Informatik Vorlesung Wintersemester 2004

Johannes Waldmann, HTWK Leipzig

26. Januar 2006

Informatik: Einführung

Informatik—Überblick

- Informatik: Wissenschaft von der Verarbeitung symbolischer Informationen
- durch Algorithmen
- implementiert als Programm
- ausgeführt auf (einem oder mehreren) Computern

Teilgebiete der Informatik

(HKF S. 23 ff).

- Theoretische: Automatentheorie, Algorithmentheorie, Berechenbarkeit, Komplexität
- Praktische: Programm- und Software-Entwicklung, Dienst- und Arbeitsprogramme (Compiler, Betriebssysteme, Datenbanken, . . .)
- Technische: Hardware (Prozessor, Speicher, Peripherie, Netze, Kommunikation)
- Angewandte:

(HKF = Horn, Kerner, Forbrig: *Lehr- und Übungsbuch Informatik, Band 1*)

Inhalt (1. Semester)

- Grundlagen der Informatik: Algorithmen
  - Definition, Beispiele
  - Entscheidbarkeit, Komplexität
- Praktische Informatik: Programme
  - einfache Daten und Kontrollstrukturen
  - Unterprogramme, Klassen, Methoden
  - Java-Applets und -Applikationen
  - konkrete Datentypen: Arrays, Listen
  - abstrakte Datentypen, Spezifikationen, Interfaces
  - effiziente Datenstrukturen: balancierte Bäume, Hashing

Inhalt (2. Semester)

- Technische Informatik: Hardware
  - maschinelle Darstellung von Information
  - Rechner-Aufbau: Prozessor, Speicher
- Praktische/Angewandte Informatik:
  - Betriebssysteme: Resourcen-Verwaltung und -Teilung
  - Netze, Kommunikation, Standards, Protokolle
  - Kompression, Kryptographie, Sicherheit

Empfohlene Literatur/Links

- Webseite zur Vorlesung/Übung, mit Skript, Folien, Aufgaben: `http://www.imn.htwk-leipzig.de/~waldmann/ws04/informatik/`
- Lehrbuch (Allgemeines, Grundlagen): Horn, Kerner, Forbrig: *Lehr- und Übungsbuch Informatik*, Fachbuchverlag (3. Auflage, 1. Band: Grundlagen und Überblick, auch für 2. Semester geeignet) `http://www.inf.tu-dresden.de/~fachbuch/`
- Software: *BlueJ* `http://www.bluej.org/`
- Lehrbuch (Programmieren): David J. Barnes / Michael Kölling: *Objektorientierte Programmierung mit Java (Eine praxisnahe Einführung mit BlueJ)*

`http://www.pearson-studium.de/main/main.asp?page=shop/bookdetails&ProductID=37199`

Organisation

- Vorlesung
  - dienstags, 7:30–9:00, G126
- Seminare: Einschreibung über ein Web-Interface `http://autotool.imn.htwk-leipzig.de/cgi-bin/Super.cgi`
  - mittwochs, 11:15–12:45, GuS ??
  - *oder* mittwochs, 13:45–15:15, GuS ??

  bitte gleichmäßig verteilen . . . wer in kleinerer Gruppe ist, lernt mehr!

  ab Woche 44 Seminare im Pool GuRL-MM

Leistungsnachweise (1. Semester)

- zur Prüfungszulassung:
  - kleinere Denk- und Programmier-Aufgaben, Kontrolle mündlich im Seminar
  - und automatisch (Web-Interface)
- Prüfung: Klausur

2. Semester ähnlich, dann Gesamtnote.

Was ist Informatik

(vgl. Kapitel 1 aus Horn/Kerner/Forbig)

- Wissenschaft von der Informationsverarbeitung
- junge Wissenschaft mit alten Inhalten (Tabelle S. 21)
- drei Typen von Maschinen (Tabelle S. 22)

Kategorie	Wissenschaft	Schlüssel-Erfindung	früher Vertreter
Energie			
	Chemie		
			Rauch-zeichen

Definition Algorithmus

ist eine in Schritte geordnete Arbeitsvorschrift

- endlich in der Notation
- endliche in der Abarbeitung
- zuständig für eine ganze Aufgabenklasse
- deterministisch (vorherbestimmt)

Computer als Werkzeug

(HKF S. 27 ff)
vier Schritte

- Problemstellung und Lösung
- externe/interne Welt

Auswirkungen für Benutzer und Hersteller von Software

Einfache (?) Algorithmen

1.) Beschreibe die Wirkung dieses Algorithmus:

```
a := a + b ; b := a - b ; a := a - b;
```

2.) Bis heute ist *unbekannt*, ob dieser Algorithmus für jede Eingabe ≥ 1 hält:

```
lies positive ganze Zahl x;
wiederhole, solange (x > 1):
    wenn x eine gerade Zahl ist,
    dann x := x / 2  sonst x := 3 * x + 1
```

Welches sind jeweils die kleinsten Startwerte, für die

- eine Zahl > 1000 erreicht wird,
- > 100 Schritte ausgeführt werden?

Rechnermodelle

Geschichte – (vgl. HKF 1.4)

- Entwicklung der Mathematik (irdische und astronomische Geometrie, Messungen von Weg und Zeit)
- Rechenvorschriften (schriftl. Multiplikation), Tafeln von Logarithmen, Winkelfunktionen
- maschinelle Hilfen bei deren Berechnung (Babbage), programmgesteuerte Webstühle (Jacquard), Lochkartenzählmaschinen
- frei programmierbare Maschinen (Zuse ab 1936)
- prinzipielle Grenzen von Algorithmen (Gödel 1931, Turing 1936)

Computer-Generationen

aus HKF 1.4

1. elektro-mechanisch (1940–1948)
2. elektronisch (Röhren, Transistoren) (1945–1955)
3. integriert (1960–1970)
4. hochintegriert (1970–1985)
5. vernetzt (1985–. . .)

beachte: Einteilung nach verschiedensten Gesichtspunkten möglich, deswegen Zurordnung zu Generationen stark willkürlich (Namen sind Schall und Rauch)

“Kleine Geschichte der EDV”

Paul E. Ceruzzi: *A History of Modern Computing*, MIT Press, 2003 (deutsch mitp 2003)

- die ersten kommerziellen Computer (1945–1956)
- die zweite Computer-Generation (1956–1964)
- die frühe Geschichte der Software (1952–1968)
- vom Großrechner zum Minicomputer (1959–1969)
- Börsenhöhenflüge und das System /360 (1961–1975)
- der Siegeszug des Chips (1965–1975)
- die PCs (Personal Computer) (1972–1977)
- die Erweiterung des menschlichen Intellekts (1975–1985)
- Workstations, Unix und das Netz (1981–1995)

Computer-Architektur nach von Neumann

frei adressierbarer Hauptspeicher, zur Erleichterung verwende nicht Adressen, sondern Namen.
Programm-zustand ist Speicherinhalt (= Abbildung von Adressen auf Werte)
Und: Programm steht selbst im Speicher.

- Zuweisungen: Name := Ausdruck
- wobei Ausdruck: Konstante oder Name oder Operator mit Argumenten
- wobei Argument: Konstante oder Name
- Sprünge (unbedingte und bedingte)

Strukturiertes Programmieren

Programm-Ablauf-Steuerung nicht durch Sprünge, sondern durch hierarchische Struktur.

- Blöcke (Anweisungsfolgen)
- Verzweigungen (if then else)
- Wiederholungen (while)
- Unterprogramme (benannte Blöcke)

Jeder dieser Bausteine hat genau einen Eingang und genau einen Ausgang.

Algorithmen

Sortier-Algorithmen: Spezifikation

Eingabe: eine Folge $(x_1, x_2, \dots, x_n)$ von Zahlen.
Ausgabe: eine Folge $(y_1, y_2, \dots, y_n)$ von Zahlen.
Bedingungen:

- die Ausgabe ist eine Permutation (= Umordnung) der Eingabe.
- die Ausgabe ist aufsteigend geordnet.

geschätzt 3/4 aller Rechenzeit der Welt wird für Sortieren verbraucht!

Sortier-Algorithmen: lineares Einfügen

```
sortiere a[0 .. n-1] = {
  für i von 1 bis n-1 führe aus {
    füge a[i] in a[0 .. i-1] ein
  }
}
füge x in a[0 .. i-1] ein = {
  für k von i-1 bis 0 führe aus {
    if a[k] < x
       then a[k+1] = x; verlasse Schleife
       else a[k+1] = a[k]
  }
}
```

Laufzeit: Einfügen linear, Sortieren quadratisch.

Sortier-Algorithmen: binäres Einfügen

Idee: vergleiche mit mittlerem Element

```
füge x in a[l .. r] ein = {
  if (l = r)
  then if x < a[l]
       then x vor a[l] else x nach a[l]
  else m := mitte von l und r
       if x < a[m]
       then füge x in a[l .. m - 1] ein
       else füge x in a[m + 1 .. r] ein
}
```

Vorsicht: werden alle Spezialfälle richtig behandelt? Diskussion im Seminar.— *Beachte:* hier tun wir so, als ob das Beiseiteschieben der Elemente nichts kostet.

Sortieren: Stand der Dinge

Algorithmen unter verschiedenen Aspekten

- möglichst wenige Element-Vergleiche (sortiere 5 mit 7 Vergleichen?)
- möglichst einfache Datenstrukturen
- möglichst fester Algorithmus (Sortiernetze)
- geänderte Aufgaben (nur zweitbester, drittbester, Median)

die diesbezügliche Bibel: Donald E. Knuth: *Art of Computer Programming* Vol. 3: Sorting and Searching `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`

Seminare

Wiederholung zur Vorlesung (Defn. Informatik, Algorithmus)
Einfache Sortiernetze: 4 Eingänge mit 6, mit 5 Komparatoren.
Beweis, daß 4 Komparatoren nicht ausreichen. Dazu: Anzahl der Permutationen von n Elementen ausrechnen. Schubfachschluß wiederholen.
Später: Sortiernetze für 5 (6,7,8,9) Eingänge als autotool-Aufgabe.
Bubblesort als Netz, induktive Definition. → als Programm. Geschachtelte Schleifen. Ausführung simulieren.

Berechenbarkeit, Komplexität (28. 10. 04)

Literatur: Horn/Kerner Kap. 8.3 (Komplexität), Kap. 8.2 (Berechenbarkeit)

Komplexität von Algorithmen

Wie gut ist ein Algorithmus?

- Ressourcen-Verbrauch (Rechenzeit/Speicherplatz/…)?
- Für *eine* spezielle Eingabe? — Besser: Mengen von ähnlichen (= gleich großen) Eingaben betrachten.
- Ressourcen für besten, mittleren, schlechtesten Fall.
- Betrachten der Verbrauchsfunktion:

  bildet Eingabegröße ab auf (minimale, durchschnittliche, maximale) Kosten aller Rechnungen für alle Eingaben dieser Größe

- Maschinen-unabhängige Aussagen durch Betrachtung des (asymptotischen) *Wachstums*, d. h. ignoriere:
  - Unregelmäßigkeiten für kleine Eingaben
  - konstante Faktoren (für alle Eingaben)

Beispiele: alle linearen Funktionen wachsen ähnlich, alle quadratischen Funktionen wachsen ähnlich, etc.

