Programmierung

Prof. Dr. Jürgen Giesl

Vorlesungsmitschrift

WS 2011/2012

Dies ist eine einfache Mitschrift, die nachträglich nicht wirklich aufbereitet wurde. Im Wesentlichen ist sie eine Aneinanderreihung der einzelnen Vorlesungen.

Inhalte der Folien habe ich im Allgemeinen weggelassen, ebenso das zigste Beispiel zu bestimmten Themen.

Für bessere Gliederungen und Aufmachungen verweise ich auf frühere Mitschriften oder – besser – auf die Euch angebotene Vorlesung... :-)

Natürlich kann ich weder eine vollständige noch eine fehlerfreie Mitschrift gewährleisten.

Alexander aus der Fünten
E-Mail: alexander110@web.de
7. Februar 2012

- 11.10.2011 -

Algorithmus: Idee der Lösungsbeschreibung
Programm: Konkrete Formulierung des Algorithmus in einer Programmiersprache
Software: Programm und Dokumentation
Hardware: Apparatur des Rechensystems

Eigenschaften eines Algorithmus:
- endlicher Text
- effektiv (durch eine Maschine ausführbar)
- besteht aus vielen Elementaranweisungen und es liegt eindeutig fest, welche als nächste auszuführen ist (Determinismus)
- ermöglicht Ein- und Ausgabe, wobei jeder Eingabe genau eine Ausgabe zugeordnet wird (Determiniertheit)
- terminiert (d.h. er bricht nach endlich vielen Schritten ab)
- die letzten drei Eigenschaften werden nicht immer verlangt.

Kennzeichen eines guten Algorithmus:
- Allgemeinheit (sollte möglichst eine ganze Klasse von Problemen lösen)
- Veränderbarkeit (leicht an modifizierte Aufgabenstellung anpassbar)
- Effizienz (sollte möglichst wenig Rechenschritte verbrauchen)
- Robustheit (sollte sich auch bei unzulässigen Eingaben wohl definiert verhalten)

Bsp für Alphabete:
- lateinische Alphabet (a, b, c, …, z)
- $A_1 = \{0, 1\}$
- $A_2 = \{ (,), +, -, *, /, a\}$

Bsp für Worte:
- $A_1^* = \{0, 1, 00, 01, 10100, 001, \ldots\}$
- $A_2^* = \{(), (+-a), (a*(a+a)), \ldots\}$

Bsp für Sprachen:
- Sprache der Binärzahlen ohne führende Nullen:
  - $L = \{1, 10100, 111, 111001, \ldots\} \leq A_1^*$
- Sprache der Korrekt geklammerten Ausdrücke:
  - $EXPR = \{ (a), (((a))), (a*(a+a)), \ldots\} \leq A_2^*$

Hier nur: Grammatiken
(Automaten im 2. Semester)

Bsp. für Ableitung von Sätzen aus dieser Grammatik:
- Satz
  - Subjekt Prädikat Objekt
  - Art Attr Subst Präd Obj
  - der Attr Subst Präd Obj
  - der bissige Attr Subst Präd Obj
  - der bissige kleine Subst Präd Obj
  - der bissige kleine Hund jagt die große Katze

- 12.10.2011 -

Grammatiken, Syntax und Struktur:

- Definition: Grammatik G ist ein 4-Tupel (N, T, P, S)
  - N: endliche Menge von Nichtterminalsymbolen
    - Symbole für syntaktische Abstraktion
    - Bsp: Satz, Subjekt, Artikel, …
    - Kommen nicht in Wörter der Sprache vor
    - werden durch Anwendung der Regeln solange ersetzt, bis nur noch Terminalsymbole übrig sind.
  - T: endlich Menge von Terminalsymbolen, disjunkt mit N ($N \wedge T = \emptyset$)
    - Zeichen des Alphabets, aus denen Wörter der Sprache bestehen
    - Bsp: der, die, das, Hund, Katze, jagt, …
  - P: endliche Menge von Produktionsregeln x → y
    - bedeutet, dass das Teilwort x durch das Teilwort y ersetzt werden kann
    - $x \in V^*NV^*$, $y \in V^*$ $V = N \vee T$
      - d.h x muss mindestens ein Nichtterminalsymbol enthalten
    - Bsp: Prädikat → jagt
      $\in N$ $\in T$

      Anwendung der Regel ergibt:
      der kleine Hund Prädikat Objekt =>
      $\in T$ | $\in N$
      der kleine Hund jagt Objekt
  - S: Startsymbol
    - $S \in N$ ist spezielles Nichtterminalsymbol, aus dem alle Wörter der Sprache erzeugt werden können
    - Bsp: Startsymbol ist Satz

Ableitung:

- => ist Relation auf V*
- Für μ, v, y $\in V^*$ und x ∈ V*NV* gilt
  - μxv => μyv
  - falls x → y ∈ P
- Die von Grammatik G erzeugte Sprache ist:
  - L (G) = {w | w ∈ T*, S => …. => w}
  - d.h. L(G) enthält alle Wörter, die durch wiederholte Ableitung aus dem Startsymbol S entstehen und die nur aus Terminalsymbolen bestehen
- Zwei Grammatiken G_1, G_2 sind äquivalent wenn sie die gleiche Sprache erzeugen ($L(G_1) = L(G_2)$).
- Kontextfreie Grammatiken sind wichtigste Klasse von Grammatik in der Informatik.
- Beispiel-Grammatik:
  - A => aBbC
    - dc
    - aaBbbc
      - adbc
      - a^3Bb^3c
        - a^2db^2c
        - a^4Bb^4c
          - …
  - L (G) = $\{a^ndb^nc \mid n \geq 0\}$

- Existiert eine kontextfreie Grammatik G', die äquivalent zu G ist? (L(G') = L(G))
  - Idee: baue die Wörter von hinten auf
    - $A \rightarrow Bc$ (B muss die Wörter $a^n db^n$)
    - $B \rightarrow aBb$
    - $B \rightarrow d$
- EBNF
  - statt $B \rightarrow y$ jetzt A = y
  - statt $A \rightarrow y_1$, $A \rightarrow y_2$, $A \rightarrow y_n$ jetzt A = (y_1 | y_2 | y_n)
  - statt $A \rightarrow xz$, $A \rightarrow xyz$ jetzt A = x[y]z
  - statt $A \rightarrow xA$, $A \rightarrow y$ jetzt A = {x}y
  - statt aBb jetzt „a“ B „b“
- Hund-Katze Grammatik in EBNF:
  - Satz = Subjekt Prädikat Objekt
  - Subjekt = Artikel Attribut Substantiv
  - Artikel = [(„der“ | „die“ | „das“)]
  - Attribut = {Adjektiv}
  - Adjektive = („Kleine“ | „bissige“ | „große“)
  - Substantiv = („Hund“ | „Katze“)
  - Prädikat = „jagt“
  - Objekt = Artikel Attribut Substantiv

<u>Grundlagen Programmierung:</u>

- Java:
  - objektorientiert
  - Syntax sehr ähnlich zu C, C++
  - große Sammlung standardisierter Programmbibliotheken
- Grundelemente: einfache Anweisungen und primitive Datentypen
  - alle Programmteile werden jeweils einer Klasse zugeordnet
  - jedes Programm enthält mindestens eine Klassendefinition, die mindestens ein Teilprogramm (<u>Methoden</u>) mit ausführbaren Anweisungen einhält
  - Hauptprogramm = erste auszuführende Methode (Main)
  - volle Java – Regeln: Java Language Specification (Internet/ Buch)
  - Name der Argumente einer Methode heißen „formale Parameter“
  - Methodenkopf von main ist in Java so vorgeschrieben
  - Variablen Deklaration: int x;
    - vereinbar, dass Speicherbereich mit symbolischen Namen x reserviert werden soll. Typ (int) gibt an, wie groß dieser Speicherplatz ist.
    - „=“ bedeutet Zuweisung

- 18.10.2011 -

Konstanten:
- dürfen nur einmal gesetzt werden
- auch möglich: final int x, y; … x=10; … y=989; … (x=11; → nicht mehr erlaubt!)

Eingabe:
- System.console() berechnet „Konsolenobjekt“ der JVM.
- Methode readline() liest eine Zeile von der Tastatur und gibt sie als Ergebnis (vom Typ String) zurück.
- 2 Arten von Methodenaufrufen:
  - als Anweisung (Ergebnistyp der Methode ist *void*)
  - als Ausdruck (Ergebnistyp der Methode ist nicht *void*)
- In Java ist die Trennung nicht streng: Es existieren auch Anweisungsmethoden mit Ergebnistyp nicht gleich *void*.
- parseInt wandelt Ziffer-String in int-Zahl um (parseInt(„123“) = 123).

Polymorphismus (griech.: Vielgestaltigkeit):
- Eine Funktion/Methode kann auf Argumente verschiedenen Typs angewandt werden:
  - z.B. + auf Zahlen: Addition (3+4 = 7)
  - + auf Strings: Verkettung ("hal"+"lo" = "hallo")
  - wenn 1 Argument String ist und 2. Argument nicht String ist: Verkettung
    - "hal"+123 = "hal123" (123 wird zu "123")
- Weitere polymorphe Methoden:
  - System.out.print("hallo") gibt hallo aus
  - System.out.print(123) gibt 123 aus (123 → "123")
  - Überführung von Objekten in Strings:
    - vordefiniert für primitive Datentypen wie int
    - durch Methode toString bei komplexeren Datentypen
      - toString ist bei vielen existierenden Datentypen vordefiniert, bei benutzerdefinierten muss die Methode selbst geschrieben werden.

Methodenaufruf:
- Name(Arg_1, …, Arg_n)
- API: beschreibt die bereits vordefinierten Klassen und ihre Methoden.
  - Klassen sind in Paketen (packages) organisiert. Das am häufigsten benutzte Paket ist *java.lang* (die Klassen aus diesem Paket werden automatisch importiert).

Ausdrücke:
- Bedingter Ausdruck: A1 ? A2 : A3
  - wertet zunächst A1 aus
  - falls A1 zu *true* auswertet, dann wertet der Gesamtausdruck zu A2 aus
  - falls A1 zu *false* auswertet, dann wertet der Gesamtausdruck zu A3 aus

Beispiele für:
- Grundwerte: 1, 2, true, false, …
- Name: x, betrag, …
- Methodenaufruf: Math.max(x,y), … (Ergebnistyp ist nicht gleich void!)
- Infix-Operatoren: +, -, *, /, … (z.B. x+2)

  - z.B. "hallo" + (2+3) = "hallo5"
  - aber: "hallo" + 2 + 3 = "hallo23"
- Präfix-Operatoren: +, - (z.B. -(2+3) = -5)
- Postfix-Operatoren: ++ (z.B. wert++) → kommt erst später!

Wichtigste Grundelemente beim Programmieren:
- Datenstrukturen (Abschnitt 2)
- Algorithmen (Abschnitt 3)

Datentypen:
- Jede Variable muss vor Benutzung deklariert werden. Bei Deklarationen wird Speicherplatz reserviert. Typ der Variable bestimmt, wie groß der Speicherplatz ist.
- z.B.: int x; (Größe: 32 Bit)
- Primitive Datentypen sind atomar (in Java vordefiniert).

Ganze Zahlen:
- 4 Datentypen:
  - byte (8 Bit)
  - short (16 Bit)
  - int (32 Bit)
  - long (64 Bit)
- mit m Bit kann man 2^m verschiedene Zahlen darstellen.
- Negative Zahlen werden durch das Zweierkomplement dargestellt.
- Bei m Bits kann man also die Zahlen von -2^(m-1) bis 2^(m-1)-1 darstellen.
  - bei byte (8 Bit) kann man die Zahlen von -2^7 = -128 bis 2^7-1 = 127 darstellen.
- Was passiert bei Überlauf/Unterlauf?
  - byte x = -128;
  - x = x – 1; // x hat nun den Wert 127 (größtmögliche Zahl)
- Achtung: int-Werte (und andere) können überlaufen/unterlaufen!
  - Sollte in Programmen vermieden werden (beliebte Fehlerquelle!)
- vordefinierte Operationen auf ganze Zahlen: +, -, *, / (Division), % (Rest)
  - 7 / 3 = 2
  - 7 % 3 = 1

Gleitkommazahlen:
- Bsp.: 2.3, -2.3, 0.5 = .5, 2.3e23 (=2,3 * 10^23), 2.3e-23, …
- 2 Datentypen:
  - float (32 Bit)
  - double (64 Bit)
  - Bsp.: 2.3f (float), 2.3d (double), 2.3 (normal: double)
- vordefinierte Operationen: +, -, *, /

Wahrheitswerte:
- Datentyp: boolean
- hat nur die Elemente true, false
- vordefinierte Operationen: && (und), || (oder), ! (nicht)
  - && und || arbeiten von links nach rechts. Wenn das Ergebnis nach Auswertung des linken Arguments feststeht, wird das rechte Argument nicht ausgewertet.
- vordefinierte Vergleichsoperationen: >, <, >=, <=, ==, !=

- 19.10.2011 -

Zeichen:

- Datentyp: char
  - Bsp.: 'a', '1','&', '\n', … (alle 2^16 Zeichen des Unicodes)
- vordefinierte Operationen: ==, !=, <, >, <=, >=
  - Vergleiche anhand der Zahl im Unicode
  - z.B.: 'a': 97, 'b': 98 → 'a' < 'b' (kann für alphabetische Vergleiche benutzt werden)

Strings (kein primitiver Datentyp!):

- Datentyp: String
- vordefinierte Operationen: + (Verkettung)
- Schreibweise: „hallo“
- Umwandlung von Elementen anderer Typen in Strings: z.B. System.out.print(345)
- bzw. Methode toString für komplexere Datentypen

Typen in Java:

- Operationen haben festgelegten Typ für Argumente
  - Bsp.: 2*3 = 6
  - 2*true (nicht erlaubt!)
  - 0.5*6.6 = 3.3
- Elemente eines Typs werden automatisch in Elemente eines „größeren“ Typs konvertiert, wenn dies für Operationen nötig ist.
  - 0.5*4 → 0.5*4.0 = 2.0 (double)
  - int x = 'a' + 1 // = 97 + 1 = 98
- Implizite Typkonversion: automatisch → vom „kleinen“ zum „großen“ Typ
- Explizite Typkonversion: vom Programmierer erzwungen → kann auch vom „großen“ zum „kleinen“ Typ
  - (int)2.6 = 2
  - x = 82.2f, y = 8.5f
    int n = (int)(x/y); // n = 9
  - char a = (char)(int)x; // a = R (82)
  - char b = (char)(a + 1); // b = S (83)
- Typkonversionen sind nicht beliebig möglich
- Java ist streng getypt: Eine Operation, die Argumente vom Typ u erwartet, darf auf Argumente vom Typ t angewandt werden, falls es automatische Konversion von Typ t nach Typ u gibt (engl.: „Type Cast“).

Zustand eines Programms:

- Werte der Variablen / Daten im Speicher (Datenfluss)
- Befehlszähler (gibt an, welche Anweisung als nächstes auszuführen ist), (Kontrollfluss)

Anweisung:

- Übergang von Zustand 1 zu Zustand 2
- Jede Anweisung und Variablendeklaration endet mit ; oder }
- Für Zuweisungen existieren Kurzschreibweisen:
  - statt x = x+1; kann man schreiben: x+=1; (ebenso mit anderen Operatoren)
  - statt x = x+1; kann man schreiben: x++; (ebenso x--;)

Anweisung: Befehl im Programm (z.B.: x = 2+3)
Ausdruck: Term, der nach Auswertung einen bestimmten Wert liefert (2+3)

In Java gibt es auch Anweisungen, die zusätzlich einen Wert zurückliefern → Manche Anweisungen können auch als Ausdrücke benutzt werden.

- y = x = 2+3;
- Die Anweisung x = 2+3; liefert zusätzlich den Wert 2+3 zurück
- → Schlechter Programmierstil!
- y = x = Ausdruck ist „erlaubt“, aber andere Verwendungen von Anweisungen als Ausdrücke sind im Rahmen der Vorlesung „verboten“.
- x++; liefert gleichzeitig den Wert x zurück
  ++x; liefert gleichzeitig den Wert x+1 zurück
- → Schlechter Programmierstil!
- → Verwende x++ und x-- nur als Anweisung, nicht als Ausdruck!

<u>If-Anweisung:</u>

- if(Ausdruck) Aw1 else Aw2 (else-Teil kann fehlen)
- if-Anweisungen kann man schachteln
- wenn sich „else“ auf das äußere if beziehen soll, muss man Klammern setzen!

<u>Switch-Anweisung:</u>

- Fallunterscheidung mit beliebig vielen Fällen
- switch(Ausdruck) → Ausdruck liefert int oder char
- {case Ausdruck1: Aw1 … break; case...}
- default: AwDef → wird ausgeführt, wenn keiner der vorherigen Fälle zutraf
- default kann fehlen (ggf. wird nichts ausgeführt)
- case-Anweisungen sind *labels*, an die gesprungen wird
- bei break wird ans Ende der switch-Anweisung gesprungen
- → switch-Anweisungen ohne break sind schlechter Programmierstil!

<u>While-Schleife:</u>

- while(Ausdruck) Aw
- Schleifenkopf, Schleifenrumpf
- ähnlich zu if

- 21.10.2011 -

while-Schleife:

- while(Ausdruck) Aw
- Schleifenkopf, Schleifenrumpf
- ähnlich zu if
- überprüft erst Schleifenbedingung und führt dann ggf. den Rumpf aus

Achtung: Schleifen sollten terminieren!

do-Schleife:

- do Aw while(Ausdruck)
- Rumpf wird mindestens einmal ausgeführt

for-Schleife:

- for(Initialisierung; Bedingung; Fortschaltung) Aw
- Bsp.: for(int i=0; i<10; i++) Aw → läuft die Schleife 10x durch
  - i wird in der Schleife initialisiert und ist nur in der Schleife verwendbar
- int i;
  for(i=0;...;...) → ok
- int i;
  for(int i=0;...;...) → nicht ok!
- for(int i=0, j=0;...;...) → ok
- for(int i=0, int j=0;...;...) → nicht ok!

