Programmieren Zusammenfassung

Thomas Jund <info@jund.ch>
Laurent Cohn <info@cohn.ch>
Andrew Mustun <andrew@mustun.com>
Martin Suter <martin@suter.to>

25. Januar 2003

Version 2.0

Zusammenfassung für das Fach Datenstrukturen und Algorithmen des Studiengangs IT an der ZHW. Die Praktikas wurden mit **Java** durchgeführt, daher sind alle Code-Beispiele in Java geschrieben.

Die Buchreferenzen in dieser Zusammenfassung beziehen sich auf die Powerpoint-Präsentationen von Peter Troxler. Die Seitennummern sind nicht auf den Folien ersichtlich, sondern eigens für diese Dokumentation erstellt worden.

Inhaltsverzeichnis

1 OOP (Object Oriented Programming) 1
1.1 Begriffe 1

2 UML 6
2.1 Klassen 6
2.2 Abstrakte Klassen 6
2.3 Interfaces 7
2.4 Subsystem 7
2.5 Attribute 7
2.6 Sichtbarkeit 7
2.7 Kardinalität 7
2.8 Assoziation 8
2.9 Aggregation 8
2.10 Generalisierung 8

3 Datentypen 9
3.1 Definition 9
3.2 Datentypen in Java 9

4 Speicherverwaltung 9
4.1 Speicherverwaltung mit Stack 9
4.2 Speicherverwaltung mit Heap 9

5 Operatoren 10
5.1 Arithmetische Operatoren 10
5.1.1 binäre 10
5.1.2 unäre 10
5.2 Vergleichs Operatoren 10
5.3 Boolsche Operatoren 10
5.4 Bit Operatoren 10
5.5 Zuweisungs Operatoren 11
5.6 Geordnet nach Priorität 11

6 Escape Sequenzen 11

7 Schlüsselwörter 11

8 Zugriffsbeschränkung **12**

9 Listen **13**

9.1 Definition 13
9.2 Einfach verkettete Listen 13
9.3 Doppelt verkettete Listen 13
9.4 Geordnete Listen 13
9.5 Skip-Listen 13
9.6 Adaptive Listen 13
9.6.1 Move To Front 13
9.6.2 Transpose 13
9.6.3 Frequency Count 14
9.7 Beispiel 14

10 Bäume **17**

10.1 Begriffe 17
10.2 Traversierung 17
10.3 AVL Bäume 18

11 Algorithmen und Graphen-Theorie **19**

11.1 NP-vollständige Probleme 19
11.2 Heuristische Verfahren 19
11.3 Greedy-Algorithmen 19
11.4 Randomisierte Algorithmen 20
11.5 Warshalls Algorithmus 20
11.6 Dijkstra's Algorithmus 21
11.7 Travelling Salesman Problem 21

12 Mengen und Hash-Verfahren **22**

12.1 Mengen 22
12.1.1 Symbole 22
12.2 Hashing 22
12.3 Sondieren 22
12.3.1 Lineares Sondieren 22
12.3.2 Quadratisches Sondieren 22
12.3.3 Doppeltes Hashing 22

12.4 Belegungsfaktor 23
12.5 Kollisionen 23

13 Kompression **24**
13.1 Entropie 24
13.2 Huffman 24
13.2.1 Idee 24
13.2.2 Konstruktion des Huffman-Codes 24
13.2.3 Beispiel 25
13.3 RLE (Run Length Encoding) 27

14 Rekursion **28**
14.1 Abstrakte Formulierung 28
14.2 Codebeispiel 28
14.2.1 Iterativ, direkt/indirekt Rekursiv 28
14.2.2 Mathematische Beispiele 29
14.3 Vorteile gegenüber Iterativen Verfahren 29
14.4 Nachteile gegenüber Iterativen Verfahren 29
14.5 Backtracking 30
14.5.1 Definition 30
14.5.2 Code Beispiel 30
14.6 Direkte Rekursion vs. Endrekursion (Tail Recursion) 30
14.7 Begriffe 31

15 Sortieralgorithmen **32**
15.1 Bubble Sort 32
15.1.1 Aufwand 32
15.1.2 Stabilität 32
15.1.3 Vor- und Nachteile 32
15.1.4 Programmierbeispiel 1 32
15.1.5 Programmierbeispiel 2 33
15.2 Shaker Sort 33
15.3 Selection Sort 34
15.3.1 Aufwand 34
15.3.2 Stabilität 34
15.3.3 Vor- und Nachteile 34

15.3.4 Programmierbeispiel . . . 34
15.4 Insertion Sort . . . 35
15.4.1 Stabilität . . . 35
15.4.2 Vor- und Nachteile . . . 35
15.4.3 Programmierbeispiel . . . 35
15.5 Shell Sort . . . 36
15.5.1 Aufwand . . . 36
15.5.2 Vor- und Nachteile . . . 36
15.6 Comb Sort . . . 36
15.6.1 Vor- und Nachteile . . . 36
15.7 Heap Sort . . . 36
15.7.1 Aufwand . . . 36
15.8 Rekursive Sortieralgorithmen . . . 36

16 Zeitkomplexität **37**
16.1 Wichtigste Aussagen zur Zeitkomplexität . . . 37

17 Events **38**
17.1 Begriffe, Anwendungen, Nachteile . . . 38
17.2 Programmbeispiel . . . 38
17.3 Button . . . 39
17.4 Label . . . 39
17.5 MouseListener . . . 40
17.6 MouseMotionListener . . . 40

18 Exceptions **41**
18.1 throw . . . 41
18.1.1 Programmbeispiel . . . 41
18.2 try und catch . . . 41
18.2.1 Programmbeispiel . . . 42
18.3 Exception und Error . . . 42
18.4 try-catch-finally . . . 43
18.5 Die Klasse RuntimeException . . . 43

19 File Handling **44**
19.1 Arten von Dateien . . . 44

19.2 Zugriffsarten 44
19.3 File-Objekt 44
19.4 Methoden der Klasse File 45
19.5 Random-Access-Datei 45
19.5.1 Öffnen 45
19.5.2 Schliessen 46
19.5.3 Lesen 46
19.5.4 Dateizeiger 47
19.5.5 Schreiben 47
19.6 Streams 47
19.6.1 Lesezugriff 47
19.6.2 Schreibzugriff 48

20 Threads **49**
20.1 Begriffe 49
20.2 Beeinflussung von Threads 49

1 OOP (Object Oriented Programming)

1.1 Begriffe

Quelle der Originalfassung: `http://www.informatik.hs-bremen.de/~brey/glossar.pdf`

- **Abstrakter Datentyp** Ein Abstrakter Datentyp fasst Daten und die Funktionen, mit denen die Daten bearbeitet werden dürfen, zusammen. Der Sinn liegt darin, den richtigen Gebrauch der Daten sicherzustellen. Mit *Funktion* ist hier nicht die konkrete Implementierung gemeint, das heisst, wie die Funktion im Einzelnen auf die Daten wirkt. Zur Benutzung eines Abstrakten Datentyps reicht die Spezifikation der Zugriffsoperation aus. Ferner sind logisch zusammengehörige Dinge an einem Ort konzentriert. Ein Abstrakter Datentyp ist ein Typ zusammen mit Datenkapselung. Eine Klasse in Java ist ein Abstrakter Datentyp.
- **Abstrakte Klasse** Eine abstrakte Klasse ist eine Klasse, von der es keine Instanzen gibt. Abstrakte Klassen definieren Schnittstellen, die durch abgeleitete Klassen implementiert werden müssen. S.194
- **Abstrakte Methode** Methode, die auf Stufe der ⇒ Oberklasse nicht implementiert wird (jedoch von den ⇒ Unterklassen). S.193
- **Aggregation** Die Aggregation ist ein Spezialfall der Assoziation, der Enthaltensein (Teil-Ganzes-Beziehung) beschreibt. Wenn im Ganzen nur Verweise auf die Teile existieren, sind diese nicht existentiell abhängig vom Ganzen. Andernfalls spricht man auch von Komposition.
- **Attribut** Attribute beschreiben die Eigenschaften eines Objekts. Der aktuelle Zustand eines Objekts wird durch die Werte der Attribute beschrieben. Zum Beispiel kann Farbe ein Attribut sein; ein möglicher Attributwert wäre rot. S.189
- **Ausnahme** Eine Ausnahme (englisch exception) ist die Verletzung der ⇒ Vorbedingung einer Operation (⇒ Methode) einer Klasse. Java bietet die Möglichkeit, Ausnahmen zu erkennen und zu behandeln (exception handling).
- **Assoziation** Die Assoziation ist eine gerichtete Beziehung zwischen Klassen. Sie kann in eine Richtung verweisen (A kennt B, aber nicht umgekehrt) oder bidirektional sein (A kennt B und B kennt A).
- **Bindung** ⇒ dynamische Bindung, ⇒ statische Bindung
- **Behälterklasse** Datenstruktur zur Speicherung von Objekten. Beispiele: Array, Liste, Vektor
- **Botschaft** In der rein objektorientierten Programmierung wird davon ausgegangen, dass ein laufendes Programm(-system) aus einer Menge von Objekten besteht, die miteinander über Botschaften (englisch messages) kommunizieren. Ein genauerer Begriff als Botschaft ist *Aufforderung*, weil das empfangende Objekt etwas tun soll. Ein Objekt, das eine Aufforderung erhält, führt eine dazu passende Operation (⇒ Methode) aus, die in der ⇒ Klasse beschrieben ist.

- **Client** Im informationstechnischen Sprachgebrauch heissen Dinge (Objekte, Rechner, ...), die eine Dienstleistung erbringen, Server. Die Dienstleistung wird erbracht für einen Client (deutsch: Klient, Kunde), der selbst ein Rechner oder Objekt sein kann.

- **Container** ⇒ Behälterklasse

- **Daten** Der Zustand eines Objekts wird durch seine Daten beschrieben. Die Daten sind die Werte der ⇒ Attribute eines Objekts.

S.172

- **Datenkapselung** Datenkapselung ist das *Verstecken* der Daten eines Objekts vor direkten Zugriffen. Zugriffe sind nur über die öffentliche ⇒ Schnittstelle der Datenkapsel (⇒ Abstrakter Datentyp) möglich. Datenbezogene Fehler sind damit leicht lokalisierbar. In Java wird Datenkapselung mit der Zugriffsspezifikation private realisiert.

