Informatik I

Wintersemester 2005/2006

(c) Prof. Dr. Wolfgang May

Universität Göttingen

may@informatik.uni-goettingen.de

Kapitel 1
Einführung

1.1 Um was geht es hier eigentlich?

INFORMATIK – PROGRAMMIEREN – MATHEMATIK

- Nur ein kleiner Teil der Diplom-Informatiker programmiert später tatsächlich.
- Aber es ist auch für Informatiker nützlich, programmieren zu *können*.
- Informatiker werden aufgrund ihrer Fähigkeit geschätzt, strukturiert denken zu können und Probleme/Aufgaben strukturiert und systematisch analysieren und lösen zu können.
- Informatik = Analyse + Synthese
  Synthese beinhaltet u.a. Programmieren.
- Wer mathematische Beweise führen kann, kann auch reale Informatik-Probleme [nach denselben Prinzipien] lösen.
- Ein Programm, das ein Problem löst ist sehr ähnlich zu einem konstruktiven Beweis.
- Die Zerlegung des Beweises eines mathematischen Satzes in Lemmata (=Hilfssätze) entspricht der Identifizierung und Lösung von Subproblemen.
- Genauso wie es in Mathematik zugrundeliegende Strukturen gibt (siehe Algebra), gibt es in der Informatik grundlegende Verfahren und Konzepte (Algorithmen-Design, theoretische Konzepte).

WAS IST "INFORMATIK"

- Kunstwort, in den frühen 60ern entstanden um eine neue wissenschaftliche Disziplin zu beschreiben:
  - *Information* + *Mathematik*
  - Wissenschaft der *Informationsverarbeitung*, Nähe zur Mathematik.
  - besserer Begriff als "*Computer Science*" im englischen Sprachraum?
- Informatik-Duden (1988): "Informatik ist die Wissenschaft von der *systematischen* Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Computern".
- Studien- und Forschungsführer Informatik (Hrsg.: Ges. f. Informatik; vor 1990): "Informatik ist die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen"
- Association of Computing Machinery (ACM; CACM 1986):
  "Computer Science is the systematic study of algorithms and data structures; specifically
  1. their formal properties
  2. their mechanical and linguistic realizations, and
  3. their applications."

Unterteilung

Technische Informatik: innere Stuktur und Bau von Computern

- Elementare Bauteile ... Halbleiterphysik
- Schaltkreise ... Logik, Mikro-Elektronik
- Bauteile, Hardware-Module → Rechnerarchitektur
- hardwarenahe Protokolle, Mikroprogrammierung

Praktische Informatik: Prinzipien und Techniken der Programmierung und Realisierung

- Betriebssysteme/(Software)systemarchitektur
- Softwaretechnik: Analyse, Entwurf, Realisierung
- Programmiersprachen/Compilerbau
- Algorithmen und Datenstrukturen (Theorie)
  Strategien, Aufwandsanalyse ...
- Netzwerke, Telematik: Kommunikation
- Datenbanken: persistente Speicherung großer Datenmengen
- Robotik
- Bildverarbeitung, Multimedia

Unterteilung (Forts.)

Angewandte Informatik: Brücke von den Computerwissenschaften im engeren Sinne zu den Problemen der realen Welt.

- Wirtschaftsinformatik
- Medizinische Informatik
- Bioinformatik

Theoretische Informatik: mathematische Modelle, zugrundeliegende theoretische Konzepte, einschließlich Hilfsmitteln zu ihrer Beschreibung.

- Formale Logik
- Spezifikation, Verifikation, Künstliche Intelligenz
- Formale Sprachen, Grammatiken ...
- Berechenbarkeit, Komplexität

Definitionen und Einordnung bestimmter Teilgebiete oft umstritten, fließende Übergänge.

Hat "Informatik" immer unbedingt mit Computer/Programmieren zu tun?

- Nein.
- Berechenbarkeit/Komplexitätstheorie rein abstrakt
- Algorithmen sind nicht notwendigerweise an Computer gebunden
  (Suche, Labyrinth, Wechselgeld, Schach ...)
- es geht zunehmend um den Umgang mit Information:
  - Organisation von verteilten (und verteilt entstehenden) Informationen
  - Workflows in Unternehmen
  - Content-Providing: Aufbereitung, "Ergonomie" von Web-Angeboten, "Semantic Web"
  - z.B. Teleteaching: zusätzliche Interaktionsmechanismen
  - Sicherheitsaspekte, kommerzielle Aspekte, rechtliche Aspekte ...
- der Computer ist oft nur noch das Mittel zum Zweck (computer*gestütztes* Arbeiten).

⇒ Programmieren/Rechneradministration ist nur ein kleiner (aber zentraler) Teil möglicher Berufsbilder für Informatiker.

Angewandte Informatik

- Hineindenken in die Begriffswelt des Anwenders
  - Verstehen, Auffassungsgabe, Vorstellungsvermögen
- Schaffen einer gemeinsamen Kommunikationsbasis
  - Modellierung, Formalisierung der Anwendungswelt
- Entwickeln eines Lösungsansatzes
  - Analyse und Klassifizierung des Problems
  - Ausarbeitung einer Lösung auf abstrakter Ebene
  - Wahl geeigneter Algorithmen, Datenstrukturen, ggf. zugekaufte Software
- Realisierung, u.a. Programmierung,
  dabei Validierung (Testen, prüfen ob das Produkt das ist, was man haben will).

Informatik I – Grundlagen

- Algorithmus-Begriff
- Algorithmen und Datenstrukturen
  - Synthese: Algorithmenprinzipien
  - Analyse: Laufzeitbedarf, Speicherbedarf
  - Verifikation
- Programmiersprachen
  - Syntax/Grammatik
  - Semantik
- “Handwerk”: Java/UNIX

Unix-Einführung

... siehe Web-Seiten

1.2 Rechenmaschinen und Computer

(aus *Informatik-Duden*)

Die Frühzeit

- Altertum (China): Abakus (Brett mit verschiebbaren Kugeln) für Grundrechenarten. 1-5-10-50-Codierung von Zahlen, geeignete Rechenalgorithmen.
- 3. Jhdt. v. Chr.: Euklids Algorithmus zur Berechnung des ggT zweier Zahlen.
- 9. Jhdt. n. Chr.: der persisch-arabische Mathematiker und Astronom *Ibn Musa Al-Chwarismi* schreibt das Lehrbuch *“Kitab al jabr w’almuqabala”* (*“Regeln zur Wiedereinsetzung und Reduktion”*). Das Wort “Algorithmus” geht auf seinen Namen zurück.

Rechnen in Europa und Mechanische Rechenmaschinen

- 1547: Adam Riese (1492-1559) veröffentlicht ein Rechenbuch, in dem er die Rechengesetze des aus Indien stammenden Dezimalsystems (5. Jhdt. n. Chr.) beschreibt. Im 17. Jhdt. setzt sich das Dezimalsystem in Europa durch - damit ist eine Automatisierung des Rechenvorgangs möglich.
- 1623: Wilhelm Schickard (1592-1635) konstruiert für Kepler (1571-1630) eine Maschine, die addieren, subtrahieren, multiplizieren und dividieren kann (bleibt aber unbeachtet).
- 1641: Blaise Pascal (1623-1662) konstruiert eine Maschine, mit der man sechsstellige Zahlen addieren kann.
- 1674: Gottfried Wilhelm Leibniz (1646-1716) konstruiert eine Rechenmaschine mit Staffelwalzen für die vier Grundrechenarten. Er befasst sich auch mit der binären Darstellung von Zahlen (1703).
- 1774: Philipp Matthäus Hahn (1739-1790) entwickelt eine mechanische Rechenmaschine, die erstmals zuverlässig arbeitet.
- Ab 1818: Rechenmaschinen nach Vorbild der Leibnizschen Maschine werden serienmäßig hergestellt und weiterentwickelt.

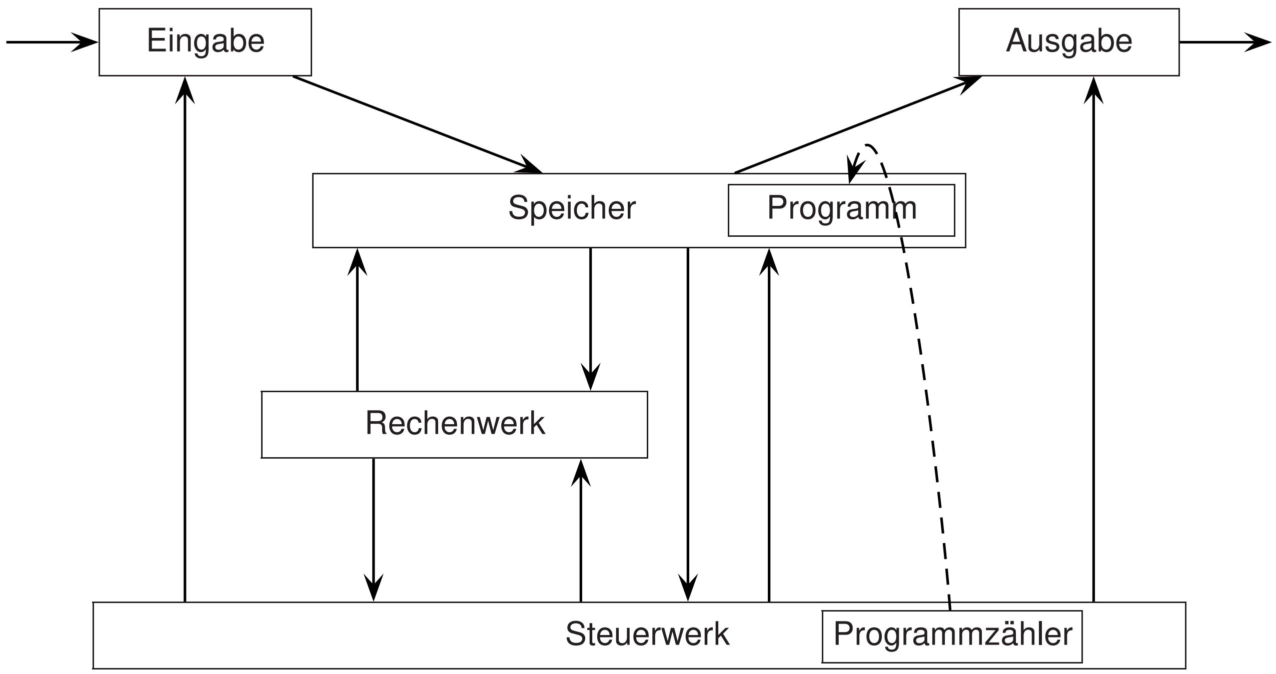
Programmierung und elektrische Rechenmaschinen

- 1833: Charles Babbage (1792-1871): *Difference Engine*. 1838 Plan für die *Analytical Engine*, bei der die Reihenfolge der einzelnen Rechenoperationen durch nacheinander eingegebene Lochkarten gesteuert wird.
- 1886: Hermann Hollerith (1860-1929) entwickelt in den USA elektrisch arbeitende Zählmaschinen für Lochkarten, mit denen die statistischen Auswertungen der Volkszählungen vorgenommen werden. Diese Firma hieß später IBM ...
- 1934: Konrad Zuse (1910-1995) beginnt mit der Planung einer programmgesteuerten Rechenmaschine. Sie verwendet das binäre Zahlensystem und die halblogarithmische Zahlendarstellung. Die Z1 wird 1937 fertig.
- 1941: die elektromechanische Z3 ist der erste funktionsfähige programmgesteuerte Rechenautomat. Das Programm wird über Lochstreifen eingegeben. Die Anlage verfügt über 2000 Relais und eine Speicherkapazität von 64 Worten à 22 Bit. Multiplikationsdauer: etwa 3s.
- März 1945 Vorstellung der elektromechanischen Z4 von Zuse in Göttingen.
- Sommer 1947 Treffen von deutschen Rechenmaschinenexperten (u.a. Zuse, Schreyer, Walther, Billing) in Göttingen, organisiert von britischen Fachleuten (u.a. Turing, Womersley) (interessanter Überblick: `http://www.susas.de/com_daten.htm`).

Der Weg zum Computer

- 1944: Howard H. Aiken (1900-1973) in Zusammenarbeit mit der Harvard-University und IBM: teilweise programmgesteuerte Rechenanlage MARK I. Additionszeit 1/3s, Multiplikationszeit 6s.
- 1946 J. P. Eckert und J. W. Mauchly: ENIAC (Electronic Numerical Integrator and Automatic Calculator) als erster voll elektronischer Rechner (18000 Elektronenröhren). Multiplikationszeit: 3s.
- 1946-1952: Entwicklung weiterer Computer auf der Grundlage der Ideen John v. Neumanns (1903-1957 Univ. Princeton) (Einzelprozessor, Programm und Daten im gleichen Speicher; *"von-Neumann-Rechner"*).
- 1949 M. V. Wihls (Univ. Manchester): EDSAC (Electronic Delay Storage Automatic Calculator) als erster universeller Digitalrechner (gespeichertes Programm).
- ab 1950: Industrielle Rechnerentwicklung und Produktion.
- 1952: G1, 1954; G2 (Göttingen); 1953: IBM 650;
  Beginn industrieller Produktion in D (u.a. IBM Sindelfingen, Zuse Z11 in Hünfeld)
- um 1957: 6 Rechner in D: G1, G2 (Gö), PERM (München), 3x IBM 650.

ABSTRAKTES ARCHITEKTURPRINZIP NACH *von Neumann*



ABSTRAKTES ARCHITEKTURPRINZIP NACH *von Neumann*

- Speicher mit einzeln adressierbaren Speicherzellen
  - Programm
  - Daten
- Rechenwerk (Operationen z.B. +/-/shift)
- Steuerwerk (Programmzähler etc)
  - Befehle der Reihe nach aus dem Speicher holen, decodieren, ausführen, ggf. Resultate im Speicher ablegen,
  - Befehle: Übertragung von Daten zwischen Speicherzellen, Tests/Verzweigungen, Sprünge, Arithmetik,
- Eingabe-/Ausgabeeinheiten.
- Im weiteren Verlauf nimmt man einen solchen Rechner als gegeben an.
- Maschinenprogramme/Assembler setzen direkt auf dieser Architektur auf.
- Höhere Programmiersprachen bieten intuitivere Befehle an.

... mehr dazu in Informatik-II und Informatik-IV.

1.3 Vom Problem zum Algorithmus

- Gegeben ist ein Problem
- Gesucht wird ein Lösungsweg (Algorithmus) ...
  der dann als Programm codiert wird

Intuitiver Algorithmusbegriff

- Handlungsanweisungen
  (Spielregeln, Kochrezepte, Gebrauchsanweisungen)
- Unterscheidung zwischen einem Ausführenden (“Prozessor”) und dem Vorgang selbst (“Prozess”)
- - sequentielle A. (z.B. Wegbeschreibung)
  - nebenläufige A. – Koordinationspunkte (z.B. Kochrezept)
  - regelbasierte A. (z.B. Spielregeln, Gebrauchsanweisungen)
- Verschiedene Abstraktionsebenen
  - abstrakt als Teilaufgabe: “Sortiere ... der Größe nach”
  - genauere Spezifikation: Sortierverfahren im Detail

ALGORITHMEN: BEISPIELE

- Suchen nach einem Briefkasten in einer fremden Stadt
- Suchen nach einem Eintrag im Telefonbuch
- Labyrinth: Suche nach einem Ausgang (oder äquivalent: nach einer Person im Labyrinth)
- Bezahlen eines gegebenen Betrages/Herausgeben von Wechselgeld
- Sortieren von Klausuren nach Matrikelnummern
- Suchen des kürzesten/schnellsten Weges von Freiburg nach Göttingen

$\Rightarrow$ alles keine typischen Computerprobleme.

- schriftliches Addieren und Multiplizieren
- Berechnung des ggT (größter gemeinsamer Teiler)
- Berechnung der Fakultät einer Zahl ($n! = n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$)

Aufgabe: Beschreiben Sie die Lösungswege dieser Probleme in natürlicher Sprache.

PROBLEMANALYSE UND - BESCHREIBUNG – DESIGN

- Wie beschreibt man ein Problem?
  - Textuelle Beschreibung
  - *Modellierung* der relevanten Objekte, ihrer Eigenschaften, Beziehungen und möglichen Aktionen
- Wie beschreibt man einen Algorithmus (abstrakt!)?
  - Zusammenwirken der relevanten Objekte
  - Aktionen der einzelnen Objekte
- Welche Eigenschaften soll ein Algorithmus haben?
  - endlich beschreibbar (durch ein Programm oder eine Menge von Regeln)
  - Verfahren sollte irgendwann enden (*“terminieren”*)
  - Verfahren sollte erfolgreich enden (*“Korrektheit”*)
  - Verfahren sollte möglichst schnell beendet sein (*“Effizienz”*)

1.4 Beschreibung von Algorithmen – Programmiersprachen

Ein Algorithmus kann – um ihn einem bestimmten Computer mitzuteilen – in einer Programmiersprache beschrieben ("codiert") werden.

- Programmiersprachen sind Sprachen ...
  ... um Computer zu programmieren:
  - *Syntax*: Zeichensatz, Worte, "Grammatik" um zulässige "Sätze" (Programme) zu bilden
  - *Semantik*: Was bedeutet ein Satz/Programm?

Formale Algorithmenmodelle

Turing-Maschine (Alan Turing, 1936)

$TM = (Q, \Sigma, q_0, q_H, \delta : (Q \times \Sigma \to Q \times \Sigma \times \{L, N, R\}))$

- Band (Programm + Daten) bestehend aus Zellen, beschrieben mit Zeichen aus einem Alphabet Σ sowie ein Zeichen B ("Blank").
- interner Zustand $q \in Q$, Anfangszustand $q_0 \in Q$
- Lesekopf: läuft über das Band, liest den darunterliegenden Wert $x \in \Sigma$ und führt in Abhängigkeit von x und q eine Aktion aus (neuer interner Zustand, Schreiben eines Wertes, Bewegung nach links oder rechts). Verhalten wird durch δ gegeben.
- Wenn sie irgendwann stehenbleibt, muss der Zustand q_H erreicht sein.

Beispiel: $Q = \{q_0, q_1, q_2, q_H\}$, $\Sigma = \{1\}$, δ gegeben durch

$$\begin{array}{lll} (q_0, B) \to (q_0, B, R) & (q_1, B) \to (q_0, B, R) & (q_2, B) \to (q_H, B, N) \\ (q_0, 1) \to (q_1, 1, R) & (q_1, 1) \to (q_2, 1, R) & (q_2, 1) \to (q_2, 1, R) \end{array}$$

bleibt stehen, wenn sie "11B" findet.

Turing-Maschinen: Beispiele und Aufgaben

Zahlen $n \in \mathbb{N}$ kann man z.B. durch eine Folge von n Einsen codieren.

- Gegeben sein ein Band mit n Einsen, einem B, und m Einsen:
  $\underbrace{1\ \dots\ 1}_{n}B\underbrace{1\ \dots\ 1}_{m}$.
  Geben Sie eine TM an, die $n+m$ berechnet.
- Dasselbe mit einer beliebig langen Folge von Bs anstelle einem einzigen.
- Geben Sie eine TM an, die für eine gegebene Zahl n die Zahl $2n$ berechnet.
- Geben Sie eine TM an, die für eine gegebene Zahl n die Zahl $n/2$ berechnet.

Vorgehensweise

- Geeignete Codierung des Problems auf dem Band (z.B. Zahl n durch n Einsen),
- Ablauf grob überlegen ... was/wann/wie,
- in Teil- und Einzelschritte zerlegen und als Zustände codieren,
- Geeignete Erweiterung des Alphabets, um den Ablauf zu steuern (Markierungen etc.).

Turing-Maschine: Lösung des Beispiels $n+m$

Idee: schreibe eine “1” in den Zwischenraum und lösche dafür die letzte “1”.
$TM = (\{q_0, q_1, q_2, q_3, q_H\}, \{1\}, q_0, q_H, \delta)$ mit Transitionsfunktion δ wie folgt:

$(q_0, 1) \to (q_1, 1, R)$	q_1: laufe in der ersten Zahl nach rechts
$(q_1, 1) \to (q_1, 1, R)$	
$(q_1, B) \to (q_2, 1, R)$	Wechsel in die zweite Zahl, schreibe eine “1”
$(q_2, 1) \to (q_2, 1, R)$	q_2: laufe in der zweiten Zahl nach rechts
$(q_2, B) \to (q_3, B, L)$	Hinter dem Ende der 2. Zahl ein Zeichen zurück
$(q_3, 1) \to (q_H, B, N)$	“1” durch “B” ersetzen und Ende

Weitere Modelle

- Registermaschine:
  idealisiertes Modell eines (von-Neumann-)Rechners. Direkt adressierbarer Hauptspeicher, ein “Akkumulatorregister” (Rechenregister), mehrere Speicherregister. Sequentielles Programm (LOAD/STORE) mit (bedingten) Sprüngen (GOTO, IF Vergleich GOTO) in getrenntem Speicher, Programmzähler.
- Lambda-Kalkül: Alonzo Church, 1930er (Grundlage für die spätere Programmiersprache LISP)
- primitive und μ-rekursive Funktionen (Gödel, Kleene, ca. 1930)
- Markov-Algorithmen (1954)
- “while”-Pseudocode als “einfachste” höhere Programmiersprache (ca. 1950-60): Variablenzuweisung, “;”, begin ... end, if ... then, while ... do.

Church’sche These/Turing-Church-These (A.Church, 1936)

“Der Begriff der *intuitiv berechenbaren* Funktionen stimmt mit der Klasse der berechenbaren [= Turing-berechenbaren] Funktionen überein.”

- intuitiv berechenbar = “man kann eine Arbeitsbeschreibung angeben”
- Ein Algorithmenmodell heißt *universell*, wenn man damit alle berechenbaren Funktionen beschreiben kann
- die oben beschriebenen Modelle sind gleichwertig und universell
- die meisten Programmiersprachen sind universell
- einige Programmiersprachen für Spezialanwendungen, z.B. SQL (eine Sprache für Datenbankanfragen) sind nicht universell

TURING-MASCHINE: NOCH EIN BEISPIEL

Die hawaiianische Sprache kennt nur die folgenden Buchstaben:

- die Vokale a, e, i, o, u und die Konsonanten h, k, l, m, n, p, w

Ein Wort beginnt mit einem Konsonanten oder einem Vokal. Auf einen Konsonanten muss mindestens ein Vokal folgen, es können beliebig viele Vokale aufeinanderfolgen. Konsonanten dürfen nicht am Ende eines Wortes stehen. Ein Wort hat mindestens einen Buchstaben.

- Gesucht wird eine Turing-Maschine, die diese Sprache "erkennt", d.h. in einem akzeptierenden Zustand stehenbleibt, falls auf dem Band ein "erlaubtes" Wort steht.

$Q = \{q_0, q_v, q_k, q_\perp, q_H\}$, $\Sigma = \{a, e, i, o, u, h, k, l, m, n, p, w\}$, δ gegeben durch

$$\begin{array}{llll}
(q_0, v \in Vok) & \to (q_v, v, R) & (q_v, v \in Vok) & \to (q_v, v, R) \\
(q_0, k \in Kons) & \to (q_k, k, R) & (q_v, k \in Kons) & \to (q_k, k, R)
\end{array}$$

$$\begin{array}{llll}
(q_k, v \in Vok) & \to (q_v, v, R) & (q_v, B) & \to (q_H, B, N) \\
(q_k, k \in Kons) & \to (q_\perp, k, R) & (q_k, B) & \to (q_\perp, B, H) \\
(q_0, B) & \to (q_\perp, B, H) & & \\
(q_\perp, x \in \Sigma) & \to (q_\perp, x, R) & (q_\perp, B) & \to (q_\perp, B, H)
\end{array}$$

TURING-MASCHINE: KOMMENTARE ZUM BEISPIEL

Diese TM hat einige Besonderheiten:

- läuft immer nur nach rechts: liese das Eingabewort einmal
- verändert das Band nicht
- effektiv: besteht nur darin, den internen Zustand zu verändern!

ENDLICHE AUTOMATEN

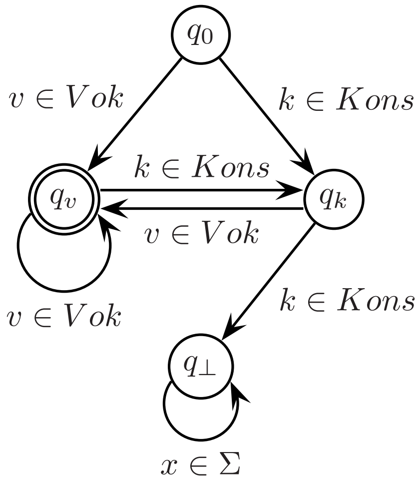
- Ein endlicher Automat liest ein Eingabewort und testet ob es "akzeptiert" wird.
- $M = (Q, \Sigma, q_0, F, \delta)$, wobei
  - F jetzt eine Menge von akzeptierenden Zuständen sein kann
  - Transitionsfunktion: $\delta : Q \times \Sigma \to Q$
  - kann graphisch angegeben werden

ENDLICHE AUTOMATEN: BEISPIEL

... endlicher Automat zur Erkennung der hawaiianischen Sprache:

$Q = \{q_0, q_v, q_k, q_\perp, q_H\}$, $\Sigma = \{a, e, i, o, u, h, k, l, m, n, p, w\}$, $F = \{q_v\}$, δ gegeben durch

$$\begin{array}{ll}
(q_0, v \in Vok) & \to q_v \\
(q_0, k \in Kons) & \to q_k \\
(q_v, v \in Vok) & \to q_v \\
(q_v, k \in Kons) & \to q_k \\
(q_k, v \in Vok) & \to q_v \\
(q_k, k \in Kons) & \to q_\perp \\
(q_\perp, x \in \Sigma) & \to q_\perp
\end{array}$$



Der Automat bleibt in einem Zustand $\in F$ stehen, wenn das Eingabewort in der Sprache erhalten ist.
(Hinweis: hier gibt es keine Blanks, der Automat bleibt einfach stehen. Wenn die Eingabe das leere Wort ist, bleibt er in q_0, also auch in einem nicht akzeptierenden Zustand)

ZUSAMMENFASSUNG UND AUSBLICK

- Turingmaschine: formales Berechnungsmodell
  - kann *jeden* Algorithmus berechnen
  - unter anderem eben auch Sprachen “erkennen”
  - keine direkte praktische Relevanz
  - in der Komplexitätstheorie verwendet
- Endliche Automaten
  - Erkennung “sehr einfacher” Sprachen
  - siehe Beispiel
  - schon einfachste Programmiersprachen sind zu komplex
    (man kann keine Klammerstrukturen überprüfen)
  - dazu später mehr unter “(formale) Grammatiken”
  - werden (zusammen mit erweiterten Formen – Kellerautomaten) z.B. bei der Implementierung von Programmiersprachen verwendet
    siehe u.a. Informatik-II
  - man kann aber auch viele allgemeine Prozesse in Form eines endlichen Automaten darstellen (z.B. Protokolle zum Verbindungsaufbau eines Modems)

Programmiersprachen: Paradigmen und Entwicklung

- 40er: hardwarenahe Programmierung: Maschinensprache, Assembler
- 50er/60er: erste Entwicklung höherer, *imperativer* Programmiersprachen: FORTRAN, ALGOL, COBOL, BASIC; typische Sprachkonstrukte:
  - Komposition: erst A dann B
  - Selektion/Verzweigung: Wenn Bedingung dann A, sonst B
  - Iteration/Schleifen: Solange Bedingung tue A
  - Variablenkonzept: setze Variable x auf v, lese Variable y
  - Sprünge: Gehe zu ...
- 70er *Strukturiertes* Programmieren: Pascal, C:
  prozedurale Strukturkonzepte für *imperative* Sprachen
- 70er/80er: *modulare* Programmiersprachen: Modula, Ada
- Funktionale Programmiersprachen: LISP (60er), Haskell, Scheme
- Deklarative Programmiersprachen (Was? anstatt Wie?):
  Prolog (PROgrammieren in LOGik), SQL (Datenbankanfragen)
- Objektorientierte Programmierung:
  Simula (1967/70), Smalltalk-76, C++(1985), Eiffel (1988), Java (1995)

Kann man "alles" programmieren? – Nein

- Die Menge der Algorithmen ist abzählbar
  Jeder Algorithmus ist durch einen endlichen "Text" darstellbar.
- Es gibt überabzählbar viele Funktionen mit Argumenten und Ergebnissen in $\mathbb{N}$.

  Beweis: ein "Cantor'scher Diagonalschluss"

  Sei $f_1, f_2, f_3, \ldots$ eine Aufzählung aller Funktionen von $\mathbb{N}$ nach $\mathbb{N}$. Definiere eine neue Funktion $f : \mathbb{N} \to \mathbb{N}$ durch

  $$f(i) = f_i(i) + 1$$

  f kommt in der o.g. Aufzählung nicht vor.

DAS “HALTEPROBLEM”

Das folgende Problem ist mit Rechnern nicht lösbar:

> *Sei P ein beliebiges Programm, i eine beliebige Eingabe $\in \mathbb{N}$. Terminiert P mit dieser Eingabe?*

- Jedem Programm P wird eine eineindeutige Nummer $index(P)$ zugeordnet.
- Annahme: es gibt ein Programm $Test$ das folgendes leistet:
  Wendet man $Test$ auf $index(P)$ und x an, so gilt:
  - $Test(index(P), x)$ terminiert mit Ausgabe “ja”, wenn P mit der Eingabe x terminiert. Ansonsten terminiert $Test(index(P), x)$ mit der Ausgabe “nein”.
- man erzeugt ein Programm R, das wie folgt operiert:
  - $R(n)$ terminiert *nicht*, wenn $Test(n, n)$ mit “ja” terminiert; $R(n)$ terminiert, falls $Test(n, n)$ mit “nein” terminiert.

DAS “HALTEPROBLEM” (FORTS.)

Nun lässt man R mit Eingabe $index(R)$ laufen.

- Annahme: $R(index(R))$ terminiert. Dies geschieht nach Konstruktion von R genau dann, wenn $Test(index(R), index(R))$ “nein” ausgibt, was wiederum nach Definition von $Test$ der Fall ist, wenn $R(index(R))$ nicht terminiert. Kann also nicht sein.
- Annahme: $R(index(R))$ terminiert nicht. Dies geschieht nach Konstruktion von R genau dann, wenn $Test(index(R), index(R))$ “ja” ausgibt, was wiederum nach Definition von $Test$ der Fall ist, wenn $R(index(R))$ terminiert. Kann also auch nicht sein.

Es kann $Test$ also nicht geben.

Fazit:

- Man kann i.a. nicht durch ein Programm prüfen, ob ein anderes Programm sich “korrekt” verhält.
- Solche und ähnliche Probleme werden in *Berechenbarkeitstheorie* (Theor. Inf.) untersucht.
- Wie verträgt sich das mit der Church’schen These?

Personen

- John von Neumann (1903–1957; Dr. in Budapest, 1926/27 Student von Hilbert in Göttingen/D, USA): Abstraktes Rechnermodell (1940er) auf dem reale Rechner dann auch basieren.
- Konrad Zuse (1910– 1995; D; (Bau- und Flugzeug)-Ingenieur):
  1938: elektrisch angetriebene mechanische Rechner Z1 etc.; 1942-1946 "Plankalkül", erste höhere Programmiersprache (nicht auf Zx lauffähig).
- Alonzo Church (1903–1995, USA; Math.), Alan Turing (1912–1954; GB; Math.; Student von Church):
  "Church-Turing-These": Vermutung, dass alle formalen Algorithmenbegriffe, auch alle zukünftigen, maximal zu den Algorithmenbegriffen von Church (1930er: Lambda-Kalkül) und Turing (1936: Turing-Maschine) äquivalent sind.
- Kurt Gödel (1906–1978; A/USA; math. Logik):
  1930: "Unvollständigkeitssatz" – "Jedes hinreichend mächtige formale System ist entweder widersprüchlich oder unvollständig".
- Noam Chomsky (1928–...; USA; Linguist):
  1956: "Chomsky-Hierarchie" formaler Grammatiken; siehe später.

1.5 Ausblick auf den weiteren Verlauf der Vorlesung

- Konzepte der Objektorientierung
- Java als objektorientierte Programmiersprache
- Grundlagen von UML als objektorientierte Modellierungssprache
- Algorithmen und Datenstrukturen
  - Aufwandsanalyse von Algorithmen
  - (Korrektheit und Terminierung)
  - Design von Algorithmen (typische Ansätze)

Literatur zur Vorlesung

- es gibt viele gute Info-I-Lehrbücher ("Einführung in die Informatik").
- zu den einzelnen Themen gibt es ebenfalls viele gute Bücher (u.a. zu "Java", "Algorithmen und Datenstrukturen").
- es gibt kein Skript speziell zu dieser Vorlesung.
- Vorlesung basiert zum großen Teil auf "Algorithmen und Datenstrukturen"; G. Saake, K.-U. Sattler, dpunkt-Verlag, 2002; 2. Aufl. 2004.
- Im Web findet man natürlich jede Menge Informationen zu Java:
  Java: `http://java.sun.com` und `http://www.javasoft.com`
  Java-FAQ: `http://www.javasoft.com/faq2html` oder
  `http://sunsite.unc.edu/javafaq/javafaq.html`
- Newsgruppen: comp.lang.java und de.comp.lang.java
  diverse Webforen

1.6 Objektorientierung: die Idee

- Vorgehensweise zur Beschreibung und *Modellierung* von Zuständen/Abläufen/Algorithmen
- Anfang der 90er: Objektorientierte Analyse/Design
  Abstrakte Beschreibung von Abläufen, nicht nur von Programmen.
- gegenwärtig weitest verbreiteter Formalismus:
  UML (Version 1.0 1997 bei der OMG (Object Management Group) zur Standardisierung eingereicht).
- Grundsatz: Wenn man ein Objekt "kennt", also es identifizieren kann, und weiss, welche "Kommandos" es kennt, und welche Effekte diese Kommandos haben, genügt das. Man muss nicht unbedingt wissen, wie es intern aufgebaut ist.
  - Kapselung von (internen) Informationen

Anmerkung:
Objektorientierung ist also weit mehr als "nur" objektorientierte Programmiersprachen!

Objektorientierung

- Organisation des Verhaltens durch *Klassen*
  Beispiel: Klasse "Person"
- Eine *Klasse* beschreibt eine Menge von "gleichartigen" *Objekten*.
  - *Struktur* der Objekte (*"Eigenschaften"*)
    - *Attribute*
      im Beispiel: Vorname: Zeichenkette, Name: Zeichenkette, Geburtsdatum: Datum
    - *Beziehungen* zu anderen Objekten
      im Beispiel: wohnt_in: Stadt, verheiratet_mit: Person
  - *Verhalten* der Objekte (*"Operationen", "Methoden"*): Anfragen an das Objekt, Verändern des Objektzustandes, Auslösen von Aktionen
    im Beispiel: sage_Name$\Rightarrow$ Zeichenkette, sage_Alter$\Rightarrow$ Zahl,
    heirate(Angabe einer Person) $\Rightarrow$ keine Ausgabe, aber Zustandsänderung

Objektorientierung

- [Klassen]
- Damit ist jedes solche Objekt eine *Instanz* dieser Klasse.
  - Zustand: Die Werte der Eigenschaften können für jedes Objekt anders sein, und können sich zeitlich ändern (Name, wohnen, verheiratet sein)
    im Beispiel:
    obj_1 {Name: Meier, Vorname: Karl, Geburtsdatum: 1.1.1950, verheiratet_mit: obj_2}
  - Verhalten: ist durch die Klasse vorgegeben (heiraten, Angabe des Alters aus dem Geburtsdatum)
    im Beispiel:
    obj_1.heirate(obj_3): keine Ausgabe, ändert Zustand von obj_1 (und von obj_3)
    obj_1.sage_Alter: gibt den Wert 52 aus
- Kapselung der Daten und Algorithmen: Nur das Objekt selber kann auf seinen Zustand zugreifen. Von außen kann man mit dem Objekt nur über sein Verhalten kommunizieren.
- ein Algorithmus wird dann dadurch gegeben, geeignete Objekte geeignet kommunizieren/zusammenarbeiten zu lassen.

JAVA

- Objektorientierte Programmiersprache mit imperativem Kern
- Organisation der Struktur und des Verhaltens durch *Klassen*
- Implementierung des Verhaltens dann durch imperativen Programmcode

Kapitel 2
Vorarbeiten

Im weiteren werden einige Dinge benötigt:

- Wie werden Zahlen im Rechner dargestellt?
  Binär. Rechner kennen nur Nullen und Einsen.
- Wie beschreibt man die zulässigen Konstrukte einer Programmiersprache?
  Durch eine Grammatik. Wie bei anderen Sprachen auch!
- Wie formuliert man “Bedingungen”, und wie wertet man sie aus?

2.1 Zahlendarstellung im Computer

- Rechner kennen nur Nullen und Einsen
  (bzw. “Ja” und “Nein”, bzw. “Spannung drauf” und “keine Spannung drauf”).
- “kleinste Daten-/Speichereinheit” ist entweder 0 oder 1 (“1 Bit”).
- Speicher wird in “Paketen” zu je 8 solcher Einheiten (“1 Byte”) verwaltet.
- Man muss Zahlen also in irgendeiner Form mit diesen Möglichkeiten codieren.
- Details: siehe Technische Informatik

2.1.1 Ganze Zahlen

- Vgl. Dezimalsystem:
  Wir verwenden “Ziffern” von 0-9, größere Zahlen werden als “Worte” aus 0-9 dargestellt, wobei jeder “Stelle” eine Wertigkeit zugewiesen ist: $1 = 10^0$, $10 = 10^1$, $100 = 10^2$; die n-te Stelle entspricht jeweils 10^n.
- Codierung im Binärsystem: Dasselbe, mit 0 und 1.
  Wertigkeit 1, 2, 4, 8, 16, 32, 64, 128 etc.
- Einfluss der Speicheraufteilung:

  kleinste vergebbare “Menge”: 8 Bits (“ein Byte”): $x = \sum_{i=0}^{7} x_i \cdot 2^i$

–	0	0	0	0	0	0	0	0	= 0.
–	0	0	0	0	0	0	0	1	= 1.
–	0	0	0	1	0	1	1	0	= 2 + 4 + 16 = 22.
–	1	1	0	0	0	0	0	0	128 + 64 = 192.
–	1	1	1	1	1	1	1	1	= 255 ist so die größte mit 8 Bit darstellbare Zahl.

Übungsaufgabe

Sie kennen das übliche Verfahren zur schriftlichen Addition im Dezimalsystem:

	1	4	9	2
+	1	7	8	9
	1	1	1	
	3	2	8	1

- Machen Sie dasselbe im Binärsystem (konvertieren Sie die Zahlen ins Binärsystem, und addieren sie dann): 42 + 56

NEGATIVE ZAHLEN

Interpretation des führenden Bits verschieden:

- Möglichkeit: 7-Bit-Betrag + 1-Bit-Vorzeichen:

1	x	x	x	x	x	x	x	für negative Zahlen.

  Damit hat man aber

0	0	0	0	0	0	0	0	= (+)0 und	1	0	0	0	0	0	0	0	= - 0.

  Additionsalgorithmus in dieser Darstellung ebenfalls problematisch.

- "Zweierkomplement-Darstellung": Das führende Bit wird als $-(2^{n-1})$ gewertet:

1	0	0	0	0	0	0	0	= $-(2^7)$ = -128
1	1	0	0	0	0	0	0	= -128 + 64 = -64
1	1	1	1	1	1	1	1	= -128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1

  ... und man kann Zahlen von -128, . . . , -1, 0, 1, . . . , 127 darstellen.

NEGATIVE ZAHLEN: ZWEIERKOMPLEMENT

Erzeugung des Zweierkomplements einer gegebenen Zahl:
alle Bits kippen, 1 dazuzählen, ggf. das vorne übergefallene Bit vergessen:

0	0	0	0	0	1	0	1

  = 4 + 1 = 5

  Bits kippen:

1	1	1	1	1	0	1	0

  und 1 dazuzählen:

1	1	1	1	1	0	1	1

  = -128 + 64 + 32 + 16 + 8 + 2 + 1 = -5

- Man hat auch nur noch eine 0:

0	0	0	0	0	0	0	0

  = 0

  Bits kippen:

1	1	1	1	1	1	1	1

  und 1 dazuzählen:

0	0	0	0	0	0	0	0

  = 0

RECHNEN MIT DEM: ZWEIERKOMPLEMENT

Weiterer Vorteil dieser Darstellung:

Man benötigt nur einen Algorithmus für Addition und Subtraktion gemeinsam:

	0	0	0	0	0	1	0	1	=	5
+	1	1	1	1	1	0	1	1	=	-5
(1)	0	0	0	0	0	0	0	0		

Übungsaufgabe

- Addieren Sie 97 und (-31). Überprüfen Sie Ihr Ergebnis im Dezimalsystem.
- Addieren Sie 55 und (-87).
- Addieren Sie (-31) und (-44).
- Übertragen Sie die Idee des Zweierkomplements in das Dezimalsystem und berechnen Sie damit 1789-1492 und 1492-1789.

Ganze Zahlen (Forts.)

- Entsprechend kann man mit 16 Bits Zahlen von $-2^{15}, \ldots, -1, 0, 1, \ldots, 2^{15} - 1$ darstellen.
- 32 Bits genügen für $-2^{31}, \ldots, -1, 0, 1, \ldots, 2^{31} - 1$
- das kann man jetzt beliebig weiterführen ... und braucht beliebig viel Speicher:
- 10^{50} bräuchte 166 Bits.
- Mit 64 Bits kann man Zahlen von $-2^{63}, \ldots, -1, 0, 1, \ldots, 2^{63} - 1$ darstellen.
  Was ist
  00110100 01111101 01010100 00111001 11110000 01101101 11010001 11100011 ?
- Fragen Sie ihren Taschenrechner, was 2^{63} ist.

2^{63} ist 9.223372037 E18.

2.1.2 Große und Reelle Zahlen

- Darstellung durch *Exponent* (18) und *Mantisse* (9.223372037) zur *Basis* 10.
- Die Anzahl k der Stellen der Mantisse legt die *Genauigkeit* der Zahl fest,
- der Exponent ist eine ganze Zahl – je nachdem wieviele Bit man dafür verwendet kann man "ziemlich große" Zahlen darstellen:
- Beispiel:
  Mantisse mit 4Byte (32 Stellen) und 1-Byte Exponent
  $2^{32} = 4.3 \cdot 10^9 \rightarrow$ 9 (Dezimal)stellen Genauigkeit
  Exponent (zur Basis 2, im Zweierkomplement): größter Exponent: $2^{2^7} = 3.4 \cdot 10^{38}$
- mit *negativen Exponenten* kann man auch *betragsmäßig* sehr kleine Zahlen darstellen – bis $2^{-(2^7)} = 2.9 \cdot 10^{-39}$.
- die Deutung der Kommastelle in der Mantisse ist sehr von der jeweiligen Hardware abhängig.

2.1.3 Das Hexadezimalsystem

- Dezimalsystem: Basis 10. In Europa seit 17.Jhdt üblich
- Binärsystem: siehe eben.
- Zwölfer-System: war in Europa zum Teil im Mittelalter für Münzen üblich.
  In England auch noch länger.
- Hexadezimalsystem: Basis $16 = 2^4$.

Idee: jeweils 4 Bit zusammenfassen - damit lassen sich Zahlen von 0 bis 15 darstellen ⇒ “Ziffern”

Ein Byte ist also eine 2-stellige Hexadezimalzahl

1	0	1	0	1	1	0	1	$= ((8 + 2) \cdot 16) + (8 + 4 + 1) = 173$

DAS HEXADEZIMALSYSTEM (FORTS.)

- Sinnvoll ist es, in einem System zu rechnen, das entsprechend viele “Ziffern” hat (Stellenschreibweise)
  - 0,1 – Binärsystem ✓
  - 0,. . . ,9 – Dezimalsystem ✓
  - 0,. . . ,9 – Zwölfersystem ??
  - 0,. . . ,9 – Hexadezimalsystem ??
    Ziffern: 0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F
    Die obige Zahl 173 wird also als “AD” geschrieben.

  Die Zahl
  00110100 01111101 01010100 00111001 11110000 01101101 11010001 11100011
  ist kurz 34 7D 54 39 F0 6D D1 E3.

2.2 Einführung in Formale Sprachen

- Programmiersprachen sind Sprachen
- Erlaubte Sätze in Sprachen werden durch eine Grammatik beschrieben:
  ein (einfacher) Satz besteht aus einem Subjekt, einem Prädikat und einem Objekt:
  "Otto isst Schokolade".
- Programmiersprachen (und ähnliche Dinge) sind (glücklicherweise) regelmäßiger aufgebaut und "einfacher" als natürliche Sprachen.

2.2.1 Grammatiken

Definition:

- Ein *Alphabet* T ist eine Menge von Symbolen.
- Ein *Wort* w über einem Alphabet T (geschrieben $w \in T^*$) ist eine Zeichenkette bestehend aus Symbolen aus T.
- dazu zählt auch das *leere Wort* – häufig mit ε bezeichnet.

- Eine Sprache über einem Alphabet T ist dann eine Menge von "erlaubten" Worten über T.

Definition:

Eine *Grammatik* $G = (V, T, P, S)$ besteht aus:

- einer Menge V von *Nichtterminalsymbolen*
- einer Menge T von *Terminalsymbolen* ($V \cap T = \emptyset$)
- einer Menge P von *Produktionen* (oder *(Bildungs)Regeln*) der Form $p \rightarrow q$ mit $p \in (V \cup T)^*$ (häufig $p \in V$) und $q \in (V \cup T)^*$
- einem Startsymbol $S \in V$.

- Ist $x = x_1 \ldots p \ldots x_n$ ein Wort aus $(V \cup T)^*$, das die linke Seite einer Regel $p \rightarrow q$ enthält, dann kann man durch Ersetzen von p durch q ein Wort $y = x_1 \ldots q \ldots x_n$ erhalten und schreibt dafür $x \Longrightarrow y$ (oder $x \stackrel{G}{\Longrightarrow} y$ oder $x \stackrel{p \rightarrow q}{\Longrightarrow} y$).
- ein Wort y heißt *ableitbar* aus einem Wort x mit Hilfe von G, kurz $x \stackrel{*}{\Longrightarrow} y$, wenn entweder $x = y$, oder es $n \geq 1$ Worte $x_1, x_2, \ldots, x_n$ gibt mit $x = x_1$, $y = x_n$, und $x_i \stackrel{G}{\Longrightarrow} x_{i+1}$ für $1 \leq i < n$.
- Die von einer Grammatik G *erzeugte Sprache* $L(G)$ ist nun die Menge aller aus dem Startsymbol S ableitbaren Worte, die nur aus Terminalzeichen bestehen:

$$L(G) = \{w \mid S \stackrel{*}{\Longrightarrow} w \text{ und } w \in T^*\} \;.$$

Grammatiken: Beispiel

Zu der hawaiianischen Sprache (siehe Folie 24) kann man verschiedene Grammatiken angeben:

$G = (\{S, Vok, Kons, V, K\}, \{a, e, i, o, u, h, k, l, m, n, p, w\}, R, S)$ mit

$$\begin{aligned} R = \{ \;\; & S \rightarrow Vok\ V \mid Kons\ K \\ & V \rightarrow \varepsilon \mid Vok\ V \mid Kons\ K \\ & K \rightarrow Vok\ V \\ & Vok \rightarrow a|e|i|o|u \\ & Kons \rightarrow h|k|l|m|n|p|w \;\} \end{aligned}$$

Hier spiegeln V und K die Zustände q_v und q_k des endlichen Automaten von Folie 26 wider.

Beispiele:

- *Ableitungsbaum* angeben
- äquivalente “rechtslineare” Grammatik entwickeln, in der das Nichtterminalzeichen immer rechts weitergeschoben wird.

Grammatiken: Beispiel

Arithmetische Ausdrücke können wie folgt *rekursiv* definiert werden:

- Jede Darstellung einer Zahl ist ein arithmetischer Ausdruck
- Ist E ein arithmetischer Ausdruck, so ist auch $(-E)$ ein arithmetischer Ausdruck.
- Sind E_1 und E_2 arithmetische Ausdrücke, so sind auch $(E_1 + E_2)$, $(E_1 - E_2)$, $(E_1 * E_2)$ und (E_1/E_2) arithmetische Ausdrücke.
- Sonst nichts.

Eine entsprechende Grammatik wäre nun

$$G_A = (\{A, Z\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)\}, P, A)$$

mit $P = \{\ Z \rightarrow 0|1|2|3|4|5|6|7|8|9|ZZ,$

$\quad A \rightarrow Z|(-A)|(A + A)|(A - A)|(A * A)|(A/A)\ \}$

wobei die Regel $X \rightarrow w_1|w_2$ als Abkürzung für die alternativen Regeln $X \rightarrow w_1$ und $X \rightarrow w_2$ steht.

Beispiel: Ableitung des Ausdruckes $((17 + 4) * 372)$ mit *Ableitungsbaum*.

Eine andere Term-Grammatik

Eine andere, etwas detailliertere Grammatik:

$G_A = ($ {Term, Produkt, Faktor, Summe, Zahl},
{0,1,2,3,4,5,6,7,8,9,+,*,(,)}, P, Term)

mit $P = \{$ Zahl → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Zahl Zahl

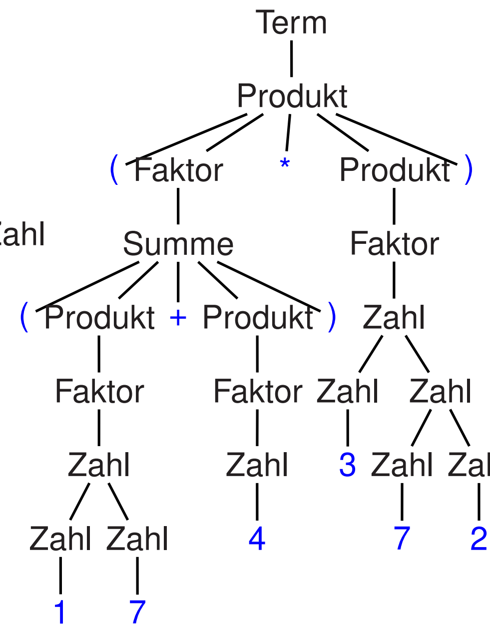
Term → Produkt

Produkt → (Faktor * Produkt) | Faktor

Faktor → Summe | Zahl

Summe → (Produkt + Produkt)$\}$

Beispiel: Ableitung des Ausdruckes $((17 + 4) * 372)$.