Sprunganweisungen:

- break, continue
- weitere „unbedingte Sprunganweisungen“: System.exit(n) → beendet das Programm
  - n ist vom Typ int, $n \neq 0$ zeigt Fehler an
- Jede Anweisung kann mit Namen versehen werden (Name: Aw)
  - wird benutzt, um anzugeben, welche Anweisung durch Sprung verlassen werden soll
- break verlässt die momentane Anweisung → springt aus der Schleife heraus (verwenden wir nur bei Schleifen und bei switch)
- continue springt nur ans Ende des Schleifenrumpfs, bricht Schleife aber nicht ab
- Bei geschachtelten Schleifen beziehen sich break und continue nur auf die innere Schleife, mit break Name; bzw. continue Name; wird Bezug auf die mit Name: versehene Schleife genommen
- Sprünge führen zu unverständlichen Programmen und sollten sehr zurückhaltend verwendet werden!

Partielle Korrektheit: (Hoare-Kalkül)

- verwendet folgende Spezifikation: <φ>P<ψ> bedeutet:
  - Wenn vor der Ausführung des Programms die Vorbedingung φ galt und wenn das Programm P terminiert, dann gilt nach der Ausführung von P die Nachbedingung ψ.
  - Bsp.: <true>P<res=n!>
- Hoare-Kalkül: 7 Regeln zur Herleitung solcher Korrektheitsaussagen
- Vorteil:
  - teilweise automatisierbar
  - Verifikation überprüfbar
  - Rahmen/Anleitung zur Verifikation
- Schreibweise der Regeln:

$$\frac{<\varphi 1>P1<\psi 1>...<\varphi n>Pn<\psi n>}{<\varphi>P<\psi>}$$

  - bedeutet: Wenn <φ1>P1<ψ1>...<φn>Pn<ψn> wahr ist, dann ist auch <φ>P<ψ> wahr.
- 1. Regel: Zuweisungsregel
  - <φ[x/t]>x=t;<φ>
  - <5>y>x=5;<x>y>
  - <5=y>x=5;<x=y>

- 25.10.2011 -

Partielle Korrektheit: (Hoare-Kalkül)

- 1. Regel: Zuweisungsregel
  - <φ[x/t]>x=t;<φ>
  - <5>y>x=5;<x>y>
  - <5=y>x=5;<x=y>
  - φ[x/5] φ
  - ersetze alle Vorkommen von x durch 5
  - <5=5>x=5;<x=5>
  - Schreibweise: <Zusicherung1>Anweisung<Zusicherung2>
  - bedeutet: Schritt von Zusicherung1 zu Zusicherung2 muss mit einer Hoare-Kalkül-Regel passiert sein.
- Konsequenzregel 1
  - <5=5>x=5;<x=5> true => 5=5
  - φ ψ α φ
  - <true>x=5;<x=5>
  - α ψ
  - Schreibweise: <Zusicherung1><Zusicherung2>
  - wenn 2 Zusicherungen direkt untereinander stehen, dann folgt die untere aus der oberen, d.h. Zusicherung1 => Zusicherung2
  - Bsp.: <x>1>x=x-1;<?>
  - mit Zuweisungsregel: <φ[x/x-1]>x=x-1;<φ>
  - mit Konsequenzregel 1: x>1 => φ[x/x-1] → x-1>0
  - Lösung: <x>1> <x-1>0> x=x-1; <x>0>
- Konsequenzregel 2
  - Bsp.: <true>x=5;<x=5> x=5 => x>=5
  - φ ψ ψ β
  - <true> x=5; <x>=5>
- Sequenzregel
  - Bsp.: <true> res=5; res=x*x+6; <res=31>
  - Spezifikation → zu zeigen ist, dass das Programm diese Spezifikation erfüllt
  - Lösung: <true> <5=5> x=5; <x=5> <x*x+6=31> res=x*x+6; <res=31>
  - denn: <true> x=5; <x*x+6=31> <x*x+6=31> res=... <res=31>

  <true> x=5; res=..., <res=31>

- Bedingungsregel 1
  - Bsp.:0
  - Spezifikation: <true> res=y; if (x>y) res=x; <res=max(x,y)>
  - Lösung: <true> <y=y> res=y; <res=y> if(x>y> { <res=y ^ x>y> <res=max(x,y)> res=x; <res=max(x,y)> }
  - <res=y ^ x>y> res=x; <res=max(x,y)> res=y ^ !x>y => res=max(x,y)

  <res=y> if(x>y) res=x; <res=max(x,y)>
- Bedingungsregel 2
  - Bsp.:
  - <true> if(x<0) { <true ^ x<0> <-x=|x|> res=-x; <res=|x|> }
    else { <true ^ x!<0> <x=|x|> res=x; <res=|x|> } <res=|x|>
  - <true> if(...) … else … <res=|x|>

- Schleifenregel
  - φ ist Schleifeninvariante, d.h.:
    - φ gilt vor Ausführung der Schleife
    - vor jeder Ausführung des Schleifenrumpfs gilt φ ^ B
    - φ gilt nach jeder Ausführung des Schleifenrumpfs
    - φ ^ !B gilt nach Ausführung der Schleife
  - Aufgabe: finde geeignete Schleifeninvariante φ, d.h.:
    - Schleifeninvariante folgt aus der gegebenen Vorbedingung
      - im Bsp.: i=n ^ res=1 => φ (d.h.: φ ist schwach genug)
    - es ist wirklich eine Schleifeninvariante
      - im Bsp.: <φ ^ i>1> res=res*i; i=i-1; <φ>
    - aus Schleifeninvariante folgt die gewünschte Nachbedingung
      - im Bsp.: φ ^ !i>1 => res=n! (d.h.: φ ist stark genug)
  - Vorgehen: Gehe von Vor- oder Nachbedingung aus und modifiziere diese, so dass es eine Schleifeninvariante wird (Nachbedingung führt meist zur Lösung).
    - Tipp: Teste die Schleife mit konkreten Werten, bilde eine Tabelle mit den jeweiligen Werten der Variable in jedem Schleifendurchlauf.
  - Bsp.:

i	**res**	**n**
6	1	6
5	6	6
4	6*5	6
3	6*5*4	6
2	6*5*4*3	6
1	6*5*4*3*2	6

    - Schleifeninvariante φ
    - Lösung: φ ist i! * res = n!
    - <true> < i! * res = n! ^ i>1 > … < i! * res = n! ^ !i>1 > <res=n!>
    - siehe Folie!
- java -ea … führt dazu, dass Assertions bei der Ausführung überprüft werden
- Bisher: nur partielle Korrektheit verifiziert (wenn Programm terminiert, dann ist es korrekt)
- Schleifenvariante V:
  - ist nicht negativ, falls Schleifenrumpf ausgeführt wird
  - wird bei jedem Durchlauf der Schleife verkleinert
  - die Variable m darf in P nicht verändert werden
  - Bsp: Variante ist i
  - zu zeigen:
    - i>1 => i>=0
    - <i=m ^ i>1> res=res*i; <i-1<m> i=i-1; <i<m>
  - Bsp.:
    - <a>=0> <a>=0 ^ a=a ^ b=b> x=a; <x>=0 ^ x=a ^ b=b> res=b; <x>=0 ^ x=a ^ res=b>

- 26.10.2011 -

Schleifenvariante V:

- ist nicht negativ, falls Schleifenrumpf ausgeführt wird
- wird bei jedem Durchlauf der Schleife verkleinert
- die Variable m darf in P nicht verändert werden
- Bsp: Variante ist i
- zu zeigen:
  - i>1 => i>=0
  - <i=m ^ i>1> res=res*i; <i-1<m> i=i-1; <i<m>
- Bsp.:
  - <a>=0> <a>=0 ^ a=a ^ b=b> x=a; <x>=0 ^ x=a ^ b=b> res=b; <x>=0 ^ x=a ^ res=b>
  - Schleifendurchlauf:

a	**b**	**x**	**res**
3	4	3	4
3	4	2	5
3	4	1	6
3	4	0	7

  - Schleifeninvariante: x>=0 ^ res = a+b-x
  - <x>=0 ^ x=a ^ res=b> <x>=0 ^ res=a+b-x> while(x>0) {
  - <x>=0 ^ res=a+b-x ^ x>0> <x-1>=0 ^ res+1=a+b-(x-1)> x=x-1; <x>=0 ^ res+1=a+b-x> res=res+1; <x>=0 ^ res=a+b-x>
  - } <x>=0 ^ res=a+b-x ^ !x>0> <res=a+b>

Terminierung:

- Variante ist x
  - x>0 (Schleifenbedingung)=> x>=0 (Variante)
  - <x=m ^ x>0>
  - <x-1<m>
    - x=x-1;
  - <x<m>
    - res=res+1;
  - <x<m>
- <x>=z>
- <x>=z ^ y=y ^ 0=0>
- z=y;
- <x>=z ^ z=y ^ 0=0>
- res=0;
- <x>=z ^ z=y ^ res=0>
- < x>=z ^ res=z-y>
- while(x>z) {
  - < x>=z ^ res=z-y ^ x>z>
  - < x>=z+1 ^ res+1=z+1-y>
  - z=z+1;
  - < x>=z ^ res+1=z-y>
  - res=res+1;

    - < x>=z ^ res=z-y>
- }
- <x>=z ^ res=z-y ^ !x>z>
- <res=x-y>

x	y	z	res
5	2	2	0
5	2	3	1
5	2	4	2
5	2	5	3

- x>=z (Schleifenbedingung) ^ res=z-y (Variante)
- Terminierung: Variante x-z
    - x>z => x-z >=0
    - <x-z=m ^ x>z>
    - <x-(z+1)<m>
        - z=z+1;
    - <x-z<m>
        - res=res+1;
    - <x-z<m>

Arrays

- für Listen, Tabellen, Vektoren, Matrizen
- int[] folge; → int[] ist der Typ des Arrays
    - folge[2] == 0 → hat den Typ int
- int[][] bestand; → 2-dimensionaler Array
    - bestand[0][1] == 0 → hat den Typ int
    - bestand[0] → hat den Typ int [] → ist ein Array
- Algorithmus, um Bestand der Artikel 0 bis 2 an Ort 3 zu berechnen
    - int summe=0;
    - for (int i=0; i<=2; i++) {
        - summe += bestand[i][3];
    - }
- Eine Array-Variable (wie folge, bestand) fasst Vielzahl von Variablen eines Datentyps zusammen.
- Einzelne Elemente werden durch Index unterschieden (erstes Element hat Index 0).
- Auch Array-Variablen müssen vor Benutzung deklariert werden. Dabei werden 2 Dinge festgelegt:
    - Datentyp der Elemente
    - Dimension des Arrays (= Anzahl der Indizes)
- Bei primitiven Datentypen folgt aus Variablendeklaration, wie viel Speicher für die Variable benötigt wird (z.B. 32 Bit bei int).
- Bei nicht-primitiven Datentypen:
    - In dem Speicherplatz für die Variable steht nicht der Wert der Variable, sondern eine Adresse a. An der Adresse a steht der Wert von x.
    - Erzeugung eines Speicherplatzes für den Wert der Variablen x geschieht mit Befehl new.
        - x = new int[3]; → schreibe in x eine Speicheradresse a, so dass an Speicherstelle a Platz für 3 int-Werte reserviert ist

  - Keller (Stack): Speicherbereich für die Werte der Programmvariablen
  - Halde (Heap): Speicherbereich, auf den Programmvariablen verweisen können
  - Vorteil: Man kann aus der Anfangsadresse des Arrays und der Größe der Einträge sofort die Adresse jedes Array-Elements ausrechnen. => Zugriff auf Array-Elemente ist sehr schnell (und gleich schnell für alle Array-Elemente)
  - Nachteil: Länge des Arrays ist nicht veränderbar. Wenn man nachher doch mehr Elemente im Array speihern möchte, dann braucht man ein neues Array und muss Elemente eines nach dem anderen vom alten ins neue Array kopieren.
  - Ob in einer Variable Werte oder Verweise (Zeiger, Pointer) gespeichert werden, hängt in Java vom Datentyp der Variable ab. In anderen Programmiersprachen kann der Programmierer entscheiden, wann Werte und wann Verweise gespeichert werden. Grund für diese Design-Entscheidung bei Java war Programm-Sicherheit.
- Seiteneffekt:
  - Auswirkung des Schreibzugriffs über die eine Referenzvariable y auf ein Objekt, das auch über eine andere Referenzvariable x erreichbar ist.
- int[] y;
- y = new int[2];
- …
- y = new int[70];

- 02.11.2011 -

Arrays:

- Garbagekollektor
  - wird automatisch aufgerufen (in Java)
  - gibt Speicher im Heap frei, der nicht mehr benötigt wird
- Statt:
  - int[] a = new int[n];
  - a[0] = 8; a[1] = 7; … a[n] = 9;
- ist auch folgende Initialisierung möglich:
  - int[] a = {8, 7, …, 9};
  - hierdurch wird sowohl Größe als auch Inhalt festgelegt
  - ist aber nur zur Initialisierung erlaubt
- Initialisierungs-Schreibweise auch bei mehrdimensionalen Arrays möglich:
  - int[][] b = {{0, a}, {10, 11}, {21, 22}}
  - d.h.: b[0][0] == 0, etc.
  - Anzahl der „Spalten“ muss nicht in allen „Zeilen“ gleich sein:
  - int[][] b = {{0, a}, {10, 11}, {21}}

Palindrom = Wort, das von links wie von rechts gelesen dasselbe ergibt (z.B. Rentner).

Benutzung des Arguments der main-Methode:

- java Palindrom wort1 … wortn
- führt dazu, dass args auf das Array gesetzt wird, dass die Strings wort1 … wortn enthält.
- Im Beispiel wollen wir nur 1 Wort auf Palindrom-Eigenschaft untersuchen, z.B.:
  - java Palindrom rentner
  - → args[0] == „rentner“

Eigenschaften von Objekten:

- args[0].toCharArray()
- String-Objekt . Name der Eigenschaft
  - 2 Arten von Eigenschaften:
    - Attribute (liegen fest)
    - Methoden (kann man berechnen, haben immer (…))
- wort.length
  - liefert die Länge des Arrays zurück
  - int[] a = {4, 5, 6};
  - a.length == 3
  - int[][] b = {{4, 5}, {8, 9, 10}};
  - b.length == 2 (Länge der ersten Dimenstion)

Sort-Programm (s. Folie):

- Idee: Vertausche Einträge, so dass an der linken Stelle das kleinste Element steht. Dann vertausche so, dass an der 2. Stelle das zweitkleinste Element steht, etc. Zum Schluss steht das größte Element an der letzten Stelle.
- Elemente vertauschen: a[i] = a[j]; a[j] = a[i]; → funktioniert nicht!
  - → int z = a[i]; a[i] = a[j]; a[j] = z;

foreach-Schleife:

- for (int x : a) x = Integer.parseInt(...);
- Abkürzung für:
  - for (int i=0; i<a.length; i++) {
    x = a[i];
    x = Integer.parseInt(...); // ändert a überhaupt nicht!
    }

Darstellung von komplexen Objekten:

- Bisher: Darstellung als Arrays
  - Vorteil: Zugriff auf Arrays ist schnell
  - Nachteile:
    - fassen nur gleichartige Einträge zusammen (int[] enthält nur int-Einträge)
    - Größe des Arrays liegt bei seiner Erzeugung fest
- Beispiel: Darstellung von Rechteck-Objekten
  - Rechteck hat folgende Eigenschaften:
    - laenge (double)
    - breite (double)
    - strichstaerke (int)
  - Wenn man für die 3 Eigenschaften 3 Arrays verwendet, sind die Daten eines Objekt auf 3 Arrays verteilt. → Eigenschaften von Objekten sind in verschiedenen Arrays.
  - Kopieren von Objekten geht nur durch einzelne Anweisungen für jede Eigenschaft.
  - Eigenschaften, die man aus anderen Eigenschaften berechnen kann (z.B. flaeche), stehen getrennt von anderen Eigenschaften.
- => objektorientierte Programmierung
  - Jedes Objekt hat Eigenschaften (Attribute und Methoden).
  - Klasse ist Zusammenfassung/ Menge/ Typ von gleichartigen Objekten.
    - Attribute dieser Objekte ^= in der Klasse deklarierten Variablen
    - Methoden dieser Objekte ^= Methoden in der Klasse
  - Erzeugung neuer Objekte:
    - r = new Rechteck(); → erzeugt ein neues Objekt vom Typ Rechteck
    - a = new int[4]; → erzeugt ein neues Objekt vom Typ int[]
  - Zugriff auf Eigenschaften der Objekte:
    - r.laenge = 2.5; r.breite = 2.0; r.strichstaerke = 3;
    - r.laenge = r.laenge + 1;
    - System.out.print(r.flaeche()); // 7.0
  - Methoden eines Objekts können auf Eigenschaften des Objekts zugreifen. „return“ beendet Methode und liefert Ergebnis zurück.
  - Klassen werden in Java für zwei Zwecke benutzt:
    - Unterprogramm (Klassen, die nur aus Methodendeklarationen bestehen)
    - Datentyp (Klassen, die auch Variablendeklarationen besitzen → beschreiben die Attribute von Objekten dieses Typs)
  - Klassen, die nur Variablendeklarationen (und keine Methoden) enthalten, bezeichnet man auch als Records (Verbunde) → gab es in imperativen Programmiersprachen schon lange vor der Objektorientierung.