- **Definition** Eine Definition liegt vor, wenn mehr als nur der Name eingeführt wird, zum Beispiel wenn Speicherplatz angelegt werden muss für Daten oder Code oder die innere Struktur eines Datentyps beschrieben wird, aus der sich der benötigte Speicherplatz ergibt. Weil auch ein Name eingeführt wird, ist eine Definition immer auch eine Deklaration. Die Umkehrung gilt nicht.

- **Deklaration** Eine Deklaration teilt dem Compiler mit, dass eine Funktion (oder eine Variable) mit diesem Aussehen irgendwo definiert ist. Damit kennt er den Namen bereits, wenn er auf einen Aufruf der Funktion stösst und ist in der Lage, eine Syntaxprüfung vorzunehmen. Eine Deklaration führt einen Namen in ein Programm ein und gibt dem Namen eine Bedeutung. Eine Deklaration kann gleichzeitig eine ⇒ Definition sein.

- **Exception** ⇒ Ausnahme

- **Identität** Ein Objekt besitzt eine Identität, die es unterscheidbar macht von einem beliebigen anderen Objekt, selbst wenn beide gleiche Daten enthalten. Die Identität zu einem bestimmten Zeitpunkt wird durch eine eindeutige Position im Speicher gewährleistet; zwei Objekte können niemals dieselbe Adresse haben, es sei denn, ein Objekt ist im anderen enthalten. Java hat keine Sprachmittel für die Identität. Die Adresse als Identitätsmerkmal gilt nur für ein ⇒ vollständiges Objekt, und dann auch bei Mehrfachvererbung. Falls dies für eine Anwendung nicht ausreichend ist, muss die Identität durch ein eigens für diesen Zweck vorgesehenes Element des Objekts definiert werden, zum Beispiel durch eine Seriennummer.

- **Instanz** Eine Instanz einer Klasse ist eine andere Bezeichnung für ein ⇒ Objekt. Die Erzeugung eines Objekts wird auch Instantiierung genannt.

- **Interface** ⇒ Schnittstelle

- **Initialisierung** Wenn ein Objekt während der Erzeugung mit Anfangsdaten versehen wird, heisst der Vorgang Initialisierung. Die Initialisierung ist die Aufgabe eines Konstruktors. Sie ist von der ⇒ Zuweisung zu unterscheiden.

- **Kapselung** ⇒ Datenkapselung

S.188

- **Klasse** Eine Klasse definiert die Merkmale (Daten) und das Verhalten (Operationen, Methoden) einer Menge von Objekten. Eine Klasse ist ein Datentyp, genauer: ein ⇒

Abstrakter Datentyp. In Java gilt die Umkehrung (ein Datentyp ist eine Klasse) nicht, weil die Grunddatentypen (zum Beispiel int) nicht als Klasse implementiert sind. Eine Klasse definiert die Struktur aller nach ihrem Muster erzeugten Objekte, entweder direkt oder indirekt durch ⇒ Vererbung.

- **Klassifikation** Klassifikation ist ein Verfahren, um Gemeinsamkeiten von Dingen herauszufinden und auszudrücken. Von Unterschieden wird abstrahiert. In Java wird ein Satz gleicher Merkmale und Verhaltensweisen durch die ⇒ Klasse beschrieben.
- **Mehrfachvererbung** Eine Klasse kann in C++ von mehr als einer Klasse erben (⇒ Vererbung). In Java nicht.
- **Methode** S.190
  - **Objektmethode** Objektmethode ist eine andere Bezeichnung für eine Operation, die auf den Daten eines ⇒ Objekts ausgeführt werden kann. In C++ heissen Objektmethoden auch Elementfunktionen (englisch member functions), um auszudrücken, dass eine Objektmethode ein Element einer Klasse ist. Sie unterscheidet sich von einer "normalen" Methode auch dadurch, dass sie Zugriff auf die internen Daten des ⇒ Objekts hat.
  - **Klassenmethode** Eine Klassenmethode wirkt sich auf die Gesamtheit aller Objekte einer Klasse aus oder ist unabhängig vom Zustand des Objekts. Deklaration in Java mit *static*. Aufruf mit *Klassenname.methodenName(. . .);*.
  - **Konstruktor** Initialisiert das Objekt und wird in Java als Methode mit gleichem Namen wie die Klasse implementiert.
- **Modell** Ein Modell ist eine Abstraktion eines Systems. Modelle werden verwendet, um die Anforderungen eines Systems zu erheben, es zu verstehen, überprüfen, damit zu experimentieren und darüber mit anderen zu kommunizieren.
- **Nachricht** ⇒ Botschaft
- **Oberklasse** Es kann verschiedene Klassen geben, die gemeinsame Anteile enthalten. Diese Anteile können "herausgezogen" werden und bilden eine Oberklasse. Die Klassen werden dann als Spezialisierung der Oberklasse aufgefasst, weil sie nur noch die Unterschiede beschreiben. Zum Beispiel haben eine Tanne und eine Eiche die gemeinsame Eigenschaft, ein Baum zu sein mit all seinen Merkmalen. In einer Beschreibung (Klasse) für eine Tanne genügt es, auf die Oberklasse "Baum" zu verweisen (⇒ Vererbung) und nur die Besonderheit "Nadeln" anzugeben. Eine Oberklasse ist eine durch ⇒ Klassifikation gewonnene Abstraktion in der Form einer "ist-ein"-Beziehung. Ein Tanne "ist-ein" Baum eine Eiche auch. Eine Oberklasse kann selbst wieder von einer weiteren Oberklasse erben. Manchmal wird eine Oberklasse oder die oberste Oberklasse "Basisklasse" (englisch base class) genannt.
- **Objekt** Ein Objekt ist die konkrete Ausprägung des durch eine ⇒ Klasse definierten Datentyps. S.187
  - **Daten** Ein Objekt hat einen inneren *Zustand*, der durch Attribute in Form von anderen Objekten oder Elementen der in der Programmiersprache vorgegebenen Datentypen dargestellt wird.

  - **Methoden** Der Zustand kann sich durch *Aktivitäten* (*Verhalten*) des Objekts ändern, also durch Ausführen von Operationen auf Objektdaten.
  - **Identität** Jedes Objekt hat eine Identität, sodass auch gleiche Objekte unterscheidbar sind.

  Im Ablauf eines Programms werden Objekte erzeugt (und wieder gelöscht), die aufgrund des Empfangs einer ⇒ Botschaft hin aktiv werden. Die Menge aller möglichen Botschaften für ein Objekt heisst ⇒ Schnittstelle.

- **Operation** ⇒ Methode, ⇒ Botschaft

- **Packages** ⇒ Packete

S.199

- **Packete** Mit Packeten können in Java Klassen gruppiert werden.

S.172

- **Polymorphismus** Methoden können in Unterklassen den gleichen Namen tragen wie in den Oberklassen.

  Allgemeiner: Die Fähigkeit von Programmelementen, sich zur Laufzeit auf ⇒ Objekte verschiedener ⇒ Klassen beziehen zu können, heisst Polymorphismus. Anders formuliert: Erst zur Laufzeit eines Programms wird die zu dem jeweiligen Objekt passende Realisierung einer Operation ermittelt. In Java müssen diese Klassen in einer Vererbungsbeziehung stehen (⇒ Vererbung).

S.196

- **Schnittstelle** Als (öffentliche) Schnittstelle (englisch (public) interface) bezeichnet man die Menge von Aufforderungen, auf die ein ⇒ Objekt reagieren kann. In Java werden Schnittstellen durch die ⇒ Deklarationen der public- ⇒ Methoden beschrieben.
  Interfaces in Java sind rein ⇒ Abstrakte Klassen. In Ihnen werden Methoden deklariert aber nicht implementiert.

- **Server** ⇒ Client

- **Unterklasse** Eine Klasse, zu der eine ⇒ Oberklasse existiert, heisst Unterklasse bezüglich dieser Oberklasse. Wenn ein Objekt der Unterklasse stets an die Stelle eines Oberklassenobjekts treten kann, ist die Unterklasse ein ⇒ Subtyp der Oberklasse. Eine Unterklasse heisst auch "abgeleitete Klasse" (englisch derived class).

S.172, 192

- **Vererbung** Vererbung wird definiert durch eine Beziehung zu einer ⇒ Oberklasse, um deren Merkmale und Verhaltensweisen zu übernehmen. Eine Klasse "erbt" von Oberklassen, indem die direkten Oberklassen in der Klassendefinition angegeben werden. Gleichzeitig wird damit von allen Oberklassen der Oberklasse geerbt, sofern sie existieren. Der Aufruf einer Operation für ein Objekt lässt nicht erkennen, ob sie der Klasse des Objekts oder einer Oberklasse zuzuordnen ist, also geerbt wurde. Der Vorteil liegt darin, dass eine für mehrere Klassen benötigte Funktion nur einmal in der gemeinsamen Oberklasse definiert sein muss. Notwendige Änderungen sind dann nur noch an einer statt an vielen Stellen vorzunehmen, was die Fehlerwahrscheinlichkeit reduziert.

- **Zustand** Der Zustand eines Objekts ist definiert durch die Menge der Werte der ⇒ Attribute dieses Objekts.

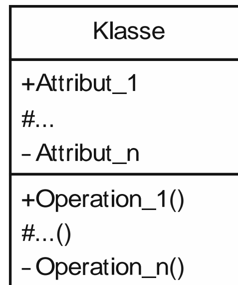
- **Zuweisung** Eine Zuweisung weist ein Objekt dem anderen zu und ändert damit dessen Wert. Im Unterschied zur ⇒ Initialisierung muss das zu ändernde Objekt vor der Zuweisung bereits existieren.