GRAMMATIKEN: ABSTRAKTE BEISPIELE

1. Gesucht: $G = (V, T, P, S)$ so dass $L(G) = \{a^n b^m \mid n, m \in \mathbb{N}, n, m \geq 1\}$.
   - $V = \{S, A, B\}$, $T = \{a, b\}$
   - $P = \{S \to aA,\ A \to aA,\ A \to B,\ B \to bB,\ B \to b\}$.

   Kann man das auch so machen, dass es gleichviele a wie b sind?
2. Gesucht: $G = (V, T, P, S)$ so dass $L(G) = \{a^n b^n \mid n \in \mathbb{N}, n \geq 1\}$.
   - $V = \{S, A\}$, $T = \{a, b\}$
   - $P = \{S \to aAb,\ A \to aAb,\ A \to \varepsilon\}$.
     (Hinweis: es muss mindestens ein a und b erzeugt werden!)

GRAMMATIKEN: ABSTRAKTE BEISPIELE (FORTS.)

3. und wenn man auch noch n cs haben will? Gesucht: $G = (V, T, P, S)$ so dass $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}, n \geq 1\}$.
   - $V = \{S, A, B\}$, $T = \{a, b, c\}$
   - $P_1 = \{S \to aAbc,\ A \to aAbc,\ A \to \varepsilon\}$.

   Mit P_1 kann man z.B. $aabcbc$ und $aaabcbcbc$ erzeugen.
   Man benötigt weitere Regeln, um die bs und cs zu *vertauschen*, und muss verbieten, dass terminale bs an der falschen Stelle stehen.
   - $V = \{S, A, B\}$, $T = \{a, b, c\}$
   - $P_2 = \{\ S \to aABc,\ A \to aABc,\ A \to \varepsilon$
     $cB \to Bc$ // vertauschen
     $aB \to ab$ // Bs terminieren nur an der richtigen Stelle
     $bB \to Bb$ // wo sie erst einmal hinkommen müssen $\}$

Grammatiken: Aufgabe

Geben Sie eine Grammatik für Telefonnummern an.

Startsymbol: G
Weitere Nichtterminalsymbole z.B.

- S: Stadtgespräch
- F: Ferngespräch
- A: Auslandsgespräch
- VF, VA: dasselbe als Call-by-Call mit Providervorwahl
- DS, DF, DA: dasselbe als Dienstgespräche aus der Uni - man muss eine Null vorwählen um eine Amtsleitung zu bekommen.

Ähnliches Beispiel: deutsche Autokennzeichen.

Grammatiken und Sprachen

- Für den Benutzer:
  Beschreiben einer Sprache (vgl. arithmetische Ausdrücke)
  insbesondere die *Syntax* einer Programmiersprache
- Für den Computer:
  Dieser muss bei einem gegebenen Ausdruck ("Satz", "Wort")
  - prüfen ob er korrekt ist
    z.B.: Sind "$+314 - (*5/)$" und "((17+4)*372)" arithmetische Ausdrücke?
  - falls er nicht korrekt ist, soll ein Hinweis erzeugt werden, wo die Probleme liegen,
  - falls er korrekt ist, muss er zerlegt und ausgewertet werden (*Semantik*). Dabei muss seine Ableitung zurückverfolgt werden:

$$A \Longrightarrow (A * A) \Longrightarrow ((A + A) * A) \overset{*}{\Longrightarrow} ((Z + Z) * Z) \overset{*}{\Longrightarrow} ((17 + 4) * 372)$$

- Der Zerlegungsprozess wird als *Parsing* bezeichnet.
- Je nach Typ der Grammatik ist dies unterschiedlich kompliziert.

⇒ Bei einer Programmiersprache ist es wünschenswert, dass es genügt, ein Programm einmal linear von links oben nach rechts unten durchzugehen, um es zu zerlegen.

Grammatiken und Sprachen: Klassifizierung

Es gibt unterschiedliche Typen von Grammatiken, klassifiziert nach der Form ihrer Regeln (Noam Chomsky, 1959 – theoretische Informatik):

Typ	Name	Regelart $p \to q$
0	Phrase-structure G., unbeschränkte G.	$p \in (V \cup T)^* \setminus \{\varepsilon\}$ $q \in (V \cup T)^*$
1	Kontextsensitive G., monotone G.	$p \in (V \cup T)^* \setminus \{\varepsilon\}$ $q \in (V \cup T)^*$, Länge(p) $\leq$ Länge(q)
2	Kontextfreie G.,	$p \in V$ $q \in (V \cup T)^*$
3 3a 3b	reguläre G., linkslineare G. rechtslineare G.	 $p \to qt \mid \varepsilon$ $p \to tq \mid \varepsilon$ $p, q \in V, t \in T$

Hierarchiesatz: Ist L_i die Menge der Sprachen vom Typ i, dann ist $L_0 \supsetneq L_1 \supsetneq L_2 \supsetneq L_3$.

Grammatiken und Sprachen (Forts.)

Anmerkung: die Chomsky-Hierarchie klassifiziert *Grammatiken*, nicht Sprachen!
Eine Sprache kann z.B. regulär sein, obwohl eine nichtreguläre Grammatik angegeben ist (z.B. die erste Sprache auf Folie 56).

- reguläre Sprachen sind sehr "einfach" zu parsen, sind aber sehr "schwach".
  - Man kann nicht einmal öffnende/schließende Klammerpaare überwachen.
  - Lineare Grammatiken haben lineare Ableitungen.
  - Für jede reguläre Sprache kann man einen endlichen Automaten (basierend auf einer rechtslinearen Grammatik für diese Sprache) angeben, der genau die Worte akzeptiert, die in dieser Sprache enthalten sind.

Aufgabe

- Geben Sie eine rechtslineare Grammatik an, die gültige Telefonnummern erzeugt.
- Geben Sie einen endlichen Automaten an, der testet ob eine gegebene Ziffernfolge eine gültige Telefonnummer ist.

GRAMMATIKEN UND SPRACHEN (FORTS.)

- kontextfreie Sprachen sind immer noch einfach zu parsen, aber mächtiger. Man kann damit z.B. Klammerpaare überwachen.

  Die zweite Sprache auf Folie 56 ist kontextfrei, aber nicht regulär.

  - Für Ableitungen in kontextfreien Grammatiken können Ableitungsbäume angegeben werden.
  - Parser (also Programme, die die Zerlegung übernehmen) können automatisch generiert werden ($\Rightarrow$ Compilerbau; Programme: lex/yacc)

  $\Rightarrow$ eignen sich sehr gut für Programmiersprachen.

GRAMMATIKEN UND SPRACHEN (FORTS.)

- Parsen kontextsensitiver Sprachen ist aufwendig und für Programmiersprachen nicht praktikabel.

  Die Grammatik auf Folie 57 ist kontextsensitiv, aber nicht kontextfrei. Es existiert auch keine kontextfreie Grammatik für diese Sprache.

- Die meisten Programmiersprachen haben eine Grammatik, deren Struktur kontextfrei ist,
- einige Dinge (z.B. die Überprüfung ob alle Variablen auch deklariert sind) gehen über Kontextfreiheit hinaus.

2.2.2 Beschreibung von Programmiersprachen durch Grammatiken

- Grammatiken sind produktionen-basiert
- Eine Beschreibung für den Benutzer soll sich an die logische Struktur einer Sprache/eines Programms halten

ERWEITERTE BACKUS-NAUR-FORM

- verwendet "::=" anstatt "→",
- Nichtterminale werden durch < ... > eingeschlossen,
- Terminalzeichen werden durch " ... " eingeschlossen,
- wie bereits oben bezeichnet | Alternativen,
- Gruppierung durch { ... } für "0 oder mehr Wiederholungen von ...",
- Gruppierung durch [...] für optionale Teile.

EBNF: BEISPIELE

1. Beschreibung der Darstellung von *Zahlen* durch die übliche Darstellung als ±123.4567 oder durch die Mantisse/Exponent-Darstellung als ±1.2345 E±67:

   <Ziffer> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

   <Zahl> ::= <Kommazahl> | <Mantisse> "E" ["+"|"-"] <Ziffernfolge>

   <Ziffernfolge> ::= <Ziffer> {<Ziffer>}

   <Kommazahl> ::= ["+"|"-"] <Ziffernfolge> ["." <Ziffernfolge>]

   <Mantisse> ::= ["+"|"-"] <Ziffer> ["." <Ziffernfolge>] (Hinweis: nur eine Vorkommastelle)

2. *Bezeichner* (z.B. als Namen von Variablen etc.) bestehen aus Buchstaben und Zahlen. Das erste Zeichen muss ein Buchstabe sein:

   <Buchstabe> ::= "a"|"b"| ... |"z"|"A"|"B"| ... |"Z"|

   <Ziffer> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

   <Bezeichner> ::= <Buchstabe>{<Buchstabe>|<Ziffer>}

... wir werden beide wieder benötigen.

Aufgabe: geben Sie eine EBNF für Telefonnummern an.

2.3 Ein bisschen Logik

"Boolesche Logik" (G. Boole, 1815-1864) bezeichnet das "logische Rechnen" mit den Wahrheitswerten "$wahr$" und "$falsch$".

- 1854: "An investigation into the Laws of Thought, on Which are founded the Mathematical Theories of Logic and Probabilities" – Abbildung von -bis dahin rein philosophischer- Logik auf die *Boole'sche Algebra* – mathematische Logik.
- Diese werden z.B. in Programmiersprachen beim Auswerten von Bedingungen benötigt.
- "Logik" ist ein spezielles Teilgebiet der theoretischen Informatik ...

Boole'sche Logik

- (boole'sche) Werte sind "$wahr$" und "$falsch$",
- (boole'sche) Operatoren sind z.B. "nicht" (Zeichen: $\neg$) "und" (Zeichen: $\wedge$), "oder" (Zeichen: $\vee$), "exklusiv-oder"
- Die Bedeutung (="Semantik") der Operatoren ist durch *Wahrheitstabellen* gegeben:

$$\neg\, wahr = falsch \qquad \neg\, falsch = wahr$$

$\wedge$	$wahr$	$falsch$
$wahr$	$wahr$	$falsch$
$falsch$	$falsch$	$falsch$

$\vee$	$wahr$	$falsch$
$wahr$	$wahr$	$wahr$
$falsch$	$wahr$	$falsch$

xor	$wahr$	$falsch$
$wahr$	$falsch$	$wahr$
$falsch$	$wahr$	$falsch$

Es gibt nun verschiedene Logiken, die auf diesen Operatoren aufbauen:

- hier und jetzt: Aussagenlogik
- später: Prädikatenlogik, First-Order-Logic
- theoretische Informatik: mehrwertige Logiken, Modallogiken, Temporallogiken, ...

Aussagenlogik: Syntax

- Die *Sprache* der Aussagenlogik verwendet ein Alphabet, das die folgenden Dinge umfasst:
  - "(" und ")" sowie die logischen Symbole $\neg$, $\wedge$, $\vee$
  - eine unendliche Menge von Variablen $A, B, A_1, A_2, \ldots$.

(aussagenlogische) Formeln sind sozusagen die "erlaubten Sätze" in dieser Sprache, die über dem o.g. Alphabet gebildet werden können. Die Menge der Formeln ist *induktiv* definiert:

- eine aussagenlogische Variable A ist eine Formel.
- Ist F eine Formel, so ist auch $\neg F$ eine Formel.
- Sind F und G Formeln, so sind auch die *Konjunktion* $(F \wedge G)$ und die *Disjunktion* $(F \vee G)$ Formeln.

Übungsaufgabe: Geben sie eine Grammatik für aussagenlogische Formeln in denen nur die Variablen "A", "B", "C" vorkommen, an.

Aussagenlogik: Semantik

"Semantik" ist "was *bedeutet* die Formel?"

Eine aussagenlogische *Interpretation* $\mathcal{I}$ weist allen aussagenlogischen Variablen A_i einen Wahrheitswert $\mathcal{I}(A_i)$ (also entweder "$wahr$" oder "$falsch$") zu. Basierend darauf wird dann berechnet, ob eine Formel F unter der gegebenen Interpretation gilt, oder nicht.

Man schreibt $\mathcal{I} \models F$ für "F ist wahr in $\mathcal{I}$". $\models$ ist durch *strukturelle Induktion* definiert (analog der Syntaxdefinition von Formeln):

- $\mathcal{I} \models \neg G$ genau dann, wenn $\mathcal{I} \not\models G$.
- $\mathcal{I} \models G \wedge H$ genau dann, wenn $\mathcal{I} \models G$ und $\mathcal{I} \models H$.
- $\mathcal{I} \models G \vee H$ genau dann, wenn $\mathcal{I} \models G$ oder $\mathcal{I} \models H$.

Übung: Sei $F = (A \wedge B) \vee (\neg A)$.

- Geben Sie eine Ableitung dieser Formel in "ihrer" Grammatik an.
- Sei $\mathcal{I}(A) = wahr$ und $\mathcal{I}(B) = falsch$. Gilt $\mathcal{I} \models F$?
- Geben Sie eine Interpretation $\mathcal{J}$, so dass $\mathcal{J} \models F$.

AUSSAGENLOGIK: WAHRHEITSTABELLEN

- Die “Gültigkeit” einer Formel für *eine* Interpretation erhält man durch bottom-up-Auswertung der Formel.
- Eine Aussage über *alle* Interpretationen kann man mit einer Wahrheitstabelle machen, z.B. für $(A \vee B) \vee (\neg A \wedge \neg B)$.
  Dabei wird schrittweise über die Teilformeln vorgegangen (strukturelle Induktion):

A	B	$A \vee B$	$\neg A$	$\neg B$	$\neg A \wedge \neg B$	$(A \vee B) \vee (\neg A \wedge \neg B)$
F	F	F	T	T	T	T
F	T	T	T	F	F	T
T	F	T	F	T	F	T
T	T	T	F	F	F	T

- Formeln, die für alle Interpretationen T ergeben, heißen “allgemeingültig”
- zwei Formeln, die für alle Interpretationen denselben Wert haben, heißen “äquivalent”.

AUSSAGENLOGIK: DIVERSES

- Prioritätsregel: “$\wedge$” bindet stärker als “$\vee$”
  erlaubt das Weglassen von Klammern.
- abgeleitete Operatoren können als “Kurzform” für Teilformeln definiert werden. So ist (i) “A xor B” als Kurzform für (ii) “$(A \vee B) \wedge \neg(A \wedge B)$” definiert

  Übung: Zeigen Sie (durch Aufstellen der Wahrheitstabelle von (ii)), dass (i) und (ii) äquivalent sind.
- Weitere häufig verwendete Regeln zur Umformung sind die *de Morgan’schen* Regeln:

$$\neg(A \wedge B) \equiv (\neg A \vee \neg B) \quad \text{sowie} \quad \neg(A \vee B) \equiv (\neg A \wedge \neg B)$$

  Beweisen Sie beide durch Wahrheitstabellen.

Kapitel 3
Java

(vgl. Folie 38)

- Objektorientierte Programmiersprache mit imperativem Kern
  (also sehr ähnlich zu z.B. C++ und Eiffel)
- Organisation der Struktur und des Verhaltens durch *Klassen*
  Eine *Klasse* beschreibt eine Menge von "gleichartigen" *Objekten*.
- Implementierung des Verhaltens dann durch imperativen Programmcode

Anmerkung: In dieser Vorlesung werden die konzeptuell wichtigen Eigenschaften von Java besprochen. Darüber hinausgehende Details finden Sie in entsprechenden Büchern.

3.1 Java – Get Started

Ein einfaches (untypisches!) Java-Programm

File: EinfacheAusgabe.java

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

- (untypische) Klassendeklaration
  - keine Attribute - also kein Zustand möglich
  - Deklaration einer Methode "main"
    * Schlüsselwort static: "main" ist eine *Klassen-Methode*
    * Klassen, die eine Klassen-Methode main anbieten, sind als *Applikation* ausführbar.
      (an sich sind sie sehr untypische Klassen)
    * formaler Dummy-Parameter "args" (wird ignoriert)
    * gibt einfach "Hello World!" aus

Programmaufruf

Wie eben besprochen, ist “Einfache Ausgabe” als *Applikation* ausführbar.

- Direkter *Java-“Programmtext”* ist nicht ausführbar
- wird erst in *Java-Bytecode* (plattformunabhängig) übersetzt (“compiliert”):

  javac EinfacheAusgabe.java

  – führt einen Syntax-Check des Programms durch,

  – erzeugt das File “EinfacheAusgabe.class”.

  Vorteil: Das erzeute Byte-Code-File kann auf *jedem* Rechner ausgeführt werden, auf dem eine *Java Virtual Machine (JVM)* installiert ist.

  Nachteil: Da die JVM bei der Ausführung dazwischengeschaltet ist, ist es etwa um den Faktor 10-20 langsamer als C/C++ (die JVM muss den Bytecode erst *interpretieren* und intern in prozessorabhängigen *Maschinencode* übersetzen).

- Der Aufruf geschieht durch

  java EinfacheAusgabe

  und erzeugt die Ausgabe

  Hello World!

3.2 Java – ein erster Einblick in das Java-Klassenkonzept

Beispiel

Klassen als Modellierungskonzept

- Eine einfache Person-Klasse:

```
public class Person{
   private String name;
   public void setName(String thename){
     name = thename;
   }
   public String getName(){
     return (name);
   }
   public void printName(){
     System.out.println(name);
   }
}
```

Die Klasse Person

- Jede Instanz der Klasse “Person”
  - hat einen Namen,
  - man kann den Namen setzen,
  - man sich den Namen geben lassen,
  - und die Person ihren Namen ausgeben (drucken) lassen.

- Jede Java-Programmanweisung wird durch ein Semikolon (“;”) abgeschlossen. (Im Gegensatz zu C/C++ gilt dies nicht für Methoden-Deklarationen)

Application-Klasse verwendet Modellierungs-Klasse

- *Eine* Klasse als Dummy zum Aufruf:

```
public class Persons{
  public static void main (String[] args){
    Person p = new Person();
    p.setName("Meier");
    p.printName();
    p.setName("Mueller");
    System.out.println(p.getName());
  }
}
```

- Übersetzen und Ausführen:

```
javac Person.java
javac Persons.java
java Persons
Meier
Mueller
```

Erklärung des Beispiels

Die Klasse Person

- dient dazu, mit `new Person()` Instanzen (=Personen) zu erzeugen
- Die Klasse definiert die Struktur dieser Person-Instanzen.
  - Jede Person hat eine Eigenschaft “name”, die String-wertig ist (Zeichenkette)
- bietet Methoden an
  - um den Namen auf einen angegebenen String-Wert zu setzen,
  - und sie nach dem Namen zu fragen (ohne Argumente anzugeben).
- Methoden können einen Rückgabewert erzeugen, oder nicht (void)
- diese Methoden gehören zu den Instanzen und sind von aussen aufrufbar (public-Keyword)

Die Klasse Persons

- Persons ist eine Applikations-Klasse
  - denn sie hat eine main-Methode, die als “public static void” deklariert ist:
    * “public” bedeutet, dass diese Methode von aussen sichtbar ist,
    * “static” bedeutet, dass es eine Klassenmethode ist (wäre bei eventuell gebildeten Instanzen nicht vorhanden)
    * void bedeutet, dass sie keinen Rückgabewert erzeugt
    * `main` wird automatisch aufgerufen, wenn die Klasse angesprochen wird,
  - Der Aufruf von “main” – also bereits der Klassenaufruf – tut etwas:
    * Erzeugung einer Instanz der Klasse “Person” und Zuweisung an die entsprechend deklarierte Variable “p”
    * Setzen des Namens dieser Instanz
    * Nachricht an die Instanz, dass sie ihren Namen ausgeben soll.
    * Umbenennung der Person in “Mueller”.
    * Nachricht an die Instanz, dass sie ihren Namen zurückgeben soll, der sogleich ausgegeben wird.
- von Persons sollen keine Instanzen erzeugt werden.

ÜBUNGSAUFGABE

Um Klassen zu *testen*, kann man direkt auch die geschriebene Klasse zur Applikationsklasse machen, indem man zu ihr die Methode “main” als Klassenmethode hinzudefiniert und aufruft.

- Vollziehen Sie dies für “Person” nach.
- Erzeugen sie mindestens zwei weitere Personen
  - einmal nacheinander, indem sie jeweils auch die Variable “p” verwenden,
  - einmal unter Verwendung weiterer Variablen, so dass alle drei Personen gleichzeitig existieren,
  - benennen Sie eine der existierenden Personen um.

BLOCKSTRUKTUR VON JAVA-PROGRAMMEN

- Im Sinne eines strukturierten Aufbaus bestehen Java-Programme aus *Blöcken*.
- Ein *Block* ist syntaktisch im Programmcode jeweils explizit durch “$\{ \ldots \}$” begrenzt.
  - Der Klassenrumpf ist jeweils ein Block,
  - Die Methodenrümpfe sind ebenfalls Blöcke (in den Klassenrumpf geschachtelt),
  - Im Zuge der Einführung von Programmkonstrukten werden weitere Blöcke eingeführt werden.
- Viele Dinge (Namen, Variablen, Methoden etc.) sind nur innerhalb gewisser Blöcke bekannt und von außen nicht sichtbar (Informationskapselung).
- ... wird später nochmal genauer behandelt, wenn *Klassen*- und *Methodendeklarationen*, *Variablendeklarationen*, sowie die weiteren Programmkonstrukte eingeführt wurden.

3.3 Datentypen, Variablendeklarationen und -zuweisungen in Java

- Java ist *streng typisiert*
- es gibt eingebaute Datentypen und benutzerdefinierte Datentypen.
- Die Deklaration von Variablen und Methoden benötigt die Angabe von Datentypen. Beim Übersetzen wird überprüft, ob die jeweils verwendeten Typen mit der Deklaration verträglich sind.

3.3.1 Primitive Datentypen

- “Primitiv” bedeutet hier im mathematischen Sinn “nicht weiter zerteilbar”
- Vom Standpunkt der Programmierung und objektorientierten Modellierung sind diese Datentypen *keine* “echten” Klassen (von denen man Instanzen mit eigener *Identität* erzeugen kann), sondern sie sind nur *Literaltypen*.
- der Unterschied zwischen *Literalen* und *Objekten* wird später noch klarer ...
- momentan ist wichtig, dass Literale die einfachsten Datentypen sind, mit denen man *Werte* “einfach so” durch hinschreiben erzeugen kann.
- Werte jedes der im folgenden beschriebenen primitiven Literaltypen benötigen eine feste Menge Speicherplatz, die nur vom Datentyp abhänig ist. Damit können sie beim Programmablauf “vor Ort” angelegt werden. Auch das wird später klarer ...

Ganze Zahlen

Datentypen für ganze Zahlen:

Name	Wertebereich
byte	$-2^7, \ldots, 0, 1, \ldots, 2^7 - 1$
short	$-2^{15}, \ldots, 0, 1, \ldots, 2^{15} - 1$
int	$-2^{31}, \ldots, 0, 1, \ldots, 2^{31} - 1$
long	$-2^{63}, \ldots, 0, 1, \ldots, 2^{63} - 1$

(Vergleiche Zahlendarstellung im Zweier-komplement (Folie 46))

Operationen:

- "+", "-", "$*$": wie üblich
- "/": "a/b" ist der ganzzahlige Anteil der Division "a durch b"
- "%": "a%b" ergibt den Rest der Division "a durch b"
- es gilt a%b = a - b$*$(a/b)

Vergleiche:

- "==": (Gleichheit; "=" ist die Variablenzuweisung), "!=": Ungleichheit,
- "$>$", "$>=$", "$<$", "$<=$" für $>, \geq, <, \leq$.

Weiteres zu ganzen Zahlen

- bitweise Operationen:

  In der Binärdarstellung kann jedes Bit als Wahrheitswert 0=$falsch$ und 1=$wahr$ aufgefasst werden:

  "&" (bitweise logisches und), "|" (bitweise logisches oder), "$\wedge$" (bitweise exklusiv-oder) , "$\sim$" (bitweise Negation), "$<<$" (shift left), "$>>$" (shift right);
  Beispiel: 27$<<$2 ist 108, entspricht 2x linksschieben und damit einer Multiplikation mit 4.

  Diese Operationen werden speziell zur hardwarenahen Programmierung benötigt

  Übungsaufgabe (für später): Bei der Behandlung der Zweierkomplementdarstellung (Folie 43) wurde gezeigt, wie die Subtraktion "a minus b" auf die Addition zurückgeführt wird. Schreiben Sie eine kleine Java-Test-Klasse, die dies tut.

- (Die verschiedenen Datentypen sind in Java-"Packages" implementiert. Dort sind weitere Konstanten verfügbar, z.B. Integer.MAX_VALUE, Long.MIN_VALUE)

REELLE ZAHLEN

Name	Wertebereich	Speicherbedarf
float	$[-3.4 \cdot 10^{38}, +3.4 \cdot 10^{38}]$	4 Byte (3+1)
double	$[-1.8 \cdot 10^{308}, +1.8 \cdot 10^{308}]$	8 Byte (6+2)

(Vergleiche Zahlendarstellung durch Mantisse/Exponent (Folie 47).)

Hier bietet das math-Package weitere Konstanten und Methoden:

- Float.MAX_VALUE etc., Float.POSITIVE_INFINITY, Float.NaN (not a number)
- Math.E ($e = 2.7\ldots$), Math.Pi ($\pi = 3.1415$)
- Funktionen Math.min, Math.max, Math.abs, Math.sin, Math.cos, Math.tan, Math.exp, Math.log, Math.sqrt, etc.

WAHRHEITSWERTE ("BOOLEAN VALUES")

- die auf Folie 66 eingeführten Wahrheitswerte sowie Operatoren werden in Java unterstützt.
- Wahrheitswerte werden implizit zur Auswertung von Bedingungen (wenn - dann, solange ...) benötigt.
- `boolean` ist aber auch ein Datentyp, mit dem solche Ergebnisse an Variablen gebunden werden können.

- Konstanten: true, false
- Operationen: "!" (Negation), "&" (Konjunktion), "|" (Disjunktion), "∧" (exklusive Disjunktion),
- "&&" und "| |" als Konjunktion bzw. Diskunktion mit "fauler Auswertung" ("lazy evaluation"): wenn das Ergebnis nach Auswertung des ersten Operanden bereits feststeht, wird der zweite nicht ausgewertet.
  Beispiel:
  Betrachten Sie die Formeln "!(b==0) & (a/b <1)" und "!(b==0) && (a/b <1)".

Einzelne Zeichen

Einzelne Zeichen ("Characters") werden durch den Datentyp "char" behandelt.

- Wertebereich: der gesamte Unicode-Zeichensatz (Buchstaben, Sonderzeichen... alles was im ASCII-Zeichensatz enthalten ist, und noch vieles mehr)
- Vergleiche: "==", "!=", "$>$", "$>=$", "$<$", "$<=$" (jeweils unicode-alphabetisch geordnet)
- Methoden: Character.toLowerCase(), Character.toUpperCase() etc.
- Konstanten: jeweils ein Zeichen, das in (einfache) Hochkommata eingeschlossen ist: Sei c eine Variable, die geeignet deklariert ist:

  c = 'a';

- Unicode-Zeichen lassen sich durch $\backslash\mathrm{u}xxxx$ erzeugen, wobei $xxxx$ eine Hexadezimalzahl mit 4 Hex-Stellen ist. Die Zahl "0" ist z.B. \u0030 (ASCII 48 = $3 \cdot 16$).
- Damit benötigt ein Char 2 Bytes.

Typkonvertierungen zwischen primitiven Typen

Bei Berechnungen müssen manchmal Werte verschiedener Datentypen verknüpft werden.

- einige (numerische) Typen bilden natürliche Teilmengenbeziehungen

$$\begin{array}{c} \text{byte} \rightarrow \text{short} \rightarrow \text{int} \rightarrow \text{long} \rightarrow \text{float} \rightarrow \text{double} \\ \uparrow \\ \text{char} \end{array}$$

- Pfeilrichtung: erweiternde Konvertierung; einfach und ggf. automatisch (z.B. Verknüpfungen `int1 + float1` oder Verwendung eines `byte` in einem `short`-Parameter)
- gegen Pfeilrichtung: einschränkende Konvertierung, muss explizit angegeben werden, kann den Wert verfälschen (schneidet ab)
- andere Konvertierungen zwischen primitiven Datentypen nicht erlaubt. Insbesondere gibt es keine Konvertierung zwischen `int` und `boolean`!

Explizite Konvertierung

Explizite Typumwandlungen können mit dem *Type-Cast*-Operator vorgenommen werden:

$(type)\ \ expr$ wandelt den Ausdruck $expr$ in einen Ausdruck des Typs $type$ um.

Beispiel:

```
public class Konvertierung{
    public static void main (String[] args){
    int a = 1;
    int b = 5;
    System.out.println(a/b);                 // gibt 0 aus (Integer-Div.)
    System.out.println((float)a/(float)b); // gibt 0.2 aus (Float-Div.)
  }
}
```


3.3.2 Variablendeklarationen, Zuweisungen und Ausdrücke

Das Variablenkonzept

Variablen spielen nicht nur in Java, sondern in vielen Programmiersprachen eine Rolle

- auch in theoretischen Konzepten: Aussagenlogik, First-Order-Logik, Lambda-Kalkül

Variablen in Programmiersprachen

- maschinennah: Werte werden einfach in Speicherzellen abgelegt
- höhere Programmiersprachen: Variablen sind “Namen” für Speicherplätze im abstrakten Modell
  - dieser Name ist nur in einem gewissen Bereich des Programms bekannt,
  - enthalten Werte (z.B. “ist an den Wert 3 gebunden”),
  - können i.a. verändert (“neu zugewiesen”) werden,
  - häufig: Angabe (*“Deklaration”*) notwendig, von welchem Datentyp eine Variable ist,
  - *Zuweisung* dann nur mit Werten des entsprechenden Typs.

VARIABLEN IN JAVA

In Java wird unterschieden, “zu wem eine Variable gehört”:

- zu einer Klasse (dann äquivalent als *Klassenvariablen* oder *Klasseneigenschaften* bezeichnet, siehe Folie 119).
- zu einer Instanz; d.h., sie beschreibt eine (veränderbare) Eigenschaft der Instanz (ist damit Teil der Instanzstruktur; äquivalent als *Instanzvariable* oder oder *Instanzeigenschaft* bezeichnet).
  In diesem Fall haben alle Instanzen einer Klasse eine eigene solche Variable.
  Bsp: Person.name
- Lokale Variablen, die temporär während eines kleinen Teils eines Ablaufs benötigt werden.

VARIABLENDEKLARATIONEN

- Variablen müssen vor ihrer Verwendung *deklariert* werden. Sie sind dann bis zum Verlassen den aktuellen *Blocks* bekannt und können verwendet (d.h., zugewiesen und gelesen) werden.

Syntax

Will man eine Variable `v` von einem Typ `t` deklarieren, schreibt man in Java einfach

```
t v;
```

(meistens zu Beginn eines Blocks, z.B. einer Klassen- oder Methodendeklaration)

- Programmablauf: Bei der Verarbeitung einer Deklaration einer Variablen mit einem der bisher beschriebenen Literaltypen wird genau der benötigte Speicherplatz angelegt.
- jede Zuweisung an die Variable verwendet dann diesen Speicherplatz um den Wert dort abzulegen.

VARIABLENZUWEISUNGEN

Einer Variablen `v` eines Literaltyps `t`, die an einer Stelle im Programm bekannt ist, kann mit

```
v = ausdruck;
```

ein Wert (der sich aus der Auswertung von `ausdruck` zum gegebenen Zeitpunkt ergibt) vom Typ `t` zugewiesen werden.

Eine *initiale* Wertzuweisung kann auch gleich bei der Deklaration mit

```
t v = ausdruck;
```

geschehen.

Bemerkung: Auf die Unterschiede bei der Zuweisung bei Literalen und Objekten wird später noch eingegangen (siehe Folie 152)

AUSDRÜCKE

- Ausdrücke zur Berechnung von Werten – zum Beispiel um diesen Wert einer Variablen zuzuweisen – werden aus Variablen, Konstanten, und Anwendung “passender” (d.h. *typkompatibler*) Operationen gebildet.
- eine Grammatik für arithmetische Ausdrücke wurde bereits gezeigt.
- Ausdrücke können auch Abfragen von Objekt-/Klassen-Eigenschaften oder -methoden (soweit diese einen Rückgabewert erzeugen) enthalten, z.B.
  p.name oder p.getName()
- Der *Typ* eines Ausdrucks (und seiner Ergebniswerte) ergibt sich aus der Anwendung der darin vorkommenden Operatoren (streng getypte Sprache).
- Die Auswertungsreihenfolge ergibt sich bei vollständig geklammerten Ausdrücken aus den Klammern, ansonsten aus der Operatorpriorität.

Inkrementierung und Dekrementierung von Zählern

Neben der Neuberechnung und Zuweisung `x = x + 1` von Variablen bietet Java kurze (und schnelle) Operationen:

- `x++` ("Postinkrement"): Ergebnis des Ausdrucks ist der vorherige Wert von x, dann wird x noch um 1 hochgezählt.
- `x--` ("Postdekrement"): analog
- `++x` ("Präinkrement"): erst wird x um 1 hochgezählt, bevor es irgendwo verwendet wird
- `--x` ("Prädekrement"): analog
- Wird häufig in Zählschleifen verwendet.

Beispiel

```
int x = 42;              // x = 42
int a = x++;             // a = 42, x = 43
int b = ++x;             // b = 44, x = 44
int c = (a++ + --b)      // c = 42 + 43 = 85, a = 43, b = 43
```


Operatorprioritäten

Art	Operator	Priorität
Zuweisung	=	0 (niedrig)
Boolesche Operatoren	|| nach && nach | nach ^ nach &	1...5
Vergleiche	== , !=	6
	$>, <, >=, <=$	7
Additive Ops	+,-	8
multiplikative Ops	*, /, %	9
unäre Ops	!, – (Vorzeichen), ++, −−	10
Typecast	(t)x	11
Methoden- oder Eigenschaftsaufruf	.	12 (hoch)

Wenn Operatoren derselben Priorität ohne Klammerung nebeneinander stehen, wird von links nach rechts ausgewertet (z.B. $3 + 10 - 8$).

Zuweisung als Operator

- Die Zuweisung "=" kann auch als *Operator* stehen:
  Der Ausdruck `v = 3` weist der Variablen `v` den Wert 3 zu und ergibt 3.
- Bei uneindeutiger Klammerung wird *in diesem Fall* von rechts nach links ausgewertet (links von einer Zuweisung darf nur eine Variable stehen):
  `a = b = c` ist äquivalent zu `a = (b = c)` und zu (klarer) `a = b; b = c;`.

Ausdrücke und Operatorprioritäten: Beispiel

Machen Sie sich klar, was die folgenden Ausdrücke tun (und von welchem Typ die Variablen x, y sein müssen):

- `x = (int1 < 1000 & char1 < 'i')`
- `x = (y = 5)` , `x = (y == 5)` , `x == (y == 5)` , `x == (y = 5)`
- `x = (y = 3 < int2 + 1)`

Hinweis: wenn jemandem Prä-/Postinkrement/-dekrement und Schachtelung von Gleichheitsausdrücken zu unübersichtlich ist, kann man es auch ignorieren und explizit die Operatoren ausschreiben.

3.3.3 Nicht-primitive Literal-Datentypen: Zeichenketten

- Strings (Zeichenketten) bestehen aus mehreren Zeichen
- String-Konstanten werden in *doppelten* Hochkommata (im Gegensatz zu Chars) angegeben: `String str = "eine Zeichenkette"`
- Strings sind *unterschiedlich* lang, benötigen also unterschiedlich viel Speicherplatz. Für eine als String deklarierte Variable kann sich der Speicherbedarf bei einer Zuweisung ändern.
- String ist ein *Referenztyp*: Mit der Deklaration einer String-Variablen wird noch kein Speicherplatz für den String belegt. Erst wenn ein der Variablen ein String zugewiesen wird, ist die Variable eine *Referenz* auf den Speicherbereich, wo der String liegt (und Speicherplatz belegt).
- Operatoren: "+" als Stringverkettung. Falls einer der beiden Operatoren ein String ist, wird alles als String aufgefasst.
- Weitere Operatoren ... siehe Bücher

Strings als Referenztypen: Illustration

```
public class Person{
   private String name;
   public void setName(String thename){
     name = thename;
   }
   :
}
```

Aufrufe:

- `Person p = new Person();`
  - legt eine neue Instanz der Klasse Person an
    (p ist selber eine Referenz auf diese Instanz; vgl. Folie 152)
  - "leeres" `name`-Attribut, wird mit dem Wert *null* initialisiert.
- `string s = "Meier";`
  - legt eine (lokale) Variable s an, die auf den irgendwo abgelegten String "Meier" zeigt
- `p.setName(s);`
  - lässt das `name`-Attribut von p auf den im Speicher abgelegten String "Meier" zeigen.

AUSGABE VON TEXT

- Die Methoden `System.out.print` und `System.out.println` geben primitive Datentypen und Strings auf die Standardausgabe aus.
- dabei hängt `println` am Ende der Ausgabe gleich noch einen Sprung in die nächste Zeile an.
- Beide Methoden sind "polymorph" (siehe später), d.h., je nachdem von welchem Datentyp das Argument ist, wird eine geeignete Implementierung aufgerufen.
- Man kann dabei auch einen auszugebenden Text erst durch Stringverkettung im Argument erzeugen.

  Dabei muss man allerdings aufpassen:
  - System.out.println("3 + 4 =" + 3 + 4) gibt "3 + 4 = 34" aus
  - System.out.println("3 + 4 =" + (3 + 4)) gibt "3 + 4 = 7" aus

3.3.4 Benutzerdefinierte Datentypen

... zum Beispiel ein Datentyp für rationale Zahlen (Zähler/Nenner).

- Prinzip der Kapselung: Datentyp soll seine interne Realisierung (Struktur und Algorithmen) verbergen, und nur über seine Methoden zugreifbar sein.
- wird also als Klasse implementiert.
- eine solche Klasse stellt nur einen Datentyp bereit
  - um Instanzen des Typs zu erzeugen
  - die selber Methoden anbieten (z.B. die Zahl als Dezimalzahl)
  - sowie *Klassenmethoden*, die nicht zu einer Instanz gehören, sondern zum *Umgang mit solchen Werten* dienen.

AUFGABE: HARDWARENAHE PROGRAMMIERUNG IN JAVA

Gegeben sei ein Mikrocontroller mit einer Schnittstelle, über die Sie 8-Bit-Zahlen schreiben und lesen möchten. Das Format der Schnittstelle sieht nun folgendermaßen aus:

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

Bit 0:	das “Enable” - Bit wird zur Übertragung auf 1 gesetzt und ist sonst 0.
Bit 1:	das “Read/Write” - Bit ist 0 wenn gelesen werden soll und 1 beim Schreiben.
Bit 2-3:	nicht näher beschrieben
Bit 4-7:	Datenbits

Ihnen stehen also nur 4 Bits zur Datenübertragung zur Verfügung. Bei der Übertragung von 8-Bit-Zahlen übertragen Sie zunächst die ersten vier Bits der Zahl und anschließend die letzten vier.

AUFGABE (FORTS.)

Beispiel zur Übertragung von 10010110:

00000000	Ruhe
10010011	Bit 0 ist auf 1 gesetzt, weil übertragen werden soll.
	Bit 1 ist 1 für Schreiben.
	Die Datenbits sind mit 1001 belegt, den ersten 4 Bits der zu übertragenden Zahl.
00000000	Ruhe
01100011	Bits 0, 1 wie oben.
	Die Datenbits sind mit 0110 belegt, den letzten 4 Bits der zu übertragenden Zahl.

Schreiben Sie nun eine Java-Klasse “Schnittstelle”, die eine Methode mit der Signatur

```
public void uebertragen(int wert) {...}
```

anbietet, die für einen Wert *wert* die 4 Binärzahlen auf dem Bildschirm ausgibt, die die Übertragung bewerkstelligen würden.

Verwenden Sie die folgenden (und auf der Web-Seite bereitgestellten) Rahmen Schnittstelle.java und Schnittstellentest.java.

AUFGABE (FORTS.)

Rahmen fuer die Aufgabe:

```
public class Schnittstelle{
  public void uebertragen(int wert){
    int toTransmit;
   // ersetzen Sie die folgenden Zeilen durch geeignete Berechnungen/Ausgaben:
    toTransmit = wert;
    printAsBinary(toTransmit);
  }

  public static void printAsBinary(int x){
    for (int i=7; i>=0; i--)
    { System.out.print(x/(int)(Math.pow(2,i)));
      x = (int)(x%Math.pow(2,i));
    }
   System.out.println();
  }
}
```

Aufgabe (Forts.)

Testprogramm:

```
public class Schnittstellentest{
  public static void main (String[] args){
    Schnittstelle s = new Schnittstelle();
    s.uebertragen(5);
    s.uebertragen(255);
    s.uebertragen(240);
  }
}
```


3.4 Klassen in Java

Wiederholung

- Organisation der Struktur und des Verhaltens durch *Klassen*
- Instanzen der Klassen bilden
  - Zustand
  - Verhalten der Instanzen
- Eigenschaften der Klasse sowie Operationen die zum Umgang mit den Objekten des Typs dienen, oder Informationen über die Klasse als ganzes bereitstellen.

3.4.1 Die Klassen-Deklaration

... setzt sich aus vielen Teilen zusammen.

- Name, Sichtbarkeitsangabe
- Deklaration von Eigenschaften
  - der Instanzen
  - der Klasse
  - jeweils mit Sichtbarkeitsangabe
- Deklaration von Methoden
  - der Instanzen
  - der Klasse
  - keine, eine, oder mehrere Konstruktormethoden, mit denen man Instanzen erzeugen kann
  - jeweils mit Sichtbarkeitsangabe und Parametern,

KLASSEN-DEKLARATION: EBNF

In aufbereiteter Form stellt sich die EBNF der Klassendeklaration wie folgt dar:

```
<Klassendeklaration> ::=
  <Sichtbarkeitsspez> ["final"] "class" <Bezeichner> <Klassendeklarationsrumpf>
<Sichtbarkeitsspez> ::= "public"|"protected"|"private"
<Klassendeklarationsrumpf> ::= "{" {<Klassendeklaration>}
                                   {<KlassenEigenschaftDeklaration>";"}
                                   {<InstanzEigenschaftDeklaration>";"}
                                   [<MainMethodeDeklaration>]
                                   {<KlassenMethodeDeklaration>}
                                   {<StatischerInitialisierungsBlock>}
                                   {<InstanzMethodeDeklaration>}
                                   {<KonstruktorMethodeDeklaration>}
                                "}"
<KlassenEigenschaftDeklaration> ::=
       <Sichtbarkeitsspez> "static" ["final"] <Datentyp> <Bezeichner>
<InstanzEigenschaftDeklaration> ::=
       <Sichtbarkeitsspez> ["final"] <Datentyp> <Bezeichner>
```

Klassen-Deklaration: EBNF (Forts.)

```
<KlassenMethodeDeklaration> ::=
       <Sichtbarkeitsspez> "static" ["final"] <Rueckgabedatentypspez>
          <Bezeichner> "("<Parameterspez>")" "{" <Methodenrumpf> "}"
<InstanzMethodeDeklaration> ::=
       <Sichtbarkeitsspez> ["final"] <Rueckgabedatentypspez>
          <Bezeichner> "("<Parameterspez>")" "{" <Methodenrumpf> "}"
<StatischerInitialisierungsBlock> ::= "static" "{" <Methodenrumpf> "}"
<MainMethodeDeklaration> ::=
       "public static void main(String args[])" "{" <Methodenrumpf> "}"
<KonstruktorMethodeDeklaration> ::=
       <Sichtbarkeitsspez> ["final"]
          <Bezeichner> "("<Parameterspez>")" "{" <Methodenrumpf> "}"

<Rueckgabedatentypspez> ::= "void" | <Datentyp>
<Parameterspez> ::= {<Datentyp> <Bezeichner>,}
<Methodenrumpf> ::= {<Anweisung> | <Variablendeklaration>}
<Variablendeklaration> ::= <Datentyp> <Bezeichner> ["=" <Ausdruck>] ";"
```


Instanz-Eigenschaften und -Methoden

... sind das was man von “Objektorientierung” erwartet:

- Man definiert, wie jede Instanz aussehen und sich verhalten soll.
- die Struktur – gegeben durch die vorhandenen Instanz-Eigenschaften – ist dieselbe für alle Instanzen. Die Werte der Eigenschaften werden natürlich dann für jede Instanz separat gesetzt und können unterschiedlich sein.

  Sei die Variable `i` an eine Instanz der betrachteten Klasse `cl` gebunden und `eig` eine Eigenschaft (die nach außen sichtbar ist). Dann ergibt der Ausdruck `i.eig` den Wert von `eig` von `i`.
- das Verhalten ist komplett dasselbe für alle Instanzen der Klasse `cl`. Damit existiert nur eine Implementierung.

  Sei `meth()` eine solche Methode. Dann lautet der Aufruf `i.meth()` (Rückgabewerte siehe unten).

SICHTBARKEITSANGABE

Elemente – also Eigenschaften und Methoden – können prinzipiell innerhalb von

- Methoden der Klasse selber
- Methoden von abgeleiteten Klassen
- Methoden anderer Klassen, die diese Klasse oder ihre Instanzen "kennen"

verwendet werden. Die Sichtbarkeit wird dabei wie folgt eingeschränkt (Kapselung interner Informationen):

public: in der Klasse selbst (also in ihre Methoden), in Methoden abgeleiteter Klassen, und für Aufrufer der Klasse sichtbar.

Sie bilden die nach außen sichtbare *Schnittstelle*.

protected: in der Klasse selbst und in Methoden abgeleiteter Klassen, und für Aufrufer der Klasse sichtbar. Methoden aufrufender Klassen können darauf nur zugreifen, wenn sie im selben *Paket* definiert sind.

private: nur in der Klasse selbst sichtbar.

EIGENSCHAFTEN ODER VARIABLEN?

Klasseneigenschaften/Instanzeigenschaften werden oft auch als "Klassenvariablen" bzw. "Instanzvariablen" bezeichnet.

- ist eine Frage der Sichtweise:
  - als "public" deklarierte Eigenschaften sind eher als Eigenschaften anzusehen (z.B. Person.name), da sie von außen verwendet werden;
  - als "private" deklarierte Eigenschaften erfüllen oft mehr die Funktion von Variablen (etwa später bei der Implementierung von Datenstrukturen und Algorithmen als Klassen)

"Echte" Variablen

Daneben gibt es (siehe später) noch *lokale Variablen*, die *innerhalb des Methodenrumpfes* (oder sogar noch in einem darin enthaltenen Block) deklariert werden.

FINAL-DEKLARATION

- Als final deklarierte Eigenschaften/Variablen dürfen nicht verändert werden (Konstanten). Sie müssen also bereits mit einem Wert initialisiert werden.
- als final deklarierte Methoden dürfe nicht überlagert werden (d.h., in keiner der abgeleiteten Klassen; siehe später). Damit kann man auf eine dynamische Suche der zu verwendenden Implementierung zur Laufzeit verzichten.
- von als final deklarierten Klassen dürfen im ganzen keine Subklassen abgeleitet werden.

ERGEBNIS- UND RÜCKGABEDATENTYP

Datentypdeklaration von Eigenschaften

- Bei der Deklaration von Eigenschaften muss der Datentyp angegeben werden.

Datentypdeklaration von Methoden

- Methoden können – als *Funktionen* einen Wert zurückgeben.
  Dies geschieht durch eine `return` *expr* -Anweisung.
- der Datentyp `t` dieses Wertes wird in der Methodendeklaration spezifiziert.
  Ein Methodenaufruf kann dann z.B. so aussehen:
  `t v = i.meth();`
- falls kein Wert zurückgegeben wird, wird void als “Rückgabetyp” spezifiziert (dann `return` argumentlos als Rücksprunganweisung).

Deklaration der formalen Parameter

Bei der Deklaration von Methoden muss der Methodenkopf eine Deklaration der Parameter die bei einem späteren Aufruf mitgegeben werden, enthalten.

- Diese Parameter heißen *formale Parameter*, im Gegensatz zu den nachher als Werte gegebenen *aktuellen Parametern*.
- Parameter stellen quasi "lokale Variablen" der Methode dar, die beim Aufruf initialisiert werden.
- innerhalb der Methode sind diese dann wie Variablen lesbar und können auch neue Werte zugewiesen bekommen.

Beispiel

```
public class Person{
     :
   public void setName(String thename){
     name = thename;
   }
     :
}
```

wird mit `p.setName(s)` aufgerufen, wobei `p` eine Variable vom Typ Person und `s` eine Variable oder Literal vom Typ String sein muss.

Signatur

- Die in der Klassendeklaration mitgegebenen Informationen werden als *Signatur* der Klasse bezeichnet. Damit wird die *Schnittstelle* der Klasse beschrieben, über die aufrufende Methoden mit Instanzen der Klasse (und auch der Klasse selber) kommunizieren können.
- betrachtet man eine einzelne Methode einer Signatur, wird die Deklaration der Parameter und des Rückgabetyps als *Methodensignatur* bezeichnet.

ÜBERLADEN

Es ist möglich, für einen Methodennamen *mehrere* Deklarationen *mit unterschiedlichen Signaturen* anzugeben. Dies wird als *Überladen* der Methode bezeichnet.

Beispiel

Man kann eine Klasse “Wecker” haben, die es zulässt, auf unterschiedliche Weise eine Alarmzeit zu spezifizieren:

- Datum mit Zeit (“an diesem Tag um diese Zeit”)
- Zeit (“jeden Tag um die gegebene Zeit”)
- Integer (“in 10 Minuten”)

```
public class Wecker {
        :
   public void setAlarm(time whentime, date whendate){ ... }
   public void setAlarm(time whentime){ ... }
   public void setAlarm(int minutes){ ... }
        :
}
```


KLASSENEIGENSCHAFTEN UND -METHODEN

Werden durch die static-Deklaration ausgezeichnet.

- Klasseneigenschaften sind an die Klasse gebunden. Sie sind in allen Instanzen sichtbar und können dort –mit globalem Effekt– geändert werden. Die Abfrage von aussen erfolgt über `cl.eig`. Sinnvoll z.B. für
  - die Anzahl der aktuell existierenden Instanzen,
  - Durchschnittswerte (oder sonstige Aggregationen) über alle Instanzen
- Klassenmethoden sind ebenfalls an die Klasse gebunden. Der Aufruf von aussen erfolgt über `cl.meth()`.

  Als Klassenmethoden kommen z.B. Operationen auf dem in der Klasse implementierten Datentyp in Frage.
- beide existieren auch, wenn keine Instanzen einer Klasse existieren.

