

Informatik I

Carsten Damm

Wintersemester 18/19

Carsten Damm
Institut für Informatik
Georg-August-Universität Göttingen
Informatik I (Stand 15. Januar 2019)

Vorbemerkungen

Informatik I ist eine Einführung in Konzepte und Formalismen der Informatik, die wir anhand der Programmierung mit der Programmiersprache Java studieren. Programmier- oder Java-Kenntnisse werden nicht vorausgesetzt, sondern im Kursverlauf systematisch eingeführt. Das Skript enthält alle wesentlichen Inhalte der Vorlesung, aber mitunter wird es Abweichungen geben (z.B. andere/zusätzliche Beispiele), die dann anders zur Verfügung gestellt werden.

Im PDF enthaltene Links sind in der Druckfassung möglicherweise nicht sichtbar.

Literaturhinweise

Wir folgen vor allem diesem Buch

Sedgewick/Wayne *Einführung in die Programmierung - mit Java*
Ausschnitte finden Sie unter <http://introcs.cs.princeton.edu/>

Einige Ergänzungen stammen aus anderen Büchern und weiteren Quellen, z.B.

Gumm/Sommer *Einführung in die Informatik*
deckt Informatik I+II begrifflich weitgehend ab, viele C- und Java-Beispiele

Pomberger/Dobler *Algorithmen und Datenstrukturen*
sehr sorgfältig, gut verständlich, keine spezielle Programmiersprache

Cormen u.a. *Algorithmen - eine Einführung*
umfassendes, theoretisch orientiertes Standardwerk zu Algorithmen und ihrer Analyse, keine spezielle Programmiersprache

Inhaltsübersicht

1. Einführung
2. Grundlegende Datentypen
3. Funktionen und Module
4. Strukturierte Daten
5. Objektorientierte Programmierung
6. Algorithmen
7. Theoretische Grundlagen
(zum Selbststudium bzw. Nachschlagen)

Inhaltsverzeichnis

Vorbemerkungen	i
Literaturhinweise	i
Inhaltsübersicht	i
1 Einführung	1
1.1 Was ist Informatik?	1
1.1.1 Zur Einordnung	1
1.1.2 Rechnersysteme	3
1.1.3 Problemlösung in der Informatik	5
1.2 Problem, Algorithmus, Programm, Prozess	10
1.2.1 Probleme und ihre Spezifikation	10
1.2.2 Algorithmen	11
1.2.3 Prozessoren	14
1.2.4 Erste Java-Beispiele	17
1.3 Java-Kurzeinführung	22
2 Grundlegende Daten und Algorithmen	29
2.1 Daten und Datentypen	29
2.1.1 Datentypen	30
2.1.2 Datentypen für Zahlen	31
2.1.3 Datentypen für Zeichen und Wahrheitswerte	37
2.2 Programmieren mit integrierten Datentypen	42
2.2.1 Rechnen mit Wahrheitswerten	42
2.2.2 Verarbeitung von Zeichen und Zeichenketten	43
2.2.3 Arithmetik	45
2.2.4 Einfache numerische Berechnungen	49
2.2.5 Typanpassungen	52
2.3 Felder	55
2.3.1 Eindimensionale Felder	55
2.3.2 Nützliche Algorithmen auf Feldern	57
2.3.3 Einige Besonderheiten von Feldern	59
2.3.4 Mehrdimensionale Felder	61
3 Funktionen und Module	65
3.1 Programmbibliotheken	65
3.1.1 Standardbibliotheken	65
3.1.2 Nutzerbibliotheken	67
3.1.3 Nachträge zur Ein- und Ausgabe	69
3.1.4 Vektorgrafik mit der Klasse StdDraw	70

3.1.5	Audio mit der Klasse <code>StdAudio</code>	73
3.2	Programmiersprachen-Nomenklatur	75
3.3	Nachträge zur strukturierten Programmierung	80
3.3.1	Variablen und Ausdrücke	80
3.3.2	Die Kontrollanweisungen von Java	84
3.4	Prozedurale Programmierung	86
3.4.1	Statische Methoden	86
3.4.2	Felder als Formalparameter	94
3.4.3	Der Laufzeitstapel	95
3.4.4	Rekursion	97
3.5	Modulare Programmierung	104
3.5.1	Ein Beispiel	104
3.5.2	Benutzerdefinierte Bibliotheken	106
4	Strukturierte Daten	111
4.1	Weiteres zu Feldern	111
4.1.1	Einige Besonderheiten	111
4.1.2	Details zu Feldern als Methodenargumente	113
4.1.3	Zwei einfache Sortierverfahren	114
4.2	Verbunddaten	115
4.2.1	Definition, Erzeugung und Zugriff	115
4.2.2	Ein Geometrie-Beispiel (Anfang)	119
4.2.3	Stringumwandlung	121
4.2.4	Verkettete Datenstrukturen	122
4.3	Referenzen als Ergebniswerte	125
4.3.1	Felder als Ergebniswerte	125
4.3.2	Verbundreferenzen als Ergebniswerte	126
4.4	Speicherorganisation	127
4.5	Weiteres zu Strings	129
4.5.1	Strings und <code>char</code> -Felder	129
4.5.2	Einige Methoden zur Arbeit mit Strings	131
5	Objektorientierte Programmierung	133
5.1	Klassen und Objekte	133
5.1.1	Klassen im Wandel der Paradigmen	133
5.2	Instanzmethoden	135
5.2.1	Erste Beispiele	135
5.2.2	Kapselung	137
5.2.3	Verbesserung des Geometrie-Beispiels	139
5.3	Vererbung	143
5.3.1	Ein Spielzeugbeispiel	144
5.3.2	Weitere Verbesserung des Geometrie-Beispiels	149
5.4	Abstrakte und finale Klassen	152
5.4.1	Dritte Variante des Geometrie-Beispiels	152
5.4.2	Finale Klassen	153
5.5	Abstrakte Datentypen	154
5.5.1	Abstrakter Datentyp Stack	154
5.5.2	Abstrakter Datentyp Queue	158
5.5.3	Einführung in generische Typen (Java Generics)	161
5.5.4	Programmierung mit abstrakten Datentypen	164

6 Algorithmen und Datenstrukturen	167
6.1 Performance-Betrachtungen	167
6.2 Wachstumsordnungen	173
6.3 Laufzeitvorhersagen	176
6.4 Ergänzungen	178
6.5 Sortieren und Suchen	179
6.5.1 Elementare iterative Sortierverfahren	179
6.5.2 Teile- und Herrsche-Algorithmen für das Sortieren	182
6.5.3 Vergleichsbasierte vs. allgemeinere Sortierverfahren	187
6.5.4 Nachtrag zur O-Notation	190
6.5.5 Suchverfahren	191
6.5.6 Binäre Suchbäume	193

Kapitel 1

Einführung

1.1 Was ist Informatik?

1.1.1 Zur Einordnung

Begriffsbestimmung

1.1-1

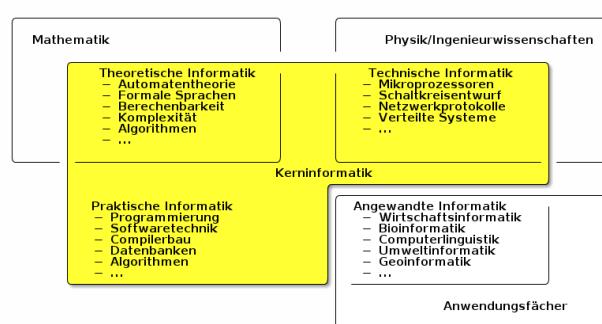
Informatik

→ Info1-Sammlung (II-ID: zlc7wut0m2i0)

- Wissenschaft von der *automatisierten* Verarbeitung von *Informationen*
- Konzepte und Formalismen der Informatik sind *Technologie-unabhängig*

Computer Science is no more about computers than
astronomy is about telescopes. *Edsger W. Dijkstra*

Teilgebiete



Angewandte Informatik

das Anwenden von Informatik-Methoden in anderen Wissenschaften

Everybody should learn how to program a computer
because it teaches you how to think. *Steve Jobs*

Anfänge

1.1-2

Altsteinzeit (>20000 v.Chr.) Entwicklung des Zahlbegriffs

Antike (ca. 2000 v.Chr.) einfachste mechanische Rechenhilfsmittel, einfache Rechenverfahren (Beispiel: Euklid's Algorithmus und Heron-Verfahren, Vorläufer von **Stellenwert-** und **Gleitkommadarstellung**)

5. Jh. n. Chr. in Indien wird Rechnen mit einem besonderen Zeichen 0 (**Null**) systematisiert

9. Jh. n. Chr. Ibn Musa Al-Chwarismi (Namens-Pate für **Algorithmus**) schreibt Lehrbuch über das Lösen von Gleichungen

1202 und 1547: durch Leonardo von Pisa (Fibonacci) und Adam Ries wird das Rechnen mit der **Dezimaldarstellung** im europäischen Raum popularisiert

1623-47 Schickard, Pascal, Leibniz konstruieren verschiedene mechanische **Rechenmaschinen**

1700 Leibniz' Traktat über die **Binärdarstellung** (die aber lange vorher in China und Indien bekannt war)

1.1-3 Rechenmaschinen und Computer

1838 Charles Babbage entwirft die (nie gebaute) **Analytical Engine**, einen **programmierbaren Rechner**

1886 Hermann Hollerith entwickelt in den USA **elektromechanische** Lochkartenmaschinen für statistische Auswertungen von Volkszählungen. Aus seiner Firma ging später IBM hervor

1937 Konrad Zuse baut die Z1 als ersten programmierbaren Rechner mit **Binärarithmetik** (Mechanik arbeitet aber nicht zuverlässig, Nachfolger Z2 beruht auf Relaistechnik)

1941 Z3 ist der erste funktionsfähige **universelle Rechner** (kurz: **Computer**), Programm wird von Lochstreifen gelesen

1944 in Zusammenarbeit mit der Harvard-University und IBM entsteht die teilweise programmgesteuerte Rechenanlage MARK I: Additionszeit 1/3s, Multiplikationszeit 6s, 35t Gewicht, 16m Frontlänge

1945 Nachfolger Z4 wird in **Göttingen** in Dienst gestellt (später nach Süddeutschland verlegt, dann nach Zürich, bis ca. 1960 in Betrieb), Tastatureingabe, Lochstreifen-/Lampen-Ausgabe

1946 John P. Eckert und J. W. Mauchly: ENIAC als erster vollektronischer Rechner, 18000 Elektronenröhren, Multiplikationszeit: 3s (ähnlich wie Z4)

1946-52 Entwicklung von Computern auf Grundlage der **von Neumann-Architektur**: Einzelprozessor, Programm und Daten im gleichen Speicher

ab 1950 industrielle Computerentwicklung und -Produktion (Beispiel: 1953: IBM 650)

1952, -55 und -61 in Göttingen entwickelte Computer G1, G2 und G3

um 1957 6 Computer in Deutschland: G1, G2, PERM (München), 3x IBM 650

1957 Begriff "**Informatik**" erstmalig in der Literatur (als Titel einer Publikation von Karl Steinbuch)

Ende der 1950er Beginn industrieller Computer-Produktion in Deutschland (u.a. IBM Sindelfingen, Zuse Z11 in Hünfeld)

1.1-4 Programmiersprachen

1842 Lady Ada Augusta Lovelace

schreibt das erste **Programm** für eine Rechenmaschine (Babbages Analytical Engine): eine Folge von maschinenorientierten **Instruktionen** Beispiel: $V_8 \leftarrow V_2 \times V_4; V_9 \leftarrow V_5 \times V_1; \dots$

1942 Zuses Plankalkül war ähnlich aufgebaut, Verwendung auf Z3 erfordert Übersetzung in maschinenlesbaren Code

bis Mitte der 1950er **maschinenorientierte Programmierung**, Übersetzung in Maschinensprache durch Datentypisten

1954 IBM entwickelt den ersten **Compiler** = automatischer Übersetzer Hochsprache \leftrightarrow Maschinensprache (für FORTRAN)

1960 ALGOL60 **Programmiersprache** (und Compiler) mit neuen Möglichkeiten, **Backus-Naur-Form** zur Beschreibung der Grammatik

1964 BASIC, für Lehrzwecke entworfene Sprache, einfache Nutzung, aber beschränkte Möglichkeiten

1970-87 Pascal, Modula, Oberon Programmiersprachen für Lehrzwecke, realisieren verschiedene **Programmierparadigmen**, Theorie des Compilerbaus ist voll entwickelt

1995 Java: erste kommerziell standardisierte objektorientierte Programmiersprache, entwickelt von Sun Microsystems

2010 Sun Microsystems wird von Oracle übernommen, Oracle entwickelt Java weiter, Version 7 wurde im Juli 2011 veröffentlicht

Personal Computer (PC)

1.1-5

1774 Philipp Matthäus Hahn entwickelt eine zuverlässige tragbare mechanische Rechenmaschine

ab 1818 Rechenmaschinen nach Vorbild der Leibnizschen Maschine werden serienmäßig hergestellt und weiterentwickelt

1981 IBM PC (16-bit, 4,77 MHz Prozessor, 16-256KB Speicher, 160KB Diskettenlaufwerk, ca. 1500 \$) entwickelt sich zum Massenprodukt und Industrie-Standard

1984/-85/-86 Apple Macintosh, Atari ST, Commodore Amiga - Computer mit graphischer Bedienoberfläche (unterschiedliche Betriebssysteme)

1985 i386 Prozessor (32-bit) erlaubt im *protected mode* bis zu 4GB Hauptspeicher zu adressieren, das MSDOS-Betriebssystem nutzt nur 640KB

1991 Linus Torvalds veröffentlicht die Urfassung des Betriebssystems Linux, nutzt protected mode

2007 Smartphones und darauf installierte intuitiv bedienbare Apps (von **Applikation** = Anwendungssoftware)

1.1.2 Rechnersysteme

Hard- und Software

1.1-6

Hardware (HW)

umfasst die mechanischen und elektronischen Bestandteile eines Computersystems

Beispiel: Grafikkarte, Verbindungsleitungen, Gehäuse, Datenträger, ...

- kann man anfassen
- schwer zu ändern

Software (SW)

jede Art digitaler Daten, die auf der Hardware gespeichert sein können

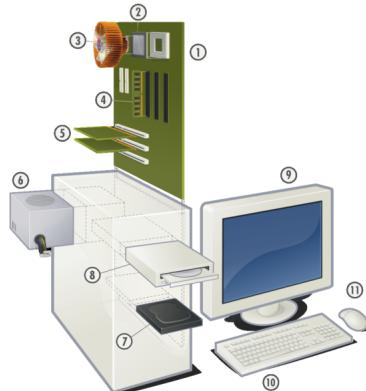
Beispiel: Firmware, Betriebssystem, Anwendungsprogramme, Dateien, ...

- kann man nicht anfassen
- leicht änderbar

speziell: Betriebssystem

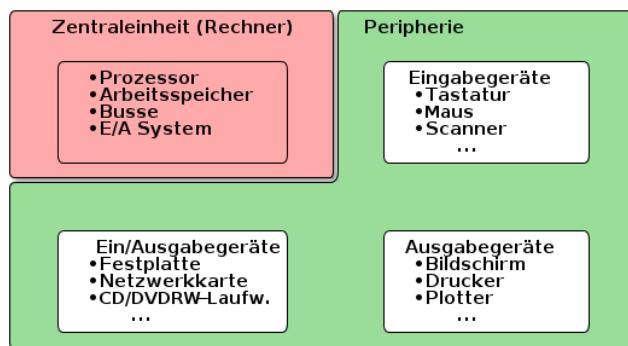
- Softwaresammlung, die die Systemressourcen verwaltet
- rotierende Zuteilung von Rechenzeit + Speicher ermöglicht praktisch gleichzeitige Abarbeitung einer Vielzahl von Prozessen

1.1-7 Ein schematisches PC-System



- (1) Hauptplatine (2) CPU/Prozessor
- (3) Prozessorkühler (4) Hauptspeicher (5) Grafikkarte(n) oder auf der Hauptplatine integrierter Grafikchip; optional: Steckkarte(n), PCI und PCIe **Die**
- (6) Netzteil / Stromversorgung f. Peripherie-Karten (7) Festplatte (8) Optisches Laufwerk (9) Monitor (10) Tastatur (11) Computermaus

1.1-8 Peripherie



- **Eingaben** (Befehle/Daten) werden von Eingabegerät gelesen
- **Verarbeitung** im Prozessor, Zwischenergebnisse werden im **(Arbeits- oder Haupt-)Speicher** gehalten
- **Ausgaben** (Resultate) werden an Ausgabegerät geliefert

1.1-9 Der Prozessor (die CPU)

Bestandteile

Register und Rechenwerk (arithmetical-logical unit - ALU) und weitere Einheiten

- **Rechenwerk (ALU)** führt **Instruktionen** aus (s.u.)
- **Register** = extrem schnelle Hilfspeicher-Zellen, direkt mit ALU verbunden

Instruktionen (Beispiele)

- LOAD: lade Wert aus dem Speicher in ein Register
- STORE: speichere Registerwert

- ADD: addiere Registerwerte
- AND, ...: logische Operation mit Registerwerten

Hauptspeicher (RAM)

1.1-10

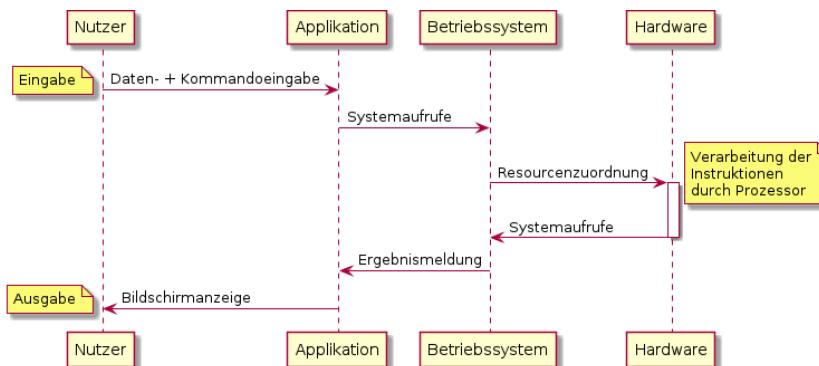
Merkmale

- speichert Instruktionen und Daten aktiver Prozesse
- *flüchtig*: nach Prozessende können zugeordnete Bereiche überschrieben werden
- fortlaufend unterteilt in kleinste Einheiten (Bytes), über Adresse (= Nummer) ansprechbar

Aufbau der Mensch/Machine-Schnittstelle

1.1-11

- Übersetzung der Informationsverarbeitung unserer Erfahrungswelt in kleinschrittige Datenverarbeitung durch schalenartig aufgebautes Softwaresystem
- von außen nach innen: (graphische) Anwendungsprogramme + Bediensystem - **Betriebssystem** - Hardware (insbesondere Prozessor)



Schichtenaufbau

- entlastet Nutzer von Details
- schützt tieferliegende SW/HW-Schichten vor Nutzerfehlern

1.1.3 Problemlösung in der Informatik

Probleme und ihre Lösung

1.1-12

Probleme

sind *allgemeine* Aufgabenstellungen der Form

gegeben Eingabe (bereits bekannte Informationen)
Beispiel: Koeffizienten einer beliebigen quadratischen Gleichung

gesucht Ausgabe (Informationen, die sich aus den bekannten ergeben)

Beispiel: Lösungen dieser Gleichung

Die Problemlösung

ist ein geeigneter **Algorithmus**, d.h ein allgemeines Lösungsverfahren i.A. gibt es *mehrere* Algorithmen, die das Problem lösen Beispiel: $p-q$ -Formel, quadratische Ergänzung, Ausprobieren, ...

Informatik-spezifisch:

Informatiker sollen *gute* Lösungen finden bei unterschiedlichsten Anforderungen
Beispiel: effizient (geringer Rechenzeit- u./o. Speicherbedarf), kostengünstig in Entwicklung und Betrieb, ergonomisch, kompatibel, sicher (Datenschutz), ...

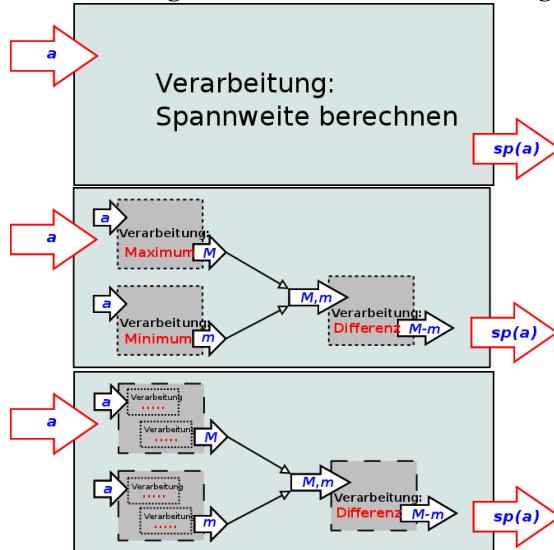
1.1-13 Beispiel: Top-Down-Entwurf

Spielzeugbeispiel: Spannweitenproblem

gegeben Zahlenfolge $\mathbf{a} = (a_0, \dots, a_{n-1})$

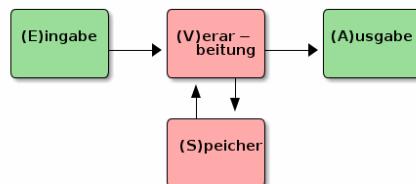
gesucht Spannweite $sp(\mathbf{a}) := \max\{|a_i - a_j| : 0 \leq i < j < n\}$

Problemlösung durch schrittweise Verfeinerung



1.1-14 Grundschema der Datenverarbeitung

im weiteren mit $E \xrightarrow[V]{(S)} A$ angedeutet **EVA(S) Prinzip**



engl.: IPO(S) für Input, Processing, Output (Storage)

Problemlösung

1. analysiere Ein-/Ausgabeanforderungen und geeignete Codierung durch Daten
2. zerlege Verarbeitung in Schrittfolge, die Eingabe in Ausgabe überführt manche Schritte kommen ohne Eingabe/Ausgabe aus, stattdessen kommuniziert der Prozessor nur mit dem Speicher
Beispiel: Schritt 1: EV(S), Schritt 2: (S)V(S), Schritt 3: (S)VA

Beispiel: Kommandozeileninterpreter

1.1-15

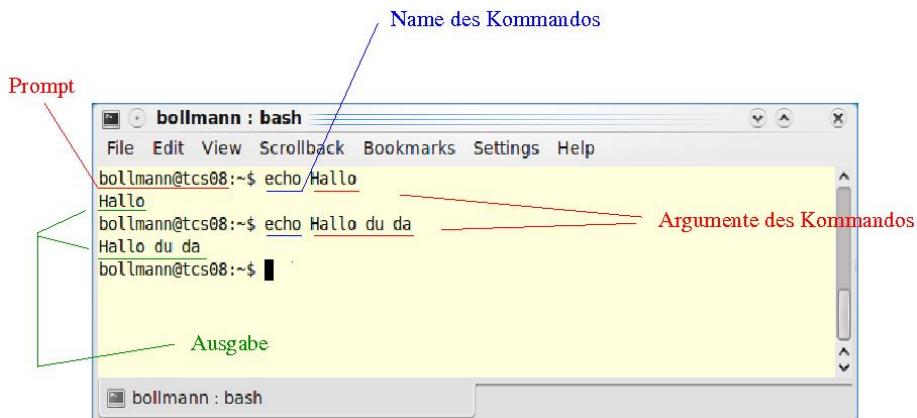
Die **Shell** (Synonyme: **Konsole**, **Terminal**, ...)

ist eine (theoretisch) endlose Folge von EVA-Vorgängen:

- mit **[Enter]** gesendete Eingaben (**Kommandos**) werden *einzeln* analysiert und an das System weitergegeben
- System verarbeitet, liefert Ergebnisse zurück
- Shell liefert Ausgaben auf Bildschirm, unmittelbar danach: **Prompt** (zeigt Eingabebereitschaft an)

Das Shell-Fenster

1.1-16



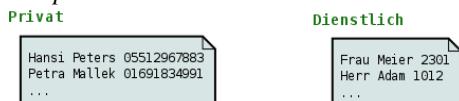
Textdateien

1.1-17

Definition

- eine **Datei** = ein auf einem Massenspeicher gehaltener Datenbestand, kann per **Dateiname** (und Speicherort) identifiziert werden
- **Textdateien** enthalten Daten, die als Text interpretiert werden (d.h. zeilenweise angeordnete Buchstaben, Ziffern, Satzzeichen, ...)

Beispiel



Textdateien kann man mit einem **Editor** anlegen und auch ändern

Beispiel: `gedit` startet einen Editor

`gedit <Datei>` lädt zusätzlich die angegebene Datei

1.1-18 Einige Shell-Kommandos auf Linux-Systemen

Kommando	Erklärung
<code>echo <WortI> ... <WortN></code>	bringt <code><WortI> ... <WortN></code> zur Ausgabe
<code>cat <Datei></code>	
<code>cat <Datei1> ... <DateiN></code>	bringt Dateiinhalt(e) zur Ausgabe
<code>sort <Datei></code>	zeilenweise alphabetisch sortiert ...
<code>sort -k 2 <Datei></code>	... nach dem zweiten Wort jeder Zeile
<code>uniq <Datei></code>	unterdrückt <i>aufeinanderfolgende</i> Duplikate
<code>tr <alph1> <alph2></code>	transformiert Eingaben: <i>i</i> -tes Zeichen in <code><alph1></code> wird als <i>i</i> -tes Zeichen von <code><alph2></code> ausgegeben (andere unverändert)

Beispiel: In `datei.txt` Klein- durch Großbuchstaben ersetzen:

`tr "abcdefghijklmnopqrstuvwxyz" "ABCDEFGHIJKLMNOPQRSTUVWXYZ" < datei.txt`

oder Leerzeichen durch Tabulatoren: `tr " " "\t" < datei.txt` (Erklärung zu < folgt)

online-Hilfe

`man <Kommando>` liefert (englischsprachigen) Erklärungstext

1.1-19 Standard-Eingabe und -Ausgabe

Datenströme

Zu jeder Applikation gehören diese abstrakten Kanäle:

Standardeingabe für eingehende Daten

Standardausgabe für ausgehende Daten

Standardfehlerausgabe für Protokolle/Fehlermeldungen

Beim Aufruf werden sie konkret mit Datenquellen/-senken verbunden.

Typische Verbindungen

Standardeingabe: Tastatureingabe der Konsole

Standardausgabe: Bildschirmausgabe der Konsole

Standardfehlerausgabe: Bildschirmausgabe der Konsole

Je nachdem, wie der Aufruf erfolgt, sind auch andere Verbindungen möglich, z.B. mit Dateien anstelle der Geräte.

1.1-20 Einstellen anderer Verbindungen

Ausgabeumleitung mit >

`cat <Datei1> <Datei2> > <neue Datei>`

speichert die Dateiinhalte (aneinandergehangt) neu ab

Eingabeumleitung mit <

(Kommando) < datei.txt liest datei.txt anstelle Tastatureingaben

in Paragraph 1.1-18 anhand `tr` erklärt

Fehlerausgabeumleitung mit 2>

`cat <Datei1> <nicht vorhandene Datei2> 2> /dev/null` gibt <Datei1> auf Bildschirm aus, die Fehlermeldung jedoch auf /dev/null

/dev/null = (virtuelles) Gerät, das alle empfangenen

Daten ignoriert

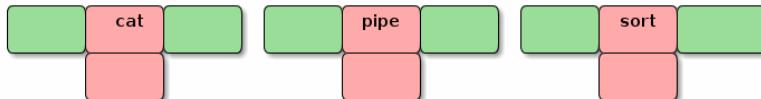
Pipelining mit |

`cat <Datei1> <Datei2> | sort` gibt Dateiinhalte zeilenweise sortiert aus

| ist das Pipe-Symbol

1.1-21

Hier wird außer `cat` und `sort` noch ein weiterer Prozess gestartet: | (Pipe) legt eine Warteschlange (Queue) an, in der `cat`-Ausgaben auf Abholung durch `sort` warten.



Beispiel: Aufbereitung von Textdaten

1.1-22

Problem

übersichtliche Zusammenfassung mehrerer Textdateien

Datei Dienstlich

```
Frau Meier 2301
Herr Adam 1012
Dieter Baumann 0551174481
Herr Klement 8013
Frau Senf 1067
Frau Emmers 0551211001
Dieter Baumann 0551174481
```

Datei Privat

```
Hansi Peters 05512967883
Petra Mallek 01691834991
Dieter Baumann 0551174481
Franzi Emmers 017449862
Michi Baum 05512378994
Manja Sibell 01768899887
```

Pipeline von Shell-Kommandos

`cat Privat Dienstlich | sort -k 2 | uniq | tr "\n" "\t"` liefert:

```
Herr    Adam    1012
Michi   Baum    05512378994
Dieter  Baumann 0551174481
Franzi  Emmers 017449862
...
usw.
```

1.2 Problem, Algorithmus, Programm, Prozess

1.2.1 Probleme und ihre Spezifikation

1.2-1 Motivation

Was bedeutet “der Algorithmus ist korrekt”?

1. liefert das gewünschte Ergebnis d.h. zwischen Ein- und Ausgaben besteht der im Problem geforderte Zusammenhang
2. er **terminiert** für jede erlaubte Eingabe (d.h. die Abarbeitung endet nach einer gewissen Zeit)

Anforderungen, um Korrektheit zu entscheiden

Problem muss hinreichend genau *spezifiziert* sein, durch

- Beschreibung der Problemstellung, Beispiel: sprachliche Umschreibung (gegeben/gesucht), Formel: Eingabegröße \mapsto Ausgabegröße, ...
- Beschreibung der **Schnittstelle**
Beispiel: erlaubte Eingaben, Wertebereich, anwendbare Techniken, ...
- oft auch: Beschreibung des Verhaltens bei Fehleingaben, der erlaubten Hilfsmittel, Art der Dokumentation etc.

1.2-2 Gute und schlechte Spezifikationen

Erster Schritt zur Lösung eines Problems ist eine gute Spezifikation:

- unzweideutig,
- vollständig und
- (idealerweise) einfach und kurz

Beispiel für unzureichende Spezifikation

gesucht Lösungen der Gleichung $a_2 \cdot x^2 + a_1 \cdot x + a_0 = 0$

Welche Größen sind gegeben? Was sind die Wertebereiche? Wie ist mit Sonderfällen (z.B. keine reelle Lösung) zu verfahren?

Besser:

gegeben drei Koeffizienten $a_0, a_1, a_2 \in \mathbb{R}$

gesucht alle reellen oder komplexen Werte x mit $a_2 \cdot x^2 + a_1 \cdot x + a_0 = 0$

1.2-3 Ein Beispiel mit Schnittstellenvorgabe

Exakte Berechnung von Binomialkoeffizienten

gegeben nichtnegative ganze Zahlen n und k mit $n \geq k$,

gesucht der exakt berechnete Binomialkoeffizient $\binom{n}{k}$

siehe Kapitel ??

Signatur int binom(int, int)¹

Fehlerfall Ist k größer als n oder liegt der exakte Binomialkoeffizient nicht im Wertebereich int, so wird Ergebnis -1 geliefert.

Eine mathematische Spezifikation

1.2-4

Abkürzung: ggT = größter gemeinsamer Teiler

Beispiel: ggT(48, 27) = 3

Das ggT-Problem

gegeben ganze Zahlen $a, b > 0$

gesucht eine ganze Zahl d mit den Eigenschaften

- d teilt a und b
- es gibt keinen Teiler d' von a und b mit $d' > d$

1.2.2 Algorithmen

Zum Begriff des Algorithmus

1.2-5

Algorithmus

= endliche, schrittweise, präzise (Rechen-)Vorschrift

- **endlich** : endlicher Text, Diagramm, ...
- **schrittweise** : einfache Einzelaktionen
- **präzise** : Schritte+Reihenfolge unmissverständlich, vollständig

Beispiele und Gegenbeispiele

- schriftliche Multiplikation
- **bebilderte Möbelbauanleitung** (unklare Reihenfolge, z.T. komplexe Einzelaktionen)
- **Kochrezept** (komplexe Einzelaktionen, unpräzise)

Eine Lösung des ggT-Problems

1.2-6

Idee in Prosa

¹Erklärung: Aufruf in der Form `binom(n, k)`, wobei n, k sowie Ergebnis aus festgelegtem Wertebereich int

Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das Kleinere vom Größeren weg, dann muss (schließlich) eine Strecke übrig bleiben, die die vorangehende misst.

strukturierte Prosa (Pseudocode)

Euklidischer Algorithmus (klassisch)

Eingabe: a, b mit $a > b \geq 0$

$A \leftarrow B$ steht für:
speichere aktuellen Wert von B in A

Initialisiere $x \leftarrow a, y \leftarrow b$

Iteriere solange $y \neq 0$

Falls ($x > y$) $x \leftarrow x - y$

sonst $y \leftarrow y - x$

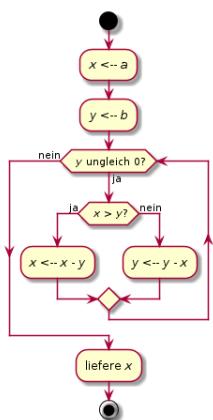
Terminiere liefere x

Der Algorithmus hat diese typische Struktur (**iteratives Grundmuster**):



1.2-7 Ablauf des Algorithmus

Ablaufdiagramm



Prozess

Vorgang einer algorithmisch ablaufenden Informationsverarbeitung (nach Informatik-Duden)

Dynamisches Verhalten

situationsbedingte Reihenfolge der Einzelschritte

Ablaufverfolgung (Tracing)

Buchführung über die Variablenwerte

Beispiel: Tabelle mit Werten nach jedem Schritt

1.2-8 Programmcode = programmiersprachliche Notation eines Algorithmus

Beispiel: Java-Code (Fragment)

```

1   int x = a;           // "int-Variablen" x und y anlegen und mit ..
2   int y = b;           // .. den Werten von a und b initialisieren.
3   while ( y != 0) {    // '!= ' steht fuer 'ungleich'
4       if ( x > y)
5           x = x - y;    // berechne x - y, weise Ergebnis x zu
6       else
7           y = y - x;    //           (analog)
8   }
9   System.out.println(x); // Textausgabe
  
```

Das Ablaufprotokoll (*Trace-Tabelle*)

1.2-9

beschreibt zeilenweise die Situation NACH Codeausführung

Beispiel: Startwerte $a = 30, b = 12$

Zeile(n)	x	y	
3,4	30	12	
5	18	12	
3,4	18	12	
5	6	12	
3,4	6	12	
7	6	6	
3,4	6	6	
7	6	0	
3	6	0	
9	6	0	Textausgabe: 6

Tracing mit variablen Startwerten $a > b \geq 0$

1.2-10

Zeile	x	y	Bemerkung
3, 4	a	b	
5	$a - b$	b	
:	:	:	
5	$r_1 = a - q_1 \cdot b$	b	r_1 = Rest bei Division $a : b$
3, 4	r_1	b	
7	r_1	$b - r_1$	
:	:	:	
7	r_1	$r_2 = b - q_2 \cdot r_1$	r_2 = Rest bei Division $b : r_1$
:	:	:	

Umformulierung des Euklidischen Algorithmus

1.2-11

Euklidischer Algorithmus (modern)**Eingabe:** Zahlen a, b mit $a > b \geq 0$ **Initialisiere** $x \leftarrow a, y \leftarrow b$ **Iteriere** solange $y \neq 0$

- $r \leftarrow x \bmod y$

$x \bmod y$ (gesprochen: "x modulo y") = Formel für
Divisionsrest von x durch y

- $x \leftarrow y$

- $y \leftarrow r$

Terminiere liefere x **Java-Code (Fragment)**

```

1     int x = a, y = b;
2     int r;
3     while (y != 0) {
4         r = x % y; // Divisionsrest
5         x = y;
6         y = r;
7     }
8     System.out.println(x);

```

1.2.3 Prozessoren

1.2-12 Begriffe

Definition

- jeder einzelne Programmablauf heißt *Prozess*, das ausführende System **Prozessor**
- **Befehlssatz** = Liste der Mikrobefehle (Instruktionen), die der Prozessor umsetzen kann
- **Maschinencode (binary)** = Folge von (binär codierten) Befehlen aus dem Befehlssatz

Befehlscode 011001010 ...

- unmittelbar umsetzbar in elektrische Signale ⇒ maschinenlesbar
- können gespeichert werden wie normale Daten

1.2-13 Prozess

Laden des Maschinencodes

- Befehlsfolge in *aufeinanderfolgende Speicherzellen* kopieren
- **Befehlszähler** = besonderes Register, das die Adresse des aktuellen Befehls enthält (*instruction pointer IP*)

Programmausführung: Der von Neumann-Zyklus

Initialisiere setze IP auf Anfangsadresse des geladenen Maschinencodes

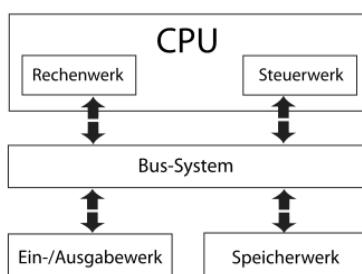
Iteriere solange HALT-Befehl nicht erreicht:

- aktuellen Befehl abrufen und decodieren
- Operanden beschaffen
- Ergebnis berechnen/speichern und Befehlszähler aktualisieren

Terminiere (z.B. Speicherzellen freigeben)

1.2-14 Das von Neumann-Prozessormodell

Einheiten:



Rechenwerk für arithm.+logische Befehle

Steuerwerk kontrolliert v. N.-Zyklus (Sprungbefehle)

Bus-System Datenleitung

E-/A-Werk Kommunikation mit Peripherie

Speicherwerk Speicher inkl. Adresslogik

[Mikrobefehle](#)

1.2-15

arithmetisch-logisch

Beispiele: Addition, Multiplikation, ..., AND, OR, NOT, ... verknüpfen
 Registerinhalte:

- $\langle \text{Operation} \rangle \langle \text{Operand} \rangle$ oder
- $\langle \text{Operation} \rangle \langle \text{Operand1} \rangle \langle \text{Operand2} \rangle$

Transport

Beispiele: STORE, LOAD, READ, ... sorgen für Datentransport zwischen Einheiten:

- $\langle \text{Befehl} \rangle \langle \text{Adresse} \rangle \langle \text{Adresse} \rangle$

Sprünge

Beispiel: JMP, JEQZ, ... setzen den Befehlszähler:

- $\langle \text{Befehl} \rangle \langle \text{Adresse} \rangle$ (**unbedingter Sprung**) oder
- $\langle \text{Befehl} \rangle \langle \text{Bedingung} \rangle \langle \text{Adresse} \rangle$ (**bedingter Sprung**)

[while-Schleife \$\mapsto\$ von Neumann-Befehlsfolge](#)

1.2-16

Alte BASIC-Varianten:

Beispiel: while (y != 0) <Befehl>

```
0100 ...
0110 IF y = 0 GOTO 0140
0120 <Befehl>
0130 GOTO 0110
0140 ...
```

analog geht das mit Assemblercode:

Assemblercode

= Folge von Befehlskürzeln für Prozessorbefehle

Beispiel: LOAD $\langle \text{Adresse} \rangle$, JEQZ ...

Maschinencode entsteht durch 1-zu-1-Übertragung der Befehle in Binärcodes
 (Grobablauf nachfolgend beschrieben)

[Fiktiver Assemblercode](#)

1.2-17

Java-Code (Fragment)

```
z = 10;
while ( z != 0 ) {
    // gib z aus
    z = z - 1;
}
```

dies wird in Assemblercode umgewandelt, Ergebnis könnte dies sein:

Assemblercode (fiktiv und stark vereinfacht)

Befehlsadresse	Kurzbefehle	Erklärung
0	SET 1 10	setze Register 1 auf den Wert 10
1	JEQZ 1 6	falls Inhalt von Register 1 gleich 0, setze Befehlszähler IP auf 6
2	WRITE 1	schreibe Inhalt von Register 1 auf Standardausgabe
3	SET 2 1	setze Register 2 auf den Wert 1
4	SUB 1 2	speichere Differenz Reg.inhalt 1 minus Reg.inhalt 2 in Register 1
5	JMP 1	setze $IP \leftarrow 1$
6	HALT	Prozessende

1.2-18 Fiktiver Maschinencode

Binärcodierung der Kurzbefehle

$\langle\text{Befehlscode}\rangle \langle\text{Operand/Adresse}\rangle \langle\text{Operand/Adresse}\rangle$
(bzw. 0-Bits für jeden fehlenden Operanden)

(fiktive) Codes:

0010 $op-1$ $op-2$ für SET
0011 $op-1$ $op-2$ für JEQZ
0100 $op-1$ $op-2$ für SUB
1010 $addr$ 0000 für JMP
1100 $op-1$ 0000 für WRITE
1101 0000 0000 für HALT

Assembler
SET 1 10
JEQZ 1 6
WRITE 1
SET 2 1
SUB 1 2
JMP 1
HALT

Maschinencode
0010 0001 1010
0011 0001 0110
1100 0001 0000
0010 0010 0001
0100 0001 0010
1010 0001 0000
1101 0000 0000

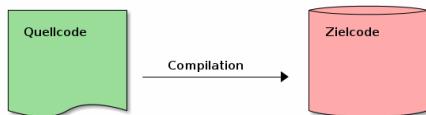
1.2-19 Compiler

Merke

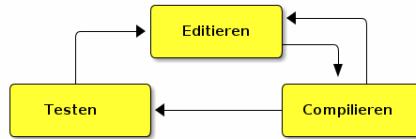
- Assembler- und Maschinencode sind auf den Befehlssatz des Prozessors abgestimmt, darum nicht portabel!
- die Übersetzung Programmcode \mapsto Maschinencode muss sowohl auf die Programmiersprache als auch den jeweiligen Prozessor zugeschnitten sein!
- den Übersetzungs vorgang nennt man **Compilation**, den Programmcode auch **Quellcode** oder **Quelltext**, den Maschinencode auch **Zielcode**

Compiler

= Programmierwerkzeug zur automatischen Übersetzung



1.2-20 Zyklus der Programmentwicklung



Editieren

mit *Editor* Quelltext-Datei erstellen und abspeichern
Beispiel: Shell-Kommando gedit HelloWorld.java &

Compilieren

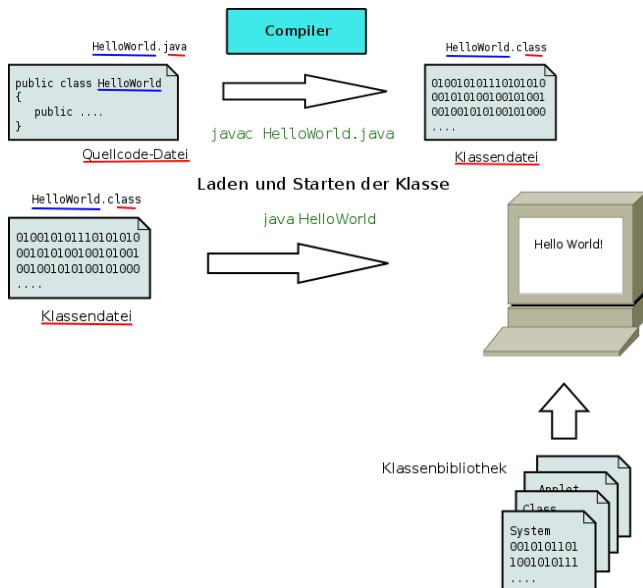
Compiler mit Quelltextdatei aufrufen
Beispiel: Shell-Kommando javac HelloWorld.java
⇒ **Klassen-Datei** HelloWorld.class entsteht

Testen (Ausführen)

Die Klasse in der **Laufzeitumgebung**² starten
Beispiel: Shell-Kommando java HelloWorld
⇒ Textausgabe Hello, world! erscheint (siehe nachfolgend)

Grobablauf vom Quelltext zum Prozess

1.2-21



benötigte vordefinierte Routinen (Bsp.: System.out.println) werden automatisch aus **Klassenbibliothek(en)** nachgeladen

1.2.4 Erste Java-Beispiele

Unser erstes Java-Programm

1.2-22

²die sogenannte **virtuelle Java-Maschine**, Java VM, JVM

Java-Programm = vollständiger Java-Code (Umgebungs-Festlegung etc.)

- Programme werden in Dateien gespeichert (**Quellcode**-Datei)
- Quellcode kann automatisch in Prozessor-Zielcode übersetzt werden

```
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello, world!"); // Anweisung
    }
}
/* Dieses Programm gibt einen Begrüßungstext aus
und endet. */
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/HelloWorld.java>

- Der Quellcode definiert die Klasse `HelloWorld`, und `public static void main (String[] args)` kündigt ihre **main-Methode** an.
- Danach steht in geschweiften Klammern die einzige **Anweisung** der Methode (durch Semikolon ; beendet): `System.out.println("Hello, world!");`. Dies ist ein **Methodenaufruf** — zu erkennen an den runden Klammern.
 - `System.out.println` ist eine vordefinierte **Methode**, sie bringt ihren **Aufrufparameter** zur Ausgabe und erzeugt einen Zeilenwechsel.
 - Der Aufrufparameter ist die Zeichenkette `Hello, world!` — die Anführungszeichen gehören nicht dazu, sie begrenzen konstante Zeichenketten (**String-Konstanten**).
- Text hinter `//` und innerhalb `/* ... */` ist Kommentar und wird vom Compiler ignoriert

Merkel

- Quellcode MUSS zur Weiterverarbeitung in Datei namens `HelloWorld.java` gespeichert werden (kommt `public class <Klassenname>` vor, so muss die Datei `<Klassenname>.java` heißen)
- main-Methode beginnt praktisch IMMER³ so:
`public static void main (String[] args)`

1.2-23 Applikationen

`public static void main (String[] args) { ... }`
 ist die **main-Methode**. Klassen mit main-Methode können so benutzt werden.

java `<Klassenname>`

Applikationen sind Klassen, die eine main-Methode enthalten.

1.2-24 Der Compiler ist dein Freund ...

Bis das Programm korrekt ist ...

... wird der Zyklus **Editieren - Compilieren - Testen** immer wieder durchlaufen

Lehrreich: Ändern und Ausprobieren!

³auch `public static void main (String args[])` ist möglich u. anstatt `args` Beliebiges

- public weglassen bei class
- static weglassen
- Semikolon entfernen
- args durch etwas anderes ersetzen
- eine Klammer weglassen usw.

Modifikation: Hello (*User*)

1.2-25

```
public class HelloUser {
    public static void main(String args[]) {
        System.out.println(args[0]);
    }
}
```

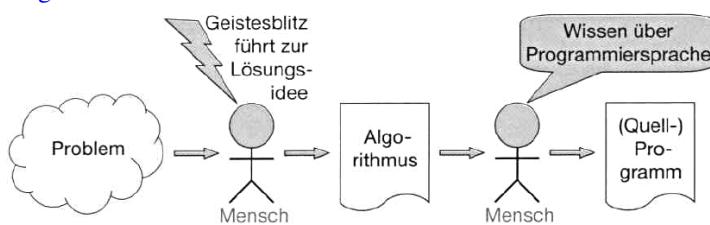
Kommandozeilenargumente sind über args zugreifbar. Nummerierung ist 0-basiert, d.h. das erste Kommandozeilenargument wird mit args[0] angesprochen. Jedes **Kommandozeilenargument** ist eine Zeichenfolge (String) ohne Leerraum.

Der Aufruf `java HelloUser Erwin` generiert die Ausgabe:

Erwin

Programmieren im Kleinen

1.2-26

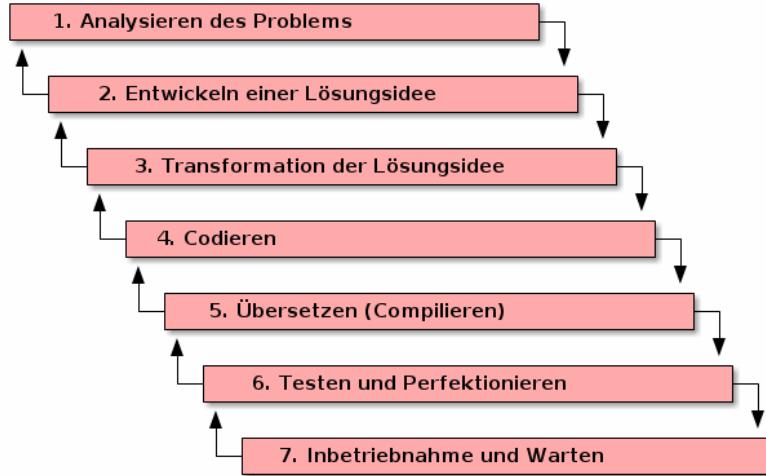


Grafik aus Pomberger/Dobler: Algorithmen und Datenstrukturen

Beispiel: Programmierung des euklidischen Algorithmus

1. Verstehen des ggT-Problems
2. geometrische Lösungsidee und Formulierung in Prosa
3. Transformierung in Pseudocode (z.B. strukt. Prosa) bzw. Aktivitätsdiagramm
4. Codierung der Kernidee in Java oder C, oder ... (vollständiges Programm sehen wir später)

1.2-27 Programmierung im Großen



Nachfolgend ein Beispiel für mehrfaches Testen und Perfektionieren:

1.2-28 Echo-Nachbau

Ziel

siehe Paragraph 1.1-18 Nachprogrammierung des Shell-Kommandos echo durch eine Java-Applikation

Idee

Beispiel: echo Hallo du da! bewirkt Ausgabe Hallo du da!

Verwende HelloUser.java als Codesteinbruch:

- cp HelloUser.java Echo.java
- editiere - compiliere - teste Klasse Echo

```

public class Echo { // Version 1
    public static void main(String[] args) {
        System.out.println(args[0]);
    }
}
  
```

Test: java Echo eins erzeugt wie gewünscht die Ausgabe eins, aber java Echo verursacht Prozessabbruch: ver-suchter Zugriff auf nicht vorhandenes Kommandozeilenargument!

```

1  public class Echo { // Version 2
2      public static void main(String[] args) {
3          int i=0;
4          while ( i < args.length ) {
5              System.out.println(args[i] + " ");
6              i = i+1;
7          }
8      }
9  }
  
```

Test: java Echo führt nicht zum Abbruch, sondern erzeugt wie gewünscht keine Ausgabe. java Echo eins zwei erzeugt die Ausgaben eins und zwei, aber *auf einzelnen Zeilen*. Im Vergleich dazu erzeugt das Shell-Kommando echo die Ausgabe *auf einer Zeile*.

```

1 public class Echo { // Version 3
2     public static void main(String[] args) {
3         int i=0;
4         while ( i < args.length ) {
5             System.out.print(args[i] + " ");
6             i = i+1;
7         }
8         System.out.println();
9     }
10 }
```

<http://www.stud.informatik.uni-goettingen.de/inf10/java/Echo.java>

- Zeile 5: `System.out.print(<argument>)` bringt `<argument>` zur Ausgabe, *ohne* Zeilenwechsel.
Durch `+ " "` wird vorher an das Argument ein Leerzeichen angehängt.
- Zeile 8: `System.out.println()` sorgt für die Ausgabe und einen Zeilenwechsel.

Euklids Algorithmus als vollständiges Java-Programm

1.2-29

```

1 public class EuklidModern {
2     public static void main(String[] args) {
3         int a = Integer.parseInt(args[0]), b = Integer.parseInt(args[1]);
4         int x = a, y = b;
5         int r;
6         while ( y != 0 ) {
7             r = x % y; // Divisionsrest
8             x = y;
9             y = r;
10        }
11        System.out.println(x);
12    }
13 }
```

<http://www.stud.informatik.uni-goettingen.de/inf10/java/EuklidModern.java>

- Zeilen 3, 4: Die Werte werden als Kommandozeilenargumente (also Strings) übergeben. Die vordefinierte Methode `Integer.parseInt` interpretiert Strings als ganze Zahlen (engl.: integer).
- Beispielauftrag: `java EuklidModern 20 12`

Kommandozeilenargumente vs. interaktive Eingabe

1.2-30

Kommandozeilenargumente

- durch Kommandozeilenargumente werden dem Aufruf Daten „mitgegeben“
- also: deren Anzahl und Werte stehen bei Programmstart fest

Interaktive Eingabe vom Standard-Eingabedatenstrom

- das laufende Programm fordert Daten an (z.B. durch entsprechendes Prompt angezeigt)
- Daten werden je nach Bedarf bereit gestellt

Beispiel: Temperatur-Umrechnung

1.2-31

```

7  public class Temperatur{
8      public static void main(String[] args) {
9          System.out.print("Eingabe: ");
10         double t;
11         t = StdIn.readDouble(); // Zugriff auf Standard-Eingabe
12         double tf,tc;
13         tf = t * 1.8 + 32; // Grad-Celsius -> Grad-Fahrenheit
14         tc = (t - 32)*5/9; // Grad-Fahrenheit -> Grad-Celsius
15         System.out.println(t + " Grad Celsius entsprechen "
16                             + tf + " Grad Fahrenheit.");
17         System.out.println(t + " Grad Fahrenheit entsprechen "
18                             + tc + " Grad Celsius.");
19     }
20 }
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Temperatur.java>

- Z. 9: nach Eingabeaufforderung kein Zeilenwechsel, d.h. Nutzereingabe unmittelbar dahinter
- Z. 10: `t` wird als Variable definiert, die `double`-Werte annehmen kann (nichtganze Zahlen), Zeile 12 entsprechend
- Z. 11: `t` wird auf eingegebenen Wert gesetzt. Die Methode `readDouble` aus der Klasse `StdIn` wartet auf eine durch **[Enter]** gesendete Eingabe. Kann sie als Zahl interpretiert werden, so wird ihr Wert zurückgeliefert und der Variablen `t` zugewiesen (andernfalls wird der Prozess mit einer Fehlermeldung abgebrochen).
- In Zeilen 13 und 14 wird rechts vom **Zuweisungsoperator** = der Wert von `t` mit anderen verknüpft. Diese "Formeln" sind **arithmetische Ausdrücke**: Wertbildung nach den üblichen Regeln ("Punkt- vor Strichrechnung", Klammern beachten etc.), dann Zuweisung an `tf, tc`.
- Z. 15, 16 (und 17, 18): berechnete Werte (genauer: die Ziffernfolgen - also Strings) werden mit konstanten Strings verkettet und zusammen auf einer Zeile ausgegeben. Ausdrücke wie
`t + " Grad Celsius entsprechen " + tf + " Grad Fahrenheit."`
dürfen über mehrere Quelltextzeilen gehen, aber jeder *einzelne* String muss komplett auf einer Zeile stehen.