2 UML

S.203

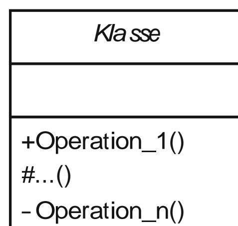
2.1 Klassen

Eine Klasse ist eine Beschreibung gleichartiger Objekte. Jedes Objekt gehört zu (ist Instanz von) genau einer Klasse.
Notation:



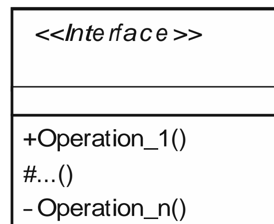
2.2 Abstrakte Klassen

Eine Klasse die nicht instanziert werden kann, nennt man eine abstrakte Klasse. Sie enthält meistens keine Attribute. Sie stellt Teilimplementationen zur Verfügung oder wird verwendet um zukünftige Methodenschnittstellen für Unterklassen vorzugeben.
Notation:



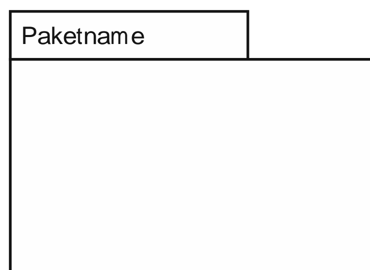
2.3 Interfaces

Interfaces werden auch als rein abstrakte Klassen bezeichnet. Sie gibt eine Menge von Methodenschnittstellen vor. Wenn eine Klasse ein Interface nutzt, muss sie die vorgegebenen Schnittstellen implementieren.
Notation:



2.4 Subsystem

Das Subsystem ist ein sogenanntes **Ist-eingebettet-in-Beziehung**.
Notation:



2.5 Attribute

2.6 Sichtbarkeit

Die Sichtbarkeit muss bei Attributen angegeben werden.

+	global (public visibility)
#	geschützt (protected visibility)
-	privat (private visibility)

2.7 Kardinalität

S.214

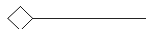
[n..m]	mindestend n-, höchstens m-mal
[n..*]	mindestens n-mal, beliebig oft
[0..1]	optionales Attribut

2.8 Assoziation

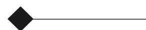
Die Assoziation ist eine sogenannte **Benutzt-Beziehung**.
Dies drückt die Abhängigkeit von Klassen aus. Eine Klasse Benutzt eine andere Klasse, wenn ihre Methoden die Objekte der Klasse beeinflusst.
Notation:

2.9 Aggregation

Die Aggregation ist eine sogenannte **Hat-Beziehung**.
Der Unterschied zwischen der Assoziation und der Aggregation ist vielfach schwer zu Unterscheiden. Faustregel: Wenn man "besteht aus" sagen kann, handelt es sich um eine Aggregation.
Notation:

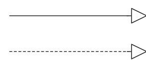


Hat-Exklusiv-Beziehung, dies ist ein Spezialfall der Hat-Beziehung. Sie drückt eine stärkere Bindung als die Hat-Beziehung aus. Ein Unterscheidungsmerkmal ist, dass Teile die gleiche Lebensdauer haben.
Notation:



2.10 Generalisierung

Die Generalisierung ist eine sogenannte **Ist-Beziehung**.
Man kann die Generalisierung ganz einfach als **Vererbung** ansehen. Sie ist aber nicht mit der Klassifikation zu verwechseln, dass heisst die Instanz oder das Objekt der Klasse ist keine Generalisierung sondern eine Klassifikation.
Notation:



3 Datentypen

S.23

3.1 Definition

Ein Datentyp ist die Festlegung der Interpretation einer gespeicherten Bitfolge. Datentypen können einfacher Art sein (ganze oder reelle Zahlen, logische Werte, Zeichen) oder zusammengesetzt (also komplex) bzw. strukturiert sein (Felder, Zeichenfolgen, Strukturen usw.).

3.2 Datentypen in Java

Java kennt folgende Datentypen:

Name	Wertebereich	Grösse in Bit
byte	-128 ... 127	8
short	-32'768 ... 32.767	16
int	-2'147'483'648 ... 2'147'483'647	32
long	-9'223'372'036'854'775'808 ... 9'223'372'036'854'775'807	64
char	\u0000 (0) ... \uFFFF (65'535)	16 (Unicode)
boolean	false, true	1
float	± (1.4023e-45 ... 3.4028e+38)	32 (IEEE 754)
double	± (4.9406e-324 ... 1.7976e-308)	64 (IEEE 754)

4 Speicherverwaltung

4.1 Speicherverwaltung mit Stack

- Verwaltung durch Laufzeitsystem
- Arbeitet nach dem Prinzip "last-in-first-out"
  Es gibt also zwei Operationen: Push und Pop
- Verwaltung von Funktionsaktivierungen
- Speicherbereich für lokale Variablen

Vorteile: Die Adressverwaltung wird durch das System übernommen. Der Benutzer muss nicht die Adresse des Elements kennen, welches er mit Push auf den Stapel legt.
Stack ist ideal als Zwischenspeicher. Das System muss nur jene Adresse im Hauptspeicher verwalten, an der sich das oberste Element des Stacks befindet.

4.2 Speicherverwaltung mit Heap

- Programmkontrollierte Verwaltung
- Speicherzuweisung per Anweisung (new/delete)
- Zugriff über Zeiger-Variablen

5 Operatoren

S.28

5.1 Arithmetische Operatoren

operieren auf Zahlen

5.1.1 binäre

```
+, -, *, / und % (Modulo, Divisionsrest)
```

5.1.2 unäre

```
+ , -    Vorzeichen
++       Pre- / Postinkrement      nur auf Variablen
--       Pre- / Postdekrement      nur auf Variablen
```

5.2 Vergleichs Operatoren

liefern boolean-Wert

```
== , !=         gleich, ungleich    auf bel. Datentypen
<, <=, >, >=    Vergleiche          auf arithmetischen Typen (Zahlen und chars)
instanceof      Typ Ueberpruefung   auf Objekte
```

5.3 Boolsche Operatoren

operieren auf boolean-Werten

```
!     Nicht (NOT, unaer)
&&    bedingtes Und
||    bedingtes Oder
```

5.4 Bit Operatoren

operieren auf integralen Typen (byte, short, int, long, char)

```
~    bitweises Komplement (unaer)
&    bitweises Und
|    bitweises Oder
^    bitweises XOR
<<   Linksshift
>>   arithm. Rechtsshift
>>>  logischer Rechtsshift
```

5.5 Zuweisungs Operatoren

```
Zuweisung:                =
Zuweisung mit Operation:  +=, -=, *=, /=, %=, &=, |=, ^ =, <<=, >>=, >>>=
```

5.6 Geordnet nach Priorität

```
Postfix Operatoren        [] . (params) expr++ expr--
Unaere operatoren         ++expr --expr +expr -expr  ~ !
Erstellung und Cast       new (type)expr
Punkoperatoren            *  %
Strichoperatoren          +  -
Shift                     << >> >>>
Beziehungen               < > <= >= instanceof
Vergleiche                == !=
Bitweises AND             &
Bitweises exclusives OR   ^
Bitweises inclusives OR   |
Logisches AND             &&
Logisches OR              ||
Bedingung                 ? :
Zuweisung                 = += -= *= /= %= ^= &= <<= >>= >>>=
```

6 Escape Sequenzen

S.52

```
\b     /* \u0008: Backspace BS */
\t     /* \u0009: horizontaler Tabulator HT */
\n     /* \u000a: Zeilenendezeichen LF */
\f     /* \u000c: Formfeed (Seitenvorschub) FF */
\r     /* \u000d: Carriage Return (Zeilenruecklauf) CR */
\"     /* \u0022: Anfuehrungszeichen " */
\'     /* \u0027: Apostroph ' */
\\     /* \u005c: Backslash \ */
\uXXXX /* Unicode Zeichen mit Hexcode XXXX */
```

7 Schlüsselwörter

```
abstract     default      if           private      throw
boolean      do           implements   protected    throws
break        double       import       public       transient
byte         else         instanceof   return       try
case         extends      int          short        void
catch        final        interface    static       volatile
char         finally      long         super        while
```

```
class       float       native      switch
const       for         new         synchronized
continue    goto        package     this
```

Bemerkung: *const* und *goto* sind ebenfalls reserviert, werden aber nicht verwendet.

8 Zugriffsbeschränkung

S.69

Specifier	class	subclass	package	world
private	x			
protected	x	x*	x	
public	x	x	x	x
package	x		x	

(*) Kein Zugriff einer Unterklasse auf protected elemente der Oberklasse, wenn Ober- und Unterklasse nicht im gleichen Package sind.

9 Listen

S.221

9.1 Definition

Folge von Elementen, in der sich an beliebiger Stelle Elemente hinzufügen oder entfernen lassen.

9.2 Einfach verkettete Listen

S.230

Jedes Element hat einen Pointer auf das nachfolgende Element. Das letzte zeigt auf null.

9.3 Doppelt verkettete Listen

S.234

Jedes Element hat einen Pointer auf das nachfolgende und einen Pointer auf das vorherige Element.

9.4 Geordnete Listen

S.235

Listen, die nach einem Kriterium (Ordnungsrelation) geordnet sind.

9.5 Skip-Listen

S.235

Geordnete Listen, die nebst den einfachen Pointern auch noch Pointer besitzen, die auf das nächste oder übernächste Element, .. verweisen. Dies führt zu einer Beschleunigung der Suche.

9.6 Adaptive Listen

S.238

Idee: Häufig gesuchte Elemente sollen an den Anfang der Liste verschoben werden und damit schneller gefunden werden.

9.6.1 Move To Front

Beim Zugriff auf ein Element wird dieses an den Anfang der Liste verschoben.

Vorteil: Häufig gesuchte Elemente kommen schnell nach vorne
Nachteil: Auch ein selten gesuchtes Element kommt schnell nach vorne und wandert dann nur langsam wieder nach hinten.

9.6.2 Transpose

Beim Zugriff auf ein Element wandert diese um eine Position nach vorne (wird mit dem vorhergehenden Element vertauscht) falls möglich.

Vorteil: Nur Elemente, auf die häufig zugegriffen wird kommen mit der Zeit nach vorne.
Nachteil: Die Liste passt sich nur langsam neuen Zugriffshäufigkeiten an.

9.6.3 Frequency Count

Jedes Element speichert, wie oft darauf zugegriffen wird. Die ganze Liste wird "gelegentlich" sortiert.

