

Helmut Seidl

Informatik 1

TU München

Wintersemester 2007 2008

Inhaltsverzeichnis

0	Allgemeines	4
1	Vom Problem zum Programm	6
2	Eine einfache Programmiersprache	12
2.1	Variablen	13
2.2	Operationen	13
2.3	Kontrollstrukturen	15
3	Syntax von Programmiersprachen	20
3.1	Reservierte Wörter	22
3.2	Was ist ein erlaubter Name?	22
3.3	Ganze Zahlen	23
3.4	Struktur von Programmen	25
4	Kontrollfluss-Diagramme	30
5	Mehr Java	37
5.1	Mehr Basistypen	37
5.2	Mehr über Arithmetik	39
5.3	Strings	40
5.4	Felder	41
5.5	Mehr Kontrollstrukturen	43
6	Eine erste Anwendung: Sortieren	49
7	Eine zweite Anwendung: Suchen	54
8	Die Türme von Hanoi	62
9	Von MiniJava zur JVM	68
9.1	Übersetzung von Deklarationen	75
9.2	Übersetzung von Ausdrücken	76
9.3	Übersetzung von Zuweisungen	77
9.4	Übersetzung von if-Statements	79
9.5	Übersetzung von while-Statements	83
9.6	Übersetzung von Statement-Folgen	84
9.7	Übersetzung ganzer Programme	85
10	Klassen und Objekte	93
10.1	Klassen-Attribute	97
11	Ein konkreter Datentyp: Listen	100

12 Abstrakte Datentypen	109
12.1 Beispiel 1: Keller (Stacks)	110
12.2 Beispiel 2: Schlangen (Queues)	117
13 Vererbung	123
13.1 Das Schlüsselwort super	126
13.2 Private Variablen und Methoden	127
13.3 Überschreiben von Methoden	128
14 Abstrakte Klassen, finale Klassen und Interfaces	134
15 Polymorphie	140
15.1 Unterklassen-Polymorphie	140
15.2 Generische Klassen	145
15.3 Wrapper-Klassen	147

0 Allgemeines

Inhalt dieser Vorlesung:

- Einführung in Grundkonzepte der Informatik;
- Einführung in Denkweisen der Informatik;
- Programmieren in **Java** :-)

Voraussetzungen:

Informatik Leistungskurs:

nützlich, aber nicht nötig :-)

Kenntnis einer Programmiersprache:

nützlich, aber nicht nötig :-)

Eigener Rechner:

nützlich, aber nicht nötig :-)

Abstraktes Denken:

unbedingt erforderlich !!!

Neugierde, technisches Interesse:

unbedingt erforderlich !!!

Unsere Philosophie:

Schreiben ist Macht

Eine alte Kulturtechnik:

- um Wissen haltbar zu machen;
- neue Denktechniken zu entwickeln ...

Schreiben als politisches Instrument:

- ⇒ um zu bestimmen, was Realität ist;
- ⇒ um das Handeln von Menschen zu beeinflussen ...
(Vorschriften, Gesetze)

Schreiben als praktisches Werkzeug:

- ⇒ um festzulegen, wie ein Vorgang ablaufen soll:
 - Wie soll das Regal zusammen gebaut werden?
 - Wie multipliziert man zwei Zahlen?
- ⇒ um das Verhalten komplexer mit ihrer Umwelt interagierender Systeme zu realisieren (etwa Motorsteuerungen, Roboter)

⇒ **Programmierung**

1 Vom Problem zum Programm

Ein **Problem** besteht darin, aus einer gegebenen Menge von Informationen eine weitere (bisher unbekannte) Information zu bestimmen.

Ein **Algorithmus** ist ein exaktes **Verfahren** zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.



Ein Algorithmus beschreibt eine Funktion: $f : E \rightarrow A$,
wobei E = zulässige Eingaben, A = mögliche Ausgaben.



Abu Abdallah Muhamed ibn Musa al'Khwaritzmi, etwa 780–835

Achtung:

Nicht jede Abbildung lässt sich durch einen Algorithmus realisieren! (↑ **Berechenbarkeitstheorie**)

Das **Verfahren** besteht i.a. darin, eine Abfolge von **Einzelschritten** der Verarbeitung festzulegen.

Beispiel: Alltagsalgorithmen

Resultat	Algorithmus	Einzelschritte
Pullover	Strickmuster	eine links, eine rechts eine fallen lassen
Kuchen	Rezept	nimm 3 Eier ...
Konzert	Partitur	Noten

Beispiel: Euklidischer Algorithmus

Problem: Seien $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $ggT(a, b)$.

Algorithmus:

1. Falls $a = b$, brich Berechnung ab, es gilt $ggT(a, b) = a$.
Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort.
Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Eigenschaften von Algorithmen:

Abstrahierung: Allgemein löst ein Algorithmus eine **Klasse** von Problem-Instanzen. Die Anwendung auf eine **konkrete** Aufgabe erfordert Abstraktion :-)

Determiniertheit: Algorithmen sind im allgemeinen determiniert, d.h. mit gleichen Eingabedaten und gleichem Startzustand wird stets ein gleiches Ergebnis geliefert. (↑ **nicht-deterministische Algorithmen**, ↑ **randomisierte Algorithmen**)

Fintheit: Die Beschreibung eines Algorithmus besitzt endliche Länge. Die bei der Abarbeitung eines Algorithmus entstehenden Datenstrukturen und Zwischenergebnisse sind endlich.

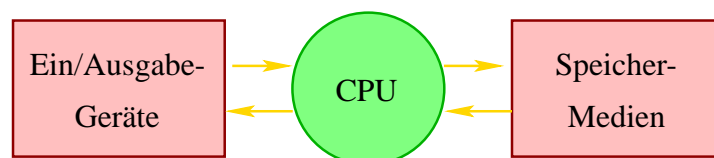
Terminierung: Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen **terminierend**. Meist sind nur terminierende Algorithmen von Interesse. **Ausnahmen:** **Betriebssysteme, reaktive Systeme, ...**

Ein **Programm** ist die **Formulierung** eines Algorithmus in einer **Programmiersprache**. Die Formulierung gestattet (hoffentlich :-)) eine maschinelle Ausführung.

Beachte:

- Es gibt viele Programmiersprachen: **Java, C, Prolog, Fortran, Cobol**
- Eine Programmiersprache ist dann **gut**, wenn
 - die **Programmiererin** in ihr ihre algorithmischen Ideen **natürlich** beschreiben kann, insbesondere selbst später noch versteht, was das Programm tut (oder nicht tut);
 - ein **Computer** das Programm leicht verstehen und **effizient** ausführen kann.

Typischer Aufbau eines Computers:



Ein/Ausgabegeräte (= input/output devices) — ermöglichen Eingabe des Programms und der Daten, Ausgabe der Resultate.

CPU (= central processing unit) — führt Programme aus.

Speicher-Medien (= memory) — enthalten das Programm sowie die während der Ausführung benötigten Daten.

Hardware == physikalische Bestandteile eines Computers.

Merkmale von Computern:

Geschwindigkeit: schnelle Ausführung auch komplexer Programme.

Zuverlässigkeit: Hardwarefehler sind selten :-)
Fehlerhafte Programme bzw. falsche Eingaben sind häufig :-(

Speicherkapazität: riesige Datenmengen speicherbar und schnell zugreifbar.

Kosten: Niedrige laufende Kosten.

Algorithmen wie Programme **abstrahieren** von (nicht so wesentlichen) Merkmalen realer Hardware.

⇒ Annahme eines (nicht ganz realistischen, dafür exakt definierten) **Maschinenmodells** für die Programmierung.

Beliebte Maschinenmodelle:

Turingmaschine: eine Art Lochstreifen-Maschine
(Turing, 1936 :-)

Registermaschine: etwas realistischerer Rechner, allerdings mit i.a. beliebig großen Zahlen und unendlich viel Speicher;

λ-Kalkül: eine minimale ↑funktionale Programmiersprache;

JVM: (Java-Virtual Machine) – die virtuelle Maschine für Java (↑Compilerbau);

...

Zur Definition eines Maschinenmodells benötigen wir:

- Angabe der zulässigen Datenobjekte/Speicherbereiche, auf denen Operationen ausgeführt werden sollen;
- Angabe der verfügbaren Einzelschritte / Aktionen / Elementaroperationen;
- Angabe der Kontrollstrukturen zur Angabe der beabsichtigten Ausführungsreihenfolgen.

Beispiel 1: Turing-Maschine

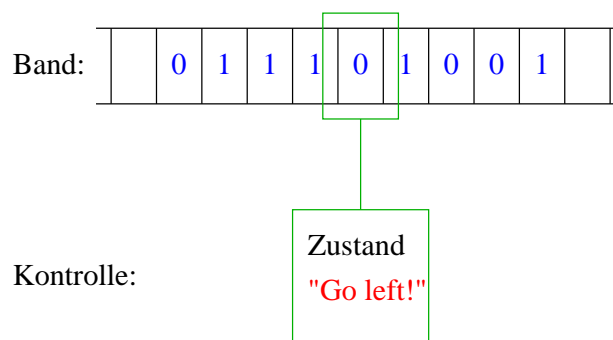


Alan Turing, 1912–1954

Daten der Turing-Maschine: Eine Folge von 0 und 1 und evt. weiterer Symbole wie z.B. “ ” (Blank – Leerzeichen) auf einem **Band** zusammen mit einer Position des “Schreib/Lese”-Kopfs auf dem Band;

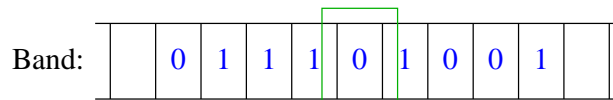
Operationen: Überschreiben des aktuellen Zeichens und Verrücken des Kopfs um eine Position nach rechts oder links;

Kontrollstrukturen: Es gibt eine endliche Menge Q von **Zuständen**. In Abhängigkeit vom aktuellen Zustand und dem gelesenen Zeichen wird die Operation ausgewählt – und der Zustand geändert.

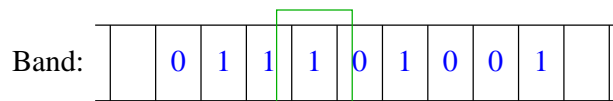


Programm:

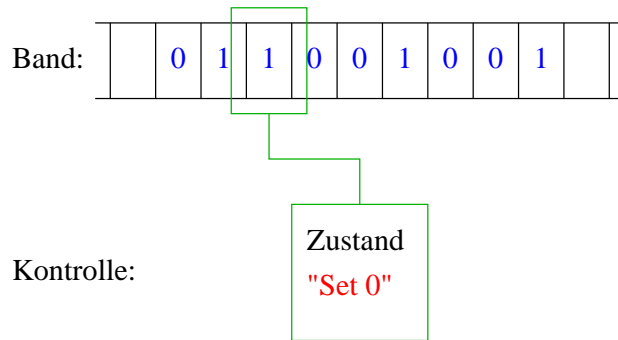
Zustand	Input	Operation	neuer Zustand
“Go left!”	0	0 links	“Set 0”
“Go left!”	1	1 rechts	“Go left!”
“Set 0”	0	0 –	“Stop”
“Set 0”	1	0 links	“Set 0”



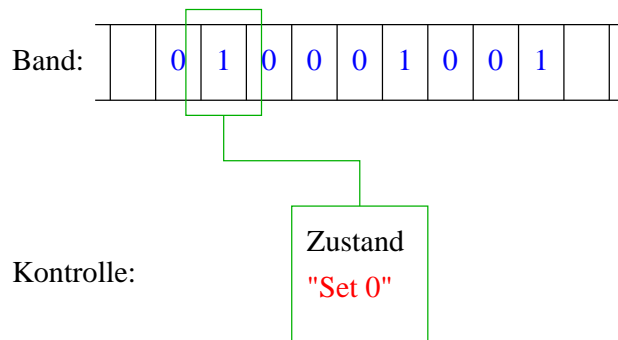
Operation = “Schreibe eine 0 und gehe nach links!”
 neuer Zustand = “Set 0”



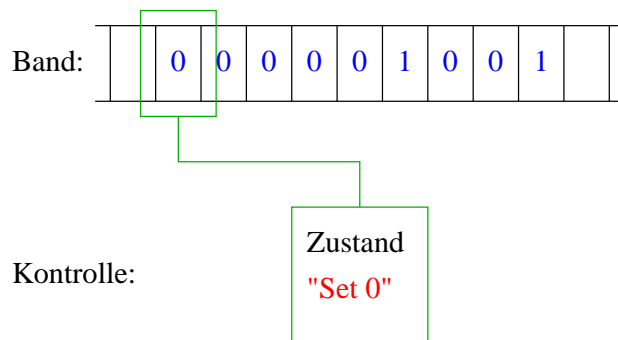
Operation = “Schreibe eine 0 und gehe nach links!”
 neuer Zustand = unverändert



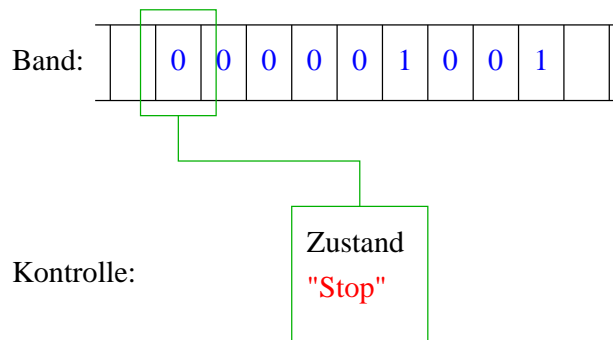
Operation = "Schreibe eine 0 und gehe nach links!"
 neuer Zustand = unverändert



Operation = "Schreibe eine 0 und gehe nach links!"
 neuer Zustand = unverändert



Operation = keine
 neuer Zustand = "Stop"



Ende der Berechnung.

Fazit:

Die Turing-Maschine ist

- ... sehr einfach;
- ... sehr mühsam zu programmieren;
- ... aber nichtsdestoweniger **universell**, d.h. prinzipiell in der Lage **alles** zu berechnen, d.h. insbesondere alles, was ein Desktop-Rechner kann :-)

⇒ beliebtes Hilfsmittel in der ↑**Berechenbarkeitstheorie** und in der ↑**Komplexitätstheorie**.

Beispiel 2: JVM

- minimale Menge von Operationen, Kontroll- sowie Datenstrukturen, um **Java**-Programme auszuführen.
 - ⇒ Um **Java** auf einem Rechner XYZ auszuführen, benötigt man nur einen Simulator für die **JVM**, der auf XYZ läuft.
 - ⇒ **Portabilität!**

Ähnliche abstrakte Maschinen gibt es auch für viele andere Programmiersprachen, z.B. **Pascal**, **SmallTalk**, **Prolog**, **SML**,... ↑**Compilerbau**

2 Eine einfache Programmiersprache

Eine Programmiersprache soll

- Datenstrukturen anbieten;
- Operationen auf Daten erlauben;
- **Kontrollstrukturen** zur Ablaufsteuerung bereit stellen.