Ein kompletter Dialog bei Aufruf `java Temperatur`:

```

Eingabe: 10.0
10.0 Grad Celsius entsprechen 50.0 Grad Fahrenheit.
10.0 Grad Fahrenheit entsprechen -12.22222222222221 Grad Celsius.
```

1.3 Java-Kurzeinführung

1.3-1 Sprachelemente

Java-Quellcode (in Dateien `<Dateiname>.java`)

ist zusammengesetzt aus folgenden **Sprachelementen** (und sonst nur **Whitespace** sowie Kommentaren)

- **Schlüsselwörter**
Beispiele: `public, class, while, return, ...`
- **Litere** = textuell angegebene konstante Werte
Beispiele: `"Hello World", 'a', 1.8, 32, ...`
- **Bezeichner** = von Programmierern gewählte Namen
Beispiele: `x, y, HelloWorld.java`
- **Trenn-, Strukturierungs- und Operatorzeichen**
Beispiele: `, ; () [] { } : + - * / % :`

Whitespace fungiert auch als Trennzeichen

Gestalt bzw. Bedeutung von Quelltextbestandteilennennt man ihre **Syntax** bzw. **Semantik****Schlüsselwörter**

1.3-3

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Tabelle nach http://en.wikipedia.org/wiki/Java_keyword**Zwecke**

- **Ablaufsteuerung:** assert break case catch continue default do else finally for if return switch throw throws try while
Beispiel: while (*<Bedingung>*) *<Anweisung>*

- **Daten, Datentypen:** boolean byte char double enum float int long short strictfp void
Beispiel: int *<Variablenname>*;

- **Objektorientierung:** abstract class extends implements import instanceof interface new package super this
Beispiel: class *<Klassenname>* { ... }

- **Zugriff u.a.:** final native protected private public static synchronized transient volatile
Beispiel: public class *<Klassenname>* { ... }

- **reserviert:** const goto

Bezeichner

1.3-4

- Folgen von Buchstaben und Ziffern, die mit einem Buchstaben beginnen

- sind verschieden von allen Schlüsselwörtern und *reservierten Namen*

Beispiel: i, j, k, ... (Laufvariablen), x, y, y1, bStrich, s, t, ... (Zwischenergebnisse), HelloWorld, GGT (Klassen), ...

Reservierte Namen

werden nur in spezieller Bedeutung verwendet, Beispiel: main, true, false, null

Variablenklärung (und -Initialisierung)

1.3-5

Variablen können Werte speichern. Der Wertebereich wird durch den **Typ** angegeben.

- jede Variable muss vor der Verwendung *deklariert* werden
- jede Variable muss vor erstem *Lesezugriff* initialisiert werden

Initialisierung = Anfangswertsetzung

Syntaxvarianten

$\langle \text{Typ} \rangle \langle \text{Bezeichner} \rangle;$
 $\langle \text{Typ} \rangle \langle \text{Bez1} \rangle, \langle \text{Bez2} \rangle, \dots, \langle \text{BezN} \rangle;$
 $\langle \text{Typ} \rangle \langle \text{Bez1} \rangle = \langle \text{Wert1} \rangle, \langle \text{Bez2} \rangle, \dots, \langle \text{BezN} \rangle;$

Beispiel: int x;
 Beispiel: int p, q, r;
 Beispiel: int p = 0, q;

1.3-6 Code-Struktur einfacher Applikationen

Bis auf weiteres haben unsere Klassen folgende einfache Struktur

```
public class <Klassenname> {
  public static void main (String[] args) {
    <Anweisung>;
    <Anweisung>;
    ...
    <Anweisung>;
  }
}
```

- Der Klassenname ist frei wählbar, die Konvention ist jedoch mit Großbuchstaben zu beginnen.
- public vor class ist optional (Erklärung später). Falls public davor steht, muss der Dateiname dem Klassennamen entsprechen.
- Es sind sogar mehrere Klassendefinitionen in einer Quelltextdatei erlaubt, maximal eine davon public.

Bemerkung

Einrückungen machen den Code gut lesbar, der Compiler benötigt sie jedoch nicht

```
public class HelloWorld{public static void main(String[ ] args){System.out.println("Hello World!");}}
```

1.3-7 Anweisungen

Aktion = Änderung der Speicherbelegung

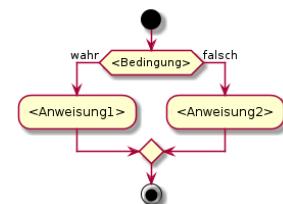
beschreiben auszuführende **Aktionen** und ihre Reihenfolge, Beispiele:

- Wertzuweisungen: a = 13; x = x - y;
- Methodenaufrufe: System.out.println(...), Integer.parseInt(...), Math.sqrt(...) usw.
- Steuerungsanweisungen:** Verzweigungs-, Block- und Schleifenanweisungen

steuern den Kontrollfluss = zeitliche Reihenfolge von Einzelaktionen

1.3-8 Verzweigungen**if-else-Anweisung**

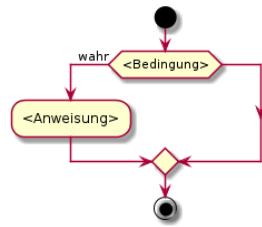
```
if ( <Bedingung> )
  <Anweisung1>;
else
  <Anweisung2>;
```

**if-Anweisung (bedingte Anweisung)**

1.3 Java-Kurzeinführung

5

```
if ( <Bedingung> )
    <Anweisung>;
```

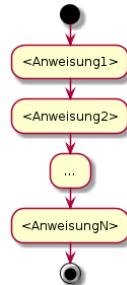


Blöcke: Zusammenfassen von Anweisungen

1.3-9

Blockanweisung

```
{
    <Anweisung1>;
    <Anweisung2>;
    ...
    <AnweisungN>;
}
```



```
// Beispiel
if (x > y) {
    int t = x; // Deklarationen innerhalb eines Blocks
    x = y;
    y = t;
} // ordnet x und y (t ausserhalb ungültig!)
```

„Kopfgesteuerte“ Schleife

1.3-10

while-Anweisung

```
while ( <Bedingung> )
    <Anweisung>;
```



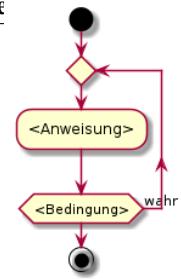
```
// Beispiel
while (y != 0) { // \
    r = x % y; // \
    x = y; // \
    y = r; // \
} // ...
```

„Fußgesteuerte“ Schleife

1.3-11

do-while-Anweisung

```
do {Anweisung};
while (Bedingung);
// Achtung: Semikolon NACH der Bedingung
```



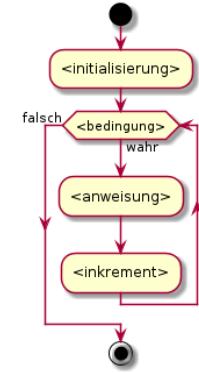
```
// Beispiel
int n;
do {
    System.out.print("Bitte positive Zahl eingeben: ");
    n = StdIn.readInt();
} while (n <= 0); // solange kleiner oder gleich 0
// ...           // gepruefte Eingabe verarbeiten
```

1.3-12 Zählschleife

for-Anweisung

```
// Variante 1: Laufvariable bereits deklariert
for (Initialisierung; Bedingung; Inkrement)
    (Anweisung);
```

```
// Variante 2: mit Deklaration der Laufvariablen
for (Dekl und Init.); (Bedingung); (Inkrement))
    (Anweisung);
```



1.3-13 Beispiel: Tabelle der Zweierpotenzen $\leq 2^n$ erzeugen

```
1 public class PowersOfTwo {
2     public static void main (String[] args) {
3         int n = Integer.parseInt(args[0]), p = 1;
4         int i;                                // <-- 
5         for ( i=0; i<=n ; i = i+1) {          // <-- 
6             System.out.println(i + " " + p);
7             p = 2*p;
8         }
9     }
10 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/PowersOfTwo.java>

Das Gleiche mit üblicher Kurzschrifweis der for-Schleife:

```

2 public class PowersOfTwo {
3     public static void main (String[] args) {
4         int n = Integer.parseInt(args[0]), p = 1;
5         for (int i=0; i<=n ; i++) {           // <-
6             System.out.println(i + " " + p);
7             p = 2*p;
8         }
9     }
10 }
```

Wird die Laufvariable so deklariert, so ist sie nur innerhalb des Schleifenrumpfs gültig. `i++` kann hier als bequeme Abkürzung für `i=i+1` verwendet werden.

→ Info1-Sammlung (II-ID: 9rfj1a1n2i0)

Ausdrücke

1.3-14

Ein **Ausdruck** ist ein Literal, eine Variable oder eine Kombination davon mit Operatoren, die einen Wert beschreibt (z.B. durch Angabe einer Berechnungsformel).

Beispiele

```

100, 3.14, x // int-Literal, double-Literal, Variable
"Hello"        // String-Literal
/* einige kombinierte Ausdrücke */
2 * a * b      // 2ab
Math.sqrt(x)   // Aufruf einer Methode, die einen Wert liefert
"Hello, " + "World!" // "Hello, World!" (Ergebnis der Verkettung)
/* einige Operatoren */
+, -, *, /, % // Addition, Subtraktion, Multiplikation, Division, Rest
++, --         // Inkrement, Dekrement
+              // String-Verkettung
()             // Vorrangklammern oder Funktionsaufruf
[]             // Feldzugriffsoperator
```

Einteilung von Software-Fehlern

1.3-15

Compilezeitfehler Kompilierfehler = Syntaxfehler in einer Quellcodedatei

Beispiel: `System.out.println(3,5);`

Laufzeitfehler = zur Laufzeit auftretende Fehler

Beispiel: Zugriff auf nicht vorhandenes Kommandozeilenargument

Insbesondere: logische Fehler

= Laufzeitfehler, die nicht zum Abbruch, aber zu falschen Ergebnissen oder unerwünschten Endlosschleifen führen können

Beispiele

```
F = G * m1*m2 / r*r;    // Gravitationsgesetz
```

```
int i=0, p=1;
while(i<=n) p=2*p; System.out.println(i+": "+p); i++;
```


Kapitel 2

Grundlegende Daten und Algorithmen

2.1 Daten und Datentypen

Daten

2.1-1

Definition

Daten sind Informationen, die zur *automatisierten Verarbeitung* codiert wurden.

frei nach DIN 44300

Beispiele

Information	Daten
Zahl 9	0000 0000 0000 1001
Musikstück	BeethovensNeunte.mp3
Bild	Gaenseliesel.jpg
Prozessorbefehl	1001 0001 1000

Codierende Symbolfolgen/Signale heißen *Nachrichten*

Beispiel: Zeichenfolge 2+2=4 (5 Zeichen).

Also: Daten sind Paare aus Nachricht + codierter Information(Syntax + Semantik)

Alphabete und Nachrichten

2.1-2

Nachrichten

- sind bestimmte *endliche* Symbolfolgen aus einem endlichen Alphabet Σ
- Menge der Symbolfolgen (“Wörter”) über Σ wird mit Σ^* bezeichnet

(siehe Kapitel ??)

Beispiele für Alphabete

- Bits $\mathbb{B} := \{0, 1\}$ (Wörter aus \mathbb{B}^* nennen wir **Bitmuster**)
- dezimale Ziffern 0, 1, ..., 9, Buchstaben A, B, ..., Z, a, b, ..., z

- **alphanumerische Zeichen** = Buchstaben, Ziffern und andere Zeichen

2.1-3 Codierung und Decodierung

Codierung

= eindeutige Zuordnung $Information \leftrightarrow Daten$, also eine Abbildung
 $C : Informationen \rightarrow Nachrichten \subseteq \Sigma^*$.

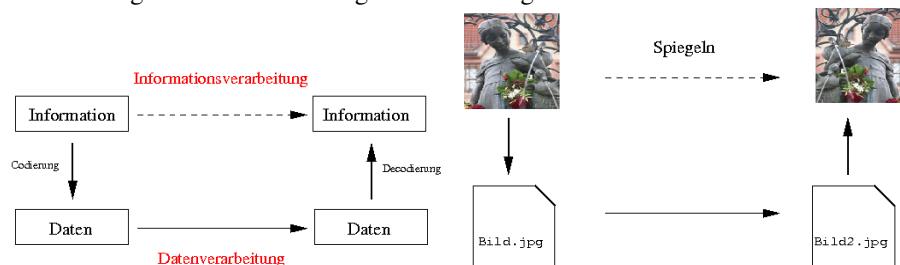
Decodierung (oder Interpretation)

= zugehörige Umkehrabbildung $D : Nachrichten \rightarrow Informationen$

2.1-4 Prinzip der Informationsverarbeitung

Informationsverarbeitung

= Codierung + Datenverarbeitung + Decodierung



Ingenieursproblem

Geeignete Codierungen auswählen und alle 3 Schritte effizient automatisieren.

2.1.1 Datentypen

2.1-5 Begriffe

Datenobjekt

= Software-Repräsentation eines Werts

Datentyp

= Menge von **Werten** zusammen mit einer Menge anwendbarer **Operationen**

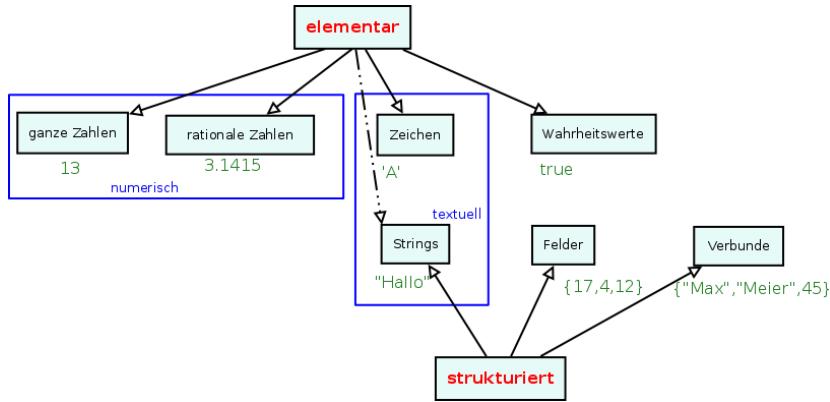
Beispiel: Datentyp `int`: bestimmte Menge ganzer Zahlen, die addiert, multipliziert ... werden können

Java bietet für grundlegende Datentypen bereits alles Nötige, z.B.

- Codierung: Wert \mapsto Bitmuster
- **String-Umwandlung**: Werte als Text ausgeben/einlesen
- **Literale**: Quelltextnotation konstanter Werte

Welche Datentypen brauchen wir?

2.1-6



- **Elementare Werte** werden nicht weiter in Einzelbestandteile zerlegt
Beispiel: Zahlen, Zeichen, Wahrheitswerte
- **Strukturierte Werte** sind zerlegbar in kleinere Bestandteile
Beispiele: Vektoren (= Zahlenfolgen), Personendaten (Name, Alter, ...)

Analog spricht man von elementaren und strukturierten Datentypen. Wir behandeln zunächst die elementaren.

2.1.2 Datentypen für Zahlen

Codierungen für natürliche Zahlen sind grundlegend für alle weiteren.

Natürliche Zahlen: Stellenwertcodierung

2.1-7

Fakt. Ist $N > 0$ (**Stellenzahl**), so ist jedes $z \in \mathbb{N}, 0 \leq z < 2^N$ eindeutig darstellbar als

$$z = z_{N-1} \cdot 2^{N-1} + z_{N-2} \cdot 2^{N-2} + \dots + z_1 \cdot 2 + z_0 \cdot 1$$

mit $z_0, \dots, z_{N-1} \in \{0, 1\}$. $\text{bin}_N(z) = z_{N-1}z_{N-2}\dots z_1z_0 \in \{0, 1\}^*$ heißt **N-stellige Binär- oder Dualcodierung** von z .

Beispiel: (umgangssprachlich) die Binärzahl 010011 hat Wert 19

Verallgemeinerung: Ist $q > 1, N > 0$, so ist jedes $z \in \mathbb{N}, 0 \leq z < q^N$ eindeutig darstellbar als

$$z = z_{N-1} \cdot q^{N-1} + z_{N-2} \cdot q^{N-2} + \dots + z_1 \cdot q + z_0 \cdot 1$$

mit $z_0, \dots, z_{N-1} \in \{0, 1, \dots, q-1\}$ (**N-stellige q-adische Darstellung**)

Beispiel: Dezimalzahl 109 in Basis 7, 8 und 16

$$\begin{aligned} 109 &= 2 \cdot 7^2 + 1 \cdot 7 + 4 \cdot 1 &= 1 \cdot 8^2 + 5 \cdot 8 + 5 \cdot 1 &= 6 \cdot 16 + (13) \cdot 1 \\ &= (214)_7 &= (155)_8 &= (6D)_{16} \end{aligned}$$

Hexadezimal-Dual-Oktal

2.1-8

- Dualcodes sind lang, Beispiel: $\text{bin}_{16}(12345) = 0011000000111001$
- Einzelbit-Verarbeitung ineffizient, Prozessor verarbeitet *Bitblöcke*
Beispiel: **Byte** = Block aus 8 Bits (**Oktett**)

In der Informatik übliche Codierungen und Schreibweisen

die Präfix-Varianten (0 . . .) sind in Java erlaubte
Schreibweisen für ganzzahlige Werte

dual (Basis 2) 0b1101101 (synonym: binär)

oktal (Basis 8) 0155 (oder auch 155o)

hexadezimal (Basis 16) 0x6D (oder auch #6D), Alphabet {0, 1, ..., 9, A, ..., F}

Codebücher aus Bitgruppen

Triplets

Quartetts

0 ↳ 000	4 ↳ 100
1 ↳ 001	5 ↳ 101
2 ↳ 010	6 ↳ 110
3 ↳ 011	7 ↳ 111

0 ↳ 0000	4 ↳ 0100	8 ↳ 1000	C ↳ 1100
1 ↳ 0001	5 ↳ 0101	9 ↳ 1001	D ↳ 1101
2 ↳ 0010	6 ↳ 0110	A ↳ 1010	E ↳ 1110
3 ↳ 0011	7 ↳ 0111	B ↳ 1011	F ↳ 1111

hexadezimal ↪ **dual** ersetze Hexziffern durch Quartetts

Beispiel: 0x6D ↳ 0110|1101 ↳ 0b01101101

dual ↪ **hexadezimal** Bitmuster *von rechts* in Quartetts einteilen und durch Hexziffern ersetzen (vorn ggf. 0-Bits auffüllen)

Beispiel: 0b11011 ↳ 0001|1011 ↳ 0x1B

dual ↔ **oktal** analog: Oktalziffern durch Triplets ersetzen bzw. Bitmuster *von rechts* in Triplets einteilen und oktal ersetzen

Beispiel: 0b11011 ↔ 011|011 ↔ 33o

2.1-9 Codierung ganzer Zahlen

Vorzeichen-Betragsdarstellung (naheliegend, aber ineffizient)

ein Bit fürs Vorzeichen, $N - 1$ Bits für Betrag Nachteile:

- mit N Bits so nur $2^N - 1$ verschiedene Werte darstellbar
- Fallunterscheidungen bei Addition/Subtraktion

Zweierkomplement-Darstellung (kurz: 2K-Codierung)

- 2^{N-1} Bitmuster für 0 und die positiven Zahlen: 0, 1, ..., $2^{N-1} - 2$, $2^{N-1} - 1$
- übrige Bitmuster für die negativen Zahlen: -2^{N-1} , $-2^{N-1} + 1$, ..., -2 , -1
- Beispiel: Codebuch bei $N = 4$

0 ↳ 0000	4 ↳ 0100	-8 ↳ 1000	-4 ↳ 1100
1 ↳ 0001	5 ↳ 0101	-7 ↳ 1001	-3 ↳ 1101
2 ↳ 0010	6 ↳ 0110	-6 ↳ 1010	-2 ↳ 1110
3 ↳ 0011	7 ↳ 0111	-5 ↳ 1011	-1 ↳ 1111

Formeln für das Zweierkomplement

2.1-10

Decodierformel des Zweierkomplementsist $\text{bin}_N^K(z) = z_{N-1}z_{N-2}\dots z_1z_0$ die 2K-Codierung einer Zahl, so gilt

$$z = (-1) \cdot z_{N-1} \cdot 2^{N-1} + z_{N-2} \cdot 2^{N-2} + \dots + z_1 \cdot 2 + z_0 \cdot 1$$

Codierformel des Zweierkomplementsumgekehrt gilt für $-2^{N-1} \leq z < 2^{N-1}$

$$\text{bin}_N^K(z) := \begin{cases} \text{bin}_N(z) & , \text{ falls } z \geq 0 \\ \text{bin}_N(z+2^N) & \text{sonst} \end{cases}$$

Einfache Rechenregeln

2.1-11

Vorzeichenregel

das erste Bit der 2K-Codierung zeigt das Vorzeichen an:

- erstes Bit = 1 \Rightarrow Zahl ist negativ
- erstes Bit = 0 \Rightarrow Zahl ist nichtnegativ (0 oder positiv)

Negationsregel $\text{bin}_N^K(-z)$ erhält man aus $\text{bin}_N^K(z)$ durch:

- Negation aller Bits ($z \mapsto \bar{z}$: aus 0 wird 1, aus 1 wird 0), dann
- Addition von 1 (ggf. überzähliges Übertragsbit ignorieren)

Beispiele $3 = (0011)_2 \mapsto \overline{0011} = 1100 \mapsto (1101)_2 = -3$ $0 = (0000)_2 \mapsto \overline{0000} = 1111 \mapsto 10000 \mapsto (0000)_2 = 0$ Binärarithmetik

2.1-12

Arithmetik nach Schulmethode

- Addition:
“eins+eins ergibt 0, eins gemerkt ...”
- Subtraktion auf Addition zurückgeführt:

$$a - b = a + (-b) = a + \bar{b} + 1$$

- Multiplikation = Mehrfachaddition,
Division = wiederholte Subtraktion^a
- einfache Hardware-Realisierung

Leibniz' Traktat zur Binärdarstellung

^asubtrahiere solange Ergebnis ≥ 0 bleibt, letztes Ergebnis = Divisionsrest, Iterationszahl = ganzzahliger Quotient

Multiplikation mit Zweierpotenz 2^ℓ entspricht Linksshift um ℓ PositionenNotation: $<<\ell$ Beispiel: $7 \cdot 2^2 = (.000111)_2 << 2 = (.011100)_2 = 28$ **ganzzahlige Division durch Zweierpotenz 2^ℓ** entspricht Rechtsshift um ℓ PositionenNotation: $>>\ell$ Beispiel: $7 / 2^2 = (.000111)_2 >> 2 = (.000001)_2 = 1$ 7 mod 2^2 entspricht den “herausgeschobenen” Ziffern, hier: $(..11)_2 = 3$

Merkmale des Datentyps int

2.1-13

Wertebereich ganze Zahlen im Intervall

$$[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$$

Einige Operationen $+, *, -, /, \%$ (Addition, Multiplikation, Subtraktion, Division, Divisionsrest) [ACHTUNG: stets ganzzahlige Division!, Bsp.: $7/2 == 3$]

Literele Ziffernfolgen, die Zahlen im `int`-Intervall beschreiben

Interpretation normalerweise dezimal, mit entsprechendem Präfix dual, oktal oder hexadezimal (siehe Paragraph 2.1-10)

Symbolische Konstanten `Integer.MIN_VALUE` / `Integer.MAX_VALUE` für den kleinsten/größten `int`-Wert

interne Codierung 4 Byte-Zweierkomplement

Weitere Ganzahl-Datentypen

`long, short, byte`: Zweierkomplement-Darstellung mit 8, 2 bzw. 1 Byte \ (ansonsten analog)

2.1-14 Reelle Zahlen und Gleitkomma-Zahlen

- geometrische Deutung reller Zahlen: Punkte auf dem Zahlenstrahl
- numerische Darstellung: *unendliche* Dezimalbrüche Beispiele: Kreiszahl $\pi = 3.1415\dots$, Eulersche Zahl $e = 2.718\dots$
- bei fester Byteanzahl nur endlich viele verschiedene Werte \Rightarrow nicht beliebig große/beliebig genaue Beträge möglich
- **Gleitkommazahlen** geben endlich viele Ziffern und eine Größenordnung an Beispiel: Taschenrechner zeigt bei 2^{128} die *dezimale* Gleitkommazahl $3.402823669E + 38$ (für $3,402823669 \times 10^{38}$)
- die rechnerinterne Darstellung beruht auf *dualen* Gleitkommazahlen

$$\begin{aligned} z &= \sum_{i=-M}^{N-1} z_i \cdot 2^i = \sum_{i=0}^{N-1} z_i \cdot 2^i + \sum_{i=-M}^{-1} z_i \cdot 2^i \\ &= \text{ganzer Anteil } \lfloor z \rfloor + \text{echt gebrochener Anteil } \{z\} \end{aligned}$$

- Achtung: Manch “einfache” Zahl hat keine endliche Dualbruchentwicklung! Beispiel: $0.2 = (0.001\bar{1})_2$

2.1-15 Duale Gleitkommazahlen (auch: binäre Gleitkommazahlen genannt) sind Zahlen der Form

$$z = s \cdot 2^e \cdot m,$$

mit **Vorzeichen** $s \in \{+1, -1\}$ und

- **Mantisse** $m = \sum_{i=-M}^{N-1} b_i \cdot 2^i$ (für die Genauigkeit) und
- **Exponent** $e \in [e_{\min}, e_{\max}]$ (für die Größenordnung)

[Normalisierte Darstellung](#)

2.1-16

Beobachtung

Für binäre GK-Zahlen sind verschiedene Darstellungen möglich:

$$\text{Beispiel: } -12 = (-1) \cdot 6 \cdot 2^1 = (-1) \cdot 3 \cdot 2^2 = (-1) \cdot 1.5 \cdot 2^3 = \dots$$

aber nur eine mit Mantisse m im Intervall $1 \leq m < 2$.

Darstellung $s \cdot 2^e \cdot m$ heißt **normalisiert**

falls die Mantisse folgende Form hat:

$$m = (1.b_1b_2\dots b_M)_2 = 1 + \sum_{i=-1}^{-M} b_i \cdot 2^i.$$

Bei normalisierter Darstellung betrachte die **reduzierte Mantisse**
 $m' = m - 1 = (.b_1b_2\dots b_M)_2$.

Sonderfall

Die Zahl 0 hat keine normalisierte Darstellung.

[Gleitkommazahlen nach IEEE-Standard 754](#)

2.1-17

GK-Zahlen einfacher Genauigkeit (Datentyp **float)**

werden durch 4 Bytes codiert:

- 1 Bit für das Vorzeichen s ($1 \hat{=} -1, 0 \hat{=} +1$)
- 8 Bits für den Exponenten $e \in \{-126..127\}$
- 23 Bits für die reduzierte Mantisse, die durch $\text{bin}_{23}(m')$ codiert wird

Z.B. wird Exponent e codiert durch das Bitmuster
 $\text{bing}(e + 127)$. Die Codierung im Detail ist nicht
klausurrelevant.

$$\text{Beispiel: } -(5.125)_{10} = -(101.001)_2 = (1.01001)_2 \cdot -2^2$$

0	1	9	32
	1 0000001	010010000000000000000000	
V.	Exponent	reduzierte Mantisse	

GK-Zahlen doppelter Genauigkeit (Datentyp **double)**

11 Bits für den Exponenten $e \in \{-1022, \dots, 1023\}$, 52 Bits für die reduzierte Mantisse

[Wertebereiche](#)

2.1-18

Vorbe trachtung

- Mantissen im Bereich $(1.00\dots)_2 \dots (1.11\dots)_2$
- Exponenten im Bereich $-126 \dots 127$ (bzw. $-1022 \dots 1023$)
- nützliche Formel $2^{10} = 1024 \approx 1000 = 10^3$, also $2^n \approx 10^{0.3n}$

Abschätzung

- größter float-Betrag (`Float.MAX_VALUE`):
 $(1.11..1)_2 \cdot 2^{127} \approx 2 \cdot 2^{127} = (2^{10})^{12.8} \approx 10^{3 \cdot 12.8} \approx 10^{38}$
- kleinster *positiver* float-Wert (`Float.MIN_VALUE`):
 $1 \cdot 2^{-126} = (2^{-10})^{12.6} \approx 10^{-3 \cdot 12.6} \approx 10^{-38}$
- analog für double (`Double.MAX_VALUE`, `Double.MIN_VALUE`):
 $2^{-1022} \dots 2^{1023} \approx 10^{-308} \dots 10^{308}$

2.1-19 Frei bleibende Bitmuster

im normalisierten Fall nicht auftretende Bitmuster werden wie folgt verwendet:

Exponentenblock 00..0

- steht in Kombination mit Mantissenblock 00..0 für die Zahl 0 (bzw. ± 0 je nach Vorzeichenbit)
- ansonsten werden sogenannte *denormalisierte Werte* dargestellt (nicht klausurrelevant)

Exponentenblock 11..1

- in Kombination mit Mantissenblock 000..0: Codierung für $\pm\infty$ (je nach Vorzeichenbit)
- sonst: **NaN** (*not a number*) = Sonderwerte, die z.B. nicht definierte Ergebnisse einer Operation anzeigen
- entsprechende symbolische Konstanten:

```
Double.POSITIVE_INFINITY
Double.NEGATIVE_INFINITY
Double.NaN
```

2.1-20 Bemerkungen zur Gleitkomma-Arithmetik

Verarbeitung von Gleitkomma-Operationen im Rechenwerk:

Multplikation/Division	Addition/Subtraktion
1. multipliziere/dividiere Mantissen 2. addiere/subtrahiere Exponenten 3. normalisiere Ergebnis	1. bringe Operanden auf gleichen (höheren) Exponenten 2. addiere/subtrahiere Mantissen 3. normalisiere Ergebnis

Beispiele

$$(1.11E+1)_2 \cdot (1.11E+2)_2 = ?$$

$$(1.11E+1)_2 + (1.11E+2)_2 = ?$$

$$(1.1)_2 \cdot (1.1)_2 = (11.0001)_2$$

$$(0.111E+2)_2 + (1.11E+2)_2 = (10.101E+2)_2$$

$$(1.11E+1)_2 \cdot (1.11E+2)_2 = (11.0001E+3)_2$$

$$= \underline{(1.1001E+4)}_2$$

$$= \underline{(1.0101E+3)}_2$$

2.1-22 Gleitkomma-Über- und Unterlauf

- **Überlauf:** Rundung zu $\pm\infty$
- **Unterlauf:** Rundung zur Null

Merke

Gleitkomma-Überlauf/-Unterlauf erzeugen keine Laufzeitfehler, aber Informationsverlust.

Beispiel

```
double x = Double.MAX_VALUE;
x = 2*x;
System.out.println(x); // Textausgabe: 'Infinity'
double y = Double.MIN_VALUE; // kleinste darstellbare POSITIVE(!) Zahl
y = y/2;
System.out.println(y); // Textausgabe: '0'
```

Merkmale der Datentypen float und double

2.1-23

Wertebereiche ca. $[-10^{38}, -10^{-38}] \cup \{0\} \cup [10^{-38}, 10^{38}]$ bzw.
 $[-10^{308}, -10^{-308}] \cup \{0\} \cup [10^{-308}, 10^{308}]$ (und Sonderwerte)

einige Operatoren $+, -, \cdot$ (Vorzeichen bzw. Addition/Subtraktion), $\cdot, /$
(Multiplikation)

interne Codierung 4 bzw. 8 Bytes nach dem IEEE 754-Standard

Literele z.B. dezimale¹ Ziffernfolgen mit Dezimalpunkt oder/und Exponent², die Werte im o.g. Bereich darstellen (für float ist f oder F angehängt)

- Beispiel: 3.1415, 31.415E-1=31.415e-1=31,41 · 10^{-1}
- Beispiel: 3.1415f, 31.415E-1f, 31.415e-1f
- Gegenbeispiel: 3.1415E400 (zu groß)

→ Info1-Sammlung (11-ID: nin33tb1v2i0)

2.1.3 Datentypen für Zeichen und Wahrheitswerte**Zeichen**

2.1-24

Zeichen sind die einzelnen unterscheidbaren Symbole, aus denen Text zusammengesetzt ist

- Alphabet = endliche Menge von Zeichen
- **Zeichensatz** = Alphabet mit vereinbarter Nummerierung

engl. character set

Klarstellung: Zeichensatz \neq Font!

Beispiel: A und Å stellen beide das Zeichen “Groß-Ah” dar

Universelles Codierprinzip

2.1-25

¹hexadezimale GK-Literele gibt es auch

²der sich auf Basis 10 bezieht

- **Zeichenwert** = Nummer des Zeichens im Zeichensatz
- **Zeichencode** = Binärkode des Zeichenwerts

$$Z_i \mapsto \text{bin}_N(i),$$

für genügend großes N .

k-Byte Codierung

Zeichen $Z_i \mapsto$ Zeichencode $\text{bin}_{8k}(i)$,

Zeichencode $\hat{=} 2^k$ Hexziffern wobei $2^{8k} \geq$ Größe des Zeichensatzes

Beispiel: ASCII (k = 1)

Zeichen	#	\$...	0	1	...	9	...	a	...
Nummer	35	36	...	48	49	...	57	...	97	...
Code	0010 0011	0010 0100	...	0011 0000	0011 0001	...	0011 1001	...	0110 0001	...
	0x23	0x24	...	0x30	0x31	...	0x39	...	0x61	...

2.1-26 American Standard Code for Information Interchange (1963)

ASCII-Zeichensatz umfasst $2^7 = 128$ Zeichen

- 95 alphanumerische Zeichen (**druckbar**)
 - 53 **Buchstaben** (je 26 Groß-/Kleinbuchstaben + Unterstrich _), 1 Leerzeichen, 10 Dezimalziffern
 - 31 Sonderzeichen: ; . , { } [] () < > = ~ ! + - * / % ^ & | ? : \ # @ \$ ' ` "
 - und 33 **Steuerzeichen** (nicht druckbar) zur Steuerung der Ausgabe und sonstigen Signalisierung, Beispiele: Zeilenwechsel, Tabulatorsprung

ASCII-Codebuch

Shell-Kommando `man ascii`

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3...	0	1	2	3	4	5	6	7	8	9	:	<	=	>	?	
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[\	^	_	
6...	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

der Zeichensatz (Grafik aus Wikipedia) und seine Codierung wurden 1963 standardisiert

2.1-27 ASCII-Erweiterungen

Um weitere Zeichen aufzunehmen, wurden verschiedene (untereinander inkompatible) Symbolbelegungen für die Zeichenwerte 128 bis 255 vorgeschlagen
weiteste Verbreitung fand ISO8859-1 (für westeuropäische Sprachen)

ISO8859-1

man iso-8859-1

Unicode

2.1-28

= in Entwicklung befindlicher Zeichensatz der Schriftzeichen aller bekannten Kulturen, Unicode 6.0 umfasst über 10^6 Zeichen

- ISO8859-1 ist eine Teilmenge (erste 256 Zeichen)
 - die Byte-Codierung der ersten $65536 = 2^{16}$ Unicode-Zeichen heißt **UTF-16**

16-Bit Unicode bietet Platz für alle international verbreiteten Sprachen

man unicode

Aus Zeile und Spalte sind die UTF-16-Codes abzulesen.

Java-Quellcode darf beliebige Unicode-Zeichen enthalten, ABER ...

wenn die beim Speichern der Datei verwendete Codierung vom Systemstandard abweicht, muss man sie per Kommandozeilenoption mitteilen

bei reinem ASCII-Quellcode ist das unnötig, daher die Empfehlung, nur ASCII-Zeichen zu verwenden.

```
javac -encoding UTF-16 Umlaute.java
```

→ Info1-Sammlung (I1-ID: nqx18v71v2i0)

Merkmale des Datentyps char

2.1-30

Wertebereich Zeichenwerte $0, \dots, 65535 = 2^{16} - 1$

Operatoren

- **Inkrement und Dekrement**, Beispiel: `c++` ersetzt `c` durch Zeichenwert des Alphabetnachfolgers, `c-` entsprechend durch den des Vorgängers

UTF-16 **interne Codierung** Zeichenwert i wird codiert durch $\text{bin}_{16}(i)$ (2 Bytes), Beispiel:
 $0x0041$ ist der Zeichencode für den Buchstaben A

Literale Einzelsymbol, eingeschlossen in Hochkommata,
 Beispiel: Literal 'A' gibt den Zeichenwert 65 an.

Bemerkung

Zeichen werden intern wie besondere ganze Zahlen (nämlich Ordnungszahlen) behandelt: anstelle der Zeichen selbst werden Zeichenwerte verarbeitet
 Darum gilt char als Ganzzahl-Datentyp. Oben nicht erwähnt: Sogar Operatoren wie +, * sind anwendbar, haben dann aber Ergebnis-Typ int (Mischtyp-Operatoren).

2.1-31 Wertvergleiche

Bei Operator-Verknüpfung ist das Ergebnis meist vom gleichen Typ wie die Operanden.

Beispiel: int-Ausdruck $7/2$ hat Wert 3, da Operanden vom Typ int sind

Mischtyp-Operatoren

verknüpfen Daten eines Typs und erzeugen Ergebnis eines anderen Typs

Beispiel: Vergleichoperatoren liefern Wahrheitswerte

Operator	Bedeutung	wahr	falsch
$==$	ist gleich	$2 == 2$	$2 == 3$
$!=$	ist ungleich	$2 != 3$	$2 != 2$
$<$	ist kleiner als	$2 < 3$	$2 < 2$
\leq	ist kleiner oder gleich	$2 \leq 2$	$3 \leq 2$
$>$	ist größer als	$3 > 2$	$2 > 2$
\geq	ist größer oder gleich	$2 \geq 2$	$2 \geq 3$

Bemerkung

$==$ und $!=$ sind anwendbar auf Werte beliebiger gleicher Datentypen und auf Werte beliebiger (auch verschiedener) numerischer Datentypen. $<$, $>$, \leq , \geq sind nur anwendbar auf Werte numerischer Datentypen.

2.1-32 Wahrheitswerte

Begriff

Eine Aussage kann wahr oder falsch sein.

Aussage	Wahrheitswert
$2 > 0$	wahr
$2 = 0$	falsch

- "wahr"/"falsch" sind Wahrheitswerte
- übliche Bezeichnungen: wahr / falsch, true / false, W / F, 1 / 0, ...
- logische Operationen = Verknüpfung von Wahrheitswerten durch logische Operationen wie UND, ODER, NICHT, ...
- bei Kontrollfluss-Steuerung mit if, else, while, ... beschreibt der Ausdruck innerhalb von $\langle\text{Bedingung}\rangle$ einen Wahrheitswert

siehe Kapitel ??

- diese Ausdrücke sind vom Typ **boolean**

[Merkmale des Datentyps boolean](#)

2.1-33

Wertebereich wahr / falsch

Operationen `&&`, `||`, `!`, `^` (für UND, ODER, NICHT, ENTWEDER ODER)

Litale `true` / `false`

Codierung `0xFF` / `0x00` (ein Byte)

[Zusammenfassung](#)

2.1-34

Die wichtigsten elementaren Datentypen von Java

Name	Information	Speicher	Operatoren
<code>boolean</code>	Wahrheitswerte (wahr/falsch)	1 Byte	<code>&&</code> , <code> </code> , <code>!</code>
<code>char</code>	16-Bit Unicode-Zeichen	2 Bytes	<code>++</code>
<code>int</code>	ganze Zahlen $\in \{-2^{31}, \dots, 2^{31} - 1\}$	4 Bytes	<code>+, -, *, usw.</code>
<code>double</code>	Zahlen mit Betrag $\leq 10^{308}$ und Sonderwerte	8 Bytes	<code>+, -, *, usw.</code>

Ergänzende Varianten

Name	Information	Speicher	Operatoren
<code>byte</code>	ganze Zahlen $\in \{-2^7, \dots, 2^7 - 1\}$	1 Byte	wie <code>int</code>
<code>short</code>	ganze Zahlen $\in \{-2^{15}, \dots, 2^{15} - 1\}$	2 Bytes	<code>dto.</code>
<code>long</code>	ganze Zahlen $\in \{-2^{63}, \dots, 2^{63} - 1\}$	8 Bytes	<code>dto.</code>
<code>float</code>	Zahlen mit Betrag $\leq 10^{38}$ und Sonderwerte	4 Bytes	wie <code>double</code>

Bemerkung: Bei `byte` und `short` haben die Operationsergebnisse Typ `int`.

Beobachtung

Speicherung eines elementaren Werts erfordert eine konstante Menge Speicher.

[Zum Vergleich: Zeichenketten](#)

2.1-35

- Zeichenketten (*Strings*) sind Folgen von Zeichen, also *strukturierte Informationen*
- werden bei Java implementiert durch den Datentyp `String`
 - `String` ist kein elementarer Datentyp, aber ähnlich bequem zu benutzen
 - `String` hat enge Beziehungen zu `char`

Wir bezeichnen elementare Datentypen sowie `String` als *integrierte Datentypen*.

Einige Merkmale des Datentyps `String` (später mehr)

Litale in Anführungszeichen eingeschlossene, einzeilige Zeichenfolgen

Beispiel: "Hallo\n Leute!"

Operatoren `+` (Verkettung), Beispiele:

String-Ausdruck	Wert
"Hallo, "+"Welt"	"Hallo, Welt"
"123"+ "456"	"123456"
"123"+ "+"+ "456"	"123+456"

Beobachtung

Die Speicherung einer Zeichenkette der Länge ℓ erfordert wenigstens $2 \cdot \ell$ Bytes, da jedes einzelne Zeichen 2 Byte erfordert.

2.2 Programmieren mit integrierten Datentypen

2.2.1 Rechnen mit Wahrheitswerten

2.2-1 Syntax von boolean-Ausdrücken

boolean-Ausdrücke sind entweder *atomar* (unzerlegbar) oder zusammengesetzt aus kleineren boolean-Ausdrücken

atomare boolean-Ausdrücke

- true, false, Vergleichsausdrücke sowie Aufrufe von Methoden mit boolean-Resultat

zusammengesetzte boolean-Ausdrücke

- $! \langle b\text{-Ausdruck} \rangle$ und $(\langle b\text{-Ausdruck} \rangle)$ sowie
- Ausdrücke der Gestalt $\langle b\text{-Ausdruck} \rangle \langle \text{zweist. } b\text{-Operator} \rangle \langle b\text{-Ausdruck} \rangle$

2.2-2 Die Semantik

... entspricht der Aussagenlogik

a	b	$!b$	$a \& b$	$a \mid b$	$a \wedge b$
false	false	true	false	false	false
false	true	false	false	true	true
true	false	true	false	true	true
true	true	false	true	true	false

Priorität wie in der Aussagenlogik

Beispiele:

$!a \mid\mid b$ ist äquivalent zu: $((!a) \mid\mid b)$

$a \mid\mid b \&\& c$ ist äquivalent zu: $(a \mid\mid (b \&\& c))$

boolean-Operatoren haben niedrigere Priorität als Vergleichsoperatoren

Beispiel: $7 > x \mid\mid 2 < c$ ist äquivalent zu: $((7 > x) \mid\mid (2 < c))$

2.2-3 Assoziativität

... bestimmt die Auswertungsrichtung

links-assoziativ: Auswertung von links nach rechts,

Beispiel:

`alter >= 17 && begleitet && !alkohol` ist äquivalent zu `((alter >= 17) && begleitet) && !alkohol)`

... ermöglicht *Lazy evaluation*

Auswertung eines boolean-Ausdrucks endet, sobald Wert fest steht:

aus $\langle \text{Ausdruck1} \rangle == \text{false}$ folgt $\langle \text{Ausdruck1} \rangle \&& \langle \text{Ausdruck2} \rangle == \text{false}$

aus $\langle \text{Ausdruck1} \rangle == \text{true}$ folgt $\langle \text{Ausdruck1} \rangle \mid\mid \langle \text{Ausdruck2} \rangle == \text{true}$

Beispiel:

```
if ( a/b < 2 && b > 0) ... verursacht Laufzeitfehler, falls b == 0
if ( b > 0 && a/b < 2) ... — kein Laufzeitfehler
```

Beispiel: Kompliziertere Bedingungstests

2.2-4

Schaltjahresregel des Gregorianischen Kalenders

Eine Jahreszahl entspricht einem Schaltjahr, wenn sie durch 4, jedoch nicht durch 100 teilbar ist oder aber durch 400 teilbar ist.

```
1 public class Schaltjahr {
2     public static void main(String[] args) {
3         int j = Integer.parseInt(args[0]);
4         boolean schalt = (j % 4 == 0); // Klammern nur fuer Lesbarkeit
5         schalt = schalt && (j % 100 != 0);
6         schalt = schalt || (j % 400 == 0);
7         if (schalt) System.out.println(j + " ist ein Schaltjahr.");
8         else System.out.println(j + " ist kein Schaltjahr.");
9     }
10 }
```

<http://www.stud.informatik.uni-goettingen.de/inf10/java/Schaltjahr.java>

Die Regel ließe sich auch als einziger boolean-Ausdruck formulieren, aber der Code würde unübersichtlicher.

Beispiel: Fallunterscheidungen

2.2-5

Drei Zahlen (Kommandozeilenargumente) aufsteigend geordnet ausgeben:

```
1 public class Ordne {
2     public static void main(String[] args) {
3         int a = Integer.parseInt(args[0]),
4             b = Integer.parseInt(args[1]),
5             c = Integer.parseInt(args[2]);
6         if (a<=b && b<=c) System.out.println(a+" "+b+" "+c);
7         else if (a<=c && c<=b) System.out.println(a+" "+c+" "+b);
8         else if (b<=a && a<=c) System.out.println(b+" "+a+" "+c);
9         else if (b<=c && c<=a) System.out.println(b+" "+c+" "+a);
10        else if (c<=a && a<=b) System.out.println(c+" "+a+" "+b);
11        else if (c<=b && b<=a) System.out.println(c+" "+b+" "+a);
12    }
13 }
```

<http://www.stud.informatik.uni-goettingen.de/inf10/java/Ordne.java>

Die if-else-Kaskade ist eine vollständige Fallunterscheidung.

→ Info1-Sammlung (11-ID: 1uy278z0m2i0)

2.2.2 Verarbeitung von Zeichen und Zeichenketten

2.2-6 Typischer Code

Zeichen auflisten

```

1 public class Alphabet {
2     public static void main(String[] args) {
3         for (char c = 'A'; c <= 'Z'; c++)
4             System.out.print(c);
5         System.out.println();
6     }
7 }
```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Alphabet.java>

Zeichen(-werte) einlesen/umwandeln/ausgeben

<pre> public class Char2Value { public static void main(String[] args) { while (!StdIn.isEmpty()) { char c = StdIn.readChar(); System.out.println(c+": "+(int) c); StdIn.readLine(); //..ignoriere Rest } } }</pre>	<pre> public class Value2Char { public static void main(String[] args) { while (!StdIn.isEmpty()) { int n = StdIn.readInt(); System.out.println(n+": "+(char) n); StdIn.readLine(); } } }</pre>
(int) in Zeile 5 erzwingt Umwandlung in die entsprechende Ordnungszahl, also den Zeichenwert. StdIn.readLine() sorgt dafür, dass nur ein Zeichen pro Eingabezeile verarbeitet wird (insbesondere nicht noch das mitgesendete Zeilenwechsel-Zeichen. Zum Verständnis: Zeile auskommentieren und ausprobieren.)	(char) in Zeile 5 erzwingt Umwandlung einer Ordnungszahl in das entsprechende Zeichen (genauer: Zahl wird als Zeichenwert interpretiert und in das entsprechende Zeichen wird umgewandelt). Das ist praktisch die Umkehrung zu (int). Beides sind sogenannte cast-Operatoren — später mehr dazu.

2.2-7 Zeichenklassifizierung

- Die Zeichen sind eingeteilt in verschiedene “Sorten”. Beispiel: Buchstaben, Ziffern, Steuerzeichen, ...
- Die Klassenbibliothek bietet Methoden zum Prüfen der Sorten

Beispiele

- `Character.isLetter(c) / Character.isDigit(c)` – Ist char-Wert c ein Buchstabe/eine Ziffer?
- `Character.isUpper(c) / Character.isLower(c)` –... Groß-/Kleinbuchstabe?
- `Character.isISOControl(c)` –... ein Steuerzeichen?
Bemerkung: alle anderen sind druckbare Zeichen
- `Character.isWhitespace(c)` –... Whitespace

→ Info1-Sammlung (II-ID: 3yacsp61m2i0)

2.2-8 Bemerkung zu Escape-Sequenzen

Besondere Zeichen können nur “mit Trick” im Quelltext angegeben werden:

- Steuerzeichen** (Beispiele: Zeilenwechsel, Tabulator, ...)
- für besondere Zwecke reservierte Zeichen (Beispiele: ‘, ”)

- nicht auf der Tastatur vorhandene Zeichen (Beispiel: ¡)
- im Quellcode ggf. zu vermeidende Zeichen (Beispiel: Umlaute)

Escape-Sequenz

= Ersatzzeichenfolge für nicht direkt angebares Einzelzeichen

Beispiele

\n, \t	Zeilenwechsel (Newline-Zeichen), Tabulator
\', \"	Hochkomma, Anführungszeichen
\\\	Backslash
\u00F6	ö (= Unicode-Zeichen mit Zeichencode 0x00F6)

→ Info1-Sammlung (II-ID: dgv4c791n2i0)

Werte durch Strings codieren („String-Umwandlung“)

2.2-9

Für die Ausgabe müssen z.B. Zahlen als Strings dargestellt werden.

Einige vorgefertigte `toString`-Methoden

Methode	Erklärung
<code>Integer.toString(int x)</code>	Dezimaldarstellung als String
<code>... toBinaryString ...</code>	Dualdarstellung als String (analog: ..Hex..., ..Octal...)
<code>Double.toString(double x)</code>	Dezimaldarstellung als String
<code>Double.toHexString(double x)</code>	Hexadezimaldarstellung als String
<code>Character.toString(char c)</code>	wandelt Zeichenwert um in String (der Länge 1)

Implizite String-Umwandlung

- `System.out.println(Math.PI);` ist gleichwertig zu
`System.out.println(Double.toString(Math.PI));`
- Umwandlung geschieht automatisch bei Verkettung mit Strings
`String s = "Anzahl: " + x, preis = p + " Euro";`

Strings als Werte decodieren

2.2-10

Bei der Eingabe werden Strings z.B. als Zahlen interpretiert.

Beispiel: `StdIn.readInt()`

Einige vorgefertigte Decodier-Routinen

Methode	Bsp. für Decodierung Arg.string → Wert
<code>int Integer.parseInt(String)</code>	"42" → int-Wert 42
<code>double Double.parseDouble(String)</code>	"2e15" → double-Wert für $2 \cdot 10^{15}$
<code>boolean Boolean.parseBoolean(String)</code>	"true" → boolean-Wert true

2.2.3 Arithmetik**Typische Codebeispiele**

2.2-11

Werte akkumulieren

```
int sum = 0; // (additiver) Akkumulator
for (int i = 1; i <= N; i++)
    sum = sum + i;
// jetzt: sum == 1 + 2 + ... + N
```

```
int p = 1; // (multiplikativer) Akkumulator
for (int i = 1; i <= N; i++)
    p = 10*p;
// jetzt: p == N-te Potenz von 10
```

Arithmetische Ausdrücke

```
int s = N*(N+1)/2; // Gauss-sche Summenformel: s == sum (s.o.)
int last = N % 10; // letzte (Dezimal-)Ziffer von N
```

2.2-12 Routine: Dezimalzahlen einlesen mit `StdIn.readInt()`

Decodierung der Dezimaldarstellung

gegeben Ziffernfolge $z_{N-1}, z_{N-2}, \dots, z_0 \in \{0, 1, \dots, 9\}$

gesucht die Zahl $z = \sum_{i=0}^{N-1} z_i 10^i$

Aufwand bei naiver Umsetzung der Formel

- berechne jeweils $z_i \cdot 10^i$ und summiere auf
- insgesamt $0 + 1 + 2 + \dots + N - 1 = \frac{(N-1) \cdot N}{2} \approx N^2/2$ Multiplikationen und N Additionen \Rightarrow **quadratischer Aufwand**

2.2-13 Effizienter: Das Horner-Schema

Initialisiere $z \leftarrow 0$

Iteriere solange noch eine Ziffer f folgt:

- $z \leftarrow z \cdot 10 + f$

Terminiere: liefere Wert z

Aufwandsabschätzung

- für jede Ziffer eine Multiplikation (außer am Ende) und eine Addition
- insgesamt $N - 1$ Multiplikationen und N Additionen \Rightarrow nur **linearer Aufwand**

Implementierung

2.2-14

```

1 public class DecimalDecode {
2     public static void main(String[] args) {
3         int z = 0; // Akku
4         char c;
5         do {
6             c = StdIn.readChar();
7             if ('0' <= c && c <= '9') {
8                 z = 10*z;
9                 if (c == '1') z = z +1;
10                ...
11                if (c == '9') z = z +9;
12            }
13        } while (c != '\n');
14        System.out.println(z);
15    }
16}

```

<http://www.stud.informatik.uni-goettingen.de/info1/java/DecimalDecode.java>

Da der Wert von Ziffer z gerade deren "Entfernung" im Zeichensatz zur 0 ist, genügt folgendes anstelle der Zeilen 9ff:

z = z + (c - '0');

Nachtrag: Schreibvereinfachungen

2.2-16

$x++$; bzw. $x--$; sind gleichbedeutend zu $x = x + 1$; bzw. $x = x - 1$;

Weitere Kurzformen (*zusammengesetzte Zuweisungen*)

Kurzform	Langform	Bemerkungen
$x += y;$	$x = x + y;$	(bei String-Variable x : hänge y an)
$x -= y;$	$x = x - y;$	
...	...	analog für $*$, $/$, $\%$
$x \&= y;$	$x = x \&& y;$	
...	...	analog für weitere boolean-Operatoren

Dualcodierung in umgekehrter Reihenfolge

2.2-17

Speicherung natürlicher Zahlen beruht auf Dualcodierung. Einfach zu ermitteln:

Iteriere solange $z \neq 0$

- gib $z \bmod 2$ aus
- $z \leftarrow \lfloor z/2 \rfloor$

$\lfloor \cdot \rfloor$ = Abrundung zu nächster Ganzzahl $\lfloor z/2 \rfloor$ ist
ganzzahliger Quotient von $z : 2$

Problem : Die Dualziffern werden *in umgekehrter Reihenfolge* ausgegeben!

Ausweg : Ziffern vor der Ausgabe in einem String sammeln!

Dualcodierung

2.2-18

Nachprogrammierung von `Integer.toBinaryString` (siehe Paragraph 2.2-10):

Eingabe : $z \in \mathbb{N}$, **Ausgabe :** Dualdarstellung von z

Initialisiere $s \leftarrow \epsilon$ (leeres Wort)

Iteriere solange $z \neq 0$:

- ergänze s am Anfang um die Ziffer $z \bmod 2$
- $z \leftarrow \lfloor z/2 \rfloor$

Terminiere Textausgabe von s

Java-Code und Trace-Tabelle

```

1  public class DualCode {
2      public static void main(String[] args) {
3          int z = StdIn.readInt();
4          String s = ""; // Akku
5          do { s = (z % 2) + s;
6              z /= 2;
7          } while (z != 0);
8          System.out.println(s);
9      }
10 }

.../java/DualCode.java
Zeile 5: letzte Dualziffer z mod 2 dem String voranstellen

```

Zeile	<i>z</i>	<i>s</i>
5	87	1
6, 7	43	
5	43	11
6, 7	21	
5	21	111
6, 7	10	
5	10	0111
6, 7	5	
5	5	10111
6, 7	2	
5	2	010111
6, 7	1	
5	1	1010111
6, 7	0	

2.2-19 Ganzahlüberlauf

Erinnerung: Eigenschaften von `int`

- endlicher Wertebereich: $-2^{31}, \dots, 2^{31} - 1$
- `int` ist **abgeschlossen**: Ergebnisse arithmetischer Operationen mit `int`-Werten sind wieder `int`-Werte (Ausnahme: Division durch 0)

Beispiele

- Endlosschleife:

```

int i = 0;
while (i <= Integer.MAX_VALUE) {
    i++;
    ...
}

```

Quelltext am Ende von Paragraph 1.3-10

- `java PowersOfTwo N (N > 32)` generiert $1, 2, 4, \dots, 2^{30}, -2^{31}, 0, 0, \dots$

Keine automatischen Warnungen \Rightarrow Programmierer müssen Plausibilität prüfen!2.2-20 Ergänzung: Datentyp `long`näherungsweise: $\pm 8 \cdot 10^{18}$

- interne Codierung: 2K-Darstellung mit 8 Byte Länge,
- Ganzahlen $\in \{-2^{63}, \dots, 2^{63} - 1\}$ darstellbar (z.B. Produkt von `int`-Werten)
- Literale: dezimale Ziffernfolgen, gekennzeichnet mit angehängtem `L` (oder `l`)

2.2-21 Restberechnung bei negativen Zahlen

Definitionsgleichung

$$(a / b) * b + a \% b == a$$

Beispiel: Wegen $-14/3 == -4$ und $14/-3 == -4$ folgt
 $(-14) \% 3 == -2$ und $14 \% (-3) == 2$

Beispiel: ggT von -12 und 30??

- Der klassische Euklidische Algorithmus liefert -2147483638.
- Die „moderne“ Variante liefert den richtigen Wert, nämlich 6.

Restberechnung mit negativen Zahlen vermeiden

- anstelle $a \bmod b$ genügt meist $|a| \bmod |b|$

2.2.4 Einfache numerische Berechnungen

Beispiel: Dualbruchentwicklung gebrochener Zahlen

2.2-22

gegeben $z \in [0, 1)$ (echt gebrochene Zahl)

gesucht Folge der Dualziffern $\text{bin}(z) = z_{-1}z_{-2}z_{-3}\dots$ von z

Beobachtung

$$\begin{aligned} z &= z_{-1} \cdot 2^{-1} + z_{-2} \cdot 2^{-2} + z_{-3} \cdot 2^{-3} + \dots \\ \Rightarrow 2z &= z_{-1} \cdot 2^0 + z_{-2} \cdot 2^{-1} + z_{-3} \cdot 2^{-2} + \dots \end{aligned}$$

\Rightarrow Falls $2z \geq 1$ (Übertrag bei Multiplikation), so ist $z_{-1} = 1$, sonst 0.

Lemma

Für $0 \leq z < 1$ gilt $\text{bin}(z) = \begin{cases} 1\text{bin}(2z - 1), & \text{falls } 2z \geq 1 \\ 0\text{bin}(2z), & \text{sonst.} \end{cases}$

```

1 public class DualFraction {
2     public static void main(String[] args) {
3         double z;
4         System.out.print("0.");
5         do {z *= 2;
6             if (2*z >= 1) { System.out.print("1"); z = 2*z - 1; }
7             else { System.out.print("0"); z = 2*z; }
8         } while (z > 0);
9         System.out.println();
10    }
11 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/DualFraction.java>

Beispiel

$$(0.703125)_{10} = (0.101101)_2$$

Beispiel: Quadratwurzel-Berechnung mit dem Newton-Verfahren

2.2-23

Gegeben Zahl $a \geq 0$

Gesucht Näherung r für \sqrt{a}

Definition: \sqrt{a} = Seitenlänge eines Quadrats mit Fläche a

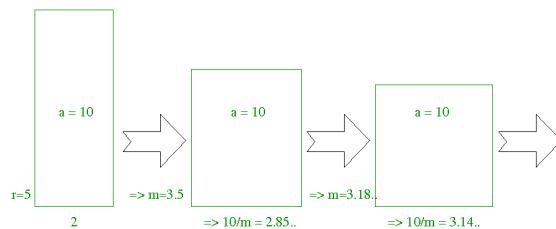
Dieser Spezialfall des sehr allgemeinen Newton-Verfahrens ist als Heron-Verfahren oder babylonisch-sumrisches Wurzelziehen bekannt.