Vorteil: Die Optimierung kann auf Befehl des Programmierers erfolgen (z.B. wenn das System nicht benutzt wird).
Nachteil: Speicherverbrauch durch Counters, Keine kontinuierliche Anpassung

9.7 Beispiel

Beispiel Implementation für eine einfach gelinkte Liste.

```
/**
 * Element in the list .
 */
class Element {
    public Object o;
    public Element next;

    public Element(Object o) {
        this.o = o;
        next = null;
    }
}

/**
 * Linked list .
 */
public class LinkedList {
    Element rootElement;        // root element
    Element currentElement;     // cursor

    public LinkedList(Object o) {
        currentElement = rootElement = new Element(o);
    }

    public void addElement(Object o) {
        currentElement = currentElement.next = new Element(o);
    }

    public void removeFirstElement() {
        rootElement = rootElement.next;
    }
```

```
public void insertAt(int index, Object o) {
    Element insertPos = findElementAt(index);
    Element newElement = new Element(o);
    newElement.next=insertPos.next;
    insertPos.next = newElement;
}

public void deleteAt(int index) {
    currentElement = rootElement;
    int counter=0;

    while (counter!=index) {
        nextElement();
        ++counter;
    }

    if (currentElement.next!=null) {
        currentElement.next = currentElement.next.next;
    }
}

public int size() {
    currentElement = rootElement;
    int counter=0;

    while(currentElement.next!=null) {
        nextElement();
        ++counter;
    }

    return counter;
}

public Object nextElement() {
    return (currentElement=currentElement.next);
}

public Element findElementAt(int index) {
    currentElement = rootElement;
    int counter=0;

    while(counter!=index) {
        nextElement();
        ++counter;
    }
```

```
        if (currentElement.next!=null) {
            currentElement.next = currentElement.next.next;
        }

        return currentElement;
    }
}
```

10 Bäume

S.239

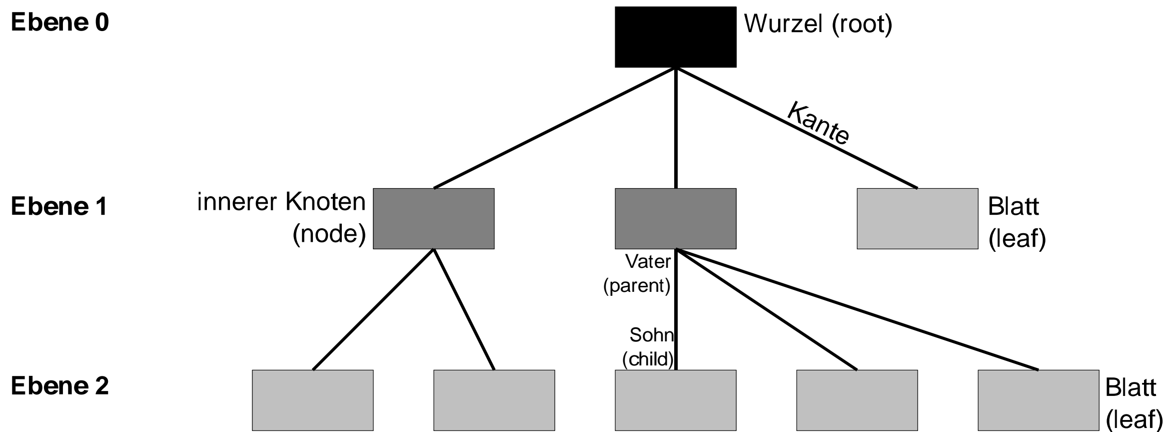


Abbildung 1: Beispiel eines Baumes

10.1 Begriffe

- **Grad:** Anzahl Kanten, die maximal von einer Wurzel ausgehen. S.240
- **Gewicht:** Anzahl Knoten des Baumes
- **Pfad:** Folge von Knoten, die durch Kanten verbunden sind. S.241
- **Länge des Pfades:** Anzahl Knoten eines Pfades
- **Tiefe eines Knotens:** Länge des Pfades von der Wurzel bis zum Knoten
- **Tiefe des Baumes:** Tiefe des am tiefsten liegenden Knotens (im Beispiel von Abbildung 1 ist die Tiefe des Baumes 3)
- **Voll:** Ein Baum vom Grade g heisst voll, wenn ausser der letzten alle Ebenen vollständig besetzt sind (d.h. g^k Knoten haben).
- **Komplett:** Der Baum ist voll und die Blätter auf der letzten Ebene sind linksbündig und dicht angeordnet S.242
- **Binärbaum:** Baum vom Grade 2
- **Balanciert:** Ein Baum mit N Knoten heisst balanciert, falls für die max. Pfadlänge Pf_{max} des Baumes $Pf_{max} \leq \lfloor \log_2(N) \rfloor + 1$ gilt. ⇒ Gute Suchzeiten S.255

10.2 Traversierung

S.249

- **Preorder:** Wurzel, linker Teilbaum in Preorder, rechter Teilbaum in Preorder S.249
- **Postorder:** linker Teilbaum in Postorder, rechter Teilbaum in Postorder, Wurzel

- **Inorder:** linker Teilbaum in Inorder, Wurzel, rechter Teilbaum in Inorder S.250
- **Levelorder:** Wurzel, Wurzel des linken Teilbaums, Wurzel des rechten Teilbaums, u.s.w.

S.257

10.3 AVL Bäume

Definition: Binärer Suchbaum, für den gilt: Unterschied der Tiefe des rechten und linken Teilbaums ist höchstens 1.

Vorteil: Im allgemeinen besser balanciert.

Nachteil: Einfügen, Löschen sehr aufwändig

11 Algorithmen und Graphen-Theorie

11.1 NP-vollständige Probleme

Um von einem Problem sagen zu können, da es prinzipiell schwierig zu lösen sein wird, muss man nachweisen, dass es sich auf ein bereits bekanntes, NP-vollständiges Problem reduzieren lässt.
In der Praxis wird bei praktisch unlösbaren Problemen nicht die bestmögliche Lösung gesucht, sondern eine Lösung, die mit vertretbarem Aufwand zu einer akzeptablen Lösung führt. Dazu werden verschiedene Verfahren benutzt:

- Greedy-Algorithmen
- Randomisierte Ansätze
- Backtracking
- Heuristische Verfahren

Es gibt eine ganze Anzahl praktisch relevanter Probleme, die NP-vollständig sind:

- **Travelling Salesman** Man finde eine kürzeste Rundreise durch eine gegebene Zahl von Städten, die genau einmal besucht werden.
- **Hamilton Zyklus** Existiert zu einer gegebenen Karte eine Rundreise, bei der jeder Ort genau einmal besucht wird.
- **Stundenplanung** Verteilen Sie eine Menge von Unterrichtsstunden so auf einen Zeitraum (etwa einer Woche), dass sie sowohl fr Schüler als auch Lehrer überschneidungsfrei und (möglichst) lückenfrei liegen.
- **Rucksackproblem** gegeben sind eine Anzahl von Waren verschiedener Grösse und von unterschiedlichem Wert. Man belade einen Wagen (Rucksack, ...) so, da man Waren von maximalem Gesamtwert transportiert.

11.2 Heuristische Verfahren

S.354

In Optimierungsaufgaben geht es darum, eine Zielfunktion über den Daten des Lösungsraums zu definieren und diese zu maxi- oder minimieren. Die Heuristik liefert im Entscheidungsfalle einen Anhaltspunkt, wie man ein lokales Maximum (oder Minimum) von der Zielfunktion erreichen kann. Das Resultat ist jedoch nicht immer global optimal.

11.3 Greedy-Algorithmen

Kurz: Immer die "lokal" beste Variante nehmen!

Greedy-Algorithmen funktionieren nach dem folgenden Prinzip:
Immer wenn eine Auswahlmöglichkeit für den nächsten auszuführenden Schritt besteht,

wähle die lokal "optimale" Möglichkeit.
Bei Greedy-Algorithmen werden Entscheidungen, die den Rechenprozess der Lösung näher bringen, auf der Basis der vom Rechenprozess bis dahin gesammelten Informationen gefällt und nicht mehr revidiert. Im Unterschied zu Verfahren, die Lösungsschritte ausprobieren und gegebenenfalls revidieren müssen, sind greedy Verfahren stets vergleichsweise effizient.

11.4 Randomisierte Algorithmen

Kurz: Mit Hilfe von Zufallszahlen werden Entscheidungen getroffen

Die Benutzung von Zufallszahlen in Algorithmen ermöglicht es, mit geringen Kosten eine Art "erwartete Ausgeglichenheit" in einem Algorithmus zu erzielen, die deterministisch nur schwer herzustellen wäre.
Setzt man Random-Verfahren bei NP-vollständigen Problemstellungen ein, so kann man zwar Laufzeiten erreichen, die besser als exponentiell sind, man kann aber nicht sicher sein, dass der Algorithmus (wenn überhaupt) eine optimale Lösung findet.

Beispiel: Quicksort-Variante mit zufälliger Auswahl des Pivotelements

11.5 Warshalls Algorithmus

S.339

Kurz: Kürzeste Entfernung zwischen je 2 Knoten
Aufwand: $O(n^3)$
Gegeben sei ein gerichteter Graph G = (V, E) mit V = 0, ..., n-1, n . Gefragt ist für alle Paare von Knoten (i, j), ob es in G einen Weg von i nach j gibt.
Der Warshall-Algorithmus berechnet als Ergebnis einen Graphen G+ = (V, E+), der genau dann eine Kante (i, j) enthält, **wenn es in G einen Weg von i nach j gibt**. Der Graph G+ heisst transitive Hülle von G, da seine Kantenrelation E+ die kleinste transitive Relation ist, die E umfasst.

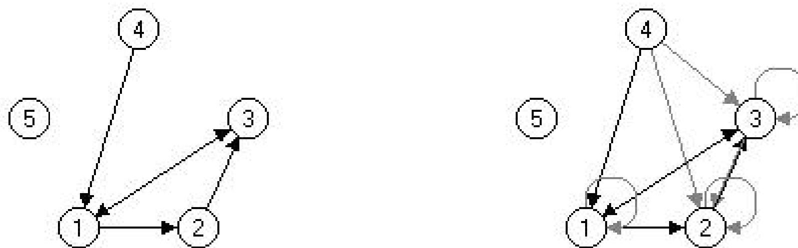


Abbildung 2: Graph G und transitive Hülle G+

```
boolean [][] A  = {...};  int dim = A.length;

void WarshallAlg(){
    int Max = dim;
    for(int y=0 ; y<Max ; y++)
```

```
    for(int x=0; x<Max; x++)
        if (A[x][y]) for (int z=0; z<Max;z++)
            if(A[y][z])A[x][z] = true;
}
```

11.6 Dijkstra's Algorithmus

Kurz: Kürzester Weg zwischen genau zwei Knoten
Aufwand: $O(n^2)$

11.7 Travelling Salesman Problem

Kurz: Kürzeste Zyklen, in welchen alle Knoten vorkommen
Aufwand: $O(2^n)$

12 Mengen und Hash-Verfahren

S.262

12.1 Mengen

12.1.1 Symbole

S.263

- $\in$ ist Element von (contains)
- $\subseteq$ ist Teilmenge von oder gleich
- $\subset$ ist Teilmenge von
- $\cap$ geschnitten mit
- $\cup$ vereinigt mit (insert)
- $\setminus$ ohne (delete)

12.2 Hashing

S.272

Eine Hashtable ist eine sehr schnelle Datenstruktur. Eine Hashtable ist ein assoziativer Speicher, der Schlüssel mit Werten verknüpft. Diese Datenstruktur ist vergleichbar mit einem Wörterbuch oder Nachschlagewerk.