Als Beispiel betrachten wir **MiniJava**.

2.1 Variablen

Um Daten zu speichern und auf gespeicherte Daten zugreifen zu können, stellt **MiniJava Variablen** zur Verfügung. Variablen müssen erst einmal eingeführt, d.h. **deklariert** werden.

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den **Namen** `x` und `result` ein.

Erklärung:

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen (“Integers”) speichern sollen.
`int` heißt auch **Typ** der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas “,” getrennt.
- Am Ende steht ein Semikolon “;”.

2.2 Operationen

Die Operationen sollen es gestatten, die Werte von Variablen zu modifizieren. Die wichtigste Operation ist die **Zuweisung**.

Beispiele:

- `x = 7;`
Die Variable `x` erhält den Wert 7.
- `result = x;`
Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.
- `result = x + 19;`
Der Wert der Variablen `x` wird ermittelt, 19 dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.
- `result = x - 5;`
Der Wert der Variablen `x` wird ermittelt, 5 abgezogen und dann das Ergebnis der Variablen `result` zugewiesen.

Achtung:

- **Java** bezeichnet die Zuweisung mit “=” anstelle von “:=” (Erbschaft von **C** ... :-)
- Jede Zuweisung wird mit einem Semikolon “;” beendet.
- In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert **vor** der Zuweisung zu.

Weiterhin benötigen wir Operationen, um Daten (Zahlen) einlesen bzw. ausgeben zu können.

- `x = read();`
Diese Operation liest eine Folge von Zeichen vom Terminal ein und interpretiert sie als eine ganze Zahl, deren Wert sie der Variablen `x` als Wert zu weist.
- `write(42);`
Diese Operation schreibt 42 auf die Ausgabe.
- `write(result);`
Diese Operation bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- `write(x-14);`
Diese Operation bestimmt den Wert der Variablen `x`, subtrahiert 14 und schreibt das Ergebnis auf die Ausgabe.

Achtung:

- Das Argument der `write`-Operation in den Beispielen ist ein `int`.
- Um es ausgeben zu können, muss es in eine **Folge von Zeichen** umgewandelt werden, d.h. einen `String`.

Damit wir auch freundliche Worte ausgeben können, gestatten wir auch **direkt** Strings als Argumente:

- `write("Hello World!");`
... schreibt Hello World! auf die Ausgabe.

2.3 Kontrollstrukturen

Sequenz:

```
int x, y, result;  
x = read();  
y = read();  
result = x + y;  
write(result);
```

- Zu jedem Zeitpunkt wird nur eine Operation ausgeführt.
- Jede Operation wird genau einmal ausgeführt. Keine wird wiederholt, keine ausgelassen.
- Die Reihenfolge, in der die Operationen ausgeführt werden, ist die gleiche, in der sie im Programm stehen (d.h. nacheinander).
- Mit Beendigung der letzten Operation endet die Programm-Ausführung.

⇒ Sequenz alleine erlaubt nur sehr einfache Programme.

Selektion (bedingte Auswahl):

```
int x, y, result;  
x = read();  
y = read();  
if (x > y)  
    result = x - y;  
else  
    result = y - x;  
write(result);
```

- Zuerst wird die Bedingung ausgewertet.
- Ist sie erfüllt, wird die nächste Operation ausgeführt.
- Ist sie nicht erfüllt, wird die Operation nach dem `else` ausgeführt.

Beachte:

- Statt aus einzelnen Operationen können die Alternativen auch aus Statements bestehen:

```
int x;
x = read();
if (x == 0)
    write(0);
else if (x < 0)
    write(-1);
else
    write(+1);
```

- ... oder aus (geklammerten) Folgen von Operationen und Statements:

```
int x, y;
x = read();
if (x != 0) {
    y = read();
    if (x > y)
        write(x);
    else
        write(y);
} else
    write(0);
```

- ... eventuell fehlt auch der else-Teil:

```
int x, y;
x = read();
if (x != 0) {
    y = read();
    if (x > y)
        write(x);
    else
        write(y);
}
```

Auch mit Sequenz und Selektion kann noch nicht viel berechnet werden ... :-)

Iteration (wiederholte Ausführung):

```
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```

- Zuerst wird die Bedingung ausgewertet.
- Ist sie erfüllt, wird der **Rumpf** des while-Statements ausgeführt.
- Nach Ausführung des Rumpfs wird das gesamte while-Statement erneut ausgeführt.
- Ist die Bedingung nicht erfüllt, fährt die Programm-Ausführung hinter dem while-Statement fort.

Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, lässt sich mit Selektion, Sequenz, Iteration, d.h. mithilfe eines **MiniJava**-Programms berechnen :-)

Beweis: ↑ **Berechenbarkeitstheorie.**

Idee:

Eine Turing-Maschine kann alles berechnen...
Versuche, eine Turing-Maschine zu **simulieren!**
MiniJava-Programme sind ausführbares **Java**.
Man muss sie nur geeignet **dekoriern** :-)

MiniJava-Programme sind ausführbares **Java**.
Man muss sie nur geeignet **dekoriern** :-)

Beispiel: Das GGT-Programm

```
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```

Daraus wird das **Java**-Programm:

```
public class GGT extends MiniJava {
    public static void main (String[] args) {

        int x, y;
        x = read();
        y = read();
        while (x != y)
            if (x < y)
                y = y - x;
            else
                x = x - y;
        write(x);

    } // Ende der Definition von main();
} // Ende der Definition der Klasse GGT;
```

Erläuterungen:

- Jedes Programm hat einen **Namen** (hier: GGT).
- Der Name steht hinter dem Schlüsselwort `class` (was eine Klasse, was `public` ist, lernen wir später ... :-)
- Der Datei-Name muss zum Namen des Programms “passen”, d.h. in diesem Fall `GGT.java` heißen.
- Das **MiniJava**-Programm ist der Rumpf des **Hauptprogramms**, d.h. der Funktion `main()`.
- Die Programm-Ausführung eines **Java**-Programms startet stets mit einem Aufruf von dieser Funktion `main()`.
- Die Operationen `write()` und `read()` werden in der Klasse `MiniJava` definiert.
- Durch `GGT extends MiniJava` machen wir diese Operationen innerhalb des `GGT`-Programms verfügbar.

Die Klasse `MiniJava` ist in der Datei `MiniJava.java` definiert:

```

import javax.swing.JOptionPane;
import javax.swing.JFrame;
public class MiniJava {
    public static int read () {
        JFrame f = new JFrame ();
        String s = JOptionPane.showInputDialog (f, "Eingabe:");
        int x = 0; f.dispose ();
        if (s == null) System.exit (0);
        try { x = Integer.parseInt (s.trim ());
        } catch (NumberFormatException e) { x = read (); }
        return x;
    }
    public static void write (String x) {
        JFrame f = new JFrame ();
        JOptionPane.showMessageDialog (f, x, "Ausgabe",
            JOptionPane.PLAIN_MESSAGE);
        f.dispose ();
    }
    public static void write (int x) { write (""+x); }
}

```

... weitere Erläuterungen:

- Die Klasse MiniJava werden wir im Lauf der Vorlesung im Detail verstehen lernen :-)
- Jedes Programm sollte **Kommentare** enthalten, damit man sich selbst später noch darin zurecht findet!
- Ein Kommentar in **Java** hat etwa die Form:

```
// Das ist ein Kommentar!!!
```

- Wenn er sich über mehrere Zeilen erstrecken soll, kann er auch so aussehen:

```
/* Dieser Kommentar geht
"über mehrere Zeilen! */
```

Das Programm GGT kann nun übersetzt und dann ausgeführt werden:

```
seidl> javac GGT.java
seidl> java GGT
```

- Der Compiler **javac** liest das Programm aus den Dateien GGT.java und MiniJava.java ein und erzeugt für sie JVM-Code, den er in den Dateien GGT.class und MiniJava.class ablegt.

- Das Laufzeitsystem `java` liest die Dateien `GGT.class` und `MiniJava.class` ein und führt sie aus.

Achtung:

- `MiniJava` ist sehr primitiv.
- Die Programmiersprache `Java` bietet noch eine Fülle von Hilfsmitteln an, die das Programmieren erleichtern sollen. Insbesondere gibt es
- viele weitere Datenstrukturen (nicht nur `int`) und
- viele weitere Kontrollstrukturen.

... kommt später in der Vorlesung :-)

3 Syntax von Programmiersprachen

Syntax (“Lehre vom Satzbau”):

- formale Beschreibung des Aufbaus der “Worte” und “Sätze”, die zu einer Sprache gehören;
- im Falle einer `Programmier`-Sprache Festlegung, wie Programme aussehen müssen.

Hilfsmittel bei natürlicher Sprache:

- Wörterbücher;
- Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- Ausnahme-Listen;
- Sprach-“Gefühl”.

Hilfsmittel bei Programmiersprachen:

- Listen von `Schlüsselworten` wie `if`, `int`, `else`, `while` ...
- Regeln, wie einzelne Worte (`Tokens`) z.B. `Namen` gebildet werden.

Frage:

Ist `x10` ein zulässiger Name für eine Variable?
oder `_ab$` oder `A#B` oder `0A?B` ...

- Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.

Frage:

Ist ein `while`-Statement im `else`-Teil erlaubt?

- Kontextbedingungen.

Beispiel:

Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

- ⇒ formalisierter als natürliche Sprache
- ⇒ besser für maschinelle Verarbeitung geeignet

Semantik (“Lehre von der Bedeutung”):

- Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine **Bedeutung**, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑**Information**)
- Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine **Bedeutung** :-)

Die Bedeutung eines Programms ist

- alle möglichen **Ausführungen** der beschriebenen Berechnung (↑**operationelle Semantik**); oder
- die definierte **Abbildung** der Eingaben auf die Ausgaben (↑**denotationelle Semantik**).

Achtung!

Ist ein Programm **syntaktisch korrekt**, heißt das noch lange nicht, dass es auch das “richtige” tut, d.h. **semantisch korrekt** ist !!!

3.1 Reservierte Wörter

- `int`
→ Bezeichner für Basis-Typen;
- `if, else, while`
→ Schlüsselwörter aus Programm-Konstrukten;
- `(,), ", ', {, }, ,, ;`
→ Sonderzeichen.

3.2 Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten Zeichen:

```
letter ::= $ | _ | a | ... | z | A | ... | Z
digit  ::= 0 | ... | 9
```

- **letter** und **digit** bezeichnen **Zeichenklassen**, d.h. Mengen von Zeichen, die gleich behandelt werden.

- Das Symbol “|” trennt zulässige Alternativen.
- Das Symbol “...” repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen :-)

Schritt 2: Angabe der Anordnung der Zeichen:

$$\text{name} ::= \text{letter (letter | digit)}^*$$

- Erst kommt ein Zeichen der Klasse **letter**, dann eine (eventuell auch leere) Folge von Zeichen entweder aus **letter** oder aus **digit**.
- Der Operator “*” bedeutet “beliebig ofte Wiederholung” (“weglassen” ist 0-malige Wiederholung :-).
- Der Operator “*” ist ein **Postfix**-Operator. Das heißt, er steht hinter seinem Argument.

Beispiele:

- `_178`
`Das_ist_kein_Name`
`x`
`-`
`$Password$`

... sind legale Namen :-)

- `5ABC`
`!Hallo!`
`x'`
`-178`

... sind keine legalen Namen :-)

Achtung:

Reservierte Wörter sind als Namen verboten !!!

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen **Konstanten**.

Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

$$\text{number} ::= \text{digit digit}^*$$

- Wie sähe die Regel aus, wenn wir führende Nullen verbieten wollen?

Beispiele:

- 17
12490
42
0
00070
... sind alles legale int-Konstanten.
- "Hello World!"
0.5e+128
... sind keine int-Konstanten.

Ausdrücke, die aus Zeichen (-klassen) mithilfe von

| (Alternative)

* (Iteration)

(Konkatenation) sowie

? (Option)

... aufgebaut sind, heißen **reguläre Ausdrücke**¹ (\uparrow Automatentheorie).

Der Postfix-Operator “?” besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

- **(letter letter)***
– alle Wörter gerader Länge (über a, . . . , z, A, . . . , Z);
- **letter* test letter***
– alle Wörter, die das Teilwort test enthalten;
- **_ digit* 17**
– alle Wörter, die mit _ anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- **exp ::= (e|E)(+|-)? digit digit***
float ::= digit digit* exp |
digit* (digit . | . digit) digit* exp?
– alle Gleitkomma-Zahlen ...

Identifizierung von

¹Gelegentlich sind auch ϵ , d.h. das “leere Wort” sowie \emptyset , d.h. die leere Menge zugelassen.

- reservierten Wörtern,
- Namen,
- Konstanten

Ignorierung von

- White Space,
- Kommentaren

... erfolgt in einer **ersten** Phase (↑**Scanner**)

⇒ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter (“Tokens”) aufgeteilt.

In einer **zweiten** Phase wird die **Struktur** des Programms analysiert (↑**Parser**).

3.4 Struktur von Programmen

Programme sind **hierarchisch** aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```

program ::= decl* stmt*
decl    ::= type name ( , name ) * ;
type    ::= int

```

- Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- Eine Deklaration gibt den Typ an, hier: int, gefolgt von einer Komma-separierten Liste von Variablen-Namen.

```

stmt ::= ; | { stmt* } |
      name = expr ; | name = read() ; | write( expr ) ; |
      if ( cond ) stmt |
      if ( cond ) stmt else stmt |
      while ( cond ) stmt

```

- Ein Statement ist entweder “leer” (d.h. gleich ;) oder eine geklammerte Folge von Statements;
- oder eine Zuweisung, eine Lese- oder Schreib-Operation;
- eine (einseitige oder zweiseitige) bedingte Verzweigung;
- oder eine Schleife.

```

expr ::= number | name | ( expr ) |
      unop expr | expr binop expr
unop ::= -
binop ::= - | + | * | / | %

```

- Ein Ausdruck ist eine Konstante, eine Variable oder ein geklammerter Ausdruck
- oder ein unärer Operator, angewandt auf einen Ausdruck,
- oder ein binärer Operator, angewandt auf zwei Argument-Ausdrücke.
- Einziger unärer Operator ist (bisher :-)) die Negation.
- Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganz-zahlige) Division und Modulo.