Initialisiere Rechteck mit Seitenlängen $r \leftarrow a, q \leftarrow 1$ (\Rightarrow Fläche a)
Genauigkeitsschranke $\varepsilon > 0$

Iteriere solange $| \frac{a}{r^2} - 1 | > \varepsilon$, ersetze durch “quadratischeres” Rechteck:

- setze $r \leftarrow m := \frac{r+a/r}{2}$ (Mittelwert) und $q \leftarrow a/m$ (Flächeninhalt bleibt)

Terminiere liefere Näherungswert r



Beobachtung

Wiederholungsbedingung $|1 - \frac{a}{r^2}| > \varepsilon$ ist äquivalent zu $|a/r - r| > \varepsilon \cdot r$ (sofern $r \neq 0$).

```
public class Sqrt {
    public static void main( String[] args ) {
        double a = Double.parseDouble(args[0]);
        double epsilon = 1e-15;
        double r = a;
        while (Math.abs( r-a/r ) > epsilon * r) {
            r = (r + a/r)/2.0; // r <- Mittelwert von r und a/r
        }
        System.out.println(r);
    }
}
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Sqrt.java>

2.2-24 Konvergenzverhalten

Fakt

Jede Iteration verdoppelt die relative Genauigkeit (= übereinstimmende Stellen von $\frac{\sqrt{a}}{r}$ und 1). Sie entspricht in etwa der Anzahl korrekter Nachkommastellen.

Beispiel

Um mit Startwert 1 den Wert $\sqrt{2}$ auf 15 Dezimalstellen genau zu berechnen, genügen 5 Iterationen.

2.2-25 Beispiel: Exponentialfunktion mit Potenzreihe berechnen³

Reihendarstellung (siehe Kapitel ??)

³nach [Sedgewick/Wayne], S. 104ff.

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots \text{ für alle } x$$

Ein Fixpunkt-Algorithmus

- Summiere $1 + \frac{x}{1} + \frac{x^2}{2!} + \dots + \frac{x^i}{i!} + \dots$ solange es noch Änderungen gibt

Ist $|\frac{x^i}{i!}|$ genügend klein, kommt es zum Unterlauf: die Summe ändert sich nicht mehr.

Ansatz zur Termberechnung

```
double nenner = 1., zaehler = 1.;
for (int i = 1; i <= n; i++) zaehler = zaehler * x;
for (int i = 1; i <= n; i++) nenner = nenner * i;
double term = zaehler/nenner;
```

- zwei Schleifen für so ähnliche Berechnungen sind umständlich und ineffizient
- Gefahr von Ungenauigkeiten/Überläufen: z.B. 100! ist zu groß für double

Geschicktere Implementation

2.2-26

Zusammenfassung beider Schleifen

<pre>double term = 1.; for (int i = 1; i <= n; i++) term = term * x/i;</pre>
Einbettung in Summenschleife
<pre>double term = 1.; double sum = 0.; // Akkumulator for (int n = 1; sum != sum + term; n++) { //Stopp bei Unterlauf sum = sum + term; term = 1.; for (int i = 1; i <= n; i++) term = term * x/i; }</pre>
Nachteil: Bei jeder Iteration der äußeren Schleife werden die Terme neu berechnet.

Kombination in einer einzigen Schleife

<pre>double term = 1.; double sum = 0.; // Akkumulator for (int n = 1; sum != sum + term; n++) { //Stopp bei Unterlauf sum = sum + term; term = term * (x/n); }</pre>
http://www.stud.informatik.uni-goettingen.de/infol/java/Exponential.java

Bemerkung zur Division durch 0

2.2-27

Führt zu Laufzeitfehler bei Ganzzahloperanden

```
int x = StdIn.readInt(), y = StdIn.readInt(); // Eingaben: 17, 0
int d = x / y; // Laufzeitfehler!
int r = x % y; // wuerde auch Laufzeitfehler bewirken!
```

Führt NICHT zu Laufzeitfehler bei Gleitkommaoperanden

```
double x = StdIn.readDouble(), y = StdIn.readDouble(); // Eingaben: 17, 0
double d = x / y; // Wert: Double.POSITIVE_INFINITY
double r = x % y; // Wert: Double.NaN
```

2.2-28 Einige mathematische Funktionen und Konstanten

Typische Codebeispiele

```
double x1 = -b/(2*a)+Math.sqrt(b*b-4*a*c)/2*a; // Mitternachtsformel
double u = 2*Math.PI*radius; // Kreisumfang
```

`public final class Math`

Signatur ^a	Erklärung	Bemerkung
double abs(double a)	Absolutbetrag von a	auch für int, long, float
double max(double a, double b)	Maximum von a und b	min analog
double sin(double theta)	Sinusfunktion ^b	analog cos, tan ^c
double exp(double a)	Exponentiellefunktion e^a	
double log(double a)	natürlicher Logarithmus	
double pow(double a, double b)	Potenz a^b	
long round(double a)	Rundung	
double random()	Zufallszahl $\in [0, 1]$	
double sqrt(double a)	Quadratwurzel \sqrt{a}	
double PI	Kreiszahl π (Näherung)	
double E	Eulersche Zahl e (Näher.)	

^adie „Schnittstelle“, d.h. Name der Methode, sowie Argument- u. Ergebnis-Typ

^btheta in Bogenmaß, Umrechnung:

double x = Math.toRadians(90); (Umkehrung: toDegrees)

^cUmkehrfunktionen: asin, acos, atan

- Funktionen meist von der Gestalt: double-Wert(e) \mapsto double-Wert
- Ausnahmen: Math.round, Math.random (kein Argument), Math.abs (Varianten für int, long, float)

2.2.5 Typanpassungen

2.2-29 Notwendigkeit von Typanpassungen

Arithmetische Operationen

Werte beliebiger numerischer Datentypen sind verknüpfbar mit +, *, -, /, %:

$$\langle \text{Operand1} \rangle \langle \text{Operator} \rangle \langle \text{Operand2} \rangle$$

Verarbeitung durch Rechenwerk

Nur Werte gleichen Typs werden verarbeitet! Ergebnis hat dann auch diesen Typ. Vorgesetzte Typanpassung ermöglicht auch Werte unterschiedlicher Typen zu verknüpfen. Es gibt implizite und explizite Typanpassung.

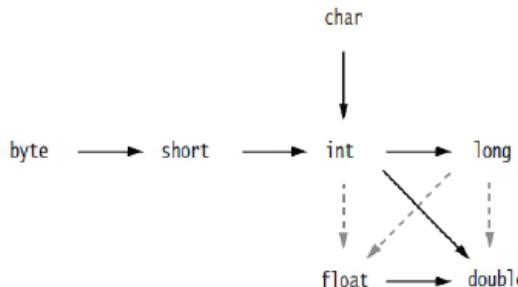
Implizite Typanpassung

Wenn der Ausgangswert im wesentlichen verlustfrei im Zieltyp dargestellt werden kann, veranlasst der Compiler diese Schritte *automatisch*:

- Operanden verschiedener Typen in gleichen Typ umwandeln, d.h. umcodieren (**Operanden-Typanpassung**)
- Ergebnis berechnen und als Wert dieses Typ codieren (**Ergebnis-Typanpassung**)

Informationsverluste sind nur durch Rundung bzw. Über-/Unterlauf möglich.

Promotion: Operanden-Typanpassung immer in Richtung des höherwertigen Typs



- Ganzzahl \rightarrow Ganzzahl: führende Nullbytes ergänzen,
Beispiel: `1 + 1L == 2L`
- Gleitkomma \rightarrow Gleitkomma: Exponent/Mantisse durch Nullen ergänzen,
Beispiel: `1.0f + 1.0 == 2.0`
- Ganzzahl \rightarrow Gleitkomma: komplizierter, Informationsverluste durch Rundung(!) möglich,
Beispiel: `Integer.MAX_VALUE + 0.0f == 2.14748365e9`
- char \mapsto Ganzzahl: Ordnungszahl des Zeichens verwenden
Beispiel: `'A' + 1 == 66`

Java ist **streng typisiert**

- jeder Ausdruck hat einen Typ, der aus dem Quelltext ablesbar ist
- Vorteil: Zulässigkeit von Zuweisungen kann zur Compilezeit geprüft werden

Typ des Ausdrucks

= der hochwertigste Operandentyp im Ausdruck, Beispiel:

$$((\underbrace{1}_{\text{int}} + \underbrace{2L}_{\text{long}}) * (\underbrace{3.0f}_{\text{float}} - \underbrace{4.0}_{\text{double}}))$$

$\underbrace{\quad\quad\quad}_{\text{long}}$ $\underbrace{\quad\quad\quad}_{\text{double}}$

2.2-30 Explizite Typanpassung

Bei Zuweisungen $\langle \text{Variable} \rangle = \langle \text{Ausdruck} \rangle$;
 muss das Ergebnis in den VariablenTyp umgewandelt werden
(Ergebnis-Typanpassung)

Problem

Zuweisungen an niederwertige Typen verbietet der Compiler: Informationsverlust!
 Beispiel: `int x = 1.0; // Compilezeitfehler!!`

Ausweg

Informationsverlust durch “cast-Ausdruck” explizit erzwingen,
 Beispiel: `int x = (int) 1.0; // kein Fehler`

2.2-31 Type-casts

cast-Operatoren

- sind einstellige Operatoren der Form ($\langle \text{Typ} \rangle$) vor Ausdrücken
- **Wirkung:** der Wert des Ausdrucks wird in den angegebenen Typ “gezwungen”

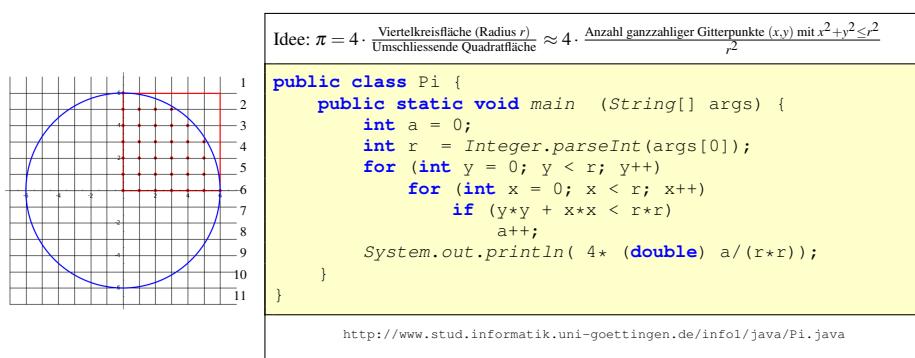
Beispiele

- Gleitkomma \leftrightarrow Ganzzahl: Nachkommastellen unterdrücken/hinzufügen,
 Beispiel: `(int) 3.14 == 3` und `(double) 3 == 3.0`
- double \leftrightarrow float: umrechnen in beste Näherung aus Zieltyp,
 Beispiel: `(float) Double.MIN_VALUE == 0.0f`
- int \leftrightarrow char: überzählige Bytes vorne ignorieren/hinzufügen,
 Beispiel: `(char) 65 == 'A'`
- int \leftrightarrow short: überzählige Bytes (vorne) ignorieren/hinzufügen,
 Beispiel: `(short) 32768 == -32768`

siehe auch Paragraph 2.2-6

analog für andere casts von einem höherwertigem in
 niedrigerwertigen Ganzzahltyp

2.2-32 cast auf double (Beispiel: Näherungsweise Berechnung von π)



cast auf int (Beispiel: Ganzzahlige Zufallszahlen)

2.2-33

```

1 public class RandomInt {
2     public static void main(String[] args) {
3         int N = Integer.parseInt(args[0]);
4         double r = Math.random(); //Zufallszahl im Intervall [0,1)
5         int n = (int) (r * N); //Zufallszahl aus {0, 1, ..., N-1}
6         System.out.println(n);
7     }
8 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/RandomInt.java>Zeile 5: $r \cdot N$ ist ein double-Wert im Intervall $[0, N]$, cast auf int bestimmt den ganzzahligen Anteil.

2.3 Felder

2.3.1 Eindimensionale Felder

Speichern vieler gleichartiger Daten

2.3-1

Feld (oder Array)

- speichert Folge von Werten des gleichen Datentyps (**Grundtyp**)
- Zugriff auf Einzelwerte mit Name und Index (wie Vektorkomponenten)
- Achtung: Nummerierung der Komponenten beginnt bei 0 — gewöhnungsbedürftig, aber vorteilhaft.

Why numbering should start at zero von E. W. Dijkstra

Speicherplatz reservieren, initialisieren, darauf zugreifen

2.3-2

Man nehme ...

- Bezeichner, z.B. a (Name der **Feldvariablen**)
- Grundtyp, z.B. double
- Länge, z.B. `int n = 8;` oder `int n = StdIn.readInt();` oder... (beliebiger int-Ausdruck)

Beispiele für die Definition von Feldern

- schrittweise Definition

```

double[] a; // a als Feldname deklarieren
a = new double[n]; // Feld anlegen ("Speicherplatz reservieren")
for (int i = 0; i < n; i++) // jede einzelne Komponente ...
    a[i] = 0.0; // ... mit 0.0 initialisieren

```

(a[i] ist ein Ausdruck, [] ist der **Feldzugriffsoperator**)

- gleichzeitiges Deklarieren und Anlegen

```
double[] a = new double[n];
```

Anders als bei elementaren Daten muss der Speicherbereich gesondert reserviert werden (hier durch `double[n]`), der Operator `new` liefert seine Anfangsadresse.

Achtung

Auf Feldkomponenten erst zugreifen, wenn das Feld angelegt wurde!

```
1 double[] a;
2 // a = new double[n];
3 for (int i = 0; i < n; i++)
4     a[i] = 0.0;           // FEHLER!
```

Zeile 4 würde spätestens zur Laufzeit Probleme machen, aber hier erkennt bereits der Compiler den Fehler: variable a might not have been initialized.

Deklarieren, Anlegen und Initialisieren gleichzeitig

```
double[] a = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
//           \____ Feldinitialisator(-Ausdruck) ____/
```

Der Feldinitialisator ist NUR bei der Deklaration verwendbar!:

```
double[] b;
b = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}; // Compilezeit-Fehler
```

Java-Besonderheit: Default-Initialisierung

Nach Anlegen des Feldes haben auch die *nicht explizit* initialisierten Komponenten einen Wert, und zwar 0, 0.0, `false` usw. je nach Grundtyp.

2.3-3 Typische Codebeispiele

Feld anlegen und mit Zufallszahlen füllen

```
double [] a = new double[n];
for (int i = 0; i < n; i++)
    a[i] = Math.random();
```

Mittelwert berechnen

```
double sum = 0.0;
for (int i = 0; i < n; i++)
    sum += a[i];
double avg = sum / n;
```

Maximum ermitteln

```
double max = Double.NEGATIVE_INFINITY;
for (int i = 0; i < n; i++)
    max = Math.max(a[i], max);
```

Sequentielle Suche

2.3-4

Suche einen bestimmten Wert q in einer Liste mit N Werten.

```
for (int i = 0; i < N; i++)
    if (f[i] == q)
        System.out.println("Position: " + i);
```

Aufwand: Wieviele Iterationen sind nötig bis q gefunden ist?

bester Fall: q am Anfang	eine Iteration
schlechtester Fall: q am Ende	N Iterationen
im Mittel	$N/2$ Iterationen

2.3.2 Nützliche Algorithmen auf Feldern

Bei sortierten Werten: Binäre Suche

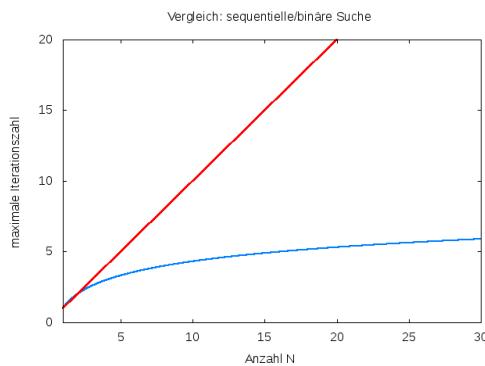
2.3-5

Idee (bei aufsteigender Sortierung)

wiederhole solange Suchintervall nicht leer und Suchwert nicht gefunden

- vergleiche Suchwert mit **Median** (= Wert in Intervallmitte)
 - ist er kleiner (größer), so suche links (rechts) weiter
 - bei Gleichheit: Rückmeldung und Halt

Aufwand



nur etwa $\log_2 N$ Iterationen!!, da Suchbereich jeweils halbiert wird

```
1 public class BinarySearch {
2     public static void main(String[] args) {
3         // ...
4         int lo = 0, hi = N-1, mid;           // unterer/oberer/mittlerer Index
5         while (lo <= hi) {
6             mid = (hi + lo)/2;
7             if (q < f[mid]) hi = mid-1;
8             else if (f[mid] < q) lo = mid+1;
9             else {
10                 System.out.println("Position: " + mid);
11                 return;
12             }
13         }
14         System.out.println("Nicht gefunden!");
15     }
16 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/BinarySearch.java>

Zeile 11: `return` führt zu sofortigem Beenden des Programms.

2.3-6 Bubblesort: Ein sehr einfaches Sortierverfahren

N beliebige Zahlen sollen aufsteigend sortiert ausgegeben werden

Idee ungeordnete Nachbarn austauschen (so oft wie nötig)

```

public class Bubblesort {
    public static void main(String[] args) {
        // ...
        boolean swapped; // flag zeigt an: "Tausch in aktueller Iteration"
        do {swapped = false; // -----
            for (int i=0; i < N-1; i++) // |
                if ( f[i]>f[i+1] ) { // |
                    int tmp = f[i]; f[i] = f[i+1]; f[i+1] = tmp; // <-
                    swapped = true; // -----
                }
            } while (swapped);
        // ...
    }
}

```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Bubblesort.java>

— Aufwand: Wieviele Iterationen der äußeren Schleife sind nötig?

bester Fall: Eingabe aufsteigend sortiert	eine Iteration
schlechter Fall: Eingabe absteigend sortiert	$N - 1$ Iterationen
im Mittel	$N/2$ Iterationen

Begründung: Das Minimum wandert in `for`-Schleife um eine Position nach links.

2.3-7 Statistiken über ganzzahlige Werte erstellen

Beispiel: Wieviele a's, b's, ... werden eingelesen?

```

1 public class CharCount {
2     public static void main(String[] args) {
3         int[] freq = new int[256]; // speichert die Häufigkeiten
4         char c;
5         while (!StdIn.isEmpty()) {
6             c = StdIn.readChar();
7             freq[c]++;
8         }
9         for (c = 0; c < 256; c++) {
10             if (freq[c]>0)
11                 System.out.println((char) c + " : " + freq[c]);
12         }
13     }
14 }

```

<http://www.stud.informatik.uni-goettingen.de/info1/java/CharCount.java>

- Zeile 3 legt ein Feld mit 256 Zählern an (für jedes Byte einen und (default-)initialisiert mit 0).
- Zeile 5: `StdIn.isEmpty()` liefert `true`, wenn das Eingabeende erreicht ist
- Zeile 7 erhöht den Zähler für das gelesene Zeichens (achten Sie auf die Schreibweise — `freq[c++]` ist syntaktisch auch korrekt, bewirkt aber etwas ganz anderes!). Dabei wird ausgenutzt, dass `char`-Werte nichtnegative ganze Zahlen sind, also als Feldindizes verwendbar sind.

Die Technik aus Zeile 7 ist allgemein verwendbar, wenn Werte aus einem endlichen, ganzzahligen Bereich zu zählen sind (z.B. Wieviele Teilnehmer erreichen bei einem Übungsblatt genau 0, 1, ... oder genau 100 Punkte?). Es lohnt sich, diese Idee zu merken. Wir werden sie noch mehrmals verwenden.

Der „Zähltrick“

Ziel Auftrittshäufigkeiten von ganzzahligen Werten ermitteln bei unbekannter Anzahl, aber bekanntem Wertebereich $[0, \dots, M]$.

Idee benutze `int[] freq = new int[M];` zum Abspeichern der Zählerwerte

→ Info1-Sammlung (I1-ID: fa1gok71s4i0)
→ Info1-Sammlung (I1-ID: xm90cao0s4i0)

Primzahlen auflisten

2.3-8

gegeben: natürliche Zahl n , gesucht : alle Primzahlen $\leq n$

Brute-Force-Methode

Iteriere über alle $p \in \{2, 3, \dots, n\}$: falls p keine echten Teiler hat (prüfen durch Probiedivision), gib p aus.

Besser: **Sieb des Eratosthenes** (> 300 v.Chr.)

```

1 public class PrimeSieve {
2     public static void main(String[] args) {
3         int n = Integer.parseInt(args[0]);
4         boolean[] isPrime = new boolean[n+1];
5         for (int k = 2; k <= n; k++) {
6             isPrime[k] = true;
7             for (int k = 2; k*k <= n; k++) {
8                 if (isPrime[k]) {
9                     for (int j = k; k*j <= n; j++)
10                         isPrime[k*j] = false; // Vielfache von Primzahlen sind nicht prim
11                 }
12             }
13             for (int k = 2; k <= n; k++)
14                 if (isPrime[k])
15                     System.out.println(k); // uebrige Kandidaten sind Primzahlen
16             }
17         }

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/PrimeSieve.java>

→ Info1-Sammlung (I1-ID: aaghnl21t4i0)

2.3.3 Einige Besonderheiten von Feldern

Feldlänge

2.3-9

Ein einmal angelegtes Feld a hat unveränderliche Länge, abzulesen in $a.length$.

```

int[] a = new int[8];
a.length = 9; // Compile-Fehler: "cannot assign a value to final variable length"

```

Aber

a kann durch anderes Feld beliebiger Länge überschrieben werden. Dies ergibt sich daraus, wie Felder gespeichert werden.

Speicherung von Feldern

2.3-10

```

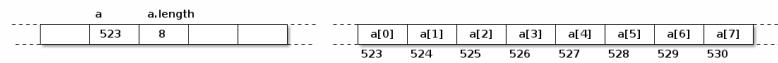
int[] a = new int[8];

```

Man muss unterscheiden zwischen dem *Wert der Feldvariablen* a und dem *Inhalt des Felds* (den Werten $a[0], \dots, a[7]$):

a beinhaltet die Anfangsadresse des Felds (= Adresse von $a[0]$)

Wir ignorieren für den Moment, dass jeder `int`-Wert 4 Bytes erfordert und betrachten vereinfachend *nummerierte Speicherstellen* anstelle von Adressen. Dann entsteht eine Belegung wie diese:



Konsequenzen

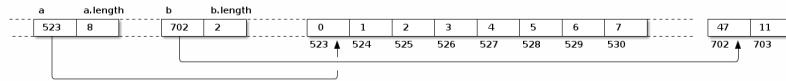
1. Speicherstelle etwa von `a[4]` ist $523 + 4 = 527$
⇒ sehr schnelle Addressberechnung aus Anfangsadresse und Index (Zugriff auf Komponenten ähnlich schnell wie auf Variablen vom Grundtyp)
2. Gewisse Effekte beim Kopieren von Feldvariablen, die man kennen muss

2.3-11 Kopieren von Feldvariablen

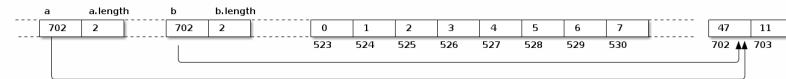
Was passiert bei Ausführung dieses Codes?

```
1 int[] a = { 0, 1, 2, 3, 4, 5, 6, 7 }, b = { 47, 11 };
2 a = b;
```

Speicher nach Ausführung von Zeile 1



Speicher nach Ausführung von Zeile 2

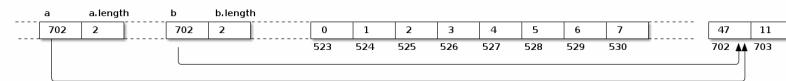


2.3-12 Alias-Effekt

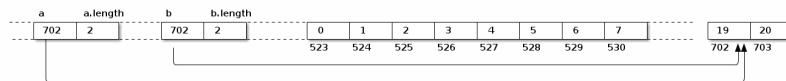
Was passiert bei Ausführung dieses Codes?

```
1 int[] a = { 0, 1, 2, 3, 4, 5, 6, 7 }, b = { 47, 11 };
2 a = b;
3 a[0] = 19; a[1] = 20; // ändere Komponenten von a
```

Speicher nach Ausführung von Zeile 2



Speicher nach Ausführung von Zeile 3



Komponenten von `b` wurden (auch) geändert!

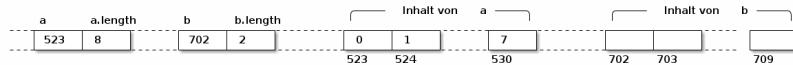
2.3-13 Kopieren des Feldinhalts

```

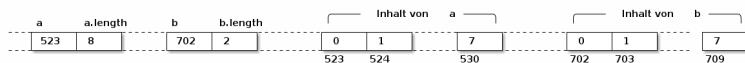
1 double[] a = { 0, 1, 2, 3, 4, 5, 6, 7}, b = new double[length.a];
2 for (int i = 0; i < length.a; i++)
3     b[i] = a[i];

```

Speicher nach Ausführung von Zeile 1



Speicher nach Ausführung von Zeile 3



Indexüberschreitung

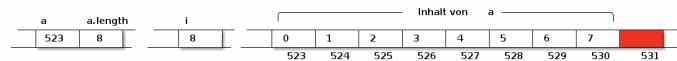
2.3-14

```

1 int[] a = { 0, 1, 2, 3, 4, 5, 6, 7};
2 int i = StdIn.readInt(); // Eingabe 8 entgegennehmen
3 int x = a[i];           // Achtung!

```

Versuchte Ausführung von Zeile 3



Java-Laufzeitsystem beendet Ablauf mit **ArrayIndexOutOfBoundsException** — ein Laufzeit-Fehler. Offenbar kann dies nicht zur Compilezeit entdeckt werden.

Der Fehler ist einfach abzufangen:

```

int[] a = { 0, 1, 2, 3, 4, 5, 6, 7};
int i;
do i = StdIn.readInt();
while (i < 0 || i >= a.length); // Plausibilitäts-Check
int x = a[i];

```

Indexüberschreitungen sind gefährlich!

Bei manchen Programmiersprachen bleiben Indexüberschreitungen zur Laufzeit u.U. unbemerkt. Wird dies nicht im Quellcode verhindert, so kann in Extremfällen Schadsoftware Kontrolle über das System erlangen oder z.B. Passwörter auslesen!

Pufferüberlauf war die Grundlage des sogenannten Heartbleed-Angriffs

2.3.4 Mehrdimensionale Felder

Tabellarische Informationen

2.3-15

Beispiele

- Info 1-Übungspunkte: Student $\hat{=}$ Zeile, Spalte $\hat{=}$ Aufgabe
- Matrix mit Zahlwerten
- Bild = matrixartige Anordnung von Grauwerten/Farbwerten

Programmiersprachliche Entsprechung

Zweidimensionale Felder (2D-Felder) = Felder deren Komponenten eindimensionale Felder sind

2.3-16 Anlegen und Initialisieren

Beispiel: 3×5-Tabelle

			Eintrag [0][3]	
			/	
Zeile 0	78	76	11	22
	98	94	99	90
	99	80	78	19
				64
			Spalte 3	

Java-Umsetzung: Feld der Länge 3, mit int-Feldern der Länge 5 als Komponenten

```
int[][] a;           // Deklaration der Feldvariablen
a = new int[3][5];  // Feld anlegen (Speicherplatz zuordnen)
a[0][3] = 22;       // Wert eines Eintrags setzen
```

Nicht explizit initialisierte Einträge werden mit 0 initialisiert.

Weitere Varianten

gleichzeitig deklarieren und anlegen, danach mit Werten füllen:

```
int[][] a = new int[3][5];      // Feldvariable deklarieren und Feld anlegen
for (int i = 0; i < 3; i++)    // in jeder Zeile ..
    for (int j = 0; j < 5; j++) // .. Spalte fuer Spalte ..
        a[i][j] = StdIn.readInt(); // .. Eintrag von Standardeingabe lesen
```

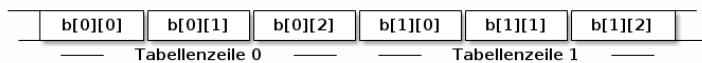
gleichzeitig deklarieren, anlegen und initialisieren:

```
int[][] a = {
    { 78, 76, 11, 22, 54},
    { 98, 94, 99, 90, 76},
    { 99, 80, 78, 19, 64}
};
```

2.3-17 Speicheranordnung

```
int[][] b = new int[2][3]; // 2 x 3 -Tabelle
System.out.println(b.length); // Ausgabe: 2
```

vereinfachte Vorstellung (momentan ausreichend):



Bezugnahme auf Zeilen als Ganzes möglich, auf Spalten nur komponentenweise!:

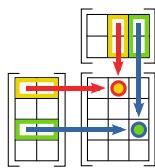
```
System.out.println(b[0].length); // Ausgabe: 3
for (int i = 0; i < b.length; i++) // Ausgabe der ..
    System.out.println(b[i][0]); // .. Spalte 0
```

Anwendung: Matrixmultiplikation

2.3-18

- Produkt $A \cdot B$ ist nur definiert, wenn Spaltenanzahl von A = Zeilenanzahl von B
- Beispiel: 4×2 -Matrix mal 2×3 -Matrix ergibt 4×3 -Matrix
- jeder Eintrag des Produkts ist Skalarprodukt einer Zeile mit einer Spalte:

$$(A \cdot B)_{i,k} = \sum_j A_{i,j} \cdot B_{j,k}$$



Bildquelle: Wikipedia

Der Java-Code bildet die Formel nach:

```

1  double[][] a = new double[1][m], b = new double[m][n];
2  for (int i=0; i<1; i++)
3      for (int j=0; j<m; j++)
4          a[i][j] = StdIn.readDouble();
5  for (int j=0; j<m; j++)
6      for (int k=0; k<n; k++)
7          b[j][k] = StdIn.readDouble();
8  double[][] c = new double[1][n];
9  for (int i=0; i<1; i++)
10     for (int k=0; k<n; k++) {
11         c[i][k] = 0; // Akkumulator
12         for (int j=0; j<m; j++)
13             c[i][k] = c[i][k] + a[i][j] * b[j][k];
14     }

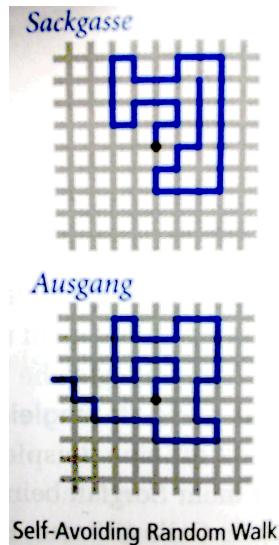
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Matrixmultiplikation.java>

Anwendung: Self-Avoiding Random Walks

2.3-19

Wir platzieren in die Mitte eines $N \times N$ -Gitters ein Tierchen, das nacheinander zufällig Nachbarpunkte besucht, bereits besuchte Punkte aber meidet. Wie wahrscheinlich ist es, dass das Tierchen “entkommt”, d.h. den Gitterrand erreicht?



siehe [Sedgewick/Wayne], nicht klausurrelevant

Bei kleinen Gittern entkommt das Tierchen, aber mit der Größe steigt die Wahrscheinlichkeit einen Punkt zu erreichen, dessen Nachbarn alle bereits besucht wurden — ein einfaches Modell z.B. für das Wachstum von Polymerketten. Der genaue Zusammenhang zwischen Sackgassen-Wahrscheinlichkeit und Gitter-Größe (-Struktur, -Dimension, ...) ist noch unbekannt, kann aber simulativ untersucht

werden. Nachfolgender Code liefert die relative Sackgassen-Häufigkeit bei Gittergröße N in T zufälligen Versuchen.

```

1  public class SelfAvoidingWalk {
2      public static void main(String[] args) {
3          int N = Integer.parseInt(args[0]);
4          int T = Integer.parseInt(args[1]);
5          int deadEnds = 0; // dead end (engl. Sackgasse)
6          for (int t=0; t<T; t++) {
7              boolean[][] a = new boolean[N][N];
8              int x=N/2, y=N/2; // Startposition
9              while (x>0 && x<N-1 && y>0 && y<N-1) {
10                  a[x][y] = true; // falls ...
11                  if (a[x-1][y] && a[x+1][y] // .. alle Nachbarn schon ..
12                      && a[x][y-1] && a[x][y+1]) // .. besucht, dann Sackgasse ..
13                      { deadEnds++; break; } // .. gefunden, while sofort verlassen
14                  // sonst:
15                  double r = Math.random(); // zufällig einen freien Nachbarn ..
16                  if (r<0.25) {if (!a[x+1][y]) x++;} // .. im Osten ..
17                  else if (r<0.5) {if (!a[x-1][y]) x--;} // .. Westen ..
18                  else if (r<0.75) {if (!a[x][y+1]) y++;} // .. Norden oder ..
19                  else if (r<1) {if (!a[x][y-1]) y--;} // .. Süden besuchen.
20              }
21          }
22          System.out.println(100.*deadEnds/T + "% Sackgassen");
23      }
24  }

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/SelfAvoidingWalk.java>

- Zeile 7: default-Initialisierung mit false
- Zeile 13: break beendet die Ausführung der while-Schleife, ohne die aktuelle Iteration zu Ende zu führen
- Zeilen 15-19: es wird zufällig ein unbesuchter Nachbarpunkt bestimmt (und bei der nächsten Iteration der while-Schleife in Zeile 10 als besucht gekennzeichnet)
- Zeile 22: double-Literal 100. anstelle 100 erzwingt Auswertung als double-Ausdruck, dadurch kein logischer Fehler durch Ganzahldivision

2.3-20 Drei- und Mehrdimensionale Felder (nicht klausurrelevant)

```

// Beispiel: 3-dim. Feld aus double-Werten (Erinnerung: double erfordert 8 Bytes)
double[][][] a = new double[L][M][N];
// ... mit Werten füllen
// i, j, k wählen
double x = a[i][j][k];

```

Speicherabbildungsfunktion

- Vorstellung: Feld der Länge L (aus Feldern der Länge M (aus Feldern der Länge N)) belegt $L \cdot M \cdot N$ Speicherstellen
- Speicherstelle von $a[i][j][k]$ befindet sich $d = i \cdot MN + j \cdot N + k$ Plätze nach $a[0][0][0]$
- im Beispiel geht es um ein double-Feld (Speicherstellen beanspruchen 8 Bytes):
 $a[i][j][k]$ befindet sich $8 \cdot d$ Bytes hinter der Anfangsadresse des Felds, also an der Adresse $SAF(a, i, j, k) := a + 8 \cdot (i \cdot MN + j \cdot N + k)$

Allgemein : Die Zugriffszeit ist proportional zur Felddimension (durch Adressberechnung mittels Horner-Schema, siehe Paragraph 2.2-13).

Kapitel 3

Funktionen und Module

3.1 Programmbibliotheken

3.1.1 Standardbibliotheken

Programmbibliotheken

3.1-1

= Sammlungen von (Quell- oder Zielcode-) Routinen, die primär nicht eigenständig wie Applikationen, sondern innerhalb anderer Software benutzt werden

- jede Bibliothek dient typischerweise einem bestimmten Anwendungsbereich, Beispiel: numerische Berechnungen, Datenorganisation, Peripherie-Zugriff usw.
- bei Java werden Bibliotheken durch Klassen implementiert, die verzeichnisartig zu **Packages** zusammengefasst sein können

Zweck

Software-Entwicklung erleichtern durch

- bewährte und geprüfte Standardlösungen
- Abstraktion von Hardware- und anderen Details

Java-Laufzeitumgebung

3.1-2

= Gesamtheit der zum Ablauf von Java-Applikationen nötigen Software-Komponenten, darunter:

- *virtuelle Java-Maschine JVM* (Start mit Befehl `java`) und
- die Java-Standard-Klassenbibliotheken (Java Class Library — JCL) für die jeweilige Hardware-Plattform (synonym verwendet: **Java-API** = *Programmierschnittstelle*)

Die Java-API (*Application Programming Interface*)

- besteht aus einer großen Sammlung von Klassen, unter anderem `String`, `System`, `Integer`, ...
- der Java-Markeninhaber (Oracle) dokumentiert die Klassen eindeutig in der *Java API Specification* (kurz auch einfach Java API genannt)

Um fremde Bibliotheken benutzen zu können, muss man nicht wissen, wie sie implementiert sind. Es genügt, die Dokumentation der Schnittstelle (API) zu kennen.

3.1-3 Das Package `java.lang`

Enthaltene Klassen

Grundlegendes wie z.B. `Integer`, `Double` usw., `String` (für Datentypen), `Math` (für mathematische Funktionen), `System` (u.a. für den Zugriff auf Standard-Ein- und Ausgabe)

Nutzung

Die Definitionen der `java.lang`-Klassen stehen Compiler und JVM automatisch (d.h. ohne weitere Vorkehrungen) zur Verfügung, wie bereits das `HelloWorld`-Beispiel zeigt

3.1-4 Der voll qualifizierte Name (FQN: *fully qualified name*)

identifiziert Bibliotheksbestandteile eindeutig durch ihren Pfad

Beispiele

- FQN von `String`

`java.lang.String`

- FQN von `Integer.MAX_VALUE`

`java.lang.Integer.MAX_VALUE`

Besonderheit bei `java.lang`

Klassen aus `java.lang` können im Quelltext verkürzt d.h. nur mit Klassennamen angesprochen werden, andere Packages erfordern eine Pfadangabe!

3.1-5 Weitere Packages

java.util Hilfsklassen z.B. für Datenstrukturen

java.applet Klasse Applet sowie entsprechende Zugriffsstrukturen für geeignete Webbrowser

java.net Klassen, die für Netzwerkzugriffe benötigt werden

und viele weitere

Beispiel: Bibliotheksmethode für aufsteigende Sortierung eines Felds

```

1  public class Sortieren {
2      public static void main(String[] args) {
3          int n = StdIn.readInt();
4          double[] f = new double[n];
5          for (int i = 0; i < n; i++)
6              f[i] = StdIn.readDouble();
7          java.util.Arrays.sort(f);           // <--FQN-Aufruf der Bibliotheksmethode
8          for (int i = 0; i < n; i++)
9              System.out.println(f[i]);
10     }
11 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Sortieren.java>

import <Klasse>; bzw. **import** <Package>.*;
macht Vereinbarungen der Klasse (bzw. aller Klassen von <Package>) im aktuellen Quellcode bekannt:

Beispiel

```
import java.util.*;
public class Sortieren {
    ...
    Arrays.sort(f);           // verkürzter Aufruf
    ...
}
```

Die **import**-Anweisung muss VOR der Klassendefinition stehen!

API-Dokumentation

Die Java-API enthält zigtausend Klassen(!), sie muss knapp, präzise und einheitlich dokumentiert sein. Die Dokumentation enthält für jede(s) Paket/Unterpaket/Klasse:

- Kurzbeschreibung des Einsatzzwecks
- vollständige Liste der Elemente mit *Signaturen* (siehe Paragraph 2.2-28)
- zu jedem Element eine präzise Funktionsbeschreibung

Beispiel

Dokumentation von `java.lang.Math` (Java Version 7) :

<http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>

3.1.2 Nutzerbibliotheken

Idee

3.1-6

Nützlichen Programmcode in einer Klasse sammeln und als dokumentierte Bibliothek zur Verfügung stellen

Beispiel

Befindet sich `StdIn.class` im gleichen Verzeichnis (siehe Paragraph 1.2-31), so stehen bequeme Eingabe-Methoden zur Verfügung.

Aus der API-Dokumentation:

nach [Sedgewick/Wayne], vollständig

hier:<http://introcs.cs.princeton.edu/java/stdlib/javadoc/StdIn.html>

<code>public class StdIn</code>	
<code>boolean isEmpty()</code>	prüft ob noch Eingaben vorliegen
<code>int readInt()</code>	liest einen Wert vom Typ int
<code>double readDouble()</code>	liest einen Wert vom Typ Double
<code>...</code>	usw. für long,...,String
<code>String readLine()</code>	liest bis zum Zeilenende
<code>String readAll()</code>	liest bis zum Eingabende

Voraussetzung

Compiler und Laufzeitsystem müssen Zugriff auf die Klasse haben

Der Klassenpfad

3.1-7

= Liste der Verzeichnisse mit Klassen, auf die Compiler/JVM Zugriff haben

Klassenpfad ausgeben

```
public class PrintClassPath {
    public static void main (String[] args) {
        System.out.println(System.getProperty("java.class.path"));
    }
}
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/PrintClassPath.java>

Fallweises Setzen des Klassenpfads

Beispiel für JVM-Aufruf:

```
cd /home/damm/inf01/skript/java
java -cp .:/home/damm/java/StdIn PrintClassPath
```

Ausgabe:

```
./:/home/damm/java/StdIn
```

Bequemer: Arbeitsumgebung konfigurieren, so dass benötigte Klassen immer zur Verfügung stehen (s. folgenden Paragraphen).

3.1-8 stdlib.jar

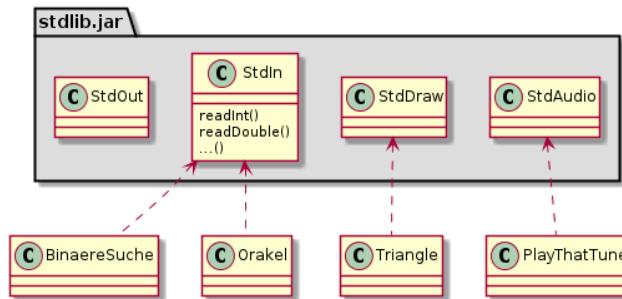
Java-Archive (Dateiendung .jar)

können *mehrere* class-Dateien in komprimierter Form enthalten— das bietet technische Vorteile. Die Java-API befindet sich z.B. in rt.jar im Verzeichnis /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/. Der Inhalt wird aufgelistet, wenn dort jar tfv rt.jar aufgerufen wird.

Weiteres Beispiel

siehe Website zu [Sedgewick/Wayne]

stdlib.jar enthält StdIn und weitere für uns nützliche Klassen. Wir verwenden das Archiv als Nutzerbibliothek. Es steht unter der GNU General Public License.



Konfiguration Angenommen Sie speichern das Archiv in ~/inf01/java/stdlib.jar. Anstatt den Klassenpfad bei jedem javac/java-Aufruf anzugeben, können Sie die Arbeitsumgebung z.B. so anpassen:

1. Linux: Datei ~/.bashrc ergänzen um die Zeile

```
export CLASSPATH=.:${HOME}/inf01/java/stdlib.jar
```

Windows: Umgebungsvariable CLASSPATH entsprechend anpassen, dabei : durch ; ersetzen

2. neues Konsolenfenster öffnen (oder aus- und wieder einloggen).

stdlib.jar steht unter der GNU Public License. Für Projekte außerhalb dieses Kurses die Änderung ggf. zurücknehmen um Lizenzkonflikte zu vermeiden.

3.1.3 Nachträge zur Ein- und Ausgabe

Erinnerung: EVA(S)-Prinzip

3.1-9

Sicht auf Datenverarbeitungsprozesse

- $E \xrightarrow[S]{V} A$: Eingabe–Verarbeitung (unter Benutzung des Speichers)–Ausgabe
- Ein- und Ausgabe sind “Dinge”, die Ströme von Daten in einer Form liefern/empfangen, so dass sie direkt verarbeitet werden können

Abstraktion

- Datenströme kann man technisch immer als (z.B. elektrisch übertragene) Bytesequenzen realisieren
- Verarbeitungsschritt muss keine weiteren technischen Details von Ein- und Ausgabe kennen, sondern bezieht alles auf **Standardeingabe** und **Standardausgabe** (bzw. Standardfehlerausgabe)

in Java ansprechbar mit `System.in`, `System.out`,

`System.err`

3.1-10

Beispiel: Standardeingabe

Interaktive Eingabe

- kompliziertes Zusammenspiel mit Betriebssystem
- einfacher: benutze Klasse `StdIn`

Beispiele

Wir haben schon viele Beispiele gesehen, z.B. `Temperatur`, `Char2Value`, `DecimalDecode`, `DualFraction`,

Vorteil der abstrakten Sicht

Eingaben könnten ebensogut von Tastatur, Netzwerkkarte, Bewegungssensor, ... geliefert werden: Die Verarbeitung bleibt dieselbe.

Ergänzungen (nicht klausurrelevant)

3.1-11

Klasse `StdOut`

gehört zu `stdlib`, wird ebenso verwendet wie `System.out`:

dokumentiert auf der Website zu [Sedgewick/Wayne]

<code>System.out.print(...)</code>	<code>StdOut.print(...)</code>
<code>System.out.println(...)</code>	<code>StdOut.println(...)</code>

Standardfehlerausgabe

`System.err.print(...)` leitet Daten an ein alternatives Ziel (bzw. auch an die Standardausgabe, falls keine Alternative vereinbart ist), und wird ebenso verwendet wie `System.out`:

siehe Paragraphen 1.1-19, 1.1-20

<code>System.out.print(...)</code>	<code>System.err.print(...)</code>
<code>System.out.println(...)</code>	<code>System.err.print(...)</code>

`stdlib` enthält keine entsprechende Klasse `StdErr`.