Objektorientierung:

- Klasse kann sowohl Variablendeklarationen als auch Methodendeklarationen enthalten
- Beziehungen zwischen Klassen (später)

- 09.11.2011 -

Primitive Datentypen: Variablen speichern Werte
Arrays und Klassen-Datentypen: Variablen speichern Referenzen (Verweise)

Unterprogramm (Methode):
- parametrisierter Anweisungsblock mit Namen
- (formale Parameter-Argumente der Methode und Methodenname)
- static bedeutet: gehört zur Klasse und nicht zu einzelnen Objekten der Klasse

Parameter-Übergabe beim Methodenaufruf:
- formaler Parameter der Methode zins: kapital
- aktueller Parameter beim Aufruf von zins: betrag1 + betrag2
- Werte den Ausdruck im aktuellen Parameter aus: 1000 + 570.22 = 1570.22
- Weise den berechneten Wert dem formalen Parameter der Methode zu: kapital = 1570.22
- Formaler Parameter ist eine normale Variable der Methode, die auch im Methodenrumpf verändert werden darf.
- Führe Methodenrumpf aus.
- Wenn return-Anweisung erreicht: Werte den Ausdruck in return-Anweisung aus, beende Methode und liefere Wert dieses Ausdrucks als Ergebnis der Methode zurück.
  - return 1.03 * 1570.22 → gewinn = 1617.3266
- Jede nicht-void Methode muss irgendwann return-Anweisung erreichen.
- Auch bei void-Methode ist return; möglich (beendet die Methode)
- Die beschriebene Art der Parameterübergabe heißt „call-by-value“ (cbv). Grund: Methode wird mit dem Wert des Ausdrucks im aktuellen Parameter aufgerufen.
- Alternative Parameterübergabe-Mechanismen:
  - call-by-reference (cbr)
  - call-by-name (später bei Fkt. Programmiersprachen)

Funktion: nicht-void Methode, z.B. drucke
Prozedur: void Methode (für Ausgabe oder für Seiteneffekte)

Call-by-value:
- aktueller Parameter (im Methodenaufruf) wird ausgewertet
- Wert des aktuellen Parameters wird in formalen Parameter der Methode kopiert
- Änderungen des formalen Parameters der Methode bewirken keine Änderung des aktuellen Parameters
- In Java: Bei primitiven Datentypen findet immer call-by-value Parameterübergabe statt.

Speicherverwaltung bei Methodenaufrufen:
- Methoden = Blöcke mit Namen und formalen Parametern
- Jeder Block entspricht einem Speicherbereich (frame), der auf dem Laufzeitkeller (runtime stack) „oben“ bereitgestellt wird. In diesem Frame werden die Werte der Variablen abgelegt, die in diesem Block deklariert werden.
- Bei Eintritt in den Block wird der neue Speicherbereich „oben“ auf den Keller gelegt und bei Verlassen des Blocks wird der Speicherbereich entfernt. („oben“ im Sinne der Wachstumsrichtung, in den Bildern wächst Speicher nach unten)
- Keller: LIFO (last in, first out) = FILO (first in, last out) → engl. stack
- (Warteschlange: FIFO = LILO → engl. queue)
- Laufzeitkeller-Prinzip: Sowohl bei geschachtelten Blöcken als auch bei Methodenaufrufen

- Die Namen der Variablen in der Methode dürfen mit Namen der Variablen in aufrufenden Methode überlappen. Sichtbar (bei Ausführung des Methodenrumpfs) sind nur die Variablen der Methode.
- Unterschied zu geschachtelten Blöcken: Hier dürfen die neuen lokalen Variablen im inneren Block nicht so heißen wie die Variablen im äußeren Block. Grund: Hier sind im inneren Block auch die Variablen des äußeren Blocks sichtbar.
  - int x; {int y; y = x;} // ist das x aus der äußeren Methode

Call-by-reference:

- Aktueller Parameter ist nicht ein beliebiger Ausdruck, sondern eine Variable.
- Beim Methodenaufruf wird der formale Parameter der Methode ein Verweis auf den aktuellen Parameter.
- Jede Änderung des formalen Parameters in der Methode bewirkt auch eine Änderung des aktuellen Parameters.
- In Java: Bei formalen Parametern von nicht-primitiven Datentypen findet eine Form von call-by-reference statt.
- Wenn der formale Parameter von einem nicht-primitiven Datentyp ist, dann zeigen beim Methodenaufruf sowohl der formale Parameter der Methode als auch der aktuelle Parameter auf das gleiche Objekt.
- Wenn dieses Objekt in der Methode verändert wird, dann hat dies auch Auswirkungen auf die aufrufende Methode (Seiteneffekt).
- Java verwendet eigentlich immer call-by-value. Da Variablen von nicht-primitiven Datentypen Verweise speichern, kann man eine Art call-by-reference simulieren. Aber aktueller Parameter wird nicht überschrieben, wenn formalen Parameter eine neue Adresse zugewiesen wird.
- Generell: Informationen von aufrufender Stelle an die Methode sollten über Parameter weitergegeben werden und nicht über globale Variablen (public static int x;)! Globale Variablen könnten in Methoden gelesen und verändert werden → Informationsfluss undurchschaubar → Schlechter Programmierstil!

Sinn von Prozeduren (void-Methoden):

- Ausgabe (drucke)
- Seiteneffekte auf die Objekte, die an die formalen Parameter übergeben wurden (sortiere)
  - Aktueller Parameter x und formaler Parameter a zeigen auf das gleiche Array-Objekt.
  - Methode sortiere ändert das Objekt. Änderung auch in main sichtbar.
  - → gewünschter Seiteneffekt

- 15.11.2011 -

vararg-Parameter: public static int addiere (int... args) {...}

- Idee: Methode darf mit beliebig vielen Argumenten aufgerufen werden.
- Formaler Parameter ist ein Array
- addiere(2, 3, 4): args zeigt auf ein Array der Länge 3 mit Einträgen 2, 3, 4
- man darf als aktuellen Parameter auch ein Array nehmen
- aber addiere(1, new int[] {2, 3, 4}) nicht möglich!
- auch möglich:
  - f(Rechteck... args) → args ist vom Typ Rechteck[]
  - f(Rechteck[]... args) → args ist vom Typ Rechteck[][]
- Man darf auch normale und vararg-Parameter „mischen“
  - add_mult(2, 3, 4) → y==2 und args ist Array mit 3, 4
- Maximal ein vararg-Parameter pro Methode, muss der letzte Parameter der Methode sein!

static/nicht-static:

- nicht static: Eigenschaft eines Objekts
  - z.B. laenge, breite, strichstaerke (jedes Objekt hat eigene Werte)
  - z.B. flaeche (nicht-statische Methode)
  - Aufruf: r.laenge, r.flaeche()
  - Jedes Objekt kann unterschiedliche laenge oder flaeche haben.
  - Bisher: Klassen, in den alle Eigenschaften nicht-statisch sind → Datenstrukturen (z.B. Rechteck)
  - Wenn die Methode auf nicht-statische Attribute eines Objekts zugreift, dann sollte auch die Methode nicht-statisch sein.
- static: Eigenschaft einer Klasse
  - Bisher: Klassen, in denen alle Eigenschafen statisch sind → Algorithmen
  - Jetzt: Klassen, in denen es sowohl statische als auch nicht-statische Eigenschafen gibt
  - z.B. flaechenberechnung (statisch)
  - Aufruf:
    - Rechteck.flaechenberechnung
    - r.flaechenberechnung → auch erlaubt
    - Rechteck.laenge → nicht erlaubt → r.laenge
  - Was passiert, wenn vor einer Eigenschaft kein Punkt steht?
    - in nicht-statischen Methode:
      - Eigenschaft des aktuellen Objekts, für das die Methode aufgerufen wird
      - Bei r.flaeche() bedeutet laenge r.laenge
      - flaechenberechnung bedeutet r.flaechenberechnung, d.h. Rechteck.flaechenberechnung
    - in statischen Methoden:
      - Eigenschaft der Klasse, in der die Methode deklariert ist
      - flaechenberechnung bedeutet Rechteck.flaechenberechnung
      - Aufruf von nicht-statischen Attributen ist in statischen Methoden nicht erlaubt!
  - Sinn von statischen Methoden: Unterprogramme
  - Sinn von statischen Attributen:
    - Mitprotokollieren, wie oft bestimmte Aktionen ausgeführt werden
    - Konstanten (final static double pi = 3.141;)
    - Datentypen, die nur eine feste, endliche Menge von Elementen haben (Aufzählungstypen, engl. Enumerations)

Aufzählungstypen (enum):

- Enums sind spezielle Klassen, aber mit bestimmten Besonderheiten:
  - man kann keine weiteren Objekte von Enums erzeugen
  - es existieren bestimmte vordefinierte Methoden (s. Folie)
    - Tag.values()[3] ergibt Tag.DO
  - String toString() (existiert in allen Klassen)
    - Tag.DO.toString() = "DO"
    - System.out.print(Tag.DO) → wird mit toString in "DO" gewandelt
- enum kann beliebige weitere statische und nicht-statische (benutzerdefinierte) Methoden und Attribute enthalten
- switch-Anweisung (für endliche Fallunterscheidung, insbesondere bei mehr als 2 Fällen) funktioniert nicht nur bei int und char, sondern auch bei Enums und bei Strings

Sichtbarkeit von Bezeichnern (insbesondere bei gleichen Namen verschiedener Bezeichner)

- Gleiche Namen lassen sich bei großen Programmen und verschiedenen Programmierern nicht vermeiden und kann auch sinnvoll sein, um gleichartigen Objekten den gleichen Namen zu geben.
- Regeln:
  - alle Bezeichner muss man vor Benutzung deklarieren
    - Ausnahme: Klassen und Methoden sind auch vor Deklaration verwendbar
  - Deklaration eines Bezeichners gilt bis zum Ende des Blocks, in dem er deklariert wurde
  - in einem Block müssen Bezeichner verschieden sein
    - Aber: Unterschiedliche Programmelemente dürfen den gleichen Namen haben
    - Unter bestimmten Voraussetzungen darf es auch mehrere Methoden mit gleichem Namen geben (überladene Methoden)
  - gleiche Bezeichner im obersten Rahmen des Laufzeitkellers überdecken Bezeichner in unteren Rahmen
    - Aber: Namensgleiche Bezeichner bei geschachtelten Blöcken verboten
    - Bsp. (Folie):
      - formale Parameter x der Methode main überdeckt statische Variable x der Klasse Gültigkeit
      - formale Parameter x der Methode setzt überdeckt nicht-statische Variable x der Klasse Dreieck
      - in flaeche: x, y, z in der ersten Zeile sind die nicht-statischen Attribute des aktuellen Objekts
      - y in der zweiten Zeile überdeckt die nicht-statische Variable y der Klasse Dreieck

- 16.11.2011 -

Datenabstraktion als Entwurfsprinzip für Klassen und Methoden:

- Lesen und Schreiben von Attributen der Objekte durch nicht-statische Methoden
- Insbesondere: Kein direktes Lesen und Schreiben der Attribute, sondern über Selektor-Methoden
  - Grund: Implementierungsdetails von Objekten (z.B. der Klasse Rechteck) sollten nur in ihrer Klasse sein (z.B. Rechtecke implementiert durch laenge, breite, strichstaerke).
  - Wenn man die Implementierung später ändern will, dann ist die Änderung auf Klasse Rechteck begrenzt und betrifft nicht die anderen Klassen, in denen Rechtecke benutzt werden.
  - Datenkapselung: Implementierungsdetails sind in Klasse Rechteck „gekapselt".
  - Zugriff auf Objekte von außerhalb nur durch bestimmte Schnittstellenmethoden (z.B. setLaenge, getLaenge).
  - Beispiel: Ändere interne Implementierung der Klasse Rechteck, aber stelle sicher, dass Schnittstellenmethoden hinterher dasselbe tun. → Gesamte anderen Klassen brauchen nicht geändert zu werden.

Datenkapselung als Entwurfsprinzip:

- Lege für jede Klasse fest, wie die nach außen sichtbaren Schnittstellenmethoden heißen und was sie tun sollen.
- Danach kann jede Klasse separat implementiert und geändert werden (ggf. von unterschiedlichen Programmierern).
- → Modularität
- Entwirf erst die öffentliche Schnittstelle, dann die Implementierung.

Pakete (engl. Packages):

- Zusammenfassung von Klassen.
- Momentan: alle benutzerdefinierten Klassen liegen im gleichen (anonymen) Paket.

Client/Server-Prinzip:

- Anbieter publiziert Schnittstellenbeschreibung (der öffentlich zugänglichen Methoden und Attribute) → API (application programmer interface) beschreibt abstrakte Datentypen
- Den Kunden interessiert nur die Schnittstelle, nicht die Implementierung davon.
- Schnittstellendokumentation (API) kann automatisch aus dem Programm erstellt werden:
  - javadoc Rechteck.java
  - beschreibt die (public) Eigenschaften der Klasse
  - Analog zur API der vordefinierten Java-Bibliotheken
  - /** … */ Kommentare, die von javadoc in die API übernommen werden.
  - @author, @param, @return → Tags für javadoc, um bestimmte Eigenschaften von Klassen und Methoden zu beschreiben.
  - Bei jeder Methode sollte
    - jeder Eingabeparameter mit @param beschrieben werden
    - die Rückgabe mit @return beschrieben werden

Konstruktoren:

- z.B. new Rechteck ();
- hierbei wird eine Konstruktor-Methode Rechteck () aufgerufen
- Konstruktoren heißen genauso wie ihre Klasse
- Syntax von Konstruktoren wie bei sonstigen Methoden, aber kein Rückgabetyp

- im Konstruktor: laenge beschreibt das Attribut des neu zu erzeugenden Objekts
- Dieses wird als Ergebnis des Konstruktors zurückgeliefert.
- Wenn man gar keinen Konstruktor schreibt, wird eine Standartimplementierung für den parameterlosen Konstruktor erzeugt. Diese setzt die Attribute des neuen Objekts auf
  - die Initialwerte bei der Deklaration der Attribute, sofern vorhanden
  - sonst bestimmten Defaultwert des richtigen Typs
    - null ist Defaultwert für alle nicht-primitiven Typen
- Konstruktoren dürfen Argumente haben (z.B. um Objektattribute auf bestimmte Werte zu setzen)

Überladene Methoden:

- Verschiedene Methoden können den gleichen Namen haben, wenn ihre Parameterlisten verschieden sind, d.h.
  - verschiedene Anzahl von Parametern
  - oder Parameter haben unterschiedliche Typen
- Es reicht nicht, wenn
  - Namen der Parameter verschieden sind
  - oder nur der Rückgabetyp unterschiedlich ist
- Auflösung von Überladung:
  - Es wird immer die speziellste Methode genommen, die von Parameteranzahl und Typen passt.
  - vararg nur dann, wenn nichts anderes passt.
    - vararg-Methoden möglichst nicht überladen!
- Überladung auch bei anderen Methoden:
  - z.B. System.out.print() → für alle möglichen Datentypen (je nach Datentyp wird unterschiedlicher Code ausgeführt)
  - → Polymorphismus (je nach Argumenttyp verhält sich Methode unterschiedlich)

- 22.11.2011 -

Selbstverweis:

- Nicht-statische Methode f, die für ein Objekt v aufgerufen wird: v.f(...)
- Wenn man in f auf das aktuelle Objekt (v) zugreifen will: this
  - f(double laenge, …) {
    laenge ← formaler Parameter (überdeckt das gleichnamige Attribut des aktuellen Objekts)
    this.laenge ← Attribut des aktuellen Objekts
    }
- Aktuelles Objekt einer Methode f:
  - bei v.f(...) ist es das Objekt v (f kein Konstruktor)
  - wenn f ein Konstruktor ist, dann ist aktuelles Objekt das gerade erzeugt Objekt
- Gleicher Name für formale Parameter und Objektattribute kann durchaus sinnvoll sein.