  Insbesondere für Klassenmethoden bietet sich damit die Möglichkeit, einen Algorithmus in einer Klasse zu modellieren (siehe z.B. Folien 127, 332, 405). Von einer solchen Klasse werden keine Instanzen erzeugt.

Konstruktormethoden

... irgendwie muss man die Instanzen ja erzeugen.

- Für jede Klassendeklaration wird *automatisch* ein parameterloser *Defaultkonstruktor* definiert.
  - Bei der Klasse "Person" wurde keine Konstruktormethode definiert
  - neue Instanzen wurden mit `p = new Person()` erzeugt.

  Der Defaultkonstruktor erzeugt eine einfache Instanz,
  - Eigenschaften werden ggf. entsprechend dem in der Deklaration angegebenen Defaultwert initialisiert

Definition eigener Konstruktormethoden

Man kann zusätzlich eigene Konstruktormethoden (mit anderer Signatur) definieren ("Überladen des Konstruktors")

- der Name der Konstruktormethode muss mit dem Klassennamen übereinstimmen (hier also eine über kontextfreie Grammatiken hinausgehende Einschränkung)
- parametrisierte Konstruktoren
- parameterlosen Defaultkonstruktor durch einen selbstdefinierten ersetzen.

Deklaration einer eigenen, parametrisierten Konstruktormethode für $klasse$:

```
public class klasse {
    :
public klasse(type_1 par_1, ... ,type_k par_k) {
      /* Initialisierung auf Basis von par_1,...,par_k */
    }
}
```

- Aufruf: `new` $klasse$`(arg`$_1$`, ... arg`$_n$`)`
- Die Konstruktormethode ist weder eine Instanzenmethode, noch eine Klassenmethode!

DEFINITION EIGENER KONSTRUKTORMETHODEN: BEISPIEL

```
public class Person{
   private String name;
   public Person() { }                          /* Default-Konstruktor */
   public Person(String thename) { name = thename; } /* Konstruktor */
   public void setName(String thename){ name = thename; }
   public String getName(){ return (name); }
   public void printName(){ System.out.println(name); }
}
```

```
public class TwoPersons{
  public static void main (String[] args){
    Person p1 = new Person("Meier");
    p1.printName();
    Person p2 = new Person("Mueller");
    p2.printName();
  }
}
```


DIE "MAIN"-METHODE

Das Vorhandensein einer `main`-Methode macht eine Klasse *klasse* zur Applikationsklasse die direkt extern durch

java *klasse*

(argumentlos) aufgerufen werden kann.

- als "public static void" deklariert:

  public class *klasse* {

      :

    public static void main(String args[]) { ... }

      :

  }

  - sichtbar, Klassenmethode, kein Rückgabewert, Argumente dürfen fehlen

- normalerweise gehört main zu einer Klasse, die selber in der Modellierung keine eigene Rolle spielt, sondern nur zum Aufruf der Anwendung dient.
- zum Testen kann man "normale" Klassen mit einer main-Methode ausstatten.

DIE "MAIN"-METHODE MIT ARGUMENTEN

- Der Aufruf von `main` erfolgt von durch Aufruf von aussen (Kommandozeile)
- eventuelle in der Kommandozeile angegebene Argumente werden als Strings in dem als formaler Parameter angegebenen Feld bereitgestellt.

```
public class MainTest{
  public static void main(String argumente[]) {
     System.out.println(argumente[0]);
     System.out.println(argumente[1]);
  }
}

// Rufen Sie dieses Programm mit
//    java MainTest eins zwei
// auf
```


INITIALISIERUNG DER KLASSENVARIABLEN

- mit einfachen Werten bei der Deklaration
- Instanzvariablen werden durch den (ggf. selbst geschriebenen) Konstruktor initialisiert
- entsprechend gibt es *statische Initialisierungsblöcke* – diese werden *ausgeführt* wenn die Klasse erstellt (d.h., ihre Deklaration ausgewertet) wird.
  Mit ihnen kann man komplexere Berechnungen durchführen, um initiale Werte für Klassenvariablen zu berechnen.
- Statische Initialisierungsblöcke können alle bis dahin definierten Klassenvariablen und -methoden "ihrer" Klasse verwenden (und diejenigen anderer bis dahin deklarierter Klassen).

Klassenvariablen – Aufgabe

Erweitern Sie die Klasse “Person” folgendermassen:

- jede Person hat eine Eigenschaft “Einkommen”, die ihr monatliches Nettoeinkommen enthält
- passende Methoden `setEinkommen(float x)` und `getEinkommen()`,
- die Klasse “Person” besitzt eine Klassenvariable, die immer die Anzahl der bisher erstellten Personen angibt.
  (Hinweis: sie sollte von einem neu geschriebenen Konstruktor hochgesetzt werden).
- die Klasse “Person” besitzt eine Klassenvariable, die immer das Durchschnittseinkommen der existierenden Personen angibt. Implementieren Sie eine Klassenmethode, die das Durchschnittseinkommen ausgibt.
- Verwenden Sie wieder die `main`-Methode von “Persons” zum Testen.

KLASSEN OHNE INSTANZEN

Klassen ohne Instanzen sind sinnvoll, um einfach Verhalten zu “sammeln”.

Auf der folgenden Folie wird eine Klasse (ohne Besprechung der Details) gegeben, die dazu dient, Eingaben von der Tastatur zu lesen:

```java
import java.io.*;
public class KeyBoard{
    private static DataInput stdIn =
            new DataInputStream(System.in);
    public static int readInteger(){
        Integer result = null;
        try{ result = Integer.valueOf(stdIn.readLine()); }
        catch(IOException e){ }
        return result.intValue();
   }
    public static int  readInteger(String s){
        System.out.print(s);
        System.out.flush();
        return readInteger();
    }
    public static double readDouble(){
        Double result = null;
        try{ result = Double.valueOf(stdIn.readLine()); }
        catch(IOException e){ }
        return result.doubleValue();
    }
    public static double readDouble(String s){
        System.out.print(s);
        System.out.flush();
        return readDouble();
    }
    public static boolean readBoolean(){
        Boolean result = null;
        try{ result = Boolean.valueOf(stdIn.readLine());}
        catch(IOException e){ }
        return result.booleanValue();
    }
    public static boolean readBoolean(String s){
        System.out.print(s);
        return readBoolean();
    }
}
```


AUFRUFEN SOLCHER KLASSEN

Von solchen Klassen werden keine Instanzen gebildet, sondern einfach *klasse.methode* aufgerufen:

```java
public class KeyBoardTest{
  public static void main (String[] args){
    int i = KeyBoard.readInteger("Geben Sie eine ganze Zahl ein: ");
    System.out.println(i);
    double j = KeyBoard.readDouble("Geben Sie eine reelle Zahl ein: ");
    System.out.println(j);
  }
}
```

3.4.2 Beispiel für einen als Klasse implementierten Datentyp

Rationale Zahlen

public class Rational {

- bestehen aus einem Zähler und einem Nenner, beide ganzzahlig und nur intern sichtbar:

  protected int Nenner = 1;
  protected int Zaehler = 0;

- Zähler und Nenner können jeweils mit einer Methode, die mit einem `int`-Wert aufgerufen wird, gesetzt werden:

  public void setZaehler(int z) { Zaehler = z; }
  public void setNenner(int n) { Nenner = n; }

Jetzt könnte man bereits rationale Zahlen erzeugen:

```
Rational r = new Rational();
r.setZaehler(1);
r.setNenner(5);
```


Rationale Zahlen (Forts.)

- Schöner wäre aber gleich ein direkter Konstruktor mit Parametern:

  public Rational(int z, int n) { Zaehler = z; Nenner = n; }

- und eine Möglichkeit, ganze Zahlen direkt zu konvertieren:

  public Rational(int z) { Zaehler = z; Nenner = 1;}

  Damit kann man rationale Zahlen folgendermaßen erzeugen:

  Rational a = new Rational(1,5); // 1/5 = 0.2
  Rational b = new Rational(5); // 5

- Abfragemethoden für Zähler und Nenner gibt es auch:

  public int getZaehler() { return Zaehler; }
  public int getNenner() { return Nenner; }

- und man kann sich den Wert als float geben lassen:
  (explizite Konvertierung notwendig!)

  public float getValue() { return (float)Zaehler/(float)Nenner; }

Rationale Zahlen (Forts.)

Um jetzt damit auch noch zu Rechnen, werden *Klassenmethoden* definiert:

- Addition: $\frac{z_1}{n_1} + \frac{z_2}{n_2} = \frac{z_1 \cdot n_2 + z_2 \cdot n_1}{n_1 \cdot n_2}$
  und dann muss noch gekürzt werden.

  ```
  public static Rational add(Rational r1, Rational r2) {
      return new Rational(r1.Zaehler*r2.Nenner + r2.Zaehler*r1.Nenner,
                          r1.Nenner*r2.Nenner);    }
  ```

  Kürzen ist nicht so einfach ...

- Multiplikation:

  ```
  public static Rational multiply(Rational r1, Rational r2) {
      return new Rational(r1.Zaehler * r2.Zaehler, r1.Nenner * r2.Nenner);
  }
  ```

  Auch hier müsste noch gekürzt werden.

Die ganze Klasse auf einen Blick

```
public class Rational{
  protected int Nenner = 1;
  protected int Zaehler = 0;
  public Rational(int z) { Zaehler = z; Nenner = 1;}
  public Rational(int z, int n) { Zaehler = z; Nenner = n; }
  public void setZaehler(int z) { Zaehler = z; }
  public void setNenner(int n) { Nenner = n; }
  public float getValue() { return (float)Zaehler/(float)Nenner; }
  public int getZaehler() { return Zaehler; }
  public int getNenner() { return Nenner; }
  public static Rational add(Rational r1, Rational r2) {
     return new Rational(r1.Zaehler*r2.Nenner + r2.Zaehler*r1.Nenner,
                         r1.Nenner*r2.Nenner);
  }
  public static Rational multiply(Rational r1, Rational r2) {
     return new Rational(r1.Zaehler * r2.Zaehler, r1.Nenner * r2.Nenner);
  }
}
```

Testprogramm

Wie üblich eine Dummy-Klasse mit “main”-Methode:

```
public class RationalTest{
  public static void main (String[] args){
     Rational a = new Rational(1,5);
     Rational b = new Rational(5);
     System.out.println("a:  " + a.getValue());
     System.out.println("b:  " + b.getValue());
     Rational c = Rational.multiply(a,b);
     System.out.println("c:  " + c.getValue());
     System.out.println("c:  " + c.getZaehler() + "/" + c.getNenner());
     Rational d = Rational.add(a,c);
     System.out.println("d:  " + d.getValue());
     System.out.println("d:  " + d.getZaehler() + "/" + d.getNenner());
  }
}
```


ÜBUNGSAUFGABE: KOMPLEXE ZAHLEN

Schreiben Sie eine Klasse, die den Datentyp “Komplexe Zahl” implementiert:

- eine komplexe Zahl $a + ib$ kann als Paar von zwei Realzahlen (a, b) aufgefasst werden: (Real- und Imaginärteil)
  Re$(a, b) = a$, Im$(a, b) = b$
- Arithmetische Operationen:
  $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$
  $(a_1, b_1) * (a_2, b_2) = (a_1 a_2 - b_1 b_2, a_1 b_2 + a_2 b_1)$
- Polarkoordinaten:
  Betrag: abs$(a, b) = \sqrt{a^2 + b^2}$
  Winkel: phi$(a, b) =$ arctan(y/x)
- Vergleich: $(a_1, b_1) == (a_2, b_2)$ genau dann wenn $a_1 == b_1$ und $a_2 == b_2$.
- keine Vergleiche auf $<$ und $>$.

ZUSAMMENFASSUNG

Bis jetzt:

- Datentypen
- Grundlagen des Klassenkonzeptes
- keine wirklichen “Programme” in den Methoden, keine Klassenhierarchie

Ausblick

- Theorie:
  - Algorithmen (z.B. beim Kürzen von Brüchen)
  - Datenstrukturen
  - Objektorientierung als Modell, abgeleitete Klassen, Vererbung, Polymorphie
- Java:
  - imperative Konstrukte
  - interne Verarbeitung (Sichtbarkeit, Methodenaufrufe, Speicherverwaltung)
  - Klassenhierarchie, Vererbung

3.5 Die imperativen Konstrukte in Java

- bisherige *elementare* Anweisungen:
  - Variablenzuweisungen:

    ```
    <Anweisung>  ::= <Variable> "=" <Ausdruck> ";"
    ```

    (wobei `<Ausdruck>` funktionale Methodenaufrufe beinhaltet)
  - Methodenaufrufe ohne Rückgabewert:

    ```
    <Anweisung> ::=
          <Ausdruck> "." <Bezeichner>"("[ <Parametersequenz> ]");"
    ```

    wobei der Wert des ersten Ausdruck eine Klasse oder ein Objekt sein muss, und der zweite Bezeichner ein dafür anwendbarer Methodenname.
- Rücksprunganweisung

  ```
  <Anweisung> ::= "return" [<Ausdruck>] ";"
  ```

- Bedingte Anweisungen
- Schleifen: Bedingungsschleifen und Zählschleifen
- Blockbildung aus mehreren Anweisungen

Die IF-Anweisung

```
<Anweisung> ::= "if" "("<Bedingung>")" <Anweisung> ["else" <Anweisung>]
```

- `<Bedingung>` ist ein Boole'scher Ausdruck
- wird dieser zu $wahr$ ausgewertet, wird die erste Anweisung ausgeführt, ansonsten die zweite.
- besteht einer der `<Anweisung>`-Teile aus mehreren Anweisungen, muss er explizit zu einem Block zusammengefasst werden.

Beispiel

```
public class IfTest{
  public static void main (String[] args){
    int i = KeyBoard.readInteger("Geben Sie eine ganze Zahl ein: ");
    if (i>=100) System.out.println("ist mindestens dreistellig");
     else if (i>=10) System.out.println("ist zweistellig");
           else if (i>=0) System.out.println("ist einstellig");
                 else System.out.println("ist negativ");}}
```


Die SWITCH-Anweisung

```
<Anweisung> ::= "switch" "("<Ausdruck>")" "{"
                    {"case" <wert> ":"  <Anweisung> ["break;"]}
                    ["default"     ":"  <Anweisung>]
                "}"
```

In diesem Fall ist die EBNF-Darstellung zur Erklärung wenig geeignet.
Eine switch-Anweisung hat die Form

switch ($Ausdruck$) {
  case K_1 : Anweisung A_1
    :
  case K_n : Anweisung A_n
  default : Anweisung A_{n+1}
  }

DIE SWITCH-ANWEISUNG (FORTS.)

$\texttt{switch}$ $(Ausdruck)$ $\{$
  $\texttt{case}$ K_1 $\texttt{: Anweisung}$ A_1
    :
  $\texttt{case}$ K_n $\texttt{: Anweisung}$ A_n
  $\texttt{default : Anweisung}$ A_{n+1}
  $\}$

- $Ausdruck$ ergibt einen Wert vom Typ byte, short, int, oder char,
- in jedem Zweig ist K_i eine Konstante dieses Typs,
- Das Ergebnis e von $Ausdruck$ wird berechnet.
- Stimmt für einen Zweig j der Wert K_j mit e überein, wird die dortige Anweisung A_j sowie alle darauffolgenden A_{j+1} bis A_{n+1} ausgeführt.
- Stimmt für keinen Zweig j der Wert K_j mit e überein, so wird A_{n+1} ausgeführt.
- `break` unterbricht die Ausführung der aufeinanderfolgenden Anweisungen und springt direkt zum Ende des switch-Blocks.

BEDINGUNGSSCHLEIFEN

bis jetzt: nur lineare Programme und Auswahl.

while-Schleife

```
<Anweisung> ::= "while" "("<Bedingung>")" <Anweisung>
```

- `<Bedingung>` ist ein Boole'scher Ausdruck
- Zuerst wird dieser ausgewertet. Ist er $wahr$, so wird die `<Anweisung>` ausgeführt, und der Ablauf beginnt von neuem. Wenn die Bedingung nicht (mehr) zu $wahr$ ausgewertet wird, ist die Schleife beendet.

do-while

```
<Anweisung> ::=  "do" <Anweisung> "while" "("<Bedingung>")"
```

- In diesem Fall wird zuerst die `<Anweisung>` ohne vorherigen Test ausgeführt. Danach wie oben.
- sinnvoll, wenn die Bedingung erst ausgewertet werden kann, wenn Werte aus dem Schleifenrumpf vorliegen.

Beispiele zu Schleifen

- Countdown von 10 bis 0:

```
int i = 10;
while (i>=0)
  { System.out.println(i);
    i = i-1;
  }
```

- Warten bis ein Messwert eine bestimmte Grenze übersteigt

```
do {
     /* lese Messwert (int x) von Sensor */
   } while (x < 100) ;
/* mache weiter */
```


ZÄHLSCHLEIFEN

```
<Anweisung> ::=  "for" "(" [<VariablenInitialisierung>] ";"
                           <Bedingung> ";"
                           [<VariablenAenderung>] ")"
                        <Anweisung>
```

- `<VariablenInitialisierung>` ist dabei eine Folge von Variablenzuweisungen. Hier dürfen auch Variablen neu deklariert werden, die dann aber nur lokal zum *Schleifenrumpf* sind. Diese Anweisungen werden einmal am Anfang der Abarbeitung ausgeführt.
- Die Bedingung wird bei jedem Schleifendurchlauf zuerst geprüft.
- Solange sie wahr ist, wird erst die Anweisung im Schleifenrumpf, und dann die `<VariablenAenderung>` ausgeführt.
- Im Gegensatz zur Initialisierung darf die `<VariablenAenderung>` aber nur aus einer Komponente bestehen, die den *Schleifenzähler* modifiziert.

Beispiel

- Zählen von 1 bis 10:

```
for (int i = 1; i < 11 ; i++) System.out.println(i);
```

BEISPIEL

Ausgabe eines Byte als Binärzahl:

```
public class PrintBinary{
 public static void printAsBinary(int x){
  for (int i=7; i>=0; i--)
  { System.out.print(x/(int)(Math.pow(2,i)));
    x = (int)(x%Math.pow(2,i)); }
  System.out.println();
} }
```

```
public class PrintBinaryTest{
  public static void main (String[] args){
    int b = KeyBoard.readInteger("Geben Sie eine Zahl <256 ein: ");
    PrintBinary.printAsBinary(b);
} }
```


BREAK UND CONTINUE

Mit `break` und `continue` lässt sich die Abarbeitung von Schleifen noch weiter beeinflussen:

- Befindet sich ein `break` in der Schleife (z.B. innerhalb einer `if`-Anweisung), wird die Schleife verlassen (bei geschachtelten Schleifen wird nur die innere Schleife verlassen).
- Befindet sich ein `continue` in der Schleife wird die aktuelle Abarbeitung des Schleifenrumpfes beendet und die nächste Iteration begonnen (ggf. mit vorheriger Variablenaktualisierung oder erneutem Auswerten einer Bedingung).

Blockbildung

Blockbildung macht eine “große” Anweisung aus vielen kleinen:

```
<Anweisung> ::= "{" {<Anweisung>} "}"
```

- solche Blöcke sind insbesondere wichtig, wenn eine `<Anweisung>` innerhalb eines if-, switch-case-, oder Schleifenkonstrukts aus einer Sequenz von mehreren Anweisungen bestehen soll.
- ansonsten können Blöcke verwendet werden, um den Sichtbarkeitsbereich von Variablen einzuschränken.

Felder (Arrays)

Ein *Feld* ist eine (ein- oder mehrdimensional) indizierte einfache Datenstruktur. Sie besteht aus mehreren Einträgen desselben Datentyps.

Eindimensionale Felder (endliche Folgen)

```
<Felddeklaration> ::=  <Typ> "[]" <Bezeichner>
```

deklariert eine Arrayvariable deren Elemente von dem gegebenen Typ sind.

- Arraytypen sind Referenztypen.
- Speicher für die Werte wird erst mit der endgültigen Zuweisung belegt
  - `int[] lottozahlen = {5,7,17,35,42,43}`
  - `int[] lottozahlen = new int[6]`
- Indizierung mit Integern $0...k-1$ für anzugebendes k:
  `lottozahlen[0] = 5;     System.out.println(lottozahlen[4]);`

Erzeugen von k Zufallszahlen

```
import java.util.Random;     // Benutzen einer Java-Package

public class Zufallszahlen{
  public static int[] ziehen (int k){
    int [] die_zahlen = new int[k];
    Random rnd = new Random(); // Zufallszahlenzieher initialisieren
    for (int i=0; i<k; i++)
      die_zahlen[i] = Math.abs(rnd.nextInt());  // eine Zahl ziehen
    return die_zahlen;
  }
  public static int[] ziehen (int k, int max){
    int [] die_zahlen = ziehen(k);
    for (int i=0; i<k; i++)
      die_zahlen[i] = die_zahlen[i] % max + 1; //modulo-div
    return die_zahlen; // nun alle Werte zwischen 1 und max (einschl.)
} }
```

- Rückgabe: eine *Referenz* auf ein Feld von k Zufallszahlen

Zufallszahlen

- Jedes Feld `f` hat neben seinem Inhalt ein read-only-Attribut `f.length`, das angibt wieviele Elemente es enthält.

```
public class ZufallTest{
  public static void main (String[] args){
    int k = KeyBoard.readInteger("Geben Sie eine ganze Zahl ein: ");
    int[] zahlen = Zufallszahlen.ziehen(k,100);
    for (int i=0; i < zahlen.length; i++)
       { System.out.print(zahlen[i]); System.out.print(" "); }
    System.out.println();
    } }
```

Lottozahlen

```
public class Lotto{
  int[] die_zahlen;
  public Lotto(int wieviele, int max){ // eigener Konstruktor
    die_zahlen = Zufallszahlen.ziehen(wieviele,max);
  }
  public void drucken(){
    for (int i=0; i < die_zahlen.length; i++)
       { System.out.print(die_zahlen[i]); System.out.print(" "); }
    System.out.println();
  }
}
```

```
public class LottoTest{
  public static void main (String[] args){
    Lotto my_lotto = new Lotto(6,49);
    my_lotto.drucken();
} }
```


Aufgabe

Erweitern Sie die Lottozahlen-Klasse um eine Methode

```
public int Vergleiche(int[] getippte_zahlen){...}
```

die eine Folge von getippten Zahlen mit den Lottozahlen vergleicht und zurückgibt, wieviele Zahlen übereinstimmen.

- Duplikate sind erlaubt, das Ergebnis soll ausgeben, wieviele Lottozahlen sie “erkannt” haben
  (d.h., für die Lottozahlen (1,4,9,16,16,25) würde der Tip (1,4,9,16,32,25) 6 Richtige ergeben, da alle Lottozahlen darin vorkommen),
- testen Sie mit unterschiedlich langen Sequenzen,
- Vergleichen Sie die Laufzeit bei 6,8,10,100,1000,... Lottozahlen
  Verwenden Sie dabei ein Testprogramm, das zufällig Tips generiert.

Kopieren von Referenztypen

- siehe eben: Arraytypen sind Referenztypen:
- Zuweisung kopiert nur die Referenz!

```
public class ArrayCopy{
  public static void main (String[] args){
    int[] feld = {5,7,17,35,42,43};  // legt Speicherbereich an
    int[] kopie = feld;   // Referenz auf denselben Speicherbereich
    kopie[2] = 18;
    System.out.println(feld[2]);
  }
}
```

- Alle Objekttypen sind Referenztypen

Aufgabe

Erweitern Sie ArrayCopy.java so, dass das Feld tatsächlich –elementweise– kopiert wird.

Mehrdimensionale Felder

Mit `char schach[][]` kann z.B. ein zweidimensionales Feld erzeugt werden

- lies: `(char[])[]` – also ein eindimensionales Feld von eindimensionalen Feldern
- ```
  schach[0][0] = 'T'; schach[1][0] = 'S';
  schach[2][0] = 'L'; schach[3][0] = 'D';
  schach[4][0] = 'K'; schach[5][0] = 'L';
  schach[6][0] = 'S'; schach[7][0] = 'T';
  for (i=0; i<8; i++) schach(i,1) = 'B';
  ```
- ```
  schach[0][7] = 't'; schach[1][7] = 's';
  schach[2][7] = 'l'; schach[3][7] = 'd';
  schach[4][7] = 'k'; schach[5][7] = 'l';
  schach[6][7] = 's'; schach[7][7] = 't';
  for (i=0; i<8; i++) schach(i,6) = 'b';
  ```

Aufgabe:
Schreiben Sie eine Klasse, die ein Schachbrett so repräsentiert, und die eine Methode besitzt, die überprüft, ob die weiße Dame ('D') den schwarzen König ('k') bedroht.

Mehrdimensionale Felder

- Mehrdimensionale Felder sind -im Gegensatz zu manchen anderen Programmiersprachen- als Feld von Feldern gespeichert:
- zu lesen: `(int[])[]`
- etwas anderes als `int[4,8]` wie z.B. in Pascal
  - Größe beim Deklarationszeitpunkt fest,
  - rechteckig!
- Java-Felder sind nicht notwendigerweise rechteckig!
  man kann mit `new int[`k_i`]` die i-te Zeile definieren.

Beispiel:

```
int a[][] = { {0},
              {1,2},
              {3,4,5},
              {6,7,8,9} }
```

Machen Sie sich die Darstellung dieses Feldes im Speicher klar.

3.6 Interne Abarbeitung, Speicherverwaltung und Sichtbarkeit

- Programmablauf: Die Abarbeitung des Programms durchläuft schrittweise den Bytecode der Methoden verschiedener Klassen (vgl. von-Neumann-Architektur, Folie 13)
- Der *Programmzähler* zeigt immer auf den gerade abgearbeiteten Befehl

⇒ springt zwischen verschiedenen Klassen (Aufrufe und Rücksprünge)

SPEICHERVERWALTUNG

Programm und Daten werden in zwei verschiedenen Bereichen des Speichers abgelegt (das ist nicht nur in Java der Fall):

- einen “Heap” (“Halde”):
  Zugriff beliebig (“assoziativ”, durch Referenzen)
  Klassen (Programm-Byte Code + Klassenvariablen), und Instanzen (Klassenzuordnung und Instanzvariablen), die mit new erzeugt werden.
- ein “Stack” (“Stapel”):
  Zugriff nach last-in-first-out-Prinzip: Methodenaufrufe: lokale Variablen, Aufrufparameter, Verwaltungsdaten (woher der Aufruf erfolgte).
  - Datenblätter der Klassen und Objekte (und Strings und Arrays) auf der Halde,
  - Datenblätter für Methodenaufrufe (“Aktivierungsrahmen”) auf einem Stapel.

INHALT DER “DATENBLÄTTER”

Klassen

- Name
- Bytecode
- Namen und Parameterangabe (Signaturen) der Methoden-Einstiegspunkte
- Datenfelder (fester Größe) für Klassenvariablen

Objekte

- ein Objekt ist implizit die Adresse, wo das Datenblatt liegt (um es referenzieren zu können)
  (abstrakt geht man von einer Objekt-ID aus)
- Klasse, wo es dazugehört (um Klassenvariablen und Methodenimplementierungen zu finden)
- Datenfelder (fester Größe) für die Instanzvariablen

Abstrakt kann man oft die auf der Halde liegenden, sich gegenseitig referenzierenden Objekte als *Graph* darstellen.

INHALT DER "DATENBLÄTTER"

Aufrufe

Bei einem Methodenaufruf ist der Programmablauf an einer bestimmten Stelle des Bytecodes der Klasse des aufrufenden Objektes. Eine Methode eines andere Objektes wird (ggf. mit Argumenten) aufgerufen.

Welche Daten müssen abgelegt werden?

- Rücksprungadresse (wo nach dem Methodenaufruf [im Bytecode der Klasse des aufrufenden Objektes] weitergerechnet werden soll),
- Adresse des aktuellen (aufgerufenen) Objektes ("this"),
- aktuelle Werte der formalen Parameter,
- Datenfelder (fester Größe) für die lokalen Variablen.

Beim Rücksprung wird der Rückgabewert kommuniziert, und der alte Programmzählerwert wieder geholt.

SICHTBARKEIT

Grundsätzlich ergibt sich daraus auch, welche Daten beim Programmablauf an einer Stelle "sichtbar", d.h., les- und schreibbar sind:

1. oberster Rahmen auf dem Stack (Aufrufparameter, lokale Variablen, außerdem das aktuelle Objekt und damit dessen Klasse(nvariablen)),

2a. das aktuelle Objekt ist als `this` zugreifbar,

2b. die Instanzvariablen/-methoden des aktuellen Objektes
(falls eine gleichnamige lokale Variable existiert, als `this.`<`variablen/methodenname`>),

3. die Klassenvariablen und -methoden der Klasse des aktuellen Objektes (ggf. ebenfalls mit `this` qualifiziert),

- Objekte auf der Halde – soweit das gegenwärtige Objekt Referenzen auf sie kennt,
- Klassenmethoden und -variablen aller Klassen (durch $klassenname.bezeichner$)

Die Reihenfolge (1)-(3) gibt an, in welcher Reihenfolge ein nicht qualifizierter Bezeichner (z.B. y) interpretiert wird (siehe Folie 170).

SICHTBARKEIT (FORTS.)

Anmerkung:

- Klassenvariablen sind sowohl in Klassenmethoden, als auch in Instanzmethoden sichtbar (unqualifiziert, oder mit `this`).
- bei geschachtelten Methodenaufrufen desselben Objektes sind die lokalen Variablen der vorherigen Aufrufe *nicht* sichtbar (Kapselung sogar zwischen Methoden desselben Objekts).
- Variablen in inneren Blöcke sind innerhalb des Blockes und innerhalb darin enthaltener Blöcke sichtbar (geschachtelte Schleifen).
- nicht erlaubt, Variablen in einem inneren Block nochmal zu deklarieren, die außen bereits deklariert wurden.

Beispiel: Programmablauf des “Lotto”-Beispiels von Folien 148 – 150.

BEISPIEL: SPEICHER- UND AUFRUFVERWALTUNG

Aufruf von `java LottoTest`.

Programmstart: Auf der Halde liegen die Klassen

- Random (und alles was in java.util.Random definiert ist),
- Zufallszahlen: Programmcode mit den beiden (Klassen)-Methoden `ziehen(int)` und `ziehen(int,int)`,
- Lotto: Programmcode mit dem Konstruktor `Lotto(int,int)` sowie der (Instanz)-Methode `drucken` (und evtl. weiteren),
- LottoTest: Programmcode mit der `main`-Methode.

Der Aufruf der `main`-Methode von `java LottoTest` erzeugt den ersten Eintrag auf dem Stack:

Rücksprungadresse:	Betriebssystem
aktuelles Objekt:	Klasse `LottoTest`
Aufrufparameter:	`args[]` ist ein NULL-Zeiger
Lokale Variablen:	`Lotto my_lotto = NULL`

`my_lotto` zeigt auf `NULL` – also auf Nichts.

Beispiel (Forts.)

Aufruf von `new Lotto(6,49)`:

- Rücksprungadresse: `LottoTest/main/Zeile1`
- aktuelles Objekt: Klasse `Lotto`
- Aufrufparameter: `wieviele = 6, max = 49`
- Lokale Variablen: keine

Rück:	Betriebssystem
Obj:	Klasse `LottoTest`
Params:	`args[] = NULL`
Vars:	`Lotto my_lotto = NULL`
Rück:	`LottoTest/main/Zeile1`
Obj:	Klasse `Lotto`
Params:	`wieviele = 6, max = 49`
Vars:	keine

Erzeugt ein Objekt auf der Halde:

obj_{0815}: `Lotto` (Referenz auf die Klasse)
`int[ ] die_Zahlen: NULL`

und springt in die erste Zeile der Konstruktormethode `Lotto(int,int)`.

Nächster Befehl:

`die_Zahlen = Lottozahlen.ziehen(wieviele = 6,max = 49)`

Beispiel (Forts.)

Aufruf von `Lottozahlen.ziehen(wieviele = 6,max = 49)`:

- Rücksprungadresse: Konstruktor `Lotto(int,int)/Zeile1`
- aktuelles Objekt: Klasse `Zufallszahlen`
- Aufrufparameter: `k = 6, max = 49`
- Lokale Variablen (der aufgerufenen Methode `ziehen(int,int)`): `int [] die_zahlen`

Rück:	Betriebssystem
Obj:	Klasse `LottoTest`
Params:	`args[] = NULL`
Vars:	`Lotto my_lotto = NULL`
Rück:	`LottoTest/main/Zeile1`
Obj:	Klasse `Lotto`
Params:	`wieviele = 6, max = 49`
Vars:	keine
Rück:	`Lotto/Lotto(int,int)/Zeile1`
Obj:	Klasse `Zufallszahlen`
Params:	`k = 6, max = 49`
Vars:	`int [] die_zahlen = NULL`

und springt in die erste Zeile der Methode `Zufallszahlen.ziehen(int,int)`.

Nächster Befehl:

`die_Zahlen = ziehen(k = 6)`

Dabei ist die Methode `ziehen(int)` des aktuellen Objekts/Klasse `Zufallszahlen` gemeint.

Beispiel (Forts.)

Aufruf von `Zufallszahlen.ziehen(k = 6)`:

- Rücksprungadresse: Konstruktor `Lotto(int,int)/Zeile1`
- aktuelles Objekt: Klasse `Zufallszahlen`
- Aufrufparameter: `k = 6`
- Lokale Variablen (der aufgerufenen Methode `ziehen(int,int)`): `int [] die_zahlen = NULL`

und springt in die erste Zeile der Methode `Zufallszahlen.ziehen(int)` mit dem Befehl `int [] die_zahlen = new int[6]`:

- Legt ein Feld mit 6 Integer-Plätzen

  irgendwo auf der Halde an und läßt `die_zahlen` darauf zeigen.

Rück:	Betriebssystem
Obj:	Klasse `LottoTest`
Params:	`args[] = NULL`
Vars:	`Lotto my_lotto = NULL`
Rück:	`LottoTest/main/Zeile1`
Obj:	Klasse `Lotto`
Params:	`wieviele = 6, max = 49`
Vars:	keine
Rück:	`Lotto/Lotto(int,int)/Zeile1`
Obj:	Klasse `Zufallszahlen`
Params:	`k = 6, max = 49`
Vars:	`int [] die_zahlen = NULL`
Rück:	`ZufZ./ziehen(int,int)/Zeile1`
Obj:	Klasse `Zufallszahlen`
Params:	`k = 6`
Vars:	`int [] die_zahlen =`

Beispiel (Forts.)

Nächste Befehle:

- zieht 6 Zufallszahlen und speichert sie in `die_zahlen[0-5]`:

5613	4	9999	8059	306	2003

- `return die_zahlen;`

Jetzt wird der neueste Aktivierungsrahmen des Stacks abgebaut:

- Rücksprung nach `Zufallszahlen/ziehen(int,int)/Zeile1`
- Mitnehmen (`return`) der Referenz auf das (in Sicherheit auf der Halde liegende) Feld.

Rück:	Betriebssystem
Obj:	Klasse `LottoTest`
Params:	`args[] = NULL`
Vars:	`Lotto my_lotto = NULL`
Rück:	`LottoTest/main/Zeile1`
Obj:	Klasse `Lotto`
Params:	`wieviele = 6, max = 49`
Vars:	keine
Rück:	`Lotto/Lotto(int,int)/Zeile1`
Obj:	Klasse `Zufallszahlen`
Params:	`k = 6, max = 49`
Vars:	`int [] die_zahlen = NULL`
Rück:	`ZufZ./ziehen(int,int)/Zeile1`
Obj:	Klasse `Zufallszahlen`
Params:	`k = 6`
Vars:	`int [] die_zahlen =`

Beispiel (Forts.)

Nach dem Rücksprung nach `Zufallszahlen./ziehen(int,int)/Zeile1`:

- Speichern der mitgebrachten Referenz auf

5613	4	9999	8059	306	2003

im lokalen `die_zahlen`

Nächste Befehle:

- `FOR`-Schleife: `die_zahlen` einzeln modulo 49

27	4	3	23	12	43

- `return die_zahlen` ... wie eben, Referenz mitnehmen.

Rück:	Betriebssystem
Obj:	Klasse `LottoTest`
Params:	`args[] = NULL`
Vars:	`Lotto my_lotto = NULL`
Rück:	`LottoTest/main/Zeile1`
Obj:	Klasse `Lotto`
Params:	`wieviele = 6, max = 49`
Vars:	keine
Rück:	`Lotto/Lotto(int,int)/Zeile1`
Obj:	Klasse `Zufallszahlen`
Params:	`k = 6, max = 49`
Vars:	`int [] die_zahlen =` •

Beispiel (Forts.)

Nach dem Rücksprung nach `Lotto/Lotto(int,int)/Zeile1`:

- Speichern der mitgebrachten Referenz auf

27	4	3	23	12	43

in der Instanzvariablen `die_zahlen` der auf der Halde liegenden `Lotto`-Instanz:

obj$_{0815}$: `Lotto` (Referenz auf die Klasse)
`int[ ] die_Zahlen:` •

- Rücksprung nach `LottoTest/main/Zeile1`; zurückgegeben wird der Zeiger auf obj$_{0815}$ als “Ergebnis” des Aufrufs der Konstruktormethode.

Rück:	Betriebssystem
Obj:	Klasse `LottoTest`
Params:	`args[] = NULL`
Vars:	`Lotto my_lotto = NULL`
Rück:	`LottoTest/main/Zeile1`
Obj:	Klasse `Lotto`
Params:	`wieviele = 6, max = 49`
Vars:	keine

Beispiel (Forts.)

Nach dem Rücksprung nach `LottoTest/main/Zeile1`:

- Zuweisung einer Referenz auf das auf der Halde liegende erzeugte Objekt obj$_{0815}$ an `my_lotto`

obj$_{0815}$: `Lotto` (Referenz auf die Klasse)
`int[ ] die_Zahlen:` •

27	4	3	23	12	43

Rück:	Betriebssystem
Obj:	Klasse `LottoTest`
Params:	`args[] = NULL`
Vars:	`Lotto my_lotto =` •

- Nächster Befehl: `my_lotto.drucken()`

Beispiel (Forts.)

Aufruf von `my_lotto.drucken()`:

- Aktivierungsrahmen auf dem Stack wie üblich.
- der Zugriff auf `die_zahlen` in `drucken()` greift auf die Instanzvariable `my_lotto` des aktuellen Objekts obj$_{0815}$ zu.

obj$_{0815}$: `Lotto` (Referenz auf die Klasse)
`int[ ] die_Zahlen:` •

27	4	3	23	12	43

Rück:	Betriebssystem
Obj:	Klasse `LottoTest`
Params:	`args[] = NULL`
Vars:	`Lotto my_lotto =` •
Rück:	`LottoTest/main/Zeile1`
Obj:	obj$_{0815}$ = •
Params:	keine
Vars:	keine

- Nach Beendigung der Methode `drucken()` sind Stack/Halde wieder so wie auf der vorhergehenden Folie.

Ein anderes Beispiel

```
public class Sinnlos {
  // sinnlos.java - ein abschreckendes Beispiel fuer Verdeckung von Variablen
  public int x = 10;
  private static int y = 20;

  public Sinnlos(int x){ this.x = x + 1; }
  public int get_y(){ return y; }
  public void set_y(int y){ this.y = y; } // aequivalent: Sinnlos.y (classvar)

  public int methode1 (int x, int a) {
     int y = 2;
     this.y = y + x + a;
     System.out.println("in m1 ist x " + x + ",y ist " + y +
                       ", this.y ist " + this.y + ", und a ist " + a);
     int tmp = methode2(this.y, y);
     System.out.println("am Ende von m1 ist y " + y  +
                       ", this.y ist " + this.y + ", und a ist " + a);
     return tmp;
  }
  public int methode2 (int x, int a) {
     System.out.println("zu Beginn von m2 ist x " + x + ", und a ist " + a);
     int b = 1;
     b = x * y;
     System.out.println("x ist " + x + ", y ist " + y +
                       ", a ist " + a + ", b ist " + b);
     int n = 0;    // (**)
     y = 3;        // Zugriff auf die Klassenvariable Sinnlos.y
     while (n<a)
     { int y = 5;  // dieses y ist nur innerhalb des Schleifenblockes bekannt
       b = b + y;
       System.out.println("in der Schleife ist y " + y  + ", und b ist " + b);
       n++; }
     System.out.println("nach der Schleife ist y " + y + ", b ist " + b);
     return b;
  }
  public static void main (String[] args){
    int choice = KeyBoard.readInteger("Was wollen Sie vorfuehren (1,2)?" );
    if (choice == 1)
    { Sinnlos object1 = new Sinnlos(1);
      Sinnlos object2 = new Sinnlos(2);
      System.out.println("1's y ist " + object1.get_y());
      object2.set_y(30);
      System.out.println("1's y ist jetzt " + object1.get_y());
    }
    else
    { int x = KeyBoard.readInteger("Geben Sie eine ganze Zahl ein:" );
      int y = KeyBoard.readInteger("Geben Sie eine ganze Zahl ein:" );
      Sinnlos my_object = new Sinnlos(x);
      int z = my_object.methode1(my_object.x,y);
      System.out.println("am Ende ist z " + z);
    }
  }
}
```


Beispiel (Forts.)

- erster Ast in “main”:
  die Klassenvariable y wird verändert und von beiden Instanzen gesehen
- zweiter Ast in “main”:
  Sichtbarkeitsbeispiel.

Aufgabe:

Verfolgen Sie den Inhalt der Halde sowie des Stacks bei dem Aufruf mit x=3 und y=4.
Der Speicherinhalt an der Stelle (**) wird auf der nächsten Folie angegeben

Beispiel (Forts.)

Interessant ist u.a. der Speicherinhalt an der Stelle (**)

- Halde:
  - Klasse "Sinnlos":
    Name, Bytecode, Signatur der Methoden und ihre Anfangspunkte im Bytecode.
    Klassenvariable $y = 9$.
  - Objekt "obj_1":
    Klasse: Sinnlos.
    Instanzvariable $x = 3$.
- Aufrufstack (von oben nach unten aufgebaut):
  Eingaben: x=3, y=4

main	(keine Rücksprungadresse) aktuelles "Objekt": Klasse "Sinnlos" Aufrufparameter: `args[] = NULL` (keine Aufrufparameter) lokale Vars: x=3, y=4, z=undef, choice=2, my_object: Ref. auf obj_1
m1	Rücksprungadresse in main aktuelles "Objekt": obj_1 Parameter: x=3, a=4 lokale Vars: y = 2, tmp = null
m2	Rücksprungadresse in m1 aktuelles "Objekt": obj_1 Parameter: x=9, a=2 lokale Vars: b=81, n=0,

Beispiel (Forts.)

- Bei (**) existiert keine lokale Variable y.
- Damit wird in der darauffolgenden Zeile gesucht, was y ist:
  - keine lokale Variable
  - keine Instanzvariable (von obj_1)
  - aha - es existiert eine so benannte Klassenvariable
  - (wenn in der Zeile "`int y`" stünde, wäre es eine lokale Variable)
- also wird die Klassenvariable verändert
- erst (und nur innerhalb) des darauffolgenden Blocks existiert eine lokale Variable y
- diese "verschattet" die Klassenvariable y.
  - Zugriff auf y verwendet dann die lokale Variable
  - die Klassenvariable wäre mit "`this.y`" oder klarer mit "`Sinnlos.y`" zugreifbar

Weitere Beispiele folgen, wenn rekursive Methodenaufrufe behandelt werden.

PROGRAMMZUSTAND

- Als (Programm)zustand bezeichnet man die aktuellen Werte der (sichtbaren) Variablen
  Beispiel:
  x=5, p="das Objekt ...", name="Meier", fertig=true
- Programmzustände betrachtet man z.B. beim
  - Debugging
  - Analyse (unbekannter Programme)
  - Programmverifikation (tut es das was es soll?)
    * Anfangszustand nach der Initialisierung
    * Endzustand – "Ergebnis"
    * Zwischenzustände – z.B. an bestimmten Stellen innerhalb einer Schleife oder einer Rekursion (siehe z.B. Folie 211)

PARAMETERÜBERGABE: CALL-BY-VALUE UND CALL-BY-REFERENCE

Beim Aufruf von Methoden mit Variablen als Argument muss unterschieden werden, ob die Parameter *Werte* enthalten, oder *Referenzen*:

- bei *call-by-value* wird der Wert übergeben. Änderungen während der Methodenausführung sind lokal.
  Immer bei primitiven Datentypen.
- bei *call-by-reference* bekommt die aufgerufene Methode eine *Referenz* auf ein bereits vorher existierendes Objekt. Änderungen am Objekt sind global.
  Immer bei Referenzdatentypen (z.B. Arrays; später auch Objekte).

Call-by-Value/Reference: Beispiel

```
public class cl {
  public static void meth(int my_x, int[] my_y) {
      my_x++;   my_y[0]++;
      System.out.println(x);
      System.out.println(y[0]);
      return; } }
```

Beispielumgebung:

```
int x = 5;
int[] y = {1,2,3};
cl.meth(x,y);         // Aufruf
     // gibt innerhalb der Methode  6  und  2  aus
System.out.println(x);       // gibt 5 aus
System.out.println(y[0]);    // gibt 2 aus
```

- Veranschaulichen Sie sich die Situation anhand des Aufrufstacks

Sichtbarkeit und Parameterübergabe: Kommentar

Die o.g. Unterscheidung ist meistens im Sinne des Benutzers sinnvoll:

- Werte primitiver Datentypen sind meistens als *Wert-Argumente* gedacht, um die aufgerufene Methode zu steuern

  printer.drucke_Uebungsblatt(anzahl_studenten);

  Will man die durchgeführten Änderungen in dem aufrufenden Ablauf auch sehen, muss man die Methode als Funktion definieren (muss ein `return`-Statement enthalten):

  preis = kalkulation.addiere_mehrwertsteuer(preis);

  Oder indem man die entsprechende Wrapper-Objekt-Klasse verwendet (siehe Folie 226).

- Komplexe Dinge werden an eine Dienstleister-Methode übergeben, um etwas mit ihnen zu tun:
  - Sortieren lassen eines Arrays: Array mitgeben, und es wird durch den Aufruf als "Seiteneffekt" sortiert.
  - Herumreichen eines komplexen Formular-Objektes in einer Büro-Workflow-Applikation, wobei jeder Einzelschritt Teile davon ausfüllt.

  sonst: Objekt vor dem Aufruf oder innerhalb der Methode kopieren (siehe Folie 325).

Speicherverwaltung

- Der Stack verwaltet sich offensichtlich selber
- Auf der Halde werden immer wieder neue Objekte explizit erzeugt.
  Kann man nicht benötigte Objekte auch wieder löschen?
  - In anderen Programmiersprachen (z.B. C++) muss man dies explizit tun – eine Buchführung, wie oft ein bestimmtes Objekt noch von anderen referenziert wird, ist unumgänglich.
  - Java hat einen automatischen "Garbage Collector", der nicht-referenzierte Objekte automatisch erkennt und löscht.
    Erleichtert das Leben, ist aber langsam (und in Grenzfällen unzuverlässig).

Programmierkurs für's erste beendet.

Konzepte:

- Objektorientierung; Klassen und Instanzen; Zustand und Verhalten; Methoden
- Imperative Konzepte: Verzweigungen, Schleifen
- Datentypen: primitive DT, Felder, Zeichenketten
- Stack/Halde-Abarbeitungsmechanismus

später noch:

- Klassenhierarchie/abgeleitete Klassen, Vererbung

Um Java und objektorientiertes Programmieren *richtig* zu verstehen, muss man damit arbeiten – am besten als Hiwi in einem größeren Projekt.

UND JETZT?

An dieser Stelle kann man mit zwei verschiedenen Themen weitermachen:

- Datenstrukturen:
  bisher primitive Datentypen, einfache Datenstrukturen (Arrays),
  Datenstrukturen sind komplexe Datentypen, die ein eigenes Verhalten (= Algorithmen) besitzen.
- Algorithmen: ggT usw.
  Die sollte man also zuerst behandeln.

Kapitel 4
Algorithmen

- Rationale Zahlen vervollständigen (vgl. Folie 130)
- ggT (größter gemeinsamer Teiler)
- funktionale Spezifikation
- Rekursion
- Umsetzung Rekursion in Iteration
- Sortieren (eines Feldes)
- Grundlagen für Laufzeitanalyse

Motivation: Kürzen von Brüchen

- Die Darstellung von rationalen Zahlen ist nicht eindeutig: 1/2 = 2/4 = 3/6 = 42/84 ...
- Es gibt eine *Normalform*: Zähler und Nenner dürfen keine gemeinsamen Teiler (Primfaktoren) mehr haben.
- "Anweisung": Nehme Zähler und Nenner und teile beide durch den ggT(z, n).

Ein Algorithmus zur Berechnung des ggT

Der folgende Algorithmus berechnet den ggT *rekursiv*:

$$\text{ggT}(x, y) = \begin{cases} x & \text{falls } x = y \\ \text{ggT}(x - y, y) & \text{falls } x > y \\ \text{ggT}(x, y - x) & \text{falls } x < y \end{cases}$$

Beispiel: ggT$(15, 9)$

4.1 Rekursion

ist ein häufiges Mittel zur Lösung großer Probleme:

- Rückführung (*Reduktion*) auf eine kleinere Instanz *desselben* Problems.
- Bekanntes Beispiel: Berechnung der Fakultät einer Zahl

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots (n-1) \cdot n$$

definiert durch $0! = 1$ und $n! = (n-1)! \cdot n$.

$$fak(n) = \begin{cases} 1 & \text{falls } n = 0, \\ fak(n-1) \cdot n & \text{sonst} \end{cases}$$

Eine rekursive Java-Methode

Klasse zur Berechnung von $fak(n)$:

```
public class FakultaetRek{
  public static long berechnen (int n){
     if (n == 0) return 1;               // Vergleich in Java mit == !
     else return(berechnen(n-1) * n);
  } }
```

```
public class FakultaetRekTest{
  public static void main (String[] args){
    int i = KeyBoard.readInteger("Geben Sie eine natuerliche Zahl ein: ");
    System.out.println(FakultaetRek.berechnen(i));
  } }
```

Aufgabe

- Vollziehen Sie die Berechnung für i=3 und i=8 nach
  ... bzw. man tut dies besser mit einem expliziteren Programm ...

Rekursion: Beispiel in Java

Betrachten Sie die folgende, äquivalente Java-Methode:

```
public class FakultaetRek2{
  public static long berechnen (int n){
     long k;
     if (n == 0) k = 1;
     else k = berechnen(n-1) * n;
     return k;
  } }
```

```
public class FakultaetRekTest2{
  public static void main (String[] args){
    int i = KeyBoard.readInteger("Geben Sie eine natuerliche Zahl ein: ");
    System.out.println(FakultaetRek2.berechnen(i));
  } }
```

- Vollziehen Sie die Entwicklung und den Inhalt des Java-Aufrufstacks für $i = 5$ nach.