```

cond ::= true | false | ( cond ) |
      expr comp expr |
      bunop cond | cond bbinop cond
comp ::= == | != | <= | < | >= | >
bunop ::= !
bbinop ::= && | ||

```

- Bedingungen unterscheiden sich dadurch von Ausdrücken, dass ihr Wert nicht vom Typ `int` ist sondern `true` oder `false` (ein **Wahrheitswert** – vom Typ `boolean`).
- Bedingungen sind darum Konstanten, Vergleiche
- oder logische Verknüpfungen anderer Bedingungen.

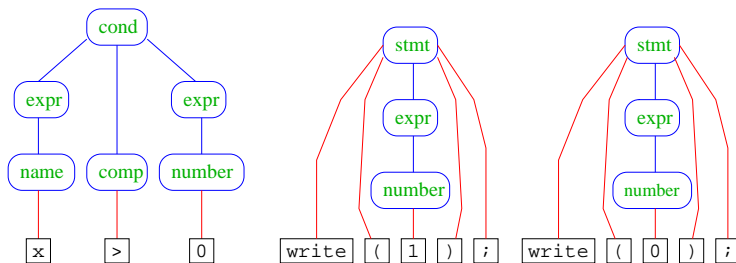
Puh!!! Geschafft ...

Beispiel:

```
int x;  
x = read();  
if (x > 0)  
    write(1);  
else  
    write(0);
```

Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch **Syntax-Bäume**:

Syntax-Bäume für $x > 0$ sowie `write(0);` und `write(1);`

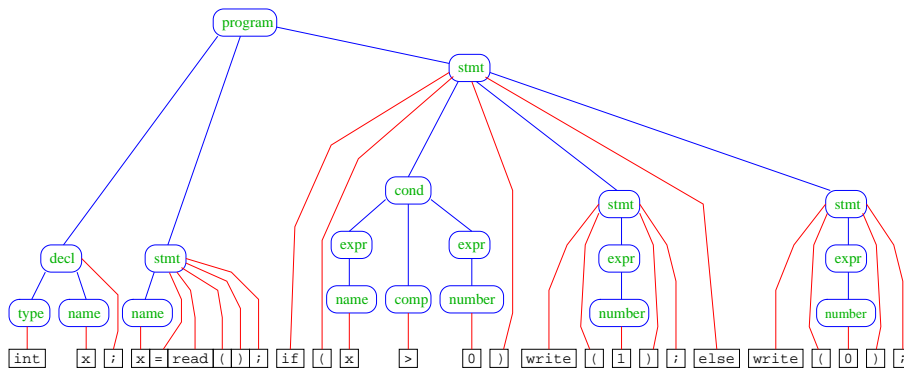


Blätter:

Wörter/Tokens

innere Knoten:

Namen von Programm-Bestandteilen



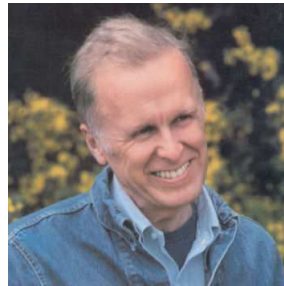
Bemerkungen:

- Die vorgestellte Methode der Beschreibung von Syntax heißt **EBNF**-Notation (**E**xtended **B**ackus **N**aur **F**orm Notation).
- Ein anderer Name dafür ist **erweiterte kontextfreie Grammatik** (↑**Linguistik**, **Automatentheorie**).

- Linke Seiten von Regeln heißen auch **Nicht-Terminale**.
- Tokens heißen auch **Terminale**.



Noam Chomsky,
MIT



John Backus, IBM
Turing Award
(Erfinder von **Fortran**)



Peter Naur,
Turing Award
(Erfinder von **Algol60**)

Achtung:

- Die regulären Ausdrücke auf den rechten Regelseiten können sowohl Terminale wie Nicht-Terminale enthalten.
- Deshalb sind kontextfreie Grammatiken **mächtiger** als reguläre Ausdrücke.

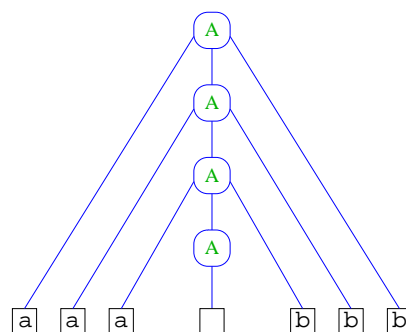
Beispiel:

$$\mathcal{L} = \{e, ab, aabb, aaabbb, \dots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

$$A ::= (a A b)?$$

Syntax-Baum für das Wort aaabbb :

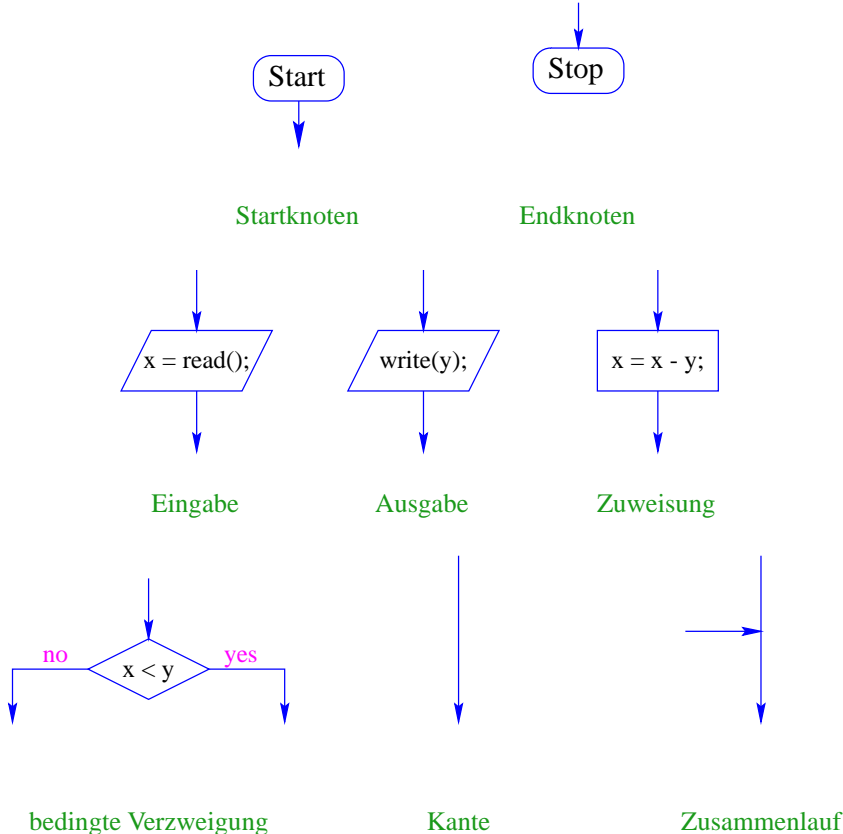


Für \mathcal{L} gibt es aber keinen regulären Ausdruck!!! (↑ **Automatentheorie**)

4 Kontrollfluss-Diagramme

In welcher Weise die Operationen eines Programms nacheinander ausgeführt werden, lässt sich anschaulich mithilfe von **Kontrollfluss-Diagrammen** darstellen.

Ingredienzien:

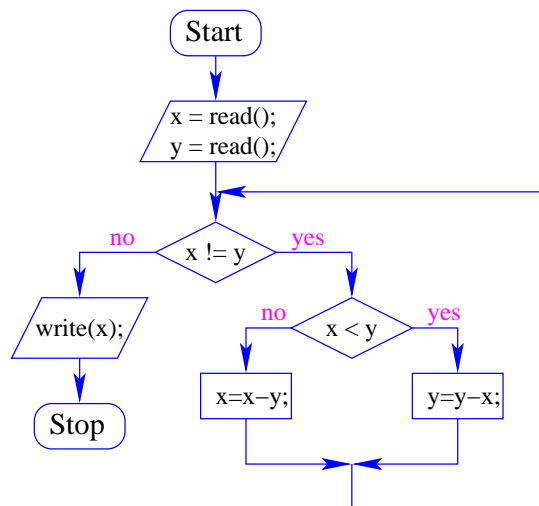


Beispiel:

```

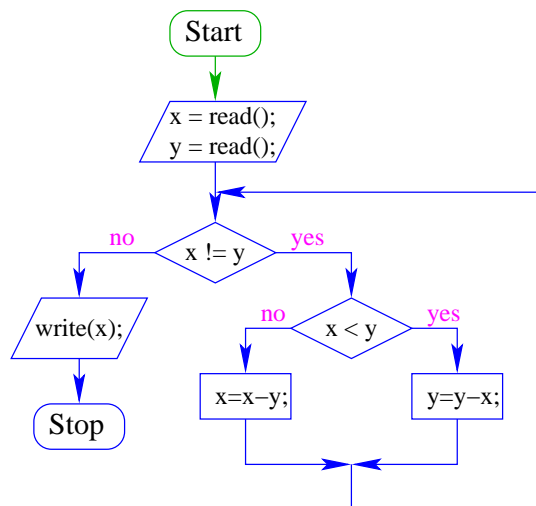
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);

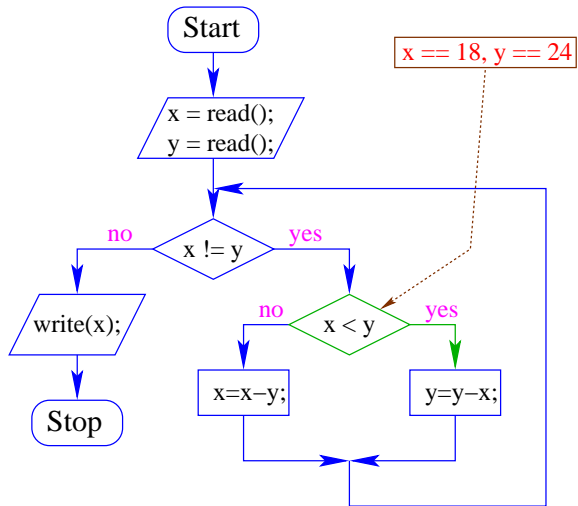
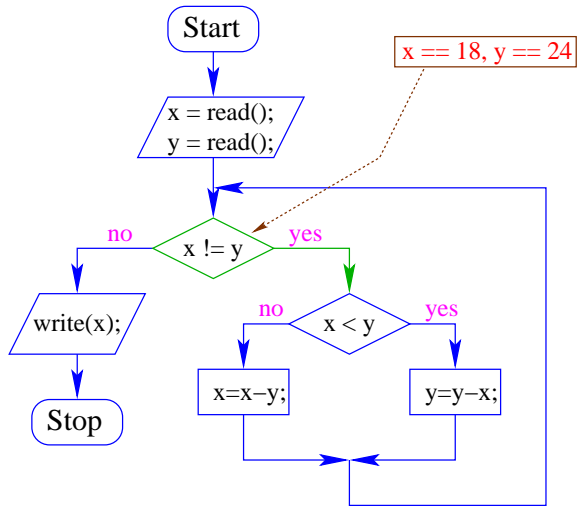
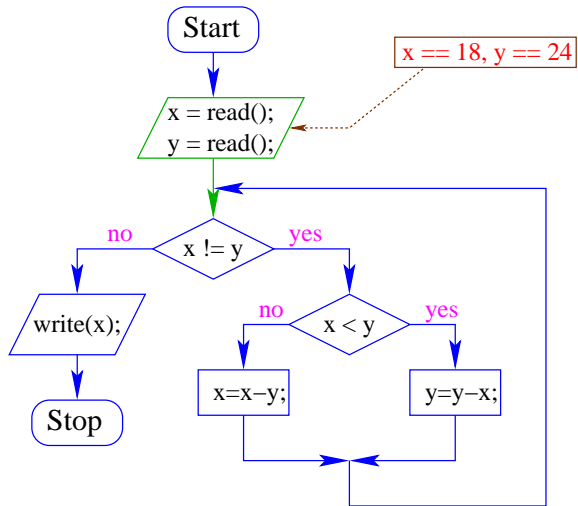
```

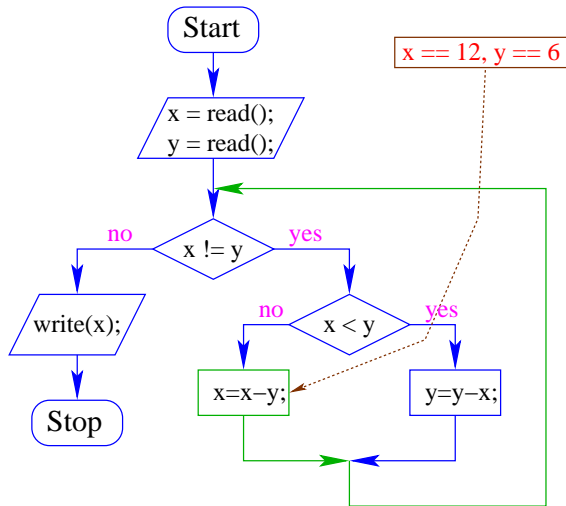
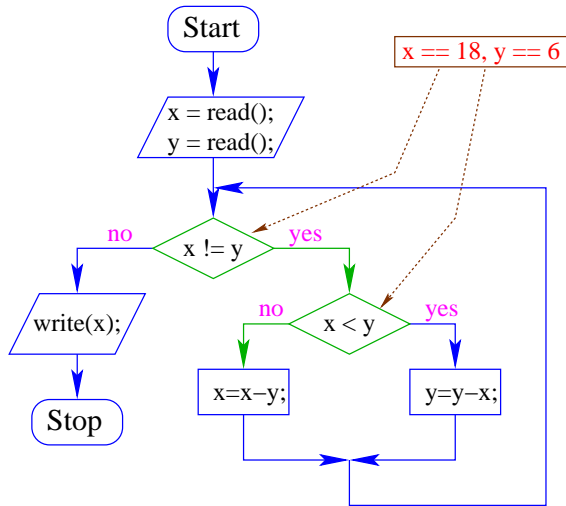
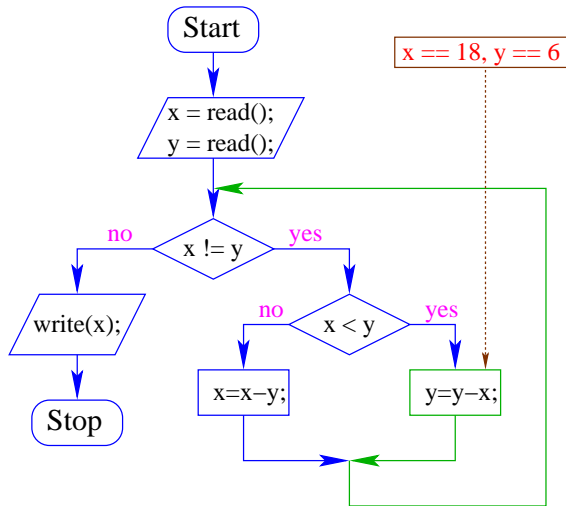


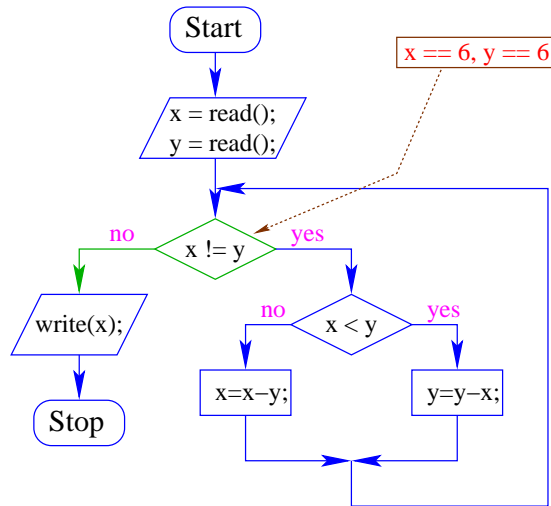
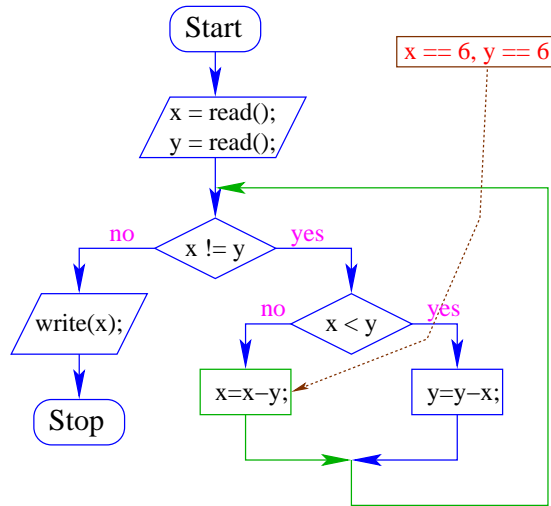
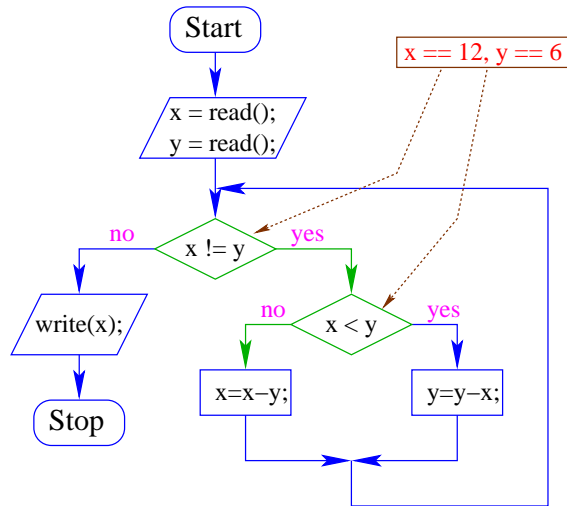
- Die Ausführung des Programms entspricht einem **Pfad** durch das Kontrollfluss-Diagramm vom Startknoten zum Endknoten.
- Die Deklarationen von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer dem Start- und Endknoten) sind die dabei auszuführenden Operationen bzw. auszuwertenden Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung für die aktuellen Werte der Variablen ausgewertet werden.

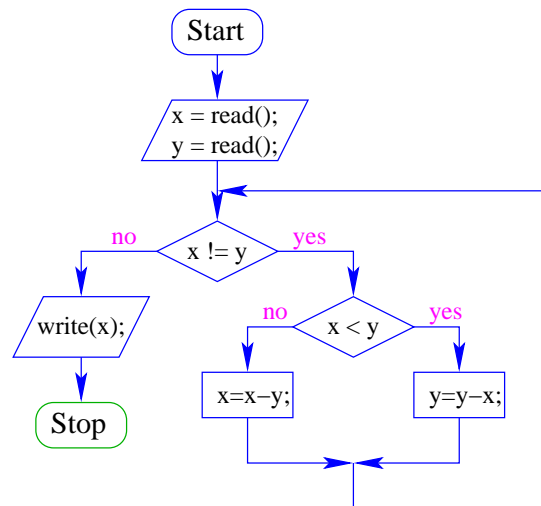
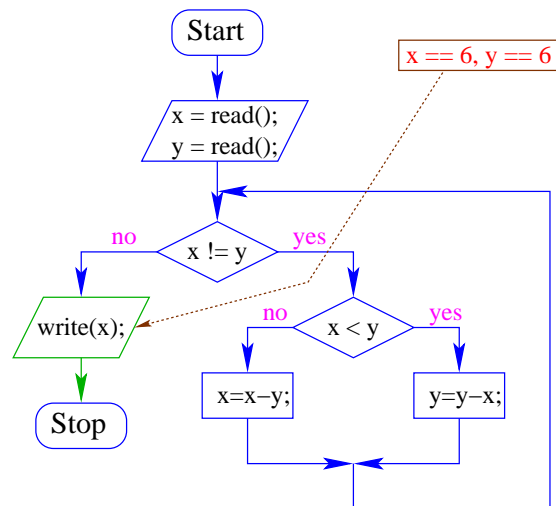
⇒ operationelle Semantik







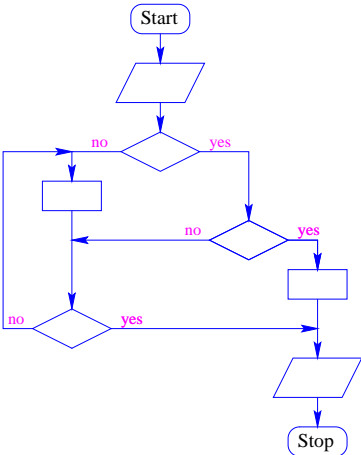




Achtung:

- Zu jedem MiniJava-Programm lässt sich ein Kontrollfluss-Diagramm konstruieren :-)
- die umgekehrte Richtung gilt zwar ebenfalls, liegt aber nicht so auf der Hand.

Beispiel:



5 Mehr Java

Um komfortabel programmieren zu können, brauchen wir

- mehr Datenstrukturen;
- mehr Kontrollstrukturen :-)

5.1 Mehr Basistypen

- Außer `int`, stellt **Java** weitere Basistypen zur Verfügung.
- Zu jedem Basistyp gibt es eine Menge möglicher **Werte**.
- Jeder Wert eines Basistyps benötigt die gleiche Menge **Platz**, um ihn im Rechner zu repräsentieren.
- Der Platz wird in **Bit** gemessen.

(Wie viele Werte kann man mit n Bit darstellen?)

Es gibt **vier** Sorten ganzer Zahlen:

Typ	Platz	kleinster Wert	größter Wert
byte	8	-128	127
short	16	-32 768	32 767
int	32	-2 147 483 648	2 147 483 647
long	64	-9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie `byte` oder `short` spart Platz.

Achtung:
Beispiel:

Java warnt nicht vor Überlauf/Unterlauf !!