Formatierte Ausgabe

Für `System.out`, `System.err` und `StdOut` gibt es Ausgaberoutinen, die einen *Formatstring* und zusätzlich ein oder mehrere Werte verlangen. Wirkung:

Formatstring wird ausgegeben, aber die darin eingeschobenen Platzhalter (für jeden Wert einer) werden vorher ersetzt durch Stringdarstellungen der Werte. Die gewünschte Form der Stringdarstellung (z.B. Stellenzahl hinter dem Dezimalpunkt) ist im Platzhalter codiert. Details in der API-Dokumentation, hier nur ein Beispiel:

Nette Spielerei: Vergleichen Sie die Ausgabe von Matrixmultiplikation (Paragraph 2.3-18) und von MatrixmultiplikationFormatiert bei Multiplikation der 1×1 -Matrizen $(0.1) \cdot (0.1)$. Nur die Zeile für die Ausgabe der Ergebnisse ist ausgetauscht:
<code>System.out.printf("% .2f ", c[i][k]);</code>
http://www.stud.informatik.uni-goettingen.de/infol/java/MatrixmultiplikationFormatiert.java

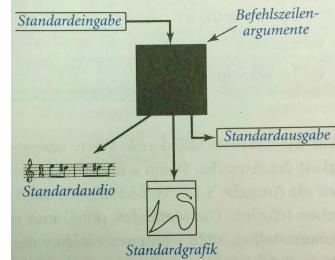
3.1-12 Grafik und Audio

Andere Datenquellen und -senken

- Sensoren transformieren physikalische/chemische Messwerte in elektrische Signale, die als Bytestrom an Standardeingabe gekoppelt werden
- analog sind Grafik- oder Soundkarte als Datensenken geeignet
ABER: Kopplung mit Standardausgabe ist kompliziert, da Hardware-abhängig

Ausweg: `stdlib`

mit den Abstraktionsklassen `StdDraw` und `StdAudio` für vereinfachte Handhabung



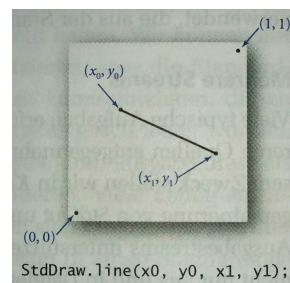
Grafiken aus [Sedgewick/Wayne]

3.1.4 Vektorgrafik mit der Klasse `StdDraw`

Wir werden die Klasse (ebenso wie später `StdAudio`) nur benutzen, nicht die Implementierung untersuchen.

3.1-13 Grundgedanke

- abstraktes Zeichengerät, das Punkte, Linien, usw. auf eine quadratische Leinwand zeichnet
- Koordinatenskalen entsprechen in der Grundeinstellung der Leinwandgröße $[0, 1] \times [0, 1]$
- Farbe, Strichstärke, Skalierung, ... durch Anweisungen änderbar



Grundlegende Zeichenanweisungen

```
public class StdDraw
void line(double x0, double y0, double x1, double y1)  Strecke zeichnen
void point(double x, double y)                           Punkt zeichnen
```

vollständige Dokumentation auf der Website zu
[Sedgewick/Wayne]

Beispiele

3.1-14

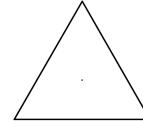
“Grafisches Hello World!”

```
public class Triangle {
    public static void main(String[] args) {
        // ...
        double s = Double.parseDouble(args[0]); // Seitenl.
        double h = s * Math.sqrt(3.0)/2.0; // Hoehe
        StdDraw.line(0.0, 0.0, s, 0.0); // Basis
        StdDraw.line(0.0, 0.0, s/2, h); // linke Seite
        StdDraw.line(s, 0.0, s/2, h); // rechte Seite
        StdDraw.setPenRadius(0.010);
        StdDraw.point(s/2, h/3.0); // Schwerpunkt
    }
}
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Triangle.java>

Zugriff immer mit Erwähnung der Klasse, z.B. `StdDraw.line (⟨Argumente⟩)`

Ergebnis in Grafikfenster



Konfigurationsanweisungen

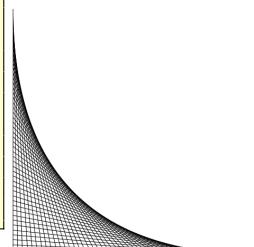
wirken für nachfolgende Zeichenanweisungen (bis zur nächsten Konfigurationsanweisung)

```
public class StdDraw
void setXscale(double x0, double x1)  setzt x-Bereich auf  $(x_0, x_1)$ 
void setYscale(double y0, double y1)  setzt y-Bereich auf  $(y_0, y_1)$ 
void setPenRadius(double r)           setzt Stiftradius auf  $r$ 
```

Anwendung

```
public class Envelope
{
    public static void main(String[] args) {
        int N = 50;
        StdDraw.setXscale(0, N);
        StdDraw.setYscale(0, N);
        for (int i = 0; i <= N; i++)
            StdDraw.line(0, N-i, i, 0);
    }
}
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Envelope.java>



3.1-15 Batch-Verarbeitung

Dieser Code ermöglicht massenhaftes Einlesen und Visualisieren von Daten, wie zum Beispiel in der Datei USA.txt.
Beispielauftrag: java PlotFilter < USA.txt

```
public class PlotFilter {
    public static void main(String[] args) {
        // lies erste 4 Werte der Standardeingabe:
        double x0 = StdIn.readDouble();
        double y0 = StdIn.readDouble();
        double x1 = StdIn.readDouble();
        double y1 = StdIn.readDouble();
        // skaliere damit die Leinwand
        StdDraw.setXscale(x0,x1);
        StdDraw.setYscale(y0,y1);
        // solange Eingaben vorhanden ...
        while (!StdIn.isEmpty()) {
            // lies paarweise und zeichne den Punkt
            double x = StdIn.readDouble();
            double y = StdIn.readDouble();

            StdDraw.point(x,y);
        }
    }
}
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/PlotFilter.java>

3.1-16 Funktionsgraphen zeichnen

Idee: *stückweise lineare Näherung*

- Kurve wie $y = \sin(4x) + \sin(20x)$ aus n geraden Strecken zusammensetzen
- Endpunkte bestimmen durch Funktionsauswertung

```
public class FunctionGraph {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]); // n Segmente
        double[] x = new double[n+1], y = new double[n+1]; // n+1 Endpkte
        for (int i = 0; i <= n; i++) {
            x[i] = Math.PI * i / n; // Endpunkt-Koordinaten der ..
            y[i] = Math.sin(4*x[i]) + Math.sin(20*x[i]); // .. Segmente
        }
        StdDraw.setXscale(0, Math.PI);
        StdDraw.setYscale(-2.0, +2.0);
        for (int i = 0; i < n; i++) { // alle Segmente darstellen
            StdDraw.line(x[i], y[i], x[i+1], y[i+1]);
        }
    }
}
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/FunctionGraph.java>

3.1-17 Animationen im Daumenkino-Prinzip

Steuerung

public class StdDraw	
void clear()	Löscht Leinwand durch Übermalen mit Weiß
void show(int dt)	zeichnet, wartet dann dt Millisekunden ^a

^aErklärung: Der erste Aufruf von show(int dt) schaltet den Animationsmodus an, d.h. Zeichnungen werden „gesammelt“ und erst nach nächstem Aufruf gezeigt.

Beispiel

```
public class EnvelopeDemo {
    public static void main(String[] args) {
        int N = 50;
        StdDraw.setXscale(0,N);
        StdDraw.setYscale(0,N);
        while( true )
            for (int i = 0; i <= N; i++) {
                StdDraw.clear();
                StdDraw.line(0, N-i, i, 0);
                StdDraw.show(500);
            }
    }
}

http://www.stud.informatik.uni-goettingen.de/infol/java/EnvelopeDemo.java
```

3.1.5 Audio mit der Klasse `StdAudio`

Prinzipien

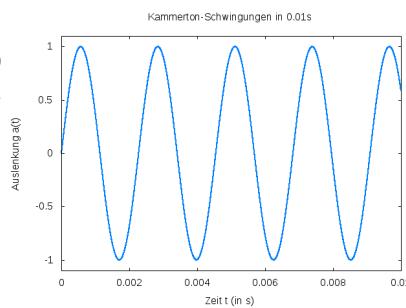
3.1-18

Klang

- ist die Wahrnehmung von Schwingungen des Trommelfells (ausgelöst durch Druckschwankungen um Mittelwert)
- Frequenz entspricht der Tonhöhe, Lautstärke der *Amplitude* (maximale Abweichung vom Mittelwert)

Beispiel: Kammerton A

- entspricht Sinuswelle mit 440 Schwingungen pro Sekunde (440Hz)
- als Funktion
(Zeit in Sek. \mapsto Auslenkung):
 $a(t) = \sin(2\pi t \times 440)$



Signal abtasten (engl. *sampling*)

3.1-19

= analoges Signal durch endlich viele Messpunkte repräsentieren (ähnlich: stückweise lineare Näherung)

Beispiel: Kammerton A

```

public class Sampling {
    public static void main(String[] args) {
        int sps = Integer.parseInt(args[0]); // Samples pro Sek
        double dauer = 0.01;
        int N = (int) (sps * dauer);
        double[] a = new double[N];
        for (int i = 0; i <= N; i++) {
            a[i] = Math.sin(2 * Math.PI * 440 * i / sps);
        }
        StdDraw.setPenRadius(0.005);
        StdDraw.setXscale(0, N+1);
        StdDraw.setYscale(-1,1);
        for (int i = 0; i <= N; i++)
            StdDraw.point(i, a[i]);
    }
}

```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Sampling.java>

Erst bei genügend hoher Abtaffrequenz ist Sinuskurve optisch gut zu erkennen. Führen Sie die Klasse aus mit den Abtastraten 5512, 11025, 22050, 44100.

Akustische Reproduktion von Klang

- erfordert Abtastrate $> 2 \times$ maximale Schwingungsfrequenz
- typische Abtastrate bei Audio-CDs: 44100Hz, entsprechend dem menschlichen Hörbereich (bis ca. 20kHz)

3.1-20 Tonerzeugung

Prinzip

- den Abtastwerten entsprechend periodisch schwankende Spannung erzeugen
- über Lautsprecher (schwingende Membran) wiedergeben

Aus der API-Dokumentation

vollständige Dokumentation auf der Website zu
[Sedgewick/Wayne].

public class StdAudio	
final int SAMPLE_RATE	Standard-Abtastrate 44.1 kHz
void play(double a)	Sample a für 1/44100s abspielen
void play(double[] a)	die durch Samples a gegebene Schallwelle wiedergeben

3.1-21 Beispiel: Kammerton 10 Sekunden lang spielen

```

1  public class Kammerton {
2      public static void main(String[] args) {
3          int sps = 44100;           // Samples pro Sekunde
4          int hz = 440;             // Frequenz
5          double duration = 10.0;   // zehn Sekunden
6          int N = (int) (sps * duration); // Gesamtzahl der Samples
7
8          double[] a = new double[N+1];
9          for (int i = 0; i <= N; i++)
10             a[i] = Math.sin(2 * Math.PI * hz * i / sps);
11
12         StdAudio.play(a);
13     }
14 }
```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Kammerton.java>

Weitere Töne

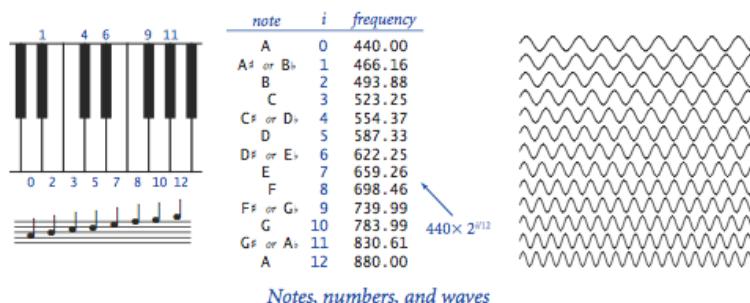
3.1-22

Chromatische Tonleiter

- Kammerton = a' (eingestrichenes A: 440Hz)
eine Oktave höher: a'' (zweigestrichenes A: 880Hz)
- jede Oktave ist in 12 gleichmäßige Schritte eingeteilt (Frequenzerhöhung um gleichen Faktor)

Frequenz nach i -ter Erhöhung: $2^{i/12} \cdot 440\text{Hz}$

Veranschaulichung



Grafik aus [Sedgewick/Wayne]

Beispiel: PlayThatTune

3.1-23

Tonhöhen (über Kammerton A) und Tonlängen (in Sekunden) einlesen und abspielen

```

1 public class PlayThatTune {
2     public static void main(String[] args) {
3         while (!StdIn.isEmpty()) {
4             // Tonhoehe (in Bezug auf Kammerton A) einlesen
5             int pitch = StdIn.readInt();
6             // Dauer in Sekunden einlesen
7             double duration = StdIn.readDouble();
8
9             double hz = 440 * Math.pow(2, pitch / 12.0);
10            int N = (int) (StdAudio.SAMPLE_RATE * duration);
11            double[] a = new double[N+1];
12            for (int i = 0; i <= N; i++) {
13                a[i] = Math.sin(2 * Math.PI * hz * i / StdAudio.SAMPLE_RATE);
14            }
15            StdAudio.play(a);
16        }
17    }
18 }
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/PlayThatTune.java>

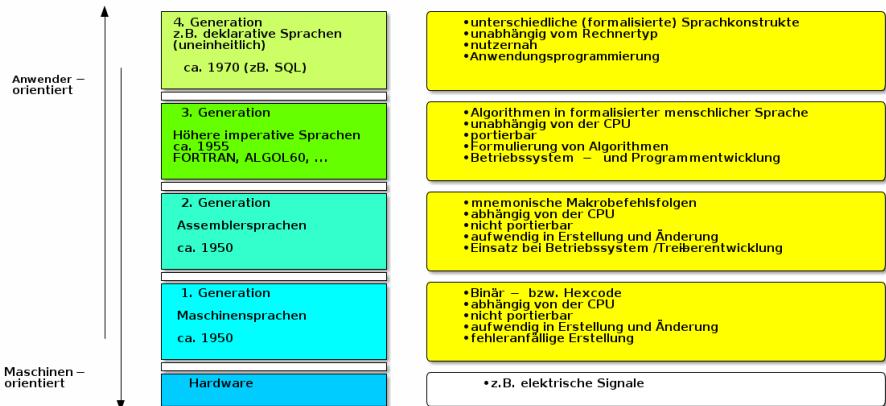
Das bekannte "Bruder Jakob" beginnt mit A B C# A A B C# A, entsprechend den Tonhöhen 0 2 4 0 0 2 4 0 (über A), jeweils eine halbe Sekunde lang gespielt. Also ist 0 0.5 2 0.5 4 0.5 ... einzulesen.

3.2 Programmiersprachen-Nomenklatur

Groeinteilung nach Abstraktionsgrad

3.2-1

nach Schwabe/Hauske "Software", Uni Zürich 2007



3.2-2 Imperativ vs. deklarativ

nach Stahlknecht/Hasenkamp "Einführung in die Wirtschaftsinformatik"

Imperative Formulierung

Gib Anweisungen, wie das gewünschte Ergebnis zustandekommen soll:

1. Nimm Buch
2. Prüfe, ob Titel "Informatik"
3. Falls JA, notiere ISBN
4. Prüfe, ob letztes Buch
5. Falls NEIN, weiter bei 1.
6. Falls Ja, Ende

Deklarative Formulierung

Beschreibe das gewünschte Ergebnis:

- Suche alle Bücher mit Titel "Informatik" und notiere deren ISBN

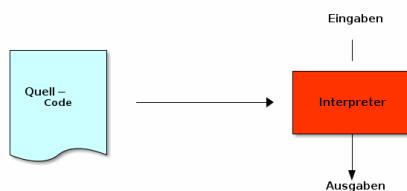
3.2-3 Einteilung nach dem Ausführungsschema

Das Bild aus Paragraph 1.2-21 wird nun verfeinert durch die Unterscheidung:

- Skriptsprachen (*interpretierte Sprachen*)
- compilierte Sprachen
- Bytecode-Sprachen (Sammelbegriff für Mischformen)

3.2-4 Skriptsprachen (*interpretierte Sprachen*)

- **Interpreter** prüft nacheinander jede einzelne hochsprachliche Anweisung auf syntaktische Korrektheit und führt sie direkt aus

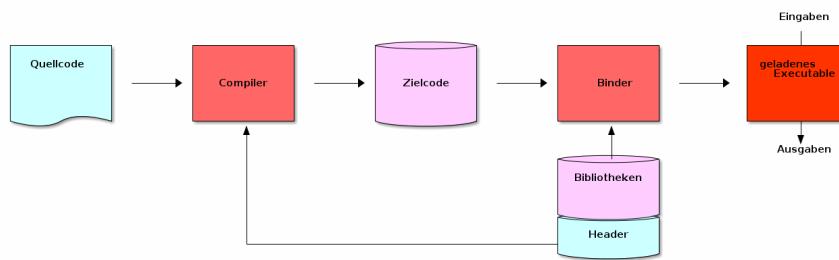


- Beispiele:
 - Shellskripte (Interpreter = Kommandozeileninterpreter `bash`)
 - JavaScript (Interpreter in Browser integriert)
- flexible und schnelle Programmentwicklung, langsam in der Ausführung, eingeschränkter Einsatz

Compilierte Sprachen

3.2-5

- Quellcode wird durch Compiler *einmal* in Maschinencode übersetzt
- weiterer Maschinencode (z.B. E/A-Routinen) wird *zur Compilezeit* dazugebunden, Schnittstellendeklarationen (“Header”) sichern korrekte Verwendung
- das vollständige Programm (*Executable*) wird *geladen* und ausgeführt (Laden und Ausführen *beliebig oft*, ohne Neucompilation)



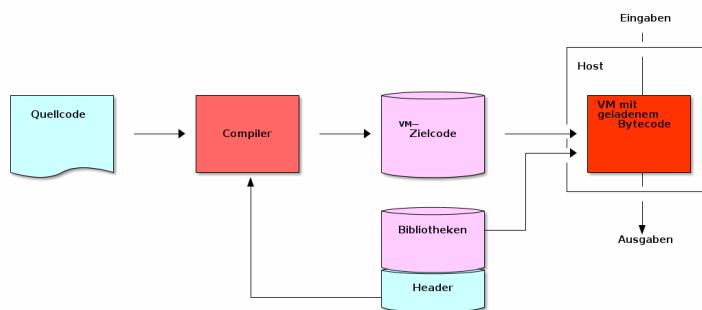
der dafür zuständige Lader ist hier nicht dargestellt

- Beispiel: Pascal, C, ...
- erfordert mehr Programmierdisziplin und Kenntnis von Bibliotheken
- hohe Ausführungsgeschwindigkeit, breites Anwendungsspektrum

Bytecode-Sprachen

3.2-6

- VM (virtuelle Maschine) bezeichnet hier ein Softwaremodell einer standardisierten CPU das auf einem realen Computer (Host) ausgeführt wird



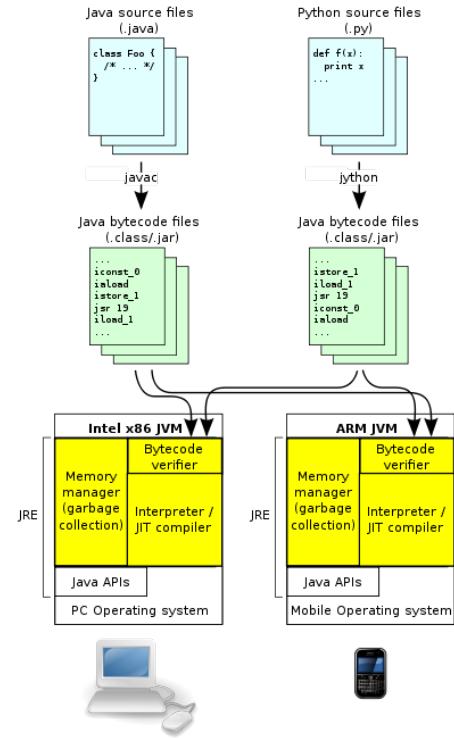
- Compiler übersetzt Quellcode in VM-Zielcode (in Analogie zum *binary Bytecode* genannt)
- VM fungiert als Interpreter für den Bytecode, arbeitet dennoch sehr schnell (z.B. entfällt Syntaxprüfung)
- nativer Code (d.h. in Host-Maschinensprache vorliegend) wird *zur Laufzeit* dazugebunden, Optimierung zur Laufzeit, Speicherbereinigung etc.

3.2.7 Beispiel: Java

Bildquelle: Wikipedia

`java <Applikation>` lädt und startet Klasse in **virtueller Java-Maschine** (*Java virtual machine, JVM*)

- unabhängig von Host-Architektur!
(erfordert JVM-Implementation für Host)
- Java-Bytecode z.B. auch aus Python-Code möglich!
(erfordert Compiler Python→Java-Bytecode)



3.2.8 Einteilung nach Programmierparadigma

Programmierparadigma

= der Programmiersprache zugrundeliegendes Prinzip der Verarbeitung von Daten

Für uns sind das *imperative* und das *objektorientierte* Paradigma am wichtigsten:

imperative Programmierung

Programme = Folgen von *Anweisungen*, die die Transformation $E \xrightarrow[V]{(S)} A$ schrittweise beschreiben (Daten werden dabei als Variablenwerte gespeichert)

- Beispiele: C, Pascal, auch Java ist im Kern imperativ

objektorientierte Programmierung

Programme = Klassen, die Datentypen beschreiben (also Mengen von Datenobjekten und anwendbaren Operationen)

- Beispiele: C++, Java, C#

nicht-imperative Programmierung

z.B. logische oder funktionale Programmierung (behandeln wir hier nicht)

Ein erster Vergleich

3.2-9

Beispiel: Manipulation geometrischer Objekte in der Ebene

rein imperativ

```
// Parameter einlesen:
double m1 = ...; n1 = ...;
double m2 = ...; n2 = ...;
// Geraden "definieren"
// g1: y = m1*x + n1
// g2: y = m2*x + n2
// Geraden und Schnittpunkt zeichnen
double px = ...; // Berechnung
double py = ...; // Berechnung
plotGerade(m1,n1);
plotGerade(m2,n2);
plotPunkt(px,py);
```

objektorientiert

```
// Parameter einlesen:
double m1 = ...; n1 = ...;
double m2 = ...; n2 = ...;
// Geraden definieren
Gerade g1 = new Gerade( m1, n1 );
Gerade g2 = new Gerade( m2, n2 );
// Geraden und Schnittpunkt plotten
Punkt p = g1.schnittpunkt(g2);
g1.plot();
g2.plot();
p.plot();
```

Imperative Programmierung ist Grundlage

3.2-10

... weiterer Abstrahierungen. Deswegen ist Wiederholung/Vertiefung einiger Kenntnisse sinnvoll.

Basis: Das Variablenkonzept

Variablen und Ausdrücke anstelle von Adressen und einschrittigen Rechenwerksoperationen der Assembler-/Maschinensprachen-Programmierung

“Ausbaustufen”

- strukturierte Programmierung (if, while, for etc. anstelle goto-Programmierung also Alles, was wir bisher gemacht haben)
- prozedurale Programmierung (Prozeduren/Funktionen als Verallgemeinerung von Anweisungen/ Ausdrücken) und
- modulare Programmierung (anstelle monolithischer Programme)

3.3 Nachträge zur strukturierten Programmierung

3.3-1

Die folgenden beiden Abschnitte sind selbstständig zu erarbeiten. Der Inhalt ist — bis auf besonders gekennzeichnete Teile — prüfungsrelevant.

3.3.1 Variablen und Ausdrücke

3.3-2 Variablen und ihre Deklaration

Definition

Variable = Speicherstelle eines bestimmten *Typs*, im Quelltext mit einem symbolischen Namen (*Bezeichner*) zugreifbar

Deklaration

Variablennamen müssen vor erster Verwendung *deklariert* werden, damit

- der Compiler die richtige Verwendung prüfen kann
- zur Laufzeit entsprechende Speicherstellen zugeordnet werden

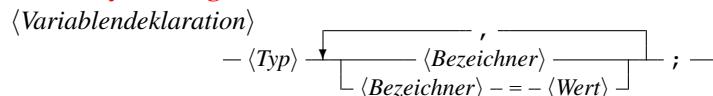
Beispiele

```
{
  ...
  // ...
  int x;           // Deklaration ..
  int y, z;        // .. ohne Initialisierung
  int a, b = 1, c; // .. mit einer Initialisierung
  ...
}
```

Variablen-deklarationen sind in jedem Anweisungsblock an beliebiger Stelle erlaubt

3.3-3 Formale Beschreibung der Syntax ...

... mit Syntaxdiagramm



Äquivalent: Die *erweiterte Backus-Naur-Form* (EBNF)

= Metasprache für *Syntaxregeln* zur Beschreibung einer Programmiersprache

- in einfachen Anführungszeichen ‘buchstäbliche Erwähnung’
- in spitzen Klammern *<Spracheinheiten>*
- in eckigen Klammern [*<optionaler Bestandteil>*]
- in geschweiften Klammern \i<0- oder mehrfach wiederholbarer Bestandteil>\

EBNF-Beschreibung von Variablen-Deklarationen

$\langle VDekl \rangle ::= \langle Typ \rangle \langle Bezeichner \rangle ['=' \langle Wert \rangle] \{ , \} \langle Bezeichner \rangle ['=' \langle Wert \rangle] \} ;$ ' ::= ' heißt "ist definiert als"

Bemerkung

Java-Syntax kann *komplett* mit EBNF beschrieben werden

Java Language Specification (JLS) - nichts für schwache

Symbolische Konstanten

Nerven
3.3-4

Deklarationen mit Initialisierung darf das Schlüsselwort `final` als *Modifizierer* vorangestellt werden, nachträgliches Überschreiben wird verboten.

Beispiel

```
final int ANSWER = 42;
... //
ANSWER = 3; // Compilezeitfehler!
```

Zuweisungen

3.3-5

Wert der Variablen

- ist definiert durch Typ-gemäße Interpretation des Bitmusters
- kann nur durch Zuweisungen geändert werden
Beispiel: $\langle Variable \rangle = \langle Wert \rangle ;$

Erinnerung :

- Variablen sind benannte Speicherstellen
- Werte werden durch Ausdrücke angegeben (enthaltene Variablen müssen vorher initialisiert sein!)

in allen Syntaxregeln ist $\langle Wert \rangle$ durch $\langle Ausdruck \rangle$ zu ersetzen

Zuweisungssyntax

$\langle Zuweisung \rangle ::= \underbrace{\langle Speicherstelle \rangle}_{\langle Zuweisungsausdruck \rangle} '=' \langle Ausdruck \rangle ;$

Wert des Zuweisungsausdrucks: der zugewiesene Wert

Beispiele

3.3-6

Die Syntaxregel erfasst auch Fälle wie die unten mit // <- markierten:

Berechne Nullstellen $x_{0,1}$ von $ax^2 + bx + c = 0$ ($a \neq 0$)

```
// ... Einlesen von a, b, c
double d = b*b - 4*a*c; // <- Bestandteile sind initialisiert, also OK
double[] x = new double[2]; // <- Operator 'new' stellt Feldadresse als Wert bereit
if (d > 0) {
    x[0] = (-b + Math.sqrt(d)) / (2*a); // <- x[0] bezeichnet eine Speicherstelle
    x[1] = (-b - Math.sqrt(d)) / (2*a); // <- x[1] ebenfalls
}
if (d == 0) {
    x[0] = x[1] = -b / (2*a); // <- Wert des Zuweisungsausdrucks an x[0] zuweisen
//      \_Zuw.ausdr._/   (Wert = zugewiesener Wert, also "durchreichen")
```

Merke

Zuweisungsausdrücke werden *von rechts nach links* ausgewertet

3.3-7 Variable vs. Ausdruck

Gegenüberstellung

Variable	Ausdruck
Beispiel: <code>c</code>	Beispiel: <code>b*b - 4*a*c</code>
muss deklariert werden	muss syntaktisch korrekt sein
hat Typ und Adresse	hat Typ und Wert, aber <i>keine Adresse</i>
beschreibt Adresse durch den Namen	beschreibt Wert durch eine Berechnung

Folgerung

- Variablen sind *links* vom Zuweisungsoperator ‘=’ erlaubt, ebenso Ausdrücke, die Speicherstellen beschreiben
- andere Ausdrücke sind nur *rechts* vom Zuweisungsoperator erlaubt

3.3-8 Seiteneffekte (oder Nebeneffekte)

Ausdruck mit Seiteneffekt

= Ausdruck, dessen Auswertung eine Änderung im Speicher bewirkt,
 Beispiel: Zuweisungsausdrücke haben offenbar einen Seiteneffekt, ebenso zusammengesetzte Zuweisungen (siehe Paragraph 2.2-16) sowie Inkrement- und Dekrement-Ausdrücke

Inkrement und Dekrement

- kommen in *Präfix-* und *Postfix-Form* vor:

| trennt Alternativen $\langle \text{Inkrementausdruck} \rangle ::= '+' \langle \text{speicherstelle} \rangle \mid \langle \text{Speicherstelle} \rangle '++'$
 $\langle \text{Dekrementausdruck} \rangle ::= '--' \langle \text{Speicherstelle} \rangle \mid \langle \text{Speicherstelle} \rangle '--'$

- bewirken Wertänderung um ± 1 (bei *allen* numerischen Datentypen, auch *double* etc.)
- Postfix- und Präfix-Variante sind gleichwertig sofern isoliert verwendet

Achtung bei Kombination mit anderen Operatoren!

```
// Beispiele
int x, y = 0;
x = ++y;           // jetzt: x == y == 1 (wie erwartet)
                  // *Praefix-Inkrement wird stets VOR dem Lesezugriff ausgefuehrt*
x = y++;          // jetzt: x == 1, y == 2 (vielleicht ueberraschend)
                  // *Postfix-Inkrement wird stets NACH dem Lesezugriff ausgefuehrt*
                  // [das wird gelegentlich (selten) ausgenutzt]
x += x++;         // x == ****
                  // Ergebnis haengt ab von der Implementierung der Laufzeitumgebung!!
                  // syntaktisch erlaubter, in Praxis+Pruefung *verbotener* Ausdruck
```

Weitere “seltsame Operatoren” (nicht klausurrelevant)

3.3-9

bedingte Auswertung (“Elvis-Operator” ?:)

Syntax $\langle \text{boolean-Ausdruck} \rangle '?' \langle \text{Ausdruck1} \rangle ':' \langle \text{Ausdruck2} \rangle$

ÄUSSERST SPARSAM VERWENDEN!

Semantik ist $\langle \text{boolean-Ausdruck} \rangle$ true, so wird der Wert durch Auswertung von $\langle \text{Ausdruck1} \rangle$ bestimmt, sonst durch Auswertung von $\langle \text{Ausdruck2} \rangle$

Beispiel (hier noch ganz gut lesbar)

```
int b = ( a<0 ) ? -a : a; // Betrag von a
```

Bitoperatoren (nur auf Ganzahl- bzw. boolean-Operanden anwendbar)

z.T. analog zu boolean-Operatoren, nur bitorientiert

Beispiel: $(a_1, \dots, a_N) \& (b_1, \dots, b_N)$ liefert Bitmuster $(a_1 \wedge b_1, \dots, a_N \wedge b_N)$

Bemerkung: $\&$ ist eine “Schablone”: die 0-Bits von b verdecken entsprechende Bits des linken Operanden.

$\&, \&=$	bitweises UND
$, =$	bitweises ODER
$^, ^=$	bitweises ExOR (entweder oder)
\sim	bitweises Komplement
$<< \ell, <<= \ell$	Linksshift um ℓ Positionen, 0-Ergänzung
$>> \ell, >>= \ell$	Rechtsshift um ℓ Positionen, Ergänzung durch Führungsbit
$>>> \ell, >>>= \ell$	Rechtsshift um ℓ Positionen, 0-Ergänzung

siehe Paragraph 2.1-12

Schriftliche Multiplikation mit Bitoperatoren programmiert

```
1 int a = Integer.parseInt(args[0]), b = Integer.parseInt(args[1]), c;
2 c = 0;
3 while ( b!= 0 ) {
4     if ( (b & 1) != 0 ) // aequiv. zu 'if ( b % 2 != 0 )'
5         c += a;
6     a <<= 1;
7     b >>= 1;
8 }
// c speichert jetzt Produkt a * b
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Multiplikation.java>

Übersicht über alle Java-Operatoren und deren Priorität

3.3-10

Level	Operator	Description	Associativity
16	[] . ()	access array element access object member parentheses	left to right
15	++ --	unary post-increment unary post-decrement	not associative
14	++ -- + - ! ~	unary pre-increment unary pre-decrement unary plus unary minus unary logical NOT unary bitwise NOT	right to left
13	() new	cast object creation	right to left
12	* / %	multiplicative	left to right
11	+ -	additive	left to right
10	<< >> >>>	string concatenation shift	left to right
9	<= >= instanceof	relational	not associative
8	== !=	equality	left to right
7	&	bitwise AND	left to right
6	^	bitwise XOR	left to right
5		bitwise OR	left to right
4	&&	logical AND	left to right
3		logical OR	left to right
2	? :	ternary	right to left
1	= += *= ^= ^= = = <=>= >>=	assignment	right to left

Quelle: [Sedgewick/Wayne]-Webseite

3.3.2 Die Kontrollanweisungen von Java

3.3-11 Warum wird `goto` nicht benutzt in Java??

naheliegende weitere Abstraktion

- unbeschränkte “GOTO-Anweisung” anstelle Assembler-JMP
- beabsichtigte Wirkung: Kontrollfluss wird bei angegebener Anweisung fortgesetzt
- erhoffter Vorteil: hohe Ausführungsgeschwindigkeit wie bei optimiertem Assemblercode
- frühe FORTRAN und BASIC-Varianten waren “GOTO-lastig”

BASIC-Beispiel (von Wikipedia)

```

10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Fully Completed."
70 END

```

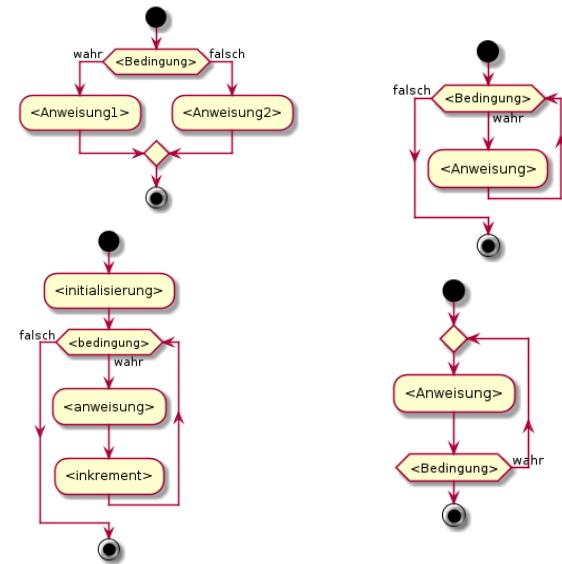
Konsequenz für große Projekte

- unübersichtlicher “Spaghetti-Code”

- hohe Wartungskosten

Ausweg: strukturierte Programmierung

verwende Sprünge nur “implizit” durch Verzweigung und Schleifen



Berühmte Polemiken gegen Spaghetti-Code: Edsger W.

3.3-12 Dijkstra: Go To Statement Considered Harmful, Donald

E. Knuth: Structured programming with go to statements

Rechtfertigung: Structured programming theorem (Böhm, Jacopini 1966)

Jedes imperativen Programm kann mittels `if` und `while` und Blöcken in ein äquivalentes GOTO-freies Programm transformiert werden.

Beobachtung:

3.3-13

Die Ausdrucksstärke von Java wäre nicht eingeschränkt, wäre **nur ein** Typ von Schleifenanweisung erlaubt (z.B. nur `while`, kein `for`, kein `do-while`)

→ Info1-Sammlung (II-ID: qln7rxh0k5i0)

Die Varianten erhöhen aber die Lesbarkeit, denn sie entsprechen typischen Abläufen.

Die Kontrollanweisungen von Java

3.3-14

Kontrollanweisungen = Anweisungen, die die Ablaufreihenfolge steuern

$\langle \text{Kontrollanweisung} \rangle ::= \langle \text{Anweisungsblock} \rangle$
 | $\langle \text{while-Anweisung} \rangle \mid \langle \text{do-while-Anweisung} \rangle \mid \langle \text{for-Anweisung} \rangle$
 | $\langle \text{if-Anweisung} \rangle \mid \langle \text{switch-Anweisung} \rangle$
 | $\langle \text{break-Anweisung} \rangle \mid \langle \text{continue-Anweisung} \rangle$

Erklärung

- `continue` benutzen wir hier nicht
- `switch` wird am nachfolgenden Beispiel erklärt
- `break` wird häufig im Zusammenhang mit `switch` benutzt (s.u.), die Verwendung im Zusammenhang mit Schleifen wurde in Paragraph 2.3-19 (bei `SelfAvoidingWalk`) benutzt: `break` bewirkt das sofortige Verlassen der innersten umfassenden Schleife, weiter geht's mit der ersten Anweisung *nach* der Schleife

3.3-15 Die switch-Anweisung

Beispiel

Manche if-Kaskade kann man durch eine switch-Anweisung ersetzen:

```
if ( tag == 0 ) {
    System.out.println("Montag");
} else if ( tag == 1 ) {
    System.out.println("Dienstag");
} else if ( tag == 2 ) {
    System.out.println("Mittwoch");
} else if ( tag == 3 ) {
    System.out.println("Donnerstag");
} else if ( tag == 4 ) {
    System.out.println("Freitag");
} else {
    System.out.println("Wochenende!");
}
```

```
switch ( tag ) {
case 0: System.out.println("Montag"); break;
case 1: System.out.println("Dienstag"); break;
case 2: System.out.println("Mittwoch"); break;
case 3: System.out.println("Donnerstag"); break;
case 4: System.out.println("Freitag"); break;
default: System.out.println("Wochenende!");
}
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Wochentag.java>

- runde Klammern hinter switch enthalten einen Ausdruck (erlaubte Typen: Ganzzahl oder String)
- case zeigt Sprungmarken an: Literale vom selben Typ wie der switch-Ausdruck
- Der Wert des switch-Ausdrucks gibt die anzuspringende Marke an.
- Durch das *optionale(!) break* springt der Kontrollfluss zur nächsten Anweisung *nach dem switch-Block*. Ohne break würden dagegen *alle* weiteren Anweisungen *innerhalb* des switch-blocks ausgeführt.
- Ist keine passende Marke vorhanden, so wird (falls vorhanden) die Marke default verwendet bzw. mit der nächsten Anweisung fortgefahren.

3.4 Prozedurale Programmierung

Wir betrachten hier *prozedurale Programmierung* als eine “Ausbaustufe” der imperativen Programmierung.

3.4.1 Statische Methoden

3.4.1 Methoden

praktisch alle imperativen Programmiersprachen (auch Assemblersprachen) bieten die Möglichkeit “Unterprogramme” zu definieren.

Unterprogramme

= benannte Programmteile, die Teilaufgaben übernehmen. Vorteile:

- überall im Programm aufrufbar mit Namen (und ggf. Parametern)
- übersichtlicher (gut wartbarer) *wiederverwendbarer* Code

- unterstützt Top-Down-Entwurf: (siehe Paragraph 1.1-13)

Andere Termini für “Unterprogramm”: z.B. Routine, Subroutine, ...

Methoden

= Bezeichnung für die Unterprogramme objektorientierter Programmiersprachen.

- wir behandeln jetzt nur **statische Methoden** (Schlüsselwort `static`)
- später: *Instanzmethoden* (ohne `static`)

Erweiterung des imperativen Paradigmas: Programme = Sammlungen von Methoden
(Transformation $E \xrightarrow[V]{(S)} A$ durch Methodenaufrufe)

Funktionen

3.4-2

Funktion im Sinne von Java

= eine Methode, die als Verarbeitungsergebnis einen Wert *liefert*

Analogie: *mathematische Funktionen*



Statische Funktionsdefinitionen

3.4-3

```

public static <Ergebnistyp> <Methodenname> (<Parameterdeklarationen>) // Kopf
{ // Rumpf:
  ... // Anweisungen
  ... // es MUSS eine return-Anweisung geben wie diese:
  return <Ausdruck>; // vom <Ergebnistyp> oder implizit umwandelbar
  ... // es SOLL KEINE System.out.println()-Anweisung geben
}
  
```

- Der Ergebnistyp heißt **Typ der Methode**.
- Warum keine Textausgabe auf `System.out`? Verarbeitungsergebnis im EVA(S)-Sinne ist der gelieferte Wert.
Textausgaben als Nebeneffekte sind in dieser Situation besser auf die Standardfehlerausgabe zu leiten.

Beispiel: Eine int-Methode

```

1  public static int signum( int n) {
2    if (n > 0)
3      return +1;
4    if (n < 0)
5      return -1;
6    return 0;
7  }
  
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Vorzeichen.java>

- Zeilen 3, 5: Wert wird geliefert und Funktionsabarbeitung endet sofort \Rightarrow else-Zweig unnötig.
- Zeile 6: Jede Möglichkeit die Methode zu verlassen muss mit `return <Ausdruck>` enden, sonst: Compilezeitfehler.

3.4.4 Typischer Code für Funktionsimplementationen

Absolutwert

```
public static int abs(int x) {
    if (x<0) return -x;
    else return x;
}
```

```
public static double abs(double x) {
    if (x<0) return -x;
    else return x;
}
```

Primzahltest

```
public static void isPrime(int n) {
    if (n<2) return false;
    for (int i=2; i<=n/i; i++) // Probdivision
        if (n%i == 0) return false;
    return true;
}
```

Hypotenuse eines rechtwinkligen Dreiecks

```
public static double hypotenuse(double a, double b) {
    return Math.sqrt(a*a + b*b);
}
```

3.4.5 Beispiel: Euklidischer Algorithmus verpackt in eine Methode

```
1 public class GGT {
2     public static int
3         euklidModern(int x, int y) {
4             int r;
5             while (y != 0) {
6                 r = x % y;
7                 x = y;
8                 y = r;
9             }
10            return x;
11        }
12        public static void main( String[] args) {
13            int a = StdIn.readInt();
14            int b = StdIn.readInt();
15            int d;
16            d = euklidModern(a, b); // Aufruf der Methode
17            System.out.println(d);
18        }
19    }
```

<http://www.stud.informatik.uni-goettingen.de/info1/java/GGT.java>

Zeile 16: **Methodenaufruf:** x, y werden mit konkreten Werten initialisiert und die Methode damit ausgeführt.

3.4.6 Beispiel: Quadratwurzel mit dem Newtonverfahren

Aus [Sedgewick/Wayne] S.205ff

```
public class Newton {
    public static double sqrt(double c) {
        if (c < 0) return Double.NaN;
        double err = 1E-15;
        double t = c;
        while (Math.abs(t - c/t) > err*t)
            t = (c/t + t) / 2.0;
        return t;
    }
    public static void main(String[] args) {
        double[] a = new double[args.length];
        for (int i = 0; i < args.length; i++)
            a[i] = Double.parseDouble(args[i]);
        for (int i = 0; i < a.length; i++) {
            double x = sqrt(a[i]);
            System.out.println(x);
        }
    }
}
```

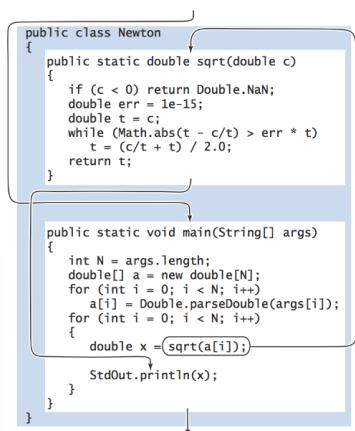
<http://www.stud.informatik.uni-goettingen.de/infol/java/Newton.java>

Test der Klasse:

```
prompt>: java Newton 1 2 3 1000000.1 -2 0 -0 NaN Infinity -Infinity
1.0
1.414213562373095
1.7320508075688772
1000.0000499999987
NaN
0.0
-0.0
NaN
Infinity
NaN
```

Kontrollfluss

3.4-7



Neue Art der Ablaufsteuerung

(neben Bedingungen und Schleifen)

Grafik aus [Sedgewick/Wayne]

1. Sprung (Funktionseinsprung) zur ersten Codezeile der Funktion: sqrt übernimmt Kontrolle
2. Rücksprung: main übernimmt Kontrolle

Tracing des Aufrufs `java Newton 1 2`

3.4-8

```
main({ 1, 2})
    sqrt( 1.0)
        Math.abs( 0.0 )
        return 0.0
    return 1.0
    sqrt( 2.0)
        Math.abs( 1.0 )
```

```

        return 1.0
    Math.abs( 0.1666666666666674)
    return 0.1666666666666674
    Math.abs( 0.004901960784313486)
    return 0.004901960784313486
    Math.abs( 4.2477996395895445E-6)
    return 4.2477996395895445E-6
    Math.abs( 3.1896707497480747E-12)
    return 3.1896707497480747E-12
    Math.abs( -2.220446049250313E-16)
    return 2.220446049250313E-16
    return 1.414213562373095
return

```

3.4-9 Tracing der lokalen Variablen

```

main({ 1, 2})
    sqrt( 1.0 )
    | t: 1.0
        Math.abs( 0.0 )
        return 0.0
    return 1.0
    sqrt( 2.0 )
    | t: 2.0
        Math.abs( 1.0 )
        return 1.0
    | t: 1.5
        Math.abs( 0.1666666666666674 )
        return 0.1666666666666674
    | t: 1.4166666666666665
        Math.abs( 0.004901960784313486 )
        return 0.004901960784313486
    | t: 1.4142156862745097
        Math.abs( 4.2477996395895445E-6 )
        return 4.2477996395895445E-6
    | t: 1.4142135623746899
        Math.abs( 3.1896707497480747E-12 )
        return 3.1896707497480747E-12
    | t: 1.414213562373095
        Math.abs( -2.220446049250313E-16 )
        return 2.220446049250313E-16
    return 1.414213562373095
return

```

→ Info1-Sammlung (II-ID: nvdj0hq0l5i0)

3.4-10 Überladen von Methoden

= gleichnamige Methoden mit unterschiedlichen Parameterlisten definieren

```

public static double sqrt(double c) {
    if (c < 0) return Double.NaN;
    double err = 1E-15;
    double t = c;
    while (Math.abs(t - c/t) > err*t)
        t = (c/t + t) / 2.0;
    return t;
}
public static double sqrt(double c, double err) {
    if (c < 0) return Double.NaN;
    double t = c;
    while (Math.abs(t - c/t) > err*t)
        t = (c/t + t) / 2.0;
    return t;
}

```

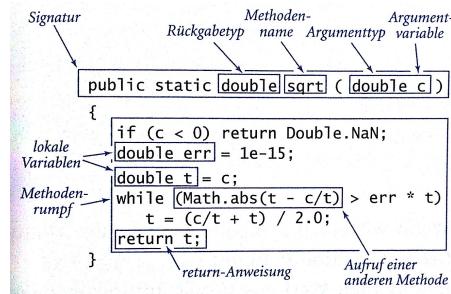
<http://www.stud.informatik.uni-goettingen.de/infol/java/Newton.java>

erste Methode: Genauigkeit voreingestellt, zweite Methode: Genauigkeit nach Vorgabe des Aufrufers

ACHTUNG: gleichnamige Methoden, die sich *nur* im Ergebnistyp unterscheiden, verbietet der Compiler

“Anatomie” statischer Methoden

3.4-11



Grafik aus [Sedgewick/Wayne], Modifizierer (`public`, `static`, ...) zählen streng genommen nicht zur Signatur

Formalparameter

= Synonym für die in der Signatur (i.W. synonym: Schnittstelle) genannten Argumentvariablen

Verwendung der Formalparameter im Methodenrumpf

3.4-12

Merke

Formalparameter sind auch lokale Variablen!

Unterschied zu im Rumpf deklarierten Variablen

- Formalparameter werden beim Aufruf mit Aufrufwerten initialisiert, lokale Variablen dagegen wie im Rumpf angegeben
- **Konsequenz:** Wertänderungen im Rumpf wirken sich nicht auf den aufrufenden Code!

Beispiel

```

1 public class Nachfolger
2 {
3     static int nachfolger( int n) {
4         n = n + 1;
5         return n;
6     }
7     public static void main(String[] argv) {
8         int k = Integer.parseInt(argv[0]);
9         System.out.println(nachfolger(k));
10        System.out.println(nachfolger(k));
11    }
12 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Nachfolger.java>

- Zeile 2: Der Wert der lokalen Variablen `n` wird geändert.
- Zeilen 7/8: Der Formalparameter `n` wird mit dem aktuellen Wert von `k` initialisiert. Der Wert von `k` wird dazu (als lokale „Arbeitskopie“) in den Speicherplatz kopiert, der beim aktuellen Aufruf für `n` angelegt wurde.

Semantik des Methodenaufrufs

3.4-13

Call-by-value-Prinzip (oder auch pass-by-value)

Zur Auswertung von

$\langle\text{Methodename}\rangle (\langle\text{Ausdruck}1\rangle, \dots, \langle\text{Ausdruck}N\rangle)$

werden (1) die Werte von $\langle\text{Ausdruck}1\rangle, \dots, \langle\text{Ausdruck}N\rangle$ bestimmt und (2) in die Formalparameter **kopiert**. Danach beginnt die Abarbeitung der Methode.

Denkbare Alternative: *call-by-reference*

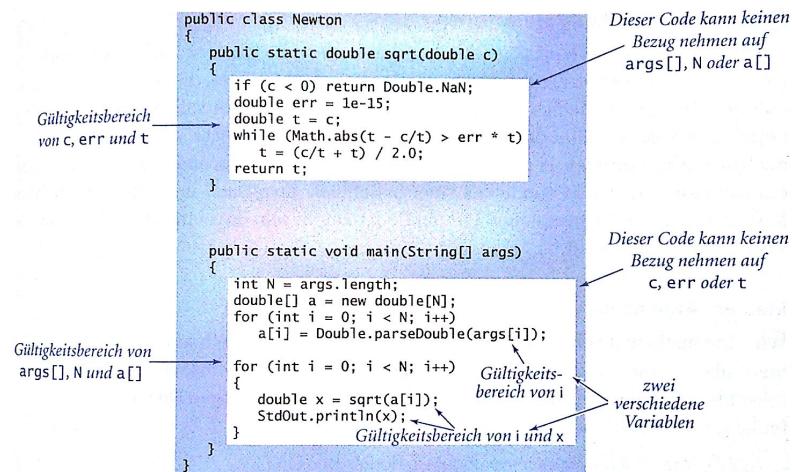
Operiere mit Variablen des aufrufenden Codes selbst

- bei Java mit elementaren Daten nicht möglich

aber: strukturierte Daten werden per *Referenz* übergeben — später mehr dazu

3.4-14 Gültigkeitsbereich (Scope) eines Bezeichners

= Quelltextbereich, in dem seine Deklaration wirkt.



Grafik aus [Sedgewick/Wayne]

– lokale Variablen

= innerhalb einer Methode deklarierte Variablen
ab Deklaration nur innerhalb des umschließenden Blocks gültig

Besonderheit: for

Laufvariable der for-Anweisung gilt nur in zugehöriger/m Anweisung/-sblock!

```
for (int i = 1; i < 4; i++)
    System.out.println("Iteration Nr.: " + i); // Rumpf d. for-Anweisung
    System.out.println(i); // Compilezeitfehler!
```

Besonderheit: Klassenscope

Methoden sind *Klassenelemente* („auf oberster Ebene“ innerhalb einer Klasse definiert) und ihre Namen sind darum im gesamten Quelltext der Klasse gültig - auch VOR ihrer vollständigen Deklaration.

3.4-15 Motto der prozeduralen Programmierung

Wann immer sich ein Problem klar in Teile gliedern lässt, sollte man diese durch separate Methoden lösen.

Vorteile

- Konzentration auf klar abgegrenzte Bereiche
- *Debugging* (Fehlersuche) und *Wartung* (Weiterentwicklung) werden vereinfacht
- Wiederverwendbarkeit des Codes

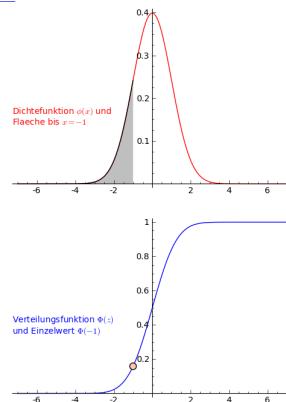
Beispiel: Implementierung mathematischer Funktionen

3.4-16

nach [Sedgewick/Wayne]

- **Dichte** der Standardnormalverteilung („Gaußsche Glockenkurve“):

$$\phi(x) := e^{-x^2/2} / \sqrt{2\pi}.$$



3.4-17

Java-Realisierung

Weder ϕ noch Φ sind in `Math` implementiert!

Idee für Eigenimplementation

- programmiere $\phi(x)$ nach Definition mit Funktionen aus `Math`
⇒ Aufruf `phi(x)` als Abkürzung für den komplizierten Ausdruck
- für $\Phi(z)$ gibt es keine geschlossene Formel, aber die Beziehung

$$\Phi(z) = \frac{1}{2} + \phi(z) \left(z + \frac{z^3}{3} + \frac{z^5}{3 \cdot 5} + \frac{z^7}{3 \cdot 5 \cdot 7} + \dots \right).$$

⇒ Berechnung ähnlich wie bei der Exponentialfunktion (siehe `Exponential.java`).