REKURSION: BEISPIEL

"Fibonacci-Zahlen"

Anzahl der Kaninchenpaare auf Sardinien: Am Anfang gibt es kein Kaninchenpaar, im ersten Monat setzt jemand ein Paar aus; jedes Paar wird im zweiten Monat vermehrungsfähig und zeugt danach jeden Monat ein weiteres Paar.

Rekursionsgleichung:

$$\begin{array}{rcl} fib(0) & = & 0 \\ fib(1) & = & 1 \\ fib(i) & = & fib(i-1) + fib(i-2) \quad \text{für } i > 2 \end{array}$$

(Es gibt auch eine Definition mit $fib(0) = fib(1) = 1$, die dieselbe Zahlenfolge um eins verschoben ergibt, aber auf Folie 223 ein unansehnliches Ergebnis hat)

- doppelt rekursiv
- Es gibt –außer Kaninchen– noch andere Fälle, in denen Fibonacci-Zahlen auftreten.

Aufgabe

- Schreiben Sie eine Java-Klasse, die die Fibonacci-Zahlen berechnet.
- Vollziehen Sie die Berechnung für i=3 und i=5 nach,
- Vollziehen Sie den Inhalt des Java-Aufrufstacks nach.
- Lassen Sie sich bei jedem rekursiven Aufruf ausgeben, welche Fibonacci-Zahl berechnet wird. Was fällt auf?

REKURSION UND INDUKTION

... sind sehr eng verwandt:

- Basisfall=Rekursionsende/Induktionsanfang
- Rekursions-/Reduktions- oder Induktions-/Erweiterungsschritt

Beispiele

- induktive Definition von Termen oder Booleschen Formeln,
- induktive Definition ihres Wertes,
- rekursive Auswertung.

Interessanter Aspekt hier:

- Fakultät und Fibonacci-Zahlen sind induktiv *definiert*.
- ggT(a,b) ist *deklarativ* definiert als “die größte Zahl, die a und b teilt”, und kann rekursiv berechnet werden.

⇒ oft ist das Vorhandensein einer rekursiven Lösung eines Problems nicht so offensichtlich.

4.2 Eigenschaften von Algorithmen

- (partielle) Korrektheit:
  Tut der Algorithmus das was er soll?
  Berechnet FakultaetRek(n) wirklich $n!$?
  Ergibt die rekursive ggT-Definition tatsächlich den ggT?
- Terminierung:
  Endet der Algorithmus irgendwann?
- (partielle) Korrektheit + Terminierung = totale Korrektheit
- Effizienz:
  - Wieviele Schritte benötigt der Algorithmus (in Abhängigkeit von der Eingabe)?
  - Ist das optimal? Oder ist ein anderer Algorithmus schneller?

4.2.1 Korrektheit von Rekursiven Algorithmen

- Wenn die Rekursion direkt eine induktive Definition der Lösung repräsentiert, ist der Korrektheitsbeweis per Induktion trivial:
  - Basisfall/fälle: ✓
    $fak(0)$ = 1 =: 0!
  - Rekursionsschritt: ✓
    $fak(n) = fak(n-1) \cdot n$ ist nach Induktionsannahme für $n-1$ gleich $(n-1)! \cdot n =: n!$.
- Ansonsten muss man die Korrektheit induktiv (analog der rekursiven Lösung) beweisen.
  zum Beispiel beim ggT-Algorithmus.

Korrektheit des ggT-Algorithmus

$$\text{ggT}(x,y) = \begin{cases} x & \text{falls } x = y \\ \text{ggT}(x-y,y) & \text{falls } x > y \\ \text{ggT}(x,y-x) & \text{falls } x < y \end{cases}$$

- Rekursionsende $\Rightarrow$ Basisfall: ggT$(x,x) = x$.
- Rekursionsschritt: Sei o.B.d.A. (Symmetrie) $x > y$.
  Gezeigt wird: Die Menge der gemeinsamen Teiler von x und y ist dieselbe wie die Menge der gemeinsamen Teiler von $x-y$ und y:

$$\{z : z|x \wedge z|y\} = \{z : z|(x-y) \wedge z|y\}$$

"$\subseteq$" : Annahme: $z|x$ und $z|y$. Also gibt es r und s so dass $x = z \cdot r$ und $y = z \cdot s$, und damit $x - y = z \cdot (r - s)$, also $z|(x-y)$.

"$\supseteq$" : Annahme: $z|(x-y)$ und $z|y$. Also gibt es t und s so dass $(x-y) = z \cdot t$ und $y = z \cdot s$. Daraus folgt, dass $x = (x-y) + y = z \cdot (t+s)$, also $z|x$.

  – Die beiden betrachteten Mengen sind gleich, also auch deren größte Elemente.

- Also ist der Induktions-/Rekursionsschritt korrekt.
- beliebig (endlich) häufige Anwendung desselben führt also zum korrekten Ergebnis.

4.2.2 Terminierung rekursiver Algorithmen

Man muss zeigen dass die Rekursion für jede beliebige Eingabe in endlich vielen Schritten einen Basisfall erreicht

- Java: d.h., dass der Stack endlich groß wird, und dann über Rücksprünge wieder abgebaut wird.
- "Natürliche Induktion/Rekursion" $n \leftrightarrow n-1$ mit Basisfall 0 oder 1:
  trivial.
  Fakultät, Fibonacci ✓
- allgemeine Rekursion: einzeln zu beweisen.

Terminierung des ggT-Algorithmus

gegeben: $x, y \in \mathbb{N}$, o.B.d.A. $x > y$

Betrachte $z := x + y$. Es werden rekursive Aufrufe mit y und $x - y$ erzeugt, also z im nächsten Schritt kleiner. Außerdem sind beide Argumente des rekursiven Aufrufs > 0.
Damit können maximal $x + y$ solche Schritte erfolgen.

- In Wirklichkeit sind es in den meisten Fällen sehr viel weniger
- die obige Abschätzung wird nur dazu benötigt, um eben zu zeigen, dass es nach endlich vielen Schritten terminiert.
- was ist der "worst case" - also der Fall, wo es bezogen auf $x + y$ am längsten braucht? Wieviele Schritte sind es in diesem Fall?

Optimierung des ggT-Algorithmus

- Man betrachte die Berechnung des ggT(81,15).
- 81-15, (81-15)-15, ((81-15)-15)-15, ... bis das Ergebnis kleiner als die abgezogene Zahl ist.
- also kann man auch gleich, falls $x > y$ ist, $(y, x \text{ mod } y)$ als Argumente des rekursiven Aufrufes verwenden.
- Abbruchkriterium: falls Modulo-Test = 0, dann war die kleinere Zahl der ggT.
- In dieser Form hat Euklid ca. 350 v.Chr. den Algorithmus ursprünglich formuliert.

Aufgaben

1. Implementieren Sie den Euklidischen Algorithmus in einer Klasse und nutzen Sie diese um die "Rational"-Klasse zu vervollständigen.
2. Implementieren Sie eine rekursive Methode – analog zu $n! = \Pi_{i=1}^{n} i$ – die $\sum_{i=1}^{n} i$ berechnet.
3. Beschreiben Sie textuell Algorithmen, die eine gegebene Folge ganzer Zahlen sortieren:
   (a) mit einem Rekursionsschritt von n auf $n-1$,
   (b) mit einem Rekursionsschritt von n auf $n/2$,
   (c) Hinweis: lassen Sie sich von der Zufallszahlen-Klasse eine Folge von 20 Zufallszahlen geben, und probieren es damit auf einem Blatt Papier aus.

4.2.3 Effizienz: Aufwandsanalyse

Was wird betrachtet?

- Rechenzeit
  - absolute Rechenzeit schlecht vergleichbar
  - Zählen "typischer" Rechenschritte
    - Anzahl Rekursionsschritte
    - Sortierverfahren:
      - Anzahl von Vergleichen zwischen zwei Elementen,
      - Anzahl von Vertauschungen oder Kopiervorgängen
- Speicherplatz

Welche Parameter muss man berücksichtigen?

- manchmal ganz einfach: $fak(n)$ benötigt immer n Rekursionsschritte, d.h. n Rekursionen.
- im Durchschnitt (average case)
  z.B. beim Sortieren: über alle Zufallsfolgen der Länge n
- im schlechtesten Fall (worst case)

Laufzeitanalyse des Fakultät-Algorithmus

```
public class FakultaetRek{
  public static long berechnen (int n){
     if (n == 0) return 1;              // Vergleich in Java mit == !
     else return(berechnen(n-1) * n);
  } }
```

Sei $T_{fak}(n)$ die Anzahl der Rekursionsaufrufe (und damit praktisch auch der Multiplikationen) zur Berechnung von $n!$.

Auch hier kommt man auf eine rekursive Funktion:

- $T_{fak}(0) = 1$
- $T_{fak}(n) = 1 + T_{fak}(n-1)$

Induktiv lässt sich jetzt ganz einfach zeigen, dass für alle n gilt $T_{fak}(n) = n$.

Man sagt "die Laufzeit des Algorithmus ist *linear* in n".

Laufzeitanalyse des Summe-Algorithmus

Analog kann $s(n) := \sum_{i=1}^{n} i$ in linearer Zeit berechnet werden (siehe Aufgabe weiter oben).

Wie man aber schon seit Gauss weiß, ist das nicht optimal:

1	2	3	...	n-2	n-1	n

- n gerade: $s(n) = n/2 \cdot (n+1)$
- n ungerade: $s(n) = ((n-1)/2 \cdot n) + n$.
  Umformung ergibt ebenfalls $(n \cdot (n+1))/2$

Beweis: Induktion.
Basis $n = 1$: $s(n) = 1 = 1/2 \cdot 2$.
Induktionsschritt: to do.

... erlaubt die Berechnung von $s(n)$ in einem Schritt, also in *konstanter* Zeit.
(wobei dabei nicht berücksichtigt wird, dass es eine Multiplikation, eine Division, und zwei oder drei Additionen sind)

Laufzeitanalyse des rekursiven Fibonacci-Algorithmus

Fibonacci-Rekursionsgleichung:

$$
\begin{aligned}
fib(0) &= 0 \\
fib(1) &= 1 \\
fib(i) &= fib(i-1) + fib(i-2) \quad \text{für } i > 2
\end{aligned}
$$

Laufzeit:

- $T_{fib}(0) = 1$ und $T_{fib}(1) = 1$
- $T_{fib}(n) = T_{fib}(n-1) + T_{fib}(n-2) + 1$
  (+1 für die Addition)

- Die Fibonacci-Zahlen (und ihr rekursiver Berechnungsaufwand) wachsen schneller als jede *polynomielle* Funktion.
- Sie wachsen – genauso wie Kaninchenpopulationen – *exponentiell*.
  (Beweis später)

Laufzeitanalyse – Systematik

Relevant ist die *Größenordnung* des Wachstums der Laufzeit:

- konstant
- linear
- polynomial
- exponentiell
- bestimmte Zwischenstufen
- asymptotische Analyse (für große n)
- nur der am stärksten wachsende Anteil wird berücksichtigt,
  z.B. bei $n^3 + n^2 + \log(n) + 5$ wird der n^2-Anteil für große n vernachlässigbar, die $\log n$ sowie +5 ebenfalls.
- konstante Faktoren spielen dabei keine Rolle
  (egal, ob n^2 oder $5n^2$)

O-NOTATION

- Größenordnung von Funktionen

Für "die Laufzeit $T(n)$ eines Algorithmus in Abhängigkeit von der Problemgröße n ist für alle genügend großen $n > n_0$ kleiner als $c \cdot n$, für geeignete Konstanten n_0 und c" (formale Def. siehe unten) sagt man kurz

$T(n)$ ist in $O(n)$.

Definition

Für eine Funktion g ist die *Klasse $O(g)$ von Funktionen* wie folgt definiert:

$$O(g) = \{f : \exists c \geq 0, n_0 \geq 0 : \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

d.h., alle Funktionen f, die asymptotisch nicht schneller wachsen als $c \cdot g$.

Beispiele

- $f(n) = 3n^2 + 6n + 7$ ist in $O(n^2)$, aber auch in $O(n^3)$, oder in $O(2^n)$.
- $f(n) = 5 \log n$ ist in $O(\log n)$, und in $O(\log^2 n)$, und in $O(n^x)$ für alle x – jede Polynomfunktion wächst asymptotisch schneller als der Logarithmus

O-NOTATION (FORTS.)

- Man schreibt auch kurz $f(n) = O(n^2)$.
- ist aber keine Gleichheitsrelation, da man dann auch $f(n) = O(n^3)$ hat, aber natürlich nicht $O(n^2) = O(n^3)$,
  was eine Gleichheit der Funktionenklassen bedeuten würde – man hat nur $O(n^2) \subsetneq O(n^3)$. Z.B. $f(n) = n^2 \cdot \log n$ ist in $O(n^3)$, aber nicht in $O(n^2)$.

Aufgaben

- Beweisen Sie: für $f_1(n) = O(s(n))$ und $f_2(n) = O(r(n))$ gilt
  - $f_1(n) + f_2(n) = O(s(n) + r(n))$
  - Was gilt in diesem Fall weiter, wenn $s(n) = O(r(n))$?
  - $f_1(n) \cdot f_2(n) = O(s(n) \cdot r(n))$
- Beweisen Sie $O(n^p \cdot \log^q n) \subset O(n^{p+x})$ für alle $x > 0$.
  (Hinweis: reduzieren Sie das Problem auf eine Aussage von der vorhergehenden Folie)
- Beweisen Sie, dass man die Basis des Logarithmus dabei nicht angeben muss, d.h., $O(\log_2 n) = O(\log_{10} n) = O(\log_e n)$.

O-NOTATION (FORTS.)

Die folgenden Klassen werden häufig vorkommen:

$O(1)$	konstant	Berechnung der Werte geschlossener Funktionen
$O(\log n)$	logarithmisch	Suche unter gewissen Bedingungen
$O(n)$	linear	Suche im ungünstigen Fall, Fakultät
		Syntaktische Analyse kontextfreier Sprachen (Programme)
$O(n \log n)$	$n \log n$	Sortieren
$O(n^2)$	quadratisch	
$O(n^3)$	kubisch	
$O(2^n)$	exponentiell	automatisches Beweisen in Prädikatenlogik, Optimierungsprobleme

- außerdem gibt es z.B. Algorithmen, die in $\log \log n$ oder $\log^2 n$ sind.

O-NOTATION (FORTS.)

- O-Notation gibt eine obere Schranke für die Laufzeit
- Untere Schranke analog Ω:
  Wenn es Konstanten c und n_0 gibt so dass $f(n) > c \cdot g(n)$ für alle $n > n_0$ gilt, schreibt man

$$f(n) = \Omega(g(n))$$

- Wenn für eine Funktion $f(n)$ sowohl $f(n) = O(g(n))$ als auch $f(n) = \Omega(g(n))$ (für zwei unterschiedliche c) gilt, schreibt man auch

$$f(n) = \Theta(g(n))$$

O-NOTATION (FORTS.)

- O-Notation für geschlossene Funktionen ziemlich einfach
- Aufwandsberechnungen sind aber meistens als Rekursionsgleichungen gegeben.

Weitere Beispiele in den folgenden Abschnitten ...

WEITERE PROBLEMKLASSEN

Neben den bisher betrachteten “deterministischen” Algorithmen und Komplexitätsklassen gibt es noch *nichtdeterministische* Problemklassen:

- “P” bezeichnet die Klasse aller in polynomieller Zeit ablaufenden Algorithmen, bzw die Klasse aller Probleme, für die solche Algorithmen bekannt sind.
- “NP” bezeichnet die Klasse der Probleme, die “nichtdeterministisch-polynomiell” lösbar sind:
  man “rät” eine (potentielle) Lösung und kann dann in polynomieller Zeit testen, ob es eine Lösung ist.
  - wenn es exponentiell viele Möglichkeiten gibt, die man jeweils in polynomieller Zeit generieren kann, ist das Problem deterministisch-exponentiell (EXP)
  - oft kann man das “raten” durch polynomielle *Heuristiken* unterstützen.
- offensichtlich ist $P \subset NP$. Ob $NP \subset P$ ist, weiß man nicht – vermutlich aber nicht.
- exponentiell und NP ist aber immer noch “besser” als “unentscheidbar” (Halteproblem).
- auch da gibt es in der Komplexitätstheorie verschiedene Zwischenklassen ...

LAUFZEITANALYSE – EIGENSCHAFTEN

- In polynomieller Zeit durchführbare Algorithmen sind noch praktikabel. Algorithmen, die exponentielle Zeit benötigen werden schon bei mäßig großem n undurchführbar.
- Die Klasse der Polynome ist gegen Addition, Multiplikation und Einsetzung abgeschlossen
- Entsprechend ist die Klasse der polynomiellen Algorithmen gegenüber Komposition, Iteration und Schachtelung abgeschlossen.
- Die polynomielle Laufzeitklasse hängt nicht vom Rechnermodell ab – jeder Schritt in einem der Modelle kann durch polynomiell viele Schritte in jedem der anderen Modelle simuliert werden.
- ... siehe Komplexitätstheorie.

Der rekursive Fibonacci-Algorithmus ist exponentiell. Gibt es etwas besseres?

- Algorithmenentwurf ist eine kreative Aufgabe.
- beliebig gute Programmierkenntnisse nützen nichts, wenn man einen suboptimalen Algorithmus ausgewählt hat.

4.3 Rekursion und Iteration

Rekursive Definition der Fakultät-Funktion:

$$fak(n) = \begin{cases} 1 & \text{falls } n = 0, \\ fak(n-1) \cdot n & \text{sonst} \end{cases}$$

- einfache Linksrekursion (analog gibt es auch Rechtsrekursion).
- andere Charakterisierung: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots (n-1) \cdot n$
- ist de facto auch das was tatsächlich durch den rekursiven Algorithmus berechnet wird (vergleiche Aufrufstack und Rückgabe der Ergebnisse des Programms auf Folie 186)
- zeigt direkt einen *iterativen* Algorithmus.

Berechne $n!$ durch:

- fange mit 1 an
- multipliziere mit 2, 3, 4, etc., bis n erreicht ist.
- ("Bottom-up"-Berechnung – im Gegensatz zu "top-down"-Ansatz der Rekursion)

Iterative Berechnung der Fakultät

Klasse zur iterativen Berechnung von $fak(n)$:

```
public class FakultaetIt{
  public static long berechnen (int n){
     int i = 1;
     long result = 1;
     while (i < n) { i++;
                     result = result * i; }
  return result;
  } }
```

```
public class FakultaetItTest{
  public static void main (String[] args){
    int i = KeyBoard.readInteger("Geben Sie eine natuerliche Zahl ein: ");
    System.out.println(FakultaetIt.berechnen(i));
  } }
```

- Laufzeit: offensichtlich wieder linear: n Schleifendurchläufe

Analyse des iterativen Fakultät-Algorithmus

Man betrachtet den Programmzustand an geeigneten Stellen (Aufruf für $n = 5$):

- z_1: Anfangszustand nach der Initialisierung
  z_1 = {i=1, result = 1}
- z_2, z_3,...: Zwischenzustände am Ende des Schleifenrumpfes:
  - z_2 = {i=2, result = 2}
  - z_3 = {i=3, result = 6}
  - z_4 = {i=4, result = 24}
  - z_5 = {i=5, result = 120}
- Endzustand – “Ergebnis”:
  z_* = {i=5, result = 120}

... und das kann man jetzt zur *Verifikation* des Algorithmus verwenden.

Korrektheit von iterativen Algorithmen

- genauso per Induktion ...
- ... über die Anzahl der Schleifendurchläufe.

Anstelle einer Induktionsvoraussetzung und eines Induktionsschrittes benutzt man:

- Schleifen-Vorbedingung: $n > i$ und $result = i!$
- Schleifeninvariante: $result = i!$
  Sie gilt immer an dem Punkt, wo ggf. die Schleife verlassen wird.
- Abbruchbedingung: $i = n$ (negierte Schleifenbedingung)
- Nachbedingung: $result = n!$

Zu zeigen ist:

- Die Vorbedingung impliziert die Gültigkeit der Schleifeninvariante
- Jeder Schleifendurchlauf (= Induktionsschritt) garantiert dass die Schleifeninvariante – wenn sie vorher gültig war – auch nachher gültig ist
- Die Schleifeninvariante zusammen mit der Abbruchbedingung impliziert die Nachbedingung.

Ersetzen von (einfacher) Rekursion durch Iteration

- Asymptotischer Aufwand bleibt derselbe
- tatsächliche Laufzeit aber kürzer, weil die Verwaltung des Aufrufstacks eingespart wird
- reiner Implementierungsaspekt
- Rekursion oft einfacher/natürlicher zu verstehen, und kürzer zu implementieren (Prototyping).

Aufgabe/Beispiel

In einer vorhergehenden Aufgabe sollte ein Algorithmus zum Sortieren eines Feldes durch Rekursion von n auf $n-1$ formuliert werden.

- Implementieren Sie diesen iterativ.
- Geben Sie seine Laufzeit an.

AUFGABE

Gegeben ist das folgende Programm:

```
public class Mystery {
  public static int compute(int n){ return computeInternal(n-1); }
  private static int computeInternal(int n){
     if (n == 0) return 1;
     return (2*n+1+computeInternal(n-1));
} }
// + MysteryTest.java zum Aufrufen
```

- Betrachten Sie den Aufruf `Mystery.compute(5)`.
  Geben Sie den Programmzustand (= die Werte der Variablen) sowie den Aufrufstack in dem zweiten Aufruf von `computeInternal` an.
- Was berechnet Mystery (als Funktion von n)?
- Beweisen Sie Ihre Aussage.
- Re-formulieren sie Mystery iterativ und beweisen Sie die Korrektheit mit Hilfe der Schleifeninvariante.

ITERATIVE BERECHNUNG DER FIBONACCI-ZAHLEN

Keine einfache Links/Rechtsrekursion, aber geht trotzdem:

- Der Aufrufbaum der rekursiven Fibonacci-Definition berechnet viele Werte mehrfach
- Wenn man diese zwischenspeichert, können sie von den nachfolgenden Aufrufen wiederverwendet werden

Aufgabe

Implementieren Sie den auf diese Weise veränderten rekursiven Algorithmus:

- Benutzen Sie eine array-wertige Klassenvariable, die beim ersten (äußeren) Aufruf initialisiert in der die berechneten Zahlen ablegt werden
- Bei jedem Aufruf wird nachgeschaut, ob der gesuchte Wert bereits gespeichert ist.
- falls nicht, wird er durch rekursive Aufrufe berechnet, im Feld abgelegt, und dann zurückgegeben.

Dieses Algorithmenprinzip wird als *Dynamische Programmierung* bezeichnet.

Fibonacci mit Rekursion und Dynamischem Programmieren (Lösung)

```
public class Fibonacci{
  private static long[] precomputed = null;   // Zahlen koennen gross werden
  public static long numberOfCalls = 0;
  public static long berechneRekursiv (int n){
     numberOfCalls ++;
     if ((n == 0) | (n == 1)) return n;
     else return (berechneRekursiv(n-1) + berechneRekursiv(n-2)); }
  public static long berechneRekursivDyn (int n){
     precomputed = new long[n+1];                             // von 0 bis n
     precomputed[0] = 0; // Basisfaelle ablegen
     if (n!=0) precomputed[1] = 1;
     numberOfCalls = 0;
     return berechneRekursivDynDoIt(n); }
  public static long berechneRekursivDynDoIt (int n){
     numberOfCalls ++;
    if (n==0 || precomputed[n] != 0) return precomputed[n];
    else
    { long tmp = berechneRekursivDynDoIt(n-1) + berechneRekursivDynDoIt(n-2);
      precomputed[n] = tmp;                          // und zwischenspeichern
      return tmp;
} } }
```


Fibonacci mit Rekursion und Dynamischem Programmieren (Forts.)

```
public class FibonacciTest{
  public static void main (String[] args){
    int i = KeyBoard.readInteger("Geben Sie eine natuerliche Zahl ein: ");
    System.out.println(Fibonacci.berechneRekursiv(i) + " in " +
       Fibonacci.numberOfCalls + " Aufrufen");
    System.out.println(Fibonacci.berechneRekursivDyn(i) + " in " +
       Fibonacci.numberOfCalls + " Aufrufen");
  } }
```

- Berechnungsaufwand: $2n$-1Aufrufe (der rechte Teilbaum des Aufrufsbaumes wird jeweils durch nachschauen “abgesägt”), also $O(n)$ (“linear”)
- Speicherbedarf: linear (Feld mit $n+1$ Elementen)

Iterative Berechnung der Fibonacci-Zahlen

- wieder “bottom-up”
- mit “Fenstertechnik”: man hält immer zwei Zwischenergebnisse

```
public class FibonacciIt{
  public static long berechnen (int n){
     if (n==0) return 0; if (n==1) return 1;
     int i = 1;
     long fib_i = 1;
     long fib_i_minus_1 = 0;
     while (i < n) {
       long next = fib_i + fib_i_minus_1;
       i++;
       fib_i_minus_1 = fib_i;
       fib_i = next;
     }
  return fib_i;
  } }
```


Iterative Berechnung der Fibonacci-Zahlen (Forts.)

- FibonacciItTest.java analog:

```
public class FibonacciItTest{
  public static void main (String[] args){
    int i = KeyBoard.readInteger("Geben Sie eine natuerliche Zahl ein: ");
    System.out.println(FibonacciIt.berechnen(i));
  } }
```

- linearer Aufwand
- konstanter Speicherbedarf (2 Speicherplätze)

Aufgabe

Beweisen Sie die Korrektheit des iterativen Fibonacci-Algorithmus.

Geht es noch schneller?

NEBENBEMERKUNG: REKURSION VS. INDUKTION

Kann man jeden rekursiven Algorithmus in einen induktiven umschreiben?

- Links-/Rechtsrekursion (Fakultät): einfach und naheliegend, da im Grunde *induktiv*, was Rekursion und Iteration ist.
- Doppelte/Mehrfache Rekursion:
  - Bei Fibonacci geht es durch die Umsetzung auf Dynamisches Programmieren bzw. die spezielle Fenstertechnik (da der eine Rekursionszweig den anderen enthält).
  - Im allgemeinen Fall geht es nicht.
- Man kann Rekursion ohne “explizite” Rekursion z.B. in Assembler mit Hilfe einer Schleife, die Berechnung + Aufrufverwaltung kombiniert *codieren*.
  Dennoch bleibt der *Algorithmus* in seiner Grundidee rekursiv.
  Probieren Sie das später bei Quicksort ...

4.4 Lösen von Rekursionsgleichungen

- Beweis per Induktion
- Raten und Ausprobieren von Lösungen bzw O-Abschätzungen
  (vgl. auch die Lösung zu $\sum_{i=1}^{n} i$)
- Mathematik (Analysis)
- bekannte Algorithmen-Schemata nutzen

Abschätzen und Lösen der Fibonacci-Rekurrenz

Rekursionsgleichung:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(i) &= fib(i-1) + fib(i-2) \quad \text{für } i > 2 \end{aligned}$$

- Naheliegend ist, dass $g(n) = 2^n$ eine obere Abschätzung ist.
- $fib(0) = 0 < 2^0$ und $fib(1) = 1 < 2 = 2^1$.
- Induktionsschritt: Einsetzen in die Rekursionsgleichung ergibt

  $$fib(i) = fib(i-1) + fib(i-2) < 2^{i-1} + 2^{i-2} < 2 \cdot 2^{i-1} = 2^i$$

  wobei man sieht, dass bei dem zweiten "$<$" die Abschätzung ziemlich großzügig ist.

Abschätzen und Lösen der Fibonacci-Rekurrenz

Ansatz mit Exponentialfunktion mit kleinerer Basis: $fib(n) = a^n$, wobei $a < 2$ eine Konstante ist. Einsetzen in die Rekursionsgleichung ergibt $a^n = a^{n-1} + a^{n-2}$ und weiter vereinfacht $a^2 = a + 1$.

Diese Gleichung hat die Lösungen $a_1 = (1+\sqrt{5})/2$ und $a_2 = (1-\sqrt{5})/2$.

Die allgemeine Lösung der Gleichung ist also $fib(i) = c_1(a_1)^n + c_2(a_2)^n$,

die sich aus $fib(0) = 0$ und $fib(1) = 1$ zu $c_1 = 1/\sqrt{5}$ und $c_2 = -1/\sqrt{5}$ ergeben.

(Hinweis: mit der alternativen Definition $fib(0) = fib(1) = 1$ erhält man andere Konstanten)

Es gilt:

$$fib(n) = 1/\sqrt{5}\left(\underbrace{\left(\frac{1+\sqrt{5}}{2}\right)}_{\sim 1.62}^{n} - \underbrace{\left(\frac{1-\sqrt{5}}{2}\right)}_{\sim 0.62}^{n}\right)$$

womit

$$fib(n) = \Theta\left(\left(\frac{(1+\sqrt{5}}{2}\right)^n\right) \quad \text{mit Konstanten } 1/\sqrt{5} \pm \varepsilon$$

Laufzeit der Fibonacci-Algorithmen

Die *Laufzeit* des iterativen Fibonacci-Algorithmus ist durch das zusätzliche “+1” in der Rekursionsgleichung höher:

- $T_{fib}(0) = 1$ und $T_{fib}(1) = 1$
- $T_{fib}(n) = T_{fib}(n-1) + T_{fib}(n-2) + 1$

Auswirkung auf die Laufzeit:

- + einen konstanten Faktor?
- + einen linearen Faktor?
- + einen exponentiellen Faktor?

Aufgabe

Geben Sie eine geschlossene Formel für die Laufzeit der rekursiven Fibonacci-Berechnung an.

Laufzeit der Fibonacci-Algorithmen

- Die *Laufzeit* des rekursiven Fibonacci-Algorithmus ist exponentiell,
- der iterative bottom-up-Algorithmus hat lineare Laufzeit,
- die direkte Nutzung der o.g. Gleichung hat *konstante* Laufzeit.

4.5 Algorithmen für Felder

- Felder sind –obwohl noch relativ einfach– ein sehr gutes Beispiel um verschiedene Probleme und Algorithmen zu untersuchen.

(vgl. Saake/Sattler: “Algorithmen und Datenstrukturen”; Kapitel 5)

Feld als Beispiel-Klasse

Definition einer *Klasse* “Feld”, die den fast-primitiven Datentyp “Array” (der kein eigenes Verhalten besitzt) anreichert.

Dieses Vorgehen ist in Java durchaus üblich – es gibt auch solche “Wrapper”-Klassen “Int”, “Long”, “Double” etc:

- kapseln Datentypen in einer objektorientierten Hülle
- erlauben *call-by-reference*
- sind dann – über die Klassenhierarchie – Instanzen der allgemeinsten Klasse “object”.

Rahmen der Feld-Klasse

```
public class IntegerFeld {
  private int[] my_array;
  // Konstruktoren
     // Initialisierung mit einem gegebenen Feld
     public IntegerFeld(int[] ein_Feld) { my_array = ein_Feld; }
     // Initialisierung mit einem zufaelligen Feld der Laenge k
     public IntegerFeld(int k) { my_array = Zufallszahlen.ziehen(k); }
     // Initialisierung mit einem zufaelligen Feld der Laenge k
     // und Werten bis max
     public IntegerFeld(int k, int max) {
        my_array = Zufallszahlen.ziehen(k,max);
     }

  // Weitere Methoden werden spaeter hier eingefuegt.
```

(bitte umblättern)

Rahmen der Feld-Klasse (Forts.)

```
    // Laenge zurueckgeben
    public int laenge(){ return my_array.length; }
    // ein Element zurueckgeben
    public int inhalt(int i){ return my_array[i]; }
    // ein Element aendern
    public void set(int i, int j){ my_array[i] = j; }
    // Feld ausgeben
    public void drucken(){
       for (int i=0; i < my_array.length; i++)
          { System.out.print(my_array[i]); System.out.print(" "); }
       System.out.println();
    }
  }
```

(Die gesamte Klasse ist unter IntegerFeld.java verfügbar)

4.5.1 Suchen in einem Feld

Um zu testen, ob eine gegebene Zahl in dem Feld enthalten ist, muss man es durchgehen:

```
     public boolean sucheLinear(int wert) {
       for (int i=0; i < my_array.length; i++)
          { System.out.print(".");
            if (my_array[i] == wert) return true; }
       return false;
     }
```

Laufzeit (Feld der Länge k):

- k falls der Wert nicht vorhanden ist
- $\leq k$ falls der Wert vorhanden ist
- average case: auf jeden Fall linear.

Aufgabe

Wie ist die durchschnittliche Laufzeit in folgendem Fall: Suche einer Zahl $i \in \{1, \ldots, 2n\}$ in einem Feld `a int[n-1]` der Länge n, in dem n paarweise verschiedene Zahlen `a[j]` $\in \{1, \ldots, 2n\}$ gespeichert sind.

Test

```
public class LinearsucheTest{
  public static void main (String[] args){
    int n = KeyBoard.readInteger("Wieviele Zahlen: ");
    int max = KeyBoard.readInteger("Maximalwert der Zahlen: ");
    IntegerFeld testfeld = new IntegerFeld(n,max);
    if (n<15) testfeld.drucken();
    int v = KeyBoard.readInteger("Welche Zahl suchen: ");
    System.out.println(testfeld.sucheLinear(v));
  }
}
```


SUCHEN IN EINEM GEORDNETEN FELD: BINÄRE SUCHE

Wenn das Feld - wie z.B. im Telefonbuch - nach der Größe (ggf. alphabetisch) geordnet ist, kann man *binäre Suche* anwenden:

1. wenn das Feld nur aus einem Element besteht, teste direkt; sonst
2. betrachte das mittlere Element
3. falls es größer als das gesuchte ist, suche *rekursiv* in der ersten Hälfte,
4. sonst suche *rekursiv* in der zweiten Hälfte

Laufzeit: in jedem Rekursionsschritt wird das Problem halbiert.
Also maximal $\log_2 k$ Schritte und Vergleiche $\Rightarrow$ *logarithmische* Laufzeit.

Binäre Suche – die Implementierung

```
public boolean sucheBinaer(int wert) {
  return sucheBinaerVonBis(wert, 0, my_array.length-1);
}
public boolean sucheBinaerVonBis(int wert, int von, int bis) {
  System.out.println("Suche im Bereich " + von + " " + bis
                     + ": " + my_array[von] + " ... " + my_array[bis]);
  if (von == bis) return (wert == my_array[von]);
  int mitte = (von + bis)/2;       // ganzzahlige div
  if (wert <= my_array[mitte])
       return (sucheBinaerVonBis(wert, von, mitte));
  else
       return (sucheBinaerVonBis(wert, mitte+1, bis));
}
```


Ist das Feld sortiert?

Jedes Feld enthält eine Instanzeigenschaft, die angibt, ob es sortiert ist:

```
public boolean sortiert = false;
```

Ansonsten werden Sortierverfahren später betrachtet.
Bis dahin nehmen wir an, dass eine Methode `Sortieren()` existiert.

Test

```
public class BinaersucheTest{
  public static void main (String[] args){
    int n = KeyBoard.readInteger("Wieviele Zahlen: ");
    int max = KeyBoard.readInteger("Maximalwert der Zahlen: ");
    IntegerFeld testfeld = new IntegerFeld(n,max);
    if (n<17) testfeld.drucken();
    if (!testfeld.sortiert) testfeld.Sortieren();  // wie ... spaeter
    if (n<17) testfeld.drucken();
    int v = KeyBoard.readInteger("Welche Zahl suchen: ");
    System.out.println(testfeld.sucheBinaer(v));
  }
}
```


4.5.2 Sortieren

BUBBLE SORT

(Sortieren durch Vertauschen)

- Man lässt größere Zahlen durch Tauschen nach “oben” steigen
- dies tut man so lange, bis es keine Vertauschungen mehr gibt.
- ziemlich naiv ...

Bubble Sort: Implementierung

```
public void BubbleSort() {
  boolean swapped; int temp;
  do {
    swapped = false;
    for (int i=0; i < my_array.length-1; i++) {
      if (my_array[i] > my_array[i+1]) {
        temp = my_array[i];
        my_array[i] = my_array[i+1];
        my_array[i+1] = temp;
        swapped = true;
      }
    }
  } while (swapped);
}
```

- Aufwandsanalyse? Terminierung?
- Korrektheit: keine Schleifeninvariante, aber $\neg$swapped garantiert Sortierung.

Selection Sort

Einfache rekursive Idee:

- n Zahlen sortieren, indem man die größte Zahl wegnimmt und den Rest $n-1$ Zahlen rekursiv sortiert:
- Gehe das Feld durch, suche die größte Zahl,
- vertausche sie mit der Zahl am Ende des Feldes
- und sortiere die ersten $n-1$ Zahlen rekursiv

Selection Sort: Implementierung

```
public void SelectionSort() {
  SelectionSort(0,my_array.length-1);
  sortiert = true;
}
public void SelectionSort(int von, int bis) {
  if (von == bis) return;
  int max = 0; int maxindex = bis;
  for (int i=von; i <= bis; i++)
    { if (my_array[i] > max)
      { max = my_array[i];
        maxindex = i; }
    }
  int temp = my_array[bis];
  my_array[bis] = my_array[maxindex];
  my_array[maxindex] = temp;
  SelectionSort(von, bis-1);
}
```


Selection Sort: Aufwandsanalyse

Rekursionsgleichung:

$$\begin{aligned}
T(n) &= n + T(n-1) \\
&= n + n - 1 + T(n-2) \\
&= n + n - 1 + n - 2 + T(n-3) \\
&= n + n - 1 + \ldots + 1 \\
&= \sum_{i=0}^{n} i \;=\; \frac{n \cdot (n+1)}{2} \;=\; O(n^2)
\end{aligned}$$

Hier kann man mit einer Datenstruktur, die die Suche nach dem größten Element effizienter macht, Verbesserungen erreichen. (Siehe Folie 434.)

AUFGABE

- Implementieren Sie eine iterative Variante von Selection Sort für die Klasse **IntegerFeld**; (Testklasse siehe unten)

```
public class SelectionSortTest{
  public static void main (String[] args){
    int n = KeyBoard.readInteger("Wieviele Zahlen: ");
    int max = KeyBoard.readInteger("Maximalwert der Zahlen: ");
    IntegerFeld testfeld = new IntegerFeld(n,max);
    testfeld.drucken();
    testfeld.SelectionSort();
    testfeld.drucken();
  }
}
```

- Beweisen Sie die Korrektheit mit Hilfe von Schleifeninvariante und Abbruchbedingung.

INSERTION SORT

Einfache rekursive Idee:

- Wenn man $n-1$ Zahlen sortiert hat, kann man die n-te an der geeigneten Stelle einfügen.

Laufzeit

Einfügen der n-ten Zahl in die bereits geordneten $n-1$ Zahlen:

- suchen, wo sie hingehört (binäre Suche),
- sie dort ablegen und alle Zahlen ab dieser Stelle um eine Stelle nach oben verschieben

Laufzeit:

$$n \cdot (\log n + O(n)) = O(n^2)$$

Anmerkung:

- beim Verschieben muss jedes nachfolgende Element angefasst werden.
- Auch hier kann man mit einer geeigneten *Datenstruktur* hat, in der das “Verschieben” effizienter erledigt werden kann, mit der Insertion-Sort-Grundidee Verbesserungen erreichen (Siehe Folie 446.)

MERGE-SORT

Einfache rekursive Idee:

- man teilt das Feld auf der Hälfte und sortiert jede der Hälften
- dann muss man nur noch zwei sortierte Folgen zusammenfassen.

Beispiel: Sortiere die Folge (6,2,8,5,10,9,12,1,15,7,3,13,4,11,16,14)

MERGESORT – DIE IMPLEMENTIERUNG

```
public void MergeSort() {
  MergeSort(0,my_array.length-1);
  sortiert = true;
}
public void MergeSort(int von, int bis) {
  if (von == bis) return;
  int mitte = (von + bis) / 2; // ganzzahlige div
  MergeSort(von, mitte);
  MergeSort(mitte+1, bis);
  MergeSorted(von,mitte,bis);
}
```

Laufzeitabschätzung

$$T(n) = 2 \cdot T(n/2) + T(\text{MergeSorted}(n/2 + n/2))$$

MergeSort – die Implementierung(Forts.)

```
public void MergeSorted(int von, int mitte, int bis) {
  // beide Haelften zusammenfassen
  int[] temp = new int[bis-von+1];
  int i = von; int j = mitte+1; int k = 0;
  while (i <= mitte & j <= bis)
  { if (my_array[i] <= my_array[j]) { temp[k] = my_array[i]; i++; }
      else                          { temp[k] = my_array[j]; j++; }
    k++;
  } // i oder j ist jetzt fertig -> Rest noch kopieren
  while (i<= mitte) { temp[k] = my_array[i]; i++; k++; }
  // falls j < bis, macht nichts, ist OK (alles groessere Elemente).
  // tmp[0..k-1] zurueckkopieren:
  for (int t = 0; t<k; t++) { my_array[von + t] = temp[t]; }
  }
```

Laufzeitabschätzung

$$T(\text{MergeSorted}(n_1 + n_2)) = n_1 + n_2$$

MergeSort – Laufzeitanalyse

- Die Problemgröße wird in jedem Rekursionsschritt halbiert
  $\Rightarrow \log_2 n$ Schritte
- Der Basisfall ist trivial: $O(1)$, n mal.
- Der Merge-Schritt ist linear
  auf jeder Rekursionsebene 2^k Merge-Probleme der Größe $n/(2^k)$, jedes Element wird dabei zweimal angefaßt (tmp-Feld).
- $\Rightarrow O(n \cdot \log n)$.

Rekursionsgleichung:

$$T(1) = 1 \qquad , \qquad T(n) = 2 \cdot T(n/2) + 2n$$

Anmerkung:

- Selection Sort und Insertion Sort sind *in-place*-Algorithmen – sie benötigen nur ein einziges Feld;
- Merge Sort ist kein in-place-Algorithmus sondern benötigt temporäre Felder in der Merge-Phase.

"DIVIDE & CONQUER"

"Teile und Herrsche" ist ein wichtiges Designprinzip bei Algorithmen:

- Löse a (signifikant) kleinere Teilprobleme der Größe $1/b$ des Ausgangsproblemes, (Sortieren der halb so großen Zahlenfolge)
- Füge die Ergebnisse in der conquer-Phase mit einem Teilalgorithmus der Laufzeit von $c \cdot n^k$ zusammen
(Füge zwei "parallele" sortierte Folgen zu einer zusammen).

Allgemeine Divide & Conquer-Rekursionsgleichung:

$$T(n) = a \cdot T(n/b) + c \cdot n^k$$

ANALYSE DER DIVIDE-AND-CONQUER-REKURSIONSGLEICHUNG

Rekursionsgleichung: $T(n) = a \cdot T(n/b) + c \cdot n^k$

Zuerst ein paarmal expandieren:

$$\begin{aligned} T(n) = \; & a \cdot (a \cdot T(n/b^2) + c \cdot (n/b)^k) + c \cdot n^k = \\ & a \cdot (a \cdot (a \cdot T(n/b^3) + c \cdot (n/b^2)^k) + c \cdot (n/b)^k) + c \cdot n^k = \ldots \end{aligned}$$

- Man erhält einen Baum mit Unteraufrufen bestimmter Größe.
- Die Hauptarbeit steckt zum einen in der Anzahl seiner Knoten (gegeben durch a und b), zum anderen in dem Conquer-Arbeitsaufwand ($c \cdot n^k$) der inneren Knoten.
- Der Gesamtaufwand hängt dann von der Verteilung des Aufwandes ab.

Ein Programm zur Veranschaulichung der Verteilung des Aufwandes innerhalb des Aufrufbaumes bei Divide-and-Conquer-Algorithmen finden Sie auf der Vorlesungs-Webseite:

- MasterTheorem.java
- MasterTheoremTest.java

DIE DIVIDE-AND-CONQUER-REKURSIONSGLEICHUNG

Rekursionsgleichung: $T(n) = a \cdot T(n/b) + c \cdot n^k$
(Annahme $n = b^m$ – für n zwischen Potenzen von b muss man beim Rechnen abschätzen)

$$\begin{aligned}\text{expandieren:}\quad T(n) = &\; a \cdot (a \cdot T(n/b^2) + c \cdot (n/b)^k) + c \cdot n^k = \\ &\; a \cdot (a \cdot (a \cdot T(n/b^3) + c \cdot (n/b^2)^k) + c \cdot (n/b)^k) + c \cdot n^k = \ldots\end{aligned}$$

Irgendwann ist der Basisfall mit $n/b^m = 1$ erreicht und man hat

$$T(n) = a \cdot (a \cdot (a \ldots aT(n/b^m) + c \cdot (n/b^{m-1})^k) + \ldots + c \cdot (n/b)^k) + c \cdot n^k$$

wobei $m = \log_b n$ die Anzahl der Rekursionsschritte bis zum Basisfall ist ($n = b^m$ ebenfalls einsetzen)

$$T(n) = a^m \cdot T(1) + a^{m-1} \cdot c \cdot b^k + a^{m-2} \cdot c \cdot \left(b^2\right)^k + \ldots + a^2 \cdot c \cdot \left(b^{(m-2)}\right)^k + a \cdot c \cdot \left(b^{(m-1)}\right)^k + c \cdot \left(b^m\right)^k$$

- von oben ($i = 0$) gesehen: auf der i-ten Ebene von oben hat man a^i Aufrufe,
- von unten ($j = 0$) gesehen: auf der j-ten Ebene von unten umfasst jeder Aufruf b^j Elemente,
- $a^i \cdot (c \cdot (b^{m-i})^k)$ ist also der summierte conquer-Aufwand auf der i-ten *inneren* Ebene.

LÖSUNG DER DIVIDE-AND-CONQUER-REKURSIONSGLEICHUNG

es ist $b^m = n$ und entsprechend $m = \log_b n$.
wissen: $(b^m)^k = b^{mk} = b^{km} = (b^k)^m$

$$T(n) = \underline{a^m} \cdot T(1) + a^{m-1} \cdot c \cdot b^k + \underline{a^{m-2} \cdot c \cdot b^{2k}} + \ldots + a^2 \cdot c \cdot b^{(m-2)k} + a \cdot c \cdot b^{(m-1)k} + c \cdot \underbrace{b^{mk}}_{=n^k}$$

Welcher Term der Summe (= welche Ebene) dominiert?

- $a = b^k$: keiner – alle Terme sind $c \cdot b^{mk}$ bzw. $d \cdot b^{mk}$,
  Auf jeder Ebene derselbe Aufwand, also alles zusammen (es sind $m = \log_b n$ Terme)
  $O(n^k \cdot \log_b n)$
  [Mergesort: $a = b = 2,\ k = 1 \ \rightarrow\ O(n \log n)$]
- $a > b^k$: für $m \to \infty$ (und damit $n \to \infty$) dominiert der erste Term (Basisfälle), die Zwischenebenen werden allerdings auch größer.
  $a^m > a^{m-i} \cdot b^{ik} > b^{mk} = n^k$
  mit $a = qb^k$: $a^m = q^m b^{km} = q^m n^k$, $a^{m-i} \cdot b^{ik} = q^{m-i} b^{m-ik} \cdot b^{ik} = q^{m-i} b^{mk} = q^{m-i} n^k$,
  also nur um einen Faktor größer, keine größere Potenz. $O(n^k)$ oder $O(n^k \log n)$?
- $a < b^k$: für $m \to \infty$ (und damit $n \to \infty$) dominiert der hintere Term (oberste Ebene), es werden allerdings auch immer mehr vordere (untere) Ebenen. Selbes Problem.

Lösung der Divide-and-conquer-Rekursionsgleichung

... also mathematisch weitermachen:

$$\begin{aligned} T(n) &= a^m \cdot T(1) + a^{m-1} \cdot c \cdot b^k + a^{m-2} \cdot c \cdot b^{2k} + \ldots + a^2 \cdot c \cdot b^{(m-2)k} + a \cdot c \cdot b^{(m-1)k} + c \cdot b^{mk} \\ &= a^m \cdot T(1) + \sum_{i=1}^{m} a^{m-i} \cdot c \cdot (b^i)^k \end{aligned}$$

Setze $d := \max(T(1), c)$ und man erhält

$$T(n) \leq d \cdot \sum_{i=0}^{m} a^{m-i} \cdot b^{ik} = d \cdot \sum_{i=0}^{m} \frac{a^m}{a^i} \cdot b^{ik} = d \cdot a^m \cdot \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$$

was wiederum eine "einfache" geometrische Folge ist (Analysis I).

Lösung der Divide-and-conquer-Rekursionsgleichung

$$T(n) = d \cdot a^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$$

1. $a > b^k$, also $b^k/a < 1$

Dann ist die Summe für $m \to \infty$ konvergent – gegen die Konstante $\frac{1}{1-(b^k/a)}$.
Für die Laufzeit ergibt sich also $T(n) = O(a^m)$.
$m = \log_b n$ war die Anzahl der Rekursionsschritte bis zum Basisfall, also ist $a^m = a^{\log_b n}$ die Anzahl der zu berechnenden Basisfälle und damit

$$T(n) = O(a^{\log_b n}) \ ,$$

was man noch weiter und ansehnlicher umformen kann:
$\log_b(a^{\log_b n}) = \log_b a \cdot \log_b n = \log_b(n^{\log_b a})$ (Logarithmusgesetze), also muss auch $a^{\log_b n} = n^{\log_b a}$ sein, und man hat

$$T(n) = O(n^{\log_b a})$$

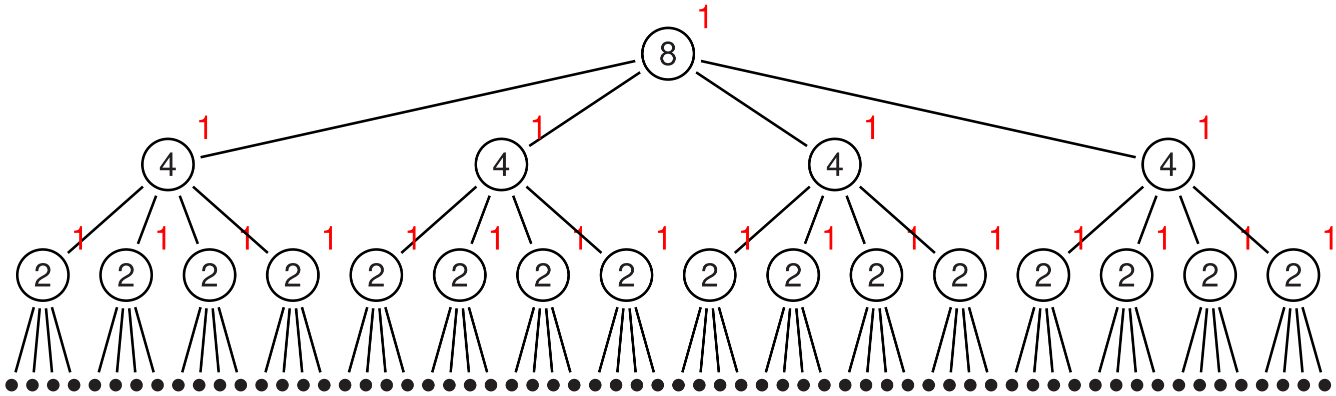
Dies ist der Fall, wenn der Aufwand von den Basisfall-Berechnungen dominiert wird.