Komplexität – Beispiel

Sortieren durch *lineares* Einfügen (bzw. entsprechendes Netzwerk) benötigt für n Elemente

$$0 + 1 + 2 + \ldots + (n-1) = (n-1)n/2$$

Vergleiche.
Egal, auf welchem Rechner wir das ausführen, die Laufzeit wird *immer* eine quadratische Funktion der Eingabegröße sein.
D. h. Eingabegröße verdoppeln $\rightarrow$ vierfache Laufzeit, verdreifachen $\rightarrow$ neunfache, usw.
Schnelleren Prozessor zu kaufen lohnt sich kaum, man gewinnt damit nur einen konstanten Faktor. Viel wirksamer sind bessere Algorithmen!

Komplexität von Problemen

Wie schwer ist ein (algorithmisches) Problem?
Wieviel Ressourcen braucht *jeder* Algorithmus, der das Problem löst?
Satz: Für *jedes* Sortierververfahren für n Elemente gibt es eine Eingabe, für die das Verfahren $\geq \log_2(n!)$ Vergleiche ausführt.
Beweis: es gibt $n!$ Permutationen, unter denen genau eine zu finden ist. Durch Elementvergleiche können wir in jedem Schritt die Anzahl der Möglichkeiten bestenfalls halbieren.
Damit brauchen wir $\geq \log_2(n!)$ Vergleiche.

Aufgabe: Werteverlauf dieser Funktion ausrechnen und mit bekannten Sortierverfahren/-Netzen vergleichen.

Abschätzung des Wachstums: $\log_2(n!) \approx n \log n$
Folgerung: Sortieren hat eine Komplexität von wenigstens $n \log n$. (D. h. mehr als linear, aber weniger als quadratisch.)
Folgerung: Sortieren durch binäres Einfügen ist optimal (durch lineares Einfügen nicht).

Suchprobleme

Viele Aufgaben lassen sich als Such-Probleme formulieren.
Beispiel 1 (COL): gegeben sind ein Netzplan (ein Graph, bestehend aus Knoten und Kanten) sowie eine Zahl c von Farben. gesucht ist eine Färbung der Knoten, so daß keine Kante zwischen gleichfarbigen Knoten verläuft.

Suchprobleme: Lunar Lockout

Beispiel 2 (`http://www.thinkfun.com/buy/games/lunarLockout.html`):
Die Großbuchstaben bezeichnen die Roboter. Die Aufgabe ist, A auf die (anfangs leere) Position a zu bewegen.
Roboter fahren geradlinig, ohne Bremse, bis zu Hindernis (= anderer Roboter).
Rand ist *kein* Hindernis

```
B . . . E . . . A
. D . . . . . C .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . a . . . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
G . . . . . . . F
```

Ähnliches Spiel: Ricochet Robot (Rasende Roboter) von Alex Randolph, `http://sunsite.informatik.rwth-aachen.de/keirat/txt/R/Rasenrob.html`

Suchprobleme (2)

Bei COL ist der Suchraum beschränkt: es gibt nur $|\text{Farben}|^{|\text{Knoten}|}$ verschiedene Färbungen.
Für jede Färbung kann man *schnell* (polynomiell) prüfen, ob sie Konflikte enthält.
… es gibt aber *exponentiell viele* Färbungen.

COL gehört zur Komplexitätsklasse NP.
Bedeutung des Namens:

- N: es ist ein Suchproblem
- P: der Suchbaum ist nur polynomiell tief …

… aber exponentiell breit.
Aufgabe: wie komplex ist 2COL (2 Farben)?

Suchprobleme

Bei *Lunar Lockout* ist eine *Folge* von Zügen gesucht.
Nach jeder Folge entsteht eine bestimmte Konfiguration.
Für ein Spielfeld mit f Feldern (z. B. $f = 9 \cdot 10 = 90$) und r Robotern (z. B. $r = 7$) gibt es $\leq (f+1)^r$ Konfigurationen.
In *kürzesten Lösungen* gibt es keine Wiederholungen von Konfigurationen.
Falls eine Konfiguration *überhaupt* lösbar ist, dann hat sie auch eine Lösung mit $\leq (f+1)^r$ Zügen.

In jeder Konfiguration gibt es $\leq 4 \cdot r$ Züge.
Der Suchraum ist ein Baum der Tiefe $\leq (f+1)^r$.
Jeder Knoten hat $\leq 4 \cdot r$ direkte Nachfolger.
Der Baum hat insgesamt $\leq (4 \cdot r)^{(f+1)^r}$ Knoten.
Das ist eine (große, aber) endliche Zahl.
$\Rightarrow$ Das Problem

- Eingabe: eine Lunar-Lockout-Konfiguration
- Frage: gibt es eine Zugfolge, die A nach a bewegt?

ist *entscheidbar*.
(D. h.: es gibt einen Algorithmus, der für jede Eingabe in endlicher Zeit die richtige Antwort ausgibt.)

Suchprobleme (3)

Beispiel 3 (PCP, Postsches Korrespondenz-Problem, Emil Post):
Gegeben ist eine Liste von Wort-Paaren. Gesucht ist eine Folge (von Kopien dieser Paare), bei der die Verkettung alle linken Seiten das gleiche Wort ergibt wie die Verkettung aller rechten Seiten.
Beispiele: $\begin{array}{c|c|c} bba & a & b \\ \hline b & b & ab \end{array} \quad \begin{array}{c|c|c} aab & a & b \\ \hline a & b & aab \end{array}$
Das linke hat kürzeste Lösung $[1, 1, 3, 2, 3, 3, 2, 3]$.
Aufgabe: finde Lösung für das rechte (es gibt eine).

Jetzt ist der Suchraum (alle Folgen) gar nicht beschränkt (die Folgen können beliebig lang sein).
Tatsächlich ist PCP *nicht entscheidbar*!
(Es gibt *keinen* Algorithmus, der für *jede* Liste von Paaren in endlicher Zeit korrekt entscheidet, ob es eine Lösungsfolge gibt.)

Algorithmisch unlösbare Probleme (1)

Behauptung: es gibt algorithmisch unlösbare Probleme.
Das sind wohldefinierte Aufgabenstellungen mit wohldefinierter Lösung, welche aber *nicht* durch Algorithmus gefunden werden kann.

Alg. unlösb. Probleme (2)

Hilfsmittel: Aufzählung (Numerierung) aller Programme (…, die aus *einer* Eingabe *eine* Ausgabe berechnen.)
Da Algorithmen (Programme) beschreibungs-endlich sind, können wir sie auch durchnumerieren:
Z. B. Java-Quelltexte erst der Länge nach und innerhalb der Länge alphabetisch. Die syntax- und typ-fehlerbehafteten Texte streichen wir, übrig bleibt eine Anordnung

$$P_0, P_1, \ldots$$

aller tatsächlich kompilier- und ausführbaren Programm-Texte.

Algorithmisch unlösbare Probleme (3)

Es gibt Rechnungen, die *halten*, und Rechnungen, die *nicht halten* (z. B. „Endlos-Schleifen").

Das *Halteproblem* ist:

- Eingabe: ein Zahl x und eine Zahl y
- Frage: hält das Programm P_x bei Eingabe y?

Wir beweisen, daß das *nicht entscheidbar* ist: es gibt keinen Algorithmus, der diese Frage für alle Eingaben (x, y) korrekt beantwortet.

Diskussion: wir könnten einfach die Rechnug von P_x auf y starten. Aber … wann können wir sie abbrechen?

Algorithmisch unlösbare Probleme (4)

Indirekter Beweis: Falls das Halteproblem doch algorithmisch lösbar ist:
Dann definieren wir das Programm

$$H : x \mapsto \begin{cases} \textbf{falls} & P_x(x) \text{ (Rechnung des Programms mit Nummer } x \text{ für Eingabe } x\text{) } \textbf{hält}, \\ \textbf{dann} & \text{irgendeine nicht haltende Rechnung (Schleife),} \\ \textbf{sonst} & \text{ein haltende Rechung (return 0).} \end{cases}$$

Das Programm H hat auch eine Nummer, diese sei n.
Also $H = P_n$.
Hält $H(n)$?

Wegen der Fallunterscheidung gilt
$H(n)$ hält $\iff H(n) = 0 \iff P_n(n)$ hält nicht $\iff H(n)$ hält nicht!
Das ist ein Widerspruch, d. h. (wenigstens) eine Annahme war falsch:
Die Funktion H ist völlig korrekt definiert, aber *es gibt keinen Algorithmus, der H berechnet*.

Alg. unl. Probleme (5)

der gegebene Beweis beruht auf Ideen von

- Georg Cantor (Diagonalisierung, es gibt überabzählbare Mengen)
- Kurt Gödel (Unvollständigkeitssätze, Grenzen der Formalen Logik)
- Alan Turing (allgemeines Maschinenmodell, Grenzen von Algorithmen)

Die Aussage gilt für *jede* Programmiersprache
(…, in der man die Funktion $x \mapsto P_x(x)$ programmieren kann)

PCPs und Programme

Betrachte $\begin{array}{c|c|c|c} abbb & ccc & ddd & d \\ \hline a & b & c & dd \end{array}$
⇒ mit PCPs kann man „rechnen“.
⇒ man kann jedes Programm in PCPs übersetzen.
Eigenschaften von PCPs (z. B. Lösbarkeit) sind (wenigstens) genauso schwer wie Eigenschaften von Programmen (z. B. Halteproblem).
Halteproblem algorithmisch unlösbar
⇒ PCP algorithmisch unlösbar.
Indiz: sehr kleine und trotzdem schwere PCP-Instanzen (d. h. mit sehr langer Lösung).
`http://www.theory.informatik.uni-kassel.de/~stamer/pcp/`

Grundlagen der (Java-)Programmierung

Modellierung

Zur Lösung von *realen* Aufgaben muß zunächst ein *(mathematisches) Modell* festgelegt werden.
Die reale Aufgabe kann damit übersetzt werden in eine Frage innerhalb des Modells.
Deren Antwort bestimmt man durch einen Algorithmus.
Wenn die Modellierung *angemessen* war, kann man aus der Antwort innerhalb des Modells Rückschlüsse auf die Realität ziehen.
vgl. Horn/Kerner/Forbrig 1.3 (Computer als Werkzeug)

Begriffe

- *Problem*: Instanz und Frage (Aufgabe)
- *Algorithmus*: Handlungsvorschrift zur Lösung von Problemen
- *Programm*: Notation eines Algorithmus in einer Programmiersprache