Kopier-Konstruktor:

- weiterer typischer Konstruktor
- null: Zeiger ins Leere (Initialwert von Variablen mit nicht-primitivem Typ)

Hüllklassen und Strings:

- zu jedem primitiven Datentyp existiert eine zugehörige Hüllklasse
- Hüllklasse hat im wesentlichen ein Attribut zum „Einhüllen“ von Werten des primitiven Datentyps.
- Klasse Integer:
  - Wenn x ein Integer-Objekt ist, dann ist x.value der darin eingehüllte int-Wert.
  - Auf x.value kann nicht zugegriffen werden, da value private ist.
  - Integer.MIN_VALUE = -2147483648
  - Integer x = new Integer (15); // x.value = 15
  - Integer y = new Integer ("15"); // y.value = 15
  - int z = Integer.parseInt("15"); // z = 15
  - String s = Integer.toString(15); // s = "15"
  - boolean b = x.equals(y); // b = true
  - boolean c = (x == y); // c = false
  - int i = x.intValue(); // i = 15
- Wenn man oft zwischen primitiven Datentyp und seiner Hüllklasse konvertieren muss, wird es unübersichtlich. → Daher erlaubt es Java, primitiven Datentyp und Hüllklasse „durcheinander“ zu benutzen und es wird automatisch konvertiert.
  - Autoboxing: automatisches Konvertieren von primitivem Datentyp zur Hüllklasse
  - Unboxing: automatisches Konvertieren von Hüllklasse zum primitiven Datentyp
  - Es existiert keine implizite Datentypkonversion von Integer → Double (nur von int → double)!
  - Bei überladenen Methoden wird nur dann Auto-/Unboxing benutzt, wenn keine andere Methode passt (außer bei Varargs → sollten eh nicht überladen werden!).
- String ist ebenfalls vordefiniert:
  - String s = "wort"; (ohne expliziten Konstruktoraufruf)
  - Beispiele von Folie:
    - u == v → false
    - s == u → false
    - s == t → true (Java protokolliert die in Kurzschreibweise erzeugten Strings mit und

verwendet sie wieder)
    - s.equals(u) → true
    - u.equals(v) → true
    - u.charAt(2) → 'r'
    - u.length() → 4
    - u.toCharArray() → liefert 4-elementiges char-Array zurück
- Auf Strings sollte man == eher nicht verwenden.
- Keine Gefahr von Seiteneffekten, denn String-Objekte sind unveränderbar.
    - s = s + "e"; // erzeugt neues String-Objekt, ändert nichts an t

Rekursion:

- Ziel: Löse Berechnungsproblem für Eingabe x.
- Oft: manche Anweisungen mehrfach ausführen
- Bisher: Schleifen (Iteration → mehrmaliges Durchlaufen eines Programmabschnitts)
    - Akkumulator-Variablen (z.B. res)
- Rekursion: Führe das Problem für x zurück auf das gleiche Problem für kleineren Wert als x.
    - Bsp.: fak(x) = x*fak(x-1), falls x>1, sonst 1
    - selbstbezügliche (rekursive) Definition
- Eine rekursive Methode ruft sich im eigenen Methodenaufruf direkt selbst wieder auf. Argumente im rekursiven Aufruf müssen geeignet geändert worden sein, damit die Methode terminiert.
- Klassifikation:
    - lineare / nicht-lineare Rekursion
    - direkte / verschränkte Rekursion
    - Endrekursion
- Lineare Rekursion:
    - Jede Ausführung vom Methodenrumpf führt zu höchstens einem rekursiven Aufruf
    - Bsp.: Fakultät
- Nicht-lineare Rekusion:
    - Bsp.: Fibonacci-Zahlen (bla bla bla... ;-( ), siehe Folie

- 23.11.2011 -

Rekursion:

- Nicht-lineare Rekusion:
  - Bsp.: Fibonacci-Zahlen (bla bla bla... ;-( ), siehe Folie
    - fib(20) = fib(19) + fib(18)
    - = fib(18) + fib(17) + fib(18)
    - = fib (17) + fib(16) + fib(17) + fib(17) + fib(16) …
    - → fib(20-n) wird fib(n+1) mal berechnet → sehr ineffizient (exponentieller Aufwand)
    - d.h.: Zur Berechnung von fib(n) braucht man ca. 2^n Berechnungsschritte.
    - → fib kann man effizienter berechnen
  - Spezialfall der nicht-linearen Rekursion: geschachtelte Rekursion (nested rekursion)
    - public static int f(int x) {
      if (x<1) return 0;
      else return f(f(x-1)); // geschachtelte Rekursion
      }
    - f(0) = 0
    - f(1) = f(f(0)) = 0
    - f(2) = f(f(1)) = 0 …
- Direkte Rekursion:
  - Methode f ruft wieder f auf (Bsp.: fak, fib)
- Verschränkte Rekursion:
  - z.B.: f ruft g auf, g ruft f auf (mutual recursion)
    - even(1) = odd(0) = false
    - even(-2) = odd(-1) = even(0) = true
- Endrekursion: (tail recursion)
  - Spezialfall der direkten Rekursion
  - Rekursive Aufrufe dürfen nur am Ende des Algorithmus auftreten (d.h.: weder in Teilausdrücken, noch vor weiteren Anweisungen).
    - fak ist nicht endrekursiv, aber sqrt ist endrekursiv
  - Da der Algorithmus nach dem rekursiven Aufruf zu Ende ist, muss man sich die Werte der lokalen Variablen nicht „merken“, da sie nach dem rekursiven Aufruf nicht mehr gebraucht werden (die alten Werte müssen nach dem rekursiven Aufruf nicht mehr zur Verfügung stehen).
  - → Endrekursive Methoden kann man direkt in iterative Methoden übersetzen.

Speicherorganisation bei Rekursion:

- Wie bei jedem Methodenaufruf führt der rekursive Aufruf zu einem neuen Rahmen auf Kellerspeicher mit Speicherplatz für alle lokalen Variablen (inkl. res bei nicht-void Methoden).
- → Rahmen bildet Kontext, in dem Anweisungen der Methode ausgeführt werden.
- Bei zu vielen (insbesondere bei unendlich vielen) rekursiven Aufrufen reicht der Platz des Kellerspeichers nicht → stack overflow

Vorteil von Rekursion:

- kann zu sehr kurzen und einfachen Programmen führen
- z.B. Türme von Hanoi

Entwurfstechnik für rekursive Algorithmen:

- Divide and Conquer (Teile und Herrsche):
  - 1) Behandle einfache Fälle.
  - 2) Divide: bei nicht-einfachen Fällen teile Problem in 2 oder mehr Teilprobleme auf.
  - 3) Conquer: Löse Teilprobleme (meist rekursiv).
  - 4) Kombiniere: Setze Teillösungen zu Gesamtlösung zusammen.
- Bsp.: Türme von Hanoi
  - Ziel: Turm der Höhe h von ALPHA nach OMEGA
  - 1) h=0 ist einfacher Fall
  - 2) 1. Teilproblem h-1, 2. Teilproblem 1
    - Verschiebe Turm der Höhe h-1 von ALPHA nach DELTA
    - Verschiebe die unterste Scheibe (h=1) von ALPHA nach OMEGA
    - Verschiebe Turm der Höhe h-1 von DELTA nach OMEGA
- Oberste Scheibe hat Nr. 1, unterste Scheibe hat Nr. h

Rekursive Datenstrukturen:

- haben wir bereits im Syntax-Diagramm benutzt (z.B. „Ausdruck“)
- → Objekte von rekursiven Datenstrukturen können beliebig groß werden.
- → Rekursion eignet sich für dynamische Datenstrukturen (Objekte können beliebig wachsen und schrumpfen)
  - anders als z.B. bei Arrays, die feste Größe haben
  - Beispiele für dynamische Datenstruktur: Liste, Bäume, Graphen, …
- Liste:
  - Folge von Elementen, beliebig lang, kann wachsen und schrumpfen
  - Realisierung mit Klasse Element
    - Element-Objekte haben 2 Attribute:
      - wert: int-Wert des Elements
      - next: Zeiger auf das nächste Listenelement, bzw. null beim letzten Element
    - Datenstruktur Element ist rekursiv, da es ein Attribut (next) gibt, das den gleichen Datentyp (Element) hat.
    - → beliebig lange Listen repräsentierbar und Listen beliebig veränderbar
    - Nachteil: leere Liste muss durch null repräsentiert werden
    - → Man kann keine nicht-statischen Methoden schreiben, die auch für leere Listen funktionieren. (z.B. fuegeVorneEin(int wert))
      - sollte vorne an die Liste ein Element mit Wert wert hängen
      - wäre nicht bei leeren Liste benutzbar, da null keine nicht-statischen Attribute oder Methoden hat (NullPointerException)
      - → leere Liste sollte nicht durch null repräsentiert werden
      - Lösung: weitere Klasse Liste
        - Die leere Liste l ist ein Objekt der Klasse Liste mit l.kopf=null.

Entwerfe typische Algorithmen auf Listen:

- Entwerfe erst die Schnittstellendokumentation der öffentlich zugreifbaren Methoden für Listen
- Danach: Implementierung (insbesondere: rekursive Algorithmen sind besonders geeignet zur Bearbeitung rekursiver Datenstrukturen wie „Element“)

- 29.11.2011 -

Rekursive Algorithmen sind besonders gut geeignet, um auf rekursiven Datenstrukturen zu arbeiten.
Oft: Rekursion analog zur Definition der rekursiven Datenstruktur (strukturelle Rekursion).

Beispiel (Folie):

- suche(x) → liefert das erste Element der Liste mit Wert x
- bei Liste l mit den Zahlen 4, 17, 25 liefert l.toString(): (4 17 25)
- l.drucke() gibt (4 17 25) auf Bildschirm aus
- f.fuegeVorneEin(2) ändert l zu (2 4 17 25)
- fuegeSortiertEin: fügt einen Wert vor dem ersten größeren Wert in der Liste ein (wenn Liste vorher aufsteigend sortiert war, dann auch hinterher)
- loesche(x): löscht erstes Element mit angegebenen Wert x
- loesche(): löscht die gesamte Liste

- Alternative: Integer.toString(wert) statt Integer(wert).toString
- Statische Hilfsmethode suche(int wert, Element kopf) → liefert das erste Element in der Liste, die mit kopf beginnt, das den vorgegebenen Wert hat.
- Strukturelle Rekursion: bearbeite die nicht-rekursiven Attribute, rekursiver Aufruf mit rekursiven Attribut der Datenstruktur (next).
- Statische Hilfsmethode durchlaufe(kopf) → liefert String zurück aus den Zahlen in der Liste, die mit kopf beginnt.
- l.drucke() → hierbei wird l automatisch mit l.toString() in einen String gewandelt.
- Statische Hilfsmethode fuegeSortiertEin(wert, kopf) → fügt neues Element mit übergebenem Wert vor dem ersten größeren Element ein, in der Liste, die mit kopf beginnt, und liefert als Ergebnis das erste Element der ursprünglich mit kopf beginnenden Liste (nach dem Einfügen) zurück.
- Statische Hilfsmethode loesche(wert, kopf) → löscht das erste Element in der mit kopf beginnenden Liste, das den übergebenen Wert hat, und liefert das erste Element der Liste (nach dem Löschen) zurück.

Man kann auch iterative Algorithmen für rekursive Datenstrukturen schreiben.

Weitere dynamische Datenstrukturen:

- Binärbäume
- Mehrwegbäume
- Graphen
- doppelt verkettete Listen
- …

Binärbaum:

- Die Welt steht Kopf... ;-(
- Elemente (Knoten) können 2 Nachfolger haben
- Wurzel steht oben, Blätter (keine Nachfolger) stehen unten

Klassen, Vererbung, Unterklassen:

- Klassen Student, Angestellter sind ähnlich, aber nicht gleich
- → viele Methoden (z.B. toString) sind für beide Klassen gleich
- → Studenten und Angestellte sind Spezialfälle von Personen
- Student ist jetzt Unterklasse von Person
- Student und Angestellter sind disjunkt
- Unterklasse erbt alle Eigenschaften (Attribute und Methoden) der Oberklasse
- Unterklasse kann weitere zusätzliche Attribute und Methoden deklarieren
- Vererbung:
  - Jede Klasse hat höchstens eine direkte Oberklasse (Einfachvererbung) [in Java]
  - später: Simulation von Mehrfachvererbung
  - public class A {…}
  - public class B extends A {…}
  - public class C extends B {…}
  - Objekte der Klasse C erben alle Eigenschaften aus B und aus A
  - Objekte der Klasse B erben nur Eigenschaften aus A

- 30.11.2011 -

Datentypanpassung und Zugriff:

- Student s; Angestellter a; Person p;
- p = s; (automatische Konversion vom speziellen Typ zum allgemeinen Typ)
  → implizite Datentypanpassung
  - Geht s.matrikelnr verloren? (Nein)
  - p.matrikelnr erlaubt?
    - (Ja, denn p zeigt auf ein Objekt vom Typ Student.) → falsch
    - Nein, denn p hat Typ Person und matrikelnr ist kein Attribut von Person. → richtig!
    - Problem: Mann kann zur Compile-Zeit im Allgemeinen nicht herausfinden, zu welcher Unterklasse ein Objekt gehört (kann z.B. von Benutzereingaben abhängen).
    - Ziel: statische Typ-Prüfung, d.h. wenn Programm zur Compile-Zeit überprüft wurde, dann soll sicher sein, dass bei Ausführung keine Typfehler auftauchen.
- s = a; → verboten! (Angestellter ist keine Unterklasse von Student)
- s = p; → verboten! (man kann im Allgemeinen nicht sicher sein, ob p wirklich auf einen Studenten zeigt)
- s = (Student) p;
  - konvertiert Personen-Objekt in Studenten-Objekt
  - Falls p nicht auf einen Studenten gezeigt hat, dann Programmabbruch (Exception).
- if (p instanceof Student) s = (Student) p;
  - überprüft zur Laufzeit, ob ein Objekt zu bestimmter Klasse gehört
- Bei der Anpassung von Unter- zu Oberklasse gehen die speziellen Attribute der Unterklasse nicht verloren (sind nur nicht zugreifbar).
- Wenn man von Ober- in Unterklasse zurückkonvertiert, ist spezielles Attribut wieder zugreifbar.

Konzeptionelles Modell der Anordnung von Objekten in Klassenhierarchien

- *Generelle Konzept bei Unterklassen*
- *Konstruktoren in Klassenhierarchien*
- *Verdecken von Attributen und Überschreiben von Methoden*

Konstruktoren in Klassenhierarchien:

- Alle Klassen sind automatisch Unterklassen von der (vordefinierten) Klasse Objekt.
  - z.B. class Rechteck {…}
  - extends Objekt wird automatisch ergänzt
- Auch bisher wurden Konstruktoren in Klassenhierarchien ausgeführt.
  - Mit super(); wird aus dem Konstruktor der Unterklasse der Konstruktor der direkten Oberklasse aufgerufen.
  - Man kann auch von einem Konstruktor aus einen anderen Konstruktor derselben Klasse aufrufen. → this();
    - 2 Bedeutungen von this:
      - this(...) → Aufruf eines Konstruktors derselben Klasse
      - this → aktuelles Objekt
  - Anweisung this() oder super() muss die erste Anweisung des Konstruktors sein.
  - Wenn die erste Anweisung nicht this() oder super() ist, dann ergänzt Java automatisch super(); als erste Anweisung.
  - Achtung: Wenn (Ober-)Klasse Konstruktoren hat (aber keinen ohne Parameter), dann wird der parameterlose Konstruktor der Oberklasse nicht automatisch erzeugt.
    → Fehlerquelle!

Verdecken von Attributen und Überschreiben von Methoden:

- Bisher:
  - Vererbung von Attributen und Methoden von Oberklasse zu Unterklasse
  - Zusätzliche (spezielle) Attribute und Methoden in Unterklasse
- Was passiert, wenn in Unterklasse Attribute/Methoden definiert werden, die genauso heißen wie in Oberklasse?
  - In Java: Unterschiedliche Regeln für Attribute und Methoden!
- Verdecken von Attributen:
  - Attribute in der Unterklasse verdecken gleichnamige Attribute der Oberklasse
    - s.hochschule → spezielles Attribut hochschule der Klasse Student
    - p.hochschule → allgemeines Attribut hochschule der Klasse Person
    - → es hängt vom Typ der Variablen ab, auf welches Attribut zugegriffen wird
- Überschreiben von Methoden:
  - Ist oft sinnvoll, denn abhängig von der jeweiligen Unterklasse soll sich die Implementierung der Methode unterscheiden.
  - Gleichnamige Methode in Unterklasse überschreibt Methode in Oberklasse
    - s.mahung() → spezielle Methode mahnung der Klasse Student
    - p.mahung() → ebenso (falls p instanceof Student)
    - → es hängt nicht vom Typ der Variablen ab, auf welche Methode zugegriffen wird, sondern nur vom Typ des Objekts, auf das p zeigt
    - → erst zur Laufzeit entscheidet sich, welche Implementierung der Methode ausgeführt wird
      - dynamisches Binden/ spätes Binden
      - ad hoc: Polymorphismus (eine Methode ist für Objekte verschiedener Typen anwendbar)
    - hier: verschiedene Implementierungen der Methode. Welche ausgeführt wird, hängt vom Typ der Argumente ab. Erst bei der Ausführung (ad hoc) wird entschieden, welche davon gewählt wird.
    - Andere Form von Polymorphismus: parametrischer Polymorphismus (später)
    - Typische Verwendung: p.mahnung() für verschiedene Objekte p. Automatisch wird immer die „richtige“ Methode für das jeweilige Objekt gewählt.
    - → man kann sendeMahnungen schon schreiben, bevor es Unterklassen von Person gibt. Unterklassen können später hinzukommen und mahnung-Methode überschreiben.

- 06.12.2011 -

Vorteil von überschriebenen Methoden:

- abhängig vom Typ des Objekts (zur Laufzeit) wird jeweils die „richtige“ Implementierung der Methode ausgeführt (ad-hoc Polymorphismus).
- final:
  - bei Methoden: Methode darf nicht überschrieben werden
  - bei Attributen: Konstanten, die nur einmal gesetzt werden dürfen
  - bei Klassen: Klasse darf keine Unterklassen haben
- überschreibende Methode muss mindestens so sichtbar sein wie die überschriebene Methode
- statische Methoden nur mit statischen Methoden überschreiben (analog: nicht-statisch)
- (diese beiden Regelungen gelten nicht beim Verdecken von Attributen)

Zugriff auf überschriebene Methode aus Oberklasse:

- super: aktuelles Objekt, aber als Objekt der Oberklasse gesehen (im Beispiel: Objekt der Klasse Person)
- this: aktuelles Objekt, für das die Methode gerade ausgeführt wird (im Beispiel: Objekt der Klasse Student)
- kann man sowohl im Konstruktor benutzen, um einen anderen Konstruktor der Oberklasse bzw. der eigenen Klasse aufzurufen: super(…) bzw. this(…)
  als auch, um eine andere Methode der Oberklasse bzw. der eigenen Klasse aufzurufen: super**.**…(…) bzw. this**.**…(…)
  genauso, um auf (verdeckte) Attribute der Oberklasse bzw. der eigenen Klasse zuzugreifen: super**.**… bzw. this**.**…

3 verschiedene Konzepte:

1. Überladen von Methoden:
   - verschiedene Methoden mit gleichem Namen, aber unterschiedlicher Signatur (d.h. Parameter sind verschieden)
   - bereits zur Compile-Zeit wird Überladung aufgelöst und entschieden, welche Methode (mit welcher Signatur) ausgeführt wird
   - damit eine Methode eine andere überschreibt, müssen sie die gleiche Signatur haben (d.h. gleichviele Parameter mit den gleichen Typen)
     - p.mahnung(10, 5) → 2-stellige mahnung-Methode aus Klasse Person
     - s.mahnung(10, 5) → 2-stellige mahnung-Methode aus Klasse Person
     - s.mahnung(10) → 1-stellige mahnung-Methode aus Klasse Student
     - p.mahnung(10) → 1-stellige mahnung-Methode aus Klasse Student
     - Was passiert, wenn die mahnung-Methode der Klasse Student einen anderen Parameter-Typ hat? (statt int: double)
       - s.mahnung(10): mahnung(int...) aus Person (kein Überschreiben, nimm die überladene Methode, die am besten „passt“)
       - p.mahnung(10): mahnung(int...) aus Person
     - void f(Person p) {…}
     - void f(Student s) {…}
     - Student s = new Student();
     - Person p = s;
     - f(s); → führt f(Student s) aus
     - f(p); → führt f(Person p) aus
     - überladen wird statisch, d.h. zur Compile-Zeit aufgelöst

2. Überschreiben von Methoden
   - Zwei Methoden mit gleicher Signatur in Ober- und Unterklasse. Wird dynamisch, d.h. zur Laufzeit, aufgelöst.
3. Verdecken von Attributen
   - Zwei Attribute mit gleichem Namen in Ober- und Unterklasse. Wird statisch zur Compile-Zeit aufgelöst.

Weitere Beispiele für vordefinierte Methoden:

- String toString()
  - gibt es bereits in Klasse Objekt, kann in jeder Klasse überschrieben werden
- boolean equals(Objekt o)
  - gibt es bereits in Klasse Objekt, kann in jeder Klasse überschrieben werden
- …
- sollten auch überschrieben werden, damit sie das „Richtige“ tun

Bsp. (s. Folie):

- Liste für Brüche, Liste für Wörter → sehr unelegant!
- Klassen Bruchelement, Wortelement, … sind nahezu identisch
- → schlecht wartbar bei Änderungen
- Besser:
  - nur eine generelle Klasse für Listen
  - sollte für Listen verschiedener wert-Typen verwendbar sein
- 1. Idee: Allgemeine Liste, in der man werte vom Typ Objekt speichern kann.
  - Alle Klassen sind Unterklasse von Objekt
  - man darf Brüche, Wörter, Integer, … in Listen speichern
- 1. Nachteil: Liste kann Objekte verschiedener Typen durcheinander enthalten.
  - z.B. sowohl Brüche als auch Wörter
  - → generische Typen (später)
- 2. Nachteil: Methoden der Klasse Liste müssen für beliebige Objekte funktionieren.
  - z.B. in Methode suche muss auf Gleichheit überprüft werden. Die einzige Gleichheit, die für bel. Objekte verfügbar ist, ist die „Objektgleichheit“ (==).
  - Man hätte gern, dass wert nur von solchem Typ sein darf, bei dem der Benutzer eine Methode "gleich" für inhaltliche Gleichheit definiert hat.
- Lösung:
  - Definiere neue (Ober-)Klasse Vergleichbar mit Methode "gleich".
  - Als wert sind jetzt nur noch Objekte der Klasse Vergleichbar (oder ihrer Unterklassen) erlaubt.
  - Da es in der Klasse Vergleichbar keine sinnvolle Implementierung der Methode "gleich" gibt, lasse dort ihren Methodenrumpf weg (stattdessen schreibe ein Semikolon).
    → Solche Methoden heißen abstrakt, müssen mit Schlüsselwort *abstract* gekennzeichnet werden.
  - Eine Klasse mit mindestens einer abstrakten Methode heißt abstrakt und muss auch mit Schlüsselwort abstract gekennzeichnet werden.
  - Sinn der abstrakten Methode:
    - Erzwingt Vorhandensein dieser Methode in allen Unterklassen.
    - Unterklasse muss alle abstrakten Methoden mit konkreten Methoden überschreiben oder Unterklasse ist ebenfalls abstrakt.
  - Abstrakte Klassen dürfen Konstruktoren haben, aber man kann keine Objekte erzeugen, die nur den Typ einer abstrakten Klasse haben.
    - Vergleichbar v = new Vergleichbar(); → verboten!

- 07.12.2011 -

Abstrakte Klassen:

- Alle Unterklassen müssen die abstrakten Methoden implementieren (oder Unterklasse ist auch abstrakt).
- Als Venn-Diagramm:
  - public abstract class Vergleichbar
  - public class Bruch extends Vergleichbar
  - public class Wort extends Vergleichbar
  - Vergleichbar

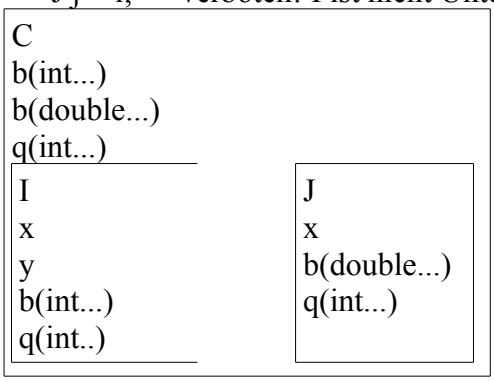
Bruch	Wort

    → new Vergleichbar(...) verboten!
  - Wäre Vergleichbar nicht abstrakt, dann:

Vergleichbar	
Bruch	Wort
- Auch bei abstrakten Klassen nur Einfachvererbung (jede Klasse hat nur eine direkte Oberklasse).
- Grund: Implementierung von Methode wird von der direkten Oberklasse geerbt. Da auch in abstrakten Klassen nicht unbedingt alle Methoden abstrakt sind, ist diese Einschränkung nötig für Eindeutigkeit.

Interfaces:

- ganz abstrakte Klassen, in denen alle Methoden abstrakt sind
- hier ist Mehrfacherbung erlaubt
- Syntax von Interfaces:
  - wie bei abstrakten Klassen, aber alle Methoden müssen abstrakt sein
  - alle Interfaces sind public
  - alle Methoden im Interface sind public und abstrakt
  - alle Attribute im Interface sind public, static und final
  - wenn Klasse A Unterklasse eines Interfaces I ist: class A implements I (statt extends)
- Wenn eine Klasse ein Interface implementiert, muss sie alle Methoden des Interfaces überschreiben (oder Klasse ist abstrakt).
- Verwende Interfaces wie andere Klassen als Datentyp.
- Achtung: Interfaces dürfen keine Konstruktoren haben.
- Bsp. (s. Folie):
  - C muss alle Methoden aus I und J überschreiben (es muss in C also sowohl b(int...) als auch b(double...) geben).
  - I i = z; → explizite Datentypanpassung von Unterklasse zu Oberklasse
  - J j = i; → verboten! I ist nicht Unterklasse von J



    - i.b(5) → führt die Methode b(int...) aus C aus, wie z.b(5)
    - j.b(5) → führt b(double...) aus C aus, wie z.b(5.0)
    - i.q(5) und j.q(5) → führt q(int...) aus C aus, wie z.q(5)
    - I.x = 4 J.x = 3 C.y = 6
    - C.x → verboten! (nicht eindeutig)
- public interface I_und_J extends I, J (nicht implements)

Modularität und Pakete:

- Software sollte modular aufgebaut sein
- sollte aus verschiedenen Modulen/ Paketen (packages) bestehen
- Änderungsaufwand auf ein Modul begrenzt
- getrennte Erstellung der Module
- package: Zusammenfassung bestimmter Klassen und Interfaces
- package-Struktur entspricht der Verzeichnisstruktur des Dateisystems
    - Verzeichnis „listen“
        - enthält die Klassen und Interfaces des Pakets „listen“
        - jede davon beginnt mit „package listen;“
- kein Schlüsselwort → sichtbar im eigenen Paket
- protected → sichtbar im eigenen Paket und in allen Unterklassen

Zugriff auf Pakete:

- Test-Klassen außerhalb des Pakets „listen“
- Name des Pakets muss mit angegeben werden: listen.Liste
- oder: „import“
    - import listen.Liste;
    - import listen.*;
- nur das Paket java.lang wird automatisch importiert (import java.lang.*;)

Compilierung:

- Test.java im Arbeitsverzeichnis
- Liste.java, Element.java im Unterverzeichnis listen
- javac Test.java → compiliert alle benötigten Klassen, auch in anderen Paketen
- javac listen/Liste.java → compiliert nur Liste.java + Element.java

Pakethierarchie:

- muss nicht flach sein
- Beispiel:
  - unbenanntes Paket = Arbeitsverzeichnis
    - Paket listen
      - Klasse Liste
      - Klasse Element
    - Paket werte
      - Paket zahlen
        - Klasse Bruch
        - Klasse Int
      - Paket zeichen
        - Klasse Wort
      - Klasse Aenderbar
      - Klasse Vergleichbar
      - Klasse Zahl
- man kann im Paket werte.zahlen nicht automatisch auf alle Komponenten aus Paket werte zugreifen → import werte.*;
- import werte.*; → importiert nur Klassen und Interfaces aus werte, nicht aus werte.zahlen oder werte.zeichen!

- 13.12.2011 -

Ausnahmen (Exceptions):

- Fehlerquellen, die erst zur Laufzeit behandelt werden können
- 2 Möglichkeiten:
  - alle möglichen Fehler vorher durch „if“ ausschließen
    - unlesbares Programm
    - nicht alle Fehler sind vorher abfragbar
  - Konzept zur Ausnahmebehandlung
    - Laufzeitsystem meldet auftretende Ausnahmen automatisch während der Ausführung
    - Behandlung von Ausnahmen kann in eigenem Programm passieren (muss nicht zum Programmabbruch führen)
- try {…} ← führt Code im Block aus, bis Ausnahme auftritt
- catch (E1 x) {…}
- catch (E2... x) {…} ← wenn im try-Block eine Ausnahme auftrat, dann breche try-Block ab und führe den Typ entsprechenden Block aus (x wird mit der geworfenen Ausnahme instantiiert)
- finally {…} ← wird zum Schluss ausgeführt (d.h. wenn im try-Block keine Ausnahme geworfen wurde oder nach der Behandlung der Ausnahme im entsprechenden catch-Block)
- Vorteil: Nicht jede mögliche Fehlerstelle braucht eine eigene Behandlung.
- Auch Exceptions sind Objekte → mehrere Klassen für Exceptions.
- Alle Objekte vom Typ Throwable kann man „fangen“ (d.h. mit catch behandeln und das Programm weiter fortsetzen), aber manche Ausnahmesituationen sind so gravierend, dass man das Programm besser abbricht (Klasse Error).
- Bei mehreren catch-Anweisungen:
  - catch (E1 x) {…}
  - catch (E2 x) {…}
  - Wenn E2 Unterklasse von E1 ist, dann werden alle E2-Exceptions schon vom oberen catch gefangen. → Compiler-Fehler
- Exceptions der Klasse Error und RuntimeException müssen nicht gefangen werden (unchecked exceptions), alle anderen müssen gefangen werden (sonst Compiler-Fehler).
- Wenn eine Exception zur Laufzeit auftritt und nicht gefangen wird → Programmabbruch
- Beispiel-Exceptions (s. Folie) sind alles RuntimeExceptions (bis auf IOException, IOError)
- Beispiel (s. Folie):
  - in M4 wird Exception vom Typ A geworfen (A nicht Unterklasse von B)
  - catch in M4 hilft nicht, kein Exception-Handler für Ausnahmen vom Typ A
  - Abbruch von M4, kehre zurück zur Aufrufstelle in M3
  - Bei Ausführung von M4 trat Ausnahme von Typ A auf, Abbruch von M3, kehre zurück zur Aufrufstelle in M2
  - breche try-Block ab
  - fange geworfene Aufnahme im entsprechenden catch-Block
- Wenn Ausnahmen geworfen werden, überprüfe erst, ob Ausnahme direkt mit catch behandelt werden kann. Ansonsten breche aktuelle Methode ab und reiche Ausnahme an die aufrufende Methode weiter (usw.).
- Wenn die Ausnahme auch in main nicht gefangen wird → Programmabbruch

Benutzerdefinierte Exception-Klassen:

- beliebige Klassen (Unterklassen von Throwable)
- Exceptions können nicht nur implizit (automatisch) geworfen werden (wie bei Division durch Null), sondern auch explizit im Programmcode:
  - throw e; → wirft das Ausnahmeobjekt e (vom Typ Throwable)
- Erzeugung von Exception-Objekten wie bei anderen Objekten:
  - new NegativeNumberException(x)
- Wenn eine Methode f eine checked exception vom Typ E oder E' werden könnte, dann muss das im Methodenkopf angegeben werden:
  - … f(…) throws E, E' {…}
- Bsp. (s. Folie):
  - bei x = 3, Ausgabe: Fakultaet von 3 ist 6
  - bei x = -17, Ausgabe: Fehler! -17 < 0
  - bei x = 17, Ausgabe: Fehler! Es trat die folgende Ausnahme auf: toString(e)
- Bsp. 2 (s. Folie): try {test();}
  - bei x = 3, Ausgabe:
    Fakultaet von 3 ist 6
    Ende des try-catch-Blocks
    Ende der Methode test
  - bei x = -17, Ausgabe:
    Fehler! -17 < 0
    Ende des try-catch-Blocks
    Ende der Methode test
  - bei x = 17, Ausgabe:
    Ende des try-catch-Blocks
    Fehler! Es trat die folgende Ausnahme auf: toString(e)
- Verwende Exceptions sinnvoll (d.h. für Ausnahmen, nicht für Sprünge oder Fallunterscheidungen)!

Generische Datentypen: (Konzept aus fkt. Programmierung)

- Damit man keine eigenen Klassen für Bruchliste, Wortliste, … schreiben muss, verwendet man eine allgemeine Klasse Liste, in der man beliebige Objekte (vom Typ Objekt) speichern kann.
- Nachteile:
  - Listen können beliebige Objekte durcheinander enthalten.
  - Wenn man wert aus der Liste holt, hat dieser Typ Objekt. Um diesen z.B. als Bruch weiterzuverarbeiten, muss man erst „casten“.
- Ziel: Typ Liste mit bel. Objekten, aber alle Objekte in der Liste müssen gleichen Typ haben.
- Lösung: eine Klasse, viele Typen
- List <T> → List <Bruch>, List <Wort>, List <String>, List <Integer>, …
- T ist eine Typvariable, steht für beliebigen Typ

- 14.12.2011 -

Korrektur zu finally (13.12.2011):

- finally wird immer ausgeführt!

Generische Datentypen:

- eine Klasse Liste <T>
- viele Typen: Liste <Bruch>, Liste <String>, …
- Im Java-Bytecode gibt es nur den „raw Type“ Liste.
- Generische Klassen
- Generische Interfaces
- Generische Methoden
- Einschränkung der möglichen Instantiierung von Typvariablen.
  - Bsp.: Instantiiere T nur mit solchen Typen, bei denen „gleich“ existiert.
  - T soll nur mit Typen instantiiert werden, die „Vergleichbar“ sind (oder Unterklassen von Vergleichbar).
- Polymorphismus: Methode kann für Objekte verschiedener Typen verwendet werden.
  - ad hoc Polymorphismus: zur Laufzeit wird anhand des Typs der Argumente entschieden, welche Implementierung der Methode ausgeführt wird (Überschreiben von Methoden)
    → eine Methode – viele Implementierungen
  - parametrischer Polymorphismus: gleiche Implementierung für Argumente verschiedener Typen
    → eine Methode – eine Implementierung
- Achtung: Da es zur Laufzeit keine generische Typen mehr gibt, existiert kein:
  - … instanceof Liste <Bruch>
  - nur: … instanceof Liste (raw Type)
- Weitere Möglichkeiten für Typvariablen:
  - wildcards ?
    - Liste <?> l;
    - Liste <Bruch> b;
    - l = b;
  - wildcards kann man einschränken:
    - Liste <? extends Vergleichbar> l;
    - Liste <? super C> r;
    - …
- Ist Liste <B> Unterklasse von Liste <A>? Nein.
- Ist B[] Unterklasse von A[]? Leider ja:
- Bsp. (s. Folie): bArray[0] sollte vom Typ B sein, ist aber vom Typ C
  - schlechte Designentscheidung
  - trotz statischem Typchecking zur Compilezeit tritt Typfehler zur Laufzeit auf
- Bsp. (s. Folie): Liste <A> aList = new Liste <A> ()
  - zur Abkürzung darf man hier „Diamond“ schreiben
  - new Liste <> ()

Collections:

- Bsp. (s. Folie): (wie x++; oder ++x;)
  - it.next() ist Anweisung:
    - setzt Iterator it um ein Element weiter
    - Anweisung hat auch ein Ergebnis (d.h. man kann sie auch als Ausdruck benutzen):
      - Ergebnis ist das überlaufene Element
- foreach-Schleife existiert nicht nur für Arrays, sondern für alle Collections (ist Kurzschreibweise für Aufruf des entsprechenden Iterators)
- Autoboxing/Unboxing: Automatisches Konvertieren zwischen primitiven Datentypen und Hüllklassen
- Grund: Man kann Typparameter nur mit Klassen/Interfaces instantiieren, nicht mit primitiven Typen.
  - LinkedList <int> → nicht möglich!
  - LinkedList <Integer>
- Dank Autoboxing/Unboxing kann man jetzt auch so tun, als ob es Collections von primitiven Datentypen gäbe.

- 20.12.2011 -

Unterschied: imperative – funktionale Programmierung:

- Java – Haskell
- Problem: Programm (s. Folie) hat einen Seiteneffekt: Hinterher ist x.kopf = null (d.h. Liste wird gelöscht)

Deklarative Programmierung:

- Programm besteht nur aus der Beschreibung des Problems. Lösung muss Rechner selbst finden.

Syntax für Listen in Haskell:

- n : x steht für die Liste, die aus Liste x entsteht, wenn man vorne Element n einfügt
  - `15 : [70, 36] = [15, 70, 36]`
  - `15 : (70: (36: [])) = [15, 70, 36]`
  - `15 : 70 : 36 : [] = [15, 70, 36]`
  - : assoziiert nach rechts
- Jede nicht leere Liste kann in der Form `kopf : rest` dargestellt werden.
  - `len [15, 70, 36]` → `15 : [70, 36]`
- `len([]) = 0` → Klammen um das Argument dürfen in Haskell weggelassen werden.
- len ist rekursiv definiert.