Veranschaulichung Fall 1

$$T(n) = 4 \cdot T(n/2) + 1 \text{ für } n > 1 \qquad , \qquad T(1) = 1$$

Master-Theorem mit $a = 4$, $b = 2$ und $k = 0$, also $a > b^k$ und damit Lösung $O(n^{\log_b a})$, also in diesem Fall $O(n^{\log_2 4}) = O(n^2)$ (Anzahl der Basisfälle).

Größe und Anzahl der rekursiven Aufrufe für $n = 8$ (rot: Conquer-Aufwand):



- Es werden $64 = 8^2 = O(n^2)$ Basisfälle der Problemgröße “1” erzeugt,
- der interne (conquer-) Aufwand ist vernachlässigbar.

Beispiel: Geschlossene Form der Aufwandsberechnung

- $T(n) = a \cdot T(n/b) + c \cdot n^k$ ist eine rekursive Gleichung.

Manchmal ist man an einer *geschlossenen Form* als Funktion $T(n) = f(n)$ interessiert.

- Bsp. wie eben $T(n) = 4 \cdot T(n/2) + 1$, $T(1) = c = 1$ und damit $d = 1$
- Aufschreiben der Wertepaare $(n, T(n))$:
  (1,1), (2,5), (4, 21), (8,85), (16,341).
- Man betrachtet wieder $T(n) = d \cdot a^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$,
- Die Summe konvergiert für $m \to \infty$ gegen $\frac{1}{1-(b^k/a)}$, im Beispiel also gegen
  $1/(1-(1/4)) = 4/3$,
  d ist 1 und a^m ist $n^{\log_b a}$, also hier n^2.
- Gesamtaufwand konvergiert also gegen $\frac{1}{1-(b^k/a)} \cdot n^2 = 4/3 \cdot n^2$
- die genaue Lösung $T(n) = (4/3 \cdot (n^2 - 1)) + 1$ kann man dann raten und per Induktion beweisen.

Weitere Beispiele zu Fall (1)

- ähnlich zu eben, $T(n) = 4 \cdot T(n/2) + n$ für $n > 1$, $T(1) = 1$:
  Wieder $O(n^2)$ Aufwand, $O(n^2)$ Basisfälle, jetzt höherer Conquer-Aufwand, aber immer noch unter $O(n^2)$.

  Auch wenn jeder einzelne Basisfall sehr billig ist, dominieren sie immer noch durch ihre Anzahl gegenüber den conquer-Schritten (bei ... $+ n^2$ ändert sich das dann, siehe Fall (2)).

- $T(n) = 3 \cdot T(n/2) + O(n)$ ist $O(n^{\log_2 3})$:
  Es werden $3^{\log_2 n}$ ($= n^{\log_2 3}$) Basisfälle erzeugt.

- Sonderfall $a = b$, $k < 1$:
  $T(n) = O(n)$
  Es werden n Basisfälle erzeugt, die aber immer noch gegenüber den sehr einfachen Conquer-Schritten dominieren.

  Beispiel: Rekursive Maximum-Bestimmung ($a = b = 2$, $k = 0$).

  Machen Sie sich dies klar, indem Sie den Aufrufbaum für $n = 8$ aufmalen und jedem Knoten (= Basisfall oder Maximumbildung) den Aufwand 1 zuordnen.

Lösung der Divide-and-conquer-Rekursionsgleichung

$$T(n) = d \cdot a^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$$

2. $a = b^k$, also $b^k/a = 1$

Die Summe ist nicht konvergent, aber man summiert nur m mal, also

$$T(n) = d \cdot a^m \cdot m$$

Mit $m = \log_b n$ wie eben hat man (1) $a^m = n^{\log_b a}$ und (2) $m = O(\log n)$.

Diesmal kann man aus $a = b^k$ weiter schliessen dass $\log_b a = k$ ist und erhält

$$T(n) = O(n^k \cdot \log n)$$

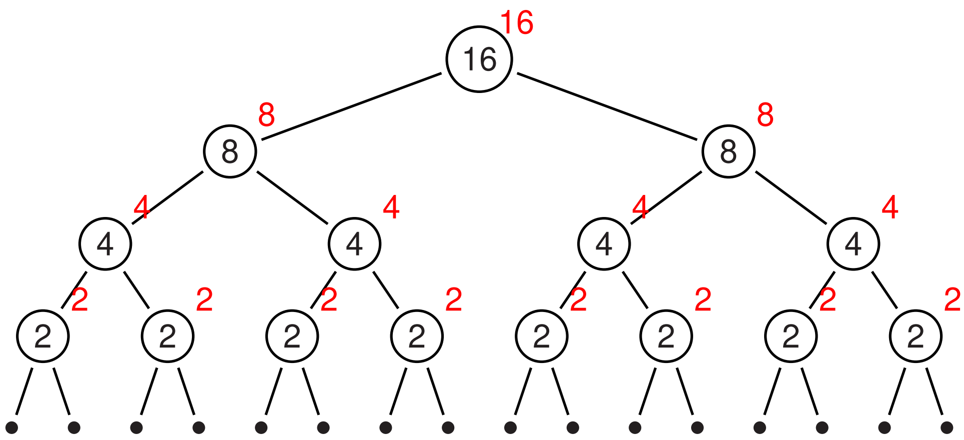
Dies ist der Fall, wenn jede der $\log_b n$ Schichten des rekursiven Aufrufbaumes denselben Aufwand $O(n^k)$ benötigt.

Im Fall eines linearen conquer-Schrittes ($k = 1$; wie bei Mergesort) also $T(n) = O(n \cdot \log n)$.

Veranschaulichung Fall (2)

$$T(n) = 2 \cdot T(n/2) + n \;\text{ für } n > 1 \qquad , \qquad T(1) = 1$$

Master-Theorem mit $a = 2$, $b = 2$ und $k = 1$, also $a = b^k$ und damit Lösung $O(n \cdot \log n)$.



Es ist auf jeder Ebene ein Aufwand von n, mit $\log_2 n + 1$ Ebenen,
also genaue Laufzeit $T(n) = n \cdot (\log_2 n + 1) = O(n \cdot \log n)$.

- Aufgabe: $T(n) = 4 \cdot T(n/2) + n^2$ für $n > 1$, $T(1) = 1$.

Lösung der Divide-and-conquer-Rekursionsgleichung

$$T(n) = d \cdot a^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$$

3. $a < b^k$, also $F := b^k/a > 1$

Die geometrische Summe divergiert (aber man benötigt nur endlich viele Summanden) und man hat

$$\sum_{i=0}^{m} F^i = \frac{F^{m+1} - 1}{F - 1} = O(F^m)$$

also insgesamt $T(n) = O(a^m \cdot F^m)$. Setzt man F wieder ein, erhält man $T(n) = O(a^m b^{km}/a^m) = O((b^m)^k)$. Erinnert man sich wieder dass $m = \log_b n$ die Anzahl der Rekursionsschritte bis zum Basisfall war, erhält man

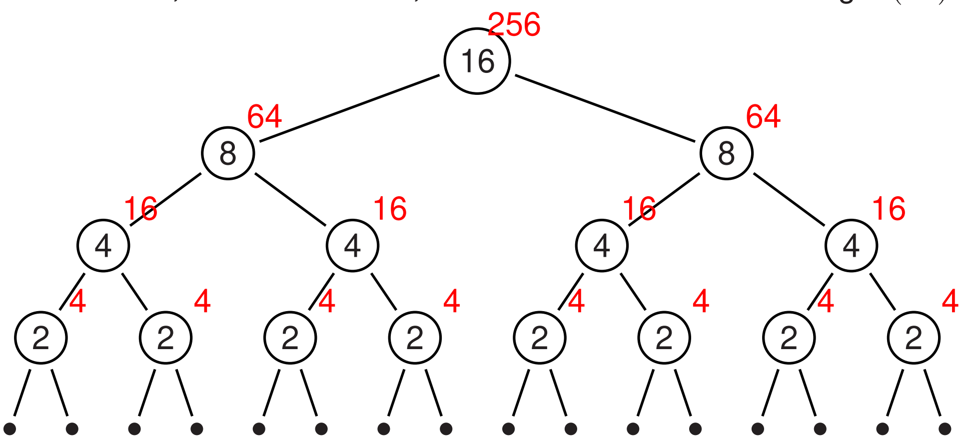
$$T(n) = O(n^k)$$

(d.h. der äußere conquer-Schritt dominiert - selbst die inneren conquer-Schritte für die jeweils kleineren Teilprobleme fallen nicht mehr auf)

Veranschaulichung Fall (3)

$$T(n) = 2 \cdot T(n/2) + n^2 \text{ für } n > 1 \qquad , \qquad T(1) = 1$$

Master-Theorem mit $a = 2$, $b = 2$ und $k = 2$, also $a < b^k$ und damit Lösung $O(n^2)$.



- Der Hauptaufwand liegt im obersten Conquer-Schritt: n^2
- nur n Basisfälle, der gesamte innere Aufwand des Baumes ist vernachlässigbar: Aufwand halbiert sich mit jeder Ebene
- $T(n) = 2 \cdot n^2 - n$ (raten anhand der Werte, Beweis per Induktion)

Quicksort

Bisher wurden zwei ineffiziente in-place-Algorithmen sowie ein effizienter nicht-in-place-Algorithmus betrachtet.

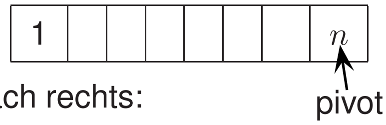
Gibt es einen effizienten in-place-Algorithmus?

- in-place: sofortiges Tauschen
- Effizienz: Divide & Conquer
- Mergesort: zwei “parallele” Teilfolgen; conquer-Schritt nach der Rekursion

Ansatz:

- *Zuerst* eine conquer-Verarbeitung um eine Hälfte mit größeren Zahlen und eine Hälfte mit kleineren Zahlen zu bekommen.
  Aber wie?
- Beide Hälften werden rekursiv bearbeitet und dann einfach aneinandergehängt.

QUICKSORT: AUFTEILEN IN 2 TEILPROBLEME



- Nehme das n-te Element als "*Pivot-Element*" p.
- bringe alle kleineren Elemente nach links, und alle größeren nach rechts:

1. Lasse einen "Zeiger" Z_1 von links (1) durch die Folge wandern, und einen anderen Zeiger Z_2 von rechts (n).
2. Z_1 wandert solange, bis ein Element e_1 gefunden wird, das größer als p ist (also nach rechts gehört). [Falls Z_2 auf Z_1 trifft, stop]
3. Dann wandert Z_2 solange, bis ein Element e_2 gefunden wird, das kleiner als p ist (also nach links gehört). [Falls Z_2 auf Z_1 trifft, stop]
4. vertausche beide Elemente [falls $Z_1 \neq Z_2$].
5. falls $Z_1 \neq Z_2$, mache weiter bei (2). Wenn sich die Zeiger treffen gilt:
   - alle Elemente links davon sind $\leq p$,
   - alle Elemente rechts davon sind $\geq p$.
   - das Element unter $Z_1 = Z_2$ ist $> p$.
6. Jetzt ist $Z_1 = Z_2$. Tausche das Element, auf das Z_2 zeigt mit p (Position n).

QUICKSORT: LÖSEN UND ZUSAMMENFÜGEN DER 2 TEILPROBLEME

Situation:

- Alle Elemente links von p sind $\leq p$,
- alle Elemente rechts von p sind $\geq p$,
- p ist bereits an der richtigen Stelle.

Komplettierung:

- Sortiere den linken und den rechten Teil separat.
- keine weiteren Schritte notwendig.

Aufgabe

Sortieren Sie die Folge 6,2,8,5,10,9,12,3,15,14,1,16,4,11,13,7.

QuickSort – eine Implementierung

```
public void QuickSort() {
  QuickSort(0,my_array.length-1);
  sortiert = true;      }
public void QuickSort(int von, int bis) {
  if (von >= bis) return;
  int pivotplace = QuickSortZerlegen(von,bis);
  QuickSort(von, pivotplace-1);
  QuickSort(pivotplace+1, bis);   }
public int QuickSortZerlegen(int von, int bis) {
  int pivot = my_array[bis];
  int l = von; int r = bis;
  while (l<r)
  { while (l<r && my_array[l] <= pivot) {l++;}
    while (r>l && my_array[r] >= pivot) {r--;}
    if (l != r) { //tauschen
     int x = my_array[l]; my_array[l]=my_array[r]; my_array[r]=x; }
  }
  // l=r, pivot an diese Stelle setzen
  int x = my_array[bis]; my_array[bis]=my_array[r]; my_array[r]=x;
  return r; }
```


QuickSort – Implementierungen

Es gibt viele in Details verschiedene Varianten von Quicksort:

- Man kann auch andere Elemente als Pivot nehmen
  (in Saake/Sattler ist eine Variante beschrieben, die das mittlere Element als Pivot nimmt)
- man kann die Zeiger aneinander vorbeilaufen lassen
  (geht nur soweit, dass zwischen r und l nur Element liegen, die gleichgross wie das Pivot-Element sind)
- Abfrage beim Vertauschungs-Schritt geeignet wählen
- Geeigneten letzten Schritt (ggf. Pivot unterbringen)

Quicksort: Aufwandsanalyse

Annahme: günstige Wahl des Pivots, dass beide Hälften gleich groß sind:

- n Schritte, $\leq n$ Vertauschungen bei der Vorverarbeitung
- 2 Teilprobleme der Größe $n/2$.
- Rekursionsgleichung: $T(n) = 2 \cdot T(n/2) + n$
- In der vorigen Formel: $a = b = 2$, $k = 1$, also Fall(2) und $T(n) = O(n \cdot \log n)$.

Wahl des Pivots

- Man kann z.B. die drei Elemente an den Positionen 1, n, und $n/2$ nehmen, und das mittlere Element der drei als Pivot verwenden.
- Worst Case-Analyse: man wählt jeweils das größte Element des Feldes als Pivot. Übungsaufgabe.
- Ein Fall für Average-Case-Analyse (siehe tiefergehende Bücher über Algorithmen und Datenstrukturen): ebenfalls $O(n \cdot \log n)$.

Übungsaufgabe: Stabilität

Ein Sortierverfahren ist *stabil*, wenn zwei Elemente, der Eingabefolge, die bezüglich des angewandten Vergleiches gleich groß sind, in der Ausgabefolge in derselben Reihenfolge erscheinen wie in der Eingabefolge.

Welche der angegebenen Verfahren sind stabil, bzw. stabil implementierbar (häufig kommt es auf Implementierungsdetails an - passen Sie die Algorithmen der Vorlesung ggf. an)?

Gibt es noch schnellere Sortierverfahren?

- Nein. Man kann zeigen (mit Entscheidungsbäumen), dass jeder *vergleichsbasierte* Algorithmus $\Omega(n \cdot \log n)$ benötigt.
- Doch. Unter gewissen Bedingungen/Zusatzinformationen.

Ein ganz anderer Ansatz: Radix Sort

Wie werden in einer Postzentrale Briefe sortiert?

- Man weiss, dass es in D Postleitzahlen von 00000 bis 99999 gibt
- Je ein Korb für 0xxxx, 1xxxx, ..., 9xxxx
- und nach Leipzig, Berlin, Hamburg, Hannover,..., München, Nürnberg schicken
- In Hannover kommt der Korb mit 30000 ... 39999 an
- und wird nach 31xxx, 32xxx, ... 39xxx sortiert und weiterverteilt.

Laufzeit: $O(5 \cdot n)$

- Wieder ein rekursives Schema: Rückführung der Sortierung von Zahlen mit n Stellen auf Zahlen mit $n-1$ Stellen.
- Setzt Wissen über den Wertebereich und die Struktur der "Sortierschlüssel" voraus
- benutzt keine $>$ und $<$-Vergleiche!
- ist nicht in-place

Sortierverfahren: Vergleich

	BubbleSort	SelectionSort	InsertionSort	MergeSort	QuickSort	RadixSort
Prinzip:	naiv schrittweise	Induktion	Induktion	Divide&Conquer Aufwand im Merge-Schritt	Divide&Conquer Aufwand im Divide-Schritt	Induktion nur unter best. Bed. anwendbar
# Schritte:	$O(n^2)$	n	n	$\log n$	$O(\log n)$	# Stellen =: k
Kompl. jedes Schrittes:	2	$O(n)$	$O(n)$	2n	n	n
Laufzeit:	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot k)$
in-place:	ja	ja	ja	nein	ja	nein

- Insertion Sort und SelectionSort: mit geeigneten Datenstrukturen anstatt des Feldes kann man den Aufwand jedes einzelnen Schrittes auf $O(\log n)$ reduzieren und erhält auch hier $O(n \cdot \log n)$-Algorithmen – siehe später.

4.5.3 Amortisierte Analyse

Wenn man in einem Feld suchen will, lohnt es sich, es vorher zu sortieren?

- $O(n \cdot \log n)$ Aufwand für Sortieren, nachher Suche in $O(\log n)$
- gegenüber Suche in $O(n)$
- lohnt sich bereits wenn man $\log n$ oft sucht.

⇒ Aufrechnen der vermutlich später angewendeten verschiedenen Operationen

- Häufig: Wahl einer geeigneten *Datenstruktur*, bei der Einfügen etc. etwas teurer ist, aber dafür Suchen, Aufzählen etc. billiger ist.

4.6 Ein kurzes Fazit

Das vorhergehende Kapitel war mit "Algorithmen für Felder" übertitelt.

- Nebenbei behandelte es "Algorithmen für Felder"
- Hauptsächlich ging es aber immer noch um Rekursion, Iteration und Aufwandsabschätzung.
- Oft gibt es für ein Problem viele verschiedene Algorithmen, die unterschiedlich "gut" sind (Sortieren, Berechnung der Fibonacci-Zahlen) und auf komplett unterschiedlichen Ansätzen basieren.
- es gibt noch viele weitere interessante Algorithmen die "zufällig" auf Feldern arbeiten.
- Felder sind einfache *Datenstrukturen*
- hier: in Java als *Klasse* realisiert, die *typisches, generisches Verhalten* sammelt

Nachteile von Feldern:

- sind statisch
- Man muss am Anfang wissen, wie groß sie werden
- Einfügen ist teuer

AUSBLICK

Weitere Schritte:

- Es gibt natürlich nicht nur Integer-Felder, sondern Felder über beliebigen Datentypen $\Rightarrow$ *Klassenhierarchie, abstrakte Klassen*
- Oft benötigt man *flexiblere Datenstrukturen*
- und *spezifischere, anwendungsorientierte Objekttypen* (Struktur, Verhalten)

Wir sind also zurück bei dem Punkt "(objektorientierte) Modellierung" von Datentypen und realen Objekt-Klassen, und anderen Dingen die man so braucht. Und das ist komplett unabhängig von Java als –zufällig gewählter– Programmiersprache.

DATENTYPEN

Einfache Datentypen (z.B. Datum, komplexe Zahlen):

- kommen nur als Werte von *Eigenschaften* vor.
- haben wenig (extern sichtbare) innere Struktur.
- auf ihnen sind Operatoren und Vergleiche definiert.
- haben kein eigenes aktives Verhalten.

... haben wir gesehen.

Kapitel 5
Objektorientierung

(vgl. Saake/Sattler: “Algorithmen und Datenstrukturen”; Kapitel 12)

- Vorgehensweise zur Beschreibung und *Modellierung* von Zuständen/Abläufen/Algorithmen
- Anfang der 90er: Objektorientierte Analyse/Design
  Abstrakte Beschreibung von Abläufen, nicht nur von Programmen.
- gegenwärtig weitest verbreiteter Formalismus:
  UML (Version 1.0 1997 bei der OMG (Object Management Group) standardisiert).
- Grundsatz: Wenn man ein Objekt “kennt”, also es identifizieren kann, und weiss, welche “Kommandos” es kennt, und welche Effekte diese Kommandos haben, genügt das. Man muss nicht unbedingt wissen, wie es intern aufgebaut ist.
- Objektorientierung ist also weit mehr als “nur” objektorientierte Programmiersprachen!
- Programmieraspekte im OMG/ODMG-Standard festgelegt (gilt auch für Java)

5.1 Klassen und Objekte der Anwendung

Objekte

- Objekte haben eine *Identität*
  - Identität ist i.a. unabhängig vom Wert der Attribute
    (z.B. Änderung des Namens einer Person ändert nicht die Objekt-Identität dieser Person)
  - damit Unterscheidung zu *Literalen* (Zahlen, Zeichenketten)
- Objekte haben einen *Zustand* (beschrieben durch *Eigenschaften*).
  - *Attribute* (Name, Geburtsdatum)
    bill[Vorname: “Bill”; Name: “Gates”; Geburtsdatum: 28.10.1955] und
  - *Beziehungen* (*Relationships*, Angestellter_von, verheiratet_mit) zu anderen Objekten
    bill[Angestellter_von: *microsoft*]

  Die Werte der Eigenschaften können sich zeitlich ändern.
- Es können gleichzeitig mehrere unterschiedliche Objekte mit denselben Attributwerten existieren.

Objekte (Forts.)

- Objekte haben ein eigenes, anwendungsspezifisches, aktives *Verhalten* (beschrieben durch *Operationen* (synonym: *Methoden, Dienste*) beschrieben. Operationen können über Parameter verfügen.
- Operationen eines Objektes werden durch Senden einer entsprechenden *Nachricht* an das Objekt aufgerufen.
  microsoft.employ(*bill*).
- Operationen können auch Anfragen an ein Objekt sein.
  microsoft.employed(*bill*)→ Boolean
- Objekte *kommunizieren* mit anderen Objekten um ein *globales* Verhalten zu erzielen.

Klassen

- Dinge mit denselben Eigenschaften werden in *Klassen* zusammengefasst (vgl. Folie 36):
  Beispiel: Klasse "Person"
- Eine *Klasse* beschreibt eine Menge von "gleichartigen" *Objekten*.
  - *Struktur* der Objekte (*"Eigenschaften"*)
    - *Attribute*
      im Beispiel: Vorname: Zeichenkette, Name: Zeichenkette, Geburtsdatum: Datum
    - *Beziehungen* zu anderen Objekten
      im Beispiel: Angestellter_von: Firma, wohnt_in: Stadt, verheiratet_mit: Person

    Alle Objekte einer Klasse haben dieselbe Struktur, aber unterschiedliche Werte der Eigenschaften.
  - *Verhalten* der Objekte (*"Operationen", "Methoden"*): Anfragen an das Objekt, Verändern des Objektzustandes, Auslösen von Aktionen
    im Beispiel: sage_Name⇒ Zeichenkette, sage_Alter⇒ Zahl,
    heirate(Angabe einer Person) ⇒ keine Ausgabe, aber Zustandsänderung bei beiden Objekten

GRAFISCHE DARSTELLUNG OBJEKTORIENTIERTER MODELLIERUNG

- *konzeptuelle Modellierung*: Beschreibung der in einer Anwendung existierenden Konzepte
- sollte *jedes* Projekt begleiten
  (nicht nur Softwareprojekte sondern allgemein Design von Problemlösungen – z.B. *Workflows* in Firmen/Verwaltungen)
- Spezifikation, Modellierung, Visualisierung, Dokumentation

UML (UNIFIED MODELING LANGUAGE)

- UML1.0 1997 bei der OMG (Object Management Group) zur Standardisierung eingereicht.
- aktuelle Version: UML 1.5 (2003); fast fertig: UML 2.0
- http://www.omg.org/uml
- auch in diversen CASE-Tools (Computer Aided Software Engineering) erhältlich
- vertiefende Literatur: Hitz, Kappel: UML@Work (1999), dpunkt Verlag.

UML

- Grafische Modellierungssprache
- verschiedene Abstraktionsebenen
- verschiedene *Modelle*, die zueinander in Beziehung stehen
- Diagramme zur Veranschaulichung: *Sichten* auf Modelle
- Modell wird durch mehrere Diagramme beschrieben,
  jedes Diagramm beschreibt einen *Aspekt* des zu entwickelnden Systems ("Teilpläne"), z.B.
  - Anwendungsfalldiagramm: Funktionalität aus Benutzersicht (Pflichtenheft)
  - Klassendiagramm: statische Modellierung
  - Aktivitätsdiagramm: dynamische Grobmodellierung
  - Interaktionsdiagramm: Sequenz- und Kollaborationsdiagramm: dynamische Modellierung im Detail
  - Zustandsdiagramm: statisch + dynamisch.

⇒ keine orthogonalen Techniken, sondern (beabsichtigte) Redundanz.

- ... wird in dieser Vorlesung nur vereinfacht und teilweise behandelt

[UML] Klassendiagramme: Klassen und Objekte

- Klassen und Objekte werden als Rechtecke dargestellt,
- Signaturen der Attribute, Beziehungen und Operationen werden angegeben,

Klasse
`attr_name:Typ = Initialwert {Zusicherung}`
`class_attr_name:Typ = Initialwert`
`/derived_attr_name:Typ {Berechnungsvorschrift}`
:
`op_name(param: Typ = Defaultwert):Typ {Zusicherung}`
:

- Initialwerte und Zusicherungen können angegeben werden
- Klassenattribute/-operationen werden unterstrichen
- Sichtbarkeit: + = public
  - # = protected (nur für Klasse und ihre Unterklassen)
  - – = private (nur für Methoden der Klasse)
- "/" : abgeleitetes Attribut

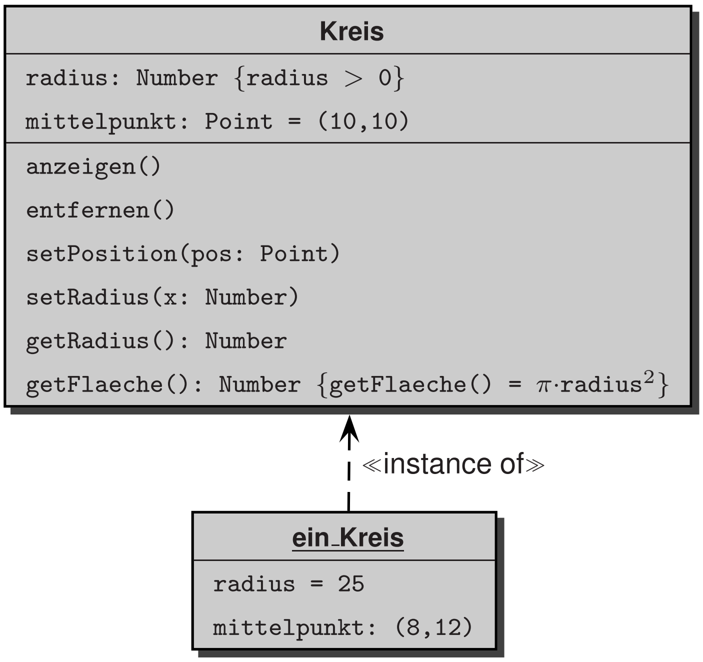
[UML] Klassen: Beispiel

Kreis
`-radius: Number` {radius $>$ 0}
`-mittelpunkt: Point = (10,10)`
`+anzahl`
`+/umfang: Number` {umfang = $2 \cdot \pi \cdot$ radius}
`anzeigen()`
`entfernen()`
`setPosition(pos: Point)`
`move(x: Number, y: Number)`
`setRadius(x: Number)`
`getRadius(): Number`
`getFlaeche(): Number` {getFlaeche() = $\pi \cdot$ radius^2}

- Ein Klassenattribut gibt an wieviele Kreise es gibt
- Umfang als abgeleitetes Attribut

[UML] OBJEKTE

- für die Instanzen werden die jeweiligen Attributausprägungen angegeben
- bei Instanzen wird der Name unterstrichen



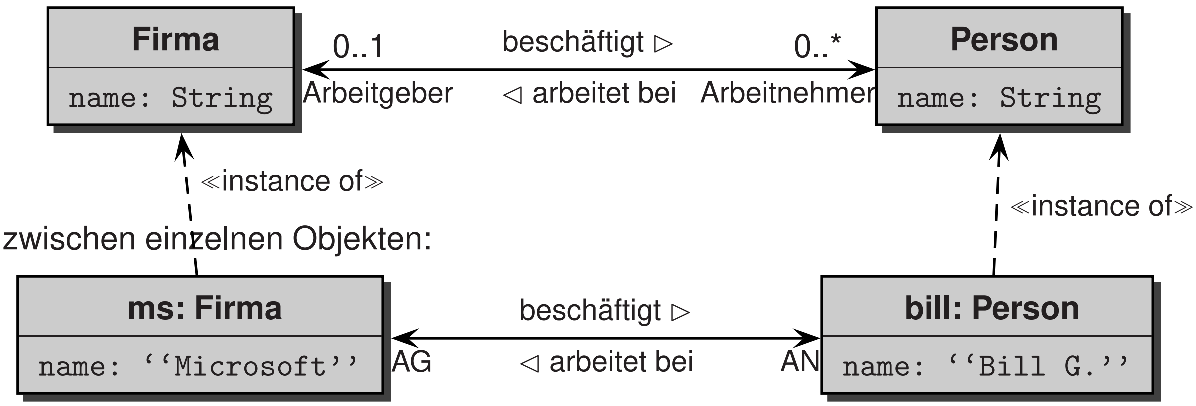
Kurzformen:

- ohne Datenangabe
  k:kreis
- anonym:
  :kreis

[UML] BEZIEHUNGEN

Assoziation

- Eine *Assoziation* ist eine *gerichtete* Beziehung zwischen verschiedenen Objekten.
- Oft besitzt auch die Gegenrichtung einen Namen (beschäftigt/arbeitet_bei)
- Assoziationen haben einen Namen, und können durch Angabe von *Multiplizitätsangaben* sowie *Rollennamen* ergänzt werden.

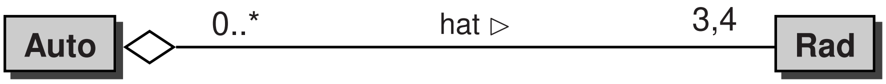


auch zwischen einzelnen Objekten:

SPEZIELLE ARTEN VON BEZIEHUNGEN

Aggregation

- Teile-Ganzes-Beziehung:



Komposition

- Existenz der Einzelteile abhängig von der Existenz des Ganzen,

⇒ Einzelteil kann nur Teil maximal *eines* Ganzen sein.

Eine Rechnung besteht aus mehreren Rechnungspositionen.

REALISIERUNG IN JAVA

- Klasse definiert die nach außen sichtbaren Eigenschaften und Verhaltensweisen (Schnittstelle)
- Umgebung kommuniziert nur über die öffentliche Schnittstelle mit den Objekten.
- interne Realisierung (Implementierung) nach außen nicht sichtbar (Kapselung),
- kann entsprechend geändert werden, ohne das Verhalten zu beeinflussen.

AUFGABE

Modellieren Sie den folgenden einfachen Sachverhalt in UML und implementieren Sie die Klassen “Punkt” und “Kreis” in Java:

- Punkte haben je eine x- und y-Koordinate (parametrisierten Konstruktor, Anfragen getX() und getY())
- Punkte können sich bei Empfang einer “move(x,y)”-Nachricht bewegen.
- Kreise analog zu Folie 279,
- ohne “anzeigen()”, aber mit Klassenmethode “anzahl()”,
- mit “move(x,y)”, “get_mitte()” und einer Anfrage “contains(punkt)”, die zurückgibt, ob ein gegebener Punkt innerhalb des Kreises liegt.
- “Kreis.move(x,y)” wird dabei auf “Punkt.move(x,y)” abgebildet.

Schreiben Sie außerdem ein kleines Testprogramm, das das Verhalten testet.

5.2 Programmablauf durch “Message Passing” über Beziehungen zwischen Objekten

- Klassen implementieren “Verhalten”, das dann von den einzelnen Instanzen ausgeführt werden kann.
- Objekte stehen in Beziehungen zueinander
- Jedes Objekt trägt zu dem Gesamtablauf bei
- Koordination durch Nachrichten (Aufrufe, Antworten) entlang der Beziehungen:
  - entlang gleichberechtigter Beziehungen (Assoziationen): Kooperation verschiedener Instanzen innerhalb eines Systems
  - über Aggregationen/Kompositionen: Teile nehmen Teilaufgaben eines ganzen wahr

Beispiel: System aus gleichartigen Instanzen

Beliebtes Spiel bei Kindergeburtstagen: “Kofferpacken”

- Kinder sitzen im Kreis,
- das erste fängt an und sagt
  “ich packe meinen Koffer und nehme eine Zahnbürste mit”.
- das jeweils nächste muss alle bisher eingepackten Gegenstände einpacken und einen weiteren dazunehmen:
  “ich packe meinen Koffer und nehme eine Zahnbürste und einen Waschlappen mit”.
- Wenn ein Kind die bisherigen Gegenstände nicht mehr korrekt aufzählt, scheidet es aus.

Veranschaulichen Sie in einem UML-Diagramm die Instanzen (Kinder) sowie die zwischen ihnen bestehenden Assoziationen.


```
public class Child {
  private String name;
  private Child next;
  private String[] things; int i = 0;
  public Child() { } /* Default-Konstruktor */
  public Child(String n, Child p, String t1, String t2, String t3)
   { name = n; next = p; things = new String[3];
     things[0] = t1; things[1] = t2; things[2] = t3;  }
  public void setNext(Child p) { next = p; }
  public void anfangen() {
   String text = things[0]; i++;
   System.out.println(name + ": ich packe meinen Koffer und nehme " + text + " mit.");
   next.weitermachen(text); }
  public void weitermachen(String text) {
    if (i < things.length)
    { text = text + " und " + things[i]; i++;
      System.out.println(name + ": ich packe meinen Koffer und nehme " + text + " mit.");
      next.weitermachen(text); }
    else System.out.println(name + ": plaerr!!!!!!"); }
}
```

```
public class Kofferpacken {
   public static void main (String[] args){
      Child andreas = new Child("Andreas",null,
        "eine Zahnbuerste","einen Teddybaer","meine Katze");
      Child britta = new Child("Britta",andreas,
        "einen Waschlappen","eine Puppe","eine Taschenlampe");
      Child carsten = new Child("Carsten",britta,
        "ein Handtuch","Schokolade","einen Wecker");
      Child daniela = new Child("Daniela",carsten,
        "ein Handy","ein Buch","einen Schirm");
      andreas.setNext(daniela);
      carsten.anfangen();
   }
}
```

- Vollziehen Sie den Austausch von Nachrichten, den “Kontrollfluss” (welches Kind ist gerade “aktiv”) sowie das dadurch gezeigte Gesamtverhalten des “Systems” nach.

Beispiel: System aus Instanzen unterschiedlicher Funktionalität

Das Gesamtsystem besteht aus den folgenden Einzelteilen:

- ein Integer-Feld beliebiger Länge.
- Datenannahme: man gibt ihr eine Zahl und ein Feld, und sie hängt die Zahl an das Feld an (falls noch Platz ist).
- Sortierer: man kann ihm ein Integer-Feld (als Referenz) geben, und er sortiert es.
- Sucher: man gibt ihm eine Zahl und ein Feld, und er prüft, ob sie in dem Feld enthalten ist.
- Der Benutzer kommuniziert nur mit dem Gesamtsystem, das die Aufgaben intern weiterverteilt.

Aufgabe:

- Stellen Sie das System als UML-Diagramm dar.
- Implementieren Sie ein solches System.
  - der Konstruktor des Gesamtsystems ruft die Konstruktoren seiner Bestandteile auf,
  - der vom Sortierer verwendete Algorithmus kann beliebig gewählt werden,
  - der Sucher muss wissen, ob das Feld sortiert ist oder nicht, er kann es aber sortieren lassen.

5.3 Klassenhierarchie und Vererbung

semantischer Begriff: “ähnliche” Klassen werden zueinander in Beziehung gesetzt:

Subklassen/Unterklassen verfeinern eine Klasse:

Spezialisierung: Festlegung des Wertebereichs von Unterklassen: Student vs. Person. Nicht jedes Element der Oberklasse muss in einer der spezialisierten Unterklassen enthalten sein (Person; Student, Angestellter).

Generalisierung: Festlegung des Wertebereichs von Oberklassen: Gewässer sind Seen, Meere, Flüsse

$$K_o = \bigcup K_u$$

(semantische) Integritätsbedingung: Jedes Objekt einer Klasse ist auch ein Element von deren Oberklassen:

$$K_o \supseteq K_u$$

KLASSENHIERARCHIE UND VERERBUNG

- disjunkte oder nicht-disjunkte Subklassen

Beispiele

	disjunkt	nicht-disjunkt
Spezialisierung	geometrische Figuren	Personen
	Kreise, Rechtecke[1],...	Studenten, Angestellte
Generalisierung	Gewässer	
	Seen, Flüsse, Meere	

[1]: Quadrate werden als Unterklasse von Rechteck betrachtet

- Klassenhierarchie kann fest sein, oder es ist erlaubt, dass Objekte ihre Klassenzugehörigkeit wechseln.

Beispiele

Subklassen verfeinern Klassen:

Fahrzeug[Treibstoff: String; Hersteller: Firma; zulGG: Number],
Leihsystem[ausleihen(Etwas) → Number]
Person[Name: Zeichenkette; Geburtsdatum: Datum; heiratet(Person)]
geom_Figur[Mittelpunkt: Punkt; Fläche() → Number; move()]

- *LKW* ist Subklasse von *Fahrzeug*
  Autoverleih und *Bibliothek* sind Subklassen von *Leihsystem*
  Student und *Angestellter* sind Subklassen von *Person*
  Kreis und *Rechteck* sind Subklassen von *geom_Figur*
- feinere Signatur:
  zusätzliche Attribute:
  *L*KW[Nutzlast: Number], *S*tudent[Matrikelnummer: Number]
  feinere Typisierung der Parameter:
  Autoverleih[ausleihen(Fahrzeug) → VertragNr] und
  Bibliothek[ausleihen(Buch) → LeihscheinNr]
- Unterschiedliche Berechungen/Zusicherungen: *PKW* $\{zulGG < 2.8t\}$
  Fläche eines Kreises/Rechtecks berechnen, gegenüber abzählen bei allg. geom. Figuren

Vererbung

Klassenhierarchie organisiert *Vererbung*.

Subklassen verfeinern ihre Oberklassen, sind also “sehr ähnlich”:

Strukturvererbung
- “erben” die Signatur der Oberklasse,
- können diese zusätzlich erweitern

Wertvererbung:
- sie erben ebenfalls die angegebenen Defaultwerte der Attribute,
- können aber auch selber typische Attributwerte definieren:
  LKW[Treibstoff: “Diesel”]

Verhaltensvererbung:

- sie erben auch die Verhaltensspezifikation von der Oberklasse (UML: gegeben durch diverse Diagramme; Java: gegeben durch die Implementierung)
- können die Implementierung auch überschreiben: die Methode *ausleihen* ist für allgemeine Leihsysteme nicht implementiert. Die Implementierung wird für die Subklassen *Autoverleih[ausleihen(Fahrzeug) → VertragNr]* und *Bibliothek[ausleihen(Buch) → LeihscheinNr]* angegeben.

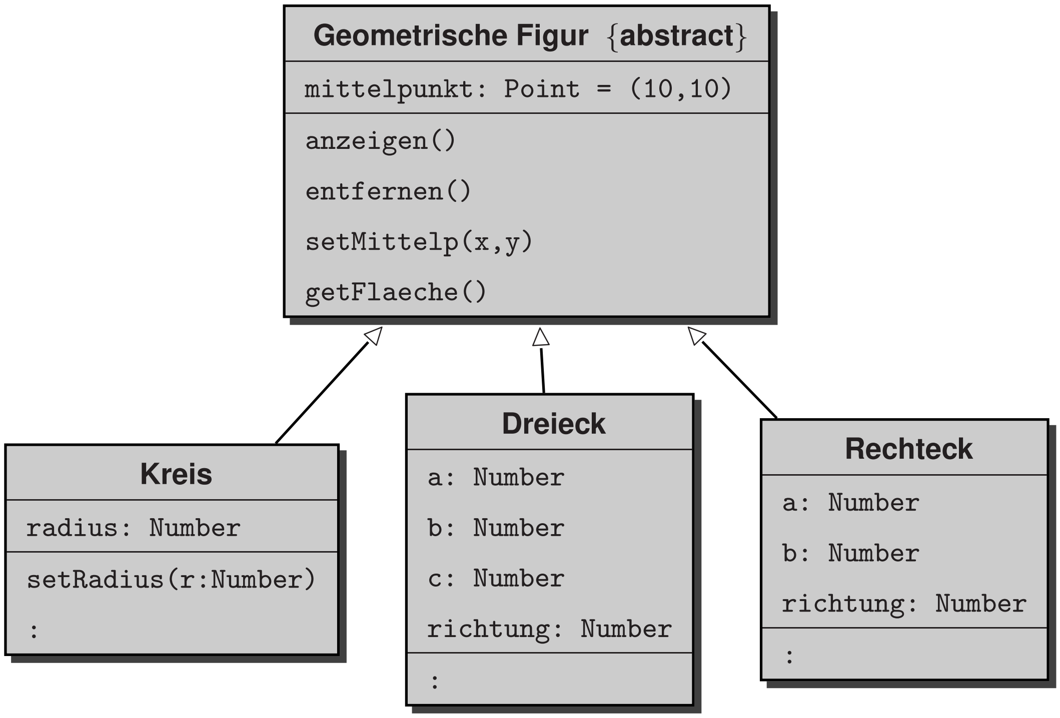
Vererbung: Überladen und Überschreiben

- Ein Ziel von *Klassen* und *Klassenhierarchie* ist, dass Eigenschaften/Operationen mit denselben Bezeichnungen für verschiedene Klassen (oder mit verschiedenen Signaturen) unterschiedlich definiert sein können; sie sind dann *überladen* (*overloading*).
  - "+" für Zahlen und Strings (beabsichtigt)
  - "anmelden" (Studenten für Prüfungen - Auto für Steuer)
    (Methoden für verschiedene Klassen, Name ist zufällig überladen)
  - Methode mit verschiedenen Argumenttypen:
    "setAlarm(Datum,Zeit)", "setAlarm(Zeit)", "setAlarm(Minuten)" für Wecker
    (vgl. Folie 118; siehe auch "Polymorphie"; Folie 303)
- Eigenschaften/Operationen einer Oberklasse können in ihren Unterklassen durch eine *speziellere Definition redefiniert*, bzw. *überschrieben* werden (*overriding*).
  - Fläche berechnen für geometrische Figuren (durch abzählen) und Kreise, Quadrate etc.

Abstrakte Klassen

- Klassifizierung dient dazu, gleichartige Objekte zu gruppieren.
- Häufig gehören alle Instanzen aber nicht direkt einer allgemeinen Oberklasse (Tier, Säugetier, Vogel, geom. Figur, Leihsystem) an, sondern erst gewissen Unterklassen.
- damit ist die Oberklasse eine *abstrakte Klasse*.
  - definiert eine Signatur (Zustand + Verhalten)
  - i.a. aber nicht alle Methoden implementierbar
    (Schlüsselwort in Java: `abstract` sowohl in der Methoden- als auch in der Klassenspezifikation)
  - es können keine Instanzen gebildet werden

Beispiel



Bildung der Klassenhierarchie

Bisher festgestellt:

- Subklassen sind spezieller als Klassen: $K_o \supseteq K_u$
  - häufig: zusätzliche Attribute
  - häufig: zusätzliche Zusicherungen

Beispiel

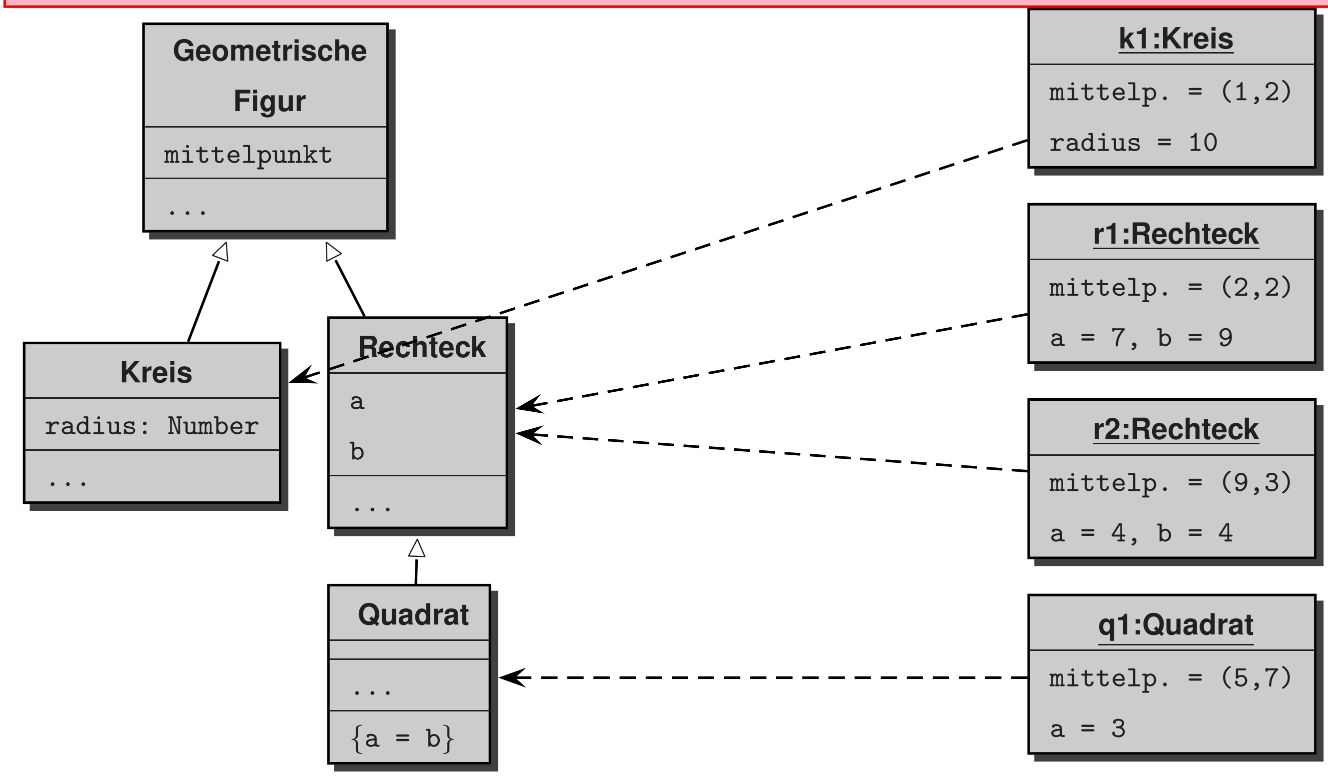
1. Rechteck: Mittelpunkt, SeiteA, SeiteB, Ausrichtung
2. Quadrat: Mittelpunkt, SeiteA, Ausrichtung

Ist nun “Rechteck” eine Subklasse von “Quadrat” (Rechteck erweitert Quadrat um “SeiteB”)?

Nein, Quadrate sind spezielle Rechtecke:

- “Quadrat” ist eigentlich eine Erweiterung von “Rechteck” um die Zusicherung “SeiteA = SeiteB”!

Klassenhierarchie: Beispiel



- Berechnung des Flächeninhalts

Klassenhierarchie in Java

Schlüsselwort: <Subklasse> `extends` <Superklasse>:

```
<Klassendeklaration> ::=
  <Sichtbarkeitsspez> ["abstract"] ["final"] "class" <Bezeichner>
     ["extends"  <Bezeichner>] "{" <Klassendeklarationsrumpf> "}"
```

- Wenn die Superklasse eine abstrakte Klasse ist, und die neue Klasse nicht abstrakt sein soll, muss sie alle noch fehlenden Methoden implementieren.
- Auf (überschriebene) Methoden der Oberklasse kann mit `super.<name>(<parameters>)` zugegriffen werden:

```
public void print() { super.print();
                      eigener Code
                    }
```

ERZEUGUNG VON INSTANZEN VON SUBKLASSEN

Als Basis für die neue Instanz wird eine Instanz der Oberklasse *klasse* benötigt.

- Default-Konstruktor:
  - wird für die neue Klasse *neue_klasse* kein selbstdefinierter Konstruktor angegeben, wird immer der parameterlose Default-Konstruktor mit `new` *neue_klasse*`()` aufgerufen, der die Instanzvariablen initialisiert.
  - dieser ruft als erstes `new` *klasse*`()` auf.
- Verwendung selbstdefinierter Konstruktoren, z.B. `public` *neue_klasse*`(type par) {...}`
  - Ist die erste Anweisung des selbstdefinierten Konstruktors kein Aufruf eines Konstruktors der direkten Oberklasse, so wird automatisch als erstes `new` *klasse*`()` aufgerufen.
  - im allgemeinen will man aber einen parametrisierten Konstruktor der Oberklasse aufrufen.
    Dies kann durch Aufruf von `super(`*args*`)` geschehen.

```
public neue_klasse(parameterdecl) { super(args);
                                    eigener Code
                                  }
```


BEISPIEL: KLASSENHIERARCHIE

```
public class Person {
   protected String name;
   public Person() { } /* Default-Konstruktor */
   public Person(String n) {name = n;}
   public void setName(String thename){ name = thename; }
   public String getName() {return name;}
   public String wasBinIch(){ return "Person"; }
   public void printName(){ System.out.println(name); } }
```

```
public class Student extends Person {
   protected long MatNo;
   public Student(String n, long mn) {super(n); MatNo = mn;}
   public String wasBinIch(){ return super.wasBinIch() + ", " + "Student"; }
   public long getMatNo() {return MatNo;} }
```

```
public class Angestellter extends Person {
   protected long gehalt;
   public Angestellter(String n, long g) {super(n); gehalt = g;}
   public String wasBinIch(){ return super.wasBinIch() + ", " + "Angestellter"; }
   public long getGehalt() {return gehalt;} }
```

Aufgabe

Erweitern Sie die zuvor geschriebenen Klassen “Punkt” und “Kreis” zu einer Implementierung geometrischer Figuren wie oben beschrieben:

- abstrakte Oberklasse “geoFigur”
- Klassen “Kreis”, “Rechteck”, “Quadrat”

Schreiben Sie außerdem wieder ein kleines Testprogramm, das das Verhalten testet (im wesentlichen Flächeninhalt).