Die Hashtabelle arbeitet mit Schlüssel-Werte-Paaren. Aus dem Schlüssel wird nach einer Funktion - der so genannten **Hashfunktion** - ein **Hashcode** berechnet.

12.3 Sondieren

S.279

12.3.1 Lineares Sondieren

$pos_0 = h(k)$
$pos_i = (pos_0 + i) \ mod \ n$
Reihenfolge des Ausweichsplatzes (entweder positiv oder negativ):
1, 2, 3, 4, 5, ... oder -1, -2, -3, -4, -5, ...

12.3.2 Quadratisches Sondieren

$pos_0 = h(k)$
$pos_i = (pos_0 + i^2) \ mod \ n$
Reihenfolge des Ausweichsplatzes (gemäss P. Troxler):
+1, -1, -4, +4, +9, -9, -16, +16, ...

12.3.3 Doppeltes Hashing

$pos_0 = h(k)$
$pos_i = (pos_{i-1} + i \cdot h2(k)) \ mod \ n$

12.4 Belegungsfaktor

$$\text{Belegungsfaktor} = \frac{\text{Anzahl Elemente}}{\text{Anzahl Behälter}}$$

12.5 Kollisionen

Zeichen	*Beschreibung*	*Einheit*
m	Anzahl Behälter	
i	i-tes Element, wird eingefügt	

$$\text{Wahrscheinlichkeit, dass keine Kollision eintritt} = \frac{m + i - 1}{m}$$

13 Kompression

13.1 Entropie

Definition: Mass für Unordnung

$$\text{Entropie} = -\sum_i p_i \cdot \text{lb}(p_i) = -\sum_i p_i \cdot \frac{\ln(p_i)}{\ln(2)}$$

$$\text{Anzahl Bits für Huffman Komprimierung} = \text{Entropie} \cdot \text{Anzahl Buchstaben}$$

Beispiel

ANANAS:

Anzahl Buchstaben: 6

3 A's: $p_1 = \frac{3}{6} = 0.5$

2 N's: $p_2 = \frac{2}{6} = 0.333$

1 S: $p_2 = \frac{1}{6} = 0.167$

Entropie = 1.459

13.2 Huffman

13.2.1 Idee

In einem deutschen oder englischen Text kommt der Buchstabe e sehr viel häufiger vor als beispielsweise der Buchstabe q. Um den Text mit möglichst wenigen Bits zu codieren, liegt die Idee nahe, häufig vorkommende Zeichen durch möglichst kurze Codewörter zu codieren. Dieselbe Idee wurde bereits beim Morse-Code verwirklicht, in welchem das Zeichen e durch ein Codewort der Länge 1, nämlich einen , das Zeichen q dagegen durch ein Codewort der Länge 4, nämlich - - - codiert wird.
Beim Huffman-Code ist jedes Codewort eindeutig bestimmt. Es gilt der folgende Satz: Wenn kein Codewort Anfangswort eines anderen Codewortes ist, dann ist jede codierte Zeichenreihe eindeutig dekodierbar.

13.2.2 Konstruktion des Huffman-Codes

1. Zu jedem Buchstaben wird die Anzahl des Vorkommens im Text gezählt (absolute Häufigkeit)

2. Es wird ein Binär-Baum nach dem folgenden Muster erzeugt:
   - Die jeweils am wenigsten vorkommenden Buchstaben werden zusammengenommen.
   - Ihre Summe wird addiert und in den dazugehörenden "Parent-Node" hineingeschrieben.

3. Wurde Schritt 2 korrekt ausgeführt, erscheint im ”Root-Node” die Summe aller im Text vorkommenden Buchstaben
4. Nun werden die linksgehenden Kanten mit ”0”, die Rechtsgehenden mit ”1” beziffert.
5. Für jeden Buchstaben resultiert nun ein binärer, eindeutiger Code

Hinweis: Immer Prefix-Sicher (kein Code startet mit einem anderen verwendeten Code)

13.2.3 Beispiel

Beispiel eines Huffman-Algorithmus’ für den Satz: ”IM WESTEN NICHTS NEUES”

Die Codierung ergibt:
1010010011001010011110110010011010010100110011101011111110100000111100111

Das Resultat sieht so aus (Berechnung auf nächsten Seite):

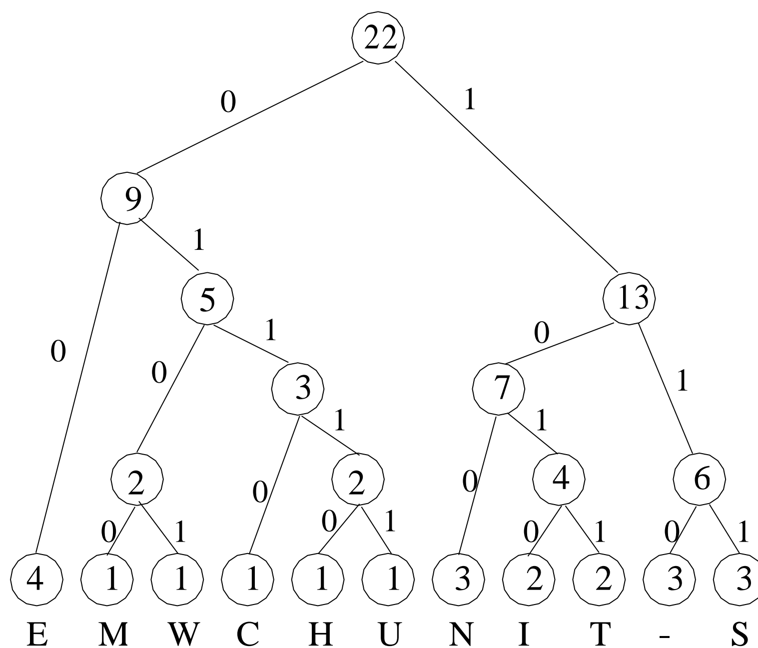


Abbildung 3: Resultat des Huffman Algorithmus-Beispiels

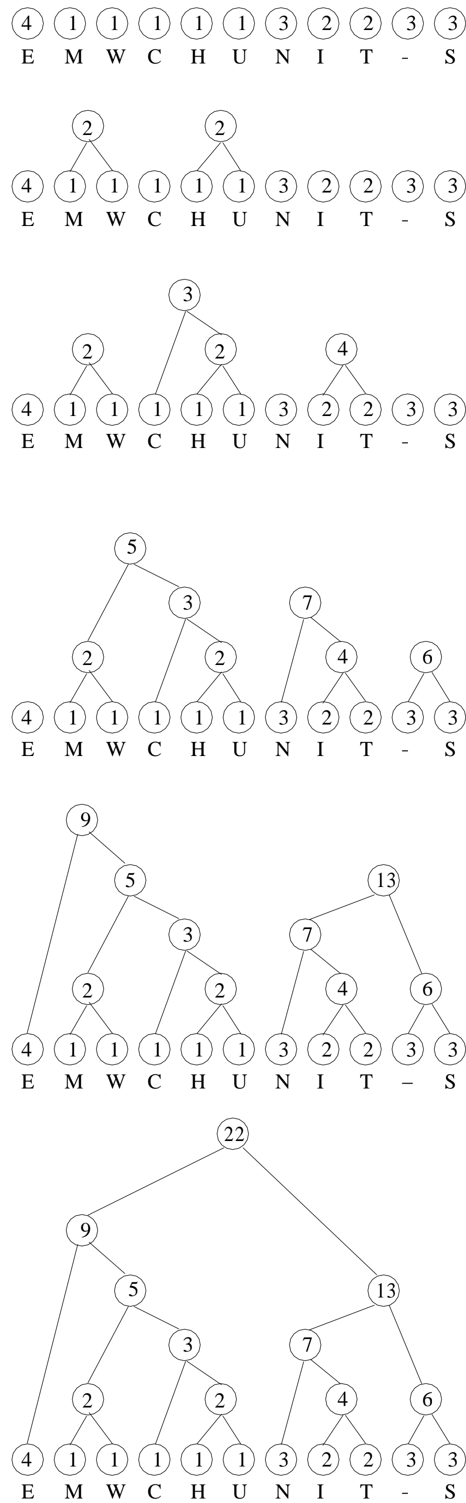


Abbildung 4: Beispiel eines Huffman Algorithmus

13.3 RLE (Run Length Encoding)

Idee: Häufiges Vorkommen des gleichen Symbols hintereinander in den Daten. Es wird immer die Anzahl und das Symbol gespeichert.

Einsatz: Fax, Bildkomprimierung

Hinweis: Für Bit-Arrays (z.B.: s/w Grafiken) genügt es nur die Anzahl zu speichern, da das Symbol immer zwischen 0 und 1 alterniert. Beispiel: 010001000000111 kann komprimiert werden als: 01 01 11 01 11 00 11 11.

14 Rekursion

Ein Objekt heisst rekursiv, wenn es sich selbst als Teil enthält oder mit Hilfe von sich selbst definiert ist. Um rekursive Routinen zu schreiben, muss immer eine Rekursionsverankerung vorhanden sein.