```

1  public class Gaussian {
2      public static double phi(double x) {
3          return Math.exp(-x*x / 2) / Math.sqrt(2 * Math.PI);
4      }
5      public static double Phi(double z) {
6          if (z < -8.0) return 0.0;
7          if (z > 8.0) return 1.0;
8          double sum = 0.0, term = z;
9          for (int i = 3; sum + term != sum; i += 2) {
10              sum = sum + term;
11              term = term * z * z / i;
12          }
13      }
14  }
15

```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Gaussian.java>

- Zeile 9: bei großem *i* sind die Termwerte zu klein für *double* (Unterlauf). Dann ändert sich *sum* nicht mehr und die Iteration endet (wie bei dem Fixpunktalgorithmus 2.2-32).

3.4-18 void-Methoden

sind Methoden, die *keinen Wert ausliefern*, sondern einen **Seiteneffekt** haben, z.B.

- eine Ausgabe erzeugen, Beispiel: `System.out.println(42);`
- oder eine „Veränderung im Systemzustand“ bewirken

`void` dient dabei als Platzhalter für den Ergebnistyp im Methodenkopf

```

public static void <Methodename> (<Parameterdeklarationen>) {
    ... // Anweisungen
    ... // darunter MEIST System.out.println()-Anweisungen
}
public static void <Methodename> (<Parameterdeklarationen>) {
    ... // Anweisungen
    return; // return-Anweisung NUR OHNE <Ergebnis> erlaubt
    ... // MEIST gibt es System.out.println()-Anweisungen
}

```

Die main-Methode ist vom Typ `void`. Aber auch Methoden der Klassen `StdDraw` und `StdAudio`.

3.4-19 Beispiele: Ein Dreieck zeichnen

```

public static void drawTriangle(double x0, double y0,
                                double x1, double y1,
                                double x2, double y2) {
    StdDraw.line(x0, y0, x1, y1);
    StdDraw.line(x1, y1, x2, y2);
    StdDraw.line(x2, y2, x0, y0);
}

```

3.4.2 Felder als Formalparameter

3.4-20 Ohne Längenangabe!

Merkel

mehr dazu im nächsten Kapitel

Feldparameter werden bei Methodendeklaration und -aufruf ohne Länge angegeben!

Beispiel: Deskriptive Statistik

```

1  public class Statistik {
2      public static double min(double[] a) {
3          double m = Double.POSITIVE_INFINITY;
4          for (int i=0; i<a.length; i++)
5              if (a[i] < m) m = a[i];
6          return m;
7      }
8      public static void main(String[] args) {
9          int N = Integer.parseInt(args[0]);
10         double[] d = new double[N];
11         for (int i = 0; i<N; i++) d[i] = StdIn.readDouble();
12         System.out.println("Minimum: " + min(d));
13         // ...
14     }
15 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Statistik.java>

Weitere elementare Statistik-Funktionen für Zahlenfolgen a lassen sich ähnlich programmieren: min , mean für den Mittelwert \bar{a} , stddev für die (empirische) Standardabweichung $\sqrt{\frac{1}{n-1} (\sum_i (a_i - \bar{a})^2)}$ usw.

(Unvollständiges) Beispiel: Sudoku-Checker

3.4-21

Partielle Sudoku-Lösung auf Widerspruchsfreiheit prüfen

```

1  public class SudokuChecker {
2      public static boolean check(int[] a) {
3          int[] count = new int[10];
4          for (int i = 1; i < a.length; i++) {
5              count[a[i]]++;
6              if (count[a[i]] > 1) return false;
7          }
8          return true;
9      }
10     public static void main(String[] args) {
11         int[][] a = new int[10][10];
12         for (int z = 1; z < 10; z++) {
13             for (int s = 1; s < 10; s++) { a[z][s] = StdIn.readInt(); }
14         }
15         int[] line = new int[10];
16         // Teste die Zeilen ...
17         for (int z = 1; z < 10; z++) {
18             for (int s = 1; s < 10; s++) line[s] = a[z][s];
19             if (!check(line)) {
20                 System.out.println("fehlerhafte Zeile: " + z);
21                 return;
22             }
23         }
24         // ... dann teste Spalten und Blöcke
25     }
26 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/SudokuChecker.java>

check prüft, ob a die "Sudoku-Bedingung" erfüllt, d.h. in Komponenten 1 bis 9 keinen Wert $\neq 0$ doppelt enthält (es sei gesichert, dass nur Werte 0 bis 9 vorkommen, wobei 0 für "nicht gesetzt" steht). Dazu wird der Zähltrick aus Paragraph 2.3-7 benutzt. Wird in der main-Methode nacheinander jede Zeile, jede Spalte und jeder Block zunächst im eindimensionalen Feld line gespeichert und dann mittels check (line) geprüft, so entsteht ein vollständiger Test: `java SudokuChecker < .../data/sudoku1.txt`

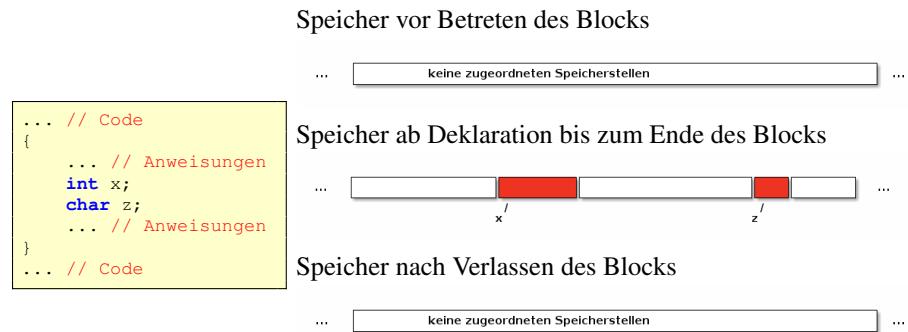
→ InfoI-Sammlung (II-ID: tcidje41t4i0)

3.4.3 Der Laufzeitstapel

Lokale Variablen werden bei jedem Aufruf neu angelegt

3.4-22

Beispiel



Der Stack (Stapel)

andere Namen für den Laufzeitstapel sind Aufrufstack,
runtime stack, call stack, ...

3.4-23 Speicherorganisation bei Methodenaufrufen

relevante Daten

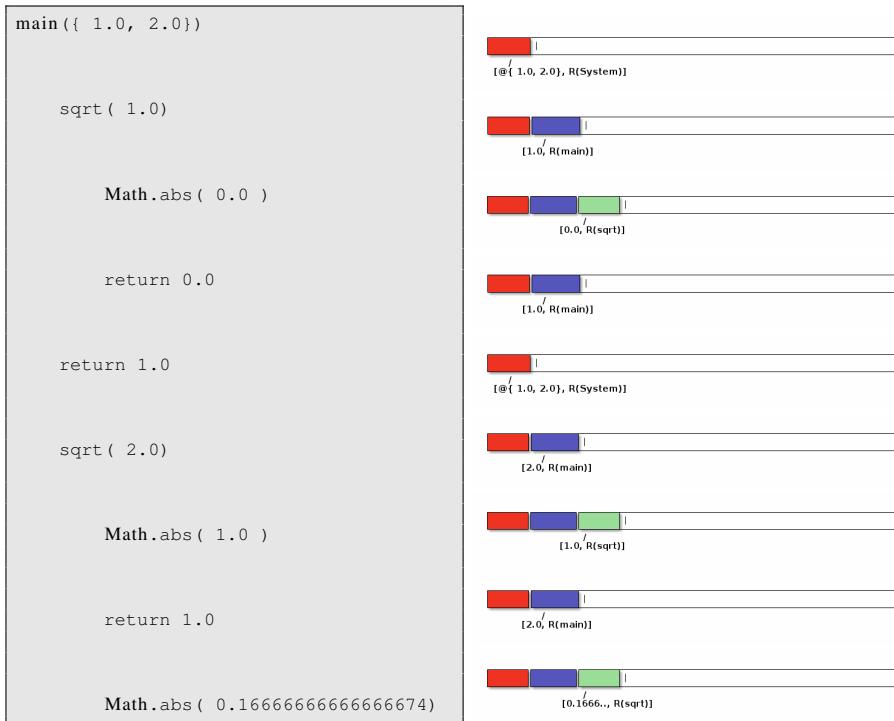
- Werte *aller* lokalen Variablen (Aufrufparameter + im Rumpf deklarierte) Bezeichnung: V
- **Rücksprungadresse** = Adresse des Befehls der dem Aufruf folgt Bezeichnung: R

Aktivierungssatz (oder Aktivierungsrahmen, stack frame)

= Abschnitt im Stack mit entsprechenden Speicherplätzen, wird

- bei Aufruf *neu angelegt* und mit $[V, R]$ initialisiert
- wird nach Wertübergabe an Aufrufer freigegeben

Veranschaulichung



Analogie: Schreibtisch-Modell

3.4-24

- Hauptvorgang:
Antragsformular bearbeiten
- falls Ergebnis von anderem Vorgang benötigt:
aktuellen Vorgang unterbrechen, entsprechendes Formular auf voriges legen
(als nun aktuellen Vorgang) bearbeiten
(Zwischenergebnisse des vorigen sind verdeckt)
- Abschluss eines Vorgangs:
dessen Formular vom Stapel entfernen, Ergebnis in oberstes Formular eintragen
- Hauptvorgang endet, wenn Stapel vollständig abgearbeitet

3.4.4 Rekursion

Beispiel: Fakultätsfunktion

3.4-25

Der Laufzeitstapel ermöglicht ohne weiteres sich selbst aufrufende Methoden. Diese Technik heißt **Rekursion**.

Standardbeispiel: *Fakultät*

- Für $n \in \mathbb{N}$ ist

$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)! & \text{sonst.} \end{cases}$$

- Die rekursive Implementation beruht unmittelbar auf dieser Formel:

```

1   public static long factorial(int n) {
2       if (n <= 1) return 1;
3       return n * factorial(n - 1);
4   }

http://www.stud.informatik.uni-goettingen.de/infol/java/Fakultaet.java

Die Korrektheit ist unmittelbar einzusehen durch vollständige Induktion (siehe auch Kapitel 7):
Anfang offenbar liefert factorial(1) den korrekten Wert 1 = 1!
Schritt liefert factorial(n-1) den korrekten Wert, so auch factorial(n) (das sichert Zeile 3)

```

Tracing des Aufrufs `factorial(5)`

```

factorial(5)
    factorial(4)
        factorial(3)
            factorial(2)
                factorial(1)
                    return 1
                return 2*1 = 2
            return 3*2 = 6
        return 4*6 = 24
    return 5 * 24 = 120

```

Der **Basisfall** $n = 1$ erfordert keine weiteren rekursiven Aufrufe.

3.4-26 Beispiel: rekursiver Euklidischer Algorithmus

```

public static int ggT( int x, int y ) {
    if ( y == 0) return x;
    return ggT( y, x % y );
}

http://www.stud.informatik.uni-goettingen.de/infol/java/EuklidReku

```

```

ggT(1440, 408)
ggT(408, 216)
ggT(216, 192)
ggT(192, 24)
ggT(24, 0)
return 24
return 24
return 24
return 24

```

Korrektheitsbeweis

Die Korrektheit der iterativen Variante (Paragraph 1.2-11) lässt sich ganz ähnlich beweisen.

- Für beliebige $x, y \in \mathbb{N}$ gilt $\text{ggT}(x, 0) = x$.
- Ist $y \neq 0$, so gilt für jeden Teiler t von y :

$$t \text{ teilt } x \iff t \text{ teilt den Rest } r = x \bmod y,$$

- Folglich** Ist $y \neq 0$, so $\text{ggT}(x, y) = \text{ggT}(y, x \bmod y)$.

3.4-27 Das rekursive Grundmuster

Jede rekursive Methode benötigt einen Basisfall, sonst wird die Berechnung mit `StackOverflowError` abgebrochen

Das Einfache zuerst

typischerweise programmiert man erst den Basisfall, dann die Rekursion:

“Initialisiere - Terminiere (falls trivial) - Rekurriere”

Beispiel: Primfaktorzerlegung

3.4-28

Das folgende ist eine Nachprogrammierung des Shell-Kommandos factor.

```

1  public class Factor {
2      static void factor (int n) {
3          int t = 2;
4          while (n%t != 0 && t*t < n)
5              t++;
6          if (t*t > n) {
7              System.out.print(n);
8              return;
9          }
10         System.out.print(t+" ");
11         factor(n/t);
12     }
13     public static void main(String[] args) {
14         int z = Integer.parseInt(args[0]); // zu faktorisierende Zahl
15         System.out.print(z + ": ");
16         if (z > 1)
17             factor(z);
18         System.out.println();
19     }
20 }
```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Factor.java>

→ Info1-Sammlung (II-ID: nsmdopt0t2i0)

Türme von Hanoi

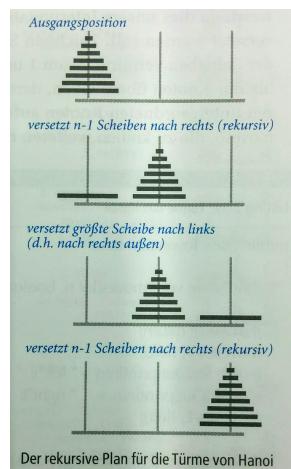
3.4-29

Aus einem alten Aufgabenbuch ...

Scheiben 1, 2, ..., n sind zu versetzen: (1) in jedem Schritt nur eine Scheibe bewegen, (2) größere Scheiben dürfen nicht auf kleineren liegen, (3) nur ein Zwischenlagerplatz erlaubt. Welche Schritte führen zum Ziel?

Rekursive Lösungsidee

Grafik aus [Sedgewick/Wayne]



Ziel: Methodenaufruf `moves (<Anzahl>, <Richtung>)` für Handlungsanweisungen dieser Form

- i rechts: lege i auf rechten Nachbarplatz/-stapel (bzw. den linken, falls wir schon ganz rechts sind)
- i links: entsprechend entgegengesetzt^a

^aoder: Stäbe in dreieckiger Anordnung, “rechts” entspricht Uhrzeigersinn, “links” entgegengesetzt

Quelltext

3.4-30

erst 1, 2, ..., n - 1 (rekursiv) nach rechts versetzen, dann n nach links und schließlich 1, ..., n - 1 nochmal nach rechts

```

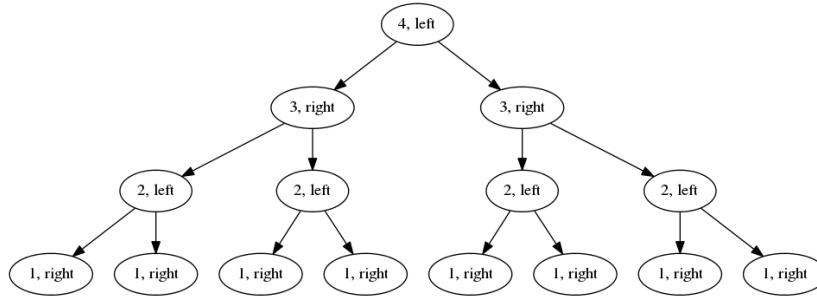
1  public class TuermeVonHanoi {
2      public static void main(String[] args) {
3          int n = Integer.parseInt(args[0]);
4          boolean left = true; // Richtungsangabe durch ..
5          moves(n, left); // .. links/rechts = true/false
6      }
7      public static void moves(int n, boolean left) { // 
8          boolean right = !left; // entgegengesetzte Richtung
9          if (n == 0) return; // Basisfall
10         moves( n-1, right);
11         if (left) System.out.println(n + " links");
12         else System.out.println(n + " rechts");
13         moves( n-1, right);
14     }
15 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/TuermeVonHanoi.java>

3.4-31 Veranschaulichung des Aufrufs moves (4, left)

Rekursionsbaum

Knoten entsprechen Aufrufen und sind markiert mit den Aufrufargumenten



3.4-32 Tracing

```

moves(4, left)
moves(3, right)
moves(2, left)
moves(1, right)
    "1 rechts"
    "2 links"
moves(1, right)
    "1 rechts"
"3 rechts"
moves(2, left)
moves(1, right)
    "1 rechts"
    "2 links"
moves(1, right)
    "1 rechts"
"4 links"
moves(3, right)
moves(2, left)
moves(1, right)
    "1 rechts"
    "2 links"
moves(1, right)
    "1 rechts"
"3 rechts"
moves(2, left)
moves(1, right)
    "1 rechts"
    "2 links"
moves(1, right)
    "1 rechts"

```

Aufrufe moves (0, *Richtung*) sind nicht gezeigt und anstelle von System.out.println(..) nur die Argumentstrings.

Das Tracing entspricht dem Besuchen aller Knoten des Rekursionsbaums (und zwar in dieser rekursiv definierten Reihenfolge, beginnend bei der Wurzel: besuche erst die Knoten des linken Teilbaums, dann den Knoten selbst, dann die Knoten des rechten Teilbaums).

Inorder Traversierung: Links – Wurzel – Rechts

Anzahl der Bewegungen

3.4-33

move (. .) ist 2-fach rekursiv:

jeder Aufruf führt (außer im Basisfall) zu zwei weiteren Aufrufen der Methode selbst. Bezogen auf die **Rekursionstiefe** = (Tiefe des Rekursionsbaums) werden exponentiell viele Aufrufe ausgelöst.

Der rekursive Algorithmus benötigt $2^n - 1$ Bewegungen. Geht es schneller?

Nein!

Jede Lösung erfordert wenigstens $2^n - 1$ Bewegungen.

Beweis: Sei a_n die Schrittzahl für n Scheiben. Offenbar ist $a_0 = 0$. Angenommen, $a_{n-1} \geq 2^{n-1} - 1$ für ein $n \geq 1$. Um n Scheiben zu versetzen, muss man die obersten $n - 1$ zwischenlagern, um an die letzte zu kommen. Macht nach Annahme $\geq 2^{n-1} - 1$ Schritte, hinzu ein weiterer für Scheibe n und noch einmal $2^{n-1} - 1$ um das Zwischenlager zu räumen. Insgesamt $a_n \geq 2^{n-1} + 1 + 2^{n-1} = 2^n - 1$.

Fibonacci-Zahlen

3.4-34

Aus einem alten Aufgabenbuch ...

Beispiel

Kaninchenpaare werfen ab 2. Lebensmonat monatlich ein neues Paar. Starten wir in Monat 1 mit einem neugeborenen Paar, wieviele haben wir in Monat n ?

Monat		Paare
0		0
1		1
2		1
3		2
4		3
5		5
6		8
7		13

Randbedingungen $f_0 = 0, f_1 = 1$

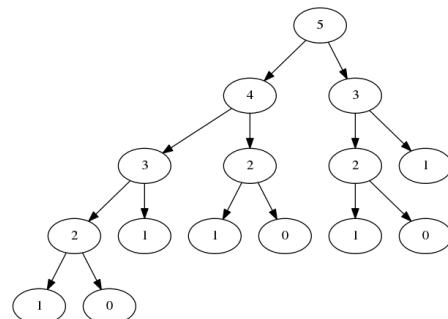
Rekurrenzgleichung $f_n = f_{n-1} + f_{n-2}$ für $n \geq 2$

Naive rekursive Implementierung

```
public static int f(int n) {
    if (n <= 1) return n;
    return f(n-1) + f(n-2);
}
```

Rekursionsbaum des Aufrufs $f(5)$

3.4-35



Oben erfordert $f(n)$ die Werte $f(n-1)$ (linker Teilbaum) und $f(n-2)$ (rechter Teilbaum), daher werden die Knoten in dieser rekursiv definierten Reihenfolge durchlaufen

Postorder Traversierung: Links – Rechts – Wurzel

→ Info1-Sammlung (II-ID: am4awa40u2i0)
→ Info1-Sammlung (II-ID: e8b3b9n056i0)

3.4-36 Geschicktere Implementierung

Idee : Wiederholungen vermeiden durch Zwischenspeichern in einem Feld

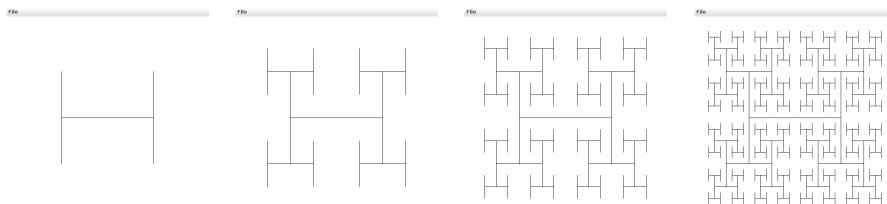
Memoisation

```
public static int f(int n) {
    int[] memo = new int[n+1];
    memo[0] = 0; if (n>0) memo[1] = 1;
    for (int i = 2; i <= n; i++)
        memo[i] = memo[i-1] + memo[i-2];
    return memo[n];
}
```

3.4-37 Ein Grafik-Beispiel

H-Bäume

sind grafische Darstellungen spezieller Bäume: Beispiel: H-Bäume der Ordnungen 1, 2, 3 und 4:



Definition

Ein H-Baum der Ordnung 0 ist leer. Ein H-Baum der Ordnung $n > 0$ besteht aus:

- 3 Strecken gleicher Länge in Form eines symmetrischen H
- 4 parallelen H-Bäumen der Ordnung $n - 1$ halber Größe, deren Mittelpunkte an den Enden des großen H's sind

Dies ergibt unmittelbar eine (vierfach) rekursive Grafikmethode für H-Bäume. Trick: neben Ordnung und Größe der H-Bäume werden die Mittelpunktskoordinaten als Parameter übergeben.

```
public class Htree {
    public static void draw(int n, double x, double y, double size) {
        if (n == 0) return; // Basisfall
        double x0 = x - size/2, x1 = x + size/2; // Koordinaten ..
        double y0 = y - size/2, y1 = y + size/2; // .. der 4 Enden
        StdDraw.line(x0, y0, x0, y1); // linkes Bein und ..
        StdDraw.line(x1, y0, x1, y1); // .. rechtes Bein vom H
        StdDraw.line(x0, y, x1, y); // "Balken"
        // zeichne rekursiv 4 kleinere H-Bäume der Ordnung n-1
        draw(n-1, x0, y0, size/2); // unten links
        draw(n-1, x0, y1, size/2); // oben links
        draw(n-1, x1, y0, size/2); // unten rechts
        draw(n-1, x1, y1, size/2); // oben rechts
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        double x = 0.5, y = 0.5; // Mittelpunktkoordinaten
        double size = 0.5; // Seitenlänge
        draw(n, x, y, size);
    }
}
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Htree.java>

Achten Sie einmal auf die Zeichenreihenfolge, z.B. beim Aufruf `java Htree 9`.

Fraktale

3.4-38

sind “selbstähnliche Strukturen” in Natur, Technik und Wissenschaft

Börsenkurse in unterschiedlicher zeitlicher Auflösung

**Brownsche Brücke**

= Zufallslinie zwischen zwei Punkten $P_0(x_0, y_0)$ und $P_1(x_1, y_1)$

Simulation Brownscher Brücken

3.4-39

Rekursives Modell

Basisfall zeichne Strecke $\overline{P_0P_1}$, falls sich x -Koordinaten nur wenig unterscheiden

Rekursionsfall • wähle **geeignet zufällig** ein δ

- verschiebe den Mittelpunkt von $\overline{P_0P_1}$ um δ längs der y -Achse.

Neue Lage:

$$P'_m\left(\frac{x_0 + x_1}{2}, \frac{y_0 + y_1}{2} + \delta\right)$$

- konstruiere Brownsche Brücken $P_0 \rightsquigarrow P'_m$ und $P'_m \rightsquigarrow P_1$.

```

1  public class Brownian {
2      // Rekursive Mittelpunktsverschiebung
3      public static void
4          curve(double x0, double y0, double x1, double y1, double var, double s) {
5              if (x1 - x0 < .005) { // Basisfall
6                  StdDraw.line(x0, y0, x1, y1);
7                  return;
8              }
9              double xm = (x0 + x1) / 2; // x- und y-Koordinaten ..
10             double ym = (y0 + y1) / 2; // .. des Mittelpunkts
11             // Zufällige Mittelpunktsverschiebung längs y-Achse:
12             double delta = StdRandom.gaussian(0, Math.sqrt(var));
13             ym = ym + delta;
14             curve(x0, y0, xm, ym, var/s, s); // rekursive Konstruktion der ..
15             curve(xm, ym, x1, y1, var/s, s); // .. beiden Kurvenabschnitte.
16         }
17         public static void main(String[] args) {
18             double H = Double.parseDouble(args[0]);
19             double s = Math.pow(2, 2*H);
20             curve(0.0, 0.5, 1.0, 0.5, .01, s);
21         }
22     }

```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Brownian.java>

- Zeile 16: `StdRandom` (siehe Paragraph 3.5-7) aus `stdlib.jar` bietet Methoden, um Zufallswerte aus verschiedenen Verteilungen zu erzeugen. Durch den `gaussian`-Aufruf wird `delta` *geeignet zufällig* gewählt — Details hier uninteressant.

`java Brownian .005` liefert schöne „Aktien-Kurse“. Probieren Sie es aus!

3.5 Modulare Programmierung

3.5.1 Ein Beispiel

nach [Sedgewick/WaSh]

SAT Scores

SAT = Scholastic Assessment Test

= Standardisierte Klausur für US-Studienbewerber
im Jahr 2000: $N \approx 1$ Mio Teilnehmer:

- im Mittel $\mu = \frac{1}{N} \sum_{i=1}^N x_i = 1019$ Punkte
- Standardabweichung $\sigma = \sqrt{\sum_{i=1}^N (x_i - \mu)^2} = 209$

Frage

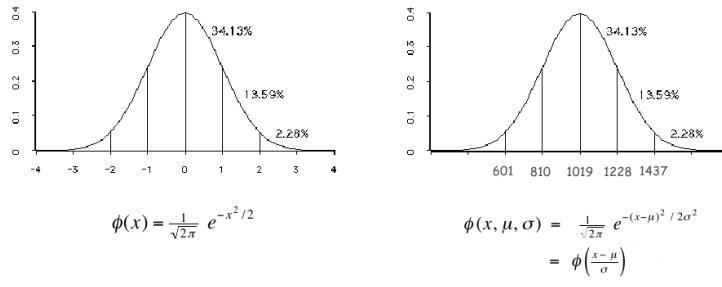
Wieviele Teilnehmer erreichten bis zu 820 Punkten?

Fakt

Bei großem N sind die Punktzahlen nahezu (μ, σ) -normalverteilt.

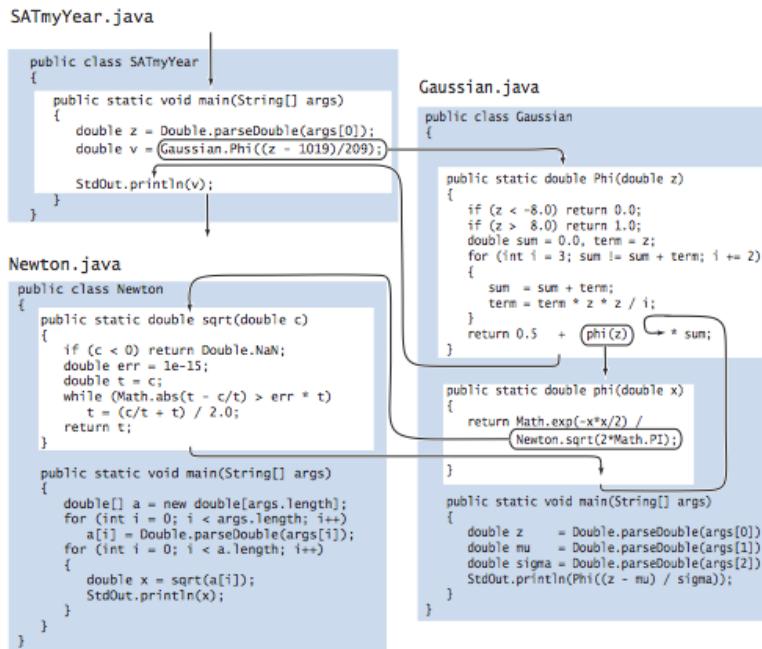
(μ, σ) -Normalverteilung

= verschobene und gedehnte Version der Standardnormalverteilung (siehe Par. 3.4-16)

**Antwort**Ungefähr der Prozentsatz $\Phi(\frac{z-1019}{209})$ hat $\leq z$ Punkte.Speziell für $z = 820$ sind das 17%, also etwa 170 000 Bewerber.**Code wiederverwenden**

3.5-2

Mit Newton im Klassenpfad können die bereits programmierten Funktionen phi, Phi, sqrt, ... einfach wiederverwendet werden, indem beim Aufruf der Klassenname vorangestellt wird: Beispiel: Newton.sqrt(2*Math.PI)

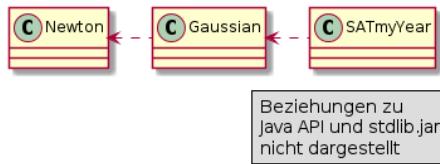


Grafik aus [Sedgewick/Wayne]

Das Schlüsselwort public

3.5-3

ClientenEine Klasse, die Dienste einer anderen Klasse C nutzt, heißt **Clientenklasse** von C.



SATmyYear ist Client von Gaussian, Gaussian wiederum von Newton.

Die Klassendefinition `public class C { ... }` ermöglicht
die Benutzung der `public`-Elemente von C durch *alle* Clientenklassen (sofern C im Klassenpfad ist) .

3.5-4 Modulare Programmierung

Ein Modul

wird durch den Java-Code in einer `*.java`-Datei definiert.
Ein Java-Programm kann so auf mehrere Klassen verteilt sein.

Paradigma der modularen Programmierung

- teile den Quellcode sinnvoll in übersichtliche Einheiten auf
- implementiere, dokumentiere und teste die Module einzeln
- implementiere die Transformation $E \xrightarrow[S]{V} A$ durch geeignete Aufrufe der angebotenen Methoden

3.5.2 Benutzerdefinierte Bibliotheken

3.5-5 Sammlungen von statischen Methoden

Idee

- thematisch zusammengehörende statische Methoden in einem Modul sammeln
- allgemeine Verwendbarkeit sichern durch
 - `public class` und `public` Methoden
 - natürliche Schnittstellenwahl
 - Dokumentation der Nutzung

Beispiel Gaussian

- bietet nützliche Methoden zur Arbeit mit der Normalverteilung
- kann weiter ausgebaut werden (z.B. Inverse der Verteilungsfunktion Φ)

Testunit (oder Testumgebung)

= main-Methode im Modul, die den gesamten Code ausführt und datengesteuerte Tests ermöglicht

Bemerkung

Der Zweck der Testunit ist nicht anwendungsbezogen, darum handelt es sich bei nutzerdefinierten Bibliotheken nicht um eigenständige Applikationen.

Beispiel 1: Die Bibliothek StdArrayIO**3.5-6**

Die im folgenden diskutierten Klassen StdArrayIO, StdStats und StdRandom gehören zu stdlib.jar. Quelltexte sind (wenn überhaupt) nur gekürzt wiedergegeben.

Methoden zur bequemen Ein- und Ausgabe von Feldern

public class StdArrayIO	
void print (double[] a)	Textausgabe von 1D double-Feld
void print (double[][] a)	Textausgabe von 2D double-Feld
double[] readDouble1D()	1D double-Feld einlesen ^a
double[][] readDouble2D()	2D double-Feld einlesen ^b

^a1D-Eingabeformat: int-Wert N gefolgt von N double-Werten

^b2D-Eingabeformat: int-Werte M, N gefolgt von $M \times N$ double-Werten

Bemerkungen

- der Bezeichner print ist überladen (siehe Paragraph 3.4-10)
- Felder werden stets mit ihrem Namen (= Anfangsadresse, siehe Paragraph 2.3-9) als Methodenargument übergeben, mehr dazu später
- die read-Methoden liefern Felder als Funktionswert

Implementierung (Ausschnitt)

```

1 public class StdArrayIO {
2     public static void print(double[] a) {
3         final int N = a.length;
4         System.out.println(N);
5         for (int i = 0; i < N; i++) {
6             System.out.printf("%9.5f ", a[i]); // 9 Stellen (5 nach Dezimalpunkt)
7         }
8         System.out.println();
9     }
10    public static double[] readDouble1D() {
11        final int N = StdIn.readInt();
12        double[] a = new double[N];
13        for (int i = 0; i < N; i++) {
14            a[i] = StdIn.readDouble();
15        }
16        return a;
17    }
18    // ...
19 }
```

Zeile 6: zu printf siehe Paragraph 3.1-11

Testunit

```

1 public class StdArrayIO {
2     // ...
3     public static void main(String[] args) {
4         double[] a = StdArrayIO.readDouble1D();
5         StdArrayIO.print(a);
6         System.out.println();
7         double[][] b = StdArrayIO.readDouble2D();
8         StdArrayIO.print(b);
9         System.out.println();
10    }
11 }
```

```

java StdArrayIO < DoubleFelder.txt
4
0.00000 0.24600 0.22200 -0.03200
4 3
0.00000 0.27000 0.00000
0.24600 0.22400 -0.03600
0.22200 0.17600 0.08930
-0.03200 0.73900 0.27000
```

3.5-7 Beispiel 2: StdStats bzw. StdRandom

Einfache Datenanalyse/-visualisierung bzw. Erzeugung geeigneter Zufallswerte

<code>public class StdStats</code>	
<code>double max (double[] a)</code>	liefert Maximum der Werte im double-Feld
<code>....</code>	analog: Mittelwert, Standardabweichung etc.
<code>void plotPoints(double[] a)</code>	Punkte an den Koordinaten <code>(i,a[i])</code>
<code>void plotLines(double[] a)</code>	Streckenzug durch alle Punkte <code>(i,a[i])</code>
<code>void plotBars(double[] a)</code>	Balken von x-Achse zu Punkten <code>(i,a[i])</code>
<code>public class StdRandom</code>	
<code>boolean bernoulli(double p)</code>	true mit Wahrscheinlichkeit p
<code>double gaussian()</code>	Standard-normalverteilte Zufallswerte
<code>double gaussian(double m, double s)</code>	(m,s)-normalverteilte Zufallswerte
<code>....</code>	weitere

Implementierung von **StdStats** (Ausschnitt)

```

public final class StdStats {
    public static double max(double[] a) {
        double max = Double.NEGATIVE_INFINITY;
        for (int i = 0; i < a.length; i++) {
            if (a[i] > max) max = a[i];
        }
        return max;
    }
    public static void plotPoints(double[] a) {
        int N = a.length;
        StdDraw.setXscale(-1, N);
        StdDraw.setPenRadius(1.0 / (3.0 * N));
        for (int i = 0; i < N; i++) {
            StdDraw.point(i, a[i]);
        }
    }
    public static void plotLines(double[] a) {
        final int N = a.length;
        StdDraw.setXscale(0, N-1);
        StdDraw.setPenRadius();
        for (int i = 1; i < N; i++) {
            StdDraw.line(i-1, a[i-1], i, a[i]);
        }
    }
    // ...
}

```

Beispiel: Illustration des zentralen Grenzwertsatzes

3.5-8

Die Summe n unabhängiger Zufallsbits (wie Münzwürfe) ist etwa $(n/2, \sqrt{n}/2)$ -normalverteilt, falls n genügend groß.

```

1  public class Bernoulli {
2      public static int binomial(int n) {
3          int heads = 0;
4          for (int i = 0; i < n; i++) {
5              if (StdRandom.bernoulli(0.5)) { // eine faire Muenze werfen
6                  heads++;
7              }
8          }
9          return heads;
10     }
11     public static void main(String[] args) {
12         int n = Integer.parseInt(args[0]); // Anzahl der Muenzen in jedem Versuch
13         int t = Integer.parseInt(args[1]); // Anzahl der Versuche
14         int[] freq = new int[n+1];
15         for (int j = 0; j < t; j++) {
16             freq[binomial(n, 0.5)]++;
17         }
18         double[] normalized = new double[n+1];
19         for (int i = 0; i <= n; i++) {
20             normalized[i] = (double) freq[i] / t;
21         }
22         StdStats.plotBars(normalized);
23         // (approximierte) Normalverteilung dazu plotten
24         // ...
25     }
26 }

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Bernoulli.java>

Aufruf z.B. mit `java Bernoulli 20 10000`

Zeilen 14–17 zählt die verschiedenen möglichen Versuchsergebnisse (siehe Zähltrick aus Paragraph 2.3-7): Zunächst wird `freq` mit 0-Einträgen initialisiert (noch kein Versuch, also alle Anzahlen Null). In der Schleife wird der Versuch (also das Werfen von n Münzen) wiederholt ausgeführt, dabei das Ergebnis (Anzahl der erhaltenen "Köpfe") als Feldindex verwendet und der dortige Feldeintrag inkrementiert.

Dieser Code nutzt die benutzerdefinierten Bibliotheken `StdDraw`, `StdRandom`, `Gaussian` und `StdStats`.

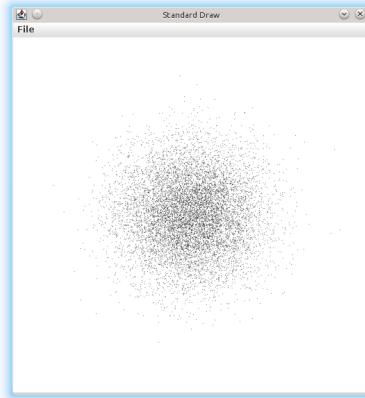
3.5-9 Demonstration von StdRandom

```
1 public class RandomPoints {  
2     public static void main(String[] args) {  
3         int N = Integer.parseInt(args[0]);  
4         double mean = .5;  
5         double stdv = .1;  
6         if (args.length == 3) {  
7             mean = Double.parseDouble(args[1]);  
8             stdv = Double.parseDouble(args[2]);  
9         }  
10        for (int i = 0; i < N; i++) {  
11            double x = StdRandom.gaussian(mean, stdv);  
12            double y = StdRandom.gaussian(mean, stdv);  
13            StdDraw.point(x, y);  
14        }  
15    }  
16}
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/RandomPoints.java>

Der Aufruf `java RandomPoints 10000 0.5 0.1` plottet 10000 zufällige Punkte, die mit einer Streuung (Standardabweichung) 0.1 normalverteilt um Punkt (0.5, 0.5) sind.

Das Plotten von Punkten mit zufälligen Koordinaten ist ein beliebter Qualitätstest für (Pseudo-)Zufallsgeneratoren.



Kapitel 4

Strukturierte Daten

4.1 Weiteres zu Feldern

4.1.1 Einige Besonderheiten

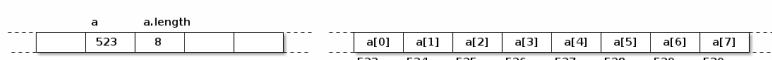
Erinnerung

4.1-1

Feldvariable und Feldinhalt

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

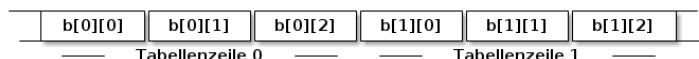
Die **Feldvariable** *a* beinhaltet die (Anfangs-)Adresse des Felds. Der **Feldinhalt** sind die im Feld gespeicherten Werte.



Bisherige Vorstellung von mehrdimensionalen Feldern

Zweidimensionale Felder aus eindimensionalen Feldern (mehrdimensionale entsprechend):

```
int[][] b = new int[2][3]; // 2 x 3 -Tabelle  
System.out.println(b.length); // Ausgabe: 2
```

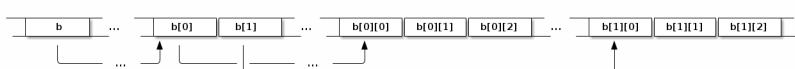


Präzisierung

4.1-2

Feldvariable bei mehrdimensionalen Feldern

Die Feldvariable beinhaltet die Adresse eines Felds, dessen Komponenten Adressen von Feldern beinhalten.



Alternative: zeilenweises Anlegen des Felds

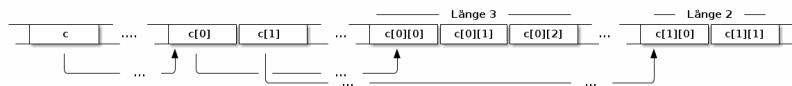
```
int[][] b = new int[2][];
b[0] = new int[3];
b[1] = new int[3];
```

- Die Speicherung und Adressberechnung für Einträge mehrdimensionaler Felder ist somit in Wirklichkeit etwas anders, als in Paragraphen 2.3-17 und 2.3-20 dargestellt. Dies beeinflusst zwar den Specheraufwand, nicht aber die Zugriffsgeschwindigkeit.

Ungleichförmige mehrdimensionale Felder

Es ist nicht notwendig, dass die Komponentenfelder gleiche Länge haben:

```
int[][] c = new int[2][];
c[0] = new int[3];
c[1] = new int[2];
```



4.1-3 Der Heap

Wozu brauchen wir new?

- Da die Feldlänge u.U. erst zur Laufzeit feststeht, kann auch erst dann Speicher zugeordnet werden.

```
int n = Integer.parseInt(args[0]);
int[] a = new int[n]; // Speicherzuordnung
```

- Der Wert des Ausdrucks `new int[n]` ist eine **Referenz** auf das neu angelegte Feld und sie wird in `a` abgelegt.

Was ist eine Referenz und wo liegt sie im Speicher?

Die Vorstellung **Referenz = Adresse** genügt (im Detail ist es etwas anders). Ist `a` zum Beispiel eine lokale Variable, so “lebt” sie im Laufzeitstapel.

Wo wird der Feldinhalt gespeichert?

In einem besonderen Speicherbereich, dem **Heap**

4.1-4 Änderbarkeit

Feldinhalt? JA

kann ohne weiteres geändert werden

```
int[] a = new int[2];
a[0] = 1;
```

Länge? NEIN

Feldlänge steht nach Speicherzuordnung fest und kann nicht überschrieben werden

```
a.length = 3; // Compilezeitfehler
```

Referenz? JA

kann ohne weiteres geändert werden

```
a = new int[3];
b = new int[3]; a = b; // Aliasing (siehe Kap. 2)
```

4.1.2 Details zu Feldern als Methodenargumente

Wirkprinzip

4.1-5

Erinnerung

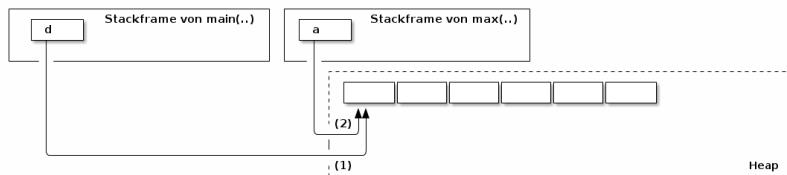
Feldargumente werden bei Methodendeklaration und -aufruf ohne Länge angegeben

siehe Paragraph 3.4-20

```
public static double max(double[] a) {
    // ...
    System.out.println("Maximum: " + max(d));
```

Call-by-value-Prinzip:

- beim Methodenaufruf werden Werte der (1) Aufrufparameter in die (2) Formalparameter kopiert (siehe Paragraph 3.4-13)
- bei Feldern als Argumenten bedeutet das: deren Anfangsadresse wird kopiert

**Seiteneffekte bei Feldargumenten**

4.1-6

- da Felder per Referenz übergeben werden, kann der Feldinhalt innerhalb der Methode überschrieben werden!!
- entsprechende Methoden sind typischerweise vom Typ `void` (liefern keinen Wert, aber bewirken Speicheränderung)

Erklärung liefert Paragraph 2.3-12

Beispiel: Platztausch

```
1 public class Array {
2     public static void swap( double[] a, int i, int j) {
3         double tmp = a[i]; a[i] = a[j]; a[j] = tmp;
4     }
5 }
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Array.java>

Beispiel: Sortieren

Siehe Paragraph 3.1-5: nach dem Aufruf `java.util.Arrays.sort(f)` sind die Werte im Feld aufsteigend sortiert.

4.1.3 Zwei einfache Sortierverfahren

4.1-7 Selection-Sort: Minima nach vorn tauschen

Phase $i \in \{0, 1, \dots, N-2\}$: Anfangsstück $\underbrace{\square \square \dots \square}_{i} \blacksquare \underbrace{\square \square \dots \square}_{N-i} \blacksquare \square \dots \square$ ist sortiert

$f_i \blacksquare$ und Minimum $f_m \blacksquare$ von $[f_i, f_{i+1}, \dots, f_{N-1}]$ austauschen

Anfangsstück $\underbrace{\square \square \dots \square}_{i+1} \blacksquare \underbrace{\square \square \dots \square}_{N-i-1} \blacksquare \square \dots \square$ ist sortiert

Phase $N-1$: Nichts mehr zu tun — das Feld ist sortiert.

```

1 public class Selection {
2     public static void sort(double[] f) { // Selectionsort
3         final int N = f.length;
4         for (int i = 0; i < N-1; i++) {
5             int m = i;
6             for (int j = i+1; j < N; j++) // Index m des Minimums von ..
7                 if (f[j] < f[m])           // .. [f[i], f[i+1], ..., f[N-1]] ..
8                     m = j;               // .. bestimmen und ..
9             Array.swap(f, m, i); // .. nach Position i tauschen
10        }
11    }
12 }
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Selection.java>

4.1-8 Wieviele Vergleiche zwischen Feldelementen sind nötig?

- innere Schleife bestimmt Position des Minimums in $[f_i, \dots, f_{N-1}]$
- erfordert $N - 1 - i$ Vergleiche
- insgesamt $(N - 1) + (N - 2) + \dots + 2 + 1 = \frac{(N-1)N}{2} \approx N^2/2$ Vergleiche

Zusammenfassung

Anzahl der Vergleiche ist *unabhängig von der Eingabe*

- Selection-Sort erfordert in jedem Fall **quadratischen Aufwand**

4.1-9 Insertion-Sort: In Anfangsabschnitte sortiert einfügen

Phase 0: Nichts zu tun — Anfangsstück $\underbrace{f_0}_1$ ist bereits sortiert.

Phase $i \in \{1, \dots, N-1\}$: Anfangsstück $\underbrace{\square \square \dots \blacksquare \square \square}_{i-1} \blacksquare \square \dots \square$ ist schon sortiert.

Füge i -tes Element ■ an richtiger Position ein: .

Nach Phase $N - 1$ ist das Feld sortiert.

Die richtige Position für f_i ■ finden

Nach links „durchtauschen“ solange ■ < linker Nachbar ist.

```

1 public class Insertion {
2     public static void sort(double[] f) { // Insertionsort
3         final int N = f.length;
4         for (int i = 1; i < N; i++) { // f[0], ..., f[i-1] ist bereits sortiert
5             // f[i] an richtiger Stelle einfuegen, d.h. soweit noetig ..
6             for (int j = i; j > 0 && f[j] < f[j-1]; j--) { // .. nach links ..
7                 Array.swap(f, j, j-1); // .. "durchtauschen".
8             }
9         }
10     }
11 }

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Insertion.java>

Wieviele Vergleiche zwischen Feldelementen sind nötig?

4.1-10

- innere Schleife bestimmt die Einfügeposition von f_i innerhalb von i bereits sortierten Werten
- erfordert mindestens einen, höchstens i Vergleiche
- insgesamt zwischen $(N - 1)$ und $1 + 2 + \dots + (N - 1) \approx N^2/2$ Vergleiche

Zusammenfassung

bester Fall	$N - 1$ Vergleiche
schlechtester Fall	$\frac{(N-1)N}{2}$ Vergleiche
im Mittel	$\frac{(N-1)N}{4}$ Vergleiche

Anzahl der Vergleiche $V(N)$ ist abhängig von der Eingabe: ist diese bereits aufsteigend sortiert, so ist $V(N)$ minimal und bei absteigender Sortierung maximal.

Bemerkung

- wird Einfügeposition mit binärer Suche bestimmt, genügen $\approx N \ln N$ Vergleiche
- Einfügen selbst erfordert bis zu $\frac{(N-1)N}{2}$ Tauschoperationen

Aufwand bleibt im schlechtesten Fall **quadratisch**

4.2 Verbunddaten

4.2.1 Definition, Erzeugung und Zugriff

Was sind Verbunddaten?

4.2-1

Ein **Verbund** (verbreiterteres Synonym: Datensatz) ist ein Datenobjekt, das eine fixierte Anzahl von Komponenten beliebigen Typs zusammenfasst.

Beispiel: Studentendaten

- Name, Matrikelnummer und Studienfach beisammen speichern

Fritze Bollmann	20091234	Biologie
name	matnr	fach

- ermöglicht u.a. Algorithmen für
 - gegeben: Name, gesucht: Matrikelnummer (oder umgekehrt)
 - gegeben: Liste solcher Daten, gesucht: Sortierung (nach Matrikelnummer oder alphabetisch nach Name)

Deklaration

- die gewünschte gemeinsame Struktur solcher Datenobjekte beschreibt man durch eine Klasse (“Bauplan”)
- jedes einzelne Datenobjekt heißt **Instanz** der Klasse (“ein verwirklichter Bauplan”)

Quellcode

```
public class Student{
    public String name; // Instanzvariablen ..
    public int matrikelnr; // .. (auch Instanz- ..
    public String fach; // .. attribute genannt )
    ...
}
```

Instanzvariablen haben Klassenscope (siehe Paragraph 3.4-14), d.h. sie sind in der ganzen Klasse gültig

4.2-2 Konstruktorddefinition

Verbunddaten sind Referenzdaten

- ähnlich wie Feldinhalte werden Verbunddaten im Heap gespeichert
- Speicher wird erst zur Laufzeit zugeordnet durch Aufruf eines **Konstruktors**

Syntax wie bei Methoden, außer:

- *kein Ergebnistyp* (demzufolge auch *kein return <Ausdruck>*) auch *kein static*

- *Name = Klassenname*

```
public class Student{
    ...
    public Student( String n, int m, String f) {
        name = n;
        matrikelnr = m;
        fach = f;
    }
}
```

Konstruktoraufruf und Zugriff auf die Daten

4.2-3

Liegt `Student` im Klassenpfad, so sind (da alles `public` deklariert ist) in Methoden jeder Klasse Zugriffe wie diese möglich:

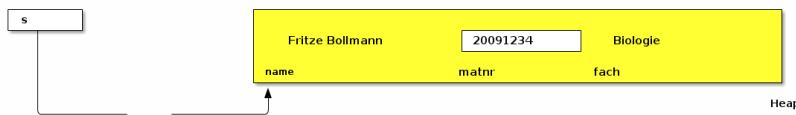
```
1     Student s, s2;
2     ...
3     s = new Student("Fritze Bollmann",20091234,"Biologie");
4     ...
5     System.out.println(s.name); // Zugriff auf Attribut dieser Instanz
```

Zeile 3: wie beim Anlegen von Feldern ist der Operator `new` notwendig, um eine Referenz auf das Datenobjekt zu erhalten

Speicherbelegung vor dem Konstruktoraufruf



Speicherbelegung nach dem Konstruktoraufruf



Bessere Formulierung des Konstruktors

4.2-4

```
public Student(String name, int matrikelnr, String fach) {
    this.name = name;           // Formalparameter verdecken ..
    this.matrikelnr = matrikelnr; // .. gleichnamige Attribute!
    this.fach = fach;           // 'this' macht sie wieder unterscheidbar.
}
```

Die `this`-Referenz

`this` ist eine Referenz auf das aktuelle Datenobjekt (hier also dasjenige, das durch den Konstruktoraufruf angelegt wurde)

Besonderheiten ähnlich wie bei Feldern

4.2-5

Fehlerquellen

```
public static void main(String[] args) {
    Student s;
    s.name = "Fritze Bollmann"; // Datenobjekt noch nicht erzeugt!
    s.telefon = 12345; // Attribut nicht definiert
}
```

Alias-Effekt

Vgl. Paragraph 2.3-12

```
Student s = new Student("Fritze Bollmann", 20091234, "Biologie");
Student s2 = new Student("Crack Nerd", 20091234, "Angew. Informatik");
s = s2; // Fritze ist weg!
```

Verbundreferenzen als Methodenparameter

```
public static void studiengangswechsel(Student s, String neu) {
    s.fach = neu;
}
```

4.2-6 Beispiel: Feld von Verbunddaten

```
1 public class StudentenFeld {
2     public static void main(String[] args) {
3         final int N = Integer.parseInt(args[0]);
4         Student[] teiln = new Student[N];
5         for (int i = 0; i < N; i++) {
6             teiln[i] = // Konstruktoraufruf mit einzulesenden Daten:
7                 new Student(StdIn.readString(), StdIn.readInt(), StdIn.readString());
8         }
9         sortiereNachMatrikelnr(teiln);
10        for (int i = 0; i < N; i++) {
11            Student t = teiln[i];
12            System.out.println(t.name + " " + t.matrikelnr + " " + t.fach);
13        }
14    }
15    public static void sortiereNachMatrikelnr(Student[] f) { // BubbleSort
16        boolean swapped;
17        do { swapped = false; // Flag loeschen
18            for (int i = 0; i < f.length - 1; i++) {
19                if (f[i].matrikelnr > f[i + 1].matrikelnr) {
20                    Student t = f[i]; f[i] = f[i + 1]; f[i + 1] = t; // Tausch
21                    swapped = true; // Flag setzen
22                }
23            }
24        } while (swapped);
25    }
}
```

Man beachte, dass die Sortiermethode hier ebenso schnell (oder langsam) ist wie die in Paragraph 2.3-7, denn es werden nur Referenzen getauscht, nicht die Verbunddaten selbst.

4.2-7 Beispiel: Verbund von Verbunddaten

```
1 public class Datum {
2     int tag; int monat; int jahr;
3     public Datum (int tag, int monat, int jahr){ // Konstruktor mit 3 Parametern
4         this.tag = tag; this.monat = monat; this.jahr = jahr;
5     }
6     public Datum (int jahr){ // Konstruktor mit einem Parameter
7         this.tag = 1; this.monat = 1; this.jahr = jahr;
8     }
9 }
```

In Zeilen 6–8 sehen wir eine Konstruktorüberladung.

```

1  public class Person {
2      String vorname; String nachname;
3      private Datum geb_datum;
4      public Person (String vorname, String nachname,
5                      int tag, int monat, int jahr) { // Geburtsdatum
6          this.vorname = vorname; this.nachname = nachname;
7          geb_datum = new Datum(tag,monat,jahr);
8      }
9      public static void main(String[] args) { // Test-Unit
10         Person p = new Person("Georg Christoph", "Lichtenberg", 1, 7, 1742);
11         System.out.println(p.geb_datum.jahr); // <---
12         System.out.println(p.alter()); // <---
13     }
}

```

Man beachte den erforderlichen Konstruktorauftruf in Zeile 7!