Polymorphie

- Polymorphie (griech): Vielgestaltigkeit
  ... wenn eine Methode in verschiedenen Formen auftritt:
- verschiedene Implementierungen für verschiedene Parametertypen:

```
class wecker { // Vorsicht: keine gueltigen java-Datentypen
  time alarmZeit; date alarmDatum;
  time jetztZeit;
  boolean jeden_tag;
  public void setAlarm(date datum, time zeit) {
         alarmDatum = datum; alarmZeit = zeit; jeden_tag = false; }
  public void setAlarm(time zeit) {
         alarmZeit = time; jeden_tag = true; }
  public void setAlarm(int minuten) {
         alarmZeit = jetztZeit + minuten; jeden_tag = false; }
}
```

  Auswahl der tatsächlich gewünschten Implementierung erfolgt aufgrund der aktuellen Parameter (manchmal bereits zur Übersetzungszeit bekannt, manchmal erst zur Laufzeit).

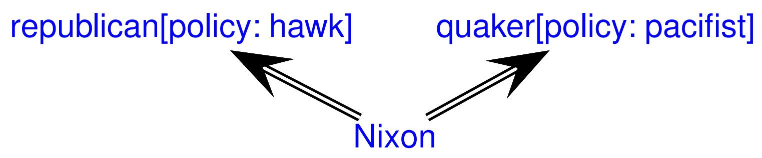
POLYMORPHIE (FORTS.)

- verschiedene Implementierungen für verschiedene Klassen: geom_Figur.getFlaeche() wird durch die einzelnen Subklassen polymorph implementiert:
  - geom_figure.getFlaeche(): abstract
  - Kreis.getFlaeche(): return $\pi \cdot r^2$
  - Rechteck.getFlaeche(): return $a \cdot b$
- verschiedene Implementierungen für Oberklasse und Subklasse "Überschreiben":
  - Rechteck.getFlaeche(): return $a \cdot b$
  - Quadrat.getFlaeche(): return a^2
- Nachteil: Der Rückgabedatentyp darf nicht verändert (spezialisiert) werden
- in beiden obigen Fällen: Auswahl der tatsächlich gewünschten Implementierung zur Laufzeit anhand des aktuellen Host-Objektes.

EINFACH- ODER MEHRFACHVERERBUNG

Für Modellierung/Wissensrepräsentation oft notwendig:

- Objekt kann Klasse wechseln (Student $\rightarrow$ Angestellter)
- Objekt kann in mehreren Klassen sein (Student/Angestellter)
- Problem: multiple Vererbung/Konflikte

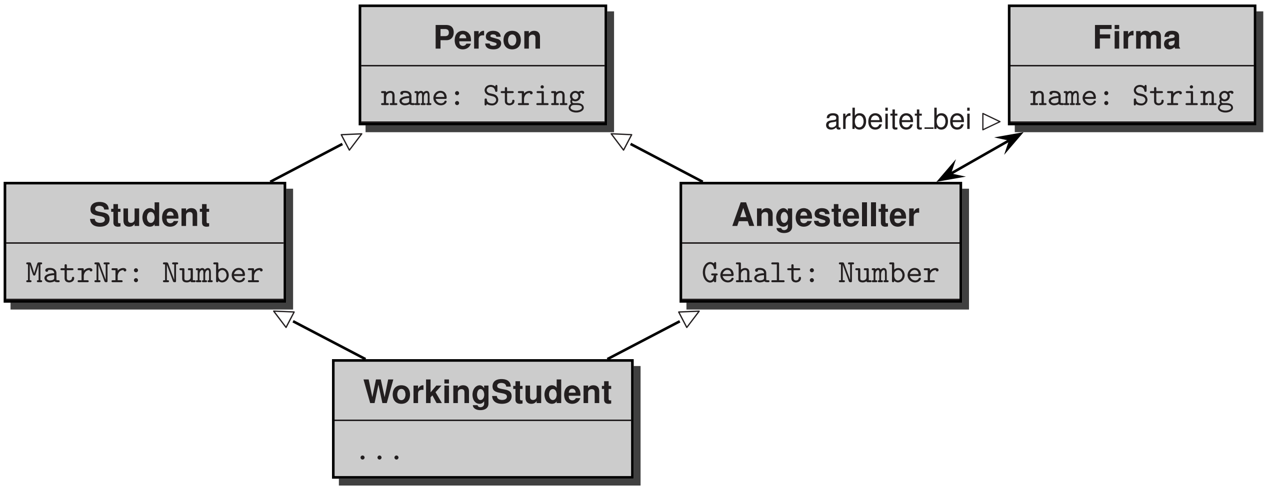


- Was ist Nixons *policy* ?? $\Rightarrow$ Konfliktlösungsstrategien für multiple Vererbung.

Java: streng baumartig – jede Klasse hat eine eindeutige direkte Oberklasse. Vererbung eindeutig. Kein Wechsel der Klasse möglich.
(andere Programmiersprachen, z.B. C++ und Eiffel erlauben Mehrfachvererbung; man muss allerdings dann beim Aufruf jedesmal angeben, von welcher Oberklasse die Implementierung geerbt werden soll.)

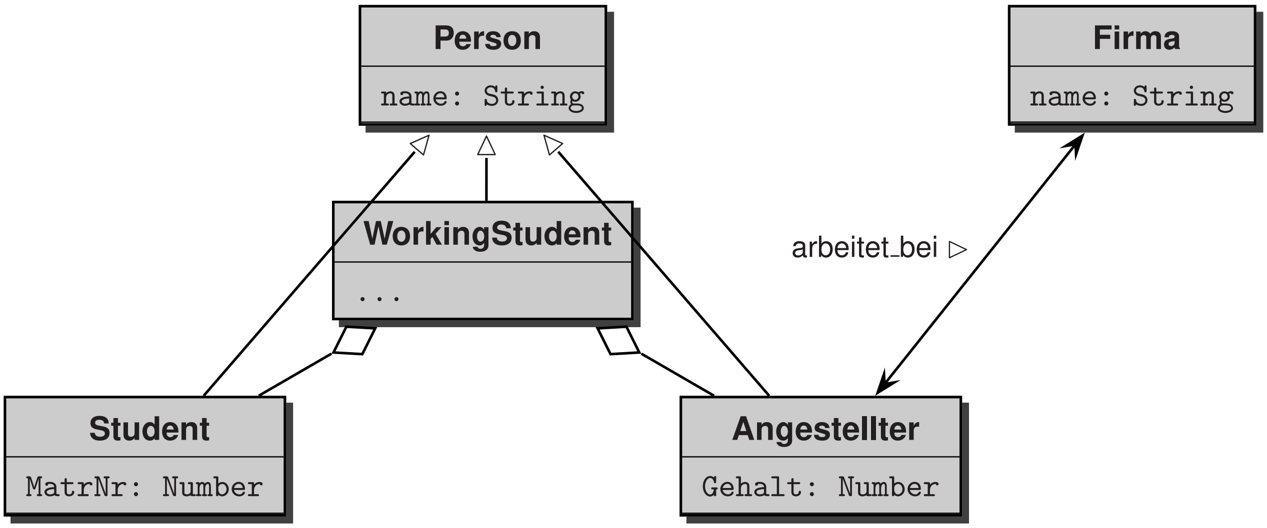
Auflösung von Mehrfachvererbung durch Delegation

- Studenten sind Personen, Angestellte sind Personen.
  WorkingStudents sind Studenten, die auch Angestellte sind.



- Konflikt bei Vererbung der Methode `wasBinIch()`

Auflösung von Mehrfachvererbung durch Delegation



- "WorkingStudent"-Objekt ist ein Person-Objekt und besitzt je ein Student-Objekt und ein Angestellten-Objekt
- Methoden-Anwendungen werden ggf. an das jeweilige *Stellvertreter*-Objekt *delegiert*.
- Nachteil: man muss die Signaturen von Student/Angestellter und WorkingStudent konsistent halten

BEISPIEL: DELEGATION

```
public class WorkingStudent extends Person {
   Student me_as_student;
   Angestellter me_as_angestellter;
   public WorkingStudent(String n, long mn, long g)
      { super(n);
        me_as_student = new Student(n,mn);
        me_as_angestellter = new Angestellter(n,g); }
   public String getName() {return name;}       // von extends Person
   public long getMatNo() {return me_as_student.getMatNo();}
   public long getGehalt() {return me_as_angestellter.getGehalt();} }
```

```
public class WorkingStudentTest {
   public static void main (String[] args){
      WorkingStudent joe = new WorkingStudent("Joe", 4711, 1000);
      System.out.println(joe.getName());
      System.out.println(joe.getMatNo());
      System.out.println(joe.getGehalt());
      System.out.println(joe.wasBinIch());
} }
```


METHODEN-SPEZIFIKATION UND MEHRFACHVERERBUNG DURCH INTERFACES

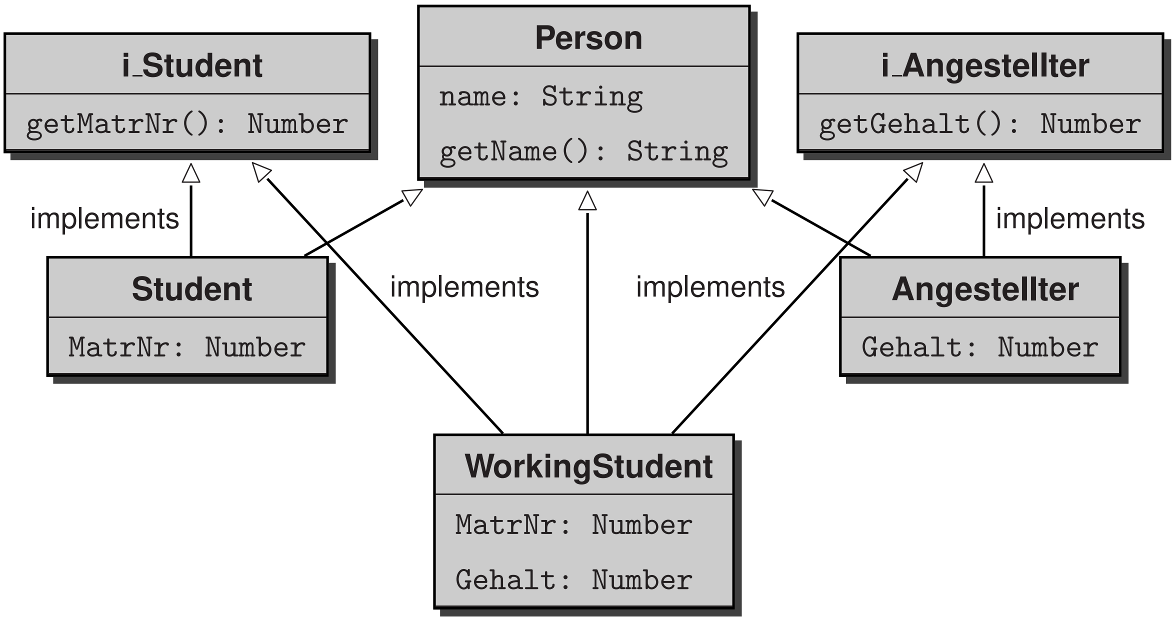
Interfaces beschreiben nur das Verhalten (Operationen):

interface <Name> extends <Ober-Interfaces-Liste>
{ <Methodendeklarationen>}

- Von Interfaces können keine konkreten Instanzen erzeugt werden (eigentlich klar, da Interfaces nur die Signatur des Verhaltens spezifizieren – eine eventuelle Instanz könnte also keinen Zustand besitzen).
- es kann von mehreren Interfaces geerbt werden
  (keine Konflikte, da ja nur Signaturen geerbt (gesammelt) werden)
- Instanzen werden stattdessen von Klassen erzeugt – die die Deklarationen der Interfaces um eine Zustandssignatur sowie Implementierungen erweitern:

  class <Name> [extends <Oberklasse >]
  [implements <Interfaces-Liste>] {<Klassenrumpf>}

Beispiel: Vererbung und Interfaces



- Nachteil: Methoden müssen mehrfach implementiert werden
  akzeptabel wenn sie sowieso jedesmal unterschiedlich wären ...

5.4 Arbeiten mit Objekten

Einige Dinge sind mit Objekten etwas anders als mit “normalen” Werten:

- Drucken
- Vergleichen
- Kopieren
- Iterieren

DIE KLASSE "OBJECT"

- Wird bei einer selbstdefinierten Klasse keine Oberklasse angegeben, ist sie eine direkte Subklasse der *Wurzelklasse* `Object`.
- diese definiert einige wichtige Methoden für alle Klassen:
  - `boolean equals(Object o)` um das aktuelle Objekt mit einem anderen zu vergleichen
  - `Object clone()` erstellt eine Kopie des Objektes
  - `String toString()` liefert eine textuelle Repräsentation des Objektes

Da "Object" natürlich nicht wissen kann, was diese Methoden für benutzerdefinierte Klassen schlussendlich tun sollen, müssen sie explizit implementiert (und damit überschrieben) werden, wenn man sie nutzen will.

ABFRAGEN DER KLASSENHIERARCHIE

Der Ausdruck "$name_1$ `instanceof` $name_2$" liefert genau dann true, wenn $name_1$ eine Instanz der Klasse $name_2$ oder einer ihrer Subklassen ist.

Sei Joe eine Instanz des "WorkingStudent" von Folie 310:

- (x `instanceof Object`) ergibt true für alle Objekte x
- `(joe instanceof Person)` ergibt true
- `(joe instanceof WorkingStudent)` ergibt true
- `(joe instanceof Student)` ergibt false
- `(joe instanceof Angestellter)` ergibt false

Aber meistens benötigt man das garnicht ...

Die Methode "toString"

- Die Methode `toString` wird implizit aufgerufen, um ein Objekt "auszugeben":

```
public class Person {
  protected String name;
  protected String vorname;
  public Person() { } /* Default-Konstruktor */
  public Person(String n) {name = n;}
  public Person(String v, String n) {this(n); vorname = v;}   // erklaeren!
  public void setName(String thename){ name = thename; }
  public String getName() {return name;}
  public String getVorname() {return vorname;}
  public void printName(){ System.out.println(name); }
  public String wasBinIch(){ return "Person"; }
  public String toString() { if (vorname != null) return (vorname + " " + name);
         return name; }}
```

```
public class PersonToStringTest {
   public static void main (String[] args){
      Person jim = new Person("Jim","Beam");
      System.out.println(jim);
      System.out.println("Diese Person ist " + jim); }}
```


Gleichheit von Referenztypen

Objekte, Arrays und Strings sind Referenztypen.

Gleichheitsbegriff

- Ausdrücke, die Referenztypen ergeben, sind gleich ("=="), wenn die auf dasselbe Objekt zeigen, d.h., wenn die dieselbe Referenz ergeben!

Verschiedenheitsbegriff

- man kann durchaus zwei verschiedene Objekte mit demselben Befehl erzeugen, die "gleich" aussehen:

  ```
  joe1 = new Person("Joe");
  joe2 = new Person("Joe");
  ```

  erzeugt zwei Instanzen der Klasse Person, die jeweils intern absolut gleich aussehen (sie haben den Namen "Joe").

  Der Vergleich "`joe1 == joe2`" ergibt aber `false`, weil es eben unterschiedliche Instanzen (und damit unterschiedliche Referenzen) sind.

  Ist ja in diesem Fall auch so gewollt.

Identität vs. Gleichheit

- "==" testet *Identität*
- "Gleichheit" von Objekten muss –jeweils für eine Klasse– definiert werden. Dazu ist der `equals`-Operator da.
- Zwei Objekte sind *identisch*, wenn sie gleiche Objektidentifikatoren besitzen.
- Zwei Objekte sind *gleich*, wenn sie gleiche Werte besitzen.
  (d.h. $o_1.eigenschaft == o_2.eigenschaft$ für alle Eigenschaften)
- Zwei Objekte sind *tiefengleich*, wenn sie nach rekursivem Dereferenzieren/Navigieren ihrer Referenz-Attribute gleiche Werte besitzen.
  (d.h. $o_1.pfadausdruck == o_2.pfadausdruck$ für alle Pfadausdrücke, die einen *Literalwert* ergeben)
- Sind sie bereits ohne Dereferenzierung gleich, so werden sie auch *oberflächengleich* genannt (also derselbe Fall wie oben "gleich").
- etwa wie in der deutschen Sprache "dasselbe Buch" (Identität) und "das gleiche Buch" ((Tiefen)gleichheit).
- Tiefengleichheit z.B. bei Molekülstrukturen in der Chemie.

Aufgabe: Tiefengleichheit

a) Bestimmen Sie in jeder der folgenden Teilaufgaben die Menge der tiefengleichen Objekte (Hinweis: stellen Sie die Objekte grafisch dar). Strings werden als elementare Objekte gesehen, die gleich sind, wenn sie syntaktisch gleich sind.

i. o1[name→"a",next→o2], o2[name→"b",next→o3], o3[name→"c"], o4[name→"b",next→o5], o5[name→"c"].

ii. o1[name→"a",next→o2], o2[name→"a",next→o1].

iii. o1[name→"a",next→o2], o2[name→"b",next→o1], o3[name→"a",next→o2].

iv. Betrachten Sie ein Wasser-Molekül (H-O-H), welche Objekte darin sind tiefengleich ?

b) Geben Sie eine (induktive) Definition für Tiefengleichheit an in der Form "Objekt A und Objekt B sind tiefengleich, wenn für alle ihre Attribute gilt, dass ...".

Lösung: Tiefengleichheit

... zuerst den theoretischen Teil:

b) Für skalare Attribute: Zwei Objekte o_1 und o_2 sind tiefengleich, wenn

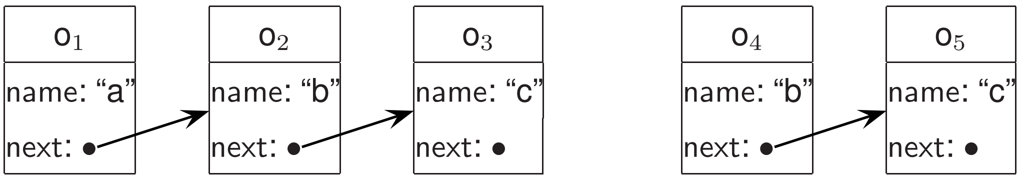
- für alle Attribute m, die direkt Werte aus einem Grundbereich (String, Integer, Boolean) ergeben (also "Nicht-Referenz-Attribute"): $o_1[m{\rightarrow}x]$ genau dann wenn $o_2[m{\rightarrow}x]$.
- für alle Referenzattribute m, die Objekte ergeben: $o_1[m{\rightarrow}x]$ genau dann wenn $o_2[m{\rightarrow}y]$ und x und y sind tiefengleich.
  ⇒ Rekursive Definition.

Äquivalent: Zwei Objekte o_1 und o_2 sind tiefengleich, wenn für alle Folgen $m_1, m_2, m_3, \ldots, m_n$ von Methodenanwendungen so dass $o_1.m_1.m_2.m_3.\ldots[m_n{\rightarrow}x]$ einen Wert x aus einem Grundbereich ergibt, auch $o_2.m_1.m_2.m_3.\ldots[m_n{\rightarrow}x]$ gilt.

Für mengenwertige Attribute: komplizierter (Übungsaufgabe?)

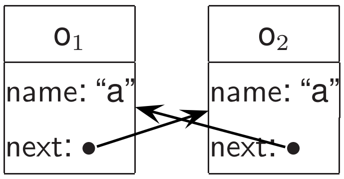
Lösung: Tiefengleichheit

a) i.



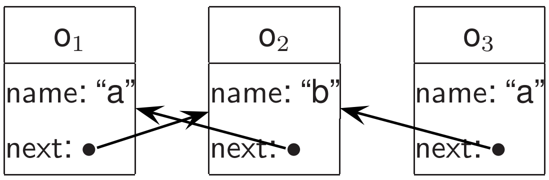
o_3, o_5 sowie o_2,o_4 sind tiefengleich.

ii.



o_1 und o_2 sind tiefengleich.

iii.



Methodenanwendungen: o_1 und o_3 sind tiefengleich.

iv. Die beiden H-Atome sind tiefengleich. Sie nehmen dieselben Rollen in dem Molekül ein und sind ununterscheidbar.

EQUALS: TIEFENGLEICHHEIT

Ein “echter” Vergleich auf “Ununterscheidbarkeit” von Objekten wird also durch *Tiefengleichheit* geliefert.

- ist mit der Methode `public boolean equals(`$object$`)` möglich.
- diese muss also für selbstdefinierte Klassen auch angegeben werden.
- im allgemeinen wird man dies induktiv (siehe Aufgabe) tun.
- Faustregel: alle Datenfelder, sowie alle Dinge die innerhalb von Objektmethoden mit `new` erzeugt werden, müssen rekursiv verfolgt werden. Referenzen auf “selbständige” Objekte werden als Referenzen verglichen.
  Beispiel: für `Person.Adresse.Stadt` oder `Person.Vater` ist Referenz-Identität sinnvoll.
- oft aber auch anwendungsspezifische Gleichheit anhand von “Schlüsselattributen”.
  Beispiel: Zwei Personen werden als gleich betrachtet wenn Name und Geburtsdatum/ort dieselben sind – unabhängig von der gespeicherten Adresse.

Anmerkung: Stringvergleiche sollte man korrekt auch mit der vordefinierten Methode $string_1$`.equals(`$string_2$`)` der built-in-Klasse `String` machen, nicht mit “==”.

EQUALS: IMPLEMENTIERUNG

Man betrachte die Situation

```
class irgendwas {
  public boolean equals(??????????){ ... }
}
```

- Die Klasse “Object” definiert die Methode

  ```
  public boolean equals(Object other)
  ```

- generische Anwendungen wissen nicht, von welcher Klasse das *Argument* `other` ist. Der Versuch

  ```
  class irgendwas {
    public boolean equals(irgendwas other){...}
  }
  ```

  klappt *nicht !*

EQUALS: IMPLEMENTIERUNG (FORTSETZUNG)

Man muss die `equals(Object other)`-Methode *im ganzen* überschreiben und dann geeignet die Klassenzugehörigkeit überprüfen und explizit casten:

```
class irgendwas {
  public boolean equals(Object other){
    if (other instanceof irgendwas)
    { irgendwas o = (irgendwas)other;
      //  ... und jetzt mit o arbeiten
      //  (oder *immer*  ((irgendwas)other)  casten) ...
    }
    return false;  // wenn es nicht instanceof ist, ist es ungleich
  }
}
```

(siehe folgende Beispiele)

AUFGABE

Betrachten Sie eine Klasse “Buch”:

```
public class Buch {
    String titel;
    Person[] autoren;
    int auflage;
    String ISBN;
    Bibliothek bib;
    String kennzeichnung;}
```

Ihr Professor gibt Ihnen eine Liste solcher Instanzen, die Sie für eine Prüfung gelesen haben sollen. Sie haben eine entsprechende Liste mit Büchern, die Sie ausgeliehen und gelesen haben.

Definieren Sie einen `equals`-Operator, der es erlaubt, zu überprüfen, ob Sie das notwendige Wissen erworben haben.

KOPIEREN

Dieselben Überlegungen gelten auch für das Kopieren:

- `Person a = new Person("Jim Beam");`
  `Person b = a;`
  ergibt, dass a und b aus dasselbe Objekt zeigen.
  `a.neueAddresse("Route 66", "0815 Glenfiddich")` ändert die Addresse dieses Objektes, so dass auch `b.Addresse` diesen Wert ergibt.

  Ist hier auch so beabsichtigt.

- Oft will man jedoch etwas kopieren, und dann verändern, ohne das Original zu verändern:

  ```
  Stundenplan plan2001 = uniGoettingen.informatik1.plan;
  Stundenplan plan2002 = plan2001;
  plan2002.delete("Informatik I", "Donnerstag", "14:00");
  plan2002.insert("Informatik I", "Freitag", "14:00");
  ```

  In diesem Fall sollte die 2.Zeile das gesamte Objekt der Klasse “Stundenplan” als Struktur verdoppeln.

KOPIEREN

- “=” (Zuweisung) kopiert *Referenzen*,
  `public Object clone()` kopiert Objekte

- Die Klasse `Object` definiert eine “native” Methode

  ```
  protected Object clone();
  ```

  Verwendung des Interfaces `Cloneable` in einer Klassendeklaration signalisiert, dass eine Klasse die Methode `public Object clone()` anbietet. (“korrekte” Benutzung und Java-spezifische Details sind nicht Info-I-geeignet)

- Info-I: man implementiert eine Methode `public Object clone()`. Im allgemeinen wird man auch dies rekursiv tun (wobei bei Referenzen wieder unterschieden werden muss, ob es genügt, die Referenz zu kopieren, oder ob tatsächlich das referenzierte Objekt verdoppelt werden muss).

- Da `clone()` nur die Ergebnisklasse `Object` hat, muss das Ergebnis zurückgecastet werden:

  class var = (*class*)(*referenz*.clone());

Beispiel: IntegerFeld-Klasse

Erweiterung der IntegerFeld-Klasse mit `toString()`, `clone()`, und `equals(object other)`:

```
  public String toString() {
    String all = "[" + my_array[0];
    for (int i=1; i < laenge(); i++)
        all = all + "," + my_array[i];
    return (all + "]");  }
  public Object clone() {
    int[] neuesFeld = new int[laenge()];
    for (int i=0; i < laenge(); i++) neuesFeld[i] = my_array[i];
    return new IntegerFeld(neuesFeld); }
  public boolean equals(Object other) {
    if (other instanceof IntegerFeld)
    { boolean eq = (laenge() == ((IntegerFeld)other).laenge());
      for (int i=0; i < laenge(); i++)
          eq = eq && (inhalt(i) == ((IntegerFeld)other).inhalt(i));
      return eq;
    }
    return false;
  }
```


Beispiel: Array-Klasse (Test)

```
public class IntegerFeldGenericTest{
  public static void main (String[] args){
    IntegerFeld testfeld = new IntegerFeld(15, 49); // 15 Zahlen
    IntegerFeld zweitesfeld = (IntegerFeld)testfeld.clone();
    System.out.println("Original: " + testfeld);
    System.out.println("Kopie: " + zweitesfeld);
    System.out.println("Kopie == Original: " + (testfeld == zweitesfeld));
    System.out.println("Kopie eq Original: " + testfeld.equals(zweitesfeld));
    zweitesfeld.set(5,100);
    System.out.println("Original: " + testfeld);
    System.out.println("Kopie: " + zweitesfeld);
    System.out.println("Kopie eq Original: " + testfeld.equals(zweitesfeld));
} }
```

Aufgabe

Erweitern Sie die Klasse um eine Methode `public boolean contains(IntegerFeld other)`, die auf Teilmengenbeziehung testet.

VERGLEICHE

- eben behandelt: Gleichheit vs. Identität
- oft *vergleicht* man jedoch Objekte, um
  - festzustellen, welches “größer” oder “besser” ist
  - sie zu ordnen
- Dieser Vergleich ist von der jeweiligen Semantik der Objekte –bzw. der Klasse– abhängig
- Beispiele:
  - einzelne Attribute: Ordnen von Büchern nach ISBN, Städten nach PLZ
  - mehrere Attribute: Tabelle einer Sport-Liga
  - Berechnungen: Klassifikation von Hotels nach Punktesystemen

VERGLEICHBARKEIT VON OBJEKTEN

Das Interface `java.lang.Comparable` deklariert eine Instanzenmethode

```
public interface Comparable{
   public int compareTo(Object o);
}
```

die folgendermaßen implementiert werden soll:

- eine negative Zahl (oft -1) zurückgibt, falls das aktuelle Objekt “kleiner” als `other` ist,
- eine positive Zahl (oft $+1$) zurückgibt, falls das aktuelle Objekt “größer” als `other` ist,
- 0 zurückgibt, falls das aktuelle Objekt “gleichgroß” wie `other` ist.

Damit können Algorithmen (z.B. binäre Suche oder Sortieren) generisch formuliert werden, indem sie immer `compareTo()` zu Vergleichen verwenden.

- Hinweis: wie schon für `equals` muß mit `compareTo(Object other)` gearbeitet werden, und überprüft und gecastet werden!
- Die Generizität von `compareTo()` erlaubt z.B. auch Objekte einer Klasse mit Objekten einer anderen Klasse zu vergleichen (deren Deklaration muß importiert werden!).

Beispiel: Rationale Zahlen

Eine vervollständigte Subklasse von `Rational` (vgl. Folie 133):

```
public class EnhancedRational extends Rational implements Comparable{
  public EnhancedRational(int z) { super(z); }
  public EnhancedRational(int z, int n) { super(z,n); }
  public String toString() { return(Zaehler + "/" + Nenner); }
  public boolean equals(Object other) {
    if (other instanceof Rational)
    { Rational o = (Rational)other;
      return((Zaehler == o.Zaehler) && (Nenner == o.Nenner)); }
    return false; }
  public Object clone() { return new EnhancedRational(Zaehler,Nenner); }
  public int compareTo(Object other) {
    if (other instanceof Rational)
    { Rational o = (Rational)other;
      if (getValue() < o.getValue()) return -1;
      if (getValue() > o.getValue()) return 1;
      return 0; }
    return 0;    // nicht vergleichbar; AUFPASSEN!
} }
```


Beispiel: Rationale Zahlen (Test)

```
  public class EnhancedRationalTest{
    public static void main (String[] args){
       EnhancedRational a = new EnhancedRational(1,5);
       System.out.println("a: " + a);
       EnhancedRational b = new EnhancedRational(1,5);
       System.out.println("b: " + b + ", aber eine andere Instanz");
       System.out.println("a==b: " + (a==b));
       System.out.println("a equals b: " + (a.equals(b)));
       EnhancedRational c = (EnhancedRational)(a.clone());
       c.setNenner(10);
       System.out.println(a);
       System.out.println(c);
       EnhancedRational d = new EnhancedRational(2,10);
       System.out.println("d: " + d);
       System.out.println(a + " compared to " + c + ": " + a.compareTo(c));
       System.out.println(a + " compared to " + d + ": " + a.compareTo(d));
       System.out.println(a + " equals " + d + ": " + a.equals(d));
  }}
```

AUFGABE

Mit Hilfe der `compareTo`-Methode kann man nun generisch z.B. Arrays (d.h., über beliebigen Objekttypen) sortieren.

Komplettieren Sie den folgenden Rahmen, der eine Klasse implementiert, die nur eine Klassenmethode (`static`) bereitstellt, die ein als Argument gegebenes Array sortiert:

```
public class Sortierer{
  public static void sortiere(Comparable[] feld) {
    // to be extended
  }
}
```

Diesen Algorithmus können (und werden) Sie dann mit

```
Sortierer.sortiere(ein_feld);
```

aufrufen, wenn `ein_feld` ein Array über einem Objekttyp ist, der `Comparable` implementiert.

- bisher: Klassen implementieren “Anwendungsklassen”.
- hier: ein *Algorithmus* wird als “Dienstleistungsklasse” implementiert.

AUFGABE

In einer früheren Aufgabe haben Sie eine Klasse für komplexe Zahlen implementiert. Ergänzen Sie diese mit

```
public class Complex implements Comparable {
    :
  public int compareTo(Object other) {
      :
  }
}
```

ebenfalls um eine Vergleichsoperation (vergleichen des Betrages)

- es gibt nun *verschiedene* komplexe Zahlen, die “gleich groß” sind.
- Implementieren Sie `CompareTo()` so, dass es auch den Vergleich mit rationalen Zahlen anhand ihres Betrages erlaubt.
- Sortieren Sie Folgen von komplexen Zahlen (die mehrere “gleich große” Zahlen enthalten) mit verschiedenen Sortierverfahren
- Veranschaulichen Sie damit die Eigenschaft “Stabilität” von Sortierverfahren.

AUFGABE

Ergänzen Sie den folgenden Klassenrahmen “Team” um eine Vergleichsmethode:

- ein Team ist umso besser, je mehr Punkte es hat,
- Bei Punktgleichheit entscheidet die Tordifferenz,
- Bei Punktgleichheit und gleicher Tordifferenz ist das Team besser, das mehr Tore geschossen hat.

Auf den folgenden Folien (und im Web) finden Sie eine Klasse “Bundesliga”, die eine Klassenmethode (`static`!) bereitstellt, die ein Feld mit den Teams der Saison 1997/1998 initialisiert, sowie einen Rahmen für ein Testprogramm.

Aufgabe (Rahmen)

```
public class Team implements Comparable{
   private String name;
   int punkte;
   int tore;
   int gegentore;
   public Team(String n, int p, int t1, int t2)
     { name = n; punkte = p; tore = t1; gegentore = t2; }
   public String toString() { return( name + " " +
           punkte + " Punkte " + tore + ":" + gegentore +  " Tore"); }
   public boolean equals(Object other) {
     if (other instanceof Team)
        return(name.equals(((Team)other).name));
     return false; }
   public int compareTo(Object other) {
     // to be extended <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
   }
 }
```

Aufgabe (Initialisierungsdaten)

```
public class Bundesliga{
  public static Team[] teams1998(){
   Team[] feld = new Team[18];
   feld[0] = new Team("Hertha BSC",43,41,53);
   feld[1] = new Team("Schalke 0:4",52,38,32);
   feld[2] = new Team("VfL Bochum",41,41,49);
   feld[3] = new Team("Hansa Rostock",51,54,46);
   feld[4] = new Team("Borussia Muenchengladbach",38,54,59);
   feld[5] = new Team("VfL Golfsburg",39,38,54 );
   feld[6] = new Team("Werder Bremen",50,43,47);
   feld[7] = new Team("1. FC Kaiserslautern",68,63,39);
   feld[8] = new Team("Karlsruher SC",38,48,60);
   feld[9] = new Team("MSV Duisburg",44,43,44);
   feld[10] = new Team("Arminia B**l*f*ld",32,43,56);
   feld[11] = new Team("Borussia Dortmund",43,57,55);
   feld[12] = new Team("1. FC Koeln",36,49,64);
   feld[13] = new Team("1860 Muenchen",41,43,54 );
   feld[14] = new Team("Bayern Muenchen",66,69,37);
   feld[15] = new Team("Bayern Leverkusen",55,66,39);
   feld[16] = new Team("Hamburger SV",44,38,46);
   feld[17] = new Team("VfB Stuttgart",52,55,49);
   return feld;
  }}
```


Aufgabe (Testprogramm)

```
public class BundesligaTest{
  public static void main (String[] args){
  Team[] teams = Bundesliga.teams1998();
  System.out.println(teams[0]);
  // Vergleichsausgaben fuer compareTo
  System.out.println(teams[14] + " compared to " + teams[10] +
     " : " + teams[14].compareTo(teams[10]));
  System.out.println(teams[16] + " compared to " + teams[9] +
     " : " + teams[16].compareTo(teams[9]));
  // Sortierer aufrufen
  Sortierer.sortiere(teams);
  // Tabelle ausgeben
  for (int i=0; i < teams.length; i++)
      System.out.println(teams[i]);
  System.out.println(teams[0]);
  }
}
```

Anmerkung: das Feld `teams` wird als *Referenz* an den Sortierer übergeben, also von ihm direkt verändert.

Anmerkung: Nicht jede “kleiner”-Relation definiert eine Ordnung:

Beispiel (Auto-Quartett-Karten)

Seien drei Objekte (Autos) o_1, o_2, o_3 mit Eigenschaften A (Höchstgeschwindigkeit), B (Hubraum), C (Motorleistung) wie folgt gegeben:

	o_1	o_2	o_3
A	180	170	160
B	1400	1800	1700
C	110	90	130

Ein Objekt ist nun “besser” als ein anderes, wenn es den “direkten Vergleich” gewinnt, d.h. in mehr Eigenschaften “besser” ist:

Die Relation

besser = $\{(o_1, o_2),\ (o_2, o_3),\ (o_3, o_1)\}$

ist eine gültige Vergleichsrelation im Sinne von `compareTo()`, definiert aber keine Ordnung (ist nicht transitiv).

Aufgabe

Untersuchen Sie das Verhalten der Sortieralgorithmen, um ein solches Feld zu sortieren.

AUFGABE

Zeigen Sie:

- Die durch `Team.compareTo()` definierte Vergleichsrelation definiert eine Ordnung $<_{team}$ (d.h., ist reflexiv und transitiv); sogar eine totale Ordnung (d.h., für a,b gilt jeweils $a \leq_{team} b$ oder $b \leq_{team} a$).
- jede Vergleichsrelation auf einer Klasse C, die auf einer Abbildung $|\_| : C \times C \to \mathbb{N}$ (“Betragsfunktion”) basiert, ist eine totale Ordnung.

ITERATIONEN ÜBER OBJEKTE

Situation: man hat ein Feld von Objekten einer ziemlich allgemeinen Klasse
z.B. `geom_Figur[] meine_Figuren;` und iteriert darüber
z.B.

```
for (int i=0; i < meine_Figuren.length; i++)
    System.out.println(meine_Figuren[i].getFlaeche());
```

- Für jede der Figuren muss die passende Implementierung von `getFlaeche()` aufgerufen werden.
- Java tut genau das (wählt zur Laufzeit die richtige Implementierung aus).
  Anmerkung: Hier wird in Abhängigkeit zum *Host-Objekt* ausgesucht, nicht wie vorher in dem ärgerlichen Fall bei `equals` und `compareTo` nach der Argumentklasse!

5.5 Polymorphie

5.5.1 Polymorphie: Formen

Eine Operation kann sich für unterschiedliche Klassen (als Host-Objekt oder auch als Argumente) unterschiedlich verhalten.

(polymorph = viele verschiedene Formen/Strukturen habend; (griech.))

- bereits aus der prozeduralen Welt bekannt: $+$ und $-$ sind auf ganze (Integer) und reelle Zahlen (Real) anwendbar außerdem ist $+$ auch für Mengen und Strings verständlich.
- Hier: *getFlaeche()* ist für jede Klasse von Figuren unterschiedlich definiert.
  Bei der Iteration über das Feld wird *getFlaeche()* für jedes Objekt aufgerufen.
  getFlaeche() ist eine abstrakte Methode von *geo_figur*, d.h., die Implementierung findet man jeweils bei den Klassen *Kreis*, *Rechteck* etc.
- Manchmal soll das Verhalten in Abhängigkeit von der Klasse der mitgegebenen Argumente unterschiedlich sein:
  Universität.einstellen(Mitarbeiter) vs. Universität.einstellen(Hiwi)

OVERLOADING (ÜBERLADEN)

Von Overloading spricht man, wenn Eigenschaften/Operationen innerhalb einer Klassenhierarchie mehrmals mit unterschiedlicher Signatur/Implementierung definiert sind (also polymorph sind). Dies kann aus völlig unterschiedlichen Gründen geschehen:

- Von der Intention her gleiche Methoden:
  + für Zahlen, Strings, Mengen ...
- rein zufällig:
  Stadt.getLaenge()/getBreite() (geographische Koordinaten) vs. Auto.getLaenge()
- unmittelbar verwandte Methoden:
  Universität.einstellen(Mitarbeiter) vs. Universität.einstellen(Hiwi) vs. Firma.einstellen(Mitarbeiter) vs. Firma.einstellen(Praktikant).
- Methoden die “dasselbe” unterschiedlich tun: Rechteck.getFlaeche() vs. Kreis.getFlaeche()

⇒ muß nicht immer mit Vererbung/Redefinition zusammenhängen!

FORMEN VON POLYMORPHIE

Ad-hoc-Polymorphismus: Operationen mit denselben Bezeichnungen können für verschiedene Empfängerklassen unterschiedlich definiert sein.
(“+” für Zahlen, Strings, Mengen ...)
Vergleiche $>$,$=$ zwischen zwei Objekten einer Klasse

- Methode ist nicht auf einer gemeinsamen Oberklasse definiert.
- Implementierungen jeweils komplett unterschiedlich

Formen von Polymorphie

parametrischer Polymorphismus: Eine generische Operation kann für verschiedene Klassen instantiiert werden, wobei die Implementierung immer dieselbe ist, die konkreten Klassen sich aber erst aus dem Typ des Host-Objektes ergeben.

Häufig werden Datenstrukturen und Algorithmen parametrisch polymorph implementiert:

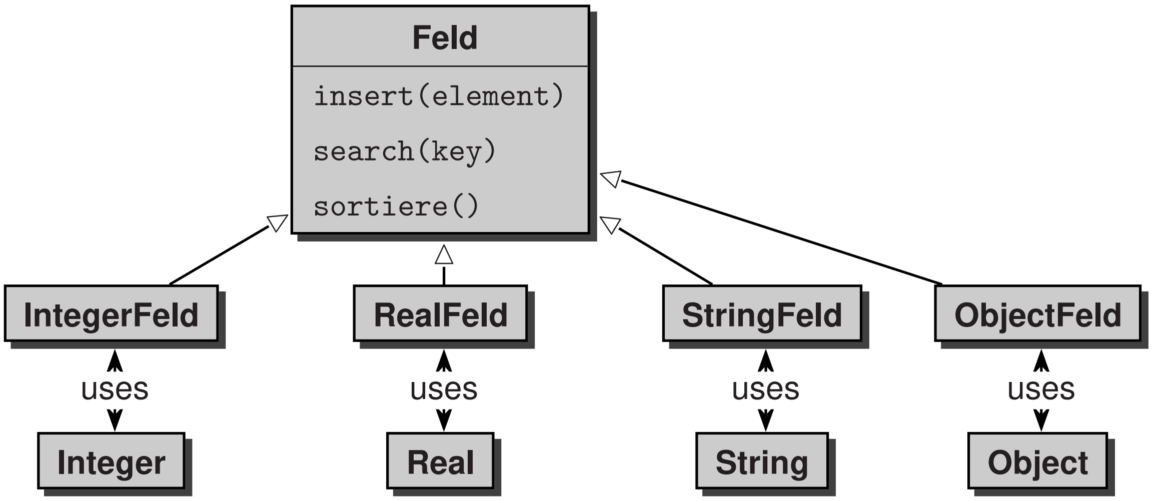
- Sortieralgorithmen
  Es ist dem Algorithmus egal, was er sortiert, er benutzt die von den zu sortierenden Objekten angebotene `compareTo()`-Operation.
- Listen, Bäume (siehe später)

Wichtig ist dabei, dass die jeweils als Parameter verwendeten Objektklassen charakteristische Methoden anbieten,

- Schlüssel, auf denen eine (Teil)Ordnung definiert ist
- Vergleichsoperationen

$\Rightarrow$ basiert auf Ad-Hoc-Polymorphismus der Parameter-Klassen

Parametrischer Polymorphismus



- Die Methoden *insert(element)*, *search(key)* und *sortiere()* sind für *Feld* *generisch* implementiert (*Parametrischer Polymorphismus* der verschiedenen Feld-Klassen).
- Methoden verwenden $\leq$ (in Form von `compareTo()`) der Parameterklasse.

FORMEN VON POLYMORPHIE (FORTS.)

Inklusions-Polymorphismus (Vererbung): Eigenschaften und Operationen, die für Objekte einer Klasse definiert sind, sind für alle Unterklassen dieser Klasse ebenfalls anwendbar.

Fahrzeug.anmelden() ist einmal für alle Fahrzeuge definiert und implementiert, und kann für alle Subklassen PKW, Motorrad, ... verwendet werden.

- in dieser Form wird immer dieselbe Implementierung verwendet
- Basis für Overriding ...

OVERRIDING (ÜBERSCHREIBEN)

Methoden, die aus einer Oberklasse geerbt werden, können in ihren Unterklassen durch eine speziellere Definition *redefiniert*, bzw. *überschrieben* werden (*overriding*):

geo_flaeche.getFlaeche()	// return (pixel abzählen)
Rechteck.getFlaeche()	// return a $\cdot$ b
Quadrat.getFlaeche()	// return a^2

- Häufig: Deklaration als abstrakte Operation der abstrakten Oberklasse, d.h., es wird nur eine Signatur vererbt, aber keine Implementierung.
- Implementierung dann in den einzelnen Subklassen
- Anwendung einer Operation über eine Kollektion von Objekten verschiedener Subklassen.

  GeomFigur.anzeigen():
  Kreis.anzeigen(), Rechteck.anzeigen()

5.5.2 Polymorphie - Auswahl der Methodenimplementierung

Betrachte einen Methoden-Aufruf

my_class x;
x.eine_methode(arg_1, ..., arg_n);

⇒ Over***ing erfordert die Auswahl der richtigen Implementierung (*operation name resolution/operation dispatching*), abhängig von zwei Dingen:

- Klasse des *Host-Objektes* x (also des Objektes, dessen Methode aufgerufen wird),
- Anzahl und Klassen der *Argument-Objekte* arg_1, ..., arg_n.

Man unterscheidet

- abhängig von der Klasse des Host-Objektes (*single dispatch*), oder
- abhängig von der Klasse des Host-Objektes und von den Klassen der Argumente (*multiple dispatch*).

  Universität.einstellen(Mitarbeiter) vs. Universität.einstellen(Hiwi) vs.
  Firma.einstellen(Mitarbeiter) vs. Firma.einstellen(Praktikant).

⇒ Konflikte, wenn es mehrere Oberklassen gibt, von denen geerbt werden kann.

ODMG/JAVA - AUSWAHL DER METHODENIMPLEMENTIERUNG

Java unterstützt eine Zwischenstufe:

- nur eine direkte Oberklasse von der die Implementierung betrachtet werden muss, möglich.
  Hinweis: es sind mehrere direkte Ober-Interfaces möglich, aber die liefern keine Implementierungen.
- Auswahl der Methodenimplementierung nach der Klasse des Host-Objekts *zur Laufzeit*,
- Auswahl der Methodenimplementierung nach der Klasse der Argumente *nach dem Wissen zur Übersetzungszeit*,

Auswahl der Methodenimplementierung: Beispiel

```
public class Firma{
  public void einstellen(Person p){
    System.out.println(p + " als Person bei Firma einstellen");}
  public void einstellen(Angestellter p){
    System.out.println(p + " als Angestellten bei Firma einstellen");}
}
```

Beispiel (Forts.)

```
public class FirmaTest{
  public static void main (String[] args){
    Firma f = new Firma();
    Person otto = new Person("Otto");
    Angestellter hans = new Angestellter("Hans", 100000);
    Object fritz = new Angestellter("Fritz",200000);
    f.einstellen(otto);
    f.einstellen(hans);
    // f.einstellen(fritz); // geht nicht
    // compiler: Explicit cast needed to convert java.lang.Object to Angestellter.
    f.einstellen((Person)fritz);
    Person karl = new Angestellter("Karl", 100000);
    f.einstellen(karl);
    f.einstellen((Angestellter)karl);
    System.out.println("Karl ist " + karl.wasBinIch());
    System.out.println("Karl ist " + ((Angestellter)karl).wasBinIch());
    // System.out.println("Fritz ist " + fritz.wasBinIch()); // geht auch nicht!
    System.out.println("Fritz ist " + ((Person)fritz).wasBinIch());
  }
}
```


Auswahl der Methodenimplementierung: Programmiertips

Wenn

methode`(oberklasse x)`
methode`(unterklasse x)`

zur Auswahl stehen, wird im allgemeinen nicht automatisch die “feinste” anwendbare Methode gewählt.

Deswegen kann man nicht

```
Comparable:   compareTo(Comparable x)
Team:         compareTo(Team x)
```

verfeinern – letzteres wird nicht aufgerufen wenn das Argument zwar ein `Team` ist, aber generisch nur als `Object` bekannt ist (wie beim Sortierer).

Programmiertips (Forts.)

- Auswahl der Implementierung nach Argumentklasse:
  wenn bekannt ist, welche Klassen in Frage kommen, kann man das durch explizites Casting

  ```
  if (x instanceof c1) y.methode((c1)x);
  if (x instanceof c2) y.methode((c2)x);
  ```

  machen (wie im Beispiel eben).

  Normalerweise ist das aber nicht bekannt, wenn man generische Klassen (wie den Sortierer) schreibt.

- Deshalb muss man man ad-hoc-polymorphe Operationen (wie z.B. `equals` oder `compareTo`) immer als

  ```
  neueKlasse: methode(Object other)
  ```

  definieren, und dann im Methodenrumpf mit

  ```
  if (other instanceof neueKlasse)
  { neueKlasse o = (neueKlasse)other;
    ... und dann mit o arbeiten ...}
  ```

  abfragen und absichern.

Zeitpunkt der Auswahl der Methodenimplementierung

- Durch die Angabe von Klassendeklarationen und Variablendeklarationen ist oft schon zur Übersetzungszeit klar, welche Implementierung verwendet werden muss ("Early Binding", "Compile Time Binding")
- Ansonsten wird es (z.B. bei o.g. Iteration) zur Laufzeit festgestellt
  (in diesem Fall wird zur Übersetzungszeit ein Codefragment eincompiliert, das später zur Laufzeit die aktuelle Instanz und die Klassenhierarchie auswertet; *Dynamic method lookup*)
  ("Late Binding", "Runtime Binding")

Java: *Dynamic method lookup* (nach der Klasse des Host-Objektes) wird immer ausgeführt, wenn eine Methode nicht als *static*, *final*, oder *private* deklariert ist, oder für eine *final* class definiert ist.

⇒ Final-Deklaration sorgt für bessere Performance.

- *Kein* *dynamic method lookup* nach den Klassen der Parameter-Objekte. Dies muss man *explizit* unter Verwendung von `instanceof` programmieren (vgl. `compareTo`-Methode).

5.6 Literal/Wert vs. Objekt

- Literale/Werte sind ... nur einfache Werte
- Objekte haben Identität, Struktur und Verhalten und sind in einer Klassenhierarchie geordnet
- sie sind Referenztypen
  - können als call-by-Reference an andere Methoden übergeben werden
  - falls das nicht der Fall sein soll, muss die aufgerufene Methode mit `clone()` eine lokale Kopie machen.
- Die Klassenhierarchie zusammen mit Late Binding erlaubt, mehrere Abstraktionsschritte im Design des Programmes zu berücksichtigen
- Wiederverwendbarer Code
- Generische Datentypen implementiert man im allgemeinen über `Object`

Die Wrapper-Klassen

- Für die Literaltypen existieren entsprechende *Wrapper*-Klassen, die solche Werte als Objekte "verpacken":
  Integer, Long, Float, Double, Boolean, Character.
- erlauben jetzt ebenfalls Call-by-reference
- Damit kann man auch z.B. Iterationen über Objekt-Felder laufen lassen, die Strings und Zahlen gemischt enthalten
- ... jetzt sieht man, dass `String` bereits eine solche Klasse ist.
- Initialisierung entweder über den Wert, oder durch einen String
  z.B. `new Integer(123)` oder `new Integer("123")`.
- Methoden `<basetype>Value` liefern den Literalwert, z.B.

```
Integer x = new Integer("123");
int y = x.intValue();
```

- statische Methoden `<wrapperklasse>.parse<Basetype>(String)` erzeugen primitive Literale aus Strings:

```
int y = Integer.parseInt("123");
```

Die Wrapper-Klassen (Forts.)