  (mit dem Ziel der Ausführung auf einer Maschine)
- *Prozeß*: die Ausführung eines konkreten Programms mit konkreten Eingaben auf einer konkreten Maschine
- (*Betriebssystem*: verwaltet Prozesse)

Bespiel-Programm (in Java)

```
static void compute (int x) {
  int count = 0;
  int max = x;
  System.out.print ("start: " + x);
  while (x > 1) {
    if (0 == x % 2) {
      x = x / 2;
    } else {
      x = 3 * x + 1;
    }
    System.out.println (x + " ");
    // TODO: schritte mitzählen
    // TODO: maximum eintragen
  }
  System.out.println (" schritte: " + count + " höchstes
}
```

Java-Programme ausführen

zu Java allgemein vgl. Horn, Kerner, Forbrig, Kaiptel 5.4, zu Einzelheiten siehe Dokumentation auf
`http://java.sun.com`

- Programmtext `Collatz.java` wird übersetzt (kompiliert) in `Collatz.class` (sog. Class-File)
- das kompilierte Programm kann ausgeführt werden:
  - als Applikation (d. h. alleinstehend)
  - als Applet (d. h. in einem Web-Browser)

Der Zwischenschritt über Class-Files dient der Effizienz (weniger Platz, schneller ausführbar)
und (eingeschränkt) der Geheimhaltung
(Quelltext ist nicht direkt sichtbar)

Definition von Programmiersprachen

vgl. Horn, Kerner, Forbrig, Kapitel 5.1

- (Lexik und) Syntax:

  was sind die erlaubten Wörter (= Folgen von Zeichen), was die erlaubten Sätze (= Folgen von Wörtern)?
- Semantik:

  was ist die Bedeutung der erlaubten Sätze?

  (von-Neumann-Modell: welche Änderung des Speicher/Welt-Zustandes bewirken sie?)
- Pragmatik:

  wie benutzt man die Sprache, um Algorithmen *zweckmäßig* auszudrücken?

Syntax: Anweisungen (statements)

- einfache Anweisungen:
  - Zuweisung: `Name = Ausdruck ;`
  - Unterprogramm-Aufruf: `Name ( Ausdruck, ...) ;`
- zusammengesetzte Anweisungen:
  - Block: Folge von Anweisungen, in `{ },`
  - Verzweigung: `if ( Bedingung ) Block`
    oder `if ( Bedingung ) Block else Block`
  - Schleife: `while ( Bedingung ) Block`

Syntax: Ausdrücke (expression)

einfacher Ausdruck	zusammengesetzter A.
• Konstante	• geklammerter Ausdruck
• Variable	• Ausdruck Operator Ausdruck

Jeder Ausdruck hat die Struktur eines *Baumes*
(die Wurzel ist oben!)

Jeder Teil-Ausdruck (Teilbaum) hat:

- einen *Typ*
- und einen *Wert*.

Seminar Java/BlueJ

- BlueJ starten, siehe auch `http://www.bluej.org/`

  BlueJ → project → new project → Name *Foo* → new class → name *Collatz* (großer Anfangsbuchstabe!) → right-click: open editor
- Programmschablone löschen, Collatz-Programm eingeben:

```
class Collatz {
  static void compute (int x) { .. }  // Text
  public static void main (String [] argv) {
    compute (27);
  }
}
```

editor: compile

- Unterprogramm `compute` aufrufen

  project: right-click *compute* → Argument eingeben
- Hauptprogramm aufrufen
- Programm ergänzen (Schrittzahl, Maximum) und autotool-Aufgaben Collatz-Quiz, Collatz-Inverse-Quiz bearbeiten

  Hinweis: falls Programm „hängt“ dann: project → view → show debugger → terminate
- Java-Sprach-Definition betrachten, siehe
  `http://java.sun.com/`
  `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`

Pragmatik

Quelltexte sinnvoll gestalten: *immer an den Leser denken*
einheitliche optische Gestaltung benutzen,
Code soll *selbsterklärend* sein

- Layout-Regeln (nächste Folie)
- sinnvolle Namen benutzen
- „schwierige“ Programmteile kommentieren

  . . . : besser noch: vereinfachen!
- eine Programmeinheit nicht länger als eine Bildschirmseite

  → modularisieren, (re-)faktorisieren

Layout: Indentation

Einrückungen (indentation) sollen Struktur ausdrücken:

- *nach* öffnender Klammer tiefer einrücken
- *vor* schließender Klammer wieder ausrücken
- *sonst* Tiefe beibehalten
- nur eine Anweisung pro Zeile

ergibt solche Bilder:

```
foo bar {
  ...
  ...
}
```

Semantik von Anweisungen

- Zuweisung: `Name = Ausdruck;`
  Wert des Ausdrucks berechnen und an Namen zuweisen (d. h. auf die durch Namen bezeichnete Speicherstelle schreiben)
- Block: `{Anw1 ; Anw2; .. }`
  Anweisungen der Reihe nach ausführen
- Verzweigung:
  `if ( Bedingung ) Block1 [ else Block2 ]`
  Wert der Bedingung (= ein Ausdruck vom Typ Wahrheitswert) berechnen, falls sich „wahr“ ergibt, dann Block1 ausführen, sonst Block2 ausführen (falls vorhanden)

- Schleife: `while ( Bedingung ) Block`
  is äquivalent zu

```
A : if ( not Bedingung ) { goto E; }
    Block
    goto A;
E:
```

D. h. *vor jeder* Ausführung des Blocks wird die Bedingung ausgewertet.

Zählschleifen

Beispiel:

```
for (int i = 0; i < 10; i++) {
  System.out.println (i);
}
```

Syntax:

```
for ( Deklaration ; Bedingung ; Anweisung )
  Block
```

Semantik:

```
Deklaration;
while ( Bedingung ) {
  Block;
  Anweisung;
}
```

Semantik (Werte) von Ausdrücken

- Konstante: bezeichnet „sich selbst“
- Name (Variable): bezeichnet eine Speicherstelle, der Wert des Namens ist der Inhalt dieser Speicherstelle
- Ausdruck1 Operator Ausdruck2: Wert w_1 von Ausdruck1 berechnen, Wert w_2 von Ausdruck2 berechnen, dann Operator auf w_1 und w_2 anwenden, das Resultat ist der Wert des Ausdrucks

das ist (wie alles andere heute auch) eine *induktive (rekursive)* Definition.
(wo sind Induktionsanfang, Induktionsschritt? was ist der Induktionsparameter?)

Typen

reale Objekte haben wesentliche Unterschiede und Gemeinsamkeiten,
das gleiche soll für ihre Darstellung (*Daten*) innerhalb des Modells (... Algorithmus, Programms) gelten: man benutzt *Typen*:

einfache Typen:	zusammengesetzte T.
• ganze Zahlen	• Zeichenketten
• gebrochene Zahlen	• Folgen von Zahlen, ...
• Wahrheitswerte	• binäre Bäume
• Zeichen	• Graphen, ...

im von-Neumann-Modell: jeder Speicherstelle (*Variable*) besitzt *Namen* und *Typ*.

Operatoren in Ausdrücken

- vom Typ Zahl $\times$ Zahl $\rightarrow$ Zahl:

  `+`, `-`, `*`, `/` (Division), `%` (Rest).
- vom Typ Zahl $\times$ Zahl $\rightarrow$ Wahrheitswert:

  `<` (kleiner), `<=` (kleiner oder gleich), `==` (gleich),
  `!=` (ungleich), `>=` (größer oder gleich), `>` (größer).

  Vorsicht: `a == b` ist Ausdruck, `a = b` ist Zuweisung.
- vom Typ Wahrheitswert $\rightarrow$ Wahrheitswert: `!` (Negation)

  vom Typ
  Wahrheitswert $\times$ Wahrheitswert $\rightarrow$ Wahrheitswert:

  `&&` (Konjunktion, und), `||` (Alternative, oder)

Deklarationen

Block: Folge von *Deklarationen* und Anweisungen
Deklaration [mit Initialisierung]:
Typname Variablenname [= Ausdruck] ;

- jeder Name (jede Variable) muß *vor* der ersten Benutzung deklariert werden
- der Typ eines Namens kann sich nicht ändern

```
for (int i = 0; i < 10; i++) {
  int q = i * i;
  System.out.println (i + " " + q);
}
```

Lokalität

jeder Name hat eine Lebensdauer und eine Sichtbarkeit:

- ein Name „lebt“ (d. h. der bezeichnete Speicherplatz existiert)

  von seiner Deklaration bis zum Ende des Blockes, in dem er deklariert wurde.
- ein Name ist in seinem Block ab der Deklaration sichtbar (d. h. kann benutzt werden)

  und allen evtl. darin enthaltenen Blöcken.

Lokalitätsprinzip (information hiding):
alle Deklarationen *so lokal wie möglich*

Unterprogramme

zu große Programme (Bildschirmseite!) zerlegen in (weitgehend) unabhängige Einheiten: *Unterprogramme.*

Unterprogramm ist

- benannter Block

  (= Folge von Deklarationen und Anweisungen)
- mit Schnittstelle

  (Datentransport bei Beginn und Ende der Ausführung)

Unterprogramme (Beispiele)

Schnittstelle festlegen:

```
// liefert true genau dann, wenn die
// Collatz-Folge von x die Länge len hat
static boolean collatz_test (int x, int len) {
}
```

Unterprogramm benutzen:

```
static void solve (int len) {
  for (int start = 0; start < 1000; start++) {
    if (collatz_test (start, len) {
        System.out.println (start);
    }
  }
}
```

Unterprogramme (Implementierung)

```
// hat Collatz-Folge von x die Länge len?
static boolean collatz_test (int x, int len) {
  int count = 0; // Deklaration mit Initialisie
  while (x > 1) {
    if (0 == x % 2) {
      x = x / 2;
    } else {
      x = 3 * x + 1;
    }
    count++;
  }
  return len == count;
}
```

Übung 22. 11.

- Unterprogramme `collatz_test, solve` aus Vorlesung ausprobieren.

  Testfälle für `collatz_test` entwerfen und ausführen.
- Unterprogramme `collatz_test, solve` sollen weiteres Argument `top` erhalten, mit dem das maximale Element der Folge getestet werden kann.

  Erst Schnittstelle und Testfälle entwerfen, dann implementieren und Tests ausführen.

  Dann autotool-Aufgabe CollatzIQ lösen.

Syntax u. Semantik von Unterprogrammen

```
static Typ Name ( Liste der formalen Parameter ) Block
mit  Typ => int, String, .. ;
     formaler Parameter=> Typ Name
```

es gibt zwei Arten von Unterprogrammen:

- *Funktion*

  liefert *Wert* vom angegebenen Typ

  Funktionsaufruf ist *Ausdruck*
- *Prozedur* liefert *keinen* Wert

  (ausgedrückt durch Ergebnis-„Typ“ `void`)

  (engl. *void* = wertlos, leer, ungültig)

  Prozedur-Aufruf ist *Anweisung*

Semantik von `return`

ist besondere Anweisung in Unterprogrammen,

- in Prozeduren: `return;` beendet (sofort) Ausführung der Prozedur
- in Funktionen: `return a;` beendet (sofort) Ausführung der Funktion, Rückgabewert ist Wert von `a`

Beachte:

- `return` in Prozeduren kann entfallen

  (dann Rückkehr nach Block-Ende),
- `return a` in Funktionen ist vorgeschrieben.