14.1 Abstrakte Formulierung

Abstrakte mathematische Definition der rekursiven Form (e. g. Potenzfunktion)

$$a^b = \text{pow}(a,b) = \begin{cases} 1, & falls \quad b == 0 \\ a \cdot a^{b-1} & (sonst) \end{cases}$$

14.2 Codebeispiel

14.2.1 Iterativ, direkt/indirekt Rekursiv

iterativ	direkt rekursiv	indirekt rekursiv

iterativ:

```
void hinUndZurueck(){
    int anzahl=0;
    while (vorn_frei()){
        vor();
        anzahl++;
    }

    links_um();
    links_um();
    while(anzahl>0){
        vor();
        anzahl--;
    }
}
```

direkt rekursiv:

```
void hinundZurueckR(){
    if(vorn_frei()){
        vor();
        hinundzurueckR();
        vor();
    }
    else{
        kehrt();
    }
}

void kehrt(){
    links_um();
    links_um();
}
```

indirekt rekursiv:

```
void hinundZurueckR(){
    if(vorn_frei()){
        laufe();
    }
    else{
        links_um();
        links_um();
    }
}

void laufe(){
    vor();
    hinundZurueckR();
    vor();
}
```

14.2.2 Mathematische Beispiele

Fakultaet

```
int fac(int n){
  if (n<=0)
    return 1;
  else
    return n * fac(n-1);
}
```

GGT

```
int ggt(int a, int b) {
  if (b==0)
    return a;
  else
    return ggt(b, a%b);
}
```

Potenzieren

```
int pow(int a, int b) {
  if(b==0)
    return 1;
  else
    return a*pow(a,b-1);
}
```

Fibonacci

```
int fib(int n){
  if (n<=2)
    return 1;
  else
    return fib(n-1) + fib(n
      -2);
}
```

McCarthy

```
int mc(int n){
  if(n>100)
    return n-100;
  else
    return mc(mc(n+11));
}
```

Ackermann

```
int ack(int n, int m){
  if (n==0)
    return m+1;
  else if (m==0)
    return ack(n-1,1);
  else
    return ack(n-1, ack(n,m
      -1));
}
```

14.3 Vorteile gegenüber Iterativen Verfahren

- kürzere Formulierung
- leichter Verständlich (?)
- Einsparung von Variablen
- teilweise sehr effizient

14.4 Nachteile gegenüber Iterativen Verfahren

- weniger effizientes Laufzeitverhalten (Funktions-Overhead)
- Verständnisprobleme (v.a. Anfänger)

14.5 Backtracking

14.5.1 Definition

Bezeichnung für ein Lösungsverfahren, bei dem man versucht, eine Teillösung eines Problems systematisch zu einer Gesamtlösung auszubauen. Falls in einem gewissen Stadium ein weiterer Ausbau einer vorliegenden Lösung nicht mehr möglich ist (Sackgasse), werden einer oder mehrere der letzten Teilschritte rückgängig gemacht. Die dann erhaltene reduzierte Teillösung versucht man auf einem anderen Weg wieder auszubauen. Das Zurücknehmen von Schritten und erneute Vorangehen wird solange wiederholt, bis eine Lösung des vorliegenden Problems gefunden ist oder bis man erkennt, da das Problem keine Lösung besitzt. Die Möglichkeit in Sackgassen zu laufen und aus ihnen wieder herauszufinden, zeichnet das Backtracking-Verfahren aus.

14.5.2 Code Beispiel

Siehe Skript S.379

14.6 Direkte Rekursion vs. Endrekursion (Tail Recursion)

Frage: Wie weit sind Rekursion und Iteration gleichwertig?
Direkte rekursive Implementierung der Summenfunktion addiert bei der Rückkehr aus der Rekursion:

sum(4)
= 4 + sum(3)
= 4 + 3 + sum(2)
= 4 + 3 + 2 + sum(1)
= 4 + 3+ 2 + 1 + sum(0)
= 4 + 3 + 2 + 1 + 0
= 4 + 3 + 2 + 1
= 4 + 3 + 3
= 4 + 6
= 10

Alternative: Ergebnis beim Abstieg in Rekursion akkumulieren (Hilfsmittel: zweiter Parameter) **Endrekursion**

sum(4, 0)
= sum(3, 4 + 0)
= sum(2, 3 + 4)
= sum(1, 2 + 7)
= sum(0, 1 +9)
= 10

→ Ergebnis ist am tiefsten Punkt der Rekursion bekannt, bei der Rckkehr gibts **nichts mehr** zu berechnen!

14.7 Begriffe

- **Backtracking** Der Algorithmus verfolgt eine potentielle Lösung so lange, bis er in eine Sackgasse gerät. Dann wird auf vorherige Schritte zurückgegriffen von wo aus der Algorithmus erneut läuft. Viele unmögliche Lösungen müssen auf diese Weise nicht geprüft werden.
- **Bruteforce** Erschöpfende Ausprobierung aller abzählbaren möglichen Zustände.
- **Inkarnation** Beim Ausführen einer Methode f entsteht eine Inkarnation von f. Zur Inkarnation gehören: Ausführungspunkt und Speicherplätze für methodenlokale Variablen.
  -**Aktionen beim Aufruf einer Funktion**
  - Es wird auf dem Stack ein Aktivierungselement angelegt
  - Der aktuelle Zustandes Programms wird im Aktivierungssegment gespeichert
  - Im Aktivierungssegment werden Speicherbereiche fr Parameter, lokale Variablen und temp. Daten reserviert

  -**Aktionen beim Verlassen einer Funktion**
  - Mit Hilfe der abgespeicherten Registerinhalte kann der Zustand des Programms von vor dem Funktionsaufruf wiederhergestellt werden
  - Der Funktionswert wird an geeigneter Stelle abgelegt
  - das Aktivierungssegment wird zerstört
- **Rekursion** Eine Methode heisst rekursiv, wenn mindestens zwei ⇒ Inkarnationen zur gleichen Zeit bestehen können.
  - **direkte Rekursion** Die Methode ruft sich selbst auf.
  - **indirekte Rekursion** Die zweite Inkarnation wird nicht direkt von der Methode selbst erzeugt.
- **Rekursionstiefe** Anzahl Inkarnationen einer Methode minus 1.

15 Sortieralgorithmen

15.1 Bubble Sort

S.391

Sortieren durch Vertauschen von Nachbarfeldern.

15.1.1 Aufwand

	Best Case	Average Case	Worst Case
Komplexität:	$O(N)$	$O(N^2)$	$O(N^2)$
Anz. Vergleiche:	$n \cdot \frac{n-1}{2}$		
Anz. Bewegungen:	0	$\frac{C \cdot n \cdot \frac{n-1}{2}}{2}$	$C \cdot n \cdot \frac{n-1}{2}$

C ist der Aufwand für die Anweisung im Inneren der Schleife. Zum Beispiel $C = 3$ für 3 Verschiebungen.

15.1.2 Stabilität

Je nach Implementation im allgemeinen **stabil**.

15.1.3 Vor- und Nachteile

Vorteile:
Nachteile: Viele Swaps, sehr ineffizient

15.1.4 Programmierbeispiel 1

Dies ist die trivialste Variante, um einen Bubble-Sort-Algorithmus zu schreiben. Dabei werden bei jedem Durchgang die Nachbarfeldern verglichen und falls nötig vertauscht.

```
static void bubbleSort1(char[] a)
{
    int hi = a.length - 1;
    for (int k = hi; k > 0; k--)
        for (int i = 0; i < k; i++)
            if (a[i] > a[i+1] swap(a, i, i+1);
}

private final static void swap(char[] a, int i, int k)
{
    char h = a[i];
    a[i] = a[k];
    a[k] = h;
}
```

15.1.5 Programmierbeispiel 2

Dies ist eine Optimierung des Bubble-Sort, welche den Sortiervorgang abbricht, wenn bei einem Durchgang keine Nachbarfelder vertauscht wurden.

```
static void bubbleSort2(char[] a)
{
    int hi = a.length - 1;
    for (int k = hi; k > 0; k--)
    {
        boolean test = true;
        for (int i = 0; i < k; i++)
            if (a[i] > a[i+1])
            {
                swap (a, i, i+1);
                test = false;
            }
    }
}

private final static void swap(char[] a, int i, int k)
{
    char h = a[i];
    a[i] = a[k];
    a[k] = h;
}
```

15.2 Shaker Sort

Der Shaker Sort ist eine Optimierung des Bubble-Sorts. Er wechselt nach jedem Durchgang die Richtung des Durchlaufs. Dadurch wandern stark aus der Ordnung abweichende Elemente schneller durch das Array.

S.395

15.3 Selection Sort

Sortieren durch Auswählen des jeweils grössten der verbleibenden Elemente und Anhängen an die Reihe der bereits sortierten Elemente.

15.3.1 Aufwand

Best Case	Average Case	Worst Case
$O(N)$	$O(N^2)$	$O(N^2)$

15.3.2 Stabilität

Stabil.

15.3.3 Vor- und Nachteile

Vorteil: Deutlich weniger Swaps als bei Bubble Sort
Nachteil: Aufwand auch bei vorsortiertem Array gleich hoch

15.3.4 Programmierbeispiel

Die zweite Methode **minPos** dient zur Ermittlung der Position des kleinsten Elementes im unsortierten Rest. Dadurch wird die eigentliche Sortierroutine erfreulich gekürzt.

```
static void SelectionSort(char[] A)
{
    int Hi = A.length − 1;
    for (int k = 0; k < Hi; k++)
    {
        int Min = minPos(A, k, Hi);
        if (Min != k) Swap (A, Min, k);
    }
}

private static int minPos(char[] A, int Lo, int Hi)
{
    int Min = Lo;
    for (int i = Lo+1; 1 <= Hi; i++)
        if (A[i] < A[Min]) Min = 1;
    return Min;
}

private final static void Swap(char[] A, int i, int k)
{
    char h = A[i];
    A[i] = A[k];
    A[k] = h;
}
```

15.4 Insertion Sort

Sortieren durch Einfügen eines beliebigen (in der Regel des nächsten) Elementes aus den unsortierten Elementen an der richtigen Position in der Reihe der bereits sortierten Elemente.

15.4.1 Stabilität

Stabil. Wird für das Suchen der Einfügeposition ein Binary-Search verwendet, wird dieser Algorithmus instabil.

15.4.2 Vor- und Nachteile

Vorteile: Bessere Laufzeit, wenn besser vorsortiert. Schneller bei relativ kurzen Arrays.
Nachteil: Mehr Swaps beim Verschieben als Selection Sort.