Da es sich um Referenz-Klassen handelt, hat man auch hier einen Unterschied zwischen "==" und `equals()`:

```
public class IntegerTest{
  public static void main (String[] args){
     Integer a = new Integer(4);
     Integer b = new Integer(4);
     System.out.println(a + " == " + b + ": " + (a == b));
     System.out.println(a + " equals " + b + ": " + (a.equals(b)));
  }
}
```


Beispiel: Rationale Zahlen - nochmal

```
public class EnhancedRational extends Rational implements Comparable{
  public EnhancedRational(int z) { super(z); }
  public EnhancedRational(int z, int n) { super(z,n); }
  public String toString() { return(Zaehler + "/" + Nenner); }
  public boolean equals(Object other) {
    if (other instanceof Rational)
      { return(getValue() == ((Rational)other).getValue()); }
    else if (other instanceof Float) // Einbettung von Float in Rational ----
      { return (getValue() == ((Float)other).floatValue()); }
    else return false; }
  public Object clone() { return new EnhancedRational(Zaehler,Nenner); }
  public int compareTo(Object other) {
    if (other instanceof Rational)
      { if (getValue() < ((Rational)other).getValue()) return -1;
        if (getValue() > ((Rational)other).getValue()) return 1;
        return 0; }
    else if (other instanceof Float) // Einbettung von Float in Rational ----
      { if (getValue() < ((Float)other).floatValue()) return -1;
        if (getValue() > ((Float)other).floatValue()) return 1;
        return 0; }
    else return 0;    // nicht vergleichbar; AUFPASSEN! ---------------------
} }
```

BEISPIEL: RATIONALE ZAHLEN (TEST)

```
public class EnhancedRationalTest2{
  public static void main (String[] args){
     EnhancedRational a = new EnhancedRational(1,5);
     Float b = new Float(0.2);
     Float c = new Float(0.3);
     System.out.println(a + " equals " + b + ": " + a.equals(b));
     System.out.println(a + " compareTo " + b + ": " + a.compareTo(b));
     System.out.println(a + " equals " + c + ": " + a.equals(c));
     System.out.println(a + " compareTo " + c + ": " + a.compareTo(c));
     // anders herum gehts nicht, weil Integer.compareTo()
     // die Klasse Rational nicht beruecksichtigt!
     // System.out.println(c.compareTo(a));
     // akzeptiert der Compiler nicht.
  }
}
```


Kapitel 6
Datenstrukturen

- kommen ebenfalls als Werte von Eigenschaften vor (als “*Kollektionen*”)
- haben komplexere innere Struktur
- sind *generisch* (Felder von ...) – also normalerweise über `object` (evtl. auch Anforderung des `Comparable`-Interfaces)
- bieten *generische Operationen* an (Einfügen, Zugriffe)
- diese Operationen stützen sich auf dem Verhalten der in der Datenstruktur enthaltenen Objekte/Werte ab
- Datenstrukturen werden *inkrementell* entwickelt
- manchmal auch etwas komplexere Operationen (sortieren)
  $\Rightarrow$ Algorithmen
- Verhalten bezieht sich aber nur “auf sich selbst”

Dynamische Datenstrukturen

- Wenn man ein Array einmal zugewiesen hat (mit new oder Wertzuweisung), kann man nur noch einzelne Elemente austauschen, aber weder anhängen noch entfernen.

$\Rightarrow$ Dynamische Datenstrukturen, können beliebig viele Elemente enthalten

- Was ist die “dynamische” Form eines Arrays?

Die Datenstruktur “Liste”

Was ist eine Liste?

Die Liste vom objektorientierten Standpunkt

- jeder Listeneintrag hat einen Wert (bzw. eine Referenz auf ein Objekt), und einen Zeiger ...
- ... auf die daranhängende Liste (*Rekursion!*)
  Anmerkung: Viele Datenstrukturen sind rekursiv aufgebaut und verwenden rekursive Algorithmen.
- “Liste” ist ein *abstrakter/generischer* Datentyp – man kann Listen über beliebigen Dingen haben.
- man kann sich den Wert geben lassen, oder den Rest der Liste,
- man kann etwas an die Liste anhängen, ein Element vorne/hinten (verarbeiten und) löschen,
- einen bestimmten Wert suchen,
- in vielen Fällen: die Liste sortieren.

6.1 Abstrakte Datentypen

- "Liste" ist eine abstrakte, generische Verhaltensspezifikation
  "Schnittstelle"

⇒ Konzept der "Abstrakten Datentypen" (ADTs)

- unabhängig von der internen Realisierung ("Geheimnisprinzip")
  es kann mehrere konkrete Datentypen (= Implementierungen) zu einem abstrakten DT geben.
- lange vor der "Erfindung" von "Objektorientierung"
- Idee aus der Softwaretechnik, realisiert z.B. in modularen Programmiersprachen (z.B. in Modula, oder in C/C++-Templates)
- schwächer als Objektorientierung: kein Klassenkonzept
- optimal kombinierbar mit Objektorientierung ...
- verschiedene Möglichkeiten, ADTs zu spezifizieren

(vgl. Saake/Sattler: "Algorithmen und Datenstrukturen"; Kapitel 11 und 13)

ALGEBRAISCHE SPEZIFIKATION

- *Signatur*: formale (*syntaktische*) Schnittstelle eines ADT
- Spezifikation der *Semantik* (Funktionalität) durch *Axiome* (Gleichungen).
  ⇒ eine *deklarative Spezifikation*, unabhängig von der operationalen Implementierung.

Beispiel: Liste

Typ: Liste <T> über Typ T

Konstruktoren:
create: → Liste
add: T × Liste → Liste

Operatoren:
head: Liste → T
tail: Liste → Liste
length: Liste → Nat
is_in: T × Liste → Bool

Axiome:

head(create) = $\perp$
head(add(e,ℓ)) = e
tail(create) = $\perp$
tail(add(e,ℓ)) = ℓ
length(create) = 0
length(add(e,ℓ)) = succ (length(ℓ))
is_in(e,create) = false

$$\text{is\_in(e,add(x,}\ell\text{))} = \begin{cases} \text{true,} & \text{falls x=e,} \\ \text{is\_in(e,}\ell\text{)} & \text{sonst.} \end{cases}$$

Begriff: Algebra

Eine *Algebra* ist eine -relativ einfache, und damit anderen Strukturen zugrundeliegende- mathematische Struktur:

- Trägermenge
- Operatoren

Beispiel: Boole'sche Algebra

- Trägermenge $\{$true, false$\}$
- Operatoren "nicht", "und" und "oder"

"freie Algebra" (gegeben durch Signatur):
true: $\rightarrow$ Bool
false: $\rightarrow$ Bool

nicht: Bool $\rightarrow$ Bool
und: Bool $\times$ Bool $\rightarrow$ Bool
oder: Bool $\times$ Bool $\rightarrow$ Bool

"Quotientenalgebra": mit Gleichheiten, die den Operatoren Semantik zuweist:

nicht(true) = false
nicht (false) = true
und (...,...) = ...
oder (...,...) = ...

Algebra

Allgemein:

- Für eine gegebene *Signatur* Σ besteht die *Termalgebra* $Term_\Sigma$ aus allen Termen, die sich aus Σ erzeugen lassen.
- Soweit werden diese Terme also *nicht interpretiert*, d.h. `(true und true) oder false` ist ein solcher Term
- erst durch die Einführung von Axiomen/Gleichungen, werden Terme gleichgesetzt, und man erhält die *Quotiententermalgebra* $Term_\Sigma / =$, die *Äquivalenzklassen* von Termen betrachtet. Dort ist dann `(true und true) oder false` äquivalent zu `true`.
- Man muss dann eine *Normalform* definieren, welcher Term eine Äquivalenzklasse repräsentiert
  (Boolesche Algebra: `true` und `false`).
  Meistens sind die Gleichungen als Ersetzungen links$\rightarrow$rechts zu lesen.

$\Rightarrow$ theoretische Informatik: Reduktionssysteme, Termersetzungssysteme

ADTs beruhen auf der Idee der Quotiententermalgebra.

Algebraische Spezifikation der natürlichen Zahlen

Typ: Nat

Die Trägermenge ist induktiv definiert.

Konstruktoren:
null: $\rightarrow$ Nat
succ: Nat $\rightarrow$ Nat

Normalform: Alle Elemente der Trägermenge lassen sich eindeutig in der Form succ(succ(...succ(null))) darstellen.

Operator: (einer von vielen möglichen)
plus: Nat $\times$ Nat $\rightarrow$ Nat

Axiome:
plus(i,null) = i
plus(i,succ(j)) = succ(plus(i,j))

Aufgabe: Erweitern Sie den Typ Nat um die Operatoren is_null, minus, mult und power (Hinweis: führen Sie mult auf plus zurück).

Algebraische Spezifikation des Datentyps "Liste"

Typ: Liste $<$T$>$ über Typ T

Konstruktoren:
create: $\rightarrow$ Liste
add: T $\times$ Liste $\rightarrow$ Liste

Operatoren:
head: Liste $\rightarrow$ T
tail: Liste $\rightarrow$ Liste
length: Liste $\rightarrow$ Nat
is_in: T $\times$ Liste $\rightarrow$ Bool

Axiome:

head(create) = $\perp$
head(add(e,ℓ)) = e
tail(create) = $\perp$
tail(add(e,ℓ)) = ℓ
length(create) = 0
length(add(e,ℓ)) = succ (length(ℓ))
is_in(e,create) = false

$$\text{is\_in}(e,\text{add}(x,\ell)) = \begin{cases} \text{true}, & \text{falls } x=e, \\ \text{is\_in}(e,\ell) & \text{sonst.} \end{cases}$$

Terme sind also z.B. (für eine Liste über Integers)

- add(5,add(4,add(3,add(2,add(1,create))))), was auch gleich in Normalform ist.
- add(head(add(3,add(2,add(1,create)))), tail(add(3,add(2,add(1,create))))), dessen NF add(3,add(2,add(1,create))) ist.
- add(4,add(3,tail(add(8,add(2,add(1,create)))))).

Zusammenfassung

- Ein abstrakter Datentyp fasst die wesentlichen Eigenschaften und Operationen einer Datenstruktur zusammen, ohne auf deren tatschliche Realisierung im Rechner einzugehen.
  - Prinzip der Kapselung: Ein ADT-Modul darf nur über seine Schnittstelle benutzt werden,
  - Geheimnisprinzip: Die interne Realisierung eines ADT-Moduls ist verborgen.
- Eine Algebra besteht aus einer **Trägermenge** (erzeugt durch die **Konstruktoren**) und **Operationen** darauf.
- Bei abstrakten Datentypen kann man diese Operationen weiter unterscheiden:
  - **Prädikate**: prüfen, ob das/die Argument(e) eine bestimmte Eigenschaft erfüllen,
  - **Selektoren**: lassen das Argument unverändert,
  - **Modifikatoren**: verändern das/eines der Argument(e),
  - **Kombinatoren**: erzeugen ein neues Element der Trägermenge als Ergebnis einer Verknüpfung.
  - Unterscheidung zwischen Modifikatoren und Kombinatoren nicht immer eindeutig.

Axiome

- Für jeden Operator wird die Semantik durch Axiome angegeben
  - linke Seite: alle Möglichkeiten, Argumente in Normalform zu kombinieren, müssen abgedeckt sein
  - rechte Seite: ein vereinfachter Ausdruck (Rückführung auf Konstruktoren oder "einfachere" Operatoren)
  - ggf. Fallunterscheidung innerhalb der Axiome

AUFGABEN ZU ABSTRAKTEN DATENTYPEN/ALGEBREN

- Nehmen Sie die algebraische Spezifikation der natürlichen Zahlen von Folie 366 und erweitern Sie ihn um die Operatoren is_null, minus, mult und power
(Hinweis: führen Sie mult auf plus zurück).
- Nehmen Sie einen Datentyp Real mit den Operationen +,-,* und sqrt (=Quadratwurzel) als gegeben an. Definieren Sie einen abstrakten Datentyp Cplx, der die komplexen Zahlen definiert mit den Operationen plus, mult und abs (Betragsfunktion).

GENERISCHE DATENTYPEN/DATENSTRUKTUREN IN JAVA

Bereits festgestellt:

- Generische Implementierung der Funktionalität über der Klasse `object`.
- Normalerweise rekursive Struktur und Implementierung:
  - jeder Listeneintrag hat einen Wert (vom Typ Object), und einen Zeiger ...
  - auf die daranhängende Liste (*Rekursion!*)
- und dann implementiert man genau die Operatoren:
  - Konstruktoren werden auf den Java-Konstruktor abgebildet
  - Modifikatoren und Selektoren als Methoden.
  - `toString()`, `clone()` und `equals(Object)` implementieren.

6.2 Der Datentyp “Liste”

6.2.1 Einfache Listen

[siehe Definition auf Folie 363]

- Der abstrakte Datentyp “Liste” fasst die wesentlichen Eigenschaften und Operationen der *abstrakten* Datenstruktur “Liste” zusammen, ohne auf deren tatsächliche Realisierung im Rechner einzugehen.
- es gibt viele mögliche Implementierungen
  - auf Basis eines Feldes
  - als verzeigerte Liste

Datentyp “Liste” in Java

```
public class Liste implements Cloneable{
  protected Object the_head = null;
  protected Liste the_tail = null;
  public Liste() {;} // fuer create()
         // dann ist the_head=null und the_tail=null (‘‘Waechterelement’’)
  public Liste(Object o, Liste l) {
         the_head = o;
         the_tail = l; }
  public Liste add(Object o) { return new Liste(o,this);}
  public Liste add(int i) { return new Liste(new Integer(i),this);}
  public boolean is_empty() {return the_head == null;}
  public Liste tail() {return the_tail;}
  public Object head() {return the_head;}
  // weitere Methoden folgen
```

(bitte umblättern)

Liste (Forts.)

```
  // es folgen noch die allgemeinen Objekt-Methoden:
  public String toString() {
    if (is_empty()) return ".";
    else return (head() + " " + tail()); }
  public Object clone() {
    if (is_empty()) return new Liste();
    else return new Liste(head(),(Liste)(tail().clone()));}
  public boolean equals(Object other) {
    if (other instanceof Liste)
    { Liste o = (Liste)other;
      if (is_empty() || o.is_empty())
        return (is_empty() && o.is_empty());
      return (head().equals(o.head()) && tail().equals(o.tail()));
    }
    return false; }
}
```

- `clone()` macht nur shallow-copies
- wichtig: `equals()` benutzen, um head-Objekte zu vergleichen!

Datentyp "Liste": Test

... mit Integers:

```
public class ListeTest{
  public static void main (String[] args){
    Liste my_liste = new Liste().add(1).add(2).add(3);
    System.out.println("1: " + my_liste);
    Liste my_second_liste = my_liste.add(4);
    my_liste = my_liste.add(5);
    System.out.println("1: " + my_liste);
    System.out.println("2: " + my_second_liste);
    System.out.println("tail(2): " + my_second_liste.tail());
    Liste my_third_liste = ((Liste)(my_second_liste.tail().clone())).add(4);
    System.out.println("3: " + my_third_liste);
    System.out.println("1 eq 2: " + my_liste.equals(my_second_liste));
    System.out.println("2 eq 3: " + my_second_liste.equals(my_third_liste));
    System.out.println("2 == 3: " + (my_second_liste == my_third_liste));
} }
```

IMPLEMENTIERUNGSHINWEISE

(I) die Spezifikation des Rückgabetyps `Liste` bei allen Konstruktoren und Modifikatoren erlaubt die Erzeugung von Listen als Terme:

```
my_liste = new Liste().add(1).add(2).add(3);
```

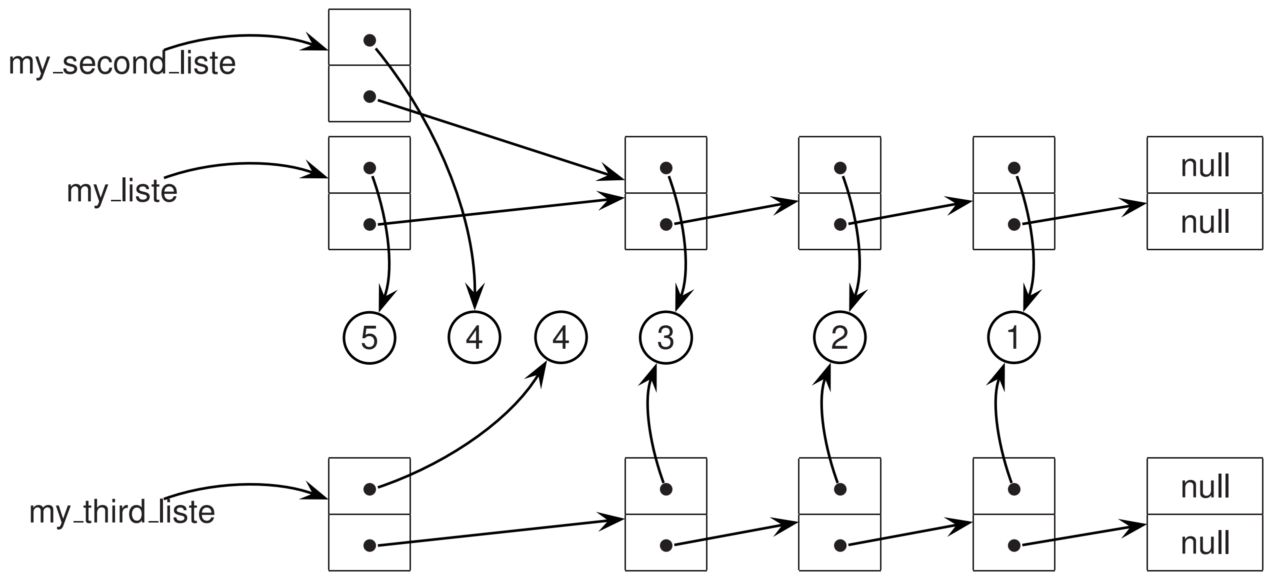
(II) Häufige Implementierungsstrategie: "Wächterelement" am Ende:

- die leere Liste ist `Liste(null,null)`.
  - `new()` gibt keine null-Referenz zurück, sondern etwas das dieselbe Struktur wie jede Liste hat.
  - darauf sind alle Liste-Methoden anwendbar.

  ⇒ homogene, übersichtliche Anwendungsprogrammierung
- eine einelementige Liste ist als `Liste(element,Liste(null,null))` dargestellt.
- dies erspart eine Fallunterscheidung, die notwendig wäre, um eine einelementige Liste als `Liste(element,null)` darzustellen.

Beispielgrafik zu ListeTest.java

Das Programm ListeTest.java erzeugt die folgende Situation:



Hinweis: die Objekte 1,2,3 werden nicht geklont, sondern von beiden Listen gemeinsam genutzt. Die 4 wird zweimal unabhängig erzeugt.

Implementierungshinweise für Fortgeschrittene

(nur eingeschränkt Info-I-verständlich)

- Tiefkopieren mit `head().clone()` wird vom Compiler nicht (mehr) akzeptiert:
- Die Klasse `Object` deklariert eine "native" `clone()`-Methode als `protected`.
- Die Angabe des Interfaces `Comparable` (das ganz anders funktioniert, als "normale" Interfaces) signalisiert, dass eine Klasse diese `clone()`-Methode ihrer Oberklasse nutzen will.
- Die Klasse muss diese dann mit einer `public`-Methode überschreiben.
- Sinnvoll wäre also eine Definition

  ```
  public class CloneableAndComparableObject implements Cloneable, Comparable {
   public CloneableAndComparableObject clone() { // real Java programmers do it like this
     CloneableAndComparableObject neu = super.clone();
     // weitere Eigenschaften setzen
     return neu;}
   public CloneableAndComparableObject compareTo(Object other) {...}
  }
  ```

  und diese dann verwenden, indem man jede Anwendungsklasse als

  ```
  public class <irgendwas> extends CloneableAndComparableObject {...}
  ```

  definiert.
- Dies geht aber nur, wenn man sich darauf verlassen kann, dass alle Objektklassen so definiert sind.
- Generische Datentypen über Object lassen sich implementieren, indem man `instanceof Cloneable` abfragt und dann geeignet castet - wozu das Programm aber über *Reflection* die genaue Klasse des Objektes herausbekommen muss. Das ist aber erst recht nicht mehr Stoff für die Info-I.

AUFGABEN

1. Geben Sie eine algebraische Spezifikation für das Aneinanderhängen ("concat") zweier Listen.
   Ergänzen Sie Liste.java entsprechend.
2. Implementieren Sie den Datentyp "Liste" ohne Wächterelement.
3. Implementieren Sie einen abstrakten Datentyp Feld (indem Sie IntegerFeld.java nehmen und anpassen (Sie können alle Sortierverfahren außer Quicksort dabei weglassen).

   Implementieren Sie dann IntegerListe neu, wobei intern ein Feld verwendet wird.
   - Gehen Sie von einem Array mit 10 Einträgen aus,
   - wenn die Liste durch `add` zu lang wird, ersetzen Sie die interne Repräsentation durch ein Array doppelter Länge.

6.2.2 Abstrakter Datentyp “Stack”

- “Kellerspeicher”, z.B. benutzt als Aufrufstack in Java, oder abstrakt zur Behandlung von Rekursion
  “Last-in-First-out”

Typ: Stack <T>

Operatoren:
create: $\rightarrow$ Stack
push: Stack $\times$ T $\rightarrow$ Stack
pop: Stack $\rightarrow$ Stack
top: Stack $\rightarrow$ T
is_empty: Stack $\rightarrow$ Bool

Axiome:
pop(push(s,x)) = s
pop(create) = $\perp$
top(push(s,x)) = x
top(create) = $\perp$
is_empty(create) = true
is_empty(push(s,x)) = false

- Es wird nur am Anfang der Datenstruktur operiert, und es sind prinzipiell dieselben Operationen wie bei der “Liste”.

Auswertung arithmetischer Terme mit Stack

Auf Folie 54 wurde eine Grammatik für arithmetische Terme vorgestellt.

Betrachten Sie den Term ((3+(4*5))*(7-4)).

– Auswerten des Terms von links nach rechts.
– Noch nicht auswertbare Termteile werden auf den Stack geschoben
– bei schließenden Klammern kann der oberste Ausdruck auf dem Stack ausgewertet werden

Term	Stack	
((3+(4*5))*(7-4))	leer	
(3+(4*5))*(7-4))	)	
3+(4*5))*(7-4))	))	
+(4*5))*(7-4))	3))	
(4*5))*(7-4))	+3))	
4*5))*(7-4))	)+3))	
5))(7-4))	4)+3))	
5))*(7-4))	*4)+3))	
))*(7-4))	5*4)+3))	
)*(7-4))	(5*4)+3))	Auswerten
	20+3))	

Term	Stack	
*(7-4))	(20+3))	Auswerten
	23)	
(7-4))	*23)	
7-4))	)*23)	
-4))	7)*23)	
4))	-7)*23)	
))	4-7)*23)	
)	(4-7)*23)	Auswerten
)	3*23)	Minus rückwärts!
	(3*23)	Auswerten
	69	

Spezifikation des abgeleiteten DT

Ein weiteres Axiom, das die Auswertung beschreibt:

push(")",s) = push(apply(top(pop(s)),top(pop(pop(s))),top(s)), pop(pop(pop(pop(s)))))

und schon hat man eine Normalform, die jeden wohlgeformten arithmetischen Term auf einen Integer-Wert abbildet.

Aufgabe: verfolgen Sie den Stack für den oben beschriebenen Term.

Aufgabe

Implementieren Sie erst den Datentyp "Stack" als Subklasse von "Liste" und leiten Sie davon wiederum den spezielleren Datentypen "ArithmeticTermStack" mit der verfeinerten push-Operation als Subklasse ab.

- In der Eingabe sind nur Zahlen von 1 bis 9 erlaubt.
- Lesen Sie den Term als Folge von Zeichen mit KeyBoard.readChar() ein, die Sie wahlweise erst in einem Array ablegen oder direkt auf den/die Stacks verteilen.
- Benutzen Sie eine kleine Hilfsklasse, die einzelne Zeichen in Zahlen umwandelt.
- Optional: erlauben Sie auch mehrstellige Zahlen in der Eingabe.

Auswertung arithmetischer Terme mit 2 Stacks

- Die Idee war schon mal ganz gut ...
- ... aber der Stack muss Elemente verschiedener Datentypen (Zahlen und Operatoren) aufnehmen.
- Es gibt eine elegantere Möglichkeit:
  - Einen Stack für Zahlen,
  - Einen Stack für Operatoren

Auswertung arithmetischer Terme mit 2 Stacks (Forts.)

Betrachten Sie wieder den Term ((3+(4*5))*(7-4)).

– Auswerten des Terms wieder von links nach rechts.
– Noch nicht auswertbare Operanden werden auf den Wert-Stack geschoben
– Noch nicht auswertbare Operatoren werden auf den Op-Stack geschoben
– bei schließenden Klammern kann der oberste Operator auf die beiden oberen Operanden angewendet werden

Term	Op-Stack	Werte-Stack	
((3+(4*5))*(7-4))	leer	leer	
(3+(4*5))*(7-4))	leer	leer	
3+(4*5))*(7-4))	leer	leer	
+(4*5))*(7-4))	leer	3	
(4*5))*(7-4))	+	3	
4*5))*(7-4))	+	3	
5))(7-4))	+	4 3	
5))*(7-4))	* +	4 3	
))*(7-4))	* +	5 4 3	

Term	Op-Stack	Werte-Stack	
)*(7-4))	+	20 3	Ausw.
*(7-4))		23	Ausw.
(7-4))	*	23	
7-4))	*	23	
-4))	*	7 23	
4))	- *	7 23	
))	- *	4 7 23	
)	*	3 23	Ausw.
		69	Ausw.

Später (Folie 423) wird ein weiteres, rekursives, Auswertungsverfahren, das eine andere Datenstruktur nutzt, vorgestellt.

6.2.3 Weitere Ergänzungen der Liste

- Bisher wurde nur am Anfang der Liste operiert.
- Anhängen am Ende der Liste:

  append: Liste $\times$ T $\rightarrow$ Liste
  append(create,x) = add(x,create)
  append(add(y,ℓ),x) = add(y,append(ℓ,x))

  Direkte Implementierung hat linearen Aufwand
  (man muss beim Anhängen jedesmal die ganze Liste bis zum Ende durchlaufen).

  Ausweg:

  – zusätzliches Datenfeld, das auf das Wächterelement zeigt
  (dieses ist also hier sehr nützlich)

  – Konstruktor muß dieses Datenfeld initialisieren

  – geeignete Anhängen-Operation

Ergänzungen der Liste (Forts.)

- Entfernen `Liste remove(Object)` aus der Liste,

  remove: Liste $\times$ T $\rightarrow$ Liste
  remove(create,x) = create
  remove(add(x,ℓ),x) = ℓ
  remove(add(y,ℓ),x) = add(y,remove(ℓ,x))

  – Wenn das Element gefunden ist, muß die `tail`-Referenz des *vorhergehenden* Elementes angepasst werden.

- also sollte man eine vorwärts *und* rückwärts verzeigerte Liste haben.
- Als Navigation werden standardmäßig `next()` (anstatt `tail()`) und `previous()` benutzt.

Doppelt verlinkte Liste mit Ende-Zeiger

```
public class BiDiListe extends Liste{
  protected BiDiListe the_end;
  protected BiDiListe the_vorgaenger = null;
  public BiDiListe(Object o, BiDiListe l) {
    super(o,l);  // Zusicherung: l ungleich null
    the_tail = l;    the_end = l.the_end;
    l.the_vorgaenger = this; }
  public BiDiListe() { the_end = this; }
  public BiDiListe append(Object o) {
    BiDiListe letztes = this.the_end.the_vorgaenger;
    BiDiListe tmp = new BiDiListe(o,this.the_end);
    letztes.the_tail = tmp;
    tmp.the_vorgaenger = letztes;
    return this; }
  public BiDiListe append(int i) { return append(new Integer(i)); }
  public BiDiListe previous() {return the_vorgaenger;}
  public BiDiListe next() {return (BiDiListe)the_tail;}
```

Doppelt verlinkte Liste: neue Manipulatoren

```
  // Fortsetzung von eben ...
  public BiDiListe remove(Object o) {  // o kann mehrfach vorkommen
    if (is_empty()) return this;
    if (the_head.equals(o)) {
       BiDiListe neuer_tail = next().remove(o);
       neuer_tail.the_vorgaenger = the_vorgaenger;
       return neuer_tail; }
    else { the_tail =  next().remove(o);
           return this; } }
  public BiDiListe remove(int i) { return remove(new Integer(i)); }
  public Object last() { return the_end.head(); }
  public BiDiListe remove_last() {
    if (is_empty()) return this;
    if (the_end.previous().previous() == null)
       { the_end.the_vorgaenger = null; return the_end; }
    the_end.previous().previous().the_tail = the_end;
    the_end.the_vorgaenger = the_end.previous().previous();
    return this; }
```


Doppelt verlinkte Liste: Anpassungen vorhandener Operationen

```
  // add anpassen, dass es auf BiDiLists arbeitet:
  // Hinweis: als Return-Datentyp darf *nicht* BiDiListe angegeben
  //   werden, da die Oberklasse "Liste" diese Methode bereits
  //   definiert (keine Verfeinerung des Rueckgabetyps erlaubt)
  public Liste add(Object o) { return new BiDiListe(o,this); }
  public Liste add(int i) { return add(new Integer(i)); }
  // toString() und equals() unveraendert
  public Object clone() {
    if (is_empty()) return new BiDiListe();
    else return new BiDiListe(head(), (BiDiListe)(next().clone()));}
     // Hinweis: clone liefert "Object" zurueck, dann als BiDiListe casten
  // jetzt kann man es auch rueckwaerts ausgeben (zum testen ...)
  public void printRueckwaerts() {
    System.out.println(toStringRueckwaerts(the_end.previous())); }
  public String toStringRueckwaerts(BiDiListe x) {
    if (x.previous() == null) return (x.head() + " .");
    else return (x.head() + " " + toStringRueckwaerts(x.previous())); }
}
```

Datentyp “Bidirektionale Liste”: Test

```
public class BiDiListeTest{
  public static void main (String[] args){
    BiDiListe my_liste =
       (BiDiListe)((BiDiListe)(new BiDiListe().add(2))).append(3).add(1);
    System.out.println(my_liste);
    BiDiListe my_second_liste = ((BiDiListe)(my_liste.clone())).append(4);
    System.out.println("2nd: " + my_second_liste);
    my_liste.append(4);
    System.out.print("myListe rueckwaerts: ");
    my_liste.printRueckwaerts();
    System.out.println("1 equals 2: " + my_liste.equals(my_second_liste));
    my_liste =  my_liste.remove(4);
    my_liste =  my_liste.remove_last();
    System.out.println(my_liste);
    my_second_liste =  my_second_liste.remove_last();
    my_second_liste =  my_second_liste.remove(1);
    System.out.println(my_second_liste);
} }
```


KOMMENTARE

- explizites Casting von Oberklassen in speziellere Unterklassen notwendig, wenn Methoden der Oberklasse verwendet werden
- Die Benennung `tail()` für die normale Liste und `next()` für die doppelt verzeigerte Liste spart schon einige Castings
- Da Stack eine eigene Benennung der Operationen verwendet, muss der “Anwender” dieser Datenstrukturen nicht casten.
- die Rückgabe der Liste bei allen Konstruktoren und Modifikatoren erlaubt die Erzeugung von Listen als Terme:

  ```
  (BiDiListe)((BiDiListe)(new BiDiListe().add(2))).append(3).add(1);
  ```

- Man kann den Rückgabewert auch ignorieren und nur

  ```
  my_liste.append(4);
  ```

  schreiben.
- Anmerkung: so wie diese Terme sieht *funktionales Programmieren* (LISP, Haskell, Scheme) aus

ABSTRAKTER DATENTYP "QUEUE"

- "Warteschlange", "First-in-First-out"

Typ: Queue <T>

Operatoren:
create: $\rightarrow$ Queue
enqueue: Queue $\times$ T $\rightarrow$ Queue
dequeue: Queue $\rightarrow$ Queue
first: Queue $\rightarrow$ T
is_empty: Queue $\rightarrow$ Bool

Axiome:
dequeue(enqueue(create,x)) = create
dequeue(enqueue(enqueue(q,y),x)) =
    = enqueue(dequeue(enqueue(q,y)),x)
first(enqueue(create,x)) = x
first(enqueue(enqueue(q,y),x)) =
    = first(enqueue(q,y))
is_empty(create) = true
is_empty(enqueue(q,x)) = false

- Axiome arbeiten sich rekursiv bis zum Ende durch

Aufgabe

Implementieren Sie den Datentyp "Queue" auf Basis der bidirektionalen Liste.

AUFGABEN

Weitere Listenoperationen

Definieren Sie die folgenden Operationen:

- Umdrehen einer Queue ("reverse")
- Finden des größten/kleinsten Elementes in einer Liste ("max"/"min")
- Sortieren einer Liste

Sortieren einer Liste

Implementieren Sie Quicksort für die doppelt verlinkte Liste.

Hinweise:

- Die Zeiger von links und rechts sind relativ einfach zu implementieren und entlang der Listenstruktur laufen zu lassen.
- Vertauschen: man muss nur die Inhalte der Listenelemente (`the_head`-Referenz) vertauschen.

6.3 Allgemeine Collections

Als *Collection* bezeichnet man Strukturen, mit denen viele einzelne Objekte organisiert werden können.

- unterschiedliches *äußeres* Verhalten: *Datentyp* (= Zugriffsoperationen, Signatur)
  - Liste, Stack, Queue:
    offensichtlich lineare Datentypen/-strukturen; naheliegende Abbildung auf Implementierungen
  - ... es gibt außer Listen/linearen Kollektionen noch weitere Arten:
  - Prioritätswarteschlange: auf höchstpriores Element einer Kollektion zugreifen
  - Menge/Set: jedes Element nur einmal vorhanden;
    Enthaltensein, Einfügen, Löschen, Differenz, Vereinigung, Schnitt
  - geordnete Menge, Multimenge, geordnete Multimenge
  - "Dictionary": Zugriff nach einem bestimmten Schlüsselwert auf einen größeren Datensatz
  - geordnetes Dictionary
- unterschiedliche *interne* Realisierungen: *Datenstruktur*

Aufgabe: Datentyp "Menge"

- Liste: geordnete Kollektion, Duplikate erlaubt
- Menge: ungeordnete Kollektion, keine Duplikate

Spezifizieren Sie den abstrakten Datentyp "Menge", der die folgenden Operationen unterstützen soll (Klassifizieren Sie diese als Selektoren/Prädikate/etc):
Hinzunehmen eines Elementes, Entfernen eines Elementes, Mächtigkeit, Enthaltensein, Teilmenge, Vereinigung, Mengendifferenz, Schnittmenge.

Hinweis:

- Beachten Sie, dass Sie zum "Hinzufügen" eines Elementes sowohl einen Konstruktor, als auch eine Operation benötigen, bei der vor dem engültigen Einfügen noch überprüft wird, ob das Element bereits in der Menge enthalten ist.

6.3.1 Iteratoren

- Oft will man irgendetwas für alle Elemente einer beliebigen *Kollektion*) tun (z.B. Adressen aller gespeicherten Personen ausgeben)
- *Iteratoren* bieten ein *generisches* Framework, um beliebige Datenstrukturen zu durchlaufen (Signatur in `java.util.Iterator` als *Interface* vorgegeben):
  - hasNext(): gibt es noch weitere Elemente?
  - next(): schaltet weiter
  - remove(): entfernt das aktuelle Element aus der Datenstruktur [nicht immer unterstützt]
- damit kann man jeden Iterator mit einer Schleife ansteuern:

```
my_iterator = ... // Iterator zu einer Datenstruktur erzeugen;
while (my_iterator.hasNext())
  { object item = my_iterator.next();
    <do something with item>          }
```

- natürlich muss der Iterator passend zur Datenstruktur implementiert sein.
- Häufig bieten Datenstrukturen Methoden an, die Iteratoren erzeugen und zurückgeben.

ITERATOR ÜBER LISTE

```
public class ListenIterator implements java.util.Iterator
{ protected Liste currentNode = null;
  Liste my_liste;
  public ListenIterator(Liste l) { my_liste = l;}
  public boolean hasNext() {
    if (currentNode == null)
    { return (!(my_liste.is_empty())); }
    return (!(currentNode.tail().is_empty())); }
  public Object next() {
    if (!(hasNext())) throw new java.util.NoSuchElementException();
    if (currentNode == null) { currentNode = my_liste; }
    else currentNode = currentNode.tail();
    return (currentNode.head());   }
  public void remove() { System.out.println("Not yet implemented"); }
}
```

- next() gibt Object zurück, also ggf. Casting erforderlich

Benutzung von Iteratoren

```
public class ListenIteratorTest{
  public static void main (String[] args){
    Liste my_liste = new Liste().add(4).add(3).add(2).add(1);
    System.out.println(my_liste);
    ListenIterator my_iterator = new ListenIterator(my_liste);
    while (my_iterator.hasNext())
       { Object item = my_iterator.next();
         System.out.println("naechstes: " + item);
       }
} }
```

Analog ist auch eine `for`-Schleife möglich
(erzeugt den Iterator lokal im `for`-Statement):

```
for (Iterator it = new ListenIterator(my_liste); it.hasNext(); )
    { object item = my_iterator.next();
      <do something with item>
    }
```


6.3.2 Java Collections

Das Java-Paket `java.util` enthält einige Klassen, die “Collections” bereitstellen.

- “Collection” ist ein generisches Interface
- “List” ist ein davon abgeleitetes Interface (*Datentyp*)
- Klassen, die “List” implementieren: “ArrayList” und “LinkedList” (*Datenstrukturen*)

Signatur einiger Methoden von “Collection”:

```
public boolean add(Object obj)
public boolean addAll(Collection coll)
public boolean remove(Object obj)
public boolean removeAll(Collection coll)
public int size()
public boolean isEmpty()
public boolean contains(Object o)
public Iterator iterator()
```

Es gibt keinen Konstruktor, da Collection nur ein Interface ist!

Komplett: `http://java.sun.com/j2se/1.3/docs/api/java/util/Collection.html`

Iteratoren zu Datenstrukturen

- Das Interface `Collection` definiert eine Methode `iterator()`, mit der man einen Iterator über die entsprechende Kollektion erhält:

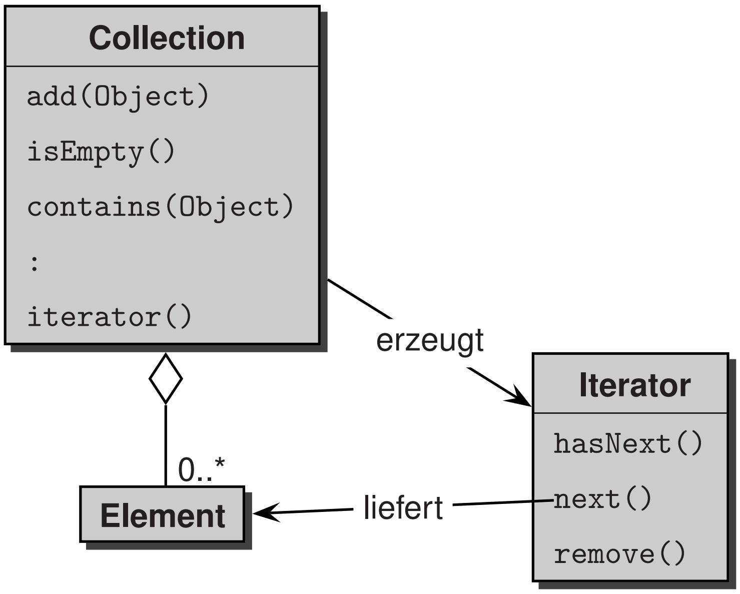
```
public class XXXMitIterator implements Collection {
   public class XXXIterator implements java.util.Iterator {
     // wie oben: lokale Iteratorklasse, die das Interface implementiert
   }
   public Iterator iterator() {...}
   //
   // Methodendefinitionen der Datenstruktur XXX
}
```

Der Aufruf lautet dann nur noch

```
Iterator my_iterator = my_XXX.iterator();
// ... benutze Iterator ...
```

Wenn man irgendwann eine Re-Implementierung eine andere Datenstruktur verwendet, muss der Code nicht geändert werden, da ein generischer Iterator verwendet wird.

Kollektionen und Iteratoren: Klassendiagramm



AUFGABE

Implementieren Sie eine Iteratorklasse für die doppelt verkettete Liste, die zusätzlich die folgende Funktionalität unterstützt:

- Rückwärtslaufen (`hasPrevious()` und `previous()`)
- überlegen Sie sich, warum das Löschen des aktuellen Elementes `remove()` problematisch sein kann. (aus diesem Grund kapselt man "professionelle" Implementierungen noch tiefer in eine äußere Datenstruktur)

Integrieren Sie diese Iteratorklasse in eine Klasse `BiDiListeMitIterator`.

ANMERKUNG

Bei komplizierteren Datenstrukturen und Anwendungen ist `next()` im allgemeinen nicht so einfach:

- Die Iteratorschritte folgen nicht notwendigerweise direkt den Referenzen in der Datenstruktur,
- Ein Iterator kann zusätzlich mit einem Test ausgestattet werden, um nur solche Knoten zu liefern, die den Test erfüllen.

JAVA: LIST

Das Interface `java.util.List` entspricht dem abstrakten Datentyp "Liste", wobei zusätzlich Einfügen und Zugriff an einer gegebenen Position möglich sind:

Signatur der Methoden von "List":

```
public boolean add(int i, Object obj)
public Object remove(int i)              // gibt o zurueck
public Object set(int i, Object o)       // gibt o zurueck
public Object get(int i)
public int indexOf(Object obj)
public ListIterator iterator()
```

- `remove` ist polymorph: sowohl `remove(Object o)` von Collection, als auch `remove(int i)` sind zugreifbar; unterschiedliche Ergebnistypen!
- man bekommt einen `ListIterator`, der auch Navigation rückwärts mit `previous()` erlaubt
- Implementierungen werden dann von den Klassen `ArrayList` und `LinkedList` angeboten.

Iterative Durchlaufen einer Liste

```
import java.util.*;
public class LinkedListTest{
  public static void main (String[] args){
    java.util.List l = new java.util.LinkedList();
    l.add(new Integer(1));
    l.add(new Integer(2));
    l.add(new Integer(4));
    l.add(2,new Integer(3));
    Iterator it = l.iterator();
    while (it.hasNext())
       { System.out.println(it.next());
       }
} }
```


Java: Liste aus vergleichbaren Objekten

Für Instanzen von Klassen, die das `Comparable`-Interface (das die Methode `compareTo(Object o)` anbietet; siehe Folie 329) unterstützen, bietet `java.util.Collection` weiterhin die folgenden *Klassen*-Methoden an:

- `int binarySearch(List l, Object key)`
  Die Liste muß dabei bereits sortiert vorliegen.
- `Object min(Collection c)`
- `Object max(Collection c)`
- `void sort(List l)`

Ein Aufruf wäre also z.B.

```
java.util.List my_liste = ...;
java.util.Collections.sort(my_liste);
System.out.println("Das Element mit dem Schluessel 42 finden
    Sie an Position" +
    java.util.Collections.binarySearch(my_liste, 42));
```

PORTABILITÄT ?

Beim Benutzen dieser (und ähnlicher) Klassen und Interfaces stellt man fest

- je nachdem welches Buch man verwendet, werden unterschiedliche Klassen beschrieben (wobei man manchmal den Eindruck hat, dass der/die Autoren auch nicht alles was sie schreiben ausprobiert haben)
- Java/JDK-Versionen haben unterschiedliche Klassen mit unterschiedlichen Signaturen,
- z.B. ältere Java/JDK-Versionen: Vector mit Enumeration und Stack,
- Collections und List mit Iterator seit JDK 1.2,
- wenn man fremde Java-Tools verwendet, verlangen diese jeweils *bestimmte* Versionen ...
- ... die untereinander inkompatibel sind.
- auch die eigenen Programme werden in 1-2 Jahren nicht mehr lauffähig sein.

Wo ist da der Fortschritt gegenüber C++ (wo das übrigens genau dasselbe ist, und man bei Java alles besser machen wollte)?

- Java ist 20-30 mal langsamer

... also zurück zur Theorie ...

6.3.3 Fazit

In diesem Kapitel wurden nicht nur abstrakte Datentypen besprochen, sondern insbesondere dokumentiert, wie unterschiedliche Funktionalität für *lineare dynamische Datenstrukturen* inkrementell auf Basis einer einfachen linearen Struktur entwickelt wurde.

- Datentypen *abstrakt* und *inkrementell* entsprechend den Anforderungen zu entwickeln
- wenn man weiß was man will, und das klar formulieren kann, ist nachher die eigentliche Programmierung einfach.

6.4 Nichtlineare Datenstrukturen: Bäume

Ein Baum ist auch eine rekursive Datenstruktur

- besteht aus einer Wurzel ...
- ... an der mehrere Bäume hängen.

Motivation

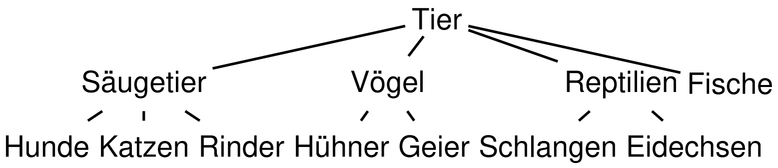
- Binäre Suche ist eine "typische" Anwendung für eine große Klasse von Bäumen
- Mit einer relativ einfachen, aber eher untypischen baumartigen Datenstruktur ist SelectionSort in $O(n \log n)$
- Mit einer etwas komplizierteren, aber typischen baumartigen Datenstruktur ist InsertionSort in $O(n \log n)$
- Mit Untersuchungen von Baum-Algorithmen kann man ganze Bücher füllen

... in dieser Vorlesung sind Bäume bereits verschiedentlich vorgekommen: Ableitungsbäume bei Grammatiken, Ableitungs- und Auswertungsbäume bei Termen/Formeln, Aufrufbäume bei Fibonacci ...

6.4.1 Die Baumstruktur

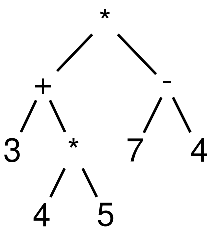
Beispiele: Stammbaum, Vererbungshierarchie, Begriffshierarchien, Buchkapitel, Dateisystem, Ableitungsbäume in Grammatiken, (arithmetische) Terme, Programme

Baum einer Vererbungshierarchie:

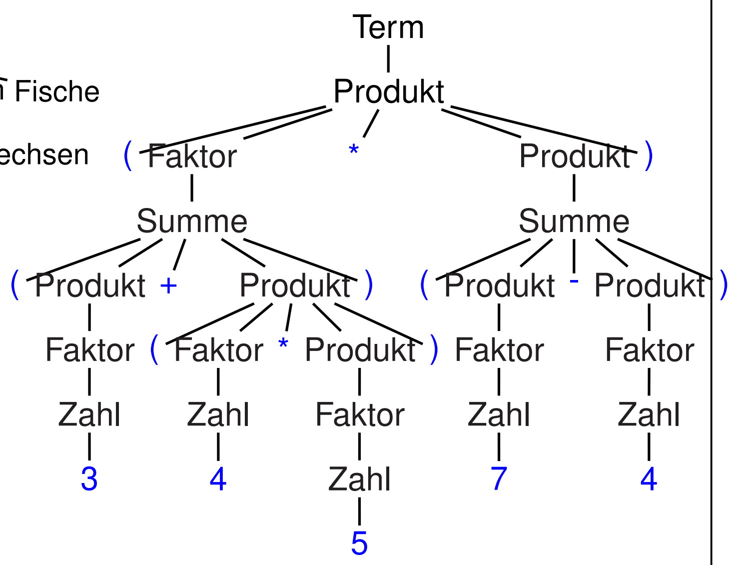


Arithmetischer Term als Baum:

$((3 + (4 * 5)) * (7 - 4))$



Ableitungsbaum (vgl. Folie 55)



6.4.2 Die Baumstruktur

- *Wurzelknoten*
- verbunden mit keinem, einem, oder mehreren *Knoten* auf der ersten Ebene
- jeder Knoten dieser Ebene ist Wurzel eines Unterbaums (“Vaterknoten”/“Kindknoten”)
- *Blätter* sind Knoten, die keine weiteren Kinder haben
- *innere Knoten* sind Knoten, die nicht die Wurzel sind, und auch keine Blätter sind
- Maximale Anzahl von (direkten) Kindern eines Knotens: *(Verzweigungs)grad* des Baumes
- die Anzahl der Ebenen (= maximale Anzahl Schritte von der Wurzel zu einem Blatt) ist die *Höhe* des Baumes

Eigenschaften (u.a.)

- Ein Baum ist eine zusammenhängende Datenstruktur
- es gibt immer genau einen Pfad zwischen der Wurzel und jedem Knoten

Arten von Bäumen

- Wo sind die *Daten* gespeichert?
  - in allen Knoten
  - nur in den Blättern (innere Knoten enthalten in diesem Fall Navigationsinformation)
- unterschiedliche Verzweigungsgrade
  im Prinzip spielen nur 2 Möglichkeiten eine Rolle: 2 (*Binärbäume*) und “sehr groß”
- Sind die Kinder untereinander geordnet oder ungeordnet?

Äquivalente Strukturen

- Schachtelung (vgl. Begriffshierarchie, Directory-Struktur, Programm)
  (Programm als geordneter Baum mit beliebig hohem Verzweigungsgrad)
- Klammerstruktur (vgl. Programm, Term)

... entsprechend unterschiedliche grafische oder textuelle Repräsentation.