Ausführung von Haskell:

- Speichere Programm in file.hs

Haskell-Programm:

- Folge von Deklarationen, die linksbündig untereinander steht
- Typdeklaration gibt an, welchen Typ eine Funktion hat
  - `len :: [Int] -> Int`
    - [Int] Typ der Listen, die Int-Zahlen als Argumente enthalten
- Funktionsdeklaration gibt die Abbildungsvorschrift an
- Kommentare:
  - `--` bis Zeilenende
  - `{-`
    `…`
    `-}`
- Variablen: beliebige Strings, die mit Kleinbuchstaben (oder _) anfangen
- Variablenbezeichner werden auch als Namen von Funktionen benutzt:
  - `square, double :: Int -> Int`
- Typdeklarationen müssen nicht mit angegeben werden (dann berechnet der Compiler sie automatisch), aber es ist guter Stil, sie mit anzugeben.
- exp (expression): Ausdruck, der das Ergebnis der Funktion beschreibt
- pat (pattern): spezieller Ausdruck, der für die Form der erwarteten Argumente steht
  - `[]` steht für die leere Liste
  - `(kopf : rest)` steht für eine nicht-leere Liste
  - `x` steht für beliebigen Wert

- Man darf auch Funktionen ohne Argumente (Konstanten) deklarieren:
  - `one :: Int`
  - `one = 1`
  - `pi :: Float`
  - `pi = 3.1415927`
- Arithmetische Grundoperationen sind vordefiniert: +, -, *, /, …
- auch: ==, <, <=, >, >=, …
- Datentyp Bool ist vordefiniert
  - mit Werten True, False
  - und Operationen: &&, ||, not, …

<u>Auswertung von Haskell-Ausdrücken:</u>

- Termersetzung:
  - Finde linke Seite, die auf einen Teil des auszuwertenden Ausdruckes passt (pattern matching)
  - Ersetzt entsprechend instantiierte linke Seite der Gleichung durch die entsprechend instantiierte rechte Seite der Gleichung
- Welche Auswertungsreihenfolge wählt Haskell?
  - call-by-value: eager evaluation (innermost/leftmost evaluation)
  - Vorteil von call-by-name (outermost/leftmost evaluation):
    - Man wertet nur dann ein Argument aus, wenn man es wirklich braucht
      - `three :: Int -> Int`
      - `three x = 3`
      - `non_term :: Int -> Int`
      - `non_term x = non_term (x+1)`
      - `three (5*(131-27))` → (call-by-name) → 3
      - `non_term 0 = non_term (0+1) = non_term …`
      - `three (non_term 0)`
        - terminiert bei call-by-name zu 3
        - terminiert nicht bei call-by-value
    - Wenn irgendeine Auswertung des Ausdruckes terminiert, dann terminiert auch call-by-name-Auswertung.
    - Es gibt Beispiele, bei denen call-by-name terminiert und call-by-value nicht.
    - Auswertungsstrategie beeinflusst Terminierungsverhalten und Effizienz.
    - Aber: wenn man ein Ergebnis erhält, dann immer dasselbe unabhängig von der Auswertungsstrategie.
  - Nachteil von call-by-name:
    - Wenn noch auszuwertende Teilausdrücke dupliziert werden, dann muss man anschließend alle diese Kopien auswerten (kann ineffizienter als call-by-value werden).
  - Haskell: Lazy Evaluation
    - call-by-name (leftmost outermost), aber:
    - duplizierte Ausdrücke werden nicht doppelt, sondern parallel ausgewertet

Bedingte def. Gleichungen und Tupel:

- `maxi :: (Int, Int) -> Int`
- Typ der 2-Tupel, wobei beide Komponenten Ints sind (Int x Int).
- Genauso möglich:
  - `(Bool, [Int], [Bool])`
  - `([Int], [[Int]])`
  - `[[Int]]` → Typ der Listen, bei denen die Elemente vom Typ `[Int]` sind
    - z.B. `[[1, 2, 3], [4, 5], []]`
- Rechte Seite einer def. Gleichung darf eingeschränkt werden
  - ```
    maxi (x, y) | x > y     = x
                | x==y      = x
                | otherwise = y
    ```
    - beliebiger boolescher Ausdruck, es wird die erste Gleichung genommen, deren boolescher Ausdruck zu True auswertet.
    - Vordefiniert:
      - `otherwise :: Bool`
      - `otherwise = True`
    ```

- 21.12.2011 -

Currying: (Haskell B. Curry)

- `plus 2 3`
  - `plus 2` → dies ist eine Funktion von `Int -> Int`
  - d.h. `plus 2` ist eine Funktion, die ein Argument y erwartet und als Ergebnis 2 + y zurückliefert.
- Jetzt ist es möglich:
  - ```
    suc :: Int -> Int
    suc = plus 1
    ```
  - Auswertung von `suc 5`:
    - ```
      suc 5 =
      (plus 1) 5 = 1 + 5 = 6
      ```
    - → dasselbe wie `plus 1 5`, d.h. Funktionsanwendung assoziiert nach links
- Vorteile des Currying:
  - Man kann Funktionen mit nur einem Teil ihrer Argumente aufrufen (partielle Anwendung).
  - Klammerersparnis
- Typ von plus:
  - `plus :: Int -> (Int -> Int)` → `plus :: Int -> Int -> Int`
    - d.h. assoziiert nach rechts
  - `plus 1 :: Int -> Int`

Pattern-Matching:

- mehrere Funktionsdeklarationen für das gleiche Funktionssymbol
- beschreiben die Definition der Funktion für unterschiedliche Formen von Argumenten
  - werden durch Patterns (bestimmte Ausdrücke) beschrieben
- Patterns sind Terme aus Variablen und Datenkonstruktoren
- Datenkonstruktoren sind Funktionssymbole, die nicht weiter ausgewertet werden. Aus ihnen werden die Datenobjekte aufgebaut. Strings, die mit einem Großbuchstaben beginnen.
  - Datenkonstruktoren des Typs Bool: `True, False`
- Um Ausdruck `und exp1 exp2` auszuwerten, testet Haskell die Gleichungen von oben nach unten und verwendet die erste Gleichung, bei der die linke Seite auf den auszuwertenden Ausdruck passt.
  - `und False True`
- Datenkonstruktoren für Listen: `[], :`
  - `x : xs` → steht für die Liste mit erstem Element x und Restliste xs
- Auswertung von `len [1, 5]`
  - ist Abkürzung für `1 : (5 : [])`
    ```
    = 1 + len (5 : [])
    = 1 + 1 + 0
    = 2
    ```
- ```
  second :: [Int] -> Int
  second [] = 0
  second (x : []) = 0
  second (x : (y : xs)) = y
  ```
- auch möglich:
  - ```
    second [] = 0
    second [x] = 0
    second (x : y : xs) = y
    ```
  - → Kurzschreibweise ist auch bei patterns erlaubt

- `len ys | ys == [] = 0`
  `| otherwise =`
  - Problem: Wie kommt man ohne Pattern-Matching an die Restliste? (ys ohne erstes Element)
- `uth [0, 1, 2, 3, 4] 2 = 1`
- ```
  uth :: [Int] -> Int -> Int
  uth [] x = 0
  uth (y : ys) x    | x > 0     = uth ys (x-1)
                    | x = 0     = y
                    | otherwise = 0
  ```
- `hundertes_element xs = uth xs 100`
- Datenkonstruktoren für Int: 0, 1, -1, 2, -2, …
- ```
  fac :: Int -> Int
  fac 0 = 1
  fac x | x > 0 = x * fac (x-1)
        | otherwise = 1
  ```
- Einige Int-Patterns sind Zahlen oder Variablen
  - ```
    start_with_5 :: [Int] -> Bool
    start_with_5 (5 : xs) = True
    start_with_5 xs       = False
    ```
- Um zu entscheiden, ob ein Pattern auf einen Ausdruck passt, kann es nötig sein, den Ausdruck ein paar Schritte auszuwerten.
  - `zeros :: [Int]`
    `zeros = 0 : zeros` ← zeros terminiert nicht (unendliche Liste [0, 0, 0, …])
    ```
    f :: [Int] -> [Int] -> [Int]
    f [] ys = []
    f xs [] = []
    f xs ys = [0]
    ```
  - Terminiert `f zeros zeros`?
- Um zu entscheiden, ob die erste f-Gleichung anwendbar ist, muss man herausfinden, ob die Patterns passen. Patterns werden von links nach rechts überprüft. Passt Patterns `[]` auf Argument zeros? Um dies zu entscheiden, wird zeros ausgewertet, aber nur soweit, bis man entscheiden kann ob Pattern passt (d.h. bis man weiß, ob Ergebnis von zeros mit `[]` oder mit : gebildet wird).
  - ```
    f zeros zeros = f (0 : zeros) zeros
    = f (0 : zeros) (0 : zeros)
    = [0]
    ```
- Ziel: Finde x mit $ax^2 + bx + c = 0$ ($a, b, c \in \mathbb{Q}$)
- Lokale Deklaration:
  - ```
    var pat_1 … pat_n =  exp
                         where {…
                                }
    ```
  - Lokale Deklarationen, die nur in exp verwendbar sind
  - Vorteil:
    - lokale Deklarationen werden nur einmal ausgewertet, auch wenn sie in exp mehrfach vorkommen
  - Schreibweise, um {} und ; zu sparen:
    - Offside-Regeln (besagen, was Einrücken von Zeilen bedeutet):
      1. Das erste Symbol einer Sammlung von Deklarationen bestimmt den linken Rand des Deklarationsblocks.
      2. Eine neue Zeile, die an diesem linken Rand anfängt, ist eine neue Deklaration in diesem Block.
      3. Eine neue Zeile, die weiter rechts anfängt, gehört zur selben Deklaration (d.h. ist Fortsetzung der darüber liegenden Zeile).
      4. Eine neue Zeile, die weiter links anfängt, bedeutet, dass der
    ```

Deklarationsblock beendet ist und das die neue Zeile nicht mehr zum alten Deklarationsblock gehört.

Ausdruck:

- hat einen Wert
- hat einen Typ
- Haskell überprüft vor der Auswertung, ob der Ausdruck einen korrekten Typ hat.
- Im GHCi.
  - `:t exp`
  - → gibt den Typ des Ausdrucks aus

- 11.01.2012 -

Currying:

- `plus :: Int -> Int -> Int`
- `plus 1 5 = 1+5 = 6`
- `plus 1` → `Int -> Int`

Ausdruck:

- hat einen Typ
- wertet zu einem Ergebnis aus
- var (Strings, die mit einem Kleinbuchstaben beginnen), z.B. x, y, square (Typ Int -> Int), plus
- constr Datenkonstruktoren (Strings, die mit einem Großbuchstaben beginnen), z.B. True, False, [], : (Ausnahmsweise keine Strings mit Großbuchstaben)
  dienen zum Aufbau von Datenobjekten, können nicht weiter ausgewertet werden
- Typ:
  - True hat Typ Bool
  - `[]` hat Typ `[a]` (a kann beliebiger Typ sein)
  - : hat Typ `a -> [a] -> [a]`
- integer: 0, 1, -1, 2, -2, … Typ: Int
- float: -2.5, 3.4e+23, … Typ: Float
- char: 'a', …, 'A', '0', …, ' ', '\n', … Typ: Char
- `[exp1, …, expn]` steht für die Liste der Ausdrücke
  Ist Kurzschreibweise für `exp1 : exp2 : … : expn : []`
  Nur möglich, wenn exp1, …, expn den gleichen Typ a haben. Dann hat `[exp1, …, expn]` den Typ `[a]`.
- string = Liste von Chars
  Statt `['h', 'a', 'l', 'l', 'o']` darf man auch `"hallo"` schreiben. Ist für Haskell dasselbe.
  Typ: `String = [Char]`

  ```
  len :: [Char] -> Int
  len "hallo" = 5
  ```
- (exp1, …, expn) mit n >= 0
  steht für Tupel der Ausdrücke
  - z.B. (10, False) hat Typ (Int, Bool) entspricht math. (Int x Bool)
- Wenn exp1 den Typ a1 hat und … expn den Typ an, dann hat (exp1, … expn) den Typ (a1, …, an)
- Einelementige Tupel: (exp) sind für Haskell dasselbe wie exp.
- Nullelementige Tupel: `()` hat Typ ( )
- (exp1, …, expn), n>= 2
  - z.B. `square 10`, Typ: `Int -> Int`
  - d.h. Ausdruck square wird auf Ausdruck 10 angewendet
  - `plus 5 3`, Typ: `Int -> Int -> Int`
  - d.h. wende erst plus auf 5 an und dann das Ergebnis auf 3
  - (exp1, …, expn) steht also für ((exp1, exp2), exp3 …)
  - d.h. Funktionsanwendung assoziiert nach links
- `if exp1 then exp2 else exp3` (exp1 ist vom Typ Bool, exp2 und exp3 vom Typ a)
  Erst wird exp1 ausgewertet, dann entweder exp2 oder exp3.
  Gesamtausdruck hat Typ a.

- `let locdecls in exp` bedeutet:
  lokale Deklarationen locdecls gelten nur in exp
  analog zu `exp where locdecls`
- \ pat1 … patn -> exp (Lambda-Ausdruck)
  Wert des Ausdrucks ist die Funktion, die die Argumente pat1, …, patn erwartet und als Ergebnis den Wert exp liefert.
  - `\ x -> 2 * x`
    steht für die Funktion, die ein x als Argument erwartet und als Ergebnis 2 * x liefert, d.h. für die Verdoppelungsfunktion
  - `(\ x -> 2 * x) 5 = 2 * 5 = 10`
  - D.h. ein Ausdruck kann bereits eine ganze Funktion beschreiben → Funktionen sind „gleichberechtigte“ Datenobjekte.
  - Man hätte solche Funktionen auch „klassisch“ definieren können:
    - `double x = 2 * x`
    - `double 5 = 10`
  - durch Lambda-Ausdrücke kann man unbenannte (anonyme) Funktionen schreiben
  - \ pat1 … patn -> exp hat Typ: a1 -> … -> an -> a
  - Beispiel hat Typ `Int -> Int`
  - statt `plus x y = x + y` ist auch folgendes möglich:
    - `plus x = \y -> x + y`
    - `plus 5 3 = (\ y -> 5 + y) 3 = 5 + 3 = 8`
  - `plus = \ x y -> x + y`
  - `wert_an_der_Stelle_0 f = f 0`
  - `wert_an_der_Stelle_0 square = 0`
  - `wert_an_der_Stelle_0 (\ y -> 5 + y) = (\ y -> 5 + y) 0 = 5 + 0 = 5`

Patterns:

- Ausdrücke, die nur aus Variablen und Datenkonstruktoren bestehen.
- Beschreiben die Form der erwarteten Argumente einer Funktion.
- Gib jeweils a:
  - auf welche Ausdrücke passt (matcht) ein Pattern
  - wie werden die Variablen des Patterns dabei instantiiert?
- Für Listenkonkatenation:
  - ```
    app :: [Int] -> [Int] -> [Int]
    app [] ys = ys
    app (x : xs) ys = x : app xs ys
    ```
  - `app [1,2] [3,4,5] = [1,2,3,4,5]`
  - app ist in Haskell unter dem Namen `++` vordefiniert
  - `[1,2] ++ [3,4,5] = [1,2,3,4,5]`

Pattern Matching:

- `len (app [1] [2]) = len (1 : app [] [2]) = 1 + len (app [] [2]) = 1 + len [2] = 1 + 1 + len [] = 1 + 1 + 0 = 2`
- Haskell möchte len auswerten. Um zu entscheiden, ob die 1. len Gleichung passt, werte Argument ein bisschen aus (bis man entscheiden kann, ob 1. len Gleichung passt).
- Einschränkung: patterns müssen linear sein.
  - f pat1 … patn = exp
  - von den Ausdrücken pat1 … patn darf keine Variable mehrfach auftreten
- Grund: jetzt könnte das Ergebnis einer Funktion von der Auswertungsstrategie abhängen.
  - `zeros = 0 : zeros`
    `equal zeros zeros` → `True` (mit der 1. equal-Gleichung)
    `equal zeros (0 : zeros)` → `False` (mit der 2. equal-Gleichung)
    ```

Arten von Pattern:

- var Jede Variable ist ein Pattern
  - `square x = x * x`
  - passt auf jeden Ausdruck
  - Variable wird beim Pattern Matching mit dem Ausdruck instantiiert
  - `square 3 = 3 * 3 = 9`
  - x wird mit 3 instantiiert
- _ (undersave) Joker-Pattern
  - passt auf jeden Ausdruck
  - es findet keine Instantiierung von Variablen beim Pattern Matching statt → _ darf mehrfach in einer linken Seite auftreten (aber seine verschiedenen Vorkommen können für unterschiedliche Werte stehen)
- int, float, char, string:
  - diese Patterns passen nur auf sich selbst und es findet keine Variableninstantiierung statt.
  - ```
    ist_5 :: Int -> Bool
    ist_5 5 = True
    ist_5 _ = False
    ```
    ```

- 17.01.2012 -

Patterns:

- Auf welche Ausdrücke passt das Pattern?
- Wie werden die Variablen im Pattern beim Pattern-Matching instantiiert?
- var:
  - `square x = x * x`
  - `square 5 = 5 * 5 = 25`
- _ (Joker-Pattern)
- integer, …, string
  - ```
    is_3 3 = True
    is_3 _ = False
    ```
- (constr pat1 … patn), n >= 0
  - n-stelliger Datenkonstruktor
  - z.B.:
    - True, False, [] (0-stellig)
    - : (2-stellig, in infix Notation)
      - `x : xs` statt `(:) x xs`
      - man kann Infix-Operationen in Präfix-Funktionen „wandeln“
- Pattern: linearer Ausdruck aus Datenkonstruktoren und Variablen
- [pat1, …, patn] mit n >= 0
  - matcht Listen der Länge n, falls pati jeweils das i-te Element der Liste matcht
- (pat1, …, patn) mit n >= 0
  - matcht Tupel mit n Komponenten, falls pati jeweils die i-te Komponente des Tupels matcht

Typen:

- Jeder Ausdruck hat einen Typ
- Typ = Menge von gleichartigen Werten
- z.B.:
  - `Bool, Int, Char, Float`
  - `(Int, Int), (Int, Bool), …`
  - `[Int], [(Int, Char)], [[Int]], …`
  - `Int -> Int, [Int] -> Bool, [Int -> Int], …`
- (tyconstr type1 … typen), mit n >= 0
- n-stelliger Typkonstruktor
- Unterscheide:
  - Datenkonstruktoren: dienen zum Aufbau von Objekten, z.B. True, False
  - Typkonstruktoren: dienen zum Aufbau von Typen, z.B. Bool
    - `True && False` (ok)
    - `True && Bool` (Unsinn)
    - `x :: Bool` (ok)
    - `x :: True` (Unsinn)
- Bsp.:
  - Bool, Char, Float, Int (sind 0-stellige Typkonstruktoren, beginnen immer mit Großbuchstaben)
  - weiterer vordefinierter 1-stelliger Typkonstruktor ist […]
    - d.h. [type] ist wieder ein Typ (Typ der Listen, deren Elemente den Typ type haben)
  - weiterer vordefinierter 2-stelliger Typkonstruktor ist ->
    - d.h. type1 -> type2 ist wieder ein Typ (Typ der Funktionen von type1 nach type2)
    ```

- weiterer vordefinierter n-stelliger Typkonstruktor ist (…)
  - d.h. (type1, …, typen) ist wieder ein Typ für n >= 0 (Typ der Tupel von n Komponenten, bei denen jeweils die i-te Komponente den Typ typei hat)
- auch eine Typvariable ist ein Typ
  - nötig für parametrischen Polymorphismus
  - Typvariable steht für beliebigen Typ
  - Bsp. (s. Folie):
    - len arbeitet auf [Bool] und [Int] vollkommen gleich
    - → man sollte nur eine Funktion len schreiben, die auf beliebigen Listen arbeitet
    - a ist eine Typvariable und kann mit beliebigen Typen instantiiert werden
  - entspricht generischen Typen in Java
  - `len [1, 2, 3] = 3` (Typ [Int])
    `len [square, double] = 2` (Typ [Int -> Int]
    `len "hallo" = 5` (Typ [Char])
  - Wenn eine Typvariable mehrfach im Typ vorkommt, muss sie überall gleich instantiiert werden.
- Unterscheide:
  - Typvariablen: können mit Typen instantiiert werden
  - „normale“ Variablen: können mit Ausdrücken instantiiert werden
  - (beide werden durch Strings mit Kleinbuchstaben repräsentiert)
- app ist in Haskell unter dem Namen ++ vordefiniert:
  - `[1, 2] ++ [3, 4, 5] = [1, 2, 3, 4, 5]`
  - `"hal" ++ "lo" = "hallo"`
  - `[a] ++ [True]` (Unsinn! → nur gleiche Typen in einer Liste)

Polymorphismus:

- parametrischer Polymorphismus (mit Typparametern/Typvariablen)
  - eine Implementierung der Funktion, die auf Argumente verschiedener Typen angewendet werden kann
  - kommt aus funktionaler Programmierung
- ad-hoc-Polymorphismus
  - verschiedene Implementierungen der Funktionen
  - es hängt vom Typ der Argumente ab, welche Implementierung ausgeführt wird
  - kommt aus objektorientierter Programmierung
- Heutzutage gibt es viele Programmiersprachen mit beiden Arten vom Polymorphismus (z.B. Java und Haskell).
- Eine Funktion vom Typ `type1 -> type2` kann also auf ein Argument vom Typ type angewendet werden, falls es eine Instantiierung s der Typvariablen gibt, so dass s(type) = s(type1). Resultat der Funktionsanwendung hat den Typ s(type2).
  s entspricht hier dem Buchstaben Sigma!
- Bsp.:
  - ++ hat den Typ `[a] -> [a] -> [a]`
  - `[True] ++ []`
  - `[Bool]  [b]` ← Typen der aktuellen Argumente
  - `[a]  [a]` ← von ++ erwarteten Argumenttypen
  - Um zu überprüfen, ob man ++ auf diese Argumente anwenden darf, braucht man eine Instantiierung s der Typen, so dass
    `s([Bool]) = s([a])` und
    `s([b]) = s([a])`
    Lösung: a = Bool, b = Bool

→ Ausdruck ist typkorrekt und hat den Typ `s([a]) = [Bool]`

- Im Allgemeinen wird versucht, die allgemeinste Lösung s zu finden (instantiiert so wenig wie möglich). Vorgang, zwei Ausdrücke durch Instantiierung von Variablen gleich zu machen: *Unifikation*
  Lösung s: *Unifikator*
  Man such den *allgemeinsten Unifikator (most general unifier - mgu)*
- Weitere Beispiele zur Typinferenz:
  - : hat den Typ `a -> [a] -> [a]`
  - `f x = x : [x]`
    `b b [b]` ← Typ der aktuellen Argumente
    `a [a]` ← erwarteter Typ von :
  - Suche s mit:
    `s(b) = s(a)` und
    `s([b]) = s([a])`
    Allgemeinste Lösung: s(b) = a
    Dann `f :: a -> [a]`
- Haskell überprüft jeden Ausdruck und jede Funktion auf Typkorrektheit und bestimmt den allgemeinsten Typ.
- Noch ein Bsp.:
  - `g x = x ++ x`
    `b b b`
    `[a] [a]`
    `→ [a]`
  - Gesucht: `s(b) = s([a])`
    Allgemeinste Lösung: `s(b) = [a]`
    Dann `g :: [a] -> [a]`
  - `h x = x : [1,2]`
    `b b [Int]`
    `a [a]`
    `→ [a]`
  - Gesucht:
    `s(b) = s(a)` und
    `s([Int]) = s([a])`
    Allgemeinste Lösung:
    `s(a) = Int`
    `s(b) = Int`
    Dann `h :: Int -> [Int]`
  - `i x = \ y -> [x] : [y]`
    `b c [b] [c]`
    `a [a]`
    `→ [a]`
  - Gesucht:
    `s([b]) = s(a)`
    `s([c]) = s([a])`
    Allgemeinste Lösung:
    `s(a) = [b]`
    `s(c) = [b]`
    Dann `i :: b -> [b] -> [[b]]`
    `i 5 [6, 7] = [5] : [[6, 7]] = [[5], [6, 7]]`
  - `j x = x : x`
    `b b b`
    `a [a]`
    `→ [a]`
  - Gesucht:
    `s(b) = s(a)`
    `s(b) = s([a])`

  - Geht nicht! Dieses Unifikationsproblem ist nicht lösbar → j ist nicht typkorrekt → wird von Haskell zurückgewiesen
- Wie kann man selbst neue Typkonstruktoren einführen, um komplett neue Datentypen zu definieren?
- topdecl: Deklarationen auf oberster Programmebene
- decl: Deklarationen, die auch als lokale Deklaration auftreten dürfen (bei where und let)
- Führe jetzt data-Deklaration ein. Diese darf nur auf oberster Programmebene auftreten.
  - data typconstr = constr1 | … | constrn
  - typconstr: neuer Typkonstruktor (0-stellig)
  - constri: Datenkonstruktoren des neuen Typs (0-stellig)
  - entspricht Enums in Java
  - Geeignet für Datentypen mit endlich vielen Objekten
  - zur Ausgabe auf dem Bildschirm benutzt Haskell eine Funktion show zum Umwandeln in Stings (entspricht toString in Java)

- 18.01.2012 -

Deklaration neuer Datentypen:

- rekursive Datentypen (mit unendlich vielen Elementen)
- Bsp.: natürliche Zahlen
  - Repräsentiere
    - 0 als Term `Zero`
    - 1 als Term `Succ Zero`
    - 2 als Term `Succ (Succ Zero)`
  - `Zero :: Nats`
  - `Succ :: Nats -> Nats`
  - Nats → rekursiver Datentyp, unendlich viele Elemente
  - i.A. `tyconstr = constr1 type1^1 … type1^k1 |`
  - `constrn typen^1 … typen^kn`
  - → `constri :: typei^1 -> … -> typei^ki -> tyconstr`
  - `plus (Succ Zero) (Succ Zero) = Succ (Succ Zero)`
  - `half (Succ (Succ (Succ Zero))) = Succ Zero`
- Bsp.: Listen-Datentyp für Elemente vom Typ a
  - `data tyconstr var1 … varm = constr1 … | … | …`
  - m paarweise verschiedene Typvariablen
  - führt einen neuen m-stelligen Typkonstruktor ein
  - → im Bsp. gibt es jetzt neue Typen
    - ```
      List Int
      List Bool
      List a
      List (List Int)
      …
      ```
  - `Nil :: List a` (entspricht der leeren Liste `[]`)
  - `Cons :: a -> (List a) -> (List a)` (entspricht dem Konstruktor `:`)
  - `Cons 1 (Cons 2 Nil)` (entspricht `1 : (2 : []) = [1, 2]`)
  - `len (Cons 1 (Cons 2 Nil)) = 2`
- Bsp.: „Binärbäume“, wobei jeder Knoten 0, 1 oder 2 Kinder hat
  - `data Tree = Leaf a | OneNode a (Tree a) | TwoNode a (Tree a) (Tree a)`
  - `Leaf :: Tree a`
  - `OneNode :: a -> (Tree a) -> (Tree a)`
  - `TwoNode :: a -> (Tree a) -> (Tree a) -> (Tree a)`
  - `TwoNode 2 (TwoNode 3 (Leaf 5) (Leaf 8)) (OneNode 1 (Leaf 0))`
- Vielwegbaum: d.h. jeder Knoten hat beliebig viele Kinder
  - `data Tree a = Node a [Tree a]`
  - `Node :: a -> [Tree a] -> Tree a`
  - ```
    Node 2 [(Node 3 [Node 6 []]),
            (Node 4 []),
            (Node 5 [Node 7 [], Node 8 []])]
    ```

Funktionale Programmiertechniken: Funktionen höherer Ordnung:

- Funktion höherer Ordnung = Funktion, die als Argument oder Resultat selbst wieder eine Funktion hat
- `square :: Int -> Int` (Funktion 1. Ordnung)
- `plus :: Int -> (Int -> Int)` (Funktion höherer Ordnung)
  - `plus 5 hat Typ Int -> Int` (Ergebnis ist Funktion)
- Beispiele für Funktionen, deren Argumente Funktionen sind (math. Funktionskomposition „o“)

  - f o g → ist die Funktion, die erst g anwendet und dann f
  - `f :: b -> c`
    `g :: a -> b`
    → `f o g :: a -> c`
  - `(half o square) (4) = half (square (4)) = half (16) = 8`
  - Ist in Haskell unter dem Namen `.` vordefiniert
    - `(half.square) 4 = 8`
- Präfix-Funktionen können in Infix-Schreibweise benutzt werden, wenn man sie in Apostrophe setzt.
  - Statt `Cons 1 Nil` → `1 'Cons' Nil`
- Infix-Funktionen können in Präfix-Schreibweise benutzt werden, wenn man sie in Klammern setzt.
  - Statt `1 : []` → `(:) 1 []`
- ```
  curry :: ((a, b) -> c) -> a -> b -> c
  curry f = g
            where g x y = f (x, y)
  ```
- Es gilt:
  - `curry (uncurry g) = g`
  - `uncurry (curry f) = f`
- Einsatz von Funktionen höherer Ordnung:
  - Es gibt viele Rekursionsmuster, die immer in verschiedenen Funktionen benutzt werden.
  - Identifiziere diese Rekursionsmuster und implementiere sie in einer eigenen Funktion (höherer Ordnung).
  - In konkreten Funktionen verwende diese Funktion höherer Ordnung anstatt das Rekursionsmuster jedes Mal neu zu implementieren.
  - → bessere Lesbarkeit und Strukturierung der Programme
- ```
  suc :: Int -> Int
  suc = plus 1 (suc x = x + 1)
  ```
- `suclist [2, 5, 7] = [3, 6, 8]`
- `sqrtlist [4, 9, 16] = [2, 3, 4]`
- suclist und sqrtlist benutzen das gleiche Rekursionsmuster :
  - Durchlaufe eine Liste und wende eine Funktion auf jedes Listenelement an.
- Unterschiede:
  - Datentyp der Listenelemente (Int bzw. Float)
  - Funktion, die auf Listenelement angewendet wird (suc bzw. sqrt)
- Lösung: Abstrahiere von den Unterschieden (→ Variablen)
  - Ersetze Int bzw. Float durch Typvariable a (nur in Programmiersprachen mit parametrischer Polymorphie)
  - Ersetze suc bzw. sqrt durch Variable g (nur in Programmiersprachen mit Funktionen höherer Ordnung), `g :: a -> b`
  - Da g eine beliebige Funktion ist, sollte sie zusätzliches Eingabeargument von f sein.
    - `f :: (a -> b) -> [a] -> [b]`
  - → map (in Haskell vordefiniert)
  - Besser:
    - ```
      suclist :: [Int] -> [Int]
      suclist xs = map suc xs
      ```
  - Noch besser:
    - ```
      sqrtlist :: [Float] -> [Float]
      sqrtlist = map sqrt
      ```

- Weiteres Beispiel für eine Fkt. höherer Ordnung, die Rekursionsmuster implementiert: filter
  - `dropEven [1, 2, 3, 4] = [1, 3]`
  - `dropUpper "GmbH" = "mb"`
  - Beide Funktionen haben das gleiche Rekursionsmuster (durchlaufe Liste und lösche alle Elemente, die ein bestimmtes Prädikat nicht erfüllen), aber 2 Unterschiede:
    - unterschiedlicher Typen der Listenelemente (Int bzw. Char)
    - unterschiedliche Funktionen zum Filtern der Listenelemente (odd bzw. isLower)
  - Abstrahiere von Unterschieden
  - Funktion g sollte weiteres Eingabeargument sein:
    - `g :: a -> Bool`
  - Funktionen wie map, filter, … können (und sollen) auch auf eigenen Datenstrukturen geschrieben werden.

- 24.01.2012 -

Prolog:

- Programm = Wissensbasis
- Computer = Inferenzmaschine
  - d.h.: Computer versucht, Anfragen zu beweisen
  - d.h.: Log. Programmieren ist aus Automat. Beweisen entstanden
  - = Expertensysteme
- Prädikatenlogik als Programmiersprache
- Bsp.: Formuliere die Verwandtschafts-Datenbasis in logischen Formeln.
- Prolog = „Programming in Logic“
- Wissensbasis = Sammlung logischer Formeln (Klauseln)
  - Fakten
  - Regeln

Faktum:

- Name der Eigenschaft (bzw. Relation)
  - name(...).
  - Prädikate, die wahr oder falsch sind.
  - Objekte, die diese Eigenschaft haben.
- Relationen sind gerichtet
  - mutterVon(monika, klaus) ist wahr
  - mutterVon(klaus, monika) ist falsch

Kommentare:

- `%` (bis Zeilenende)
- `/* … */`

Laden:

- consult(datei.pl)
- Programmdateien müssen mit Kleinbuchstaben anfangen

Ausführen des Programms:

- Stelle Anfragen an das Programm
- ```
  ?- maennlich(gerd).
  true
  ```
- ```
  ?- verheiratet(werner, monika).
  true
  ```
- ```
  ?- verheiratet(monika, werner).
  false
  ```

Variablen in der Wissensbasis:

- Variablen beginnen mit einem Großbuchstaben oder einem Unterstrich.
- Prädikatssymbole (verheiratet, ...) und Konstanten (monika, ...) beginnen mit einem Kleinbuchstaben.
- `mensch(X). % alle Objekte X sind Menschen`
- Variable in Wissensbasis steht für alle möglichen Belegungen.
- `mag(X, Y). % jeder mag jeden`
- `mag(X, X). % jeder mag sich selbst`
- Gleiche Variablen in derselben Klausel müssen gleich instantiiert werden.
- Gleiche Variablen in verschiedenen Klauseln haben nichts miteinander zu tun.
  ```

Variablen in Anfragen:

- Variable in Wissensbasis: allquantifiziert
- Variable in Anfrage: existenzquantifiziert
- `?- mutterVon(X, susanne). % Wer ist die Mutter von Susanne?`
- `?- mutterVon(renate, Y). % Welche Kinder hat Renate?`
- In Prolog ist nicht festgelegt, was die Eingabe und was die Ausgabe eines Prädikats ist. Hängt nur von der Anfrage ab:
  - Ausgabe = Variablen
  - Eingabe = sonstige Argumente
- Prolog durchsucht die Wissensbasis von oben nach unten und liefert das erste Ergebnis.

Kombination von Fragen:

- `?- verheiratet(gerd, F), mutterVon(F, susanne). % Ist Gerd Vater von Sus.?`
  - Die F's in einer Klausel müssen überall gleich instantiiert werden.
- Prolog löst Anfragen, indem es einen Beweisbaum aufbaut. Anfragen werden von links nach rechts bearbeitet:
- `?- verheiratet(gerd, F), mutterVon(F, susanne).`
  - F = renate
    - `?- mutterVon(renate, susanne).`
    - □ ← leere Klausel (entspricht q.e.d. am Ende von Beweisen)
- Lösung wird von dem erfolgreichen Beweispfad (von ursprünglicher Anfrage zu □) abgelesen.
- Bsp.: Wer ist die Oma von Aline?
  - `?- mutterVon(Oma, Mama), mutterVon(Mama, aline).`
    - Oma und Mama sind Variablen.
- Beweisbaum:
  - `?- mutterVon(Oma, Mama), mutterVon(Mama, aline).`
    - Oma = monika, M = karin
      - `mutterVon(karin, aline).`
      - Keine Lösung!
    - Oma = monika, M = klaus
      - `mutterVon(klaus, aline).`
      - Keine Lösung!
    - Oma = renate, M = susanne
      - `mutterVon(susanne, aline).`
      - □
- Bei Fehlschlag setzt Prolog einen Schritt zurück und versucht dort die nächste Alternative (Backtracking).
- Effizienz wird beeinflusst durch:
  - Reihenfolge der Klauseln im Programm
  - Reihenfolge der Literale in Anfrage:
    - `?- mutterVon(Mutter, aline), mutterVon(Oma, Mutter).`
    - wäre wesentlich effizienter
- Prolog geht immer davon aus, dass seine Wissensbasis „vollständig“ ist (d.h. alles relevante Wissen enthält). → Wenn etwas nicht aus der Wissensbasis folgt, dann wird es als falsch angesehen (Closed World Assumption).

Regeln:

- Wissensbasis kann neben Fakten auch Regeln enthalten (um aus bekanntem Wissen neues Wissen herzuleiten).
- z.B.: Eine Person V ist Vater eines Kindes K, falls V mit einer Frau F verheiratet ist und F die Mutter des Kindes K ist.
  - `vaterVon(V, K) :- verheiratet(V, F), mutterVon(F, K).`
- `:-` bedeutet „falls“ oder „wenn“, nicht „genau dann wenn“!
- Regeln = wenn-dann-Beziehungen
  - Wenn die Aussagen auf der rechten Seite der Regel (Rumpf der Regel) wahr sind, dann auch die Aussage auf der linken Seite (Kopf der Regel).
- Beweisbaum:
  - `?- vaterVon(gerd, susanne).` (Prolog such nach der ersten Klausel, deren Kopf passt. Dann wird der instantiierte Klauselkopf durch den instantiierten Klauselrumpf ersetzt, d.h.: um Klauselkopf zu beweisen, reicht es, stattdessen den Klauselrumpf zu beweisen.)
    - V = gerd, K = susanne
    - `?- verheiratet(gerd, F), mutterVon(F, susanne).`
      - F = renate
      - `?- mutterVon(renate, susanne).`
      - □
- Prolog gibt nur die Belegung der Variablen aus, die schon in der Anfrage vorkamen.
- Fakten = Regeln mit leerem Klauselrumpf
- Literale im Regelrumpf sind mit „und“ (Komma) verknüpft.
- Wie könnte man „oder“-Verknüpfung realisieren?
  - Mehrere Regeln für das gleiche Prädikat:
    - `elternteil(X, Y) :- mutterVon(X, Y).`
    - `elternteil(X, Y) :- vaterVon(X, Y).`
- Beweisbaum:
  - `?- elternteil(X, susanne).`
    - Y = susanne
      - `?- mutterVon(X, susanne)`
      - X = renate
      - □
    - Y = susanne
      - `?- vaterVon(X, susanne)`
      - X = gerd
      - □

Rekursive Regeln:

- Prädikat „vorfahre“:
  - V ist Vorfahre von X, falls V ein Elternteil von X ist oder falls es ein Elternteil Y von X gibt und V Vorfahre von Y ist.

- 25.01.2012 -

Rekursive Regeln:

- `?- vorfahre(X, aline).`
  - `?- elternteil(X, aline).`
    - `?- mutterVon(X, aline).`
      - X = susanne
      - □
    - `?- vaterVon(X, aline).`
      - X = klaus
      - □
  - `?- elternteil(X, Y), vorfahre(Y, aline).`
    - `?- mutterVon(X, Y), vorfahre(Y, aline).`
      - X = monika
      - Y = karin
        - `?- vorfahre(karin, aline).` → false
      - X = monika
      - Y = klaus
        - `?- vorfahre(klaus, aline).`
        - … □
      - …

- Programmklauseln werden von oben nach unten abgearbeitet
- Literale in Klauseln werden von links nach rechts abgearbeitet
- Beweisbaum wird in Tiefensuche von links nach rechts durchlaufen.
- Sobald leere Klausel erreicht wird, gib Antwortsituation aus.
  - Antwortsituation ergibt sich aus dem Pfad von Wurzel zur □. Es werden nur die Instantiierungen für die Variablen ausgegeben, die in der ursprünglichen Anfrage auftraten.

Syntax von Prolog:

- Programm = Folge von Klauseln (nach jeder Klausel kommt ein Zeilenumbruch oder ein Leerzeichen)
- Klausel = Faktum oder Regel (bestehen aus Literalen)
- Literal = Prädikatsymbol(Term1, …, Termn) → n-stellig
  - verschiedenstellige Prädikate werden in Prolog als unterschiedlich betrachtet
    - weiblich/1 (Schreibweise, um Stelligkeit des Prädikats anzudeuten)
    - weiblich/2
  - bei 0-stelligen Prädikaten: keine Argumentklammern
    - z.B. true
- Prädikatssymbol = String, der mit Kleinbuchstaben beginnt
  - vordefinierte Prädikatssymbole: =, >, …
- Terme = Variablen oder Funktionssymbol(Term1, …, Termn)
  - Variablen = String, der mit Großbuchstaben oder Unterstrich beginnt
    - _ ist anonyme Variable
      - mehrfache Vorkommen von _ können unterschiedlich instantiiert werden
      - Belegung von _ wird in der Antwortsubstitution nicht mit ausgegeben
      - `?- verheiratet(X, Y).`
        - X = werner, Y = monika
      - `?- verheiratet(X, _).`
        - X = werner

- Funktionssymbol = String, der mit Kleinbuchstaben beginnt.
  - Jedes Funktionssymbol hat eine Stelligkeit.
    - Monika/0 ← 0-stellige Funktionssymbole heißen „Konstanten“
  - Motivation für mehrstellige Funktionssymbole:
    - Ergänze Wissensbasis um Geburtsdaten
    - 1. Möglichkeit:
      - neues Prädikatssymbol geboren/4
        - `geboren(monika, 25, 7, 1972).`
    - 2. Möglichkeit (besser):
      - neues Prädikatssymbol geboren/2
      - neues Funktionssymbol datum/3
        - Term: `datum(25, 7, 1972).`
        - Literal: `geboren(monika, datum(25, 7, 1972)).`
  - Bsp.: Wer ist im Juli geboren?
    - `?- geboren(X, datum(_, 7, _)).`
  - Prolog ist eine untypisierte Sprache, Funktionssymbole und Prädikatssymbole müssen nicht deklariert werden.
    - `datum(monika, karin, 3+4).`
  - Was passiert bei `?- geboren(X, datum(_, 3+4, _)).`
    - false, denn 3+4 ist nicht syntaktisch gleich zu 7
  - Funktionen werden nicht ausgewertet, dienen nur zum Aufbau von Datenobjekten (Funktionen in Prolog entsprechen Datenkonstruktoren in Haskell)
- Anwenden von Klauseln beruht auf syntaktischer Gleichheit
- In Prolog werden nur die Prädikatssymbole ausgewertet:
  - Wenn man eine Funktion in Prolog programmieren will, muss man dies mithilfe von Prädikatssymbolen machen.

<u>Datenstrukturen in Prolog:</u>

- Datenstruktur für natürliche Zahlen
  - typische Algorithmen (add, mult, …)
  - Repräsentiere die Datenobjekte als Terme (wie in Haskell)
  - 2 Funktionen: zero, succ
  - Terme: zero, succ(zero), succ(succ(zero)), …
- Additionsprogramm: Prädikatssymbol add/3
  - add(X, Y, Z) = Die Addition von X und Y ist Z.
  - z.B. `add(succ(zero), succ(zero), succ(succ(zero)))` ist wahr
  - `add(zero, zero, succ(zero))` ist falsch
  - Um eine n-stellige Funktion zu programmieren, benutze (n+1)-stelliges Prädikatssymbol.
  - Analoges Haskell-Programm:
    - `add x Zero = x`
    - `add x (Succ y) = Succ (add x y)`
  - Was ist 1 + 1?
  - `?- add(succ(zero), succ(zero), U).`
    - U = succ(Z)
    - `add(succ(zero), zero, Z).`
      - Z = succ(zero)
      - □
  - Antwort: U = succ(succ(zero))

  - Programm kann auch zum Subtrahieren benutzt werden.
  - Was ist 2 – 1?
    - `?- add(succ(zero), X, succ(succ(zero))).`
  - Antwort: X = succ(zero).
  - Mit welchen Summanden erreicht man die Zahl 2?
    - `?- add(X, Y, succ(succ(zero))).`
  - `?- add(X, Y, Z).`
    - unendlicher Suchbaum, unendlich oft □
- Multiplikation
  - mult(X, Y, Z) = Multiplikation von X und Y ist Z.
- Reihenfolge der Programmklauseln ist wichtig.
- Was passiert, wenn man die zwei mult-Klauseln vertauscht?
  - Terminiert nicht!

- 31.01.2012 -

Darstellung der natürlichen Zahlen:

- zero, succ(zero), succ(succ(zero)), …
- Additionsalgorithmus:
  - statt 2-stelliger Funktion benutze 3-stelliges Prädikat:
    - `add(X, Y, Z)`
    - die Addition von X und Y ist Z
- Da Eingabe und Ausgabe nicht festliegt, bekommt man automatisch auch die Umkehrfunktion: add kann auch zum Subtrahieren benutzt werden (s. Folie).

Multiplikation:

- (s. Folie)
- `?- mult(succ(zero), succ(zero), U).`
  - X = succ(zero)
  - Y = zero
  - U = Z
    Variablen in Programmklauseln werden umbenannt, sodass sie verschieden von den Variablen der Anfrage sind.
  - (Beweisbaum usw.)
- Multiplikations-Algorithmus kann auch zum Dividieren benutzt werden (analog wie oben).
- Division durch 0?
- `?- mult(zero, V, succ(zero)).`
- … (unendlicher Beweisbaum → terminiert nicht!)
- Prolog durchläuft in Tiefensuche von links nach rechts, bis die erste Lösung gefunden wird. Wenn man weitersucht (es gibt keine weiter Lösung), terminiert unser Beispiel (Division durch 1, s. Folie) nicht.
- `?- mult(V, zero, succ(zero)).`
- `false`
- Was passiert, wenn man die Reihenfolge der Programmklauseln ändert?
  - z.B. erst rekursive mult-Klausel, dann die nicht-rekursive
  - Beweisbaum von `?- mult(succ(zero), V, succ(succ(zero)))` wäre dann gespiegelt: Prolog läuft dann direkt in den linken, unendlichen Pfad und findet keine Lösung.
- Faustregel: Erst die nicht-rekursiven, dann die rekursiven Klauseln.
- Grund: Reihenfolge der Programmklauseln beeinflusst Terminierungsverhalten.
- Ist auch die Reihenfolge der Literale im Klauselrumpf wichtig?
  - Bsp.: vertausche mult(...) und add(...) in der rekursiven mult-Klausel.
  - Und wieder ein Beweisbaum: blablabla... ;-)
  - terminiert nicht mehr, wenn man nach der ersten Lösung weitersucht
- Reihenfolge der Literale im Klauselrumpf beeinflusst auch das Terminierungsverhalten.

Listen in Prolog:

- wie in Haskell:
  - `nil = []`
  - `cons(0, nil) = [0]`
  - `cons(0, cons(1, nil)) = [0, 1]`
- Längenalgorithmus:
  - statt 1-stelligem Funktionssymbol len benutze 2-stelliges Prädikatssymbol len
  - len(L, N) = Die Länge der Liste L ist N … (s. Folie)

- Prolog versucht allgemeinste Lösung zu finden: D.h. wenn die Antwortsubstitution Variablen enthält, dann dürfen diese beliebig belegt werden.
- Zahlen und Listen gibt es in Prolog auch vordefiniert.

Vordefinierte Listen:

- statt `nil` benutzt man `[]`
- statt `cons` benutzt man `.`
- statt `cons(0, cons(1, nil))` benutzt man `.(0, .(1, []))`
- Prolog konvertiert Listen aus `.` und `[]` in eine Kurzschreibweise:
  - `.(0, .(1, [])) = [0, 1]`
- Generell:
  - Man kann `[t1, …, tn | l]` schreiben für `.(t1, .(... , .(tn, l)...))`
  - z.B. `[0 | [1, 2]] = [0, 1, 2] = .(0, .(1, .(2, []))) = [0, 1 | [2]] = [0, 1, 2 | []] = [0 | .(1, [2])]`
  - `[Kopf | Rest] = .(Kopf, Rest)`

Unifikation:

- kann man die Variablen in 2 Ausdrücken so instantiieren, dass die Ausdrücke gleich werden?
- z.B.:
  - Typ-Checking bei polymorphen Typen (Haskell, Java)
  - Auswertung in Prolog
  - …
- Kann es zu s, t mehrere Unifikatoren geben?
  - Ja, s. Folie
- „Beste“ Lösung ist der allgemeinste Unifikator (most general unifier, MGU)
- Alle anderen Unifikatoren gehen aus dem MGU hervor.
- Satz:
  - Wenn 2 Ausdrücke unifizierbar sind, dann haben sie auch einen MGU.
  - MGU ist eindeutig bis auf Variablenumbenennung.
- Bsp.: Was ist der MGU von f(X) und f(Y)?
  - 1. Lösung: m1 = {X = Y} → ergibt f(Y)
  - 2. Lösung: m2 = {Y = X} → ergibt f(X)
- Bsp.: Was ist der MGU von f(X) und f(X)?
  - m = {} ← leere (identische) Substitution
- Notation:
  - m = {X1 = t1, … , Xn = tn}
  - dann m(s)
    - s, wobei X1 durch t1 ersetzt wird, usw.

- 01.02.2012 -

Unifikation:

- Gegeben: s, t
- Gesucht: MGU σ σ(s) = σ(t)
- 1. Bsp.: (s. Folie)
  - s: X t: X
  - σ = {} (d.h. σ(X) = X σ(Y) = Y)
- 2. Bsp.:
  - s = X t: succ(Y)
  - σ = {X = succ(Y)} (d.h.: σ(X) = succ(Y) σ(Y) = Y)
- Was passiert, wenn t die Variable s enthält?
  - s = X t = succ(X)
  - bei σ = {X = succ(X)}: σ(s) = succ(X) σ(t) = succ(succ(X))
  - OCCUR FAILURE
- 3. Bsp.: Analog zum 2. Fall
- 4. Bsp.:
  - s: f(...) t: g(...)
  - CLASH FAILURE
- Bsp.:
  - s: f(X, Z, succ(succ(W))) t: f(succ(Y), X, Z)
  - σ1 = MGU(X, succ(Y)) = {X = succ(Y)}
  - σ2 = MGU(σ1(Z), σ1(X)) = {Z = succ(Y)}
    Z succ(Y)
  - σ3 = MGU(σ2(σ1(succ(succ(W)))), σ2(σ1(Z))) = {Y = succ(W)}
    succ(succ(W)) succ(Y)
  - Lösung: σ = σ3(σ2(σ1)) = {X = succ(succ(W)), Y = succ(W), Z = succ(succ(W))}
- σ = {X = Y} nicht gleich σ' = {Y = X}
- Bsp.:
  - s = f(g(h(X, Z)), Z)
  - t = f(g(Y), g(X))
  - σ1 = MGU(g(h(X, Z)), g(Y)) = {Y = h(X, Z)}
  - σ2 = MGU(σ1(Z), σ1(g(X))) = {Z = g(X)}
    Z g(X)
  - σ = MGU(s, t) = σ2(σ1) = {Y = h(X, g(X)), Z = g(X)}
- Bsp.:
  - s = f(g(X), Y) t = f(Y, a)
  - σ1 = MGU(g(X), Y) = {Y = g(X)}
  - σ2 = MGU(σ1(Y), σ1(a)) → CLASH FAILURE
    g(X) a
- s = plus(2, 3) t = plus(1, 4) → nicht unifizierbar!
- Bsp.:
  - s = f(g(X), Y, X)
  - t = f(Y, g(h(Z)), g(Z))
  - σ1 = MGU(g(X), Y) = {Y = g(X)}
  - σ2 = MGU(σ1(Y), σ1(g(h(Z)))) = {X = h(Z)}
    g(X) g(h(Z))
  - σ3 = MGU(σ2(σ1(X)), σ2(σ1(g(Z)))) → CLASH FAILURE
    h(Z) g(Z)

- Bsp.:
  - s = f(Y, g(X))
  - t = f(X, Y)
  - σ1 = MGU(Y, X) = {Y = X}
  - σ2 = MGU(σ1(g(X)), σ1(Y)) → OCCUR FAILURE
    g(X) X

Prolog-Auswertungsverfahren:

- Resolution
  - 2 Wahlmöglichkeiten:
    - Welches der Literale G1, …, Gm wird zuerst bearbeitet?
      - In Prolog das linkeste (d.h. G1)
      - d.h.: Um ?- G1, …, Gm zu lösen, suche nach einer Programmklausel, die auf G1 passt (d.h. es existiert eine Programmklausel H :- B1, … Bn, sodass μ = MGU(H, G1))
      - Dann: ersetze G1 in der Anfrage durch B1, …, Bn
      - → neue Anfrage ?- μ(B1), …, μ(Bn), μ(G2), …, μ(Gm)
    - Welche Programmklausel wird genommen?
      - In Prolog: arbeite Programmklauseln von oben nach unten ab
  - Variablen in Programmklauseln werden so umbenannt, dass sie verschieden von Variablen im Beweisziel sind.
  - Bsp.: (s. Folie)
    - G1 = add(succ(zero), succ(zero), U)
    - H :- B1 ist: add(X, succ(Y), succ(Z)) :- add(X, Y, Z)
    - MGU im 1. Schritt μ1 = {X = succ(zero), Y = zero, U = succ(Z)}
    - MGU im 2. Schritt μ2 = [X1 = succ(zero), …}
    - Antwortsubstitution ist m2(m1)) = …
- Beweisbäume („SLD-Bäume“) für Verwandtschafts-Bsp.: (s. Folie)
  - Gib nur die Belegungen der Variablen im jeweiligen Beweisziel an.
  - Was passiert, wenn man die Literale im Rumpf von (4) vertauscht?
  - Was passiert, wenn man Klausel (3) und (4) vertauscht?
- Faustregeln:
  - nicht-rekursive Klauseln vor rekursiven Klauseln
  - bei rekursiven Klauseln, die nur Variablen im Klauselkopf haben, sollte das rekursive Literal im Klauselrumpf nicht ganz vorne stehen.
- Wie implementiert man einen Unifikations-Algorithmus in Prolog?
  - gleich(X, X).
  - Warum?
  - gleich(f(Y, g(Z)), f(g(U), W)).
    s t
  - Prolog berechnet: μ: MGU(gleich(X, X), gleich(s, t))
  - d.h. μ(X) = μ(s) = μ(t)
- Aus Effizienzgründen benutzt Prolog einen Unifikations-Algorithmus ohne „occur check“ (d.h. ohne zu überprüfen, ob occur failure vorliegt)
  - MGU(X, f(X)) = {X = f(f(...f(...)...))}