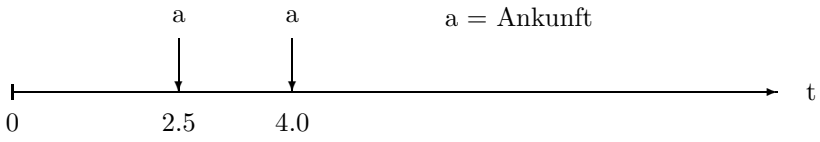
Simulation (Discrete Event Simulation)

Wir verwenden das Prinzip der Discrete Event Simulationen. Dabei werden Ereignisse (Events) zu bestimmten Zeitpunkten verarbeitet. Die Zeit wird in einer Variablen t (double) nachgeführt, wobei nur Zeiten von Ereignissen durchlaufen werden, d.h. in unserem Fall Ankünfte von Läufern in einer Stadt.

Beim Start der Simulation ($t = 0$) wird von der Hauptstadt aus in jede Richtung ein virtueller Läufer ausgesandt. Die zugehörigen Ankünfte der Läufer an ihren Zielen werden als Ereignisse (Events) in eine verkettete Liste, die sog. *Event-Queue* der Simulation, gestellt. Die Event-Queue ist *sortiert* nach den Zeiten der Ereignisse.

Event-Queue

Die Geschwindigkeit der Läufer sei gleich 1, sodass die Laufzeit gleich der Weg-Strecke ist.



Nach der Initialisierung wird die Event-Queue sequentiell abgearbeitet, d.h. die Ereignisse werden in der richtigen zeitlichen Reihenfolge verarbeitet. Dabei werden auch weitere Ereignisse in die Queue gestellt.

Verarbeitung einer Ankunft:

- Wenn die betreffende Stadt schon besucht wurde, ist keine Verarbeitung erforderlich: return.
- aktuelle Zeit nachführen: t = Zeit des Ankunft-Ereignisses
- Stadt als 'besucht' markieren
- Ausgabe der Ankunftszeit und der Stationen des Weges:

  t = 3.7 : Ankunft in Stadt 3 von: 2 0

- Folgeläufer aussenden: für jeden wegführenden Weg (ausser für den Herkunftsweg) werden Ankunfts-Ereignisse erzeugt und zu den entsprechenden Zeiten in die Event-Queue gestellt.

Programm-Output:

```
t = 2.5 :   Ankunft in Stadt  2   von:  0
t = 3.7 :   Ankunft in Stadt  3   von:  2  0
t = 5.1 :   Ankunft in Stadt  1   von:  3  2  0
```

Bemerkung:

Die Initialisierung der Simulation kann auch so erfolgen, dass eine Ankunft zur Zeit $t = 0$ in der Hauptstadt in die Event-Queue gestellt wird.

Index

0
8-Damen Problem 113

A
abstract 75
abstrakte Klasse 75, 84
abstrakte Methoden 75
ActionListener 84
Attraktor 112
Auslösung einer Exception 92

B
Basis-Klasse 63
Baum 50
Bild-Algorithmus 103
Bildpunkt 80
binäre Bäume 42, 50, 85
Binäre Suche 20
Blatt 50
Bubble-Sort 25, 38

C
Call Frame 5
Call Stack 5
catch 94
Chaos-Theorie 109
Clock 37
ComparableObj 87

D
dequeue 47
deterministisches Chaos 112
diffuse Reflexion 123
Directory 13
Discrete Event Simulation 132
doppelt verkettete Liste 49
Drehung 80
Dualziffern 10
Durchstosspunkte 120
Dynamische Datenstrukturen 41
Dynamisches System 110

E
Einfügeposition 54
Einfügung Knoten 53
enqueue 47
Ereignisverarbeitung 84
Erweiterungsklasse 63
Escape-Funktion 102
Escape-Number 102
Event-Queue 133
Exceptions 91
Explizite Konversionen 71

F
Fakultät 7
Fibonacci-Folge 9
FIFO 47
File 13
finally 96
Fraktal 99

G
Geraden 120
Geschwindigkeitsfeld 110
Gesetz von Lambert 123
Gruppenverarbeitungen 34

H
Hoare 27
Höhe eines Baumes 57, 62

I
IBM 60
Implementierung 80
Inorder-Traversierung 56
instanceof 71
Interfaces 79
Iteration 5
iterativ 5

K
Key 21
KeyListener 79
Klassenerweiterung 63, 84
Knoten 50
Konstruktor 64, 65
Konto-Verarbeitung 40
Kopie eines Baumes 62
Kugel 121
Kürzeste Wege 131

L
Leaf 50
LIFO 49
lightIntens 129
lineare Kongruenzmethode 59
Lineare Suche 20
Lorenz-Attraktor 109

M
Mandelbrot-Menge 99
Mehrfach-Vererbung 68
Merge 33
Methoden-Aufrufe 70
Mischen 33
Münzenproblem 16

N
new-Operator 80
Node 50

O
Object 76
Objekt-Referenzen 90
Orbit 100

P
Performance 56
Perioden 59
Permutationen 11
plzort.txt 36
Pointer 41
Polymorphismus 70
pop 49
Postleitzahlen 17, 36
Postorder 56
Potenz-Algorithmus 19
Precondition 19
Preorder-Traversierung 56
private 69
protected 69
Pseudozufallszahlen 59
public 69, 80
Punkt-Transformation 80
push 49

Q
qSort 27
Queue 47, 90
Quicksort 27

R
Rangliste 45
Raytracing-Verfahren 117
read ahead 35
Referenztyp 79
Referenzvariable 41
Rekursion 5
rekursiv 5
Root 50
Runtime-System 91, 93

S

Schatten 130
Schleifeninvariante 18
Schlüssel 21
Schmetterlings-Effekt 112
Schnittstelle 79
Select-Sort 24
Sierpinski-Dreieck 15
Simulation 131
SortDemo 38
Sortier-Algorithmen 24
sortieren 87
sortiert 51
sortierte Liste 45
sortierte verkettete Liste 74
Sortierzeit 37
Sortierzeiten 31
Spezifikation 79
spiegelnde Reflexion 127
Spiegelung 80
Stack 6, 49, 59, 72, 77
Stack Pointer 6
Stapel 6
Stop-Uhren 90
Strahlen-Verfolgung 117
Streckung 80
Strömung 110
Subklasse 67
Suchbaum 50, 51
Suche 52
super 64
Superklasse 67

T

Textfile 43
throw 92
totale Helligkeit 128
traverse 85
Traversierung 55
Tree 50
try/catch 94
Türme von Hanoi 115

U

überschreiben 63
Überschreiben von Methoden 65
UML Diagramm 68
Unified Modeling Language 68
Untermenge 67

V

Vek3d 119
Vektoren 119
Vektorfeld 110
vergleichbare Objekte 87
Verkettete Listen 42, 43
Verschiebung 80
Vielgestaltigkeit 70
Vollperiode 60
Vorauslesen 35
Vorbedingung 19

W

Warteschlange 47
Wurzel 50
Würfel-Kombinationen 16

Z

Zeiger 41
Zufalls-Generator 60
Zufallszahlen 59
Zugriffs-Kontrolle 69