Prominentes Beispiel: HTML/XML

(Geordneter) Baum als Java-Klasse

```
public class Baum {
  protected Object contents = null;
  protected Baum[] children;
  public Baum(int k) { children = new Baum[k];}
  public Object getContents() { return contents; }
  public Baum getChild(int i) { return children[i-1]; }
  public Baum setContents(Object o) { contents = o; return this; }
  public Baum setChild(int i,Baum b) { children[i-1] = b; return this; }
  public Baum deleteChild(int i) { children[i-1] = null; return this; }
  public int height() { int height = 0;
    for (int i=0; i < children.length; i++)
        if (children[i].height() > height) height = children[i].height();
    return height+1; }
  public String toString() { // eine von mehreren Moeglichkeiten
    String the_children = "";
    for (int i=0; i < children.length; i++) the_children += children[i].toString();
     return "[" + contents + the_children + "]"; }
  // analog fuer clone() und equals()
}
```

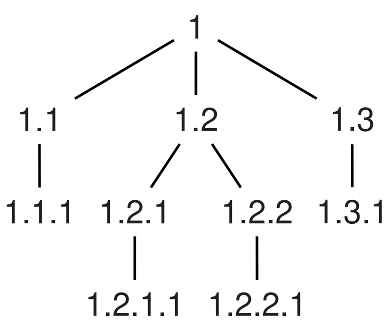
Baum: Beispiel

```
public class BaumTest{
  public static void main (String[] args){
    Baum my_baum = new Baum(3);
    my_baum.setContents(new String("1"));
    my_baum.setChild(1,(new Baum(1).setContents(new String("1.1"))));
    my_baum.getChild(1).setChild(1,new Baum(0).setContents(new String("1.1.1")));
    my_baum.setChild(2,(new Baum(2).setContents(new String("1.2"))));
    my_baum.getChild(2).setChild(1,(new Baum(1).setContents(new String("1.2.1"))));
    my_baum.getChild(2).getChild(1).setChild(1,new Baum(0));
    my_baum.getChild(2).getChild(1).getChild(1).setContents(new String("1.2.1.1"));
    my_baum.getChild(2).setChild(2,new Baum(1).setContents(new String("1.2.2")));
    my_baum.getChild(2).getChild(2).setChild(1,new Baum(0));
    my_baum.getChild(2).getChild(2).getChild(1).setContents(new String("1.2.2.1"));
    my_baum.setChild(3,new Baum(1).setContents(new String("1.3")));
    my_baum.getChild(3).setChild(1,(new Baum(0).setContents(new String("1.3.1"))));
    System.out.println(my_baum);
} }
```

Anmerkung: Wieder erlauben die Modifikatoren die Verwendung komplexer Terme.

Allgemeiner Baum: Beispiel

Das angegebene Programm erzeugt den folgenden Baum:



- Nummerierung der Knoten
- vgl. Kapitelstruktur eines Buches
- beliebiger Verzweigungsgrad
- Reihenfolge der Kinder
- Navigation/Suche

“Drucken” des Baumes:

[1[1.1[1.1.1]][1.2[1.2.1[1.2.1.1]][1.2.2[1.2.2.1]]][1.3[1.3.1]]]

als Schachtelungs- bzw. Klammerstruktur.

6.4.3 Binärbäume

- Jeder Knoten hat zwei Referenzen: einen linken und einen rechten Unterbaum.

BINÄRBAUM ALS ABSTRAKTER DATENTYP

Konstruktoren:
create: $\rightarrow$ BiBa
biba: BiBa $\times$ T $\times$ BiBa $\rightarrow$ BiBa

Selektoren:
value: BiBa $\rightarrow$ T
left: BiBa $\rightarrow$ BiBa
right: BiBa $\rightarrow$ BiBa
height: BiBa $\rightarrow$ Nat
is_empty: BiBa $\rightarrow$ Bool

Axiome:
value(create) = $\perp$
value(biba(x,b,y)) = b
left(create) = $\perp$
right(create) = $\perp$
left(biba(x,b,y)) = x
right(biba(x,b,y)) = y
is_empty(create) = true
is_empty(biba(x,b,y)) = false
height(create) = 0
height(biba(x,b,y)) = max(height(x),height(y)) + 1

- bisher keinerlei "sinnvolle" Operatoren - nur Struktur.

STRUKTURELLE EIGENSCHAFTEN VON BINÄRBÄUMEN

Über Induktion kann man leicht folgendes beweisen:

- Auf der i-ten Ebene jeweils 2^i Einträge
- ... also bei Höhe k maximal $\sum_{i=1}^{k} 2^k = 2^{k+1} - 1$ Knoten
- man *kann* also n Knoten in einem Baum der Höhe $\log_2 n$ speichern

Anforderungen an Bäume

- je nach Anwendung muß ein Baum
  - zusätzliche Bedingungen an seine Knoten/Struktur erfüllen
  - anwendungsorientierte Operationen unterstützen (einfügen, entfernen, suchen, durchlaufen)
- ein Baum soll so niedrig wie möglich sein
- möglichst wenige Ebenen
- Ebenen möglichst gut gefüllt ("*ausgeglichen*")

$\Rightarrow$ Algorithmen, die dies ermöglichen (basierend auf dem Umhängen von Teilbäumen)

EINSCHUB: BÄUME MIT HOHEM VERZWEIGUNGSGRAD

Es gibt auch Bäume mit höherem *Verzweigungsgrad* $k > 2$ – speziell im Datenbankbereich als *Indexe*:

- Viele kleine Knoten in einem relativ niedrigen Baum
- kurze Suche in einer großen Datenmenge
- wenn der Baum einigermaßen ausgeglichen ist
- teilweise komplexe Restrukturierungsalgorithmen

SPEICHERUNG VON BINÄRBÄUMEN

- Man kann Binärbäume explizit in einer dynamischen Datenstruktur mit Referenzen speichern:
  - die explizite Speicherung erlaubt Umstrukturierungen (umhängen von kompletten Teilbäumen) mit relativ geringem Aufwand.
- oder in einem Feld `Object[]`:
  - `baum[1]` enthält die Wurzel,
  - Für den in `baum[`i`]` gespeicherten Knoten enthält `baum[`$2i$`]` das linke Kind und `baum[`$2i+1$`]` das rechte Kind, und `baum[`$\lfloor i/2 \rfloor$`]` seinen Vaterknoten.
- die implizite Form ist manchmal effizienter, aber
  - viel “leerer Raum” bei nicht ausgeglichenen Bäumen
  - nur anwendbar, wenn man weiß, wie groß der “Baum” werden kann
  - interne Restrukturierungen sind “teuer”

Binärbaum als Java-Klasse

```
public class BiBa {
  protected Object contents = null;
  protected BiBa leftChild = null; protected BiBa rightChild = null;
  public BiBa(Object o) { contents = o;}
  public BiBa(BiBa left, Object o, BiBa right) {
        leftChild = left; contents = o; rightChild = right;}
  public Object getContents() { return contents; }
  public BiBa getLeftChild() { return leftChild; }
  public BiBa getRightChild() { return rightChild; }
  public BiBa setContents(Object o) { contents = o; return this; }
  public BiBa setLeftChild(BiBa b) { leftChild = b; return this; }
  public BiBa setRightChild(BiBa b) { rightChild = b; return this; }
  public int height() { int hl = 0; int hr = 0;
     if (leftChild != null) hl = leftChild.height();
     if (rightChild != null) hr = rightChild.height();
     return (1 + java.lang.Math.max(hl,hr)); }
  public String toString() { // eine von mehreren Moeglichkeiten
    return "[" + contents + leftChild + rightChild + "]"; }
  // analog fuer clone() und equals()
}
```


Binärbaum als Java-Klasse

- Man hätte BiBa natürlich auch als Subklasse von “Baum” ableiten können.

Aufgabe

Erweitern Sie die Klasse *IntegerFeld* um geeignete Methoden, um das Feld als binären Baum aufzufassen (einschließlich einer Methode `getParent()`).

6.4.4 Algorithmen für Baumstrukturen

1. Bäume als reine Speicherungsstruktur für Daten (später mehr)
2. Bäume als Strukturierung des Problems an sich

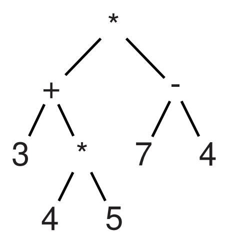
Häufig treten Baumstrukturen bei der Verarbeitung von Sprachen/Grammatiken auf (“Ableitungsbäume”, “Operatorbäume”):

- Programme
- Terme
- logische Formeln
- Nicht immer Binärbäume, aber hier werden exemplarisch solche Grammatiken betrachtet
- in diesen Fällen muss der Baum (rekursiv) durchlaufen werden, um ein gegebenes Problem zu lösen.

Arithmetische Terme als Bäume

- Term “(operand operator operand)” als Baumknoten BiBa(operand,operator,operand),
- Wurzelknoten enthält den Operator
- jeder Operand als Teilbaum,

Betrachten Sie wieder den Term ((3+(4*5))*(7-4)).



- innere Knoten: Operatoren
- Blätter: Zahlen
- “(” lesen: neuen Baum anlegen; nächster Teilterm wird linker Unterbaum
- Zahl lesen: Blatt anlegen
- “)” lesen: Unterbaum abschliessen
- Operator lesen: Operator als Knoteninhalt ablegen

ARITHMETISCHE TERME ALS BÄUME

Auf das als Baum strukturierte Problem kann man verschiedene Algorithmen anwenden:

- Auswerten: linken Teilbaum rekursiv auswerten, rechten Teilbaum rekursiv auswerten, Operator darauf anwenden (*Post-order*)
  - post-order-Schreibweise von Termen auch als UPN (umgekehrte polnische Notation) bezeichnet:

    3 4 5 * + 7 4 - *   keine Klammerung notwendig!
  - anderes Beispiel: Disk-usage bei UNIX-Systemen
- drucken: linken Teilbaum rekursiv drucken, Operator drucken, rechten Teilbaum rekursiv drucken (*In-order*)

  ((3+(4*5))*(7-4))
- funktionale Termschreibweise: erst Operator, dann linken und rechten Teilbaum (*Pre-order*)

  mult(plus(3, mult(4, 5)), minus(7, 4))
  - anderes Beispiel: Anzeigen der Verzeichnisstruktur eines Dateisystems

Spezifikation der arithmetischen Auswertung eines Operatorbaumes

Der entsprechende abstrakte Datentyp erweitert den generischen allgemeinen BiBa:

- Konstruktor:

  biba: ArithBiBa $\times$ Operator $\times$ ArithBiBa $\rightarrow$ ArithBiBa
  biba: ArithBiBa $\times$ Nat $\times$ ArithBiBa $\rightarrow$ ArithBiBa
- Auswertungs-Selektor:

  eval: ArithBiBa $\rightarrow$ Nat
  eval(create) = $\perp$
  eval(biba(create,n,create)) = n
  eval(biba(l,op,r)) = apply(op,eval(l),eval(r))

Aufgabe

Implementieren Sie eine Klasse “ArithBiBa” basierend auf “BiBa”, die arithmetische Terme in einen Baum einliest und auswertet:

- In der Eingabe sind nur Zahlen von 1 bis 9 erlaubt.
- Lesen Sie den Term als Folge von Zeichen mit KeyBoard.readChar() ein, die Sie wahlweise erst in einem Array ablegen oder direkt den Baum daraus erzeugen.
- Benutzen Sie eine kleine Hilfsklasse, die einzelne Zeichen in Zahlen umwandelt.
- Optional: erlauben Sie auch mehrstellige Zahlen in der Eingabe.

Im folgenden werden Bäume als Datenstrukturen über einer totalgeordneten Wertemenge betrachtet.

DURCHWANDERUNG (“TRAVERSIERUNG”) VON BÄUMEN

... wurde eben im Zusammenhang mit Termen behandelt.

kann man als induktiv definierte Selektoren auf Binärbäumen sehen:

- Pre-order: Wurzel - links - rechts

  preorder: BiBa $\rightarrow$ Liste
  preorder(create) = create
  preorder(biba(x,b,y)) = add(b, concat(preorder(x),preorder(y)))

- Post-order: links - rechts - Wurzel

  postorder: BiBa $\rightarrow$ Liste
  postorder(create) = create
  postorder(biba(x,b,y)) = append(concat(postorder(x),postorder(y)),b)

- In-order: links - Wurzel - rechts

  inorder: BiBa $\rightarrow$ Liste
  inorder(create) = create
  inorder(biba(x,b,y)) = concat(inorder(x),add(b,inorder(y)))
  = concat(append(inorder(x),b),inorder(y))

Aufgabe

- Erweitern Sie die Klasse BiBa um Methoden

  ```
  public String preorder()
  public String postorder()
  public String inorder()
  ```

  die alle Knoten des Baumes in der entsprechenden Reihenfolge ausgeben.

- Implementieren Sie Iteratoren über der Klasse BiBa, die den Baum in Post-order/Pre-order/In-order durchlaufen und über

  ```
  class PreorderIterator implements java.util.Iterator { ... }
  public Iterator preorderIterator()
  ```

  etc. verfügbar sind.

- Erweitern Sie die Klasse BiBa um eine Eigenschaft “Summe”, die jedem Baum die Summe der in ihm enthaltenen Elemente zuordnet
  - implementieren Sie die Eigenschaft als rekursiv definierte Funktion
  - implementieren Sie die Eigenschaft als Instanzeigenschaft, deren Initialisierung einen der obigen Traversierungsiteratoren verwendet (welchen?).

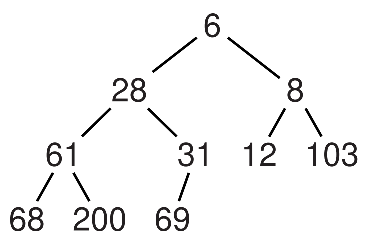
6.4.5 Der “Heap” als spezieller binärer Baum

... erstmal eine relativ einfache (aber untypische) Baumstruktur

Ein *Heap* (“Haufen”) ist ein binärer Baum, der die folgenden Eigenschaften erfüllt:

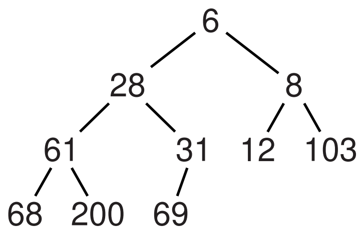
- Der Werte eines Knotens ist kleiner oder gleich groß wie der Wert jedes Wurzelknotens seiner beiden Unterbäume

- Der Baum ist *vollständig*, d.h. alle Blätter befinden sich auf derselben Ebene, und diese Ebene ist von links nach rechts gefüllt.
  (dies macht eine Speicherung als Array effizient)



HEAP: STRUKTUREIGENSCHAFTEN

Speicherung als Array effizient möglich (vgl. Folie 419):



6	28	8	61	31	12	103	68	200	69					

Es gilt folgendes:

- Für jeden Teilbaum ist der Wert der Wurzel der kleinste Wert im Baum,
- für jeden Knoten ist jeder seiner Unterbäume ein Heap,
- ein Heap der Höhe n enthält mindestens 2^{n-1} (und höchstens $2^n - 1$) Knoten
- Ein Heap mit n Knoten hat die Höhe $\lceil \log_2(n+1) \rceil$.

EIGENSCHAFTEN UND OPERATIONEN

- keine sortierte Reihenfolge, aber man kann auf das kleinste Element in $O(1)$ zugreifen ("Prioritätswarteschlange"):

  Signatur:
  top: Heap $\rightarrow$ T

Heap-Eigenschaft muss aufrechterhalten werden:

- Was geschieht, wenn man das oberste Element entfernt?

  Signatur:
  remove: Heap $\rightarrow$ Heap

  - Eine Lücke
  - Man könnte jetzt das kleinere der beiden Kind-Elemente nach oben holen und rekursiv fortfahren
  - $O(\log n)$ bis man an einem Blatt angekommen ist ...
  - ... und dann hat man eine Lücke unten im Baum, wo man sie nicht haben will.
  - ... also anders machen.

OPERATIONEN

Entfernen

- Das oberste (kleinste) Element wird entfernt
- das “letzte” Element wird entfernt und an die oberste Stelle kopiert
- solange eines seiner Kinder kleiner ist als es selbst, wird es (rekursiv) mit den kleineren seiner Kind-Elemente vertauscht (“Durchsickern”) ($O(\log n)$)
- bis die Heap-Eigenschaft wieder erfüllt ist.

Einfügen: analog

- Anhängen eines neuen “letzten” Elements
- “aufsteigen”, durch Vertauschen mit seinem Eltern-Element, solange es kleiner als dieses ist ($O(\log n)$).
- Korrektheits-Überlegungen.

Aufgabe: Fügen Sie die Zahlen 76, 56, 8, 13, 34, 98, 3, 27, 14, 41, 61 in einen Heap ein und entnehmen Sie danach zweimal das obere Element.

Stellen Sie Heap-Baum nach jeder Einfüge- und Entfernungs-Operation grafisch dar.

ANWENDUNG

- Prioritätswarteschlange: Zugriff auf das Element “mit der höchsten Priorität”.
  - Zugriff auf dieses Element: $O(1)$
  - Entnehmen dieses Elements: $O(\log n)$
  - Einfügen eines Elements: $O(\log n)$

Aufgabe

Implementieren Sie eine Klasse “Heap” mit den Operationen

- new(Integer i): Vorgeben der maximalen Größe des Heaps
- public Heap insert(Object o)
- public Object top()
- public Heap remove(Object o)
- Hinweis:
  - definieren Sie geeignete private Operationen zum versickern und vertauschen,
  - merken Sie sich immer den letzten von Heap belegten Index im Feld.

Aufgabe:

Fügen Sie die Zahlen 76, 56, 8, 13, 34, 98, 3, 27, 14, 41, 61 in einen Heap ein und entnehmen Sie danach 4 mal das obere Element und schreiben die Elemente auf. Was fällt auf?

HEAPSORT

- $n \cdot \log n$-Variante von Selection Sort (auch im worst-case):
  Man nimmt immer das kleinste Element vom Heap und stellt dessen Heap-Eigenschaft wieder her.
- elegant als in-place-Algorithmus:
  man vertauscht das kleinste Element mit demjenigen an der letzten Stelle und stellt für die Elemente außer dem letzten die Heap-Eigenschaft wieder her. Und Heap-sortiert diese rekursiv.
- i.a. bekannt, wieviele Elemente sortiert werden; ausgeglichener Baum, "lokale" Operationen (tauschen von Werten)
  $\Rightarrow$ Repräsentation als Feld sehr effizient

Aufgabe

Erweitern Sie die Klasse *IntegerFeld* um eine Methode `HeapSort()`.

HEAP: ZUSAMMENFASSUNG

- Der Heap als Struktur erlaubt nichts (sinnvolles) außer dem Zugriff auf das kleinste Element (dies aber sehr effizient)
- kein Suchen/entfernen bestimmter Elemente
- kann effizient als Feld implementiert werden.
- bei einer Implementierung als verlinkter Binärbaum muss immer einen Zeiger auf das letzte Element sowie auf das Element, wo ein neues Kind eingefügt werden soll, unterhalten werden.

6.4.6 Binäre Suchbäume

Eine häufige, wichtige Menge von Operationen ist (*"Dictionary"*):

- search(x): Suchen, ob ein Element mit dem Wert x vorhanden ist
- insert(x): Einfügen eines Elementes mit dem Wert x
- delete(x): Löschen des Elementes mit dem Wert x
- (o.B.d.A. keine Duplikate erlaubt)

Idee: Binäre Suche

- effizient in gegebenem sortiertem Feld
- man kann einen Baum so aufbauen, dass er binäre Suche (Teilung bei der Wurzel w) effizient unterstützt
  - alle kleineren Elemente $x < w$ im linken Unterbaum
  - alle größeren Elemente $x > w$ im rechten Unterbaum

Operationen

Suchen(x)

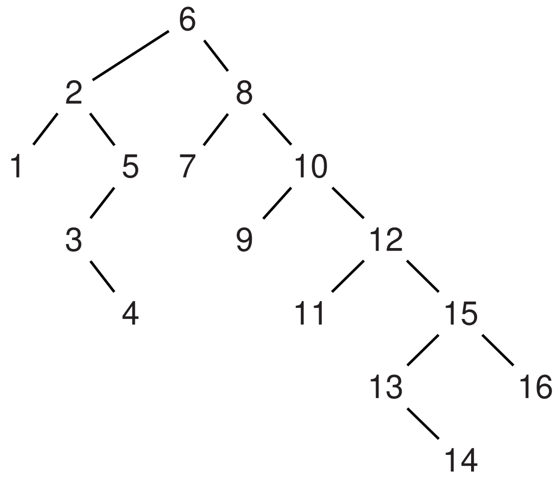
- Rekursiv, bei der Wurzel beginnend:
  - hat das aktuelle Element e den Wert x, gebe eine Referenz auf das Element zurück
  - ist $e > x$, suche im linken Unterbaum weiter
  - ist $e < x$, suche im rechten Unterbaum weiter
  - existiert der linke/rechte Unterbaum nicht, gebe "nicht gefunden" aus

Einfügen(x)

- wie suchen,
- falls das Element nicht gefunden wird, füge x an der Stelle, wo die Suche endet, ein (passend, als linkes oder rechtes Kind)

Beispiel

Fügen Sie die Zahlen 6,2,8,5,10,9,12,1,15,7,3,13,4,11,16,14 nacheinander in einen leeren binären Suchbaum ein.



- Suchen Sie, ob die Werte 7.5, 13, 0.5, 14.5, 15.5 in dem Baum enthalten sind.
- Wie können Sie die o.g. Zahlenfolge sortiert ausgeben?
- Fügen Sie die sortierte Zahlenfolge in einen BSB ein.

OPERATIONEN

Entfernen(x)

- Suche x. Falls nicht gefunden, fertig.
- falls gefunden:
  - man kann x nicht einfach löschen (Lücke)
  - wenn x ein Blatt ist: löschen möglich
  - wenn x nur einen Unterbaum hat: diesen Baum anstatt x an diese Position hängen
  - wenn x zwei Unterbäume hat??
    entferne das größte Element des linken Unterbaumes (dieses hat kein Kind oder nur ein linkes Kind, kann also leicht herausgenommen werden) und setze es in die Lücke (oder das kleinste Element des rechten Unterbaumes).

⇒ es werden ganze Teilbäume verschoben

⇒ explizite Repräsentation mit Referenzen sinnvoll

Beispiel

- Löschen sie in dem obigen Baum die 10, die 15, und dann die 6.

ABSTRAKTER DATENTYP "BSB" ("BINÄRER SUCHBAUM")

- erweitert BiBa.

Modifikatoren:
insert: BSB $\times$ T $\rightarrow$ BSB
delete: BSB $\times$ T $\rightarrow$ BSB

Selektoren:
search: BSB $\times$ T $\rightarrow$ Bool
max: BSB $\rightarrow$ T
min: BSB $\rightarrow$ T

Axiome:

$$\text{search(create,e)} = \text{false}$$

$$\text{search(bsb(x,b,y),e)} = \begin{cases} \text{if e=b then true} \\ \text{if e<b then search(x,e)} \\ \text{if e>b then search(y,e)} \end{cases}$$

$$\text{insert(create,e)} = \text{bsb(create,e,create)}$$

$$\text{insert(bsb(x,b,y),e)} = \begin{cases} \text{if e=b then bsb(x,b,y)} \\ \text{if e<b then bsb(insert(x,e),b,y)} \\ \text{if e>b then bsb(x,b,insert(y,e))} \end{cases}$$

ABSTRAKTER DATENTYP "BSB" (FORTS.)

Axiome (Forts.):

$$\text{max(create)} = -\infty$$
$$\text{max(bsb(x,b,y))} = \text{maximum(b,max(y))}$$
$$\text{min(create)} = \infty$$
$$\text{min(bsb(x,b,y))} = \text{minimum(min(x),b)}$$

$$\text{delete(create,e)} = \text{create}$$

$$\text{delete(bsb(x,b,y),e), b} \neq \text{e} = \begin{cases} \text{if e<b then bsb(delete(x,e),b,y)} \\ \text{if e>b then bsb(x,b,delete(y,e))} \end{cases}$$

$$\text{delete(bsb(create,e,create),e)} = \text{create}$$
$$\text{delete(bsb(create,e,r),e) , r} \neq \text{create} = \text{r}$$
$$\text{delete(bsb(l,e,r),e), l} \neq \text{create} = \text{bsb(delete(l,max(l)),max(l),r)}$$

Implementierung

- Ziemlich straightforward
- Suche entweder rekursiv oder iterativ
- Java-Code siehe nächste Folie oder [Saake, Kap. 14.3]

Binärer Suchbaum: Implementierung

```
public class BSB extends BiBa {
  public BSB(int i) { super(new Integer(i)); }
  public BSB(Comparable o) { super(o); }
  public void insert(int i) { insert(new Integer(i));}
  public void insert(Comparable o) {
      if (((Comparable)getContents()).compareTo(o) == 1 )
         { if (leftChild == null) {setLeftChild(new BSB(o)); }
           else ((BSB)leftChild).insert(o);}
      else { if (rightChild == null) {setRightChild(new BSB(o)); }
             else ((BSB)rightChild).insert(o); }}
  public boolean search(int i) { return search(new Integer(i));}
  public boolean search(Comparable o) {
      if (((Comparable)getContents()).compareTo(o) == 0 ) { return true; }
      else if (((Comparable)getContents()).compareTo(o) == 1 )
         { if (leftChild == null) {return false; }
           else return ((BSB)leftChild).search(o); }
      else { if (rightChild == null) {return false; }
             else return ((BSB)rightChild).search(o); }}
  // sowie delete(...) und weitere Methoden
}
```

Testprogramm

```
public class BSBTest {
  public static void main (String[] args){
  BSB myBaum = new BSB(18);
  myBaum.insert(23);
  myBaum.insert(4);
  myBaum.insert(8);
  myBaum.insert(20);
  myBaum.insert(28);
  myBaum.insert(1);
  myBaum.insert(19);
  myBaum.insert(12);
  System.out.println(myBaum.height());
  System.out.println(myBaum);
  System.out.println(myBaum.search(8));
  System.out.println(myBaum.search(24)); }}
```

Aufgabe

- Ergänzen Sie BSB um eine delete()-Methode
  Hinweis: dazu müssen Sie die Struktur um einen Zeiger zum Elternelement ergänzen.

AUFWAND

- alle Operationen basieren auf “Suchen eines Elementes”
- Durchlaufen eines Pfades im Baumes bis zu einem Blatt
  - Die Operation wird immer in höchstens einem der beiden Teilbäume fortgesetzt
- wie tief ist der Baum bei n Knoten?
  - ausgeglichener Baum: $\log n$
  - nicht ausgeglichener Baum: bis zu n
    (Elemente in sortierter Reihenfolge eingefügt)

SORTIEREN MIT EINEM BINÄREN SUCHBAUM

- Ein In-order Durchlauf des Baumes liefert die in ihm enthaltenen Zahlen in aufsteigend sortierter Reihenfolge.

  Aufgabe: Beweisen Sie durch Induktion, dass das korrekt ist.
- Das Einfügen einer Folge von Zahlen (vgl. Folie 438) ist damit ein InsertionSort-Verfahren (vgl. Folie 241)
- grob geschätzt $O(n \cdot \log n)$ – solange der Baum einigermaßen ausgeglichen ist

AUSGEGLICHENE BÄUME

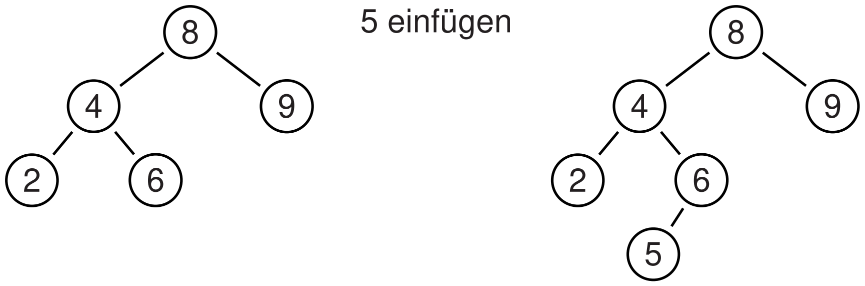
- Beim Einfügen und Löschen jeweils Baum ausgleichen
  - total ausgeglichener Baum (alle Blätter auf maximal 2 Höhen): aufwendig zu unterhalten
    Bsp: Baum, in den [5,3,7,6,4,2] eingefügt sind. Füge dann 1 ein. Alle Knoten/Verbindungen müssen geändert werden.

Also: weniger strenge Kriterien ansetzen

- AVL-Baum: Binärbaum mit einem abgeschwächten Ausgeglichenheitskriterium
- B-Baum (und B^*-Baum): ausgeglichene Höhe, aber unausgeglichener Verzweigungsgrad (z.B. in Datenbanksystemen als Indexstrukturen verwendet)

6.4.7 AVL-Bäume

- AVL = G.M. **A**delson-**V**eltskii und E.M. **L**andis (1962)
- Für jeden Teilbaum unterscheidet sich die Höhe des rechten Teilbaums von der des linken Teilbaums (betragsmäßig) höchstens um 1
- "*Balance*" eines Knotens:
  bal: BSB $\rightarrow$ Nat bal(bsb(x,e,y)) = height(x) - height(y)
- Implementierung: für jeden Knoten Höhe des Teilbaumes speichern
- Suchen: wie vorher
- Einfügen, Löschen: ggf. Baum restrukturieren



(a) AVL (b) nicht AVL

Einfügen

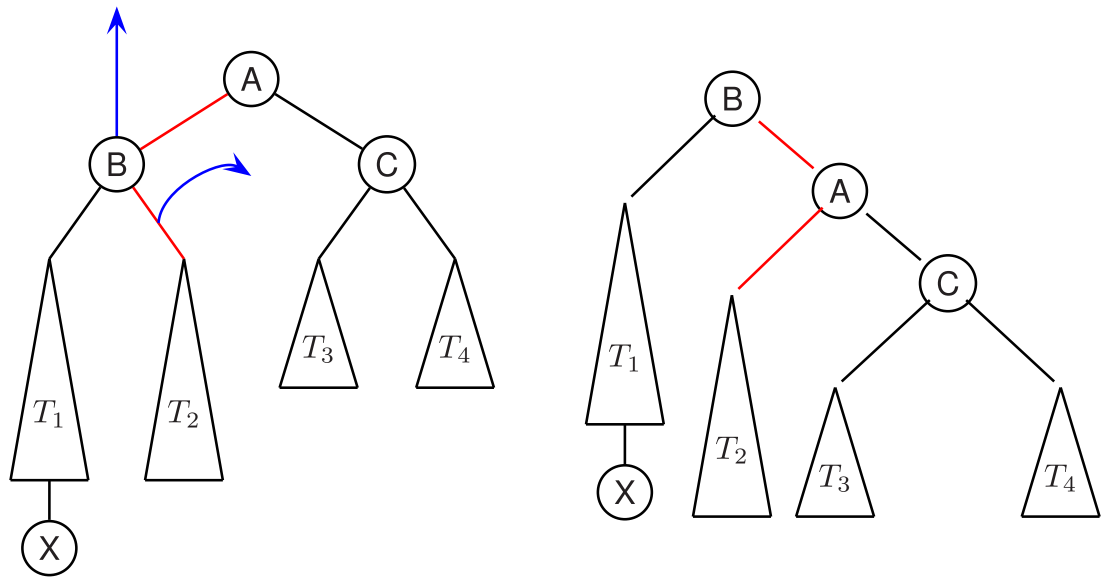
- Der neue Knoten wird wie üblich eingefügt.
- Möglicherweise erfüllt der resultierende Baum die AVL-Eigenschaft nicht
- es gibt einen –auf dem Pfad von dem neuen Knoten zur Wurzel– untersten Knoten A, der die Eigenschaft nicht erfüllt (balance betragsmäßig $= 2$).
- dessen Unterbäume werden geeignet restrukturiert (*"Rotation"*)

Fallunterscheidung

- O.B.d.A.: A's linker Unterbaum ist um 1 höher als A's rechter Unterbaum, und es wird in den linken Unterbaum eingefügt, und dieser wird dadurch nochmal um 1 höher
  - linker Unterbaum des linken Kindknotens von A
  - rechter Unterbaum des linken Kindknotens von A

AUSGLEICHEN VON AVL-BÄUMEN

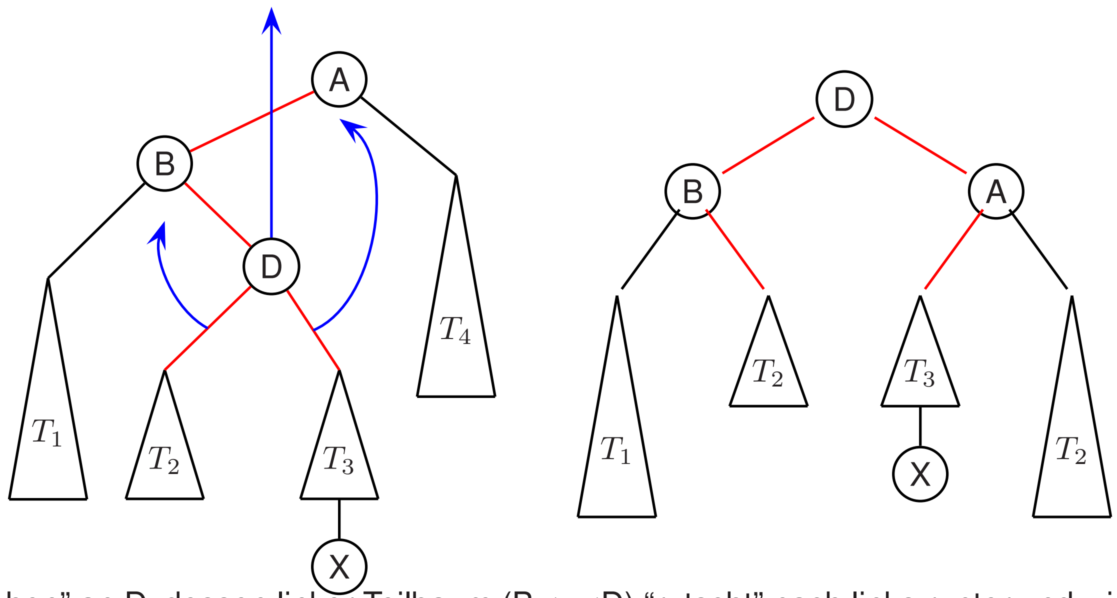
“Einfache” Rotation



- “ziehen” an B, dessen rechter Teilbaum (B<x<A) “rutscht” nach rechts runter und wird linker Teilbaum von A.

AUSGLEICHEN VON AVL-BÄUMEN

“Doppelte” Rotation



- “ziehen” an D, dessen linker Teilbaum (B<x<D) “rutscht” nach links runter und wird rechter Teilbaum von B. D’s rechter Teilbaum (D<x<A) “rutscht” nach rechts runter und wird linker Teilbaum von A.

EINFÜGEN IN AVL-BÄUMEN

Man kann durch Betrachtung des Pfades von dem eingefügten Element zur Wurzel alle notwendigen Neuberechnungen durchführen:

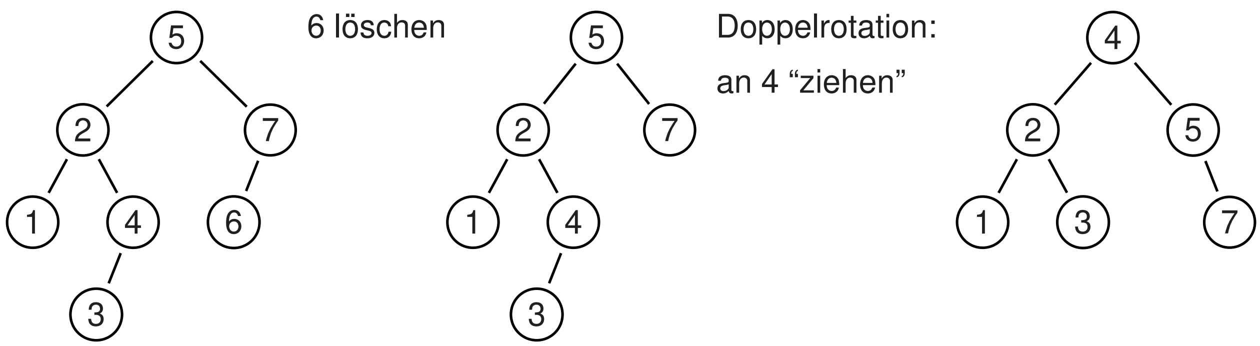
- Wenn der betrachtete Knoten die Balance null hatte, muss seine Höhe und seine Balance neu berechnet werden. Sowohl Höhe als auch Balance ändern sich, und die Betrachtung muß nach oben fortgesetzt werden.
- Wenn der betrachtete Knoten die Balance ± 1 hatte (*“kritischer Knoten”*):
  - Einfügen auf der “richtigen” Seite: Balance wird null, Höhe unverändert.
  - Einfügen auf der “falschen” Seite: Balance wird ± 2. Nach einer geeigneten Rotation (mit Neuberechnung der Höhe/Balance aller betroffenen Knoten) ist die Höhe des Teilbaumes unverändert.

  $\Rightarrow$ In beiden Fällen muß die Betrachtung nicht mehr nach oben fortgesetzt werden,

  $\Rightarrow$ Der gesamte Baum erfüllt die AVL-Eigenschaft,

  $\Rightarrow$ eine solche Rotation genügt.
- Aus obigem folgt: alle Knoten auf dem betrachteten Pfad unterhalb des kritischen Knotens haben Balance 0 (wie in den Bildern gezeichnet).

LÖSCHEN IN AVL-BÄUMEN



(a) AVL (b) nicht AVL (c) AVL um 1 niedriger

- Der Baum kann nach dem Löschen und Ausgleichen um 1 niedriger als vorher sein

$\Rightarrow$ Kontrolle muss nach oben fortgesetzt werden (bis zu $O(\log n)$ Rotationen notwendig)

Aufgabe

Geben Sie einen Baum an, bei dem nach dem Löschen eines Elementes zwei Rotationen notwendig sind.

AVL-BÄUME: IMPLEMENTIERUNG

Aufgabe:
Implementieren Sie eine Klasse AVLBaum auf Basis von BSB.

- Es ist empfehlenswert, `height()` jetzt nicht mehr als Funktion zu definieren, sondern fest als Eigenschaft der Knoten zu verwalten (und bei Operationen zu aktualisieren).

6.4.8 B- (und B*-) Bäume

- "B" steht *nicht* für "Binär", sondern für "balanciert", "breit" (Verzweigungsgrad $>>$ 2), "buschig", oder für R. Bayer (der mit E. McCreight diese Art Bäume entwickelt hat).
- Baumhöhe ist völlig ausgeglichen: alle Blätter auf derselben Höhe
- dafür variiert der Verzweigungsgrad:
  - innere Knoten haben die Form $(p_0, k_1, p_1, k_2, p_2, \ldots, k_n, p_n)$ wobei $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ und
    - k_i sind geordnet: $k_i < k_j$ für $i < j$
    - p_i zeigt auf den $i + 1$ten Unterbaum, in dem für alle Werte x $k_i \leq x < k_{i+1}$ gilt
  - m ist die "Ordnung des Baumes"
- Blätter enthalten die Dateneinträge zu den Schlüsselwerten
- Für n Knoten gilt
  - $h \leq \lceil log_{m/2} N \rceil$ (Knoten halb gefüllt)
  - $h \geq \lceil log_m N \rceil$ (Knoten komplett gefüllt)
- Suche benötigt maximal h Schritte; innerhalb der Knoten Binärsuche $O(\log m)$

B-BÄUME: OPERATIONEN

- Einfügen:
  - solange noch Platz im Blattknoten ist: abspeichern
  - sonst: Knoten teilen, und zusätzlichen Eintrag im Vaterknoten anlegen
  - ggf. läuft dieser wieder über ...
  - ggf. wird also der Wurzelknoten geteilt und *darüber* eine neue Ebene angelegt.
- Löschen:
  - Element im Blattknoten löschen:
    - solange Knoten danach noch mehr als $m/2$ Elemente enthält: einfach löschen
    - sonst: Ausgleichen: Elemente aus einem Nachbarknoten umsetzen; ggf. Knoten vereinigen

  $\Rightarrow$ Restrukturierungsaufwand
  - Element im inneren Knoten löschen:
    - erstes Element aus dem darunterliegenden Knoten nach oben verlagern, unten löschen

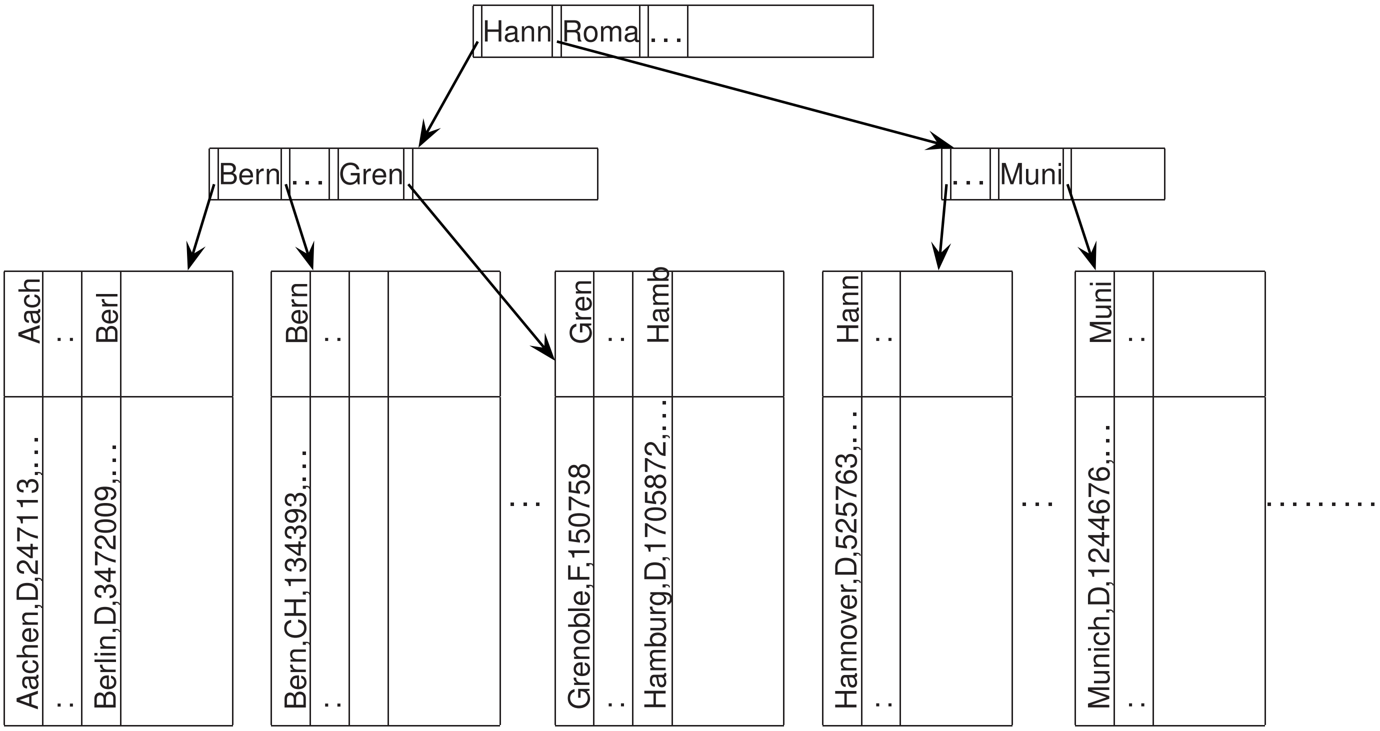
B*-BÄUME

- Häufig in Datenbanken als Indexbaum zum Suchen (z.B. alphabetisch nach Namen geordnete Dateneinheiten) eingesetzt
- es geht also i.a. nicht darum, nur Zahlen (wie eben im AVL-Baum) zu speichern, sondern einen Index auf größere Datensätze zu haben

Trennung von *Suchinformation* und *Daten*:

- innere Knoten enthalten nur noch die Suchinformation (B-Baum der Ordnung m über den *Suchschlüsselwerten*)
- Blätter enthalten die Datensätze und haben die Form $([k_1, s_1], [k_2, s_2], \ldots, [k_g, s_g])$ wobei k_i ein Suchschlüsselwert ist, und s_i der dazugehörige Datensatz.
- Jedes Blatt enthält maximal l Datensätze,
- falls mehr als 1 Blatt existiert, enthält jedes Blatt mindestens $l/2$ Datensätze.

Beispiel: B*-Baum



B*-BÄUME: EIGENSCHAFTEN

- Für n Datensätze gilt:
  - $h \leq \lceil log_{m/2}(2N/l) \rceil$
    ($l/2$ Einträge pro Blatt, innere Knoten halb gefüllt)
  - $h \geq \lceil log_m(N/l) \rceil$
    (l Einträge pro Blatt, innere Knoten komplett gefüllt)
- Suche benötigt h Schritte, innerhalb der Knoten Binärsuche $O(logm)$
- Wenn die Blätter verlinkt sind, kann man dann die Datensätze auch sequentiell aufzählen
- Einfügungen und Modifikationen wie im B-Baum
- sehr geeignet für den abstrakten Datentyp "(geordnete) Dictionary" – und exakt das ist oft ein Index einer Datenbank.

Beispiel

Die eben beschriebene Datenbank enthält 3000 Städte, wobei ein Index über den Namen als B*-Baum organisiert ist:

- $l = 10$: jedes Blatt enthält maximal 10 Datensätze (minimal 5)
- $m = 30$: jeder innere Knoten enthält maximal 30 Verweise (minimal 15)

Dann:

- jeder innere Knoten auf der untersten Ebene erfaßt 75-300 Städte
- man benötigt also 10-40 solche Knoten
- Bei guter Füllung sind es weniger als 30, und man benötigt nur noch einen Wurzelknoten (siehe Grafik)
- Bei schlechter Füllung sind es mehr als 30, und es werden zwei innere Knoten der 2.Ebene benötigt, und darüber noch ein Wurzelknoten.

6.4.9 Bäume: Zusammenfassung

- Datenstruktur
- Speichern *geordneter* Daten
- Suchen
  - Gleichheitssuche nach einem bestimmten Wert/Schlüssel in $O(\log n)$
  - Bereichssuche:
    - Suche + Inorder-Durchgang in BSBs/AVL-Bäumen
    - Suche und Durchwanderung der Blattknoten in B*-Bäumen
- Einfügen und Löschen:
  ggf. Reorganisation (AVL, B, B*)
  aber immer noch in $O(\log n)$
- nicht zu vergessen: Insertion-Sort in $O(n \log n)$:
  n Elemente in den Baum einordnen: $O(n \log n)$
  mit inorder-Durchlauf aufzählen: $O(n)$.

6.4.10 Java und Bäume

Java bietet Bäume “alleine” nicht an, aber diverse abstrakte Datentypen, die als Bäume implementiert sind:

- Interface `Set` mit Klasse `TreeSet` (JDK 1.2), die geordnete Mengen als Baum verwaltet:
  - verwendet Rot-Schwarz-Bäume
  - wahlweise “natürliche Ordnung” wie mit `compareTo` gegeben,
  - aber kann auch mit speziellem `Comparator`-Object erstellt, werden, das eine beliebige Vergleichsmethode implementiert:

    ```
    compare(Object o1, Object o2)
    ```
- Interface `Map` mit Klasse `TreeMap` (JDK 1.2),
- Einfügen, Enthaltensein, Löschen in $O(\log n)$
- geordnetes Ausgeben als Inorder-Iterator.

6.5 Hashing

Zugreifen in O(1): Hashing

Grundidee: Abbildung von Elementen durch eine “Hashfunktion”

$$h : \mathsf{T} \to \{1 \ldots n\}$$

in (viele) “Körbe”, in denen man dann sehr schnell etwas findet (evtl. nochmal mit anderer Funktion “hashen”).

- Suchen: $h(e)$ berechnen und in diesem Korb schauen; $O(1)$
- Einfügen: $h(e)$ berechnen, Element in diesen Korb hinzufügen; $O(1)$
- Bei Zahlen: modulo einer Primzahl n nehmen $\to$ n Körbe
- Bei Strings: z.B. ASCII-Summe modulo n.
- Eventuell problematisch: interne Organisation der Körbe (können überlaufen)
- völlig verschiedene Elemente landen zufällig im gleichen Korb
  - Vorteil: zufällige Verteilung
- Nachteil: keine lineare Ordnung, also keine Bereichssuche

Java und Hashing

- Interface `Set` mit Klasse `HashSet` (JDK 1.2), die Mengen durch Hashing verwaltet:
  - schneller Test auf Enthaltensein eines Elementes ($O(1)$)
  - schelles Einfuegen und Loeschen ($O(1)$)
  - einfaches Aufzählen ($O(n)$)
  - aber keine Ordnung der Elemente in der Speicherung
- Interface `Map` (JDK 1.2), das Dictionaries unterstützt und u.a. als `HashMap` realisiert ist.
- Interface Hashtable (JDK 1.0), veraltet

Aufgabe

Schauen Sie sich das Interface `Set` und die Implementierung von *HashSet* an, und implementieren Sie eine Lottozahlen-Klasse.

Anhang A
Ausblick

- Allgemeine Entwurfsmuster für Algorithmen: Saake, Kap.8:
  Greedy, Backtracking, D&C, Dynamisches Programmieren
- Induktion und nochmals Induktion: Manber, Kap. 5
- Graphen-Algorithmen: Manber, Kap. 7
- Geometrische Algorithmen: Manber, Kap. 8

Empfehlenswerte sonstige Literatur:

- Udi Manber: Introduction to Algorithms - a Creative Approach (Info-I-III-geeignet)
- Thomas Ottmann und Peter Widmayer: Algorithmen und Datenstrukturen (auf Deutsch).
- Cormen, Leiserson, Rivest: Algorithms. Eine “Bibel”. In jeder Hinsicht “erschöpfend”.

Java: Sonstiges zur Programmierung

Wenn Sie richtig mit Java programmieren wollen, sollten Sie sich in einem Java-Buch u.a. die folgenden Dinge anschauen:

- Package-Konzept, `import`-Anweisung
  (Java-spezifisch, aber allgemeine Strategie ("Libraries"))
- Setzen von `CLASSPATH` zur Verwendung von Packages (Java-spezifisch)
- Klasse `StringBuffer` zum Arbeiten mit Strings (Behandlung von Strings ist in jeder Programmiersprache unterschiedlich)
- Ausnahmen/Exceptions: `try - catch`
  (gibt es in den meisten imperativen Programmiersprachen)
- Klicki-Bunti-Benutzerschnittstellen: AWT – Abstract Windowing Toolkit und Swing
- Java im Internet: Applets

... Learning by Doing. Und: Jede Programmiersprache ist anders. C++ ist ähnlich.
Wichtig sind die *Konzepte*!

Disclaimer ... Das war's

Diese Folien wurden komplett mit LaTeX & friends erstellt, wobei die folgenden Packages verwendet wurden:

seminar, amssymb, url, moreverb, pstricks, dbicons, version, ntheorem

Empfehlenswerte Literatur:

- H. Kopka: Eine Einführung in LaTeX; Addison-Wesley.