```
int x = 2147483647; // grösstes int
x = x+1;
write(x);
```

... liefert `-2147483648` ... :-)

- In realem **Java** kann man bei der Deklaration einer Variablen ihr direkt einen ersten Wert zuweisen (**Initialisierung**).
- Man kann sie sogar (statt am Anfang des Programms) erst an der Stelle deklarieren, an der man sie das erste Mal braucht!

Es gibt **zwei** Sorten von Gleitkomma-Zahlen:

Typ	Platz	kleinster Wert	größter Wert	
float	32	ca. $-3.4e+38$	ca. $3.4e+38$	7 signifikante Stellen
double	64	ca. $-1.7e+308$	ca. $1.7e+308$	15 signifikante Stellen

- Überlauf/Unterlauf liefert die Werte `Infinity` bzw. `-Infinity`.
- Für die Auswahl des geeigneten Typs sollte die gewünschte **Genauigkeit** des Ergebnisses berücksichtigt werden.
- Gleitkomma-Konstanten im Programm werden als **double** aufgefasst :-)
- Zur Unterscheidung kann man an die Zahl `f` (oder `F`) bzw. `d` (oder `D`) anhängen.

... weitere Basistypen:

Typ	Platz	Werte
boolean	1	<code>true</code> , <code>false</code>
char	16	alle Unicode -Zeichen

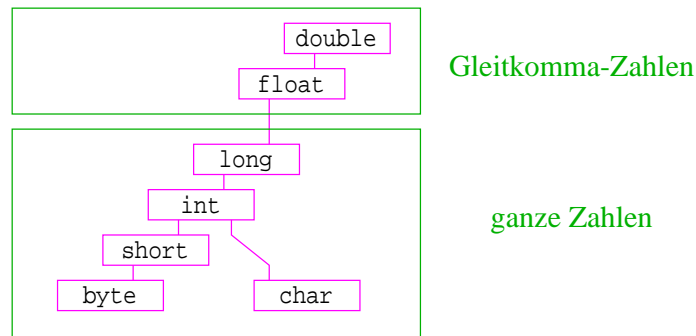
Unicode ist ein Zeichensatz, der alle irgendwo auf der Welt gängigen Alphabete umfasst, also zum Beispiel:

- die Zeichen unserer Tastatur (inklusive Umlaute);
- die chinesischen Schriftzeichen;
- die ägyptischen Hieroglyphen ...

char-Konstanten schreibt man mit Hochkommas: `'A'`, `' ; '`, `'\n'`.

5.2 Mehr über Arithmetik

- Die Operatoren +, -, *, / und % gibt es für **jeden** der aufgelisteten Zahltypen :-)
- Werden sie auf ein Paar von Argumenten **verschiedenen** Typs angewendet, wird automatisch vorher der speziellere in den allgemeineren umgewandelt (**impliziter Type Cast**)
...



Beispiel:

```
short xs = 1;
int x = 999999999;
write(x + xs);
```

... liefert den int-Wert **1000000000 ... :-)**

```
float xs = 1.0f;
int x = 999999999;
write(x + xs);
```

... liefert den float-Wert **1.0E9 ... :-)**

... vorausgesetzt, write() kann Gleitkommazahlen ausgeben :-)

Achtung:

- Das Ergebnis einer Operation auf float kann aus dem Bereich von float herausführen. Dann ergibt sich der Wert Infinity oder -Infinity.
Das gleiche gilt für double.
- Das Ergebnis einer Operation auf Basistypen, die in int enthalten sind (außer char), liefern ein int.
- Wird das Ergebnis einer Variablen zugewiesen, sollte deren Typ dies zulassen :-)
- Mithilfe von **expliziten Type Casts** lässt sich das (evt. unter **Verlust** von Information) stets bewerkstelligen.

Beispiele:

```
(float) 1.7e+308 liefert Infinity
(long) 1.7e+308 liefert 9223372036854775807
                                     (d.h. den größten long-Wert)
(int) 1.7e+308 liefert 2147483647
                                     (d.h. den größten int-Wert)
(short) 1.7e+308 liefert -1
(int) 1.0e9 liefert 1000000000
(int) 1.11 liefert 1
(int) -2.11 liefert -2
```

5.3 Strings

Der Datentyp String für Wörter ist kein Basistyp, sondern eine **Klasse** (dazu kommen wir später :-)

Hier behandeln wir nur drei Eigenschaften:

- Werte vom Typ String haben die Form "Hello World!";
- Man kann Wörter in Variablen vom Typ String abspeichern.
- Man kann Wörter mithilfe des Operators "+" **konkateneren**.

Beispiel:

```
String s0 = "";
String s1 = "Hel";
String s2 = "lo Wo";
String s3 = "rld!";
write(s0 + s1 + s2 + s3);
```


... schreibt `Hello World!` auf die Ausgabe `:-)`

Beachte:

- Jeder Wert in `Java` hat eine Darstellung als `String`.
- Wird der Operator `“+”` auf einen Wert vom Typ `String` und einen anderen Wert `x` angewendet, wird `x` automatisch in seine `String`-Darstellung konvertiert ...

⇒ ... liefert einfache Methode, um `float` oder `double` auszugeben !!!

Beispiel:

```
double x = -0.55e13;
write("Eine Gleitkomma-Zahl: "+x);
```

... schreibt `Eine Gleitkomma-Zahl: -0.55E13` auf die Ausgabe `:-)`

5.4 Felder

Oft müssen viele Werte gleichen Typs gespeichert werden.

Idee:

- Lege sie konsekutiv ab!
- Greife auf einzelne Werte über ihren Index zu!

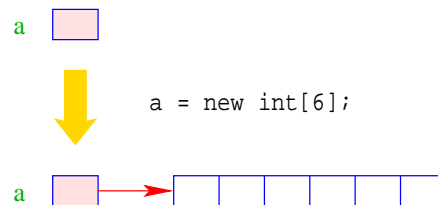
```
Feld:  [17|3|-2|9|0|1]
Index:  0  1  2  3  4  5
```

Beispiel: Einlesen eines Felds

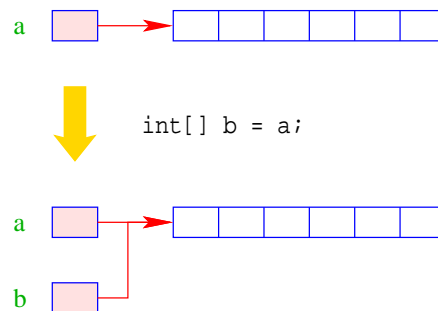
```
int[] a; // Deklaration
int n = read();

a = new int[n];
    // Anlegen des Felds
int i = 0;
while (i < n) {
    a[i] = read();
    i = i+1;
}
```

- `type [] name` ; deklariert eine Variable für ein Feld (**array**), dessen Elemente vom Typ `type` sind.
- Alternative Schreibweise:
`type name []` ;
- Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen **Verweis** darauf zurück:



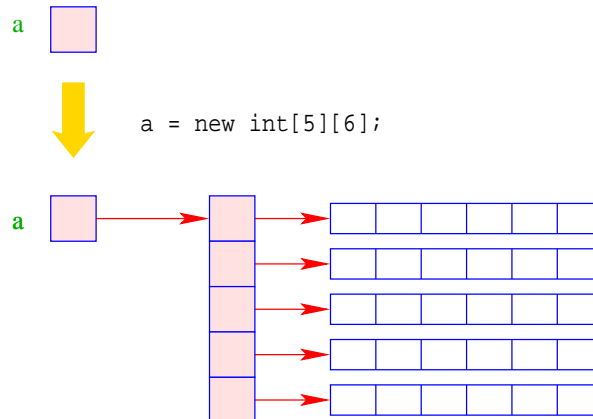
- Der Wert einer Feld-Variable ist also ein Verweis.
- `int[] b = a` ; kopiert den Verweis der Variablen `a` in die Variable `b`:



- Die Elemente eines Felds sind von 0 an durchnummeriert.
- Die Anzahl der Elemente des Felds `name` ist `name.length`.
- Auf das i -te Element des Felds `name` greift man mittels `name[i]` zu.
- Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall $\{0, \dots, \text{name.length}-1\}$ liegt.
- Liegt der Index außerhalb des Intervalls, wird die `ArrayIndexOutOfBoundsException` ausgelöst (**Exceptions**).

Mehrdimensionale Felder

- **Java** unterstützt direkt nur ein-dimensionale Felder.
- Ein zwei-dimensionales Feld ist ein Feld von Feldern ...



5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- Initialisierung des Laufindex;
- `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- Modifizierung des Laufindex am Ende des Rumpfs.

Beispiel (Forts.): Bestimmung des Minimums

```
int result = a[0];
int i = 1;      // Initialisierung
while (i < a.length) {
    if (a[i] < result)
        result = a[i];
    i = i+1;    // Modifizierung
}
write(result);
```

Mithilfe des for-Statements:

```
int result = a[0];
for (int i = 1; i < a.length; ++i)
    if (a[i] < result)
        result = a[i];
write(result);
```

Allgemein:

```
for ( init; cond; modify ) stmt
```

... entspricht:

```
{ init ; while ( cond ) { stmt modify ; } }
```

... wobei ++i äquivalent ist zu i = i+1 :-)

Warnung:

- Die Zuweisung $x = x-1$ ist in Wahrheit ein **Ausdruck**.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variable x erfolgt als **Seiteneffekt**.
- Der Semikolon “;” hinter einem Ausdruck wirft nur den Wert weg ... :-)

⇒ ... fatal für Fehler in Bedingungen ...