Zugriffsmodifizierer

4.2-8

Schlüsselwörter `private`, `public` oder `protected` steuern die Sichtbarkeit von *Klassenelementen* (Attribute, Methoden, Konstruktoren, ..) vom Clientcode aus.

`public`-Elemente

sind von jeder anderen Klasse im Klassenpfad sichtbar

`private`-Elemente

sind außerhalb der definierenden Klasse nicht sichtbar Beispiel: `geb_datum` von `Person`-Objekten

- Clients können `private`-Daten weder direkt überschreiben noch lesen
- Stattdessen: Bereitstellung von `public`-”Instanz-Methoden”, die in vorgeschriebener Weise auf die Attribute zugreifen (behandeln wir später)

Bemerkungen

`protected`-Elemente sowie Elemente ohne Zugriffsmodifizierer sind nur von „bevorrechtigten“ Klassen aus zugreifbar (die Details benötigen und behandeln wir hier nicht, sondern betrachten `protected` im weiteren gleichwertig zu `public`). Auch Klassendefinitionen können mit Zugriffsmodifizierern ausgestattet werden (siehe auch Paragraph 1.3-6). Einschränkungen beziehen sich dann auf alle Klassenelemente.

4.2.2 Ein Geometrie-Beispiel (Anfang)

Erinnerung

4.2-9

Ebene Geometrie

- durch zwei Punkte verläuft genau eine Gerade
- zwei nicht-parallele Geraden schneiden sich in genau einem Punkt

Wie kann man einen Punkt beschreiben?

Durch ein Paar (x,y) von Koordinaten (double-Werte).

Wie kann man eine Gerade beschreiben?

Z.B. durch ein Paar von double-Werten (m, n). Die Gerade besteht aus Punkten (x, y) mit $y = mx + n$. ($m = \text{Anstieg}$, $(0, n) = \text{Schnittpunkt mit } y\text{-Achse}$. Sonderfall: Ist Gerade parallel zur y -Achse, so sei $m = +\infty$ und $(n, 0)$ Schnittpunkt mit der x -Achse.

Name

Optional können Punkte und Geraden *benannt* werden (String-Wert)

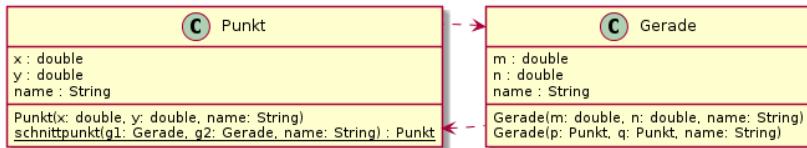
4.2-10 Ein Klassenentwurf (noch stark verbesserungswürdig)

Konstruktoren und Methoden für Punkte

```
Punkt(double x, double y, String name)
static Punkt schnittpunkt(Gerade g1, Gerade g2, String
name) analog ohne Namensangabe (Default-Name = leerer String)
```

Konstruktoren für Geraden

```
Gerade(double m, double n, String name)
Gerade(Punkt p, Punkt q, String name)
```



4.2-11 Java-Code

```

1 public class Punkt {
2     final double x, y;
3     final String name;
4     public Punkt(double x, double y, String name) {
5         this.x = x; this.y = y; this.name = name;
6     }
7     public Punkt(double x, double y) {
8         this.x = x; this.y = y; this.name = "";
9     }
10    public static Punkt schnittpunkt(Gerade g1, Gerade g2, String name) {
11        // ...
12    }

```

Zeile 2, 3: Änderungen an Punkten sind nicht vorgesehen, daher final

```

1 public class Gerade {
2     final double m, n;
3     final String name;
4     public Gerade(double m, double n, String name) {
5         this.m = m; this.n = n; this.name = name;
6     }
7     public Gerade(double m, double n) {
8         this(m, n, ""); // ruft anderen Konstruktor dieser Klasse auf
9     }
10    public Gerade(Punkt p, Punkt q, String name) {
11        // ...
12    }
13    public Gerade(Punkt p, Punkt q) {
14        this(p, q, ""); // ruft anderen Konstruktor dieser Klasse auf
15    }
16    // ...

```

Zeile 8: Verwendung der this-Referenz (s. Par. 4.2-3) in einem anderen Kontext.

this (<Parameter>)

- ist nur als erste Anweisung in einem Konstruktor erlaubt
- bezeichnet Aufruf eines anderen Konstruktors dieser Klasse. Welchen? Den, der zu den Parametern im Aufruf passt. Dazu muss dieser natürlich irgendwo im Code definiert sein.
- der Compiler verbietet innerhalb eines Konstruktors jede andere Form des Aufrufs anderer Konstruktoren dieser Klasse (wie etwa Punkt (x, y,))

4.2.3 Stringumwandlung

Textausgabe von Referenzdaten

4.2-12

```
public class HelloReference {
    public static void main(String[] args) {
        int[] a = { 0, 1, 2 };
        int[][] b = {{ 0, 1, 2}, {3, 4, 5}};
        Punkt p = new Punkt(2.0,1.0,"p");
        System.out.println(a);
        System.out.println(b);
        System.out.println(p);
    }
}
```

Ausgabebeispiel:

```
[I@fbb41d7e
[[I@60d70b42
Punkt@4d871a69
```

Erklärung

Es wird ein "Identifikator" ausgegeben: der Klassenname (oder eine Beschreibung davon) zusammen mit einem hexadezimalen int-Wert = Hashcode des Datenobjekts.

Der Hashcode ist typischerweise eine Art Prüfsumme der Speicheradresse

Informative Objektbeschreibungen erzeugen

4.2-13

nützlicher als Hashcode: textuelle Wertbeschreibung, z.B. Punkt p(2.0,1.0)

toString() -Methoden bei strukturierten Daten

```
public class Punkt {
    // ...
    public String toString() {                                // KEIN static !!!!
        return "Punkt " + name + "(" + x + "," + y + ")";
    }
    public static void main(String[] args) {
        Punkt p = new Punkt( 2.0, 1.0, "p");
        System.out.println(p);                                // Ausgabe: "Punkt p(2.0,1.0)"
    }
}
```

```
public class Gerade {
    // ...
    public String toString() {
        // ...
        return "Gerade " + name + "(" + m + "," + n + ")";
    }
    // ...
}
```

4.2-14 Test-Unit

```

1  public class Gerade {
2      // ...
3      public static void main(String[] args) { // Test-Unit
4          // ...
5          final int N = Integer.parseInt(args[0]);
6          // Vorbereitung: N mal Koordinaten einlesen, Punkt erzeugen
7          Punkt[] p = new Punkt[N]; // ..... for-Schleife zum Einlesen ..
8          // fuer jedes Paar von Punkten aus der Liste ..
9          // .. erzeuge Gerade durch die Punkte und ..
10         // .. gebe sie als String aus:
11         Gerade g;
12         for (int i = 0; i < N; i++) {
13             for (int j = i+1; j < N; j++) {
14                 g = new Gerade(p[i],p[j],"durch "+p[i]+" und "+p[j]
15                             + ": ");
16                 System.out.println(g);
17             }
18         }
19     }

```

Zeilen 13, 14 verketten `p[i]` und `p[j]` mit Strings. Implizit wird dabei die `toString()`-Methode mit dem Objekt aufgerufen.

Beispiel: Aufruf `echo 1 1 2 4 | java Gerade 2` liefert Ausgabe
 Gerade durch Punkt P0(1.0,1.0) und Punkt P1(2.0,4.0) : (3.0,-2.0)
 Das bedeutet: Die Gerade durch die beiden Punkte hat Anstieg 3.0 und y-Achsenabschnitt -2.

4.2.4 Verkettete Datenstrukturen

4.2-15 Statische und dynamische Datenstrukturen

Statische Datenstrukturen

Elementare und Feld- oder Verbunddaten haben veränderliche Werte aber Struktur und Größe sind *statisch*, d.h. bleiben über die gesamte Laufzeit gleich:

- `int x = 42;` — 4 aufeinanderfolgende Bytes
- `int[] f = new int[n];` — n aufeinanderfolgende int-Speicherzellen
- `Punkt p = new Punkt(...);` — Verbund verschiedener Komponenten

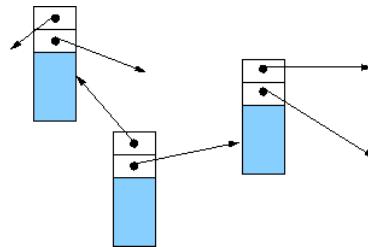
Dynamische Datenstrukturen

bestehen aus gleichartigen statischen Datenobjekten (*Knoten*) veränderlicher Anzahl und Beziehungsstruktur, Beispiele:

- Listen veränderlicher Länge
- baumartige Strukturen (linker/rechter Nachfolger ...)
- Netzwerke beliebiger Nachbarschaften

4.2-16 Genereller Ansatz

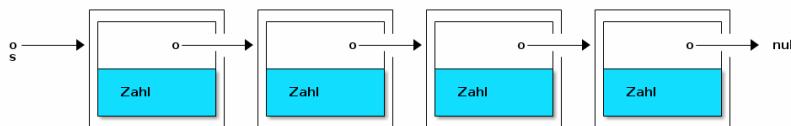
Knoten sind Verbunde aus „Nutzdaten“ von einem Grundtyp T und Referenzen, die die Verkettung angeben.



Beispiel: Speicherung in einfacher verketteter Liste

4.2-17

- Zahlen werden eingelesen mittels `StdIn.readDouble()`, solange `StdIn.isEmpty() == false`
- Speicherung in Feld wäre möglich
`double[] zahlen = new double[MAXIMUM];`
- Aber wie ist MAXIMUM zu wählen???
- Ausweg: Speicherung als **einfach verkettete Liste**



Knotentyp und Verkettungsstruktur

4.2-18

```

1 class ZahlKnoten {
2     double zahl;
3     ZahlKnoten next;
4     public ZahlKnoten (double zahl, ZahlKnoten next) {
5         this.zahl = zahl;
6         this.next = next;
7     }
8 }
```

Zeile 3: Die Klasse nimmt auf sich selbst Bezug! Das geht, denn `next` ist selbst kein Datenobjekt, sondern nur eine Referenz — der Speicherbedarf für Referenzen ist für alle Datenobjekte gleich.



Die Liste aufbauen ...

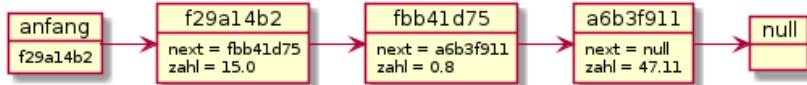
4.2-19

```

1 public class Zahlspeicher {
2     private static ZahlKnoten anfang = null;
3     public static void speichere( double zahl) {
4         anfang = new ZahlKnoten( zahl, anfang);
5     }
6     public static void main(String[] args) {
7         while (!StdIn.isEmpty()) {
8             speichere( StdIn.readDouble()); // Einlesen: 47.11, 0.8, 15.0, ...
9         }
10    // ...
11 }
```

Zeile 2: `anfang` ist eine **statische Variable**, d.h. global in der Klasse gültig und nur einmal vorhanden (anders als Instanzvariablen). Sie wird mit `null` initialisiert. `null` ist ein Literal vom Referenztyp, das sich auf kein Objekt bezieht.

Nur die Daten des Anfangsknotens sind im Quelltext direkt zugreifbar (nämlich über die Referenz `anfang`), die anderen nur durch Nachverfolgen der Referenzenkette.



Beobachtung: Die Daten sind schließlich in umgekehrter Reihenfolge in der Liste.

4.2-20 ... und traversieren

```

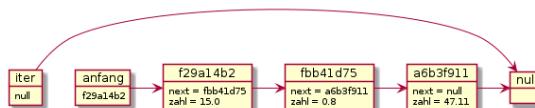
1 // ...
2 ZahlKnoten iter = anfang;
3 while (iter != null) {
4     System.out.println(iter.zahl);      // Ausgeben: ..., 15.0, 0.8, 47.11
5     iter = iter.next;
6 }
7 }
```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Zahlspeicher.java>

Initialisierung:



Erreichen der Abbruchbedingung:



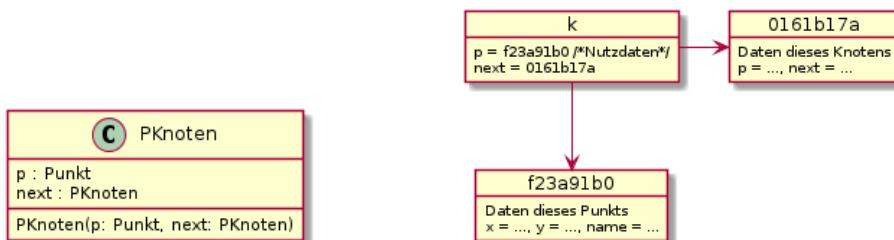
4.2-21 Anwendungen: Verarbeitung unbekannter Anzahl von Daten

Clientenklasse von Punkt und Gerade

PunktGeradeClient.java

erlaubt Operation mit *beliebig vielen* geometrischen Objekten durch "InventarListen"
Beispiel: PKnoten speichern Punkt-Referenzen als Nutzdaten, pInventar zeigt auf Anfang einer Liste von PKnoten

Knotentyp und Verkettungsstruktur



Sortierfilter

neu ankommende Daten nach Größe einordnen

Beispiel: Zahlen sortieren

4.2-22

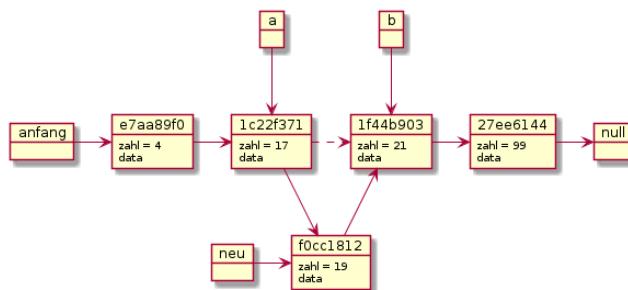
```

1 public class Sort {
2     private static ZahlKnoten anfang = null;
3     public static void insert( double zahl) {
4         ZahlKnoten neu = new ZahlKnoten( zahl, null);
5         ZahlKnoten a = null, b = anfang;
6         while ( b != null && b.zahl < neu.zahl ) {
7             a = b; b = b.next;
8         }
9         if ( a == null) anfang = neu; else a.next = neu;
10        neu.next = b;
11    }
12 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Sort.java>

Das Verfahren ähnelt dem Insertion-Sort von Paragraph 4.1-9 und scheint sogar etwas einfacher, da in natürlicher Richtung nach der Einfügestelle gesucht wird und so das "Durchtauschen" entfällt: der neue Knoten wird durch Zeilen 9 und 10 einfach an der richtigen Stelle "eingehängt".

Unten: Das Ergebnis des Aufrufs `insert(19)`, angewendet auf die bereits mit den Werten 17, 4, 21, 99 aufgebauten Datenstruktur.



4.3 Referenzen als Ergebniswerte

4.3.1 Felder als Ergebniswerte

Beispiel: Kopieren des Feldinhalts

4.3-1

Erinnerung: die Feldvariable beinhaltet die Referenz (Anfangsadresse) des Feldes und seine Länge, darüber ist der Inhalt zugreifbar

Folglich: Rückgabe mittels `return <Referenz>` (ohne Längenangabe)

```

public static double[] copy1D( double[] orig) {
    double[] c = new double[ orig.length];
    for ( int i=0; i < orig.length; i++)
        c[i] = orig[i];
    return c;
}
public static double[][] copy2D( double[][] orig) {
    double[][] c = new double[ orig.length][];
    for ( int i=0; i < orig.length; i++) {
        int l = orig[i].length;
        c[i] = new double[l];
        for ( int j = 0; j < l; j++)
            c[i][j] = orig[i][j];
    }
    return c;
}
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Array.java>

4.3-2 Matrixprodukt

```
1 public static double[][] multiply( double[][] a, double[][] b ) {
2     int n = a[0].length;
3     if ( n != b.length )
4         throw new RuntimeException("Formate passen nicht!");
5     int z = a.length, s = b[0].length;
6     double[][] c = new double[z][s];
7     for ( int i = 0; i < z; i++ )
8         for ( int j = 0; j < s; j++ ) {
9             c[i][j] = 0; // Akkumulator
10            for ( int k = 0; k < n; k++ )
11                c[i][j] += a[i][k] * b[k][j];
12        }
13    return c;
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Array.java>

Zeilen 3-4: Passen die Matrixformate nicht, so gibt es keine Möglichkeit, dies dem Aufrufer mitzuteilen. Da Weiterrechnen sinnlos ist, wird eine Laufzeitausnahme "geworfen": Programmausführung wird sofort beendet, der übergebene String erscheint im Stacktrace der JVM.

4.3.2 Verbundreferenzen als Ergebniswerte

4.3.4 Abläufe bei Rückgabe einer Referenz an den Aufrufer

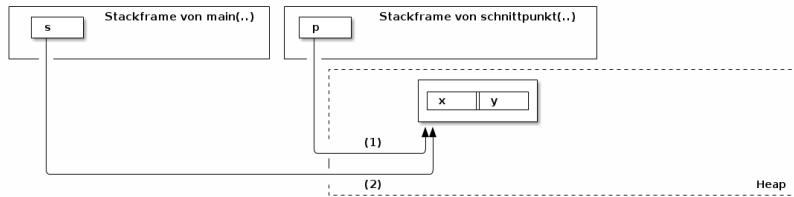
siehe Paragraph 4.1-5 ähnlich wie bei Übergabe von Referenzen an Methoden

Beispiel: Schnittpunkt berechnen

Parameter der beteiligten Geraden beschaffen, dann

(1) Schnittpunkt-Koordinaten berechnen, Punkt-Konstruktor aufrufen und (2) mit new beschaffte Referenz als Rückgabewert ausliefern

Veranschaulichung



4.3-5 Fallunterscheidung zur Schnittpunktberechnung

wir betrachten Geraden $g_{1,2}$ mit Gleichungen $y = m_1x + n_1$ und $y = m_2x + n_2$

Fallunterscheidung

parallele Geraden: es gibt keinen Schnittpunkt

sonst: Gleichungslösen

Fakt

- Geraden sind genau dann parallel, wenn $m_1 = m_2$
 - Sind die Geraden nicht parallel und Anstiege m_1, m_2 endlich, dann ist der Schnittpunkt eindeutig gegeben durch die Koordinaten $(x_0, y_0) = \left(-\frac{n_1 - n_2}{m_1 - m_2}, -m_1 \cdot x_0\right)$. Ist dagegen $m_i = \infty$ (für $i \in \{1, 2\}$), so sind die Schnittpunktkoordinaten $(x_0, y_0) = (n_i, m_{3-i} \cdot n_i + n_{3-i})$

Java-Code

4.3-6

```

public class Punkt {
    final double x, y;
    final String name;
    public Punkt(double x, double y, String name) {
        this.x = x; this.y = y; this.name = name;
    }
    // ...
    public static Punkt schnittpunkt(Gerade g1, Gerade g2, String name) {
        double m1 = g1.m, n1 = g1.n,
               m2 = g2.m, n2 = g2.n;
        if (m1 == m2)
            throw new RuntimeException("Schnittpunkt nicht definiert!");
        if (m1 == Double.POSITIVE_INFINITY)
            return new Punkt(n1, m2 * n1 + n2, name);
        if (m2 == Double.POSITIVE_INFINITY)
            return new Punkt(n2, m1 * n2 + n1, name);
        else {
            double x0 = -(n1 - n2) / (m1 - m2), y0 = m1 * x0 + n1;
            return new Punkt(x0, y0, name);
        }
    }
}

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie0/Punkt.java>

Fallunterscheidung um Gerade durch P, Q zu ermitteln

4.3-7

identische Punkte: Gerade nicht definiert

verschiedene Punkte, gleicher x -Wert: Geradenanstieg $+\infty$

sonst: Gleichungslösen

```

public class Gerade {
    //..
    public Gerade(Punkt p, Punkt q, String name) {
        if (p.x == q.x && p.y == q.y)
            throw new RuntimeException("Gerade nicht definiert!");
        if (p.x == q.x) {
            this.m = Double.POSITIVE_INFINITY;
            this.n = p.x;
            this.name = name;
            return;
        }
        this.m = (p.y - q.y) / (p.x - q.x);
        this.n = p.y - this.m * p.x;
        this.name = name;
    }
}

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie0/Gerade.java>

4.4 Speicherorganisation

Lebenszeit von Bezeichnern

4.4-1

Erinnerung: Gültigkeitsbereich (Scope) eines Bezeichners

= Quelltextbereich, in dem seine Deklaration wirksam ist (steht zur *Compilezeit* fest)

Lebenszeit eines Bezeichners

= Zeit, in der er physikalischen Speicher belegt (liegt innerhalb der *Laufzeit*)

Abläufe, die die Speicherung beeinflussen

Compilezeit Compiler legt anhand der Deklarationen relative Adressen fest (z.B. bzgl. Anfangsadresse eines Speicherbereichs)

Ladezeit statischer Code wird in Hauptspeicher geladen (z.B. werden statischen Variablen absolute Speicheradressen zugeordnet)

- Laufzeit**
- statischer Code bleibt im Speicher
 - bei jedem Methodenaufruf wird im Stack Speicher für lokale Variablen angelegt und initialisiert
 - bei jedem Konstruktoraufruf wird im Heap Speicher für erzeugte Datenobjekte angelegt und initialisiert

4.4.2 Speicherbereiche

Method area

auch *Codesegment* genannt

- speichert Bytecode benötigter Klassen, inklusive ihrer statischen Variablen
- Lebenszeit = Laufzeit der Klassen

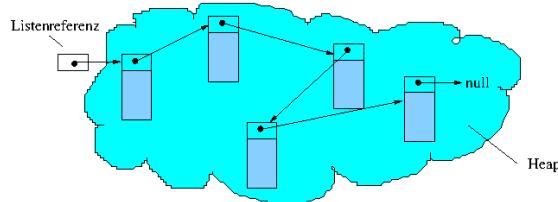
Stack

Laufzeitstapel, siehe Abschnitt 3.4.2

- speichert Aktivierungssätze von Methoden und Konstruktoren, d.h. lokale Variablen und Rücksprungadressen (letztere verweisen ins Method-Area)
- Lebenszeit = Ausführungszeit der Methode/des Konstruktors (Speicher wird freigegeben, wenn nicht mehr benötigt)

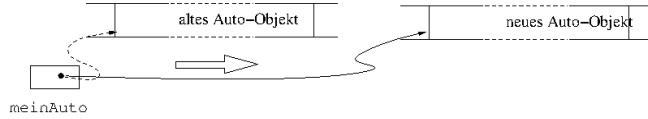
Heap

- speichert per Konstruktor angelegte Datenobjekte
- Lebenszeit: ab Initialisierung mindestens solange, wie Datenobjekt durch Referenzkette von Method-Area oder Stack aus erreichbar ist

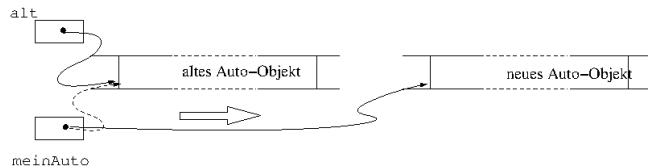


4.4.3 Speicherlecks (verwaiste Datenobjekte)

```
Auto meinAuto = ...;
meinAuto = new Auto(...); // Juju!
```



```
Auto meinAuto = ...;
Auto alt = meinAuto;
meinAuto = new Auto(...); // Juju! (und das alt-Auto kann ich verkaufen)
```



Garbage-Collection (nicht klausurrelevant)

4.4-4

- verwaiste Objekte sind nicht mehr zugreifbar, obwohl sie Speicher belegen
- sie sollten aus dem Speicher „entfernt werden“, d.h. der zugeordnete Speicher sollte wieder freigegeben werden

Java-Besonderheit

- keine *aktiv steuerbare* Möglichkeit zur Speicherfreigabe auf dem Heap!
- stattdessen wird die Erreichbarkeit von Datenobjekten *im Hintergrund* überwacht durch den **Garbage-Collector** (Müllsampler)
- Speicher unerreichbarer Objekte wird automatisch freigegeben, sobald erkannt

4.5 Weiteres zu Strings

4.5.1 Strings und char-Felder

Warum wir eine eigene Klasse für String-Daten brauchen

4.5-1

- Strings sind Folgen von char-Werten \Rightarrow char[] ist prinzipiell ausreichend
- sogar sehr bequeme Ausgabe ist bei char-Feldern möglich:
`char[] ca = {'I','n','f','o'};
System.out.println(ca);`

NICHT bei int- oder anderen Feldern!!

Problem: Anfälligkeit für Alias-Effekte

Clientenklassen können char-Felder (wie auch andere Felder) manipulieren:

```

1 class CharArray {
2     public static final char[] greeting = {'h','e','l','l','o'};
3     public static void toUpper(char[] ca) {
4         for (int i=0; i<ca.length; i++)
5             ca[i] = Character.toUpperCase(ca[i]);
6     }
7 }
8 public class CharArrayClient {
9     public static void main(String[] args) {
10         char[] copy = CharArray.greeting;
11         CharArray.toUpper(copy);
12         System.out.println(CharArray.greeting);
13     }
14 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/CharArrayClient.java>

java CharArrayClient erzeugt Ausgabe HELLO, obwohl greeting in Zeile 2 als final deklariert ist!! Erklärung:
Lediglich die Referenz greeting ist dadurch vor Überschreiben geschützt, nicht der Inhalt!

Strings sind so zentral bei der Programmierung, dass bequeme Formulierungen wie String copy = ...; echte Vorteile bringen. Wir wollen uns dabei nicht ständig Gedanken machen, ob vielleicht etwas Wichtiges überschrieben wird.

4.5-2 Die Klasse String: erste Eigenschaften

String-Objekte sind unveränderlich! (engl. immutable)

- String bietet keine Möglichkeit, den Inhalt eines String-Objekts zu ändern
- Alias-Effekte sind unmöglich

Konstruktoraufrufe+Zuweisungen erzeugen neue String-Objekte

```

String s1, s2;
s1 = "Hallo"; // ist äquivalent zu: s1 = new String("Hallo");
s2 = new String("Hallo"); // ist äquivalent zu: s2 = "Hallo";

```

Auch Verkettung bewirkt Konstruktoraufrufe!

DualCode.java (Paragraph 2.2-18) ist in diesem
Sinne ziemlich verschwenderisch

```
s2 += ", Welt!"; // erzeugt neues String-Objekt (altes Objekt verwäist)
```

4.5-3 Besonderheiten (nicht klausurrelevant)

Speicherung von String-Literalen

- JVM hält String-Literale in einem “String-Pool” (ein Teilbereich im Heap)
- bei Zuweisungen wird nur dann neues String-Objekt angelegt, wenn kein Inhaltsgleiches im Pool gefunden wird. Diese Technik wird effizient ermöglicht durch Hashcodes und Garbage-Collection.

StringBuilder

Diese Klassen realisieren eine Art *veränderlicher* Strings. Wegen möglicher Längenänderungen, müssen ggf. Umspeicherungen angestoßen werden (ähnliche Techniken betrachten wir im Abschnitt 5.5). Die Anfangskapazität sieht eine Speicherreserve vor, damit nicht zu oft umgespeichert werden muss. Ausschnitt aus der [StringBuilder-Dokumentation](#):

<u>public class StringBuilder</u>		<small>vollständige Dokumentation siehe Java-API</small>
`StringBuilder (String s)	initialisiert neues Objekt mit Wert s und Kapazität größer als Länge von s	
`append(String s)	ändert Wert durch Anhängen von s an <u>dieses</u> Objekt	
`setCharAt(int i, char c)	setzt i-tes Zeichen dieses Objekts	

4.5.2 Einige Methoden zur Arbeit mit Strings[String-Details ermitteln](#)

4.5.4

Länge beschaffen mit `length()`

```
String eingabe = StdIn.readLine();
System.out.println("Eingabe hat " + eingabe.length() + " Zeichen");
```

Einzelzeichen beschaffen mit `charAt(..)`

```
System.out.println("Anfangszeichen: " + eingabe.charAt(0));
```

Anfangs-/Endetest mit `startsWith(..)`, `endsWith(..)`

```
if ( eingabe.startsWith("-"))
    System.out.println("Keine negativen Werte erlaubt!");
```

Merk

Diese Methoden sind keine statischen, sondern **Instanzmethoden**: Sie beziehen sich auf die String-Instanz, mit der sie aufgerufen werden.

[Vergleich von Strings](#)

4.5.5

Gleichheitstest mit `equals(..)`

```
do { String eingabe = StdIn.readLine(); // speichere komplette Eingabezeile in 'eingabe'
} while ( ! eingabe.equals(password));
```

Test `if (eingabe == password) ...` vergleicht stattdessen nur Referenzen!

Beispiel: Ein Orakel

```

class Orakel {
    public static void main(String[] argv) {
        String eingabe;
        do {
            System.out.println("Nenne eine Frage: "); // Prompt
            eingabe = StdIn.readLine();
            if (eingabe.endsWith("?"))
                System.out.println("Weiss ich nicht.");
            else
                System.out.println("Das ist keine Frage!");
        } while (true); // erzwingt Endlos-Schleife
    }
}

```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Orakel.java>

4.5-6 Lexikographischer Vergleich mit compareTo(...)

```

1   String s1 = StdIn.readLine(), s2 = StdIn.readLine();
2   if (s2.compareTo(s1) < 0)
3       System.out.println(s2 + " kommt vor " + s1);
4   else if (s2.compareTo(s1) == 0)
5       System.out.println(s2 + " ist gleich " + s1);
6   else
7       System.out.println(s2 + " kommt nach " + s1);

```

Beispiel: Sortierfilter für Strings

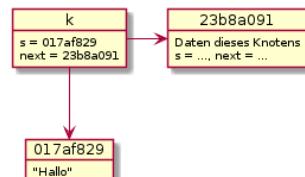
siehe Paragraph 4.2-22

- verwende ZahlKnoten und Sort als “Codesteinbrüche” für StringKnoten und StringSort
- ersetze if (x>y) ... durch if (x.compareTo(y) > 0) ...

```

class StringKnoten {
    String s;
    StringKnoten next;
    public StringKnoten(String s, StringKnoten next){
        this.s = s;
        this.next = next;
    }
}

```



insert(..)-Methode

```

public static void insert( String s ) {
    StringKnoten neu = new StringKnoten( s, null );
    StringKnoten a = null, b = anfang;
    while (b != null && neu.s.compareTo(b.s) > 0) {
        a = b; b = b.next;
    }
    if (a == null) anfang = neu; else a.next = neu;
    neu.next = b;
}

```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/StringSort0/StringSort.java>

cat Privat Dienstlich | java StringSort leistet das Gleiche wie cat Privat Dienstlich | sort (siehe Paragraph 1.1-22)

Kapitel 5

Objektorientierte Programmierung

5.1 Klassen und Objekte

5.1.1 Klassen im Wandel der Paradigmen

Monolithische imperative Programme

5.1-1

HelloWorld-Struktur

Klassen nur als “Verpackung” der main-Methode

- main-Methode ist das einzige Klassenelement
- kann Aufrufe von Java-API-Methoden enthalten

= Sichtweise bis inklusive Kapitel 2

Prozedurale Programmierung

Klassen als “Behälter” für mehrere Methoden

- statische Methoden sind die einzigen Klassenelemente
- enthalten Aufrufe von Methoden dieser Klasse oder von Bibliotheksklassen
- Informationsaustausch über die Methodenschnittstellen

= Sichtweise ab Kapitel 3

Imperative Programme verteilt auf mehrere Klassen

5.1-2

Modulare Programme

Applikation besteht typischerweise aus einer Clientenklasse und mehreren Bibliotheken (Klassen, die Methoden etc. thematisch sammeln)

= Sichtweise ab Abschnitt 3.5

Statische Variablen (Modifizierer static)
sind global in der Klasse sichtbar, Beispiel:

- statische Referenz auf Listenanfang in Zahlspeicher.java

```
private static ZahlKnoten anfang = null;
```

[5.1-3 Nützliche Terminologie](#)

Klassenelemente

= syntaktische Einheiten auf oberster Ebene innerhalb einer Klassendefinition,
Beispiele:

- Methodendeklaration
- Konstruktordeklaration
- Deklaration einer Instanzvariablen

Klassenelemente sind gültig in der gesamten Klasse (sie haben „Klassenscope“, siehe Paragraph 3.4-14)

Klassenmethoden und Klassenvariablen

= Synonyme für statische Methoden und Variablen einer Klasse

- innerhalb der definierenden Klasse uneingeschränkt zugreifbar mit dem Namen:
 $\langle \text{methode} \rangle (\langle \text{Parameter} \rangle)$ bzw. $\langle \text{Variable} \rangle$
- Zugriff auf sichtbare statische Elemente einer anderen Klasse
 $\langle \text{Klasse} \rangle . \langle \text{methode} \rangle (\langle \text{Parameter} \rangle)$ bzw. $\langle \text{Klasse} \rangle . \langle \text{Variable} \rangle$

[5.1-4 Klassen als Baupläne für Datenobjekte](#)

= Sichtweise in Kapitel 4

Beispiele

Student, Datum, Person, Punkt, Gerade, Zahlnoten, StringKnoten

Klassenelemente

außer Klassenvariablen und -methoden wie bisher, sind weitere *nicht-statische* Elemente möglich, z.B. `toString()`-Methoden

Zugriff auf Instanzvariablen und `toString()`

nur mit Hilfe einer Referenz auf ein erzeugtes Datenobjekt:
 $\langle \text{Referenz} \rangle . \langle \text{Variable} \rangle$ bzw. $\langle \text{Referenz} \rangle . \text{toString}()$

[5.1-5 Jetzt: Klassen als Datentyp-Beschreibungen](#)

Objekte = Datenobjekte, die neben einem *Zustand* (Wert) auch ein *Verhalten* haben können (beschrieben durch nicht-statische Methoden = **Instanzmethoden**)

Beispiele

- Punkt-Instanz (“verwirklichter Punkt-Bauplan”)
 - Zustand: Koordinaten und Name (Instanzvariablen `x`, `y`, `name`)
 - Verhalten: Erzeugung einer textuellen Beschreibung (Instanzmethode `toString()`)
- String-Instanzen

- Zustand: Folge von char-Werten
- Verhalten: Längenermittlung (Instanzmethode `length()`), Einzelzeichenzugriff (`charAt(int)`), Vergleich (`equals(String)`, ...)

Erinnerung: *Datentyp* = Menge von Werten mit Menge anwendbarer Operationen (siehe Paragraph 2.1-5)

Klassen, die Konstruktoren enthalten, beschreiben Datentypen!

- Werte = Menge der Instanzen, die erzeugt werden können
- auf Instanzen anwendbare Operationen = Aufrufe von Instanzmethoden

5.2 Instanzmethoden

5.2.1 Erste Beispiele

Terminologie

5.2-1

Beispiel: Klasse Orakel

siehe Paragraph 4.5-5

Beim Aufruf `eingabe.equals("Was ist ...")` ist `eingabe` die Referenz auf das String-Objekt, mit dem `equals(..)` aufgerufen wird. Dieses Objekt wird das **aktuelle Objekt** genannt.

Dokumentationen im Fachjargon

`boolean equals(String t)` liefert `true`, falls dieser String und `t` den gleichen Wert haben.

wobei mit "Wert eines Strings" die entsprechende Folge von char-Werten gemeint ist, also der Inhalt des Strings

`String toString()` liefert textuelle Beschreibung dieses Objekts

Nachtrag: Aus der API für String

5.2-2

public class String	vollständige Dokumentation siehe Java-API
<code>String(String s)</code>	konstruiert neuen String gleichen Werts wie <code>s</code>
<code>int length()</code>	liefert Stringlänge <u>dieses</u> Strings
<code>charAt(int i)</code>	liefert <code>i</code> -tes Zeichen <u>dieses</u> Strings
<code>indexOf(char c)/lastIndexOf(char c)</code>	Index des ersten/letzten Vorkommens von <code>c</code> in <u>diesem</u> String
<code>char[] toCharArray()</code>	konvertiert <u>diesen</u> String in neues char-Feld
<code>String String(char[] c)</code>	umgekehrt: konvertiert char-Feld in neuen String
<code>boolean equals(String t)</code>	liefert <code>true</code> , falls <u>dieser</u> String und <code>t</code> gleichen Wert haben
<code>int compareTo(String t)</code>	liefert positiven/negativen Wert, falls <u>dieser</u> String größer/kleiner ist ^a als <code>t</code> ist (sonst 0)
<code>String replaceAll(char a, char b)</code>	liefert neuen String dessen Wert aus <u>diesem</u> durch Ersetzten aller <code>a</code> durch <code>b</code> entsteht
<code>String concat(String t)</code>	liefert neuen String, durch Anhängen von <code>t</code> an <u>diesen</u> String

^aim lexikographischen Sinne

Instanzmethoden selbst definieren

5.2-3

Merke

- Instanzmethoden werden *ohne static* definiert - ansonsten genauso wie unsere bisherigen Methoden
- Instanzmethoden können *nur* mit Hilfe einer Referenz aufgerufen werden - ansonsten genauso wie unsere bisherigen Methoden

Beispiel: falls innerhalb der Instanzmethode `a` eine Instanzmethode `b` aufgerufen wird, muss der Aufruf `this.b(<Parameter>)` lauten.

5.2-4 Beispiel: Klasse Person

```

1 import java.util.GregorianCalendar;
2 public class Person {
3     String vorname; String nachname;
4     private Datum geb_datum;
5     public Person (String vorname, String nachname,
6                     int tag, int monat, int jahr) { // Geburtsdatum
7         this.vorname = vorname; this.nachname = nachname;
8         geb_datum = new Datum(tag,monat,jahr);
9     }
10    public int jahrgang () { // Methode
11        return geb_datum.jahr; // liefert Geburtsjahrgang
12    }
13    public int alter(){ // Methode
14        int jetzt = new GregorianCalendar().get(GregorianCalendar.YEAR);
15        return jetzt - geb_datum.jahr; // liefert das Lebensalter
16    }
17 }
```

Zeile 14: Der parameterlose Konstruktorauftrag `GregorianCalendar()` erzeugt ein Objekt, das die aktuellen lokalen Zeitangaben speichert.

5.2-5 Beispiel: Klasse Stopwatch

Stopwatch-API

<code>public class Stopwatch</code>	
<code>Stopwatch()</code>	erzeugt neue Stoppuhr
<code>double elapsedTime()</code>	liefert verstrichene Zeit (in s) seit Erzeugung dieser Stoppuhr

Implementierung

```

1 public class Stopwatch {
2     private final long start;
3     public Stopwatch() { start = System.currentTimeMillis(); }
4     public double elapsedTime() {
5         long now = System.currentTimeMillis();
6         return (now - start) / 1000.0;
7     }
8     public static void main(String[] args) {
9         int N = Integer.parseInt(args[0]);
10        // Test-Client ...
11    }
12 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Stopwatch.java>

Eine Beispielanwendung ist im Testclient realisiert: Der Wert $\sum_{i=1}^N \sqrt{i}$ wird einmal mit `Math.sqrt()`-Aufrufen und einmal mit `Newton.sqrt()`-Aufrufen berechnet und sowohl der Quotient der berechneten Werte als auch der der jeweiligen Zeidauern ausgegeben.

Der Wertequotient ist = 1 für jedes $N > 1$, so wie es sein sollte. Der Laufzeitquotient ist (prozessorabhängig) erst bei Größenordnungen ab ca. $N = 2000$ aussagekräftig und strebt bei wachsendem N gegen 1 d.h. die Eigenimplementation ist praktisch ebenso effizient wie die der Standardbibliothek.

Stopwatch-Objekte sind offenbar unveränderlich.

Beispiel: Klasse Histogram

5.2-6

API

<code>public class Histogram</code>	
<code> public Histogram(int N)</code>	erzeugt <i>dynamisches</i> Histogramm für N int-Werte in $[0, N]$
<code> public void addDataPoint(int i)</code>	erhöht in diesem Histogramm die Häufigkeit des Werts i
<code> public void draw()</code>	plottet Balkengrafik dieses Histogramms

Code

```

1  public class Histogram {
2      private final double[] freq; // freq[i] = # Vorkommen des Werts i
3      public Histogram(int n) { freq = new double[n]; }
4      public void addDataPoint(int i) {
5          freq[i]++;
6      }
7      public void draw() {
8          StdStats.plotBars(freq);
9      }
10     public static void main(String[] args) {
11         final int N = Integer.parseInt(args[0]); // Anzahl der Klassen
12         Histogram histogram = new Histogram(N+1);
13         while (!StdIn.isEmpty()) {
14             double m = StdIn.readDouble(); // Messwert einlesen
15             histogram.addDataPoint((int) m);
16             StdDraw.clear();
17             histogram.draw();
18             // StdDraw.setCanvasSize(500, 100);
}

```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Histogram.java>

Beim Zählen der Vorkommen wieder die übliche Technik (siehe Paragraphen 3.4-19 und 3.5-8): Datenpunkte als Feldindizes verwenden.

Histogram-Objekte sind veränderlich (und zwar nur durch addDataPoint).

Beispieldaufruf

java Histogram 90 < data/Klausurpunkte.txt

5.2-7

5.2.2 Kapselung

Begriff und Motivation

Kapselung (Geheimnisprinzip)

- Verbergen von Daten vor direktem Zugriff aus anderen Klassen
- nur indirekten Zugriff ermöglichen über angebotene Methoden

“Man muss nicht wissen, wie ein Datentyp implementiert ist, um ihn zu nutzen.”

Motivation

- *modulare Programmierung*: Code-Verbesserungen wirken sich nicht auf die Funktionsweise von Clients aus, auch die API-Dokumentation bleibt erhalten

- *klarer Programmcode*: Information fließt über möglichst wenige Schnittstellen, die kurz und prägnant zu programmieren sind
- *leichteres Debuggen*: Auswirkungen von Fehlern werden begrenzt, so dass sie gut zu lokalisieren sind

[5.2-8 Beispiel: IPv4 vs. IPv6](#)

Internet-Protokoll (IP)

= Standard für Datenaustausch zwischen Geräten über das Internet

- jedem Gerät ist eine eindeutige *IP-Adresse* zugewiesen (i.W. eine Zahl), um Quelle und Ziel von Daten anzugeben
- IPv4: 32-Bit-Adressen (z.B. 192.0.2.41 = 4 Zahlen zwischen 0 und 255), ca. 4 Milliarden Möglichkeiten
- IPv6: 128-Bit-Adressen, über 10^{38} Adressen!!

“*Internet of Things*”

- eigene IP-Adressen für Radio, Kühlschrank, Personenwaage ... IPv4 reicht nicht
- immense Kosten für nötige Anpassung von nicht-gekapseltem Code an IPv6

[5.2-9 Beispiel: Zählen von Wählerstimmen](#)

Negative Stimmenzahl bei US-Präsidentenwahl 2000

Beispiel aus [Sedgewick/Wayne]

```
Counter c = new Counter("Volusia", VOTERS_IN_VOLUSIA_COUNTY);
c.count = -16022;
```

Mit Kapselung wäre das nicht passiert

```
public class Counter {
    private final String name;
    private final int maxCount;
    private int count;
    public Counter(String id, int max) {
        name = id; maxCount = max; }
    public void increment() { // eine "set-Methode" ...
        if (count < maxCount) // .. fuer ...
            count++; } // .. count.
    public int value() { // eine "get-Methode" ...
        return count; } // .. fuer count.
    public String toString() {
        return name + ": " + count; }
}
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Counter.java>

[5.2-10 set- und get-Methoden \(“getter und setter”\)](#)

Merke

- direkter Zugriff auf private-Attribute einer Klasse nur durch Code dieser Klasse möglich

```
Counter c = new Counter("Volusia", VOTERS_IN_VOLUSIA_COUNTY); // Neu-Implementierung
c.count = -16022; // error: count has private access in Counter
```

Kapselung

biete Clients Zugriffsmethoden an (soweit benötigt)

- set-Methode (Beispiel: public void increment())
- get-Methode (Beispiel: public int value())

Typische Schnittstellen und beliebte Namenskonventionen

für int-Variable x z.B.:

- public void setX(int v) (oder set(int v), oder x(int v))
- public int getX() (oder get(), oder x())

5.2.3 Verbesserung des Geometrie-Beispiels

Ziele für Version 1

5.2-11

Kapselung

- private-Instanzvariablen: Clients sollen nur public-Methoden nutzen, nicht die innere Struktur der Objekte/der Implementierung
- weitere Entwurfsverbesserungen gegenüber Kapitel 4

Schnittpunkt zweier Geraden

ist Ergebnis der Verknüpfung dieser Geraden mit einer anderen ⇒ Instanzmethode
public Punkt schnittpunkt(Gerade h)

siehe Paragraph 3.2-9

Verbindungsgerade zweier Punkte

Analog: ist Ergebnis der Verknüpfung dieses Punktes mit einem anderen ⇒
Instanzmethode public Gerade verbindungsGerade(Punkt q)

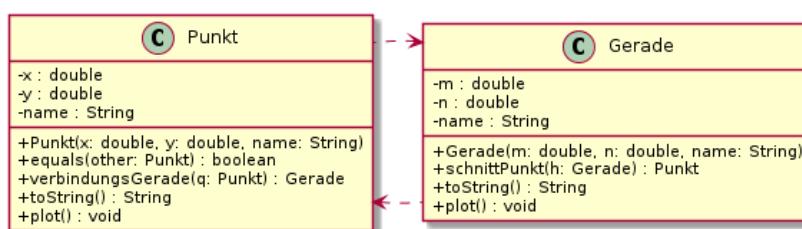
Visualisierung

Instanzmethode plot() für Punkt- bzw. Geraden-Objekte

Überarbeitetes Klassendiagramm

5.2-12

vgl. Paragraphen 3.2-9 und 4.2-10



Wir betrachten hier nur ein paar Implementierungsdetails von `Punkt.java`. Details zu `Gerade.java` bitte im Quelltext ansehen.

5.2-13 Instanzmethoden für die Klasse Punkt

public verbindungsGerade(Punkt q)
liefert “Gerade durch diesen Punkt und q”¹

```

1  public Gerade verbindungsGerade(Punkt q) {
2      if ( this.equals(q) )
3          throw new RuntimeException("Gerade nicht definiert!");
4
5      if (this.x == q.x)
6          return new Gerade( Double.POSITIVE_INFINITY, q.x);
7      double m = (this.y - q.y) / (this.x - q.x), n = this.y - m * this.x;
8      return new Gerade (m,n);
9 }
```

<http://www.stud.informatik.uni-goettingen.de/inf101/java/Geometrie1/Punkt.java>

Aufruf: `Gerade g = p.verbindungsGerade(q);`, Zeile 2: Methode `equals(..)` ist unten gezeigt

public equals(Punkt other)

zwei “reale” Punkte sind *gleich*, wenn ihre Koordinaten übereinstimmen

```

public boolean equals(Punkt other) {
    return (this.x == other.x && this.y == other.y);
}
```

<http://www.stud.informatik.uni-goettingen.de/inf101/java/Geometrie1/Punkt.java>

5.2-14 Punkt-Objekte visualisieren

Vorgaben

- roter Punkt in richtiger Position (x,y) auf der Leinwand
- Name in schwarzer Schrift an geeigneter Position $(x+dx, y+dy)$

Ansatz zur Implementation

- verwende `StdDraw.setPenColor(..)`, `StdDraw.Point(..)`, `StdDraw.text(..)` geeignet
- nutze Klassenvariablen `dx`, `dy` zur Positionierung der Beschriftung

¹ abgesehen von Sonderfällen (wie z.B. Gerade durch (∞, ∞) und (x, y)), die nicht alle implementiert sind

```

1  public class Punkt {
2      private static double dx = 0.025, dy = 0.025; // rel. Beschriftungsposition
3      // ...
4      public void plot() {
5          if (x == Double.POSITIVE_INFINITY ||
6              y == Double.POSITIVE_INFINITY) return;
7          StdDraw.setPenColor(StdDraw.RED);
8          StdDraw.point(x,y);
9          StdDraw.setPenColor(StdDraw.BLACK);
10         StdDraw.text(x + dx,y + dy,name); // Name rechts ueber dem Punkt
11     }
12     // ...
13 }
```

– Zeile 10: `StdDraw.text(..)` setzt String an angegebene Position
 – die Test-Unit benutzt `StdArrayIO.readDouble2D()` zum Einlesen von N Punktkoordinaten aus einer $N \times 2$ -Matrix, erwartet daher von der Standardeingabe ein Datenformat wie hier für 3 Punkte: echo 3
`2 0.0 0.0 0.5 0.5 1.0 1.0 | java Punkt`

[Skalierung der Leinwand](#)

5.2-15

Problem

x - und y -Bereiche der Leinwand müssen angepasst werden, damit auch Punkte außerhalb des Standardquadrats $[0, 1] \times [0, 1]$ sichtbar sind!

Idee

- inventarisiere erzeigte Punkte und aktualisiere dabei minimale und maximale Koordinatenwerte (“Zeichenhorizont”)
- skaliere Leinwand entsprechend vor `plot()`-Aufrufen

Welche Art Daten sind erforderlich?

- Zeichenhorizont kann nicht aus einzelnen `Punkt`-Objekten ermittelt werden, sondern aus der Gesamtheit der erzeugten Objekte
- also: verwende *Klassenvariablen* für den Horizont und *Klassenmethode* für Anpassungen

5.2-16 public plot() für die Klasse Punkt

```

1  public class Punkt {
2      private static final int MAXANZAHL = 50;
3      private static Punkt[] inventar = new Punkt[MAXANZAHL];
4      private static int anzahl = 0;
5      private static double hMin = 0, hMax = 1; // Zeichenhorizont
6      private static double dx = 0.025, dy = 0.025; // rel. Beschriftungsposition
7      private static void setScales() {
8          for (int i = 0; i < anzahl; i++) {
9              hMin = Math.min(hMin, Math.min(inventar[i].x, inventar[i].y));
10             hMax = Math.max(hMax, Math.max(inventar[i].x, inventar[i].y));
11         }
12         StdDraw.setScale(hMin, hMax); // setzt x- und y-Bereich wie angegeben
13         dx = dy = 0.025 * (hMax - hMin);
14     }
15     public Punkt(double x, double y, String name) {
16         if (anzahl == MAXANZAHL)
17             throw new RuntimeException("Maximalzahl an Punkten ueberschritten");
18         this.x = x; this.y = y; this.name = name;
19         inventar[anzahl] = this;
20         anzahl++;
21     }
22     // ...
23 }

```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Geometrie/Punkt.java>

- Zeilen 2, 3: der Einfachheit halber wird anstelle einer verketteten Liste ein Feld zur Inventarisierung verwendet (vgl. Paragraph 4.2-21)
- die Design-Entscheidung “Feld anstelle dynamischer Liste” erzwingt, die Anzahl der Punkt-Objekte zu überwachen: Zeile 4 und entsprechende Teile des Konstruktors
- Zeilen 9, 10: in setScales() werden die äußersten Koordinaten bestimmt und das Ergebnis für die Skalierung beider Achsen verwendet (Zeile 12)
- Zeile 13: die relative Beschriftungsposition muss mitskaliert werden
- Der Testclient interpretiert das Kommandozeilenargument als Ganzzahl und liest entsprechend viele Koordinatenpaare ein: echo 0.0 0.0 1.0 1.0 2.0 2.0 | java Punkt 3.

5.2-17

Gerade-Objekte visualisieren

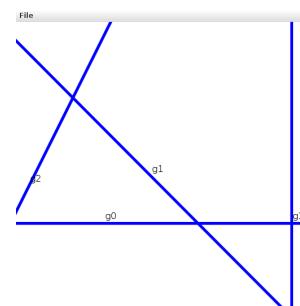
Prinzipiell gleicher Ansatz (Inventarisierung, Horizontberechnung), aber ...

Problem

Geraden sind unendlich lang, welcher Teil soll auf der Leinwand erscheinen?

Idee

- lege besonderen Punkt der Geraden als “Sichtpunkt” fest, stütze Horizont und Beschriftungsposition darauf
 - Beispiel: Fußpunkt des Lots von $(0,0)$ auf Gerade
 - Sichtpunkt ist Eigenschaft der Gerade (Instanzvariable!), muss beim Konstruktorauftrag berechnet werden
- Zeichne ansonsten mittels StdDraw.line(...) bis an Leinwandrand



Instanzvariablen und Realisierung des Konstruktors

5.2-18

```

1 public class Gerade {
2     private final double m, n; // Parameter der ..
3     private final String name; // .. Geraden
4     private final double vx, vy; // Koordinaten eines Sichtpunkts
5     // Hilfsmethode:
6     private static double[] fußpunkt(double m, double n) {
7         double[] f = new double[2];
8         if (n == Double.POSITIVE_INFINITY)
9             { f[0] = n; f[1] = 0; }
10        else { // z.B. aus Formelsammlung entnehmen:
11            f[0] = - (m * n) / (m * m + 1);
12            f[1] = m * f[0] + n;
13        }
14        return f;
15    }
16    public Gerade(double m, double n, String name) {
17        if (anzahl == MAXANZAHL)
18            throw new RuntimeException("Maximalzahl an Geraden überschritten");
19        this.m = m; this.n = n; this.name = name;
20        double[] f = fußpunkt(m,n);
21        this.vx = f[0]; this.vy = f[1];
22        inventar[anzahl] = this;
23        anzahl++;
24    }
25    // ...
26}

```

- Zeile 3: Fußpunkt wird nicht `Punkt`-Objekt realisiert, sondern nur Koordinaten beim Objekt gespeichert
- Zeilen 5 und 20: die Hilfsmethode liefert ein Feld der Länge 2, das die beiden Fußpunktkoordinaten ermittelt, die Komponenten werden im Konstruktor ausgelesen und den Instanzvariablen `fx`, zugewiesen
- Die Test-Unit ist ähnlich wie bei `Punkt` implementiert, nur dass die Skalierung *immer* wirksam wird echo 4 2
0.0 0.0 1.0 1.0 2.0 2.0 Infinity 2 | java Gerade.

public `plot()` für die Klasse `Gerade`

5.2-19

```

public class Gerade {
    public void plot() {
        setScales(); // skaliere Leinwand
        StdDraw.setPenRadius(0.01);
        StdDraw.setPenColor(StdDraw.BLUE);
        double span = hMax-hMin; // StdDraw-Doku: Leinwand ragt 10% über ..
        double min = hMin - 0.1 * span, max = hMax + 0.1 * span; // .. x/y-Bereiche
        if (this.m == Double.POSITIVE_INFINITY) // vertikale Gerade:
            StdDraw.line(this.n, min, // Koordinaten, in denen ..
                         this.n, max); // .. Leinwand verlassen wird
        else
            StdDraw.line( min, this.m*min+this.n, // analog bei nicht-
                         max, this.m*max+this.n); // vertikalen Geraden
        StdDraw.setPenColor(StdDraw.BLACK);
        StdDraw.text(vx+dy, vy+dy, name);
    }
    // ...
}

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie1/Gerade.java>

Die Test-Unit ist ähnlich wie bei `Punkt` implementiert, nur dass die Skalierung *immer* wirksam wird echo 4 2 0.0
0.0 -1.0 1.0 2.0 2.0 Infinity 2 | java Gerade.

5.3 Vererbung

5.3.1 Ein Spielzeugbeispiel

5.3-1 Idee

Klassenerweiterung

- ergänze eine bereits definierte Klasse (**Basisklasse**) durch zusätzliche Attribute und Methoden zu einer neuen Klasse (**Erweiterungsklasse**)
- Objekte der Erweiterungsklasse haben
 - Eigenschaften/Verhalten der Objekte der Basisklasse (“direkt geerbte Merkmale”)
 - zusätzliche/s Eigenschaften/Verhalten (“spezielle Merkmale”)

Beispiel: Studenten sind auch Personen

- jede Person
 - hat: Vorname, Nachname, Geburtsdatum
 - kann: das Geburtsjahr mitteilen
- jeder Student
 - ist: eine Person (hat/kann alles, was Personen haben/können)
 - hat: Matrikelnummer, Studienfach, Jahr des Studienbeginns
 - kann: Jahr des Studienbeginns mitteilen
- Klassen-Hierarchien: jeder Bachelorstudent ist ein Student, ...

5.3-2 Implementierung

Wir erweitern `Person` aus Paragraph 4.2-7.

```

1  public class Student extends Person { // Student erbt direkt von Person
2      static int next_mat_nr = 100000;
3      int mat_nr; // Matrikelnr
4      String fach; // Studienfach
5      int jsb; // Jahr des Studienbeginns
6      public Student(String vn, String nn, // Name, Vorname
7                      int t, int m, int j, // Geburtstag (Tag, Monat, Jahr)
8                      String f, int jsb) { // Studienfach+Jahr d. Studienbeginns
9          super(vn, nn, t, m, j); // Konstruktor der Basisklasse
10         fach = f; this.jsb = jsb; // initialisiere nun noch ..
11         mat_nr = next_mat_nr++; // .. die anderen Instanzattribute.
12     }
13     public int jahrgang() { // studiert seit welchem Jahr
14         return jsb; } // Jahr des Studienbeginns
15     public int geburtsjahrgang() { // Zugriff auf jahrgang()...
16         return super.jahrgang(); } // .. Implementation von Person
17 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Student.java>

Zeile 9: expliziter Aufruf von Konstruktor der Basisklasse *nur* mittels `super(..)` (nicht etwa mit `Person(..)`) und *nur* als erste Anweisung im Konstruktor einer Erweiterungsklasse. Zeile 16: Aufruf einer Methode der Basisklasse.

5.3-3 Überschreiben von Methoden

Beobachtung

- Person bietet eine Instanzmethode der Form public int jahrgang()
- Erweiterungsklasse Student bietet auch diese Schnittstelle, d.h. die Methode aus Person wird in Student überschrieben

Bedingungen für das Überschreiben

- Name und Parametertypen sind gleich, Rückgabetypen auch
- Zugreifbarkeit der Methode wird in Erweiterungsklasse nicht eingeschränkt (z.B. verhindert der Compiler das Überschreiben einer public-Methode durch eine private-Methode)
- Zugriffsart: Instanzmethode (der Compiler verbietet das Überschreiben von Klassenmethoden)

Dynamisches Binden (auch *spätes Binden* genannt)

5.3-4

```

1 Person p;
2 p = new Person("Lisa", ..., 1990);
3 Student s = new Student("Fritze", ..., 1993, "BWL", 2010);
4 p = s; // möglich, da Student-Objekte AUCH Person-Objekte sind
5 System.out.println(p.jahrgang()); // Geburtsjahr oder Jahr des Studienbeginns?

```

Statischer und dynamischer Typ einer Referenzvariablen

- **statischer Typ:** der bei Deklaration verwendete Typ
Beispiel: p hat laut Zeile 1 den statischen Typ Person
- **dynamischer Typ:** der Typ des aktuell referenzierten Objekts
Beispiel: p hat am Ende statischen Typ Person und dynamischen Student auch *aktueller Typ* genannt

Binden = die Zuordnung des bei Aufruf auszuführenden Codes

siehe auch Paragraph 3.2-5

- Klassenmethoden und Konstruktoren werden *statisch gebunden*: Code wird zur Compilezeit fest zugeordnet
- Instanzmethoden werden *dynamisch gebunden*: Code wird zur Laufzeit zugeordnet, entsprechend dem dynamischen Typ

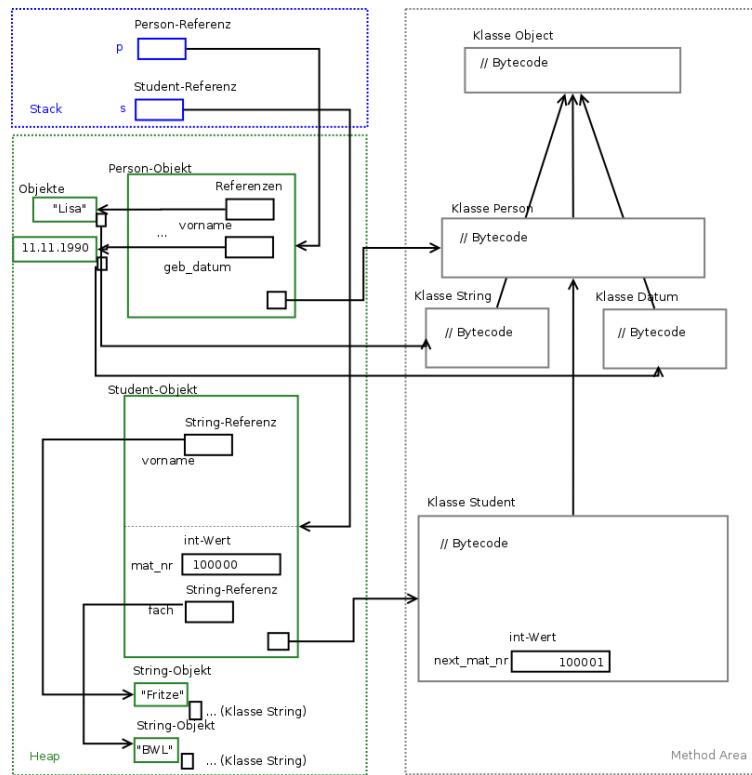
5.3-5 Einige Codebeispiele

Codefragmente (main-Methode)	Ergebnis/Speicherbild bzw. Fehlerart
1. Person p=new Person("Lisa",...); Student s=new Student("Fritze",...);	
2. ... p.jahrgang() ..	1990
3. ... p.fach ..	Compilezeitfehler
4. ... ((Student) p).fach ..	Laufzeitfehler
5. s = p;	Compilezeitfehler
6. p = s;	
7. p = s; .. p.fach ..	Compilezeitfehler
8. p = s; .. ((Student) p).fach ..	"BWL"
9. p = s; .. p.jahrgang() ..	2010

Erklärung

3. der statische Typ (Person) erlaubt kein Attribut fach für die Referenz p
4. aktuelles Objekt ist kein Student-Objekt, cast darauf nicht möglich
5. Compiler verbietet Zuweisung von Referenz der Basisklassen an eine der Erweiterungsklassen
7. nach der Zuweisung ist Student der dynamische Typ von p
8. der statische Typ Person ermöglicht keinen direkten Zugriff auf fach
9. durch cast-Operator (Student) wird Objekt als Student-Objekt angesprochen und auf Attribut fach zugegriffen
9. es wird die jahrgang()-Methode entsprechend des aktuellen dynamischen Typs von p (also: Student) gerufen.

5.3-6 Spechersituation nach Tabellenzeile 1

**Erklärung**

5.3-7

(siehe auch Paragraph 4.4-2)

- Bytecode benötigter Klassen wird bereits beim Laden der Applikation im Method Area abgelegt, auch Klassenattribute (z.B. `next_mat_nr`) werden dort angelegt
- `p` und `s` sind lokale Variablen der `main`-Methode, werden daher bei Aufruf vom `main` (ausgelöst von der JVM) im Laufzeitstapel angelegt
- durch Konstruktorausfülle werden im Heap `Person`- und `Student`-Objekte angelegt (sowie Objekte für deren `String`- oder `Datum`-Bestandteile)
- wird eine Instanzmethode aufgerufen, so kommt der entsprechende Bytecode zur Ausführung (deswegen benötigt jedes Objekt eine Referenz auf den Bytecode seiner Klasse), wobei solange in der Vererbungshierarchie aufwärts gesucht wird, bis eine Implementation gefunden wird. Automatisch wird so immer die speziellste Implementierung gewählt (da der Code kompiliert wurde, ist ja bereits gesichert, dass es irgendwo in der Hierarchie eine Implementierung gibt).
- Die Klasse `java.lang.Object` steht an der Spitze jeder Vererbungshierarchie von Java-Klassen: Jede Klasse, die nicht explizit von einer anderen erbt (angezeigt durch `extends`) ist direkte Erweiterungsklasse von `Object`.

[5.3-8 Überschreiben der `toString\(\)`-Methode](#)

Erinnerung

- Klassenmethoden können nicht überschrieben werden
- Compiler verhindert auch das Überladen von Klassen- durch Instanzmethoden

toString() -Methode der Klasse Object

- steht in *jeder* Klasse zur Verfügung
- liefert den Hashcode dieses Objekts
- ist eine `public`-Methode

Folgerung

- siehe Paragraph 4.2-12
- wenn in irgendeiner Klasse eine Instanzmethode `toString()` definiert wird, so überschreibt sie sie und **muss** daher diese Schnittstelle haben:

```
public String toString()
```

Genau eine solche Methode ist gemeint, wenn von “*der* `toString()`-Methode” einer Klasse die Rede ist.

[5.3-9 Konstruktoren und Vererbung \(nicht klausurrelevant\)](#)

Noch ein paar Erklärungen zur Definition und Funktionsweise von Konstruktoren. Wir betrachten die Klassenerweiterung `E extends B`. Offenbar muss der Aufruf eines Konstruktors von `E` initial auch einen von `B` bewirken, um die in `B` definierten Instanzvariablen zu initialisieren. Der `B`-Konstruktorauftrag kann z.B. *explizit* im `E`-Konstruktor stehen, dabei ist zu beachten:

Merk

Im Konstruktor-Code sind explizite Konstruktoraufrufe nur als erste Anweisung erlaubt und nur in dieser Syntax:

- `this(...)` für Konstruktoren der aktuellen Klasse
- `super(...)` für Konstruktoren der Erweiterungsklasse

Ohne *expliziten* initialen Konstruktorauftrag am Anfang eines `E`-Konstruktors erfolgt ein *impliziter*: der Compiler ergänzt *automatisch* `super();` als erste Anweisung. Das erfordert aber den parameterlosen Konstruktor `B()`! Dies wird erreicht, indem `B` entweder

- `B()` bereitstellt oder
- *gar keinen* Konstruktor implementiert! ⇒ Compiler ergänzt implizit den sogenannten *Default-Konstruktor*.

Der Default-Konstruktor ist parameterlos. Sein Aufruf initialisiert die Instanzvariablen mit Default-Werten:

Typ	Default-Wert
Ganzzahl bzw. Gleitkomma	0 bzw. 0.0
boolean	false
Referenzdatentyp	null

Ist dies in einer Anwendungssituation nicht sinnvoll, muss ein eigener parameterloser Konstruktor bereitgestellt werden:

```
public class Datum {
    ...
    int tag; int monat; int jahr;
    public Datum (int tag, int monat, int jahr){ // Konstruktor mit 3 Parametern
        this.tag = tag; this.monat = monat; this.jahr = jahr;
    }
    public Datum () { // parameterloser Konstruktor
        this( 1, 1, 1);
        // tag = 1; monat = 1; jahr = 1; // ist auch moeglich
    }
}
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Datum.java>

5.3.2 Weitere Verbesserung des Geometrie-Beispiels

Motivation

5.3-10

Erinnerung

- Version 1 skaliert die Leinwand mit Hilfe getrennter Inventarlisten für Punkt bzw. Gerade-Objekten
- Skalierung beim Zeichnen von Punkt- UND Gerade-Objekten nur durch Clientcode möglich

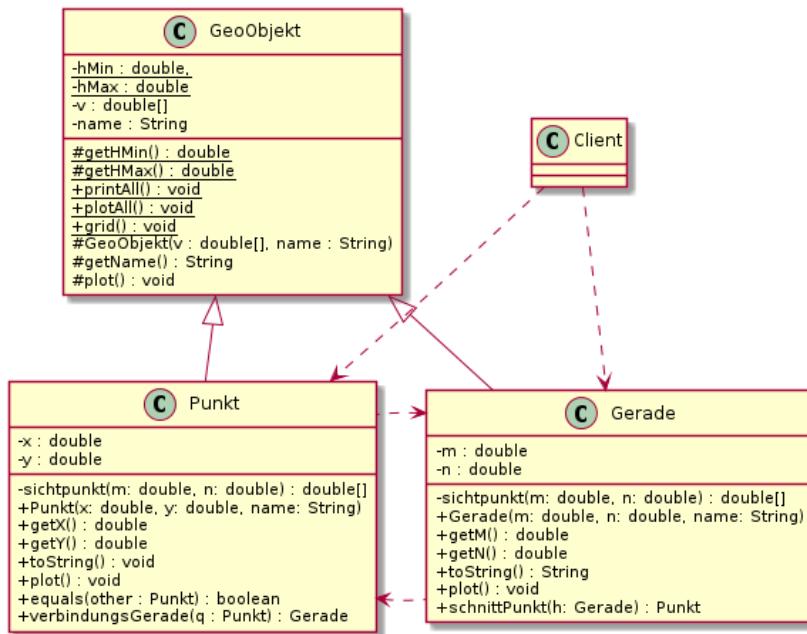
Wünschenswert: Objekte “sorgen selbst” für angemessene Zeichendarstellung (*objektorientierte Modellierung*)

Idee: Wiederverwendung von Code

- Punkt- und Gerade-Objekte haben
 - einen Namen und Sichtpunkt-Koordinaten
 - können gezeichnet werden
- programmiere Gemeinsames in Basisklasse GeoObjekt, Spezielles in Erweiterungsklassen Punkt und Gerade

Klassendiagramm

5.3-11



Erläuterung

Durchgezogene Pfeile zeigen Vererbung an, gestrichelte die sonstigen Abhängigkeiten zwischen den Klassen. Zugriffsberechtigungen sind am Vorzeichen abzulesen: `+`, `-` bzw. `#` stehen für `public`, `private` bzw. `protected` (siehe Paragraph 4.2-8). Statische Klassenelemente sind unterstrichen.

5.3-12 Beispiel: Die `plot()`-Methode

```

public class GeoObjekt {
    protected void plot() { // Beschriftung auf die Leinwand setzen
        StdDraw.setScale(hMin,hMax); // Skalierung der Leinwand (beide Achsen)
        StdDraw.setPenColor(StdDraw.BLACK);
        double dx, dy = dx = 0.025 * (hMax - hMin);
        StdDraw.text(v[0] + dx, v[1] + dy, name); // Name rechts ueber Sichtpunkt
    }
    // ...
}
  
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie2/GeoObjekt.java>

Jeder `plot()`-Aufruf mit einer **GeoObjekt**-Instanz erzwingt die Reskalierung der Leinwand — der Client-Code kann sich darauf verlassen. Ebenfalls verbessert gegenüber Version 1: die Aktualisierung der Horizontwerte `hMin`, `hMax` erfolgt gleich beim Konstruktoraufzug — das ist effizienter, als über die (diesmal gemeinsame) Inventarliste zu iterieren. Weitere Details bitte im Quelltext nachlesen.

```

public class Punkt extends GeoObjekt {
    public void plot() {
        super.plot();
        StdDraw.setPenColor(StdDraw.RED);
        StdDraw.point(x,y);
    }
    // ...
}
  
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie2/Punkt.java>

```

public class Gerade extends GeoObjekt {
    public void plot() {
  
```

```

super.plot();
StdDraw.setPenColor(StdDraw.BLUE);
double min = GeoObjekt.getHMin(), max = GeoObjekt.getHMax();
double span = max-min; // Spannweite, StdDraw-Doku: Leinwand ragt noch ...
min = min - 0.1 *span; max = max + 0.1*span; // .. 10% darueber hinaus
StdDraw.line( min, m*min + n,
              max, m*max + n);
}
// ...
}

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie2/Gerade.java>

Polymorphie

5.3-13

Erinnerung: Dynamisches Binden

- betrachte class E extends B { ... } und Referenz b vom (statischen) Typ B
- der Aufruf b.<methoden>(...); nutzt den *dynamischen Typ* von b: verweist b aktuell auf ein E-Objekt, so wird die Implementierung aus E verwendet

Beispiel: Person p; ..., int j = p.jahrgang()

Polymorphie (Vielgestaltigkeit)

- derselbe Code kann je nach Kontext andere Bedeutung haben
Beispiele auf Quellcode-Ebene:
 - Operator + (Addition bzw. String-Verkettung)
 - Überladen von Bezeichnern (z.B. gleichnamige Methoden mit verschiedenen Argumenttypen)
- dynamische Bindung berücksichtigt sogar den Laufzeit-Kontext

Anwendung

5.3-14

```

public class GeoObjekt {
    public static void printAll() {
        for (int l = 0; l < anzahl; l++)
            System.out.println(inventar[l]);
    }
    public static void plotAll() {
        for (int l = 0; l < anzahl; l++)
            inventar[l].plot();
    }
    // ...
    protected static void grid() { // Gitterlinien an ganzzahligen Koordinaten
        // ...
    }
}

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie2/GeoObjekt.java>

- Aufruf aus Methode im Client-Code: GeoObjekt.plotAll(); (siehe GeoClient.java)
- alle erzeugten geometrischen Objekte werden ihrem Typ entsprechend gezeichnet

5.4 Abstrakte und finale Klassen

5.4.1 Dritte Variante des Geometrie-Beispiels

5.4.1 Abstrakte objektorientierte Modellierung

Beispiel 1

Lebensmittel sind eine Abstraktion von speziellen Dingen, die wir essen (Kartoffeln, Brot, Suppe, ...)

- gemeinsame Eigenschaften: Kalorien, Beschaffenheit, ...
- verschiedene Art der Zubereitung (z.B. schälen, schneiden, kochen, ..)
- objektorientierte Modellierung: Klasse `Lebensmittel` fordert, dass Erweiterungsklassen `Kartoffel`, `Brot`, `Suppe`, ... eine Instanzmethode `zubereiten()` implementieren

Beispiel 2: Klasse GeoObjekt

- `plot()` aus `GeoObjekt` sorgt für die Beschriftung geometrischer Objekte
- ansonsten eher sinnlos: Zeichnen von Punkt- oder Gerade-Objekten sehr verschieden (die Methode war allerdings nützlich für die anschauliche Demonstration von Polymorphie)
- besser: implementiere stattdessen `showLabel()` zum Beschriften und fordere die Implementation von `plot()` in Erweiterungsklassen, wobei `showLabel()` genutzt werden kann
- analog für hypothetische Klassen `Kreis`, `Strecke`, ...

5.4.2 Abstrakte Klassen definieren

Problem

Wie kann man Entwickler von `Kreis`, `Strecke`, ... zwingen, eine `plot()`-Methode zu implementieren?

Lösung

- deklariere `plot()` in `GeoObjekt` als **abstrakte Methode**, indem lediglich die Schnittstelle angegeben wird, keine Implementierung
- Schnittstellen nicht-implementierter Methoden *und* die Klasse müssen mit `abstract` gekennzeichnet werden
- Wirkung:
 - Compiler akzeptiert nur Erweiterungsklassen, die die abstrakten Methoden implementieren
 - die Klasse *muss* erweitert werden, um die verlangte Implementierung nachzuliefern

- es können keine Objekte dieser Klasse erzeugt werden! — nur Objekte von Erweiterungsklassen

Java-Code

5.4-3

```

public abstract class GeoObjekt {
    protected void showLabel() { // Beschriftung auf die Leinwand setzen
        // ...
    }
    public abstract void plot();
    // ...
}

http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie3/GeoObjekt.java

```



```

public class Punkt extends GeoObjekt {
    public void plot() {
        showLabel(); // aequivalent: super.showLabel();
        StdDraw.setPenRadius(0.02);
        StdDraw.setPenColor(StdDraw.RED);
        StdDraw.point(x,y);
    }
    // ...
}

http://www.stud.informatik.uni-goettingen.de/infol/java/Geometrie3/Punkt.java

```

(Gerade.java analog) Entwickler sind frei in der Implementation der Methode, nur die Schnittstelle muss stimmen.

5.4.2 Finale Klassen

Beispiel: Elementare Daten in Objekte “verpacken”

5.4-4

Motivation

- Daten vom Typ `int`, `double`, `char` usw. sind *keine* Objekte
- es gibt aber Situationen, in denen eine Art Objektmechanismus erforderlich ist (z.B. Kapselung, Polymorphie)

Wrapper-Klassen

- Java-API bietet Klassen `Integer`, `Double`, `Character` usw., um elementare Daten
 - “artgerecht in Objekte zu wickeln”:
`Integer i = new Integer(42);`
 - und sie wieder “auszuwickeln”: `int j = i.intValue();`
- Außerdem sammeln diese Klassen `static`-Methoden zum Umgang mit den entsprechenden elementaren Daten
Beispiel: `int j = Integer.parseInt(args[0]);`
- die Objekte sind unveränderlich (wie `String`- und auch `Punkt`- und `Gerade`-Objekte), d.h. Wertänderungen müssen mit Hilfe neuer Objekte simuliert werden:

```
Integer k = new Integer(i);
k = new Integer(k.intValue() + 1);
```

5.4.5 Finale Klassen

Versuch (den der Compiler verhindert)

Eigene Klasse für erleichterten Umgang mit “eingewickelten” Objekten:

```
public class MyInteger extends Integer { // Compilezeit-Fehler!
    public void set(int v) {
        // ... veraenderlicher Inhalt
    }
}
```

final deklarierte Klassen

Wrapper-Klassen sind `final` deklariert, solche Klassen *dürfen* nicht erweitert werden (im Gegensatz zu abstrakten Klassen, die erweitert werden *müssen*):

```
public final class Integer {
    // ...
}
```

- (1) Finale Klassen sind ein Sicherheitsmechanismus: Angenommen, es gelänge, `MyInteger` wie versucht zu implementieren. Client-Code der mit einer `Integer`-Referenz `i` arbeitet, die auch `MyInteger`-Objekte referenzieren kann, benutze an einer Stelle den Aufruf `i.compareTo(100)`. Falls `MyInteger` diese Methode anders implementiert als in der API-Dokumentation beschrieben, so entsteht je nach aktuellem dynamischen Typ von `i` ggf. ein unerwartetes Ergebnis. Genau aus diesem Grund sind zentrale Java-API-Klassen `final` deklariert.
- (2) Mittels `final` kann man auch einzelne Klassenelemente, z.B. Methoden vor dem Überschreiben schützen. Ist die ganze Klasse `final`, so ist — trivialerweise — jedes seiner Elemente geschützt.

5.5 Abstrakte Datentypen

5.5.1 Abstrakter Datentyp Stack

5.5.1 Konkrete und abstrakte Datentypen

Konkreter Datentyp

= Menge T von Werten und Operationen mit bestimmten Eigenschaften (*Axiome*)

Beispiel int ist eine Menge T ganzer Zahlen ...

... mit Addition $+ : T \times T \rightarrow T$, Multiplikation $: T \times T \rightarrow T$ usw.*

Axiome

+ ist kommutativ	$x + y = y + x$
+ ist umkehrbar	zu x ex. genau ein y mit $x + y = 0$
* ist kommutativ	$x * y = y * x$
* ist distributiv	$(x + y) * z = x * z + y * z$

Ein abstrakter Datentyp (ADT)

beschreibt Menge von Operationen mit Eigenschaften

- zu verstehen als Sammlung von Forderungen an Klassen, die solche Operationen *konkret* implementieren
- Art der Implementierung ist nicht vorgeschrieben

vollständige Trennung von Schnittstelle und Implementation!

ADT Stack (algebraische Spezifikation)

5.5-2

DefinitionEin **Stack** (*Stapel*) über einer Menge T ist definiert durch:

Operation	anschaulich (Kartenstapel)
<code>create : → Stack<T></code>	Stapelplatz anlegen
<code>push : Stack<T> × T → Stack<T></code>	Karte oben auflegen
<code>pop : Stack<T> → Stack<T> × T</code>	oberste Karte abheben
<code>empty : Stack<T> → boolean</code>	Karten alle?

Einige Axiome	anschaulich
<code>pop(push(s, x)) = (s, x)</code>	Karte auflegen und abheben ändert nichts
<code>pop(create)</code> nicht erlaubt	von leerer Stapel keine Karte entfernbare
<code>empty(create) = true</code>	gerade angelegter Stapel ist leer
<code>empty(push(s, x)) = false</code>	nach Karte auflegen ist Stapel nicht leer

Die vollständige algebraische Spezifikation ist im Detail etwas anders — für uns nicht so wichtig.

Wozu braucht man das?

- z.B. zur Organisation des Laufzeitstapels (siehe Abschnitt 3.4.2)

ADT Stack (programmiertechnische Spezifikation)

5.5-3

Idee

- jede Klasse erweitert (direkt oder indirekt) die Klasse `Object` (siehe Paragraph 5.3-7)
- ⇒ verwende `Object` anstelle T

Quellcode

```
public interface Stack {
    // kein Konstruktor!
    public boolean empty();      // true falls nichts gespeichert ist, sonst false
    public void push(Object o); // speichert o ein
    public Object pop();        // liefert und entfernt juengste Referenz
}
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/ObjectADT/Stack.java>

- Name der Quelldatei entspricht Name des interface
- interface-Definition darf keinen Konstruktor und keine Methodendefinition enthalten, nur Schnittstellendefinitionen (abstract-Kennzeichnung ist erlaubt aber unnötig, da schon durch interface impliziert)
- Axiome sind im Kommentar informell beschrieben (Kartenstapel-Analogie)

5.5-4 EVLStack — eine konkrete Implementierung von Stack

```
class ObjectKnoten {  
    protected Object data; protected ObjectKnoten next;  
    ObjectKnoten (Object d, ObjectKnoten n) {  
        data = d; next = n;  
    }  
}
```

```
ObjectKnoten ist wie Zahlknoten implementiert (Paragraph 4.2-18), nur mit Object als Nutzdaten-
```

```
public class EVLStack implements Stack { // als einfach verkettete Liste
    private ObjectKnoten first;
    public EVLStack() { first = null; }
    public boolean empty() { return (first == null); }
    public void push(Object o) {
        first = new ObjectKnoten( o, first);}
    public Object pop() {
        if (empty())
            throw new RuntimeException("pop() von leerem Stack!");
        Object o = first.data;
        first = first.next;
        return o;}
}
```

- Zeile 1: Zusammenhang zu Stack wird durch `implements` angezeigt, ansonsten eine ganz normale Klasse
 - Zeile 2: `first` verweist auf Anfangsknoten bzw. auf keinen Knoten (`null`), falls Liste leer.
 - Zeile 3: Der Konstruktor setzt `first` nur auf den Default-Wert.
 - `push(o)` (Zeile 5) aktualisiert `first`, so dass auf neuen Knoten verwiesen wird, dessen Nachfolger der bisherige Anfangsknoten ist. `pop()` dagegen (Zeile 7) "biegt" `first` um auf Nachfolger des bisherigen Anfangsknotens.

Nicht gezeigt: Die Klasse überschreibt auch die `toString()`-Methode.

Der Code hat Verwandtschaft mit dem von **Zahlspeicher** in Paragraph 4.2.19: Die dortige Methode speichere ist die 1-zu-1-Entsprechung von push. Dortige Referenz anfang entspricht offenbar der hierigen first, jedoch mit einem entscheidenden Unterschied: anfang ist eine **statische Variable**, darum ist bei **Zahlspeicher** nur der Umgang mit einer einzigen Liste möglich, während die **Instanzvariable** start offenbar beliebig viele Stacks ermöglicht.

5.5-5 Test

```
public static void main(String[] args) { // Test-Unit
    Stack s = new EVLStack(); // Statischer Typ: Stack, dynamischer Typ: EVLStack
    String str;
    while (!StdIn.isEmpty()) {
        str = new String(StdIn.readLine());
        s.push(str);
    }
    while (!s.empty() ) { // s.pop() liefert Object-Referenz ..
        str = (String) s.pop(); // .. => cast auf String erforderlich
        System.out.println(str);
    }
}
```

Der Stack wird hier zum Speichern von Strings benutzt, aber int-Werte sind möglich, wenn Zeilen 3, 5–6 und 9 Integer-entsprechend angepasst werden (siehe Paragraph 5.4–4).
Nicht ganz so: Bei Aufruf mit Kommandozeileparametern werden Stack-Inhalte als Text ausgetauscht.

5.5-6 ArrayStack – zweite Implementierung von Stack

Idee

- speichere Object-Referenzen in einem Feld (*Puffer*)

- count hält Index der nächsten freien Speicherstelle

Code

```

1  public class ArrayStack implements Stack { // als Feld
2    final private int MAX = 16; // Kapazität
3    private Object[] elements;
4    private int count;           // Anzahl gespeicherter Werte
5    public ArrayStack() { elements = new Object[MAX]; count = 0; }
6    public boolean empty() { return (count == 0); }
7    public void push(Object o) {
8      elements[count++] = o;
9    }
10   public Object pop() {
11     if (empty())
12       throw new RuntimeException("pop von leerem Stack!");
13     --count;
14     Object o = elements[count];
15     elements[count] = null; // Garbage-Collector kann entspr. Objekt entfernen
16     return o;
17   }
}

```

- Zeile 1: Zusammenhang zu Stack wird durch implements angezeigt, ansonsten eine ganz normale Klasse
- Zeile 10 kombiniert Inkrement mit anderen Operatoren. Das ist meist riskant, hier nicht: Das *Postfix-Inkrement* count++ sorgt dafür, dass count erst *nach* der Zuweisung .. = o inkrementiert wird. Alternative Formulierung: elements[count] = o; count++;
- Zeilen 9 und 13: count wird bei Speichervorgängen angepasst

Dynamische Größenanpassung

5.5-7

Problem

Wie gezeigt können nur MAX viele Elemente gespeichert werden!

Ausweg: Umspeicherung

- in größeres Feld, wenn Kapazität erreicht
- in kleineres Feld, wenn Kapazität deutlich unterschritten

(MAX ist nur die Anfangskapazität)

```

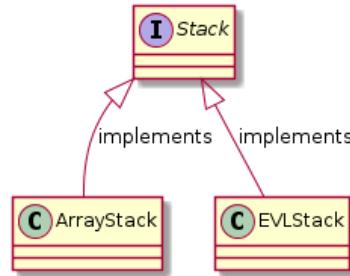
private void resize(int len) { // dynamische Laengenanpassung
    Object[] tmp = new Object[len]; // durch ..
    for (int i = 0; i < count; i++) // .. Umspeicherung ..
        tmp[i] = elements[i];      // .. in neues Feld und ..
    elements = tmp;               // .. Bindung an Instanzvariable
public void push(Object o) {
    if (count == elements.length) // Kapazität erreicht
        resize(2 * elements.length);
    elements[count++] = o;
}
public Object pop() {
    if (empty())
        throw new RuntimeException("pop von leerem Stack!");
    --count;
    Object o = elements[count];
    elements[count] = null; // Garbage-Collector kann entspr. Objekt entfernen
    if (count == elements.length/4)
        resize(elements.length/2);
    return o;
}

```

Ähnlich kann man sich die Implementierung von `StringBuilder` und `StringBuffer` vorstellen (siehe Paragraph 4.4-7).

5.5-8 Test der Implementationen

zwei äquivalente Implementierungen von Stack?



Ja!

- Testunits speichern beliebig viele Textzeilen als Strings und geben sie in umgekehrter Reihenfolge aus
- mit cat <Textdatei> | java EVLStack | java ArrayStack kann man überprüfen, dass die Datei korrekt wiedergegeben wird

5.5.2 Abstrakter Datentyp Queue

5.5-9 ADT Queue (Algebraische und technische Spezifikation)

Eine **Queue** (Schlange) über T ist definiert durch

Operation	Beispiel: Warteschlange
create : \rightarrow Queue(T)	Kundenschalter öffnen
enqueue : Queue(T) \times $T \rightarrow$ Queue(T)	Kunde stellt sich an
dequeue : Queue(T) \rightarrow Queue(T) \times T	Kunde wird abgefertigt
empty : Queue(T) \rightarrow boolean	alle abgefertigt?

Einige Axiome

deq. (enq. ((create()), x)) = create()	einzeln Wartenden abfertigen
deq. (enq. (create(), x)) = x	wer zuerst kommt mahlt zuerst
deq. (enq. (enq. (q, y), x)) = enq. (q, y)	einmal Erster immer Erster
empty(create()) = true	anfangs keine Wartenden

Interface

```

public interface Queue {
    public boolean empty();           // true falls nichts gespeichert ist, sonst false
    public void enqueue(Object o);    // fuegt o ein
    public Object dequeue();          // liefert und entfernt aelteste Referenz
}
  
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/ObjectADT/Queue.java>

Wozu braucht man das?

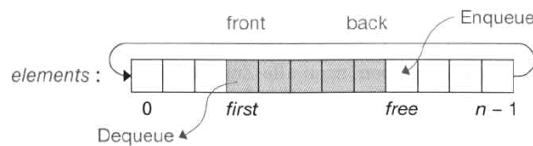
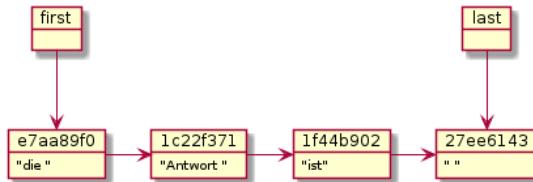
siehe Paragraph 1.1-21 z.B. zur Pufferung von Daten: Erzeugung und Verarbeitung zeitlich entkoppeln

Datentyp Queue: Ideen zur Implementierung

5.5-10

Ringpuffer

Feld + Indizes first und free: "erster Kunde" und "nächster Anstellplatz"

*Einfach verkettete Liste (Einfügen am Ende, Entnehmen am Anfang)*

Inhalt:

< "die ", "Antwort ", "ist", " " <

Ausschnitt aus EVL-Implementierung

5.5-11

```

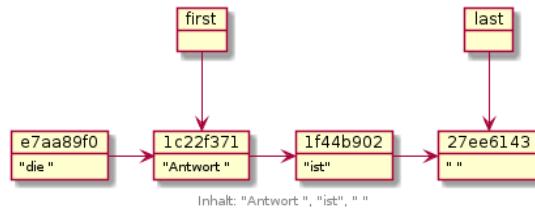
public class EVLQueue implements Queue { // als einfache verkettete Liste
    private ObjectKnoten first, last;
    public EVLQueue() { first = null; last = null; }
    public boolean empty() { return (first == null); }
    public void enqueue(Object o) {
        ObjectKnoten oldlast = last;          // (1)
        last = new ObjectKnoten( o, null);    // (2)
        if (!empty())
            oldlast.next = last;           // (3a)
        else
            first = last;                // (3b)
    }
    public Object dequeue() {
        if ( empty())
            throw new RuntimeException("dequeue() von leerer Queue!");
        Object o = first.data;
        first = first.next;               // (1)
        if (empty()) last = null;         //
        return o;
    }
}

```

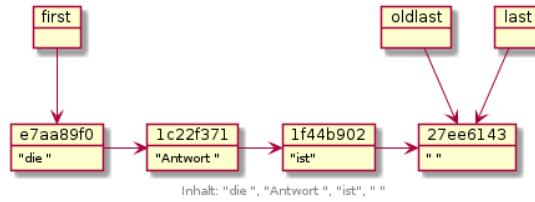
<http://www.stud.informatik.uni-goettingen.de/infol1/java/ObjectADT/EVLQueue.java>

Details zum Entfernen eines Elements: `q.dequeue();`
`first = first.next; // (1)`

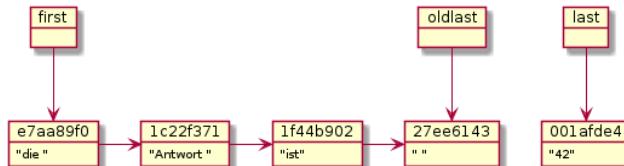
5.5-12



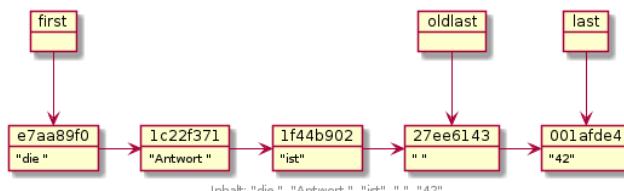
5.5-13 Details zum Einfügen: `q.enqueue("42");`
`ObjectKnoten oldlast = last; // (1)`



`last = new ObjectKnoten(new String("42") , null); // (2)`



`ObjectKnoten oldlast.next = last; // (3)`

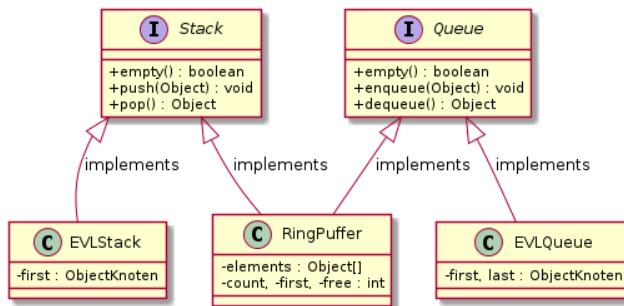


5.5-14 Trennung von Schnittstelle und Implementierung

Klassen können

- direkt nur von *einer Klasse erben* (nur *Einfachvererbung!!*)
- aber *mehrere Interfaces implementieren* (*Schnittstellenvererbung*)

```
public class RingPuffer implements Stack, Queue { ... }
```



→ Info1-Sammlung (II-ID: 61qk69g0l6i0)

Typischer Code (aus Testunit von EVLQueue)

Queue q = new EVLQueue(); // statisch: Queue, dynamisch: EVLQueue
Allgemein: <Interface> q = new <Konstruktor der Implementierklasse>(...);

5.5.3 Einführung in generische Typen (Java Generics)

Für nachfolgend diskutierte Quelltexte mit generischen Typen wird “passives Verständnis” erwartet: Sie sollen sie verstehen und benutzen, brauchen Sie in Info I aber nicht selbst programmieren.

Probleme mit `Object`-basierten ADTs

5.5-15

Beispiel

Der Compiler erlaubt folgenden Code, der offensichtlich Probleme bereiten muss!:

<pre> 1 public class ObjectADTClient { 2 public static void main(String[] args) { 3 Stack s = new EVLStack(); Object o; 4 5 double d = StdIn.readDouble(); // z.B. 1.0 einlesen 6 s.push(Double); // Eingabe als Double-Object einspeichern, .. 7 o = s.pop(); // .. ausspeichern, dann .. 8 double d1 = ((Double) o).doubleValue(); // .. Wert bestimmen 9 10 String str = StdIn.readString(); // z.B. "2.0" einlesen 11 s.push(str); // Eingabe als String-Objekt einspeichern, .. 12 o = s.pop(); // .. ausspeichern, dann versuchen, Wert zu bestimmen: 13 double d2 = ((Double) o).doubleValue(); // Laufzeitfehler! 14 } 15 }</pre>	<p>http://www.stud.informatik.uni-goettingen.de/inf1/java/ObjectADT/ObjectADTClient.java</p> <p>Object-basierte Stacks können beliebige Objekte verwalten, denn alle Klassen erben ja von <code>Object</code>. Deswegen sind die beiden Codeabschnitte erlaubt. Allerdings steht der Typ des an <code>o</code> gebundenen Objekts erst zur Laufzeit fest, so dass Type casts (Zeilen 8 und 13) unter Umständen nicht anwendbar sind!</p> <p>Ergebnis des Aufrufs <code>echo 1.0 2.0 java ObjectADTClient</code>:</p> <pre> Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Double at ObjectADTClient.main(ObjectADTClient.java:13) </pre>
--	---

Möglichkeiten, die Typsicherheit zu erzwingen

5.5-16

Separate ADTs für jeden Typ

Beispiel: Interfaces DoubleStack, StringStack, ... mit entsprechenden Implementierklassen

Vorteil Implementationen können auf den jeweiligen Typ optimiert werden

Nachteil Code-Wiederholung

Code mit instanceof absichern

```
if (o instanceof Double)
    double d2 = ((Double) o).doubleValue();
```

Vorteil Optimierungsmöglichkeiten (s.o.)

Nachteil Gefahr logischer Fehler, da schlecht lesbarer Code

5.5-17 Stack mit generischem Typ (Beispielimplementation: EVL)

```
1  public interface Stack<Item> {
2      public boolean empty();
3      public void push(Item d);
4      public Item pop();
5  }

1  public class EVLStack<Item> implements Stack<Item>{
2      private class Knoten { // innere Klasse
3          Item data; Knoten next;
4          Knoten (Item d, Knoten n) {
5              data = d; next = n;}
6      }
7      private Knoten first;
8      public boolean empty() { return (first == null); }
9      public void push(Item d) {
10         first = new Knoten( d, first);}
11      public Item pop() {
12          if (empty())
13              throw new RuntimeException("pop() von leerem Stack!");
14          Item d = first.data;
15          first = first.next;
16          return d;
17      }
18  }
```

- Zeile 1: Durch diese Schreibweise wird `Item` als Platzhalter für einen tatsächlichen Typ definiert — hier sind beliebige Namen möglich.
- Zeilen 2–6: Hier wird eine Klasse innerhalb einer Klasse gekapselt. Die entsprechende `class`-Datei zur Knotenklasse ist nur aus `Stack` heraus zugreifbar — Probieren Sie es aus!

Übrigens ist in der Klasse gar kein Konstruktor programmiert! Deswegen kann nur der Default-Konstruktor benutzt werden (Paragraph 5.3-9), der auch genau das Richtige leistet, wie z.B. anhand von `EVLStack` zu sehen (Paragraph 5.5-4).

```

1  public static void main(String[] args) { // Test-Unit
2      Stack<String> s = new EVLStack<String>();
3      String str;
4      while (!StdIn.isEmpty()) {
5          str = new String(StdIn.readLine());
6          s.push(str);
7      }
8      while (!s.empty()) {
9          str = s.pop(); // kein cast auf String erforderlich!
10         System.out.println(str);
11     }
12 }
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/EVLStack.java>

- Zeile 2: Beim Konstruktorauftrag wird der gewünschte Grundtyp festgelegt.
- Zeile 9: Laufzeitfehler wie oben diskutiert sind ausgeschlossen: bereits der Compiler prüft auf Typverträglichkeit

Beachte :

- Stack generiert den Datentyp, je nach Wahl von Item
- ebenso gibt es generische Interfaces im Java-API

Dieses Interface muss man kennen

5.5-18

public interface Comparable<T>
spezifiziert nur eine Methode:

```
// vergleicht _dieses_ Objekt mit other
// liefert Wert > 0, = 0 oder < 0, je nachdem, ob
// _dieses_ > = oder < other ist
public int compareTo(T other);
```

- jede Klasse T, deren Objekte auf natürliche Weise vergleichbar sind, sollte Comparable<T> implementieren
- Beispiel: Integer, Character, ..., String

→ Info1-Sammlung (II-ID: nzccrug0l6i0)

Queue mit sortierter Einfügeoperation (auch *PriorityQueue* genannt)

5.5-19

```

1  public class SortedQueue<T extends Comparable<T>> {
2      private class Knoten { // innere Klasse ...
3          private Knoten anfang;
4          public void insert( T s ) {
5              Knoten neu = new Knoten( s, null );
6              Knoten a = null, b = anfang;
7              while (b != null && neu.data.compareTo( b.data ) > 0) {
8                  a = b; b = b.next;
9              }
10             if ( a == null ) anfang = neu;
11             else a.next = neu;
12             neu.next = b;
13         }
14     }
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/SortedQueue.java>

Die Knotenklasse ist wie in Paragraph 5.5-17 programmiert. insert ist praktisch übernommen von StringSort (Paragraph 4.5-6). Zeile 1 fordert, dass die Objekte der Klasse T untereinander vergleichbar sind. Interessant ist dabei extends anstelle implements. Neugierige finden hier Erklärungen:
<http://stackoverflow.com/questions/8537500/java-the-meaning-of-t-extends-comparable>

Anwendung: Ein Sortierfilter

```

1     SortedQueue<String> ps = new SortedQueue<String>();
2     while (!StdIn.isEmpty()) {
3         String s = StdIn.readLine();
4         ps.insert( s );
5     }
6     System.out.println(ps); // Ausgabe aller gespeicherten Werte
7     // ...
8     SortedQueue<Double> pi = new SortedQueue<Double>();
9     while (!StdIn.isEmpty()) {
10        double s = StdIn.readDouble();
11        pi.insert((Double) s);
12    }
13    System.out.println(pi); // Ausgabe aller gespeicherten Werte
14    // ...

```

<http://www.stud.informatik.uni-goettingen.de/infoi/java/SortFilter.java>

Zeilen 6 und 13 nutzen die `toString()`-Methode von `SortedQueue` (oben nicht gezeigt)

5.5.4 Programmierung mit abstrakten Datentypen

5.5-20 Sortieren vergleichbarer Objekte

Wie in Paragraph 4.2-22 angesprochen beruht obiger Sortierfilter im wesentlichen auf Insertion-Sort. Dieses Sortierverfahren hat quadratischen Aufwand (Paragraph 4.1-10). In `java.util.Arrays` ist ein effizienteres Verfahren implementiert, darum ist folgende Implementierung vorzuziehen:

Voraussetzung: `T` implements `Comparable<T>`

Sortieren mit `java.util.Arrays`

siehe Paragraph 3.1-5

```

import java.util.Arrays;
public class SortierClient {
    public static void main(String[] args) {
        final int N = Integer.parseInt(args[0]);
        Comparable[] f = new T[N]; // Komponententyp: stat. Comparable, dynam. T
        for (int i = 0; i < N; i++)
            f[i] = StdIn.readT(); // hypothetische Eingabemethode
        Arrays.sort(f); // <- Aufruf des Sortieralgorithmus
        for (int i = 0; i < N; i++)
            System.out.println(f[i]);
    }
}

```

- `Arrays.sort(..)` ist ein (als statische Methode realisierter) effizienter Sortieralgorithmus, wie man ihn z.B. für `double[]`-Daten programmieren könnte (siehe Kapitel 6)
- einziger Unterschied: Größenvergleich beruht auf `compareTo(..)` anstelle <

5.5-21 Auswerten von Ausdrücken

Vollständig geklammerte Ausdrücke

$\langle \text{Ausdruck} \rangle ::= \langle \text{Zahl} \rangle \mid (' \langle \text{Ausdruck} \rangle \langle \text{Operator} \rangle \langle \text{Ausdruck} \rangle ')$

In Worten: Ein Ausdruck ist entweder eine Zahl oder eine linke Klammer gefolgt von Ausdruck, Operator, Ausdruck und rechter Klammer. Das ist eine Vereinfachung: so werden nur zweistellige Operatoren erfasst und Klammern sparende Vorrangregeln werden auch nicht berücksichtigt. Die nachfolgende algorithmische Idee kann aber auch auf diese Fällen angepasst werden.

Beispiel: (1 + ((2 + 3) * (4 * 5)))

Gedankliche Auswertung beim Lesen von links nach rechts

Zahl als Wert merken

Operator separat merken

rechte Klammer letzten Operator/letzte Werte verrechnen, Wert merken

(Gesamt-)Ergebnis ändert sich offenbar nicht, wenn anstelle eines geklammerten Teilausdrucks dessen Wert stehen würde:

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) ) =  
( 1 + ( 5 * ( 4 * 5 ) ) ) =  
( 1 + ( 5 * 20 ) ) =  
( 1 + 100 ) =  
101
```

Das rechtfertigt die algorithmische Idee im nächsten Paragraphen. Vorher überlegen wir uns noch:

Wie merkt man sich die Ergebnisse/Operanden?

Das geht mit einem *Wertestack* (`double`) und einem *Operatorstack* (`char`).

Überlegen Sie sich, dass man das bei gedanklicher Auswertung des Ausdrucks im Prinzip selbst so macht!

Verarbeitung von Einzelzeichen

5.5-22

```
if (c == '('); // ignorieren
if (c == <ziffer>) valstack.push(c);
if (c == <operator>) opstack.push(c);
if (c == ')') { ... // hole Operanden/Operator und verrechne ..
    valstack.push(ergebnis); } // .. speichere Ergebnis
```

Restausdruck	gelesen	Op-Stack	Werte-Stack
(1 + ((2 + 3) * (4 * 5)))	(
1 + ((2 + 3) * (4 * 5)))	1		1
+ ((2 + 3) * (4 * 5)))	+	+	1
((2 + 3) * (4 * 5)))	(+	1
(2 + 3) * (4 * 5)))	(+	1
2 + 3) * (4 * 5)))	2	+	1 2
+ 3) * (4 * 5)))	+	+	1 2
3) * (4 * 5)))	3	++	1 2 3
) * (4 * 5))))	+	1 5
* (4 * 5)))	*	+	1 5
(4 * 5)))	(+	1 5
4 * 5)))	4	+	1 5 4
* 5)))	*	**	1 5 4
5)))	5	***	1 5 4 5
))))	+	1 5 20
))))	+	1 100
))))		101

5.5-23 Ein Interpreter für vollständig geklammerte Ausdrücke

```

1  public class Evaluate {
2      public static void main(String[] args) {
3          Stack<String> ops = new EVLStack<String>();
4          Stack<Double> vals = new EVLStack<Double>();
5          while (!StdIn.isEmpty()){// Strings anstatt Einzelzeichen! => Bestandteile ...
6              String s = StdIn.readString(); // .. müssen whitespace-getrennt sein!
7              if (s.equals("(")) ;
8              else if (s.equals("+"))    ops.push(s);
9              else if (s.equals("*"))    ops.push(s);
10             // usw.: alle Operatoren
11             else if (s.equals(")")) {
12                 String op = ops.pop();
13                 double v = vals.pop();
14                 if (op.equals("+"))     v = vals.pop() + v;
15                 else if (op.equals("*")) v = vals.pop() * v;
16                 // usw.: alle Operatoren
17                 vals.push(v);
18             }
19             else vals.push(Double.parseDouble(s));
20         }
21         StdOut.println(vals.pop());
22     }
23 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Evaluate.java>

5.5-24 Infix und Postfix

Beobachtung

Evaluate berechnet den gleichen Wert auf der Eingabe

(1 ((2 3 +) (4 5 *) *) +)

- fast der gleiche Ausdruck, nur stehen Operatoren *hinter* den Operanden
- Klammern könnte man im Prinzip sogar weglassen: 1 2 3 + 4 5 * * +

der Quelltext müsste dafür allerdings angepasst werden

Definition

Postfix-Notation Operatoren *hinter* den/dem Operanden (klammerfrei)

Infix-Notation gewohnte Schreibweise mit Klammern (Operatoren *zwischen* Operanden)

→ Info1-Sammlung (II-ID: bbve4zg0l6i0)

Anwendung: Compiler

- Assembler-Code (auch Java-Bytecode) enthält Ausdrücke nur in Postfix-Format
- Rechenwerk-Steuerung benutzt Op-Stack und Value-Stack
- auch die Auswertung von Methodenaufrufen ist Stack-basiert (siehe Abschnitt 3.4.2) — die Compilation eines imperativen Programms folgt letzten Endes ähnlichen Prinzipien wie der in der Aufgabe angesprochene “Compiler”

Kapitel 6

Algorithmen und Datenstrukturen

6.1 Performance-Betrachtungen

Ressourcenverbrauch bei Ausführung eines Programms

6.1-1

Informatiker müssen die Kosten für Software-Einsatz beurteilen können, insbesondere maschinelle Ressourcen:

Rechenzeit (Laufzeit)

= Zeit, in der der Prozessor das Programm ausführt

- Zeit ist nicht wiederverwendbar

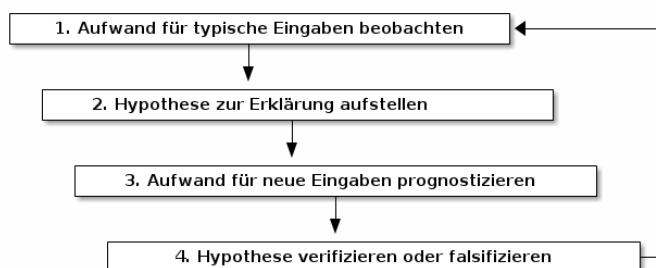
Speicher

= zur Ausführung benötigte Menge an Hauptspeicher

- Speicher ist wiederverwendbar (Bsp. Speicher für verwaiste Objekte wird freigegeben und erneut belegt)

Die naturwissenschaftliche Methode

6.1-2



Anforderungen an Hypothesen

- *Reproduzierbarkeit*: Experimente müssen unter gleichen/ähnlichen Bedingungen wiederholt werden können, ohne die Gültigkeit der Hypothese einzuschränken
Beispiel: gleiche/ähnliche Daten
- *Falsifizierbarkeit*: fehlerhafte Hypothesen anhand Experimenten erkennen
Beispiel: “Das Programm benötigt endliche Zeit.” ist experimentell nicht falsifizierbar

6.1-3 Das 3-Sum-Problem

gegeben N Zahlen a_0, a_1, \dots, a_{N-1}

gesucht Anzahl der Tripel (i, j, k) mit $a_i + a_j + a_k = 0$

Beispiel: Bei Eingabe 30, -30, -20, -10, 40, 0, 10, 5 erfüllen 4 Tripel die Bedingung:
 $(30, -30, 0), (30, -20, -10), (-30, -10, 40), (-10, 0, 10)$

Eine brute-force-Lösung

```

1  public class ThreeSum {
2      public static int count(int[] a) { // generiere alle Tripel ...
3          final int N = a.length;
4          int cnt = 0; // ... und zaehle wie oft ...
5          for (int i = 0; i < N; i++) {
6              for (int j = i+1; j < N; j++) {
7                  for (int k = j+1; k < N; k++) {
8                      if (a[i] + a[j] + a[k] == 0) // ... es passt
9                          cnt++;
10                 }
11             }
12         }
13     }
```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/ThreeSum.java>

Beispielaufruf: java ThreeSum < ../../data/8ints.txt

6.1-4 Laufzeitmessungen auf der Kommandozeile

bash-Befehl date liefert aktuelles Datum+Uhrzeit:

Fri Jan 5 15:48:28 CET 2018

⇒ Zeitdifferenz zwischen Programm-Start und -Stop ermitteln (z.B. bei Shell-Skripten, die lange laufende Prozesse anstoßen)

bash-Befehl time <Kommando> liefert Rechenzeiten

Beispiel: 128000 Zahlen einlesen und numerisch sortieren (Ausgabe ignorieren)

```
time sort -n < 128Kints.txt > /dev/null
real    0m0.148s
user    0m0.140s
sys     0m0.008s
```

real Gesamtzeit zur Ausführung vom Laden und Starten bis zur Beendigung, wie von außen mit Stoppuhr gemessen (Uhr läuft auch während Unterbrechungen für andere Prozesse)

user für Ausführung des “user-Codes” (inklusive Bibliotheksaufen) verbrauchte Zeit

sys für Ausführung von Systemfunktionen verbrauchte Zeit (z.B. Plattenzugriffe)

Beobachtungen

6.1-5

Typische Ergebnisse von `time java Threesum <Datei>`
(N = 500, 1000, 2000, 4000 Zahlen)

Datei	user	Faktor
500ints.txt	0.21s	
1Kints.txt	0.47s	2.23
2Kints.txt	3.73s	8.01
4Kints.txt	20.52s	5.51

Hypothesen

qualitativ Die Laufzeit wird wesentlich bestimmt durch **Problemgröße N**.

quantitativ Bei Verdopplung der Problemgröße steigt die Laufzeit um etwa einen Faktor zwischen 2 und 8.

Laufzeit-Prognose für **8Kints.txt**: ca. 10s..160s

Aber die Messung ergibt eine deutlich längere Laufzeit:

Datei	user	Faktor
8Kints.txt	264.56s	12.89

Diskussion

6.1-6

Messmethode ist zu ungenau

- `time java ...` erfasst auch die Zeit zum Starten der virtuellen Maschine und für „Aufräumarbeiten“ am Ende
- Zeit für Daten-Einlesen / Feld-Anlegen ebenfalls von Problemgröße abhängig, daher nicht vernachlässigbar

Ausweg

- Klasse `Stopwatch` verwenden (siehe Paragraph 5.2-5)
- nur die *Kernberechnung* erfassen: Start *nach* Anlegen des Feldes, Stop unmittelbar nach Ende der Berechnung

6.1-7 Verdopplungstest

Zeitmessung+Vergleich für 3Sum auf N, 2N, 4N, ... Zahlen

```

1 public class DoublingTest {
2     public static double timeTrial(int N) {
3         int[] a = new int[N];
4         for (int i = 0; i < N; i++) {
5             a[i] = StdRandom.uniform(2000000) - 1000000;
6         }
7         Stopwatch s = new Stopwatch();
8         int cnt = ThreeSum.count(a);
9         return s.elapsedTime();
10    }
11    // in main: Aufruf mit N=256, 512, 1024, ... und Tabelle ausgeben
12 }
```

<http://www.stud.informatik.uni-goettingen.de/infol/java/DoublingTest.java>

Typische Messergebnisse

N	Zeit (s)	Faktor
512	0.06	2.90
1024	0.48	7.79
2048	0.92	1.94
4096	7.28	7.90
8192	57.92	7.96
16384	463.49	8.00

6.1-8 Extrapolation

Die „8-fach-Hypothese“

- Verdopplung von N führt zu 8-fachem Zeitaufwand für die Kernberechnung

Bei großem N wird auch time vergleichbare Werte liefern. (Der Aufwand für Start/Stop der JVM und für Feld-Anlegen und -Befüllen fallen kaum noch ins Gewicht.)

Überprüfung wäre sehr aufwendig!!

N	Zeit (s)	Faktor
...
16384	463.49	8.00
32764	ca. 1h	8
65528	ca. 8h	8
...

6.1-9 Die analytische Methode: Detaillierte Quellcode-Betrachtung

Bestimmende Faktoren der Laufzeit

- Einzelkosten = Kosten für Ausführung einzelner Anweisungen

- **Vielfachheit** $v(N)$ ihrer Ausführung (in Abhängigkeit von Eingabegröße)

Beispiel: ein Aufruf von `count (int [] a)`

```

1 // ----- Anweisung ----- // - Vielfachheit -
2 public static int count(int[] a) { // 1
3     final int N = a.length; // 1
4     int cnt = 0; // 1
5     for (int i = 0; i < N; i++) // 1
6         for (int j = i+1; j < N; j++) // N
7             for (int k = j+1; k < N; k++) // N(N-1)/2 = N*N/2 - N/2
8                 if (a[i] + a[j] + a[k] == 0) // N(N-1)(N-2)/6 = N*N*N/6 - N*N/2 + N/3
9                     cnt++; // datenabhängig
10    return cnt; // 1
11 }
```

Asymptotische Betrachtung (Teil 1)

6.1-10

Für große N fallen nur führende Terme ins Gewicht

Beispiel: Für $N = 1000$ ist $\frac{N^3}{6} \approx 166666667$ und $-\frac{N^2}{2} + \frac{N}{3} \approx -499667$.

Trick 1: Die Tilde-Notation benutzen

Sei $f(N)$ ein einfacher Term mit $v(N) \sim f(N)$.

Vereinfache komplizierte Ausdrücke durch Ersetzen von $v(N)$ durch $\sim f(N)$.

Was bedeutet \sim genau?

Falls

$$\lim_{N \rightarrow \infty} \frac{v(N)}{f(N)} = 1$$

so heißen $v(N)$ und $f(N)$ **asymptotisch äquivalent** (symbolisch: $v(N) \sim f(N)$).

Beispiel: $\frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3} \sim \frac{N^3}{6}$

Asymptotische Betrachtung (Teil 2)

6.1-11

```

1 // ----- Anweisung ----- // - Vielfachheit -
2 public static int count(int[] a) { // 1
3     final int N = a.length; // 1
4     int cnt = 0; // 1
5     for (int i = 0; i < N; i++) // 1
6         for (int j = i+1; j < N; j++) // N
7             for (int k = j+1; k < N; k++) // N(N-1)/2 ~ N*N/2
8                 if (a[i] + a[j] + a[k] == 0) // N(N-1)(N-2)/6 ~ N*N*N/6
9                     cnt++; // datenabhängig
10    return cnt; // 1
11 }
```

Trick 2: Konzentration auf die Anweisung größter Vielfachheit

Beispiel: Für große N gilt

$$1 \ll N \ll \frac{N^2}{2} \ll \frac{N^3}{6}.$$

Die **Kernoperation** („innere Schleife“) des Programms ist der Gleichheitstest. Der Aufwand dafür wächst *kubisch* mit der Eingabegröße.

Wachstumsordnung

6.1-12

Beobachtung

Für die Laufzeit $T(N)$ vieler Programme gilt

$$T(N) \sim cf(N),$$

wobei c konstant ist und die **Wachstumsordnung** $f(N)$ relativ einfache Gestalt hat, wie z.B. $\log N, N, N^2, N^3, \dots$

Die Konstante c

- ist für allgemeine Betrachtungen meist unwichtig

Beispiel: Die “8-fach-Hypothese” ergibt sich bereits aus

$$\frac{T(2N)}{T(N)} \sim \frac{c(2N)^3}{cN^3} = 8.$$

- hängt von den Einzelkosten der Kernoperation ab

6.1-13 Einzelkosten

- Zeit für vielfache Ausführung einzelner *typischer Aufrufe* mit messen, durch Vielfachheit teilen

Beispielhaftes Ergebnis:

Anweisungstyp	Kosten (Zeit)	allgemein
int-Addition/-Subtraktion/Vergleich	< 1ns	konstant
int-Multiplikation/-Division	< 4ns	konstant
double-Arithmetik + Vergleich	< 2ns	konstant
Schleifeniteration	< 1ns	konstant
Parameterübergabe an Funktion	nicht messb.	konstant
Aufruf von Math.sin	≈ 100ns	konstant
Aufruf von Math.random	≈ 50ns	konstant
Objekt anlegen		konstant
Feld anlegen		proportional zu Länge

- genaue Werte SEHR stark vom System und von Messmethode abhängig

6.1-14 Beispiel: Intel Core2 Duo CPU (2.66GHz)

java TimePrimitives 10000
[Link auf den Quelltext](#)

```
Nanoseconds per operation
empty loop:          0.64
integer addition:    2.13
integer subtraction: 0.12
integer multiply:    1.13
integer comparison:  0.25
integer remainder:   3.5
integer division:    3.5
floating-point addition: 0.0
floating-point subtraction: 1.39
floating-point division: 1.38
floating-point multiplication: 1.38
```

```

function call: 0.0
Math.sin:      95.3
Math.atan2:    101.29999999999998
Math.random:   43.09999999999994
integer addition: 0.0

```

→ InfoI-Sammlung (II-ID: cl5kwza1o7i0)

6.2 Wachstumsordnungen

Wachstumsordnung, Asymptotik und Laufzeit

6.2-1

Zusammenfassung: Aufwandsanalyse für eine Implementation

Erst Problemgröße N identifizieren, dann

1. Vielfachheiten und Einzelkosten einschätzen
2. Vielfachheiten durch Tilde-Notation vereinfachen
3. dominanten Term bestimmen
4. konstante Faktoren ignorieren, um Wachstumsordnung zu bestimmen

Übung/Erfahrung ermöglicht in Standardfällen schnelle Analyse ⇒ Routineuntersuchung vor Softwareeinsatz

Beispiel

```

// ----- Anweisung ----- // Vielfachheiten | Befehle |beispielhafte ...
public static int count(int[] a) { // ... |... Einzelkosten
    final int N = a.length; // 1 ---\ 
    int cnt = 0; // 1 > ----- | 6 x int-Arithm. |< 6ns
    for (int i = 0; i < N; i++) // 1 ----/
        for (int j = i+1; j < N; j++) // N | 4 x int-Arithm. |< 4ns
            for (int k = j+1; k < N; k++) // N*N/2-N/2 | 4 x int-Arithm. |< 4ns
                if (a[i] + a[j] + a[k] == 0) // N*N*N/6-N*N/2+N/3| 3xZugr.+3xdouble|<12ns
                    cnt++; // datenabh ngig | < 1 x int |< 1ns
    return cnt; // 1 | 1 x int |< 1ns
}

```

Gesamtkosten (in ns) $< 12\left(\frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3}\right) + 4\left(\frac{N^2}{2} + \frac{N}{2}\right) + 8 = 2N^3 - 4N^2 + 6N + 8$

Asymptotik $\sim \frac{N^3}{6}$ Kernoperationen (mit < 12ns Einzelkosten)

Wachstumsordnung N^3

Bemerkungen

1. echte Laufzeitschätzungen haben nur für ein konkretes System gewisse (eingeschränkte) Bedeutung.
2. Tilde-Notation erlaubt Leistungsvergleich zwischen Algorithmen gleicher Wachstumsordnung Beispiel: $\sim \frac{N^3}{6}$ ist besser als $\sim 2 \cdot N^3$

3. Wachstumsordnung bestimmt, ob der Algorithmus für große N praktikabel ist oder nicht. Für kleine N sind jedoch asymptotische Werte und vor allem Vielfachheiten $V(N)$ aussagekräftiger.

Wichtigste Wachstumsordinungen

Oft trifft man Wachstumsordinungen $f(N)$ dieser Form an:

$1, \log N, N, N \log N, N^2, N^3, 2^{cN}$, die Algorithmen
konstanter — **logarithmischer** — **linearer** — **loglinearer** — **quadratischer** — **kubischer**
— **exponentieller**

Laufzeit charakterisieren.

Verkürzte Ausdrucksweise (Beispiele)

- sequentielle Suche ist ein *linearer Algorithmus*
- SelectionSort ist ein *quadratischer Algorithmus*

usw.

6.2-2 Konstante Wachstumsordnung

charakterisiert Algorithmen die auf jeder Eingabe nur eine konstante Anzahl von Operationen ausführen

Beispiele (siehe Kapitel 1 und 2)

- Ausgabe eines festen Textes: HelloWorld
- Umrechnungen: Temperatur, Char2Value, Value2Char
- Fallunterscheidungen: Schaltjahr
- Berechnung von Ausdrücken: Gaussian.phi
- Aufruf von Funktionen der Math-Bibliothek

6.2-3 Logarithmische Wachstumsordnung

Algorithmen, die bei Eingabegröße N höchstens Faktor $\times \log N$ Operationen ausführen: fast so gut wie konstante Wachstumsordnung

Beispiele

- binäre Suche (Paragraph 2.3-5)
- Newton-Verfahren (Problemgröße: $1/\varepsilon$, ε = vorgegebene relative Genauigkeit — siehe Paragraph 2.2-24)

Bemerkung

$\log N$ steht für $\log_2 N$

Übergang zu anderer Basis bewirkt nur Änderung um konstanten Faktor (siehe Kapitel 7), ändert nicht die Wachstumsordnung

6.2-4 Lineare Wachstumsordnung

- falls jeder Eingabe-Bestandteil konstante Zeit erfordert, so ist die Wachstumsordnung linear
- ebenfalls linear: Programme mit einer einzigen `for`-Schleife der Lauflänge N

Beispiele

- `PlotFilter`
- ein Feld anlegen und default-initialisieren, Feld kopieren
- viele Algorithmen auf Feldern: Summe, Maximum, Minimum, Mittelwert, . . . , sequentielle Suche, kostet auch linearen Aufwand
- memoisierte Berechnung von Fibonacci-Zahlen (Paragraph 3.4-36)
- Fakultät berechnen, Horner-Schema, . . .
- Evaluation von Ausdrücken, Infix2Postfix-Compilation

Loglineare Wachstumsordnung

6.2-5

Algorithmen, die bei Eingabegröße N höchstens Faktor $\times N \log N$ Operationen ausführen: kaum schlechter als lineare Wachstumsordnung

bei Sedgewick/Wayne wird diese Wachstumsordnung
linearithmisch genannt

Beispiele

schnelle Sortieralgorithmen wie QuickSort und Mergesort, die wir noch behandeln haben loglineares Verhalten

Quadratische, kubische, . . . Wachstumsordnung

6.2-6

Algorithmen, die Paare, Tripel, . . . von Eingabeteilen einzeln behandeln haben meist (nicht immer!) quadratische, kubische, . . . allgemein: **polynomiale** Wachstumsordnung

Beispiele für Quadratische Wachstumsordnung

- bestimme alle Geradenschnittpunkte bei N gegebenen Geraden
- einfache Sortieralgorithmen, wie Bubble-, Selection- und Insertionsort (siehe Kapitel 2 und 3)
- clevere Lösung von 3Sum (siehe unten)

Beispiele für kubische Wachstumsordnung

- brute force-Lösung für das 3Sum-Problem (siehe oben)
- Matrixmultiplikation (falls $N = \text{Zeilenzahl der Matrix!}$)
- Euklidischer Algorithmus ($N = \text{Stellenzahl der Operanden}$)

Exponentielle Wachstumsordnung

6.2-7

Algorithmen, die „alle Möglichkeiten durchlaufen“ (müssen), haben oft exponentielles Verhalten

Beispiele

- TuermeVonHanoi
- zähle alle Teilmengen einer Menge auf
- Primzahltest durch Probewertdivision (N = Stellenzahl der Eingabe)
- naive rekursive Berechnung von Fibonacci-Zahlen
- Faktorisierung

6.2-8 Übersicht

Name	Funktion	Änderung bei Übergang $N \mapsto 2N$
konstant	1	keine
logarithmisch	$\log N$	+1
linear	N	$\times 2$
loglinear	$N \log N$	$\times 2 \left(1 + \frac{1}{\log N}\right) \sim 2$
quadratisch	N^2	$\times 4$
kubisch	N^3	$\times 8$
exponentiell	2^N	$\times 2^N$

Verdopplungstests

können helfen, Hypothesen über die Wachstumsordnung aufzustellen.

Beispiel: Wird z.B. von Wachstum $f(N) = N^c$ ausgegangen, so ist $c = \log \frac{f(2N)}{f(N)}$

6.2-9 Worst-case vs. average-case

- beobachtete Laufzeiten können stark von den Eingaben abhängen (ThreeSum und z.B. auch SelectionSort sind (seltene!) Ausnahmen)
- die analytische Methode ist meist am einfachsten im *worst case* (Verhalten im schlechtesten Fall) anzuwenden, die Resultate sind aber u.U. zu pessimistisch
- Verdopplungstests mit *geeignet* zufällig gewählten Eingaben können *average-case*-Hypothesen liefern (Verhalten im Mittel)

„Geeignete zufällige Wahl“ klingt seltsam. Gemeint ist damit, dass auch Zufallseingaben sorgsam „konstruiert“ sein müssen. Zum Beispiel gibt es $52! > \cdot 10^{66} \approx 2^{220}$ Anordnungen von 52 Spielkarten. Ein `Math.random()`-Aufruf kann höchstens 2^{64} verschiedene `double`-Werte (8 Byte) liefern. Wie sind also mehrere `Math.random()`-Aufrufe zu organisieren, um daraus jede mögliche Anordnung *mit gleicher Wahrscheinlichkeit* zu ermitteln?

6.3 Laufzeitvorhersagen

Betrachtung der Wachstumsordnung ist wichtig: Stellt sich der verwendete Algorithmus für interessante Problemgrößen als unpraktikabel heraus, muss man Alternativen suchen.

6.3-1 Beispiel: Clevere Lösung für das 3-Sum-Problem

Bei großen N sind brute-force-Lösungen generell unrealistisch. Für das 3-Sum-Problem sind Verbesserungen in der Literatur zu finden.

Idee

- sortiere die Zahlen: $a_0 \leq a_1 \leq \dots \leq a_{N-1}$
- $a_i + a_j + a_k = 0$ (gleichwertig: $a_i = -(a_j + a_k)$) kommt nur in Frage für j, k aus einem gewissen Intervall $B_i \subseteq \{i+1, i+2, \dots, N-1\}$
- wegen Sortierung: $B_i \supset B_{i+1} \supset \dots \supset B_{N-3}$
⇒ Suchbereich in jedem Schritt geeignet verkleinern

→ Info1-Sammlung (II-ID: 9azg8hl0q7i0)

Der Verdopplungstest zeigt: Innerhalb einer Minute werden Problemgrößen bearbeitet, mit denen die brute-force-Lösung tagelang zu tun hätte

N	Zeit	Faktor
8192	0.17	3.86
16384	0.67	3.95
32768	2.69	4.05
65536	10.70	3.97
131072	43.15	4.03

Beherrschbarkeit von Problemgrößen

6.3-2

100fache Vergrößerung der Problemgröße

Ein Programm, das bei Problemgröße N nur Sekunden läuft, läuft dann voraussichtlich wie lange?

Wachstumsordnung	Laufzeit bei Problemgröße $100N$
konstant	Sekunden
linear/loglinear	Minuten
quadratisch	Stunden
kubisch	Wochen
exponentiell	nicht beherrschbar

Die Wirkung schnellerer Rechentechnik

6.3-3

„Moore’s law“ (1965)

- alle 18 Monate wird die Integrationsdichte von Schaltkreisen verdoppelt
- (zu) optimistische Auslegung: alle 18 Monate verdoppelt sich bzw. alle 5 Jahre verzehnfacht sich die Rechenleistung verfügbarer Computer

10fach schnellere Rechner

Die gerade noch beherrschbare Problemgröße ist auf einem 10mal so schnellen Rechner um wieviel größer?

Wachstumsordnung	beherrschbare Vergrößerung um Faktor
linear/linearithmisch	10
quadratisch	3–4
kubisch	2–3
exponentiell	1

[6.3-4 Schlussfolgerungen für Programmierer](#)

Routine-Entscheidungen vor Beginn der Programmierung

- Lässt sich das Problem “auf was Bekanntes” zurückführen?
- mit welchem Laufzeit-/Speicherbedarf muss ich rechnen? (grob)

Kosten abwägen

- Algorithmen, die oft laufen werden, sollten möglichst effizient sein
- ein cleverer $\sim 1000 \cdot N$ -Algorithmus ist auf kleinen Eingaben kaum besser als ein einfach zu programmierender $\sim 2 \cdot N^2$ -Algorithmus
- Entwicklung und Wartung kosten mehr als man durch Codeoptimierungs-Hacks einsparen kann!

Klar denken, programmieren und dokumentieren!

- kenne und verwende etablierte Algorithmen und Datenstrukturen!
- achte auf Lokalität der Daten!
- verwende “sprechende” Bezeichner!
- dokumentiere angemessen!
- ermöglche bequeme Tests!

6.4 Ergänzungen

Dieser Abschnitt ist nicht klausurrelevant.

[6.4-1 Analyse des Speicherbedarfs](#)

Ähnlich wie bei der Laufzeitanalyse kann der asymptotische Speicherbedarf analysiert werden.

Besonderheiten

- “nomineller Speicherbedarf” laut Spezifikation bezieht sich auf die JVM
- tatsächlich belegter Speicher auf dem JVM-Host kann größer sein.

Elementare Daten

siehe Tabelle in Paragraph 2.1-34

Referenzvariablen

- Speicherbedarf (je nach JVM-Implementation) typischerweise 4 Bytes

Objekte und Felder

Zum Speicherbedarf der einzelnen Instanzvariablen kommt noch ein *Objekt-Overhead* (abhängig von JVM-Implementation und Art des Datenobjekts):

- Informationen für Garbage-Collection, Synchronisation usw.
- bei Feldern die Länge (in der Instanzvariablen `length`)
- bei Objekten eine Referenz auf den Bytecode der Klasse (siehe Paragraph 5.3-6)

Empirische Messung und Steuerung des Speicherbedarfs

6.4-2

Messung durch Shell-Kommando

`/usr/bin/time -v Kommando` listet die in Anspruch genommenen Systemressourcen auf. Achtung: `/usr/bin/time` erfasst die *Gesamt-Ressoucen*, also auch die Kosten, die nur durch die laufende JVM entstehen!

Steuerung durch Aufrufoptionen

Aufrufe wie `java -Xmx8m Klasse` `java -Xss8m Klasse` begrenzen Heap bzw. Stack der JVM bei diesem Lauf auf 8 MB. Siehe `man java` für weitere interessante Aufrufoptionen.

6.5 Sortieren und Suchen

6.5.1 Elementare iterative Sortierverfahren

Allgemeines Setup

6.5-1

Problemstellung

gegeben Feld `double[] f = new double[N];` mit beliebigen Werten

gesucht gleiche Werte in *aufsteigender* Sortierung, also

$$f[0] \leq f[1] \dots \leq f[N-1]$$

Allgemeine Code-Struktur

```
public class XYZ { // Modul zum Sortierverfahren XYZ
    public static void sort(double[] f) {
        // ...
    }
    public static void main(String[] args) { // Test-Unit
        // mit int-Kdzeilenargument: => Feld dieser Laenge initialisieren
        sort(f); // XYZ.sort(f) aufrufen
        // ohne Kdzeilenelement: => Verdopplungstests (mit Zufallsdaten)
    }
}
```

anstelle XYZ: Klassen Bubble, Selection, Insertion, Merge, Quick (Name entsprechend Verfahren)

6.5-2 Selection-Sort (Wiederholung)

Idee: Minimum nach vorn tauschen, dann ebenso weiter mit dem Rest.

Details: siehe Paragraph 4.1-7

```

1   for (int i = 0; i < N-1; i++) {
2       m = i;
3       for (int j = i+1; j < N; j++)
4           if ( f[j] < f[m])
5               m = j;
6           t = f[m]; f[m] = f[i]; f[i] = t;
7   }

```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Sorter/Selection.java>

Asymptotische Analyse

- Kernoperation: Bestimmung des min-Index in innerer Schleife
- Anzahl der Befehle i.W. proportional zu Anzahl $V(N)$ der Vergleiche zwischen Feldelementen $V(N) = (N - 1) + (N - 2) + \dots + 1 = \frac{N \cdot (N-1)}{2}$
- in jedem Fall $V(N) \sim \frac{N^2}{2}$ auch bei vorsortiertem Feld!

Empirische Analyse (Verdopplungstest)

java Selection	size	time	ratio
	512	0.00	0.33
	1024	0.00	0.75
	2048	0.00	1.00
	4096	0.01	4.67
	8192	0.05	3.43
	16384	0.19	4.04
	32768	0.77	3.97
	65536	3.05	3.95

Verallgemeinerung

public static void sort(Comparable<T>[] f) (anwendbar auf Felder, die z.B. Referenzen auf umfangreiche Daten speichern) ist ebenso effizient bzw. ineffizient, da bei Zuweisungen (wie in Zeile 6) nicht die Daten selbst, sondern nur die Referenzen kopiert werden.

6.5-3 Insertion-Sort (Wiederholung)

Idee: Nacheinander richtig in bereits sortierte Liste einfügen.

Details: siehe Paragraph 4.1-9

```

1   for (int i = 1; i < N; i++)
2       for (int j = i; j > 0 && f[j] < f[j-1]; j--) {
3           double t=f[j-1]; f[j-1]=f[j]; f[j]=t;
4       }

```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Sorter/Insertion.java>

Asymptotische Analyse

- Kernoperation: Durchtauschen in innerer Schleife
 - Anzahl der Befehle i.W. proportional zu Anzahl $V(N)$ der Vergleiche zwischen Feldelementen
 - schlechtester Fall: Feld ist anfangs *absteigend sortiert*

$$V(N) = (N - 1) + (N - 2) + \dots + 1 = \frac{N \cdot (N - 1)}{2}$$
 - worst-case: $V(N) \sim \frac{N^2}{2}$
 - best-case: $\sim N$ für Sortiertheitstest Anfangsfeld aufsteigend sortiert
 - average-case: $\sim \frac{N^2}{4}$
- Heuristik: bei zufälligem Anfangsfeld ist auch Einfügeposition zufällig, d.h. im Schnitt nur halb soviel Vergleiche wie im worst-case

—

Empirisch (Verdopplungstest)

java Insertion	size	time	ratio
	512	0.01	5.00
	1024	0.00	0.60
	2048	0.00	0.67
	4096	0.01	2.50
	8192	0.03	5.40
	16384	0.12	4.52
	32768	0.50	4.10
	65536	2.06	4.11

Bemerkung

Sortierfilter wie in Paragraph 5.5-19 beruhen ebenfalls auf der Insertion-Idee, haben daher auch quadratisches Laufzeitverhalten.

Übungsaufgabe: Bubble-Sort

6.5-4

Frage

Welches worst-, best-, average-case-Verhalten hat BubbleSort?

→ Info1-Sammlung (II-ID: 4u1e78l0q7i0)

Antwort

- worst-case: $\sim \frac{N^2}{2}$
- best-case: $\sim N$
- average-case: $\sim \frac{N^2}{4}$

Speicherbedarf

6.5-5

Beobachtung

Die bisher betrachteten Algorithmen benötigen nur $\sim N$ Speicherzellen

Begründung

Außer dem Bedarf für das Feld selbst wird nur Speicher benötigt für

- Laufvariablen i , j und Feldlänge N
- Zwischenspeicher für aktuell besondere Indizes (z.B. m für Minimum-Index) oder aktuelle Einträge (z.B. t für den Tausch)

Definition

Sortierverfahren, die zusätzlich zur Eingabe nur konstant viel Speicher benötigen, heißen **in-place** oder **in-situ**-Verfahren.

6.5.2 Teile- und Herrsche-Algorithmen für das Sortieren

6.5-6 Merge-Sort

Idee: Teile und Herrsche

- falls $N = 1$, so ist das Feld sortiert
- anderenfalls
 - zerlege Feld in Teilstücke der Größe $\lceil N/2 \rceil$
 - sortiere Teilstücke rekursiv
 - füge Teilstücke sortiert zusammen, indem die Werte “der Kleine nach” ausgewählt werden

sort(..) rekursiv organisieren

```

1  public static void sort(double[] f) {
2      sort(f, 0, f.length);
3  }
4  public static void sort(double[] f, int lo, int hi) { // Indizes lo <= i < hi
5      final int N = hi - lo;
6      // ...
7  }

```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Merge.java>

6.5-7 Beispiel

1	4	2	7	8	3	6	5
1	4	2	7 8	3	6	5	
1	4 2	7 8	3 6	5			
1 4 2 7 8 3 6 5							
1 4 2	7 3	8 5	6				
1 2	4	7 3	5	6	8		
1	2	3	4	5	6	7	8

Realisierung

6.5-8

```

1  public static void sort(double[] f, int lo, int hi) { // Indizes lo <= i < hi
2      final int N = hi - lo;
3      if (N <= 1) return; // Feld der Laenge 1 ist bereits sortiert
4      int mid = lo + N/2; // Mitte bestimmen
5      // sortiere rekursiv "unteres" u. "oberes" Teilstfeld:
6      sort(f,lo,mid); sort(f,mid,hi);
7      double[] hilfs = new double[N];
8      // Merging:
9      // Komponenten der Teilstfelder "der Kleine nach" in hilfs kopieren:
10     int i = lo, j = mid, k; // naechste zu betrachtenden Komponenten
11     for (k = 0; i < mid && j < hi; k++)
12         hilfs[k] =
13             f[i] <= f[j] ?
14                 f[i++] : f[j++]; // das Kleinere gewinnt!
15     // nun gilt: i == mid oder j == hi => kopiere noch den Rest
16     while (i < mid)
17         hilfs[k++] = f[i++];
18     while (j < hi)
19         hilfs[k++] = f[j++];
20     for (k = 0; k < N; k++) // alles aus dem Hilfsfeld ...
21         f[lo + k] = hilfs[k]; // .. zurueckkopieren
22     }

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Merge.java>

Zeilen 12-14 ist eine Anwendung des “Elvis-Operators” zu sehen (siehe Paragraph 3.3-9). Äquivalente Formulierung:
`if (f[i]<=f[j]) hilfs[k]=f[i++]; else hilfs[k]=f[j++];`
 Eine Zuweisung wie `a=b[m++]`; wiederum ist gleichwertig zu `{a=b[m]; m++}`.

Laufzeitanalyse

6.5-9

Der Einfachheit halber sei N eine Zweierpotenz.

- Teilungsschritt (“Teile”):
 - bestimme Mitte `mid` und starte rekursiven Aufruf (konstant viele Schritte)
 - über alle Aufrufe gerechnet werden $N - 1$ Teilungsschritte gemacht (bis jede Komponente in einem Feld der Länge 1 ist)
 - \Rightarrow insgesamt $\sim N$ Schritte für Aufteilungen

• Beobachtung :

- es entstehen 2 Teilstfelder der Länge $N/2$, 4 der Länge $N/4$, ..., N der Länge 1
- der Rekursionsbaum hat die Tiefe $\log_2 N$

siehe Paragraph 3.4-31

• Zusammenfügen (“Herrsche”):

- 2 Teilstfelder der Länge ℓ erfordern maximal 2ℓ Vergleiche zum Zusammenfügen
- auf Ebene i des Rekursionsbaums (Aufrufe in Entfernung i von der Wurzel) werden 2^i Felder der Länge $N/2^i$ zusammengefügt, d.h. maximal $\sim 2^i \times 2 \cdot N/2^i = 2N$ Vergleiche
- bei $\log_2 N$ Ebenen also insgesamt $\sim 2N \log_2 N$ Vergleiche

• Gesamtaufwand (Anzahl der Vergleiche für Teilen und Herrschen)

$$\sim N + \sim 2N \log_2 N = \sim 2N \log_2 N$$

6.5-10 Empirische Analyse

Verdopplungstest

```
java Merge
size      time   ratio
 512      0.00   NaN
1024      0.00   Infinity
2048      0.01   6.00
4096      0.01   1.83
8192      0.03   2.82
16384     0.01   0.19
32768     0.01   1.17
65536     0.07   9.29
131072    0.03   0.46
262144    0.06   2.13
524288    0.14   2.11
1048576   0.30   2.20
2097152   0.65   2.19
4194304   1.24   1.90
8388608   2.46   1.99
```

Bemerkungen zum Speicherplatz

Merge-Sort wie gezeigt ist *kein* in-place-Verfahren: `hilfs` hat im ersten Teilungsschritt Länge N !
Es gibt jedoch auch in-place-Varianten von Merge-Sort.

6.5-11 Warum ist zweifache Rekursion hier so erfolgreich?

Erinnerung

- naive rekursive Berechnung der Fibonacci-Zahlen führt zu exponentiellem Aufwand
- auch dort handelte es sich um eine zweifach-rekursive Methode, aber die Problemgröße wird nur von N auf $N - 1$ bzw. $N - 2$ reduziert

Vergleich

- bei Merge-Sort wird die Problemgröße jeweils halbiert!!
- das führt insgesamt zur Verbesserung

6.5-12 Quicksort

Idee (Teile-und-Herrsche mit anderer Aufteilung)

- **Falls** $N \leq 1$, so ist das Feld sortiert
- **Sonst** wähle einen Index v und organisiere das Feld so in zwei Bereiche, dass Werte $\leq f[v]$ *links*, die anderen *rechts* von diesem Wert landen. Das erreicht man durch Platztausch „um diesen Wert herum“ ($f[v]$ heißt **Pivot** — engl./frz. für “Achse”)
- sortiere rekursiv beide Teilbereiche, füge sie zu sortierem Feld zusammen

Beispiel

Eine naive Implementation würde vielleicht Queues für die Teilbereiche anlegen. Wählt man z.B. jeweils $v = \text{letzter Index im Feld}$, so entstünde eine Anordnung wie im Bild

1	4	2	7	8	3	6	5
1	4	2	3	5	7	6	8
1	2	3	4	5	7	6	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

- Endplatz des Pivots (gefunden durch Zwischensetzen im vorherigen Teile-Schritt)
- Endplatz des Elements (gefunden durch Rekursionsende)
- im weiteren Verlauf beibehaltene Position

Bemerkungen

6.5-13

Wie wird das Pivot bestimmt?

Zum Beispiel diese Strategien sind möglich:

- *Pivotindex = größter Index im Teilstück* (wie in der Grafik oben)
- *Pivotindex zufällig wählen* oder
- komplizierter: *Median of Three* usw.

Wir benutzen im weiteren die Strategie *Pivotindex = kleinster Index im Teilstück* (also anders als in der Grafik oben!)

Beobachtung

- Aufteilungsschritt ist kompliziert, während er bei Merge-Sort trivial ist
- **Besonderheit** keine Hilfsfelder!: Elemente werden “um Pivot herum” getauscht
- **Folgerung** das Zusammenfügen ist trivial: $f[v]$ steht nach dem Aufteilen richtig!

Realisierung

6.5-14

sort(. .) rekursiv organisieren wie bei Merge

- `sort(f)` ruft `sort(f, 0, f.length-1)` auf
- `sort(f, lo, hi)` sortiert im Bereich $lo \leq i < hi$

Zerlegen

- `sort(f, lo, hi)` ruft `partition(f, lo, hi)` auf
- `partition(f, lo, hi)` erledigt das “um das Pivot herum tauschen”, dabei wird auch das Pivot an seinen finalen Platz bewegt
- `partition(. .)` liefert den finalen Index des Pivots als Wert

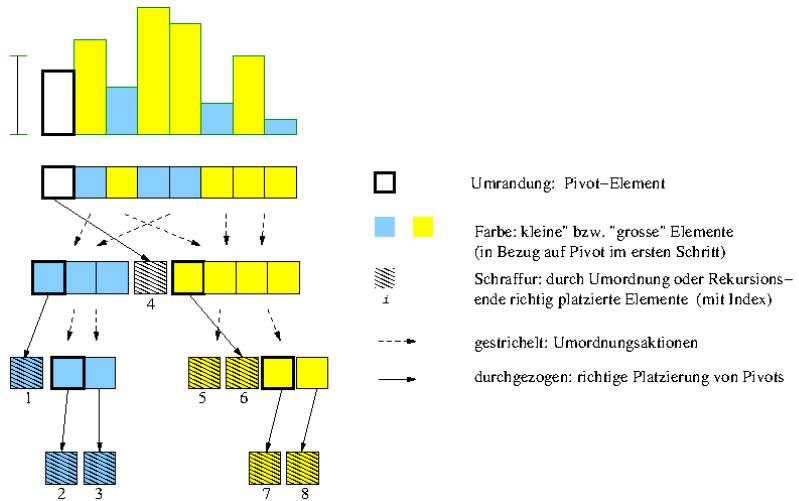
6.5-15 Der algorithmische Kern

```

1  public static void sort(double[] f, int lo, int hi) {
2      if (hi <= lo) return;
3      int j = partition(f, lo, hi);
4      sort(f, lo, j-1);
5      sort(f, j+1, hi);
6  }
7  public static int partition(double[] f, int lo, int hi) {
8      int i = lo, j = hi+1;
9      double v = f[lo], tmp;
10
11     while (true) {
12         while (f[++i] < v)
13             if (i == hi) break;
14         while (v < f[--j])
15             if (j == lo) break;
16         if (i >= j) break;
17         tmp = f[i]; f[i] = f[j]; f[j] = tmp;
18     }
19     tmp = f[lo]; f[lo] = f[j]; f[j] = tmp;
20     return j;
21 }
```

<http://www.stud.informatik.uni-goettingen.de/info1/java/Quick.java>

6.5-16 Veranschaulichung



6.5-17 Analyse

• best-case:

- nehmen wir an, dass das Pivot immer der *Median* ist (d.h., die eine Hälfte der Werte ist kleiner, die andere größer als $f[v]$), dann entsteht ein perfekt balancierter Rekursionsbaum wie bei Merge-Sort und die Laufzeit ist $\sim N \log_2 N$

• average-case:

- im Mittel werden die Teilstücke nicht gleich groß sein, sondern das eine vielleicht doppelt so groß wie das andere (d.h. wir haben eine 1/3, 2/3-Aufteilung des Feldes)
- dann sind $\sim N \log_{2/3} N \approx 1.7N \log_2 N$ Vergleiche zu erwarten

- worst-case:
 - entsteht bei absolut unbalancierter Aufteilung, z.B. wenn das Pivot immer den kleinsten Wert im Feld hat!
 - das passiert genau dann, wenn das Feld bereits aufsteigend sortiert ist(!), in dem Fall gibt es $\sim N^2/2$ Vergleiche

Empirisch

6.5-18

```
java Quick
  size   time ratio
  512    0.00 Infinity
 1024    0.00 0.00
 2048    0.00 Infinity
 4096    0.00 1.50
 8192    0.00 0.67
16384    0.00 2.00
32768    0.01 2.00
65536    0.01 1.00
131072   0.02 1.88
262144   0.03 2.13
524288   0.07 2.06
1048576   0.14 2.11
2097152   0.29 2.08
4194304   0.60 2.08
8388608   1.23 2.05
```

Trotz ähnlicher Analyse sind die absoluten Zeiten besser als bei Merge-Sort: Kosten für Anlegen/initialisieren der Hilfsfelder entfallen

6.5.3 Vergleichsbasierte vs. allgemeinere Sortierverfahren

Eine untere Schranke

6.5-19

Vergleichsbasiertes Sortieren

- obige Verfahren werden durch Anweisungen der Form `if (f[i] < f[j])` gesteuert, d.h. hängen NUR von *Vergleichen der Feldelemente untereinander* ab, nicht von den Werten
- solche Verfahren heißen **vergleichsbasiert**

Beispiele (nicht erlaubte Vergleiche)

- `if (f[i] > 1000) ...`
- `if (f[i] % 10 == 2) ...`

Satz

Jedes vergleichsbasierte Sortierverfahren erfordert im schlechtesten Fall wenigstens $\sim N \log_2 N$ Vergleiche, zum Sortieren von N Zahlen.

Beweis: baumartiger Ablauf
 (nicht klausurrelevant)

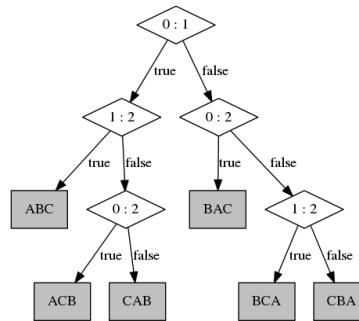
6.5-20

- an der Wurzel starten beliebige Eingaben: alle Permutationen von $\{1, 2, \dots, N\}$

- jeder Knoten steht für einen Vergleich: \ ein Teil der Inputs wandert nach links, der andere nach rechts
- an den Blättern stehen Permutationen, die dem entsprechenden Weg gefolgt sind
- da es genau $N!$ verschiedene Permutationen gibt, muss wenigstens einer der Wege die Länge $\log_2 N! \sim N \log N$ haben

Beispiel

i : j steht für if (a[i] < a[j]) ...



6.5-21 Radix-Sort: Ein nicht-vergleichsbasiertes Verfahren

(nicht klausurrelevant)

Voraussetzungen

- Werte mit bekannter Stellenzahl N (z.B. dezimale int-Werte)
- vereinfachende Annahme: alle $a[i] \geq 0$

Ablauf

- Phasen $j = 0, 1, \dots, K - 1$, jede besteht aus *Partitionieren* und *Sammeln* und bezieht sich auf die j -te Ziffer von rechts!!
- benutze anfangs leere Queue q
- Phase j :

```

// Partitionieren in Abschnitte je nach Ziffer in Position j
for (int z = 0; z < 10; z++) {
    for (int i = 0; i < N; i++) {
        if (ziffer(j, f[i]) == z)
            q.enqueue(f[i]);
    }
}
// Sammeln: Q leeren und in f speichern:
for (int i = 0; i < N; i++)
    f[i] = q.dequeue();
  
```

Beispiel: [124, 523, 483, 128, 923, 584]

Beispiel von Wikipedia $K = 3 \Rightarrow$ 3-stellige (positive) Zahlen sortieren

- Phase 0:
 - Partitionieren: <523, 483, 923, 124, 584, 128<
 - im Feld sammeln: [523, 483, 923, 124, 584, 128]
- Phase 1:
 - Partitionieren: <523, 923, 124, 128, 483, 584<
 - im Feld sammeln: [523, 923, 124, 128, 483, 584]
- Phase 2:
 - Partitionieren: <124, 128, 483, 523, 584, 923<
 - im Feld Sammeln: [124, 128, 483, 523, 584, 923]

Java-Code

6.5-22

```

1 public class Radix {
2     static int K = 5; // Stellenzahl
3     public static void sort(int[] f) {
4         int N = f.length;
5         Queue<Integer> q = new EVLQueue<Integer>();
6         for (int j = 0; j < K; j++) {
7             for (int z = 0; z < 10; z++)
8                 for (int i = 0; i < N; i++)
9                     if (ziffer(j, f[i]) == z)
10                         q.enqueue(f[i]); // Auto-Boxing (in Integer-Objekt ..)
11                 for (int i = 0; i < N; i++) // .. "einwickeln"
12                     f[i] = q.dequeue(); // Auto-Unboxing ("auswickeln")
13             }
14         }
15         private static int ziffer(int j, int n) {
16             for (int k = 0; k < j; k++)
17                 n /= 10;
18             return n % 10;
19         }
20     }

```

<http://www.stud.informatik.uni-goettingen.de/inf01/java/Sorter/Radix.java>

Analyse

6.5-23

Laufzeit & Speicher

- Kernoperation: $V(N)$ = Anzahl der en-/ dequeue (...) -Operationen
- für jeden Wert offenbar genau K Kernoperationen, also $V(N) = K \cdot N$ (best = worst = average case!)
- Laufzeiten hängen stark von Queue-Implementation ab
- Speicherbedarf zusätzlich $\sim N$ für die Queue \Rightarrow kein in-place Verfahren

Stabilität

Stabile Verfahren behalten die relative Anordnung von Daten bei, die nach dem Sortierkriterium ununterscheidbar sind.

Beispiel: Alphabetisch kommt Anton vor Zwillingsschwester Berta. Dann Caesar, Dora, ... Wird diese Anordnung als Eingabe für eine *stabile* Sortierung nach Alter verwendet, so bleibt Anton vor der gleichaltrigen Berta.

6.5-24 Vergleich der Sortierverfahren (Wachstumsordnung)

Verfahren	Idee	in-place	stabil	best	worst	average
BubbleSort	Vergl.-b.	ja	ja	N	N^2	N^2
SelectionSort	Vergl.-b.	ja	nein	N^2	N^2	N^2
InsertionSort	Vergl.-b.	ja	ja	N	N^2	N^2
MergeSort	Vergl.-b.	nein	ja	$N \log N$	$N \log N$	$N \log N$
QuickSort	Vergl.-b.	ja	nein	$N \log N$	N^2	$N \log N$
RadixSort	Wert-b.	nein	ja	$K \cdot N$	$K \cdot N$	$K \cdot N$

Bemerkung

Es gibt auch in-place-Varianten von Mergesort.

6.5.4 Nachtrag zur O-Notation

6.5-25 Die O-Notation

Man findet häufig Aussagen der Form "die Komplexität ist $O(g(N))$ ".

Beispiel: Die Laufzeitkomplexität von Mergesort beträgt $O(N \log N)$, die Speicherkomplexität $O(N)$ (sinngemäß bei Wikipedia zu lesen).

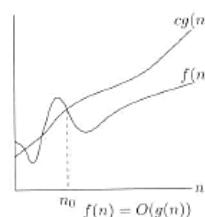
Erklärung

- *Komplexität* eines Algorithmus = (z.B. Laufzeit- oder Speicher-)Aufwand hängt von der Problemgröße N ab \Rightarrow prinzipiell ist es eine Funktion $N \mapsto f(N)$, die oft gar nicht detailgenau bestimmbar ist .
- „Komplexität in $O(g(N))$ “ heißt „ $f(N)$ wächst höchstens so schnell wie $g(N)$ “ Das heißt, die Schreibweise drückt eine obere Schranke aus. Formal wird das durch diese Definition ausgedrückt:

Definition

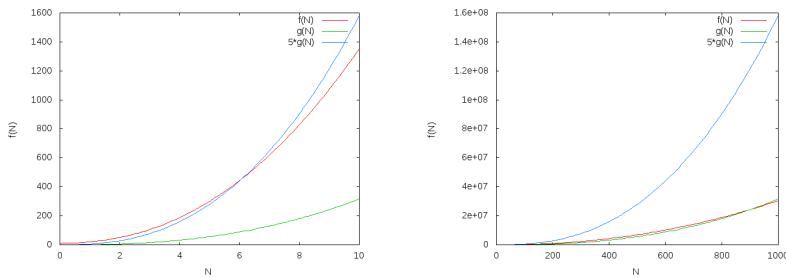
Grafik aus Cormen, Leiserson, Rivest, Stein:
Algorithmen — eine Einführung

$O(g(N))$ ist die Menge der Funktionen $f : \mathbb{N} \rightarrow \mathbb{N}$, für die es eine Mindestzahl N_0 und einen konstanten Vorfaktor c gibt, so dass für $N > N_0$ gilt: $f(N) \leq c \cdot g(N)$.



6.5-26 Beispiel

$$f(N) = \log_2(2N+1) \cdot (2N+1)^2 + 10 \text{ und } g(N) = \sqrt{N^5}$$



- die (falsche) Vermutung $g(N) \leq f(N)$ für große N ist u.U. experimentell nicht zu widerlegen
- $f(N) \in O(g(N))$ ist zu schwach für Laufzeitvorhersagen: Laufzeit ist vllt. viel geringer, als der für die Abschätzung benötigte Vorfaktor c (hier 5) suggeriert.
- dagegen ist offenbar $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 0$

6.5.5 Suchverfahren

Allgemeine Problemstellung

6.5-27

gegeben Suchwert q , Feld `int [] f`

gesucht Index i mit $f[i] == q$ (oder Sonderwert bei Misserfolg)

Kernoperation: Vergleiche mit q

Beispiel: Sequentielle Suche (Wiederholung)

```
int seqsearch(int[] f, int q) { // V(N)
    int N = f.length;
    for (int i=0; i < N; i++) {
        if (f[i] == q) // <= N (datenabhaengig)
            return i;
        return -1;
    }
}
```

	Tilde
best case	1
worst-case	$\sim N$
average-case	$\sim N/2$

Binäre Suche (Wiederholung und Ausbau)

6.5-28

gegeben Suchwert q , Feld `int [] f, int lo, hi`
(f im Index-Intervall $[lo, hi]$ aufsteigend sortiert)

gesucht Index i mit $f[i] == q$ (oder Sonderwert)

Iterativ

siehe auch Paragraph 2.3-5

```

1   static int bsearch(int q, int[] f, int lo, int hi)
2   {
3       int mid, median;
4       while (lo < hi) {
5           mid = (hi + lo)/2; median = f[mid];
6           if (q < median) hi = mid;
7           else if (median < q) lo = mid+1;
8           else
9               return mid;
10      }
11  }

```

Rekursiv

```

1   static int bsearch(int q, int[] f, int lo, int hi)
2   {
3       if (lo >= hi) return -1;
4       int mid = (hi + lo)/2, median = f[mid];
5       if (q < median) return bsearch(q,f,lo ,mid);
6       else if (q > median) return bsearch(q,f,mid+1,hi);
7       else
8           return mid;
9   }

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/BSearch.java>

6.5-29 Analyse

Beobachtung 1

- bei iterativer Lösung: *in jeder Iteration* eine Median-Bestimmung
- bei rekursiver Lösung: *vor jedem rekursiven Aufruf* eine Median-Bestimmung

Beobachtung 2

Intervallgrenzen werden bei jeder Iteration/jedem rekursiven Aufruf in gleicher Weise angepasst \Rightarrow Anzahl $V(N)$ der Median-Bestimmungen ist bei beiden Lösungen gleich

Satz

$V(N) \leq \lfloor \log_2 N \rfloor + 1$, somit $V(N) \sim \log_2 N$ im worst-case.

Beweis

- stimmt offenbar für $N = 1$
- sonst: $V(N) \leq 1 + V(\lfloor N/2 \rfloor) \leq 1 + \lfloor \log_2 \lfloor N/2 \rfloor \rfloor + 1 = \lfloor \log_2 N \rfloor + 1$

6.5-30 Verwandte Algorithmen und Anwendungen

Beispiel 1: Funktionsumkehr mittels Bisektion

siehe Paragraph 3.4-16ff Beispiel: $\Phi^{-1}(z)$: gesucht Argumentwert z so dass $\Phi(z) = y$

```

1  public static double PhiInverse(double y) {
2      return PhiInverse(y, .00000001, -8, 8);
3  }
4  private static double PhiInverse(double y, double delta, double lo, double hi) {
5      double mid = lo + (hi - lo) / 2;
6      if (hi - lo < delta) return mid;
7      if (Phi(mid) > y) return PhiInverse(y, delta, lo, mid);
8      else                 return PhiInverse(y, delta, mid, hi);
9  }

```

<http://www.stud.informatik.uni-goettingen.de/infol/java/Gaussian.java>

Beispiel 2: Spamfilter

1. implementiere binäre Suche für String-Felder
2. darauf aufbauend: static boolean found(String s, String[] list)
3. white = sortiertes Feld mit Kontakt-Adressen (*whitelist*)
4. black = sortiertes Feld mit bekannten Spam-Adressen (*blacklist*)

⇒ ermöglicht sehr schnelle Routine-Prüfung von E-Mail-Absendern:

```

if (found( mail.sender, white) && !found(mail.sender, black))
    pass(mail, inbox);
else
    pass(mail, junkbox);

```

6.5.6 Binäre Suchbäume

aktuell nicht klausurrelevant