Unterprogramme und Typen

- Für jedes Unterprogramm müssen die Typen der Argumente und des Resultats festgelegt werden.
- Bei jeder Verwendung eines Unterprogramms prüft der Compiler (!), ob die Typen der verwendeten Argumente und des Resultats mit der Deklaration übereinstimmen.
- Die Prüfung findet nur einmal statt: zur Compile-Zeit — ist also zur Laufzeit *nicht* mehr nötig.
- möglichst frühe und möglichst strenge Typ-Prüfung verhindert Laufzeitfehler und ermöglicht schnellere Programm-Ausführung.

Nutzen von Unterprogrammen/Schnittstellen

- in größeren Software-Projekten geschehen

  Schnittstelle, Test-Entwurf, Implementierung, Test

  zu verschiedenen Zeitpunkten, von verschiedenen Personen (Abteilungen, Firmen)

  ⇒ Modularisierung, Wiederverwendung
- auch in kleinen Projekten:

  Verwendung von vorgefertigten Unterprogrammen (Bibliotheken)

  Kenntnis der Schnittstelle reicht für Benutzung.

Re-Faktorisierung

re-faktorisieren: „hinterher (anders) zerlegen“
bereits benutzte Software besser strukturieren, damit man sie besser warten und wiederverwenden kann.

Beispiel:
ein Unterprogramm, das einen Schritt der inneren Schleife der Collatz-Folge berechnet:
`static int step (int x) { ... }`
Beispiel-Werte: `step(5) = 16; step (8)=4;`
dann einen geeigneten Unterprogramm-Aufruf in `collatz_test` einsetzen.

Aufgaben zu Unterprogrammen

Schnittstellenbeschreibungen in `/** .. */`, daraus kann HTML-Seite erzeugt werden (ansehen mit Implementierung/Schnittstelle im BlueJ-Editor)

- ```
   /**
    * Zeile von 'x' drucken, dann Zeilenschaltung
    *
    * @param  n  Anzahl der 'x'
    */
  void zeile (int n) {
      for ( int i = 0 ; i < n ; i++ ) {
          System.out.print ("x");
      }
      System.out.println ();
  ```
  ```

```
}

zeile (15) liefert Ausgabe:

xxxxxxxxxxxxxxx
```

- ```
  /** drucke Rechteck
   * @param b Breite
   * @param h Hoehe
   */
  static void rechteck ( int b, int h ) { ... }

  rechteck (3,2) liefert Ausgabe:

  xxx
  ```

```
xxx
```

- ```
  dreieck (5) liefert Ausgabe:

  x
  xx
  xxx
  xxxx
  xxxxx
  ```
- Quadrat der Seitenlaenge b
- Folge von Quadraten

```
x
```

```
xx
xx

xxx
xxx
xxx
```

- Aufgabe: erzeuge Schachbrett-Rechtecke, z. B.

```
x x x x x x
 x x x x x x
x x x x x x
 x x x x x x
```

Unterprogramme: Türme von Hanoi

- drei Türme A, B, C.
- anfangs auf A die Scheiben $[32, 31, \ldots, 2, 1]$, B leer, C leer.
- gesucht: eine Folge von Bewegungen *einzelner* Scheiben, so daß:
- niemals eine größere über einer kleineren liegt
- und schließlich alle auf B liegen.

Türme von Hanoi: Entwurf

Spezifikation (Schnittstelle) einer Prozedur

```
static void hanoi
    (int s, String v, String n, String h)
```

soll Folge der Bewegungen drucken, die nötig sind, um Scheiben $[s, s-1, \ldots, 2, 1]$ von Turm `v` nach Turm `n` zu bewegen, unter (eventueller) Benutzung des Turms `h`.

Beispiele:

```
hanoi (1, "A", "B", "C")   => (1 von A nach B)
hanoi (2, "B", "C", "A")   =>
   (1 von B nach A)(2 von B nach C)(1 von A nac
hanoi (3, "A", "C", "B")   => ?
```

Beachte: zum Festlegen der Schnittstelle von `hanoi` war schon ein Teil der Problem-Analyse nötig

Türme von Hanoi: Implementierung (I)

```
// Scheiben [ 1 .. s ]
// von  v  nach  n  über  h
static void hanoi
   (int s, String v, String n, String h)
```

- wenn `0 == s`, dann tun wir gar nichts
- wenn `0 < s`, dann

  Scheibe s muß wenigstens einmal bewegt werden (von v nach n).

  Wo liegen zu diesem Zeitpunkt die anderen Scheiben?

  Wie kommen sie dorthin? von dort weg?

Hanoi: Implementierung (II)

```
static void move
    (int s, String v, String n) {
  System.out.println
    (s + " von " + v + " nach " + n);
}
// Testfall:  hanoi (4, "A", "B", "C");
static void hanoi
    (int s, String v, String n, String h)
{ if (0 < s) {
    hanoi (s - 1, v, h, n);
    move  (s, v, n);
    hanoi (s - 1, h, n, v);
} }
```

Hanoi: Bewertung/Aufgabe

- die angegebene Lösung ist optimal

  (Grund: Scheibe s muß wenigstens einmal bewegt werden, und sie wird nur genau einmal bewegt.)
- wieviele Schritte (für $3, 4, 5, \ldots, s$ Scheiben) erfordert angegebene Lösung?
- Aufgabe (autotool): wie lautet die optimale Lösung für *vier* Türme und $6, 8, 10, \ldots$ Scheiben?

  (Es sollte ja schneller gehen als für drei.)
- gibt es einen einfachen *iterativen* Algorithmus (wie kann man die nächste Bewegung ausrechnen, ohne ihre Vorgeschichte zu kennen)?

Rekursion

Definition: ein Unterprogramm heißt *rekursiv*, wenn es sich selbst aufruft.

Beispiel: Verarbeitung von rekursiven Datenstrukturen

```
int size (Tree b) {
  if (b ist Blatt) { return 1; }
  else { return
    1 + size (links(b)) + size (rechts(b));
} }
```

Beispiel (John McCarthy): Berechne `f(7); f(77);` für

```
static int f (int x) {
  if (x > 100) {    return x - 10; }
  else { return f (f (x + 11));  }
}
```

Seminar 29. 11.

Median, Hanoi

Objekt-Orientiertes Programmieren

Objekte, Methoden

Literatur: HKF Kapitel 5.4 (ab S. 291)

- Daten sind *passiv* (man kann sie nur lesen/schreiben).
  Operationen mit Daten durch Unterprogramme.
- Entwurfsziel: Daten und passende Operationen verbinden.
- Lösung: Aus Daten werden *Objekte*:
  Definition: ein Objekt besteht aus
  - Attributen (Daten)
  - Methoden (Unterprogrammen)

Klassen

Definition: Eine Klasse beschreibt gleichartige Objekte (gleiche Namen und Typen für Attribute und Methoden).

Klasse definieren:

```
public class Counter
{
  int ticks = 0;
  void reset ()
    { ticks = 0; }
  void step  ()
    { ticks ++ ; }
}
```

Objekt deklarieren, initialisieren, Methode aufrufen, Attribut lesen:

```
{ Counter c =
    new Counter ();
  c.step ();
  System.out.println
    (c.ticks);
}
```

- *Attribut:* Objektname . A.-Name
- *Methoden-Aufruf:* Objektname . M.-Name Argumentliste

Objekte in Bluej

Rechts-Klick auf Klassen-Symbol zeigt Konstruktor `new Name ()`.
Klick darauf legt neues Objekt an, erscheint als (rotes) Symbol.
Rechts-Klick auf Objekt-Symbol zeigt Methoden (Click auf Methode ruft auf).
Click auf `inspect` zeigt Attribute.

Lebenslauf eines Objekts

- (Deklaration) `Counter c ...;`
- Konstruktion/Initialisierung `... = new Counter ();`
- Leben:
  Methoden aufrufen, Attribute lesen und schreiben
- Finalisierung (erfolgt automatisch)

Ein Objekt wird durch Aufruf eines *Konstruktors* hergestellt
Form: `new` Klassenname Argumentliste;
Dabei werden Attribute initialisiert (`int ticks = 0;`)

Zeichenketten

vordefinierte Klasse `String`

```
String s = "foobar";
int  l = s.length (); // Wert: 6
char c = s.charAt (3); // Wert: 'b'
String t = s.substring (1, 3); // Wert: "oob"
```

Strings (Aufgabe)

- implementieren Sie eine Funktion

  `static boolean ist_palindrom (String s)`

  so daß `palindrom ("reliefpfeiler") = true`.

  Benutzen Sie `s.length(), s.charAt(..), while`.
- Suchen Sie damit Zahlen n mit:

  n ist kein Palindrom, aber n^2 ist ein Palindrom.

  Beispiel: $798644^2 = 637832238736$.
- (Gibt es unendlich viele solche Zahlen?)

  (desgl. für dritte Potenz)

- Hinweis: benutzen Sie nicht `int n`, sondern `long n`, sowie `Long.toString (n)`.

Statik und Dynamik

solange nichts anderes deklariert ist, gehören jedes Attribut und jede Methoden zu einem Objekt:

- jedes Objekt hat eigenen Speicherplatz für Attribute
- Benutzung der Attribute und Aufruf der Methoden ist nur über Objektnamen möglich

Durch Deklaration `static`: Attribute/Methoden gehören zur *Klasse* (und nicht zu einem einzelnem Objekt).
Benutzung über Klassen- (nicht: Objekt-)Namen.
Bsp: `int x = Integer.parseInt ("123");`
In statischen Methoden sind nur statische Attribute und Methoden benutzbar (warum?)

Der Operator $+$, die Methode `toString`

- Operator + ist überladen:

  hat Typen *Zahl* $\times$ *Zahl* $\rightarrow$ *Zahl*

  und *String* $\times$ *String* $\rightarrow$ *String*: erzeugt Verkettung von zwei Strings.

  `"foo" + "bar"` hat Wert `"foobar"`
- Jedes Objekt besitzt Methode `String toString ()`.
- Wenn genau ein Argument von + ein String ist, dann wird auf das andere `toString()` angewendet.

Beispiel:

```
int x = 3; System.out.println ("x = " + x);
```

Überladen von Namen

Ein Methoden-Name kann *überladen* sein: er bezeichnet *verschiedene* Methoden.
Das ist gestattet, falls man die Methoden anhand der Argumentliste (Länge und Typen) unterscheiden kann.