```
boolean x = false;
if (x = true)
    write("Sorry! This must be an error ...");
```

- Die Operatoranwendungen ++x und x++ inkrementieren beide den Wert der Variablen x.
- ++x tut das, **bevor** der Wert des Ausdrucks ermittelt wird (**Pre-Increment**).
- x++ tut das, **nachdem** der Wert ermittelt wurde (**Post-Increment**).
- $a[x++] = 7;$ entspricht:

```
a[x] = 7;
x = x+1;
```

- $a[++x] = 7;$ entspricht:

```
x = x+1;
a[x] = 7;
```

Oft möchte man

- Teilprobleme **separat** lösen; und dann
- die Lösung **mehrfach** verwenden;

⇒ Funktionen, Prozeduren

Beispiel: Einlesen eines Felds

```
public static int[] readArray(int n) {
    // n = Anzahl der zu lesenden Elemente
    int[] a = new int[n]; // Anlegen des Felds
    for (int i = 0; i < n; ++i) {
        a[i] = read();
    }
    return a;
}
```

- Die erste Zeile ist der **Header** der Funktion.
- `public` sagt, wo die Funktion verwendet werden darf (↑kommt später :-)
- `static` kommt ebenfalls später :-)
- `int[]` gibt den Typ des Rückgabe-Werts an.
- `readArray` ist der Name, mit dem die Funktion aufgerufen wird.
- Dann folgt (in runden Klammern und komma-separiert) die Liste der **formalen Parameter**, hier: `(int n)`.
- Der Rumpf der Funktion steht in geschwungenen Klammern.
- `return expr` beendet die Ausführung der Funktion und liefert den Wert von `expr` zurück.
- Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von “{” und “}”, sind nur innerhalb dieses Blocks **sichtbar**, d.h. benutzbar (**lokale Variablen**).
- Der Rumpf einer Funktion ist ein Block.
- Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- Bei dem Aufruf `readArray(7)` erhält der formale Parameter `n` den Wert `7`.

Weiteres Beispiel: Bestimmung des Minimums

```
public static int min (int[] a) {
    int result = a[0];
    for (int i = 1; i < a.length; ++i) {
        if (a[i] < result)
            result = a[i];
    }
    return result;
}
```

... daraus basteln wir das **Java-Programm** Min :

```
public class Min extends MiniJava {
    public static int[] readArray (int n) { ... }
    public static int min (int[] a) { ... }
    // Jetzt kommt das Hauptprogramm
    public static void main (String[] args) {
        int n = read();
        int[] a = readArray(n);
        int result = min(a);
        write(result);
    } // end of main()
} // end of class Min
```

- Manche Funktionen, deren Ergebnistyp void ist, geben gar keine Werte zurück – im Beispiel: write() und main(). Diese Funktionen heißen **Prozeduren**.
- Das Hauptprogramm hat immer als Parameter ein Feld args von String-Elementen.
- In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

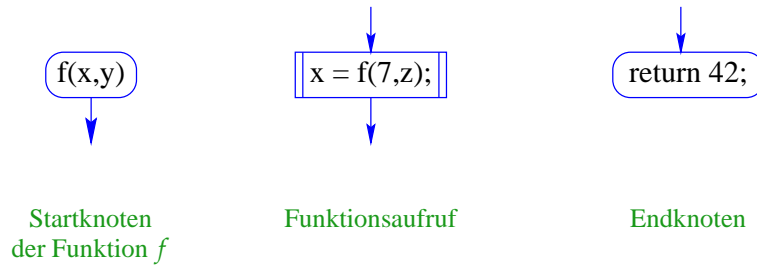
```
public class Test extends MiniJava {
    public static void main (String [] args) {
        write(args[0]+args[1]);
    }
} // end of class Test
```

Dann liefert der Aufruf:

```
java Test "Hel" "lo World!"
```

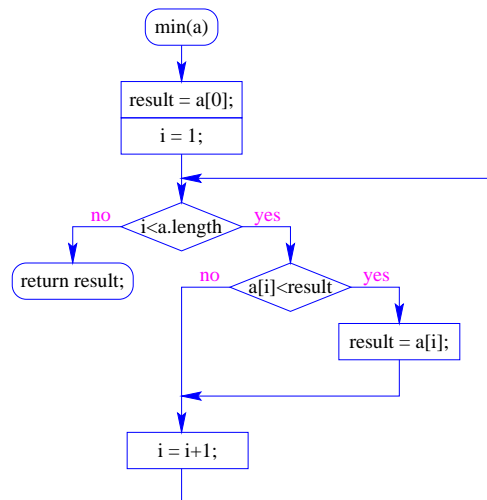
... die Ausgabe: **Hello World!**

Um die Arbeitsweise von Funktionen zu veranschaulichen, erweitern/modifizieren wir die Kontrollfluss-Diagramme:

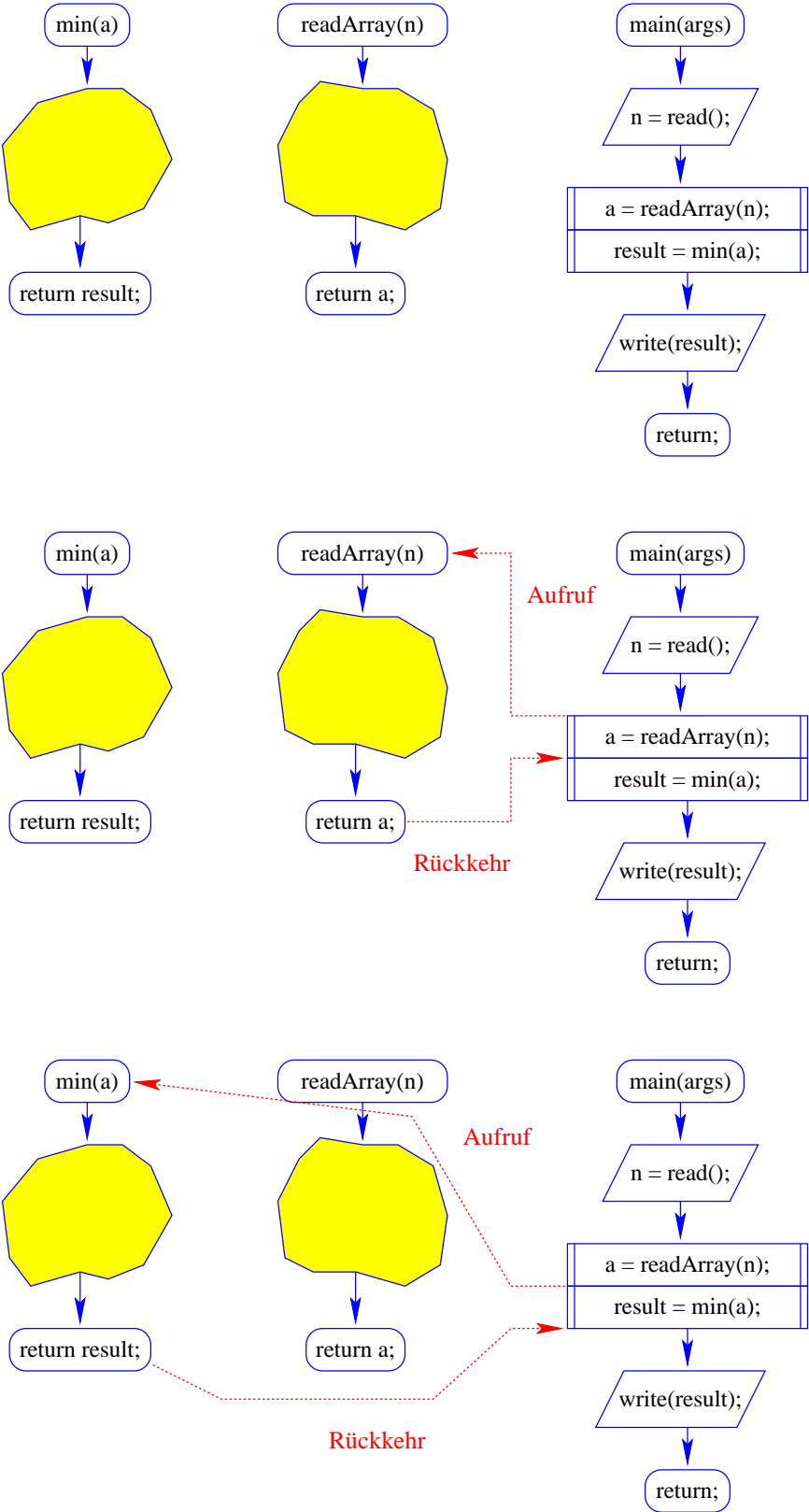


- Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- Ein Aufrufknoten repräsentiert eine Teilberechnung der aufgerufenen Funktion.

Teildiagramm für die Funktion `min()`:



Insgesamt erhalten wir:



6 Eine erste Anwendung: Sortieren

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

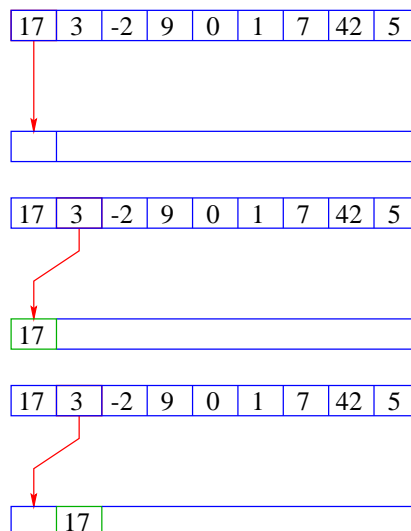
Idee:

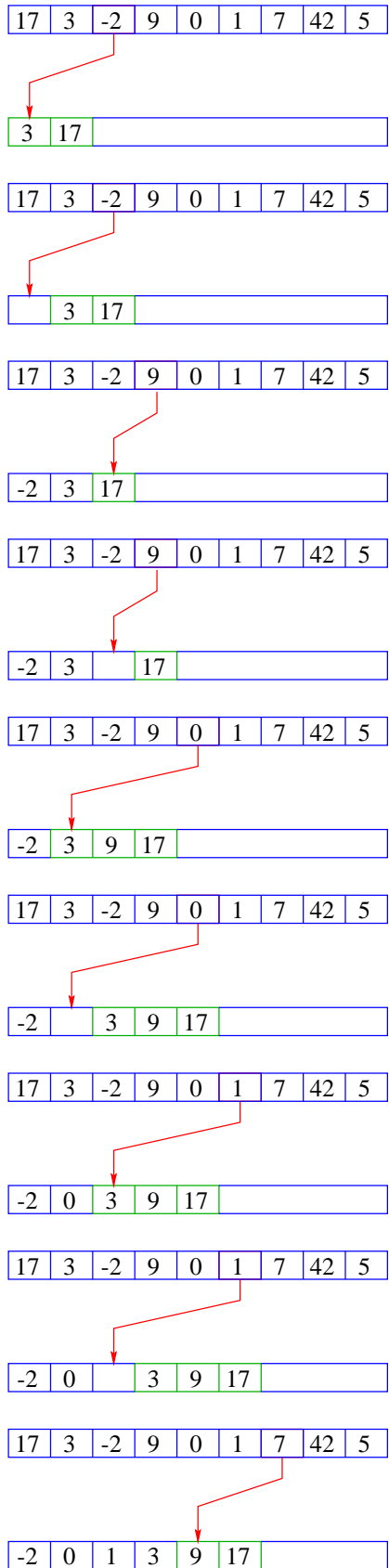
- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

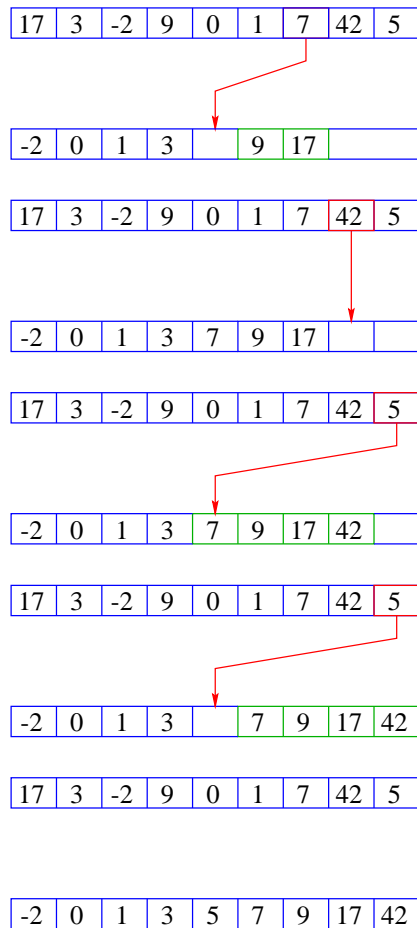
⇒ Sortieren durch **Einfügen** ...

```
public static int[] sort (int[] a) {
    int n = a.length;
    int[] b = new int[n];
    for (int i = 0; i < n; ++i)
        insert (b, a[i], i);
        // b    = Feld, in das eingefügt wird
        // a[i] = einzufügendes Element
        // i    = Anzahl von Elementen in b
    return b;
} // end of sort ()
```

Teilproblem: Wie fügt man ein ???







```

public static void insert (int[] b, int x, int i) {
    int j = locate (b,x,i);
    // findet die Einfügestelle j für x in b
    shift (b,j,i);
    // verschiebt in b die Elemente b[j],...,b[i-1]
    // nach rechts
    b[j] = x;
}

```

Neue Teilprobleme:

- Wie findet man die Einfügestelle?
- Wie verschiebt man nach rechts?