15.4.3 Programmierbeispiel

```
static void InsertionSort(char[] A)
{
    int Hi = A.length - 1;
    for (int k = 1; k <= Hi; k++)
        if (A[k] < A[k-1])
        {
            char x = A[k];
            int i;
            for (i = k; ((i>0) && (A[i-1] > x)); i--)
                A[i] = A[i-1];
            A[i] = x;
        }
}
```

15.5 Shell Sort

Variante von Insertion Sort. Arbeitet in mehreren Durchgängen (Vorsortieren). Letzter Durchgang stimmt mit Insertion Sort überein.

Idee: Parametrisierung von InsertionSort

15.5.1 Aufwand

Bewiesen	Experimentell
$O(N^{1.5})$	$O(N^{1.25})$

15.5.2 Vor- und Nachteile

Vorteil: Ist für nicht-vorsortierte Daten besser als Insertion Sort.

S.404

15.6 Comb Sort

Verbesserung von Bubble Sort.

15.6.1 Vor- und Nachteile

Vorteil: Schneller als Bubble Sort
Nachteil: Langsamer als Shell Sort

S.405

15.7 Heap Sort

Die Daten werden in einem Heap (Baum in dem der Schlüssel eines Knotens extremer (grösser oder kleiner) ist als der des Parent Knotens) angeordnet. Heap Sort baut den Baum von oben nach unten ab.

15.7.1 Aufwand

$O(N \log N)$

S.405

15.8 Rekursive Sortieralgorithmen

Prinzip: Teile und Hersche

Beispiele: Quick Sort, Merge Sort, Distribution Sort

Hinweis: Quick Sort ist instabil

16 Zeitkomplexität

16.1 Wichtigste Aussagen zur Zeitkomplexität

S.355

- Die Zeit- und Speicherplatzkomplexität eines Algorithmus hängt von der Grösse der Eingabe ab. Im schlechtesten Fall bildet man die Berechnungschritte für Eingaben der gleichen Grösse. Analog nimmt man im mittleren Fall den Durchschnitt.
- Speicher- und Zeitkomplexität werden nach ihrer Grössenordnung, der O-Notation, klassifiziert. Diese hängt im wesentlichen von der Anzahl der nötigen Schleifendurchläufe ab. Geschachtelte Schleifen erhöhen die Komplexität, im Gegensatz zu hintereinander ausgeführten Schleifen.
- Polynominal berechenbare Algorithmen heissen auch praktisch berechenbar (obwohl schon die Komplexität von n^3 häufig Probleme bereitet). Exponentielle Algorithmen sind nicht praktische berechenbar.

17 Events

Idee: Eine zentrale Schleife, die fortwährend **Benutzerinteraktion** entgegennimmt und die passenden **Aktionen auslöst** (z.B. Methodenaufrufe).

17.1 Begriffe, Anwendungen, Nachteile

Ereignis: Events

- Tastatureingabe, Mausklick, etc.
- wird durch ein Objekt repräsentiert: das Event-Objekt

Ereignisquelle:

- Objekt, welches das Ereignis erzeugt: Scrollbar, Button Checkbox etc.

Ereignisverarbeiter:EventListener

- Methode, welche das Ereignis verarbeitet

Anwendung

- Steuerung von Haushaltgeräten
- Graphische Benutzeroberflächen

Anwendung

- Jedes Ereignis muss an alle Bearbeiter weitergereicht werden
wenn sehr viele Bearbeiter: ineffizient

17.2 Programmbeispiel

```
public class FirstEvent extends Applet implements AdjustmentListener
{
    private Scrollbar slider;

    private int sliderValue = 0;

    public void init()
    {
        // Eine Instanz der Klasse Scrollbar erzeugen
        slider = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,100);
        // hinzufuegen der Scrollbar-Instanz zum Applet
        add(slider);
        // slider beim AdjustmentListener anmelden
        slidder.addAdjustmentListener(this);
    }

    public void paint(Graphics g)
```

```
{
    g.drawString("Value: " + sliderValue,100,100);
}

public void adjustmentValueChanged(AdjustmentEvent e)
{
    sliderValue = slider.getValue();
    repaint();
}
```

17.3 Button

Klassendefinition

`public class ... extends Applet implements ActionListener`
`{`
`    private Button but;`
`    ...`
`}`

Initialisierung in der Methode init

`but = new Button("Start");`
`this.add(but);`
`but.addActionListener(this);`

Ereignis-Methode

`public void actionPerformed(ActionEvent e)`
`{`
`    if (e.getSource() == but)`
`    {`
`        ...`
`        repaint();`
`    }`
`    ...`
`}`

17.4 Label

Klassendefinition

`public class ... extends Applet`
`{`
`    ...`
`}`

Initialisierung in der Methode init
`beschriftung = new Label("rot");` `this.add(beschriftung);`
Ereignis-Methode
Diese Objekte lösen in der Regel kein Ereignis aus.

17.5 MouseListener

Klassendefinition

public class ... **extends** Applet **implements** MouseListener
{
...
}

Initialisierung in der Methode init
this.addMouseListener(**this**);
Ereignis-Methode
public void mousePressed(MouseEvent e) {...} **public void** mouseReleased(MouseEvent e) {...} **public void** mouseClicked(MouseEvent e) {...} **public void** mouseEntered(MouseEvent e) {...} **public void** mouseExited(MouseEvent e) {...} **Es müssen stets alle Methoden deklariert sein, auch wenn nur eine benutzt wird!**

17.6 MouseMotionListener

Klassendefinition

public class ... **extends** Applet **implements** MouseMotionListener
{
...
}

Initialisierung in der Methode init
this.addMouseMotionListener(**this**);
Ereignis-Methode
public void mouseMoved(MouseEvent e) {...} **public void** mouseDragged(MouseEvent e) {...} **Es müssen stets beide Methoden deklariert sein, auch wenn nur eine benutzt wird!**

18 Exceptions

Ausnahmen (engl.:exception) sind Sonderfälle, die der Programmierer oder das System versuchen abzufangen (engl.:to catch).

Das Grundprinzip des Exception-Mechanismus in Java kann wie folgt beschrieben werden:

- Ein Laufzeitfehler oder eine vom Entwickler gewollte Bedingung löst eine Ausnahme aus.
- Diese kann nun entweder von dem Programmteil, in dem sie ausgelöst wurde, behandelt werden, oder sie kann weitergegeben werden.
- Wird die Ausnahme weitergegeben, so hat der Empfänger der Ausnahme erneut die Möglichkeit, sie entweder zu behandeln oder selbst weiterzugeben.
- Wird die Ausnahme von keinem Programmteil behandelt, so führt sie zum Abbruch des Programms und zur Ausgabe einer Fehlermeldung.

18.1 throw

- Anstatt eines "normalen" Rückgabewertes kann eine Ausnahme (Exception) zurückgegeben werden.
- Exception-Objekte werden nicht mit return retourniert, sondern mit **throw** dem Aufrufer quasi "angeworfen".

18.1.1 Programmbeispiel

```
public double mySqrt(double d) throws Exception
{
    if (d < 0)
        throw new Exception("Argument negativ");
    else
        return Math.sqrt(d);
}
```

18.2 try und catch

Das Behandeln von Ausnahmen erfolt mit Hilfe der try-catch- Anweisung.

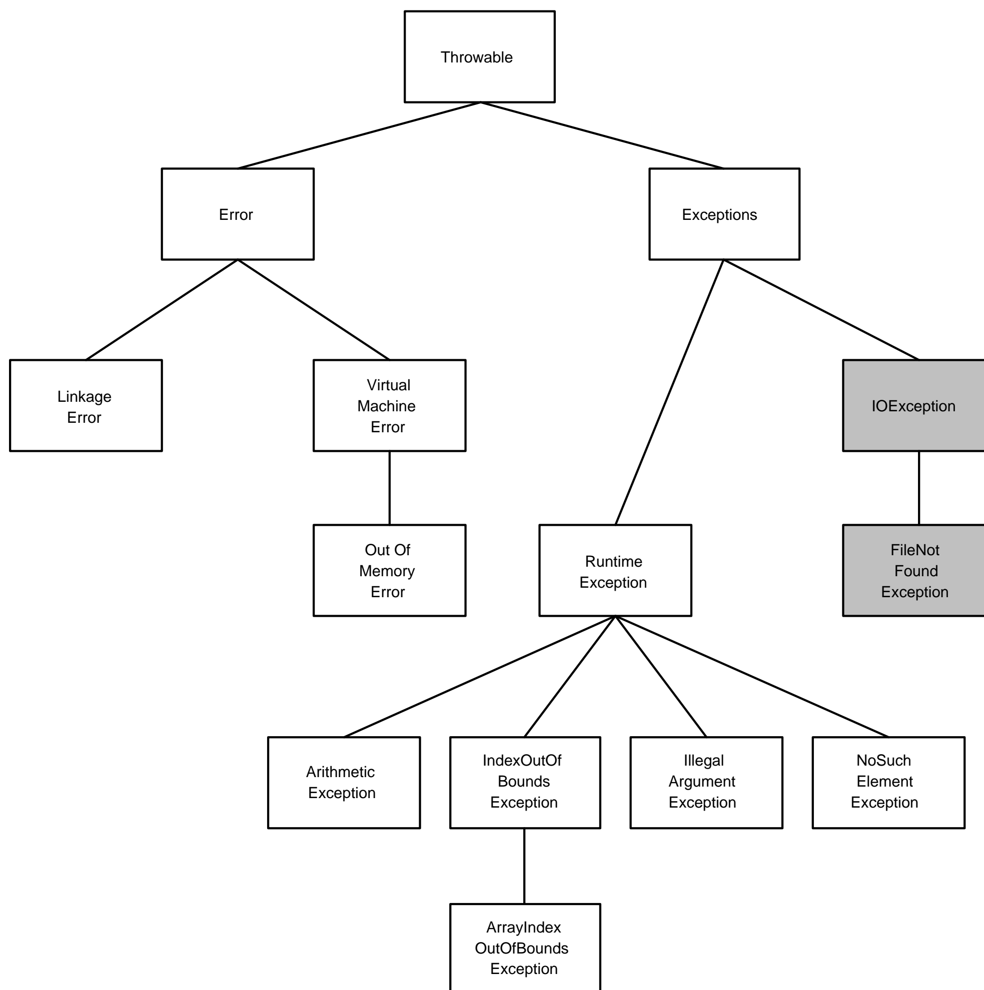
Der **try-Block** enthält dabei eine oder mehrere Anweisungen, bei deren Ausführung ein Fehler des Typs Ausnahmetyp auftreten kann. In diesem Fall wird die normale Programmausführung unterbrochen und der Programmablauft fährt mit der ersten Anweisung nach der **catch-Klausel** fort, die den passenden Ausnahmetyp deklariert hat. Hier kann nun Code untergebracht werden, der eine angemessene Reaktion auf den Fehler realisiert.

18.2.1 Programmbeispiel

```
String resultat;
try
{
    double d = mySqrt(p);
    resultat = "Wurzel" + d;
}
catch (Exception ex)
{
    resultat = ex.getMessage();
}
```

18.3 Exception und Error

- Unterklassen von **Exception** sollten verwendet werden, wenn es sich um echte Ausnahmen handelt.
- Unterklassen von **Error** sollten verwendet werden, wenn es sich um Fehler handelt, die einen Abbruch des Programms erforderlich machen.



- Die grau hinterlegten Klassen werden zu den **checked Exceptions** gezählt.
- Der Compiler kontrolliert, dass diese Exceptions behandelt werden und meldet einen Fehler, wenn dies nicht der Fall ist.

18.4 try-catch-finally

Mit Hilfe der **finally**-Klausel, die als letzter Bestandteil einer try-catch-Anweisung verwendet werden darf, kann ein Programmfragment definiert werden, das immer dann ausgeführt wird, wenn die zugehörige try-Klausel betreten wurde. Dabei spielt es keine Rolle, welches Ereignis dafür verantwortlich war, dass die try-Klausel verlassen wurde.
Die **finally**-Klausel wird insbesondere dann ausgeführt, wenn der try-Block durch eine der folgenden Anweisungen verlassen wurde:

- wenn das normale Ende des try-Blocks erreicht wurde
- wenn eine Ausnahme aufgetreten ist, die durch eine catch-Klausel behandelt wurde
- wenn eine Ausnahme aufgetreten ist, die nicht durch eine catch-Klausel behandelt wurde
- wenn der try-Block durch eine der Sprunganweisungen break, continue oder return verlassen werden soll

Die **finally**-Klausel ist also der ideale Ort, um Aufräumarbeiten durchzuführen. Hier können beispielsweise Dateien geschlossen oder Ressourcen freigegeben werden.

18.5 Die Klasse RuntimeException

- Alle Ausnahmen, die Unterklassen von Exception sind, müssen behandelt oder weitergereicht werden (sogenannte catch-or-throw Regel)
- Diese Regel hat eine Ausnahme: Alle Ausnahmen, die aus der Unterklasse **Runtime-Exception** (Laufzeit-Ausnahme) abgeleitet sind, können, müssen aber nicht behandelt werden.
- Die Klasse **RuntimeException** und ihre Unterklassen sind zur Behandlung häufig vorkommender Laufzeitfehler gedacht.
- Es ist eine kleine Inkonsequenz von Java diese nicht von vornherein als Unterklasse von Error zu definieren.

19 File Handling

19.1 Arten von Dateien

- Anwendungsdaten (Text, Bilder, Sound, Quellcode, ...)
- Ausführbare Dateien (Executables, Programme)
- System-Dateien und Verzeichnisse

19.2 Zugriffsarten

- **sequentiell**, wie bei einem Tonband:
  Stream-Access, die häufigste Art auf Dateien zuzugreifen
- **wahlfrei**, wie auf die Tracks einer CD:
  Random-Access, aufwendiger, z.B. Datenbanken

19.3 File-Objekt

Um Dateien (auch Verzeichnisse) zu erstellen, löschen, ordnen, umbenennen,kopieren... oder um Informationen über Dateien zu erfragen, wird in Java ein **File-Objekt** aus der Klasse java.io.File benutzt.

Ein **File-Objekt** kann auf verschiedene Arten erzeugt werden:

- File EingabeDatei=new File(String path)
  Erzeugt ein File-Objekt mit komplettem Pfadnamen
- File EingabeDatei=new File (String path, String name)
  Pfadname und Dateiname sind getrennt.
- File EingabeDatei=new File(File dir, String name)
  Der Pfad ist mit einem anderen File-Objekt verbunden.

19.4 Methoden der Klasse File

Name	**Beschreibung**
boolean exists()	true, wenn das File-Objekt existiert.
boolean isDirectory()	Gibt true zurück, wenn es sich um ein Verzeichnis handelt.
boolean isFile()	true, wenn es sich um eine "normale" Datei handelt (kein Verzeichnis und keine Datei, die vom zu Grunde liegenden Betriebsystem als besonders markiert wird; Blockdateien, Links unter UNIX).
boolean canRead()	true, wenn wir lesend zugreifen dürfen.
boolean canWrite()	true, wenn wir schreibend zugreifen dürfen.
long length()	Gibt die Länge in Bytes zurück oder 0, wenn die Datei nicht existiert.
String getName()	Gibt Dateinamen zurück.
String getParent()	Gibt Pfadnamen des Vorgängers zurück.
String getPath()	Gibt Pfadnamen zurück.
boolean isAbsolute()	true, wenn der Pfad in der systemabhängigen Notation absolut ist.
String getAbsolutePath()	Liefert den absoluten Pfad. Ist das Objekt kein absoluter Pfadname, so wird ein String aus aktuellem Verzeichnis, Separaturzeichen und Dateinamen des Objekts verknüpft.
boolean mkdir()	Legt das Verzeichnis an.
boolean mkdirs()	Legt das Verzeichnis inklusive notwendiger Unterverzeichnisse an.
String[ ] list()	Retourniert Liste aller Dateien eines Verzeichnisses.
boolean renameTo(File dest)	Benennt File um.
boolean delete()	Löscht Datei;liefert true, falls File vorhanden war.

19.5 Random-Access-Datei

19.5.1 Öffnen

public RandomAccessFile(String name, String mode)

- Öffnet die Datei. Löst eine FileNotFoundException aus, falls die Datei nicht geöffnet werden kann.
- Mode bstimmt, ob aus der Datei gelesen oder in die Datei geschrieben wird. Es sind "r" und "rw" erlaubt. Ist der Modus falsch gesetzt, zeicht eine IllegalArgumentException dies an.
- Vor dem Öffnen der Datei wird geprüft, ob die erforderlichen Zugriffsrechte für die Datei vorhanden sind. Eine ausgelöste SecurityException zeigt fehlende Schreib- oder Leserechte an.

19.5.2 Schliessen

Eine geöffnete Datei muss mit **close()** wieder geschlossen werden.

19.5.3 Lesen

int read()
Liest genau ein Byte und liefert es als int zurück.

int read(byte[ ] b)
Liest b.length()-Bytes und speichert sie im Feld b.

int read(byte[ ] b, int off, int len)
Liest len Bytes aus der Datei und schreibt sie in das Feld b ab der Position off. Konnten mehr als ein Byte gelesen werden, aber weniger als len, dann wird die gelesene Grösse als Rückgabewert zurückgegeben.

boolean readBoolean()
Liest einen boolean-Wert.

short readShort()
Liest ein short.

char readChar()
Liest einen char-Wert.

double readDouble()
Liest ein double.

float readFloat() Liest ein float.

String readLine()
Liest eine Textzeile.

19.5.4 Dateizeiger

long getFilePointer()
Liefert die momentane Position des Dateizeigers. Das erste Byte steht an der Stelle Null. Da der Rückgabewert long ist und nicht BigInteger, ist die maximale Dateilänge auf 2GB begrenzt.

long length()
Liefert die Grösse der Datei in Bytes.

void seek(long pos)
Setzt die Position des Dateizeiger auf pos. Diese Angabe ist absolut und kann daher nicht negativ sein. Falls doch, wird eine Ausnahme ausgelöst.

int skipBytes(int n)
Mit skipBytes() kann im Gegensatz zu seek() relativ positioniert werden. n ist die Anzahl, um die der Dateizeiger bewegt wird. Ein negativer Wert setzt den Zeiger nach vorne. Falls versucht wird, den Zeiger vor die Datei zu setzen, wird eine IOException ausgelöst.

void setLength(long newLength)
Setzt die Grösse der Datei auf newLength.

19.5.5 Schreiben

Zu jeder read***()-Methode existiert eine entsprechende write***()-Methode (Siehe Lesen).

19.6 Streams

19.6.1 Lesezugriff

```
BufferedReader ipf;
String line;
try {
    ipf = new BufferedReader(new FileReader("Test_in.txt"));
    while ((line = ipf.readLine()) != null)
        System.out.println(line);
    ipf.close()
    }
catch (Exception e)
{
```

```
    System.out.println(e.toString());
}
```

19.6.2 Schreibzugriff

```
BufferedWriter opf;
try {
    opf = new BufferedWriter(new FileWriter("Test_out.txt"));
    opf.write("Hello␣");
    opf.write("<<World>>",2,5);
    opf.newLine();
    opf.write(65);
    opf.write("bacus");
    opf.close();
    }
catch (Exception e)
{
    System.out.println(e.toString());
}
```

20 Threads

20.1 Begriffe

- **Gegenseitiger Ausschluss** Jedes Betriebsmittel eines Systems ist zu einem Zeitpunkt entweder nur von genau einem einzigen Prozess exklusiv belegt oder aber frei.
- **Deadlock** Ein Deadlock ist eine permanente, gegenseitige Blockierung von einer Menge von Prozessen.
- **Vier Deadlock Bedingungen** Wechselseitiger Ausschluss, Belegungs- und Wartebedingung, Nicht-Unterbrechbarkeit, Zyklische Wartebedingung.
- **Scheduler** Zentrale Einheit, welche den einzelnen Prozessen Zeitintervalle zuordnet.

20.2 Beeinflussung von Threads

Aktueller Thread kann:

- **sich selbst ...**
  - schlafen legen: **sleep()**
  - CPU entziehen: **yield()**
  - anhalten: **suspend()**
  - Priorität ändern: **setPriority()**
  - beenden: **stop()**
- **andere Threads ...**
  - starten: **t.start()**
  - anhalten: **t.suspend()**
  - fortsetzen: **t.resume()**
  - Priorität ändern: **t.setPriority()**
  - beenden: **t.stop()**