```
public class C {
    void p () { ... }
    void p (int x) { ... }
}
```

Beachte: Überladung mit gleichen Argument-Typen und verschiedenen Ergebnis-Typen ist nicht erlaubt.
Aufgaben: kann man Prozedur durch Funktion überladen? Methode durch Attribut? Attribut durch Attribut?

Konstruktoren

Jede Klasse besitzt den *default*-Konstruktor (mit leerer Argumentliste).
Seine Ausführung stellt ein Objekt der Klasse her und initialisiert alle Attribute.
Man kann weitere Konstruktoren hinzufügen (d. h. den Konstruktor-Namen überladen)

```
public Counter (int x) { ticks = x; }
...
Counter c = new Counter (3);
```

Konstruktor-Methode ist Funktion ohne Namen, Ergebnistyp ist die Klasse.

Weitere Aufgabe zu Palindromen

Hält das folgende Programm immer?

```
while ( x ist kein Palindrom ) {
    x =  x + Spiegelzahl von x;
}
```

Beispiel: 199, 1190, 2101, 3113.
Überprüfen Sie alle Start-Zahlen ≤ 1000.

Vererbung, Interfaces

Beziehungen zwischen Klassen

D *ist abgeleitet von* C (oder: D *erweitert* C):
D besitzt

- alle Attribute und Methoden von C
- und weitere, eigene.

```
class C {
  int a;
  void m () { ... }
}
class D extends C {
  int b;
  void p () { ... }
}
```

Beispiele:

- Basis: Zähler mit step, abgeleitet: ... und reset
- Basis: Grafik-Element, abgeleitet: ... mit Farbe

Kompatibilität

überall, wo Objekte einer Basisklasse C erwartet werden, dürfen auch Objekte davon abgeleiteter Klassen D benutzt werden:

```
class C { .. } ; class D extends C { .. }
```

- bei Deklarationen, Zuweisungen:

```
C x = new D ();
```

- bei Unterprogramm-Aufrufen:

```
static void p (C x) { .. }
D y = new D (); p (y);
```

Überschreiben von Methoden

Abgeleitete Klassen können Methoden der Basisklasse neu implementieren (*überschreiben*).

```
class C {
  int a;
  void m () { ... }
}
```

```
class D extends C {
  void m () { ... }
}
```

Es wird immer die *speziellste* Methode benutzt:

```
C x; x.m ();  D y; y.m ();
```

Überschreiben $\neq$ Überladen

- *Überschreiben*: Methoden
  - in *verschiedenen* Klassen,
  - mit *übereinstimmender* Schnittstelle
- *Überladen*: Methoden
  - in *einer* Klasse,
  - mit *unterschiedlicher* Schnittstelle

```
class C { int  f (int x) { .. }
          int  f (String x) { .. }
          void g (boolean y) { .. }
}
class D extends C { int f (int x) { .. }
          void g (boolean y, String z) { .. }
}
```

Objektorientierung (Überblick)

OO =
- Objekte mit Attributen und Methoden,
- Beziehungen: Vererben und Überschreiben.

- Simula 68 (Ole-Johan Dahl, Kristen Nygard) (Prozess-Simulation) Coroutinen, Klassen, Objekte
- Smalltalk `http://www.smalltalk.org`, (Adele Goldberg, Alan Kay, Xerox Parc, ab 1972) (Grafische Nutzeroberflächen)
- C with Classes, C++ (Bjarne Stroustrup, ab 1980)
- Java (James Gosling, 1995) `http://java.sun.com/features/1998/05/birthday.html`

(Grafik, Kommunikation, für mobile Endgeräte)

Objektorientierte Analyse/Modellierung

nicht nur Programme, sondern (technische) Systeme beschreiben:

- Komponenten (Objekte),
- Eigenschaften (Attribute, Methoden)
- Gemeinsamkeiten (Klassen)
- Hierarchien (Vererbung)

dafür gibt es standardisierte Verfahren, Werkzeuge und Sprachen (UML).
automatische Unterstützung von OO-Programmierung durch Generierung von Programmtexten (-Bausteinen).

Vererbung: Vor/Nachteile

- Vorteil: Nachnutzung von Programmen

  (gleicher Code für verschiedene Typen: *Polymorphie*)

- Nachteil: abgeleitete Klasse sieht alle Details der Basisklasse

verletzt das Prinzip des *information hiding*: so wenig wie nötig interne Klassen-Informationen nach außen geben, damit nachträglich Implementierung verbessert oder ausgetauscht werden kann.
Für größere Projekte: Modularisierung und Information Hiding durch andere Techniken erzwingen (OO kann das gar nicht alles leisten)

Information Hiding

Idee: verstecke die internen Daten eines Objektes.
Attribute nur innerhalb der Klasse lesbar (`private`),
Zugriff von außen nur durch Methoden (`set, get`).

```
public class Counter {
  private int ticks;
  void reset ()
    { ticks = 0; }
  void step  ()
    { ticks ++ ; }
  int  get  ()
    { return ticks; }
}
```

```
{ Counter c = new C ()
  c.reset ();
  c.step  ();
  System.out.println
        (c.get ());
}
```

Vorteil: Klasse „bemerkt“ Änderung der Attribute, Methoden sorgen für Konsistenz.

Analogie: Buchhalter, doppelte Buchführung

Abstrakte Methoden und Klassen

Attribute verstecken → wichtig sind Methoden.

```
abstract class C {
  void p () { .. }; // Deklaration und Implementierung
  abstract void m (); // nur Deklaration
}
class D extends C {
  void m () { .. } // Implementierung
}
```

abstrakte Methode ist in Basisklasse deklariert, (aber nicht implementiert), muß in abgeleiteten Klassen implementiert werden.
Basisklasse muß als *abstract* deklariert werden, wenn wenigstens eine Methode abstrakt ist.
Abstrakte Klasse kann nicht instantiiert werden (≈ besitzt keinen Konstruktor).

Schnittstellen (Interfaces)

Schnitstelle = Klasse mit *nur* abstrakten Methoden

```
statt: abstract class C { abstract void m (); }
schreibe: interface C { void m (); }
```

Klassen können Schnittstellen *implementieren*:

```
statt:    class D extends C { .. }
schreibe: class D implements C { void m () { ..
```

Eine Klasse kann mehrere Schnittstellen implementieren:

```
class D implements C1, C2 {
  void m1 () { .. } ; void m2 () { .. }
}
```

Beachte: in Java gibt es Mehrfach-Vererbung nur für Schnittstellen (implements), nicht für Klassen (extends)
. . . und das ist gut so. Klassen soll man *gar nicht* vererben.

Applet-Programmierung

Applets

(Literatur: HKF ab S. 299)
Applet: in Webseite eingebettetes Programm
Ausführung: im Web-Browser (zum Testen: im Applet-Viewer).

```
import java.applet.Applet;
import java.awt.*;
public class Counter extends Applet {
  Label lab; Button inc;
  public void init () {
    lab = new Label ("0");
    inc = new Button ("inc");
    add (lab); add (inc);
  }
}
```

Applets mit BlueJ programmieren und testen

- Project/New Class/*applet* (erzeugt Beispiel)
- Edit (Beispiel nur ansehen), Compile
- rechts-click auf Klassensymbol: Run Applet/Run Applet in appletviewer

Ereignis-Behandlung in Applets (I)

```
public class Counter {
  private int ticks = 0;
  int get () { return ticks; }

  Label lab = new Label ("");
  void update () {
    lab.setText (Integer.toString (get ()));
  }
  ...
  public void init () {
    add (lab);
    ...
  }
}
```

Ereignis-Behandlung in Applets (II)

```
public class Counter { ...
 void step () { ticks++; }
 Button inc = new Button ("inc");
 class Inc_Listen implements ActionListener {
  public void actionPerformed (ActionEvent a) {
    step(); update ();
  }
 }
 public void init () { ...
  add (inc);
  inc.addActionListener (new Inc_Listen ());
 }
}
```

Aufgaben: füge Knöpfe für *decrement* und *reset* hinzu.

Applets in Webseiten einbetten

erzeuge Datei `Counter.html`:

```
<html>
<head> <title>Counter Applet</title> </head>
<body> <h1>Counter Applet</h1>
   <applet code="Counter.class"
           width=500 height=500
   >
   </applet>
</body> </html>
```

(BlueJ: Run Applet/Generate Web Page Only)

Datei `Counter.class` enthält Bytecode, entsteht durch Kompilieren von `Counter.java`

(Quelltext ist zur Applet-Ausführung nicht nötig.)

Applet und andere Klassen (20. 12.)

```
java.lang.Object
  extended by java.awt.Component
    extended by java.awt.Button
    extended by java.awt.Container
      extended by java.awt.Panel
        extended by java.applet.Applet
```

vgl. `http://java.sun.com/j2se/1.4.2/docs/api/java/applet/Applet.html`

typische Methoden:

- Applet: init, (start, stop, destroy)
- Container: add, (setLayout)
- Button: addActionListener, (setBackground)

Layout-Manager

- die darzustellenden Elemente (Component) werden der Zeichenfläche (Panel, Container) durch `add` hinzugefügt.
- jeder Container besitzt einen Layout-Manager, der die Anordnung der Elemente bestimmt.
- der Default-Manager ist `FlowLayout ()`, es gibt andere, zum Beispiel:

```
    public void init () {
        setLayout (new GridLayout (3,7));
        ...
```

Zusammenfassung Applet-Grundlagen

- Applet ist in Webseite eingebettetes Programm, das vom Browser des Betrachters ausgeführt wird.
- Applet enthält Zeichenfläche (`Panel extends Container`), zu dieser werden darzustellende Elemente (`Component`) hinzugefügt (`add`)
- Herstellen der Zeichenfläche geschieht in einer Methode `public void init ()`

Zusammenfassung Ereignis-Behandlung

Bei Betätigung eines Eingabe-Elements `e` sollen Anweisungen `A1; A2; ...` ausgeführt werden:

- eine Klasse `C` schreiben, die das Interface `ActionListener` implementiert:

  besitzt eine Methode

  ```
  public void
  actionPerformed (ActionEvent e) { A1; A2; ...
  ```

- dem Eingabe-Element `e` eine Instanz der Klasse `C` zuordnen:

  ```
  e.addActionListener (new C ());
  ```

Applikationen

= Programme (Anwendungen), die direkt auf dem eigenen Rechner ausgeführt werden
müssen Hauptprogramm (Methode `main` von *genau* diesem Typ) besitzen:

```
class Foo {
  public static void main (String [] argv) {
    System.out.println ("bar");
  }
}
```

Kompilieren, dann Start von Kommandozeile:

```
javac Foo.java # Name der Quelltext-Datei
java  Foo      # Name der Klasse
```

Grafische Applikationen

Hauptprogramm macht Fenster-Objekt(e) sichtbar:

```
import java.awt.*;
class Bar {
  public static void main (String [] argv) {
    Frame f = new Frame ("Bar");
    f.setSize (200, 300);
    f.setVisible (true);
  }
}
```

dabei `Frame extends Window extends Container`
vgl. `Applet extends Panel extends Container`
d. h. Objekte mit `add` darstellen

Fenster schließen

Das „Schließen"-Ereignis behandeln:

```
static class Closer extends WindowAdapter {
  public void windowClosing (WindowEvent e) {
    e.getWindow().dispose();
    System.exit (0);
  }
}
public static void main (String [] argv) {
  Frame f = new Frame ("Foobar");
  f.addWindowListener (new Closer ());
  ...
}
```

Anonyme Klassen

(Wiederholung) zur Behandlung eines Ereignisses:

```
class AL implements ActionListener {
    public void actionPerformed (ActionEvent ae
        ...
    }
}
c.addActionListener (new AL ());
```

kürzere Schreibweise mit *anonymer Klasse*:

```
c.addActionListener ( new ActionListener () {
    public void actionPerformed (ActionEvent ae
        ...
    }
} ) ;
```

Ausnahmen

Ausnahmen (Exceptions)

Ausführung einer Anweisung kann fehlschlagen (Exception auslösen), Exception kann behandelt werden:

```
String c = input.getText();
try {
    int i = Integer.parseInt(c);
    result.setText(Integer.toString(i * i));
} catch (Exception ex) {
    result.setText(ex.toString());
} finally {
    doLayout();
}
```

Weiterreichen von Exceptions

Wenn in einem Unterprogramm eine Exception auftreten kann,
aber dort *nicht* behandelt wird,
dann muß das deklariert werden:

```
void foo () throws IOException {
    ...
}
```

Die Exception wird dann an das aufrufende Programm weitergegeben.

Layout-Manager

GUIs und Layout-Manager

Erklärungen und Beispiele: `http://java.sun.com/developer/onlineTraining/GUI/AWTLayoutMgr/`
Ein *GUI* (graphical user interface) enthält mehrere *Komponenten* (z. B. Labels, Buttons), die in einem *Container* (z. B. Panel) angeordnet werden:

```
public class Thing extends Applet {
  public void init () {
    Button b = new Button ("bar"); add (b);
    Button f = new Button ("foo"); add (f);
  }
}
```

Explizite Positionierung (pfui)

```
setLayout (null);
Button b = new Button ("bar");
b.setBounds (200,300,50,30);
add (b);
Button f = new Button ("foo");
f.setBounds (100,200,100,40);
add (f);
```

- keine Anpassung an variable Rahmengrößen
- keine Anpassung an variable Elementgrößen
- viel zu viele Zahlen

PS: Zahlen in Programmtexten

im Programmtext sollten höchstens die Zahlen `0` und `1` einzeln vorkommen,
alle anderen sind als benannte Konstanten deklarieren
nicht:

```
   Punkt [] [] feld = new Punkt [11] [9]; ...
   for (int i=0; i<9; i++) { ... }
```

sondern:

```
   final int breit = 11;
   final int hoch  =  9;
   Punkt [] [] feld = new Punkt [breit] [hoch];
   for (int zeile=0; zeile<hoch; zeile++) { ...
```

Programmtext wird besser lesbar, weniger fehleranfällig, besser konfigurierbar.

Implizite Positionierung durch Manager (gut)

jedem `Container` ist ein `LayoutManager` zugeordnet:
fließende Anordnung (FlowLayout):

```
setLayout (new FlowLayout ()); // ist Default
for (int k = 0; k < 100; k++) {
    add (new Button
           ( "B" + Integer.toString (k) ));
}
```

Beachte Wirkung von Window-Resize!
Gitter-Anordnung (GridLayout)

```
setLayout (new GridLayout (10,0));
```

Manager: BorderLayout

Rahmen-Anordnung:

```
setLayout (new BorderLayout ());

add (new Button ("Top"), BorderLayout.NORTH);
add (new Button ("Foo"), BorderLayout.WEST);
add (new Button ("Bar"), BorderLayout.EAST);
add (new Button ("Bot"), BorderLayout.SOUTH);
add (new Button ("Mid"), BorderLayout.CENTER);
```

Hier kann aber jeweils nur ein Element stehen — schade.

Container als Elemente von Containern

```
setLayout (new BorderLayout ());

add (new Button ("Top"), BorderLayout.NORTH);
add (new Button ("Foo"), BorderLayout.WEST);

Panel p = new Panel ();
p.setLayout (new GridLayout (10,0));
for (int k = 0; k < 97; k++) {
  p.add (new Button( "B" + k ));
}
add (p, BorderLayout.CENTER);
```

beachte: das ist möglich wegen

```
class Container extends Component
```

(Bevorzugte) Abmessungen

Der Typ `Dimension` beschreibt Rechtecke.
(die meisten) Komponenten haben *fließende* Abmessungen (d. h. können in verschiedenen Größen dargestellt werden).
Jede Komponente hat Methoden

```
public Dimension getPreferredSize ();
public Dimension getMinimumSize ();
public Dimension getMaximumSize ();
```

Ein Layout-Manager *kann* diese `Size`s seiner Komponenten berücksichtigen,
… und muß selbst die `Size`s seines Containers ausrechnen.

Management von Abmessungen

- FlowLayout:
  - stellt jede Komponente in preferredSize dar
  - preferred size des Containers: alles in einer Zeile
  - falls Container starr, dann Zeilenumbrüche
- GridLayout:
  - Umbrüche nach festgelegter Zeilen- *oder* Spalten-Zahl `GridLayout(z,0)` oder `GridLayout(0,s)`
  - stellt alle Komponenten gleichgroß dar
  - bestimmt dazu Maximum aller preferred sizes
  - verkleinert/vergrößert alles so, daß es paßt
  - preferredSize: nicht verkleinern

BorderLayout und preferred sizes

- Nord und Süd: Höhe ist preferred Höhe der Komponente,
- West und Ost: Breite ist preferred Breite,
- Mitte: was übrigbleibt

Tip: oft hilft BorderLayout mit nur zwei oder drei Komponenten.
Aufgabe: wann ist FlowLayout innerhalb anderer Container sinnvoll? (selten!)

Layout (Zusammenfassung)

- Durch geeignete Schachtelung von Containern (Panels)
- und jeweils geeignete Manager
- lassen sich *alle* vernünftigen Layout-Aufgaben lösen,
- *ohne* auch nur eine einzige explizite Koordinate anzugeben.

Für GUI-Entwurf: benutze Skizze (Zeichnung):

- gegenseitige Lage der Komponenten (Rechtecke)
- Verhalten bei Resize (Pfeile)

Übung 13. 1.

`http://www.imn.htwk-leipzig.de/~waldmann/edu/ws05/informatik/manage/`

Datenstrukturen

Felder (Arrays)

sind Realisierung von Vektoren (und Matrizen)
Feld ist Behälter für mehrere Werte,
Zugriff (Lesen/Schreiben) erfolgt über Index.
Notation mit eckigen Klammern:

```
int [] a = { 3, 1, 4, 1, 5, 9 };
int sum = 0;
for ( int i = 0; i < a.length; i++ ) {
   sum = sum + a[i];
}
```

Felder: Deklarationen

Deklaration mit Initialisierung des Feldes ...

- ... ohne Initialisierung der Inhalte:

  ```
  String [] msg = new String [3];
  ```

- ... mit Initialisierung der Inhalte

  (Länge muß nicht angegeben werden)

  ```
  String [] msg = { "foo", "bar", "oof" };
  ```

ein Feld kann seine Länge nicht ändern.

Anwendung: Sortieren

```
static void bubblesort (int [] a) {
 for (int i=a.length-1; i>=0; i--) {
  for (int j=0; j<i; j++) {
   if (a[j] > a[j+1]) {
    int h = a[j]; a[j] = a[j+1]; a[j+1] = h;
   }
  }
 }
}
```

Übung: ein Feld mit zufälligen Zahlen füllen, dann sortieren, dabei vor jedem Test Feld-Inhalt ausgeben, Vertauschungen protokollieren

Mehrdimensionale Felder

als Feld von (Feld von ...) Elementen

```
int size = 9;
Cell [][] brett = new Cell [size] [size];

public void init () {
  this.setLayout(new GridLayout(size,0));
  for (int row=0; row < size; row++) {
    for (int col=0; col < size; col++) {
      brett[row][col] = new Cell ();
      this.add ( brett[row][col].visual() );
    }
  }
}
```

Listen

repräsentiert Folge von Elementen $[y_0, y_1, \ldots, y_{n-1}]$

- Einfügen: `void add (int i, Object o)`:
  aus Liste $[y_0, y_1, \ldots, y_{i-1}, y_i, \ldots, y_{n-1}]$ wird Liste $[y_0, y_1, \ldots, y_{i-1}, o, y_i, \ldots, y_{n-1}]$.
- Lesen: `Object get (int i)`:
  Liste ist (und bleibt) $[y_0, y_1, \ldots, y_{n-1}]$, Resultat ist y_i.
- Entfernen: `Object remove (int i)`:
  aus Liste $[y_0, y_1, \ldots, y_{i-1}, y_i, y_{i+1}, \ldots, y_{n-1}]$ wird Liste $[y_0, y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{n-1}]$, Resultat ist y_i
- testen: `size`, `isEmpty` (Deklarationen?)

Beachte: bei add und remove ändern sich die Indizes der Elemente auf bzw. nach i.

Implementierungen von Listen

`List` ist ein *abstrakter Datentyp*
(Java: ein `interface` und keine `class`)
es gibt verschiedene Implementierungen, die wichtigsten:

- `ArrayList`

  Zugriff schnell, Einfügen teuer

- `LinkedList`

  Zugriff langsam, Einfügen schnell

Kellerspeicher (Stacks)

ein *Keller* is eine Folge von Elementen $[y_1, y_2, \dots, y_n]$
Zugriffe (Lesen, Schreiben) passieren *nur* am linken Ende!

- `Stack ()` Konstruktor, erzeugt leeren Keller []
- einkellern: `void push (Object o)`: aus Keller $[y_1, y_2, \dots, y_n]$ wird Keller $[o, y_1, y_2, \dots, y_n]$
- ansehen: `Object peek ()`: Keller ist (und bleibt) $[y_1, y_2, \dots, y_n]$, Resultat ist y_1.
- auskellern: `Object pop ()`: aus Keller $[y_1, y_2, \dots, y_n]$ wird Keller $[y_2, \dots, y_n]$, Resultat ist y_1.
- testen: `boolean empty ()`: ist Keller leer (gleich [])?

Implementiere peek() durch die anderen Methoden.

Wie kann man einen Keller kopieren (nur unter Benutzung der angegebenen Methoden)?

Warteschlangen (Queues)

Eine Schlange ist eine Folge $[y_1, y_2, \dots, y_n]$, Schreiben nur links gestattet, und Lesen nur rechts.
(in Java-1.4 als Teil von `LinkedList`

- `LinkedList ()` Konstruktor, erzeugt []
- schreiben `void addFirst (Object o)`: aus $[y_1, y_2, \dots, y_n]$ wird $[o, y_1, y_2, \dots, y_n]$
- lesen: `Object removeLast ()`: aus $[y_1, y_2, \dots, y_{n-1}, y_n]$ wird $[y_1, \dots, y_{n-1}]$, Resultat ist y_n.
- testen: `boolean isEmpty ()`: ist Schlange leer, d. h. gleich []?

Übung: suche Deklaration der Methode `isEmpty`

Bäume

Hierarchisch angeordnete Sammlung von *Knoten*
Jeder Knoten hat

- einen Schlüssel (Wert, Inhalt) (ein Objekt)
- eine Liste von Knoten (den Kindern)

Beispiele: Verwaltungen, HTML-Dokumente, biologische Taxonomien, Menüs bei AV-Geräten,
Bezeichnungen:

- *Wurzel*: der Knoten, der kein Kind ist
- *Blatt*: Knoten mit leere Kinderliste
- *innerer* Knoten: kein Blatt.

Eigenschaften von Bäumen

Beziehung zwischen Anzahl der Knoten und Kanten?
Jeder Baum ist

- minimal zusammenhängend

  (durch Löschen einer beliebigen Kante zerfällt der Graph)
- maximal kreisfrei

  (durch Einfügen einer beliebigen weiteren Kante entsteht ein Kreis)

in jedem Baum gibt es zwischen je zwei Knoten genau einen Pfad.
Pfadlänge = Anzahl der Kanten (Verbindungen)
Höhe eines Baumes: maximale Pfadlänge.

Binäre Bäume

- binärer Baum: jeder innere Knoten hat *genau zwei* Kinder (links, rechts)
- vollständiger binärer Baum: alle sind Blätter gleich weit von Wurzel entfernt.

Wieviele Knoten hat ein vollständiger binärer Baum der Höhe h?

Baum-Durchquerungen

Beispiel: Ausgabe von Operator-Ausdrücken:

```
void print (Knoten t) {
  if t ist Blatt { print (t.key); }
  else { print ( t.left );
         print ( t.key );
         print ( t.right );
       }              }
```

Beispiel: Auswertung von Operator-Ausdrücken:

```
int wert (Knoten t) {
  if t ist Blatt { return t.eintrag; }
  else { int l = wert ( t.links );
         int r = wert ( t.rechts );
         return (l `t.key' r);
```

```
        }                }
```

Pre-, In-, Post-Order

- pre-order: Wurzel, linker Teilbaum, rechter Teilbaum
- in-order: linker Teilbaum, Wurzel, rechter Teilbaum
- post-order: linker Teilbaum, rechter Teilbaum, Wurzel

Ordne zu: Operator-Ausdruck drucken, Türme von Hanoi, Operator-Ausdruck auswerten, Erlaß einer Regierung bekanntgeben/umsetzen
(autotool) Rekonstruiere den binären Baum aus:
pre-order [5, 1, 7, 0, 9, 8, 2, 4, 3],
in-order [7, 1, 0, 5, 2, 8, 4, 9, 3]

Durchquerung ohne Rekursion

In welcher Reihenfolge werden hier die Knoten besucht?

```
int wert (Knoten t) {
  Stack s = new Stack ();
  push (t);
  while (! s.empty ()) {
    Knoten x = s.pop ();
    print ( x.key );
    if x ist kein Blatt {
      s.push (x.right);
      s.push (x.left);
 } } }
```

Mit der Schlange durch den Baum

In welcher Reihenfolge werden hier die Knoten besucht?

```
int wert (Knoten t) {
  LinkedList s = new LinkedList ();
  addFirst (t);
  while (! s.empty ()) {
    Knoten x = s.removeLast ();
    print ( x.eintrag );
    if x ist kein Blatt {
      s.addFirst (x.links);
      s.addFirst (x.rechts);
 } } }
```

heißt *level-order* (auch Rekonstruktions-Aufgabe)

Übung 17. 1.

- Abnahme der Hausaufgaben (Layout-Manager)
- (gleichzeitig) autotool-Aufgaben zu Bäumen `Baum-{Pre,In,Post,Level}`

- Stacks ausprobieren:

```
import java.util.*;
class Data {
  public static void main (String [] argv) {
    Stack s = new Stack ();
    for (int i=0; i<10; i++) {
      s.push (new Integer (i));
      System.out.println (s);
    }
  }
}
```

- Ändern Sie das o. g. Beispiel von Stack/push auf LinkedList/addFirst oder . . . /addLast.

- Fügen Sie diesen Code an:

```
for (int i=0; i < s.size(); i++) {
    LinkedList h = new LinkedList ();
    while (! s.isEmpty()) {
        h.addFirst (s.removeFirst ());
        h.addFirst (s.removeLast  ());
    }
    s = h;
    System.out.println (s);
}
```

und erklären Sie, was er tut.

- in Abhängigkeit von `s.size()` (muß gerade Zahl sein): nach wieviel Schritten hat die Liste den gleichen Inhalt wie anfangs?

Binäre Bäume in Java

binäre Bäume mit Schlüsseltyp-Parameter `<A>`

```
class Knoten<A> {
    A         key;
    Knoten<A> left;
    Knoten<A> right;
}
```

Bemerkungen:

- Deklarationen mit Typschablonen ab Java 1.5 (wird aber auch Zeit!)
- Bäume bereits im Java-Collections-Framework implementiert

Suchbäume

Ein Suchbaum ist ein binärer Baum, bei dem *für jeden inneren Knoten* gilt:

- jeder Schlüssel im Teilbaum `t.links` ist kleiner als `t.key`
- und `t.key` ist kleiner als jeder Schlüssel im Teilbaum `t.rechts`

D. h. preorder-Reihenfolge ist eine monoton steigende Liste.
Suchbäume benutzt man, um Schlüssel schnell wiederzufinden.

Suchen

gesuchten Schlüssel mit Schlüssel der Wurzel vergleichen, nach links oder rechts absteigen und weitersuchen.

```
search (Knoten t, Key k) {
  if ( t.key == k ) { return t; }
  else {
    if t ist kein Blatt {
      if ( k < t.key ) {
        return search (t.left, k);
      } else {
        return search (t.right, k);
      } } } }
```

Einfügen

```
insert (Knoten t, Key k) {
  if t ist kein Blatt {
    if ( k < t.key ) {
      insert (t.left, k);
    } else {
      insert (t.right, k);
    }
  } else {
    if ( k < t.key ) {
      t.left = new Blatt (k);
    } else {
      t.right = new Blatt (k);
    }
```

```
} }
```

Löschen?

Blatt löschen ist einfach.
Wie löscht man einen inneren Knoten?

Balancierte Bäume: (2,3)-Bäume

(2,3)-Baum:

- alle *Blätter* sind gleich tief

  und haben keine Schlüssel
- jeder *innere Knoten* ist
  - ein 2-Knoten: 2 Kinder $[l, r]$, 1 Schlüssel x
    jeder Schlüssel in $l <x<$ jeder Schlüssel in r
  - ein 3-Knoten: 3 Kinder $[l, m, r]$, 2 Schlüssel $[x, y]$
    jeder Schlüssel in $l < x <$ jeder Schlüssel in $m < y <$ jeder Schlüssel in r

Diese Eigeschaften (Invarianten) müssen bei allen Operationen erhalten bleiben.

Eigenschaften von (2,3)-Bäumen

Ein (2,3)-Baum der Höhe n
hat wenigstens $1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$ Knoten
und höchstens $1 + 3 + 3^2 + \ldots + 3^n = (3^{n+1} - 1)/2$ Knoten.
Aufgabe: wieviele Knoten hat ein (2,3)-Baum der Höhe 10 wenigstens? höchstens?
Aufgabe: welche Höhe hat ein (2,3)-Baum mit 10^6 Knoten wenigsten? höchstens?

Einfügen in (2,3)-Bäume

(Suchen funktioniert ähnlich wie bei binären Suchbäumen.)

- suchen den Knoten (direkt über Blättern), der neuen Schlüssel aufnehmen sollte.
- aus 2-Knoten wird 3-Knoten (OK),
- aus 3-Knoten wird 4-Knoten (verboten).
- Lösung:
  - diesen 4-Knoten in zwei 2-Knoten (und einen Schlüssel) zerlegen,
  - diesen Schlüssel in Vorgänger einfügen

(a, b)-Bäume

Das funktioniert statt $(2, 3)$ genauso für Zahlen (a, b) mit $a \leq (b + 1)/2$. (Warum diese Einschränkung?)
Für große Datenbanken benutzt man z. B. $a = 10, b = 100$.

Der abstrakte Datentyp *Menge*

Abstrakter Datentyp: Schnittstelle (ohne Implementierung), beschreibt Operationen und zugesicherte Eigenschaften

```
interface Set extends Collection {
    boolean isEmpty ();
    void add (Object o);
    void remove (Object o);
    boolean contains (Object o);
}
```

Eigenschaften: z. B.

```
nach s.add (o)  gilt  s.isEmpty() == false.

wenn  s.add (o),
dann danach  s.contains(o) == true
```

konkreter Datentyp (Klasse) beschreibt Implementierung:

```
class TreeSet implements Set { ... }
```

Abstrakte und konkrete Datentypen im Java Collections Framework:
alt: `http://java.sun.com/j2se/1.4.2/docs/guide/collections/index.html`,
neu: `http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html`

Hash-Tabellen

bieten andere Implementierung von Mengen (oft die schnellste)

```
class HashSet implements Set { ... }
```

benutze schnelle Hashfunktion $h : \Sigma^* \to \{0 \dots m - 1\}$
Idee: Deklariere Array $t[\dots]$, speichere x in $t[h(x)]$.
Parameter (Tabellengröße, Hashfunktion) geeignet wählen, dann "praktisch konstante" Laufzeit für alle Operationen.

Hash-Kollision: $x \neq y$ und $h(x) = h(y)$.
x ist schon in $t[h(x)]$, wo soll y hin?

Kollisionen behandeln:

- außerhalb der Tabelle: $t[i] =$ *Liste aller* x mit $h(x) = i$.
  Nachteil: Extraplatz für Listenzeiger
- innerhalb der Tabelle:
  - speichere y in $t[h(y) + 1]$ oder$t[h(y) + 2]$ oder …
    Nachteil: Tabelle „verklebt“
    (belegte Blöcke erzeugen weitere Kollisionen)
  - doppeltes Hashing: benutze h_1, h_2 und benutze Indizes $h_1(y), h_1(y) + h_2(y), h_1(y) + 2h_2(y), \dots$.
    Vorteil: kein Verkleben (Schrittweiten sind verschieden!)

was ist mit Löschen?

Re-Hashing

- Tabelle zu klein $\to$ zu voll $\to$ viele Kollisionen $\to$ langsam.
- Tabelle zu groß: Platz verschenkt.

Lösung: falls Tabelle gewissen Füllstand erreicht, dann zu neuer, größerer Tabelle wechseln (= re-Hashing).
am einfachsten: Tabellengröße ist immer Potenz von 2; dann: vergrößern = verdoppeln.
Beim re-Hashing müssen alle Einträge betrachtet werden, das findet aber nur selten statt, so daß die amortisierte Laufzeit trotzdem konstant ist

Software-Technik

Ergonomische Software

Ziel: angenehm für Benutzer *und* Programmierer
erfordert fachmännisches Vorgehen auf verschiedenen Ebenen:

- Gesamt-Projekt
- Implementierung im Großen
- Programmieren im Kleinen

Software-Projekt-Manangement

Arbeitsschritte (klassische: nacheinander)

- Analyse
- Spezifikation
- Entwurf
- Implementierung
- Test
- Betrieb

Software-Projekt-Manangement (II)

beachte auch (in jedem Schritt):

- Qualitätssicherung
- Kontakt zum Kunden

„modern“: extreme programming, rapid prototyping, refactoring

Richtlinien zur Algorithmenkonstruktion

(vgl. Horn/Kerner/Forbrig S. 213 f.)

- Hierarchische Struktur:
  - Algorithmus ist ein Baum,
  - jeder Teilbaum löst ein Teilproblem,
  - in jedem inneren Knoten steht die Spezifikation,
  - seine Kinder sind die Teilschritte der Lösung.
  - Die Blätter sind elementare Operationen.
- Top-Down-Entwicklung, schrittweise Verfeinerung

Richtlinien (II)

- Blockstruktur:
  - jeder Teilbereich hat genau einen Ein- und einen Ausgang
  - und erfüllt einen Vertrag (wenn Vorbedingung zutrifft, dann ist nach Ausführung die Nachbedingung wahr)
- Lokalität der Daten: Programmbereiche, die bestimmte Daten benutzen, sollten
  - wenig überlappen
  - den Kontrollbereichen entsprechen

Modulare Programmierung

Modul

- funkionell abgeschlossener Teil eines Softwareprojektes,
- der separat entwickelt wird.

modulares Programmieren erlaubt

- Anwenden der Richtlinien (top-down usw.)
- getrenntes, zeitgleiches Entwickeln
- Einsparungen durch Nachnutzen von eigenen und fremden Modulen

Modul-Schnittstellen

Modul besteht aus

- Schnittstelle und
- Implementierung

Die Nutzer eines Modules (= aufrufende Programme) kennen nur die Schnittstelle. (Lokalität der Daten, Datenabstraktion)
Damit kann Implementierung leicht ausgetauscht werden ($\rightarrow$ flexibel, portabel)

Re-Factoring

auch während und nach der Entwicklung immer weiter modularisieren:

- Können weitere Module genutzt werden?
- Kann Funkionalität in neue Module ausgelagert werden?

nach rapid prototyping „mit der Axt reingehen“, vornehme Bezeichnung: *re-factoring*.
ist geboten bei

- überlangen Modul-Quelltexten (d. h. $>$ 1 Bildschirmseite !)
- Code-Verdopplungen (verschiedene Programmteile mit ähnlicher Funktion)

Module in Java?

Hilfsmittel für modulares Programmieren:

- Klassen,
- Interfaces,
- Packages.

beachte: Objekt-Orientierung $\neq$ Modularisierung,
in Java wurde Objekt/Klassen-Konzept erweitert (Wdhlg: wodurch?), und es muß nun auch Modularisierung ausdrücken.

Klassen als Module

Klasse beschreibt Objekte (Attribute und Methoden). Dient damit zum Strukturieren und Zusammenfassen von Algorithmen.
Wie kann Implementierung versteckt werden?
Attribute und Methoden, die als `private` deklariert sind, können *nur innerhalb der Klasse* benutzt werden.
Damit bilden die *nicht-privaten* Attribute und Methoden die *Schnittstelle* der Klasse.

Regel: grundsätzlich *alle Attribute privat* deklarieren, damit jedes Lesen *und Schreiben* „von außen“ bemerkt wird und korrekt behandelt werden kann.

Interfaces

Ein Interface beschreibt Gemeinsamkeiten von Modul-Schnittstellen.
andere Sprechweise:

- Interface = abstrakter Datentyp = Spezifikation
- Klasse = konkreter Datentyp = Implementierung

Programmierer entscheidet zunächst, welcher abstrakter Datentyp benötigt wird,
und wählt dann einen dazu passenden konkreten Datentyp aus.
Beispiel: ADT Menge, konkret: Bitfolge, Liste, Suchbaum.

Packages

Ein Package ist eine Sammlung von Klassen (und Interfaces).
Beispiel: `java.applet` ist ein Package.

Man bezeichnet Methode m der Klasse C aus Package P durch `P.C.m`
Man kann `P.` weglassen, wenn vorher `import P.C` stand.
`import P.*` macht alle Klassen aus P sichtbar.

Die Schnittstelle eines Packages enthält:
alle als `public` deklarierten Klassen, Attribute und Methoden.

Packages in Archiven

Interesse an Modularisierung und Daten-Abstraktion nicht nur wegen der schönen Entwurfsprinzipien.
Programmtext der Implementierung einer Schnittstelle soll versteckt werden oder ganz entfallen, weil er

- zu groß ist
- noch verarbeitet (kompiliert) werden muß
- geheim bleiben soll

Zur korrekten Benutzung eines Modules ist es ausreichend und effizient, wenn die Implementierung kompiliert vorliegt.
Kompilierte Klassen eines Packages werden in Package-Archive (`P.jar`) komprimiert gespeichert.

„Freie“ Software

Der „naive “ unternehmerische Ansatz ist, ein Softwareprojekt zu entwickeln und dann das kompilierte Produkt zu verkaufen oder zu vermieten.
Andererseits sind Programme Texte, also Gedanken, und diese sind bekanntlich frei.
Die Idee der Freien Software ist es, Software (Quelltexte) grundsätzlich zu veröffentlichen, weil so der größte Nutzen für die Allgemeinheit entsteht.

„Freie“ Software

Warum sollte das ein Entwickler/eine Firma tun? Wovon soll er dann leben? Vom Verkauf von Dienstleistungen (Installation, Wartung, Schulung).
Damit leben die späteren Dienstleister auf Kosten der früheren Programmierer? Nein, sie schreiben selbst freie Software.

„Freie“ Software (II)

Frei sein heißt für Software:

- Programmierer bleibt Autor und Urheber
- Quelltext ist frei verfügbar, jeder darf ihn nutzen (d. h. lesen, kompilieren, ausführen)
- und auch verändern und erweitern
- aber alle Änderungen *müssen frei bleiben*.

Siehe auch Free Software Foundation `http://fsf.org/`, GAOS e.V. Leipzig `http://goas.org/`.

„Freie“ Software (II)

Bekannte und berühmte freie Software-Projekte sind

- Emacs (Editor)
- GCC (Compiler), ab 197?
- GNU (= GNU’s Not Unix): Dienst- und Anwendungsprogramme für ein UNIX-ähnliches Betriebssystem, ab 198?
- Linux, ein Betriebssystem-Kern, ab ca. 1990

und vieles andere mehr: (TEX), KDE, Mozilla, gnugo, …
Free Software Directory:
`http://www.gnu.org/directory/`,

Freie Software als Wirtschaftsfaktor

für den Anwender ist es natürlich billiger …
allerdings entstehen Dienstleistungskosten
freie Software-Entwicklung ist flexibler (anpassungsfähiger, schneller)
nutzt z. B. schnelle Hardware viel besser aus als veraltete Systeme
auch große Firmen wollen da mitspielen (und sich street credibility kaufen)
IBM unterstützt Linux, Sun gibt Java-Technologie frei

Freie Spezifikationen

Zur Informatik-Anwendung gehören nicht nur Software, sondern auch Hardware und Protokolle.
Wenn deren Spezifikationen aber nicht frei sind, kann dafür niemand Software (Treiber) schreiben,
und so bleibt der Hersteller nach einer Weile auf seiner Hardware sitzen, weil sie nur (wenn überhaupt) mit anderen eigenen Produkten kompatibel ist.
Davon hängt also viel Geld ab! Die Erkenntnis, daß freie Spezifikationen (Standards) der Gesamtwirtschaft mehr nutzen (als sie Einzelnen schaden) hat sich im wesentlichen durchgesetzt.
Bsp: Internet, ISO 9660 (CD-ROM, DVD), IEEE 1394

Wissenschaft, Freiheit, Sicherheit

Jede Wissenschaft lebt vom Publizieren von Ideen (und nicht vom Geheimhalten).
Nur dadurch können diese geprüft, implementiert und entwickelt werden.
Beispiel: Kryptographie, Systemsicherheit.
Sollte man „kritische" Algorithmen lieber doch geheimhalten?
Ganz falsch!
Sicher sind nur Verfahren, deren Sicherheit wissenschaftlich bewiesen wurde. Gerade dazu müssen sie veröffentlich werden.
Zahlentheorie, Komplexitätstheorie usw.

Ales frei = alles gut?

wie Prof. J. Winkler `http://psc.informatik.uni-jena.de/personen/perso.htm`, der bei Siemens die (natürlich unfreien) Compiler für CHILL und Ada gebaut hat, an dieser Stelle zu sagen pflegte:

There is no such thing as a free lunch.

RTFC

Der Bildungs- und Unterhaltungswert freier Quelltexte ist jedenfalls unbestritten.
Von UCB (Univ. Calif. Berkeley) wird berichtet, daß dort (in den goldenen 70er Jahren) überhaupt kein Programm installiert werden durfte, *ohne gleichzeitig* den kompletten Quelltext im gleichen Directory abzulegen.

In diesem Sinne … *RTFC = read the fXXXing code*!

Zusammenfassung

- Informatik und Algorithmen
  - Geschichte der Informatik
  - Sortier-Algorithmen (durch lineares Einfügen, durch binäres Einfügen)
  - Komplexität von Algorithmen
- Grundlagen der Programmierung
  - Anweisungen/Kontrollstrukturen
  - Ausdrücke (Ausdrucksbäume)
  - Daten (einfache, zusammengesetzte)
  - Typen

Zusammenfassung (II)

- Objektorientiertes Programmieren
  - Objekte, Klassen, Interfaces
  - Methoden, Attribute
  - Vererben, Überladen, Überschreiben
  - GUI: Layout-Management, Ereignis-Behandlung
- Datenstrukturen
  - Listen, Stacks, Queues
  - Bäume, Durchquerungen, Balance

- Softwaretechnik
  - Entwurfsregeln
  - Module, Schnittstellen
  - abstrakte und konkrete Datentypen

Autotool – Highscore – Auswertung

```
Top Ten
----------------
100 :  MNr 36916 Dirk Eidam

 93 :  MNr 36943 Daniel Graumüller

 70 :  MNr 36919 Georg Fürst
 69 :  MNr 37162 Kerstin Werner

 54 :  ...
```