```

public static int locate (int[] b, int x, int i) {
    int j = 0;
    while (j < i && x > b[j]) ++j;
    return j;
}

public static void shift (int[] b, int j, int i) {
    for (int k = i-1; k >= j; --k)
        b[k+1] = b[k];
}

```

- Warum läuft die Iteration in `shift()` von `i-1` **abwärts** nach `j` ?
- Das zweite Argument des Operators `&&` wird nur ausgewertet, sofern das erste `true` ergibt (**Kurzschluss-Auswertung!**). Sonst würde hier auf eine **uninitialisierte** Variable zugegriffen **!!!**
- Das Feld `b` ist (ursprünglich) eine **lokale** Variable von `sort()`.
- Lokale Variablen sind nur im eigenen Funktionsrumpf sichtbar, nicht in den aufgerufenen Funktionen !
- Damit die aufgerufenen Hilfsfunktionen auf `b` zugreifen können, muss `b` explizit als Parameter übergeben werden !

Achtung:

Das Feld wird nicht kopiert. Das Argument ist der Wert der Variablen `b`, also nur eine **Referenz** !

- Deshalb benötigen weder `insert()`, noch `shift()` einen separaten Rückgabewert **:-)**
- Weil das Problem so **klein** ist, würde eine **erfahrene** Programmiererin hier keine Unterprogramme benutzen ...

```

public static int[] sort (int[] a) {
    int[] b = new int[a.length];
    for (int i = 0; i < a.length; ++i) {
        // begin of insert
        int j = 0;
        while (j < i && a[i] > b[j]) ++j;
        // end of locate
        for (int k = i-1; k >= j; --k)
            b[k+1] = b[k];
        // end of shift
        b[j] = a[i];
        // end of insert
    }
    return b;
} // end of sort

```

Diskussion:

- Die Anzahl der ausgeführten Operationen wächst quadratisch in der Größe des Felds
a :-)
- Glücklicherweise gibt es Sortier-Verfahren, die eine bessere Laufzeit haben (↑[Algorithmen und Datenstrukturen](#)).

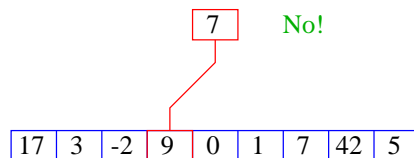
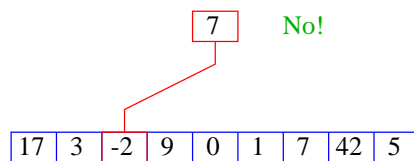
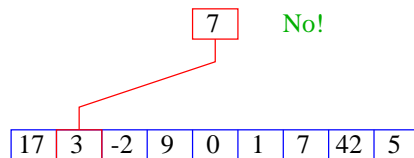
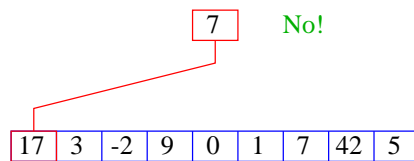
7 Eine zweite Anwendung: Suchen

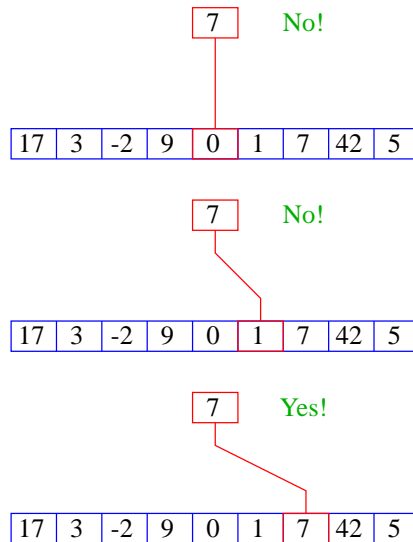
Nehmen wir an, wir wollen herausfinden, ob das Element 7 in unserem Feld a enthalten ist.

Naives Vorgehen:

- Wir vergleichen 7 der Reihe nach mit den Elementen $a[0]$, $a[1]$, usw.
- Finden wir ein i mit $a[i] == 7$, geben wir i aus.
- Andernfalls geben wir -1 aus: "Sorry, gibt's leider nicht :-(")

```
public static int find (int[] a, int x) {  
    int i = 0;  
    while (i < a.length && a[i] != x)  
        ++i;  
    if (i == a.length)  
        return -1;  
    else  
        return i;  
}
```





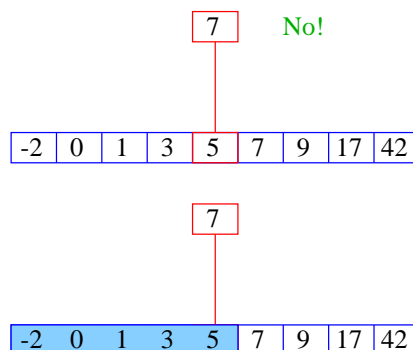
- Im Beispiel benötigen wir 7 Vergleiche.
- Im schlimmsten Fall benötigen wir bei einem Feld der Länge n sogar n Vergleiche :-((
- Kommt 7 tatsächlich im Feld vor, benötigen wir selbst im **Durchschnitt** $(n + 1)/2$ viele Vergleiche :-(((

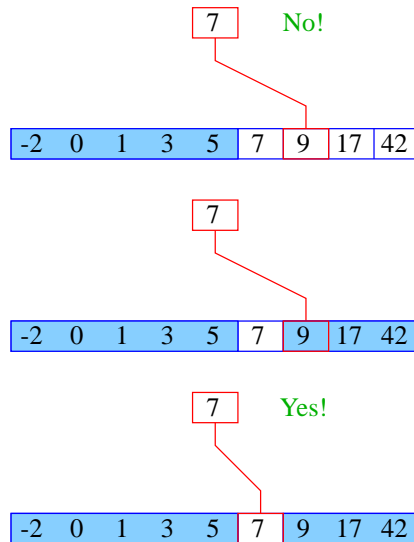
Geht das nicht besser ???

Idee:

- Sortiere das Feld.
- Vergleiche 7 mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist 7 kleiner, brauchen wir nur noch links weitersuchen.
- Ist 7 größer, brauchen wir nur noch rechts weiter suchen.

⇒ binäre Suche ...





- D.h. wir benötigen gerade mal **drei** Vergleiche.
- Hat das sortierte Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche.

Idee:

Wir führen eine Hilfsfunktion

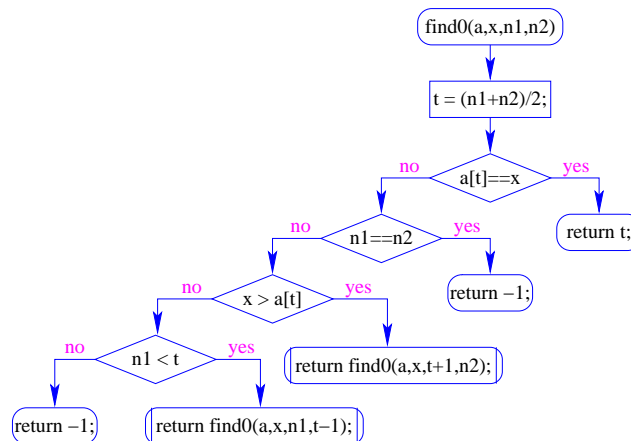
```
public static int find0 (int[] a, int x, int n1, int n2)
```

ein, die im Intervall $[n1, n2]$ sucht. Damit:

```
public static int find (int[] a, int x) {
    return find0 (a, x, 0, a.length-1);
}

public static int find0 (int[] a, int x, int n1, int n2) {
    int t = (n1+n2)/2;
    if (a[t] == x)
        return t;
    else if (n1 == n2)
        return -1;
    else if (x > a[t])
        return find0 (a,x,t+1,n2);
    else if (n1 < t)
        return find0 (a,x,n1,t-1);
    else return -1;
}
```

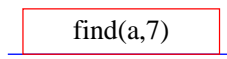

Das Kontrollfluss-Diagramm für `find0()`:



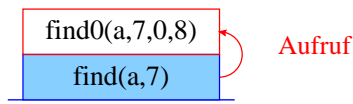
Achtung:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können :-)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen **rekursiv**.

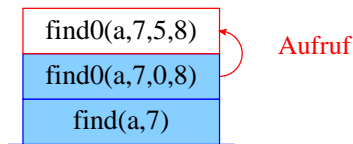
Ausführung:



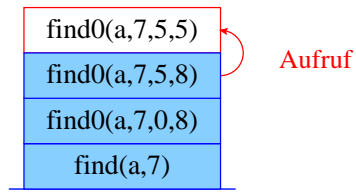
Ausführung:



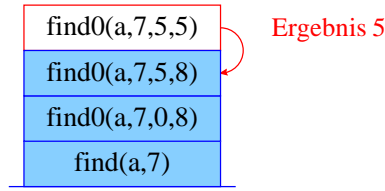
Ausführung:



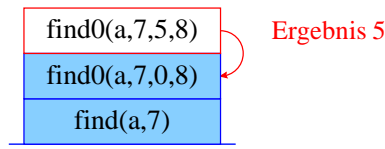
Ausführung:



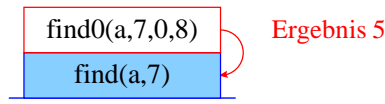
Ausführung:



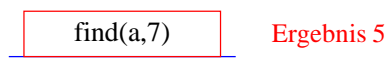
Ausführung:



Ausführung:



Ausführung:



- Die Verwaltung der Funktionsaufrufe erfolgt nach dem **LIFO**-Prinzip (**L**ast-**I**n-**F**irst-**O**ut).
- Eine Datenstruktur, die nach diesem Stapel-Prinzip verwaltet wird, heißt auch **Keller** oder **Stack**.
- Aktiv ist jeweils nur der oberste/letzte Aufruf.
- **Achtung:** es kann zu einem Zeitpunkt mehrere weitere **inaktive** Aufrufe der selben Funktion geben !!!

Um zu **beweisen**, dass `find0()` terminiert, beobachten wir:

1. Wird `find0()` für ein ein-elementiges Intervall $[n, n]$ aufgerufen, dann terminiert der Funktionsaufruf direkt.
2. wird `find0()` für ein Intervall $[n1, n2]$ aufgerufen mit mehr als einem Element, dann terminiert der Aufruf entweder direkt (weil x gefunden wurde), oder `find0()` wird mit einem Intervall aufgerufen, das **echt** in $[n1, n2]$ enthalten ist, genauer: sogar maximal die Hälfte der Elemente von $[n1, n2]$ enthält.

⇒ ähnliche Technik wird auch für andere rekursive Funktionen angewandt.

Beobachtung:

- Das Ergebnis eines Aufrufs von `find0()` liefert **direkt** das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt **End-** oder **Tail-Rekursion**.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden ...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern **ersetze** den bereits dort liegenden Aufruf !

Verbesserte Ausführung:

`find(a,7)`

Verbesserte Ausführung:

`find0(a,7,0,8)`

Verbesserte Ausführung:

`find0(a,7,5,8)`

Verbesserte Ausführung:

find0(a,7,5,5)

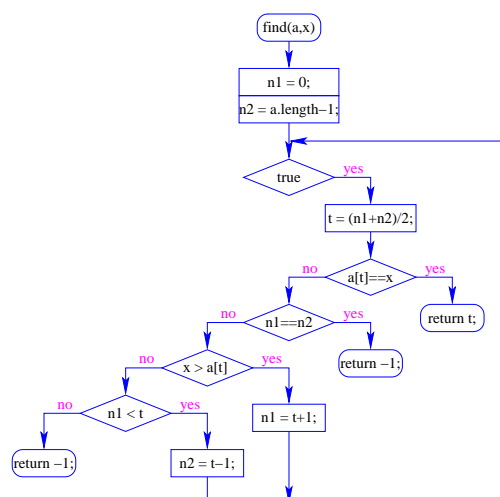
Verbesserte Ausführung:

find0(a,7,5,5) Ergebnis: 5

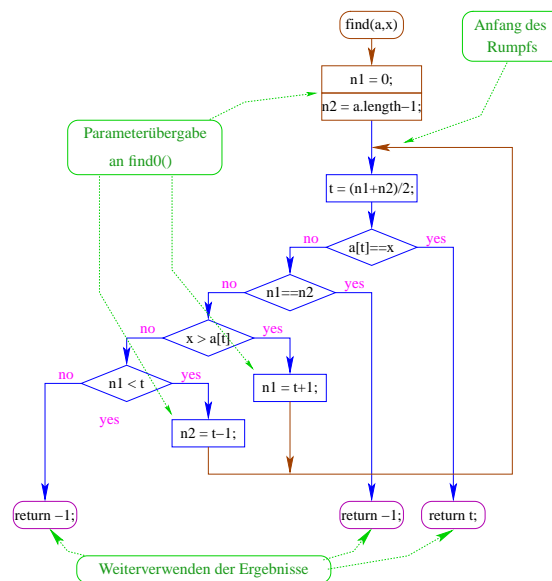
⇒ end-Rekursion kann durch **Iteration** (d.h. eine normale Schleife) ersetzt werden ...

```
public static int find (int[] a, int x) {  
    int n1 = 0;  
    int n2 = a.length-1;  
    while (true) {  
        int t = (n2+n1)/2;  
        if (x == a[t]) return t;  
        else if (n1 == n2) return -1;  
        else if (x > a[t]) n1 = t+1;  
        else if (n1 < t) n2 = t-1;  
        else return -1;  
    } // end of while  
} // end of find
```

Das Kontrollfluss-Diagramm:



- Die Schleife wird hier alleine durch die return-Anweisungen verlassen.
- Offenbar machen Schleifen mit **mehreren** Ausgängen Sinn.
- Um eine Schleife zu verlassen, ohne gleich ans Ende der Funktion zu springen, kann man das break-Statement benutzen.
- Der Aufruf der end-rekursiven Funktion wird ersetzt durch:
 1. Code zur Parameter-Übergabe;
 2. einen **Sprung** an den Anfang des Rumpfs.
- Aber **Achtung**, wenn die Funktion an **mehreren** Stellen benutzt wird !!!
(Was ist das Problem ?-)

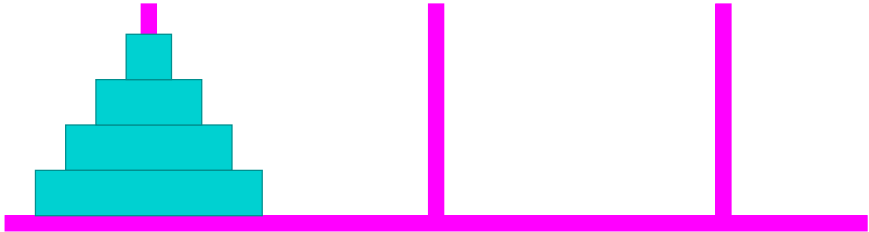


Bemerkung:

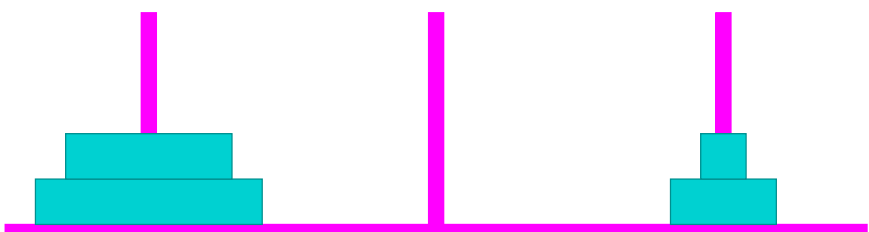
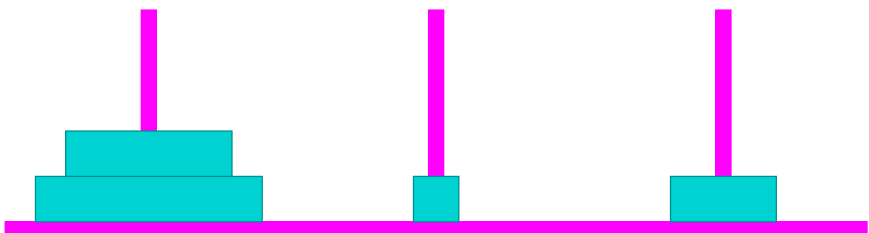
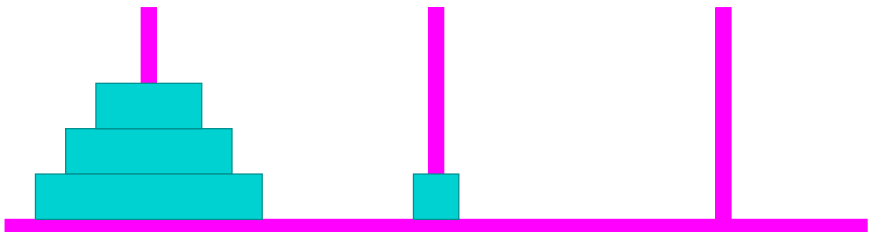
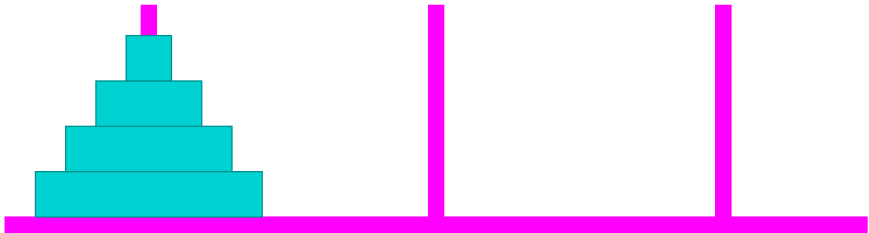
- Jede Rekursion lässt sich beseitigen, indem man den Aufruf-Keller **explizit** verwaltet.
- Nur im Falle von End-Rekursion kann man auf den Keller verzichten.
- Rekursion ist trotzdem nützlich, weil rekursive Programme oft **leichter zu verstehen** sind als äquivalente Programme ohne Rekursion ...

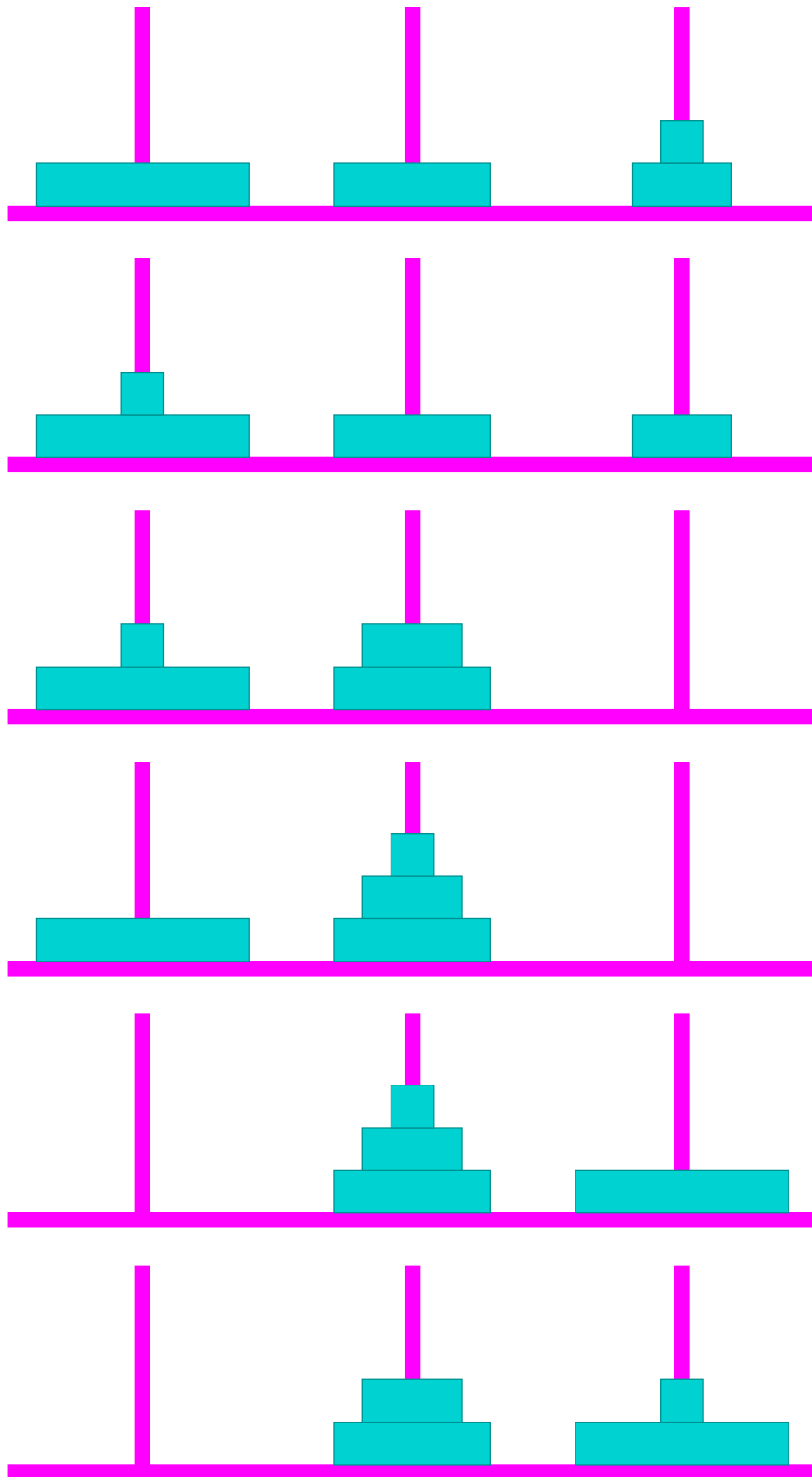
8 Die Türme von Hanoi

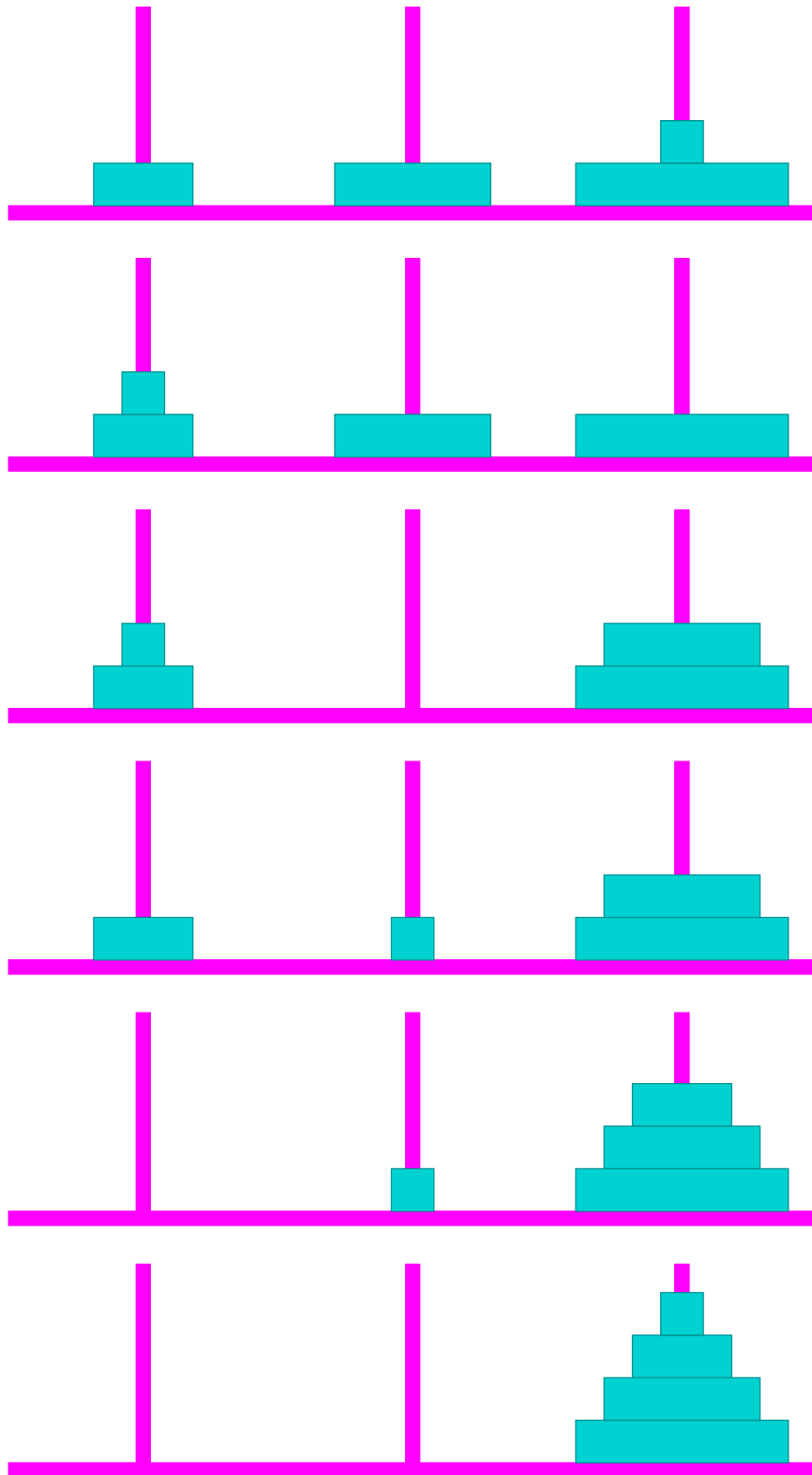
Problem:



- Bewege den Stapel von links nach rechts!
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer Ring auf einen kleineren gelegt werden.







Idee:

- Versetzen eines Turms der Höhe $h = 0$ ist einfach: wir tun nichts.
- Versetzen eines Turms der Höhe $h > 0$ von Position a nach Position b zerlegen wir in drei Teilaufgaben:
 1. Versetzen der oberen $h - 1$ Scheiben auf den freien Platz;
 2. Versetzen der untersten Scheibe auf die Zielposition;
 3. Versetzen der zwischengelagerten Scheiben auf die Zielposition.
- Versetzen eines Turms der Höhe $h > 0$ erfordert also zweimaliges Versetzen eines Turms der Höhe $h - 1$.

```
public static void move (int h, byte a, byte b) {  
    if (h > 0) {  
        byte c = free (a,b);  
        move (h-1,a,c);  
        System.out.print ("\tmove "+a+" to "+b+"\n");  
        move (h-1,c,b);  
    }  
}
```

Bleibt die Ermittlung des freien Platzes ...

	0	1	2
0		2	1
1	2		0
2	1	0	

Offenbar hängt das Ergebnis nur von der **Summe** der beiden Argumente ab ...

	0	1	2
0		1	2
1	1		3
2	2	3	

Um solche Tabellen **leicht** implementieren zu können, stellt **Java** das `switch`-Statement zur Verfügung:

```

public static byte free (byte a, byte b) {
    switch (a+b) {
        case 1:    return 2;
        case 2:    return 1;
        case 3:    return 0;
        default:   return -1;
    }
}

```

Allgemeine Form eines switch-Statements:

```

switch ( expr ) {
    case const0 :  ss0 ( break; ) ?
    case const1 :  ss1 ( break; ) ?
        ...
    case constk-1 :  ssk-1 ( break; ) ?
    ( default:  ssk ) ?
}

```

- `expr` sollte eine ganze Zahl (oder ein char) sein.
- Die `consti` sind ganz-zahlige Konstanten.
- Die `ssi` sind die alternativen Statement-Folgen.
- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren :-)
- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren :-)

Eine **einfachere Lösung** in unserem Fall ist :

```

public static byte free (byte a, byte b) {
    return (byte) (3-(a+b));
}

```

Für einen Turm der Höhe $h = 4$ liefert das:

```
move 0 to 1
move 0 to 2
move 1 to 2
move 0 to 1
move 2 to 0
move 2 to 1
move 0 to 1
move 0 to 2
move 1 to 2
move 1 to 0
move 2 to 0
move 1 to 2
move 0 to 1
move 0 to 2
move 1 to 2
```

Bemerkungen:

- `move ()` ist rekursiv, aber nicht end-rekursiv.
- Sei $N(h)$ die Anzahl der ausgegebenen Moves für einen Turm der Höhe $h \geq 0$. Dann ist

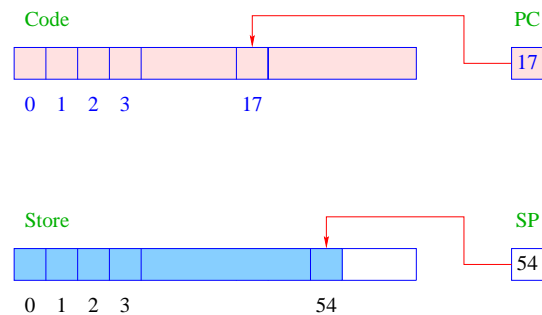
$$\begin{aligned} N(0) &= 0 && \text{und für } h > 0, \\ N(h) &= 1 + 2 \cdot N(h - 1) \end{aligned}$$

- Folglich ist $N(h) = 2^h - 1$.
- Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden ... (wie könnte der aussehen? :-)

Hinweis: Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter ...

9 Von MiniJava zur JVM

Architektur der JVM:

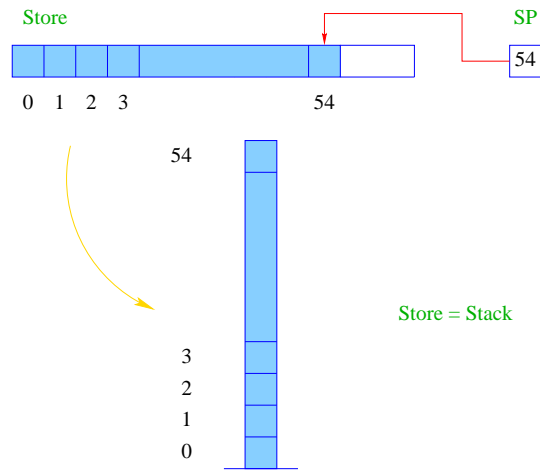


- Code** = enthält JVM-Programm;
jede Zelle enthält einen Befehl;
- PC** = Program Counter –
zeigt auf nächsten auszuführenden Befehl;
- Store** = Speicher für Daten;
jede Zelle kann einen Wert aufnehmen;
- SP** = Stack-Pointer –
zeigt auf oberste belegte Zelle.

Achtung:

- Programm wie Daten liegen im Speicher – aber in verschiedenen Abschnitten.
- Programm-Ausführung holt nacheinander Befehle aus Code und führt die entsprechenden Operationen auf Store aus.

Konvention:



Befehle der JVM:

int-Operatoren:	NEG, ADD, SUB, MUL, DIV, MOD
boolean-Operatoren:	NOT, AND, OR
Vergleichs-Operatoren:	LESS, LEQ, EQ, NEQ
Laden von Konstanten:	CONST i, TRUE, FALSE
Speicher-Operationen:	LOAD i, STORE i
Sprung-Befehle:	JUMP i, FJUMP i
IO-Befehle:	READ, WRITE
Reservierung von Speicher:	ALLOC i
Beendigung des Programms:	HALT

Ein Beispiel-Programm:

```

        ALLOC 2      LOAD 0      B:  LOAD 0
        READ         LOAD 1      LOAD 1
        STORE 0     LESS        SUB
        READ         FJUMP B     STORE 0
        STORE 1     LOAD 1      C:  JUMP A
A:  LOAD 0         LOAD 0      D:  LOAD 1
    LOAD 1         SUB         WRITE
    NEQ           STORE 1     HALT
    FJUMP D       JUMP C
    
```

- Das Programm berechnet den GGT :-)
- Die Marken (Labels) A, B, C, D bezeichnen symbolisch die Adressen der zugehörigen Befehle:

A = 5
 B = 18
 C = 22
 D = 23

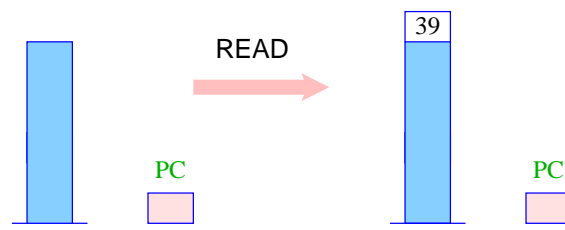
- ... können vom Compiler leicht in die entsprechenden Adressen umgesetzt werden (wir benutzen sie aber, um uns besser im Programm zurechtzufinden :-)

Bevor wir erklären, wie man MiniJava in JVM-Code übersetzt, erklären wir, was die einzelnen Befehle bewirken.

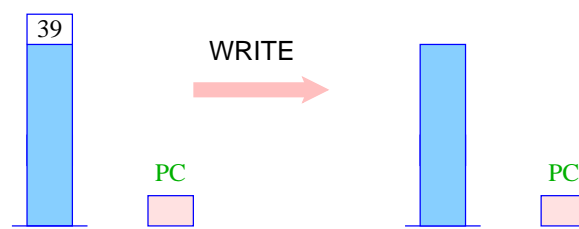
Idee:

- Befehle, die Argumente benötigen, erwarten sie am oberen Ende des Stack.
- Nach ihrer Benutzung werden die Argumente vom Stack herunter geworfen.
- Mögliche Ergebnisse werden oben auf dem Stack abgelegt.

Betrachten wir als Beispiele die IO-Befehle READ und WRITE.



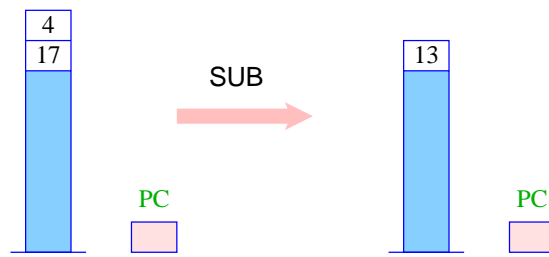
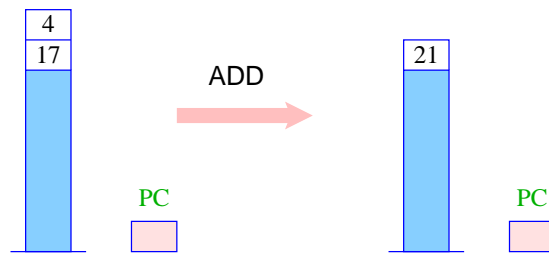
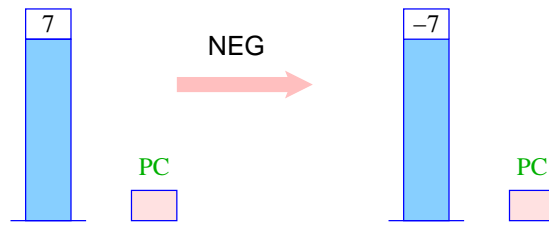
... falls 39 eingegeben wurde



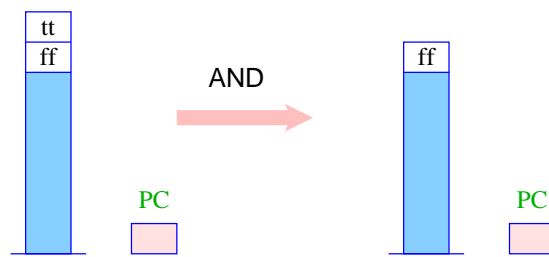
... wobei 39 ausgegeben wird

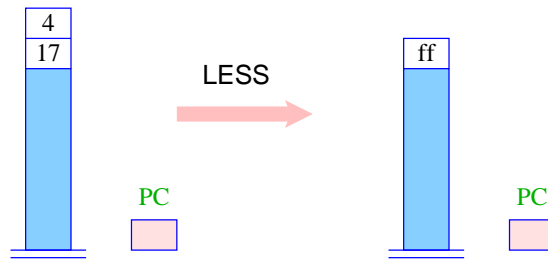
Arithmetik

- Unäre Operatoren modifizieren die oberste Zelle.
- Binäre Operatoren verkürzen den Stack.



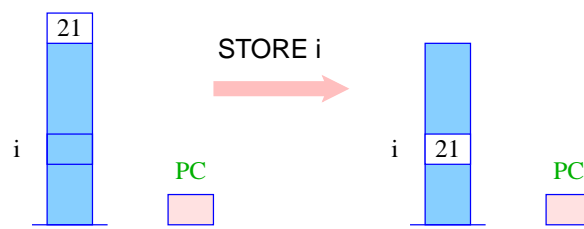
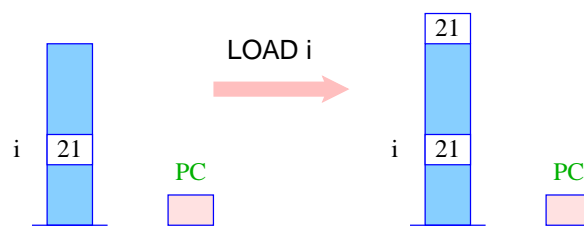
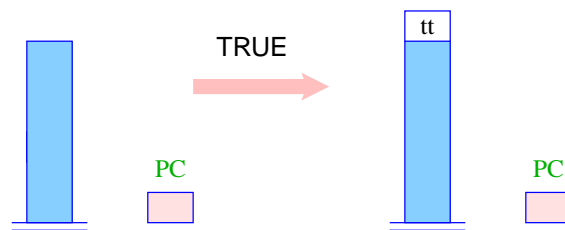
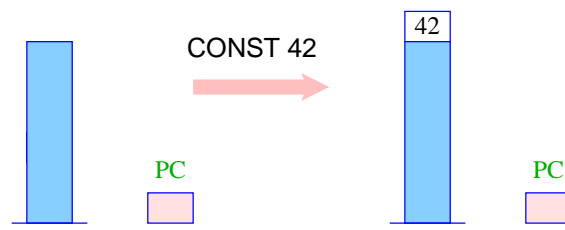
- Die übrigen arithmetischen Operationen MUL, DIV, MOD funktionieren völlig analog.
- Die logischen Operationen NOT, AND, OR ebenfalls – mit dem Unterschied, dass sie statt mit ganzen Zahlen, mit Intern-Darstellungen von true und false arbeiten (hier: “tt” und “ff”).
- Auch die Vergleiche arbeiten so – nur konsumieren sie ganze Zahlen und liefern einen logischen Wert.





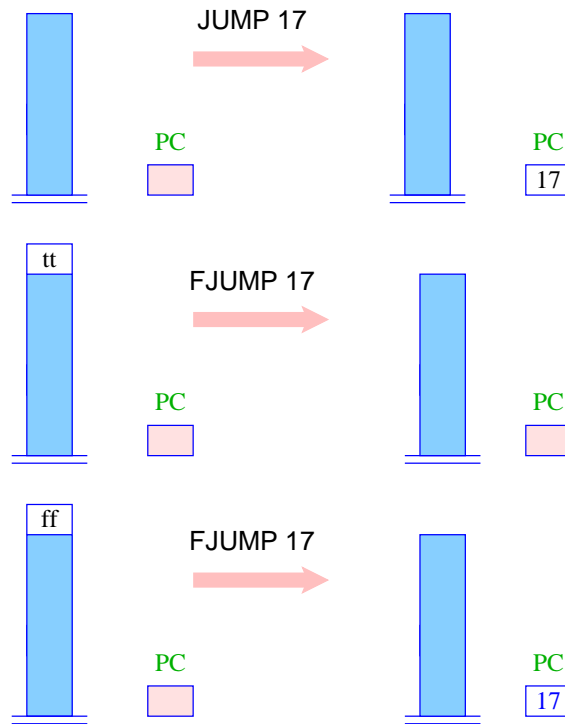
Laden und Speichern

- Konstanten-Lade-Befehle legen einen neuen Wert oben auf dem Stack ab.
- LOAD i legt dagegen den Wert aus der i -ten Zelle oben auf dem Stack ab.
- STORE i speichert den obersten Wert in der i -ten Zelle ab.



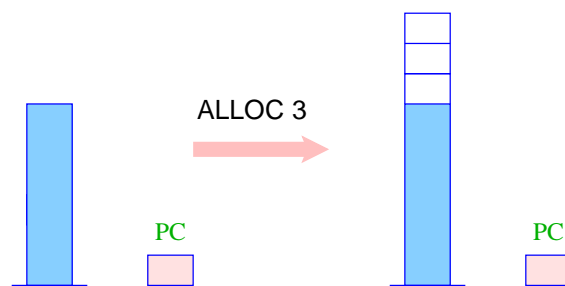
Sprünge

- Sprünge verändern die Reihenfolge, in der die Befehle abgearbeitet werden, indem sie den **PC** modifizieren.
- Ein unbedingter Sprung überschreibt einfach den alten Wert des **PC** mit einem neuen.
- Ein bedingter Sprung tut dies nur, sofern eine geeignete Bedingung erfüllt ist.



Allokierung von Speicherplatz

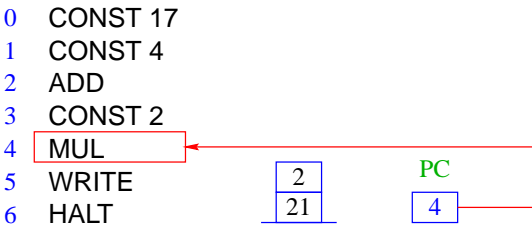
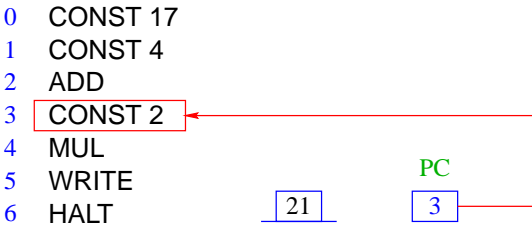
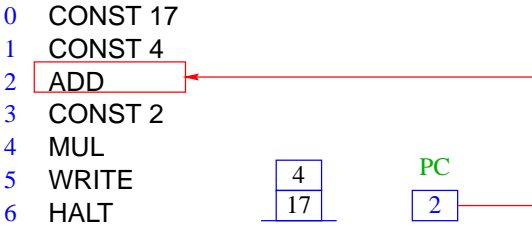
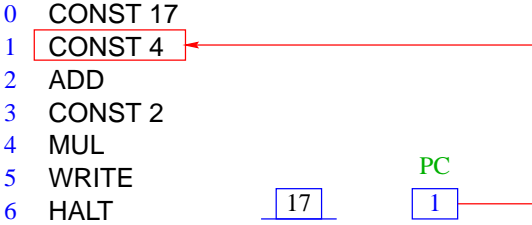
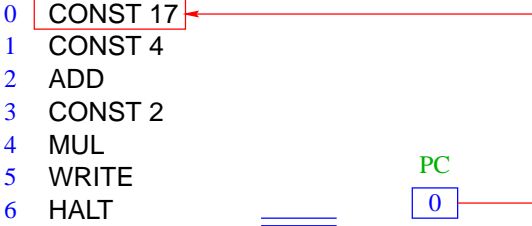
- Wir beabsichtigen, jeder Variablen unseres **MiniJava**-Programms eine Speicher-Zelle zuzuordnen.
- Um Platz für i Variablen zu schaffen, muss der **SP** einfach um i erhöht werden.
- Das ist die Aufgabe von **ALLOC i** .



Ein Beispiel-Programm:

```

CONST 17
CONST 4
ADD
CONST 2
MUL
WRITE
HALT
    
```





Übersetzung von $\text{int } x_0, \dots, x_{n-1};$ = ALLOC n

9.2 Übersetzung von Ausdrücken

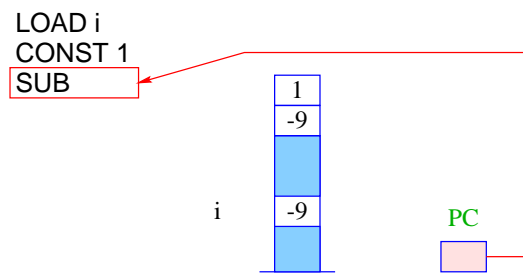
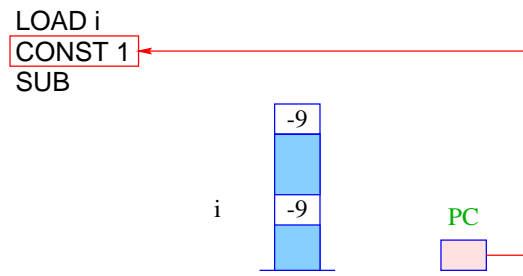
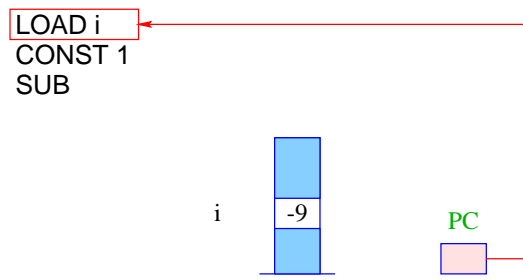
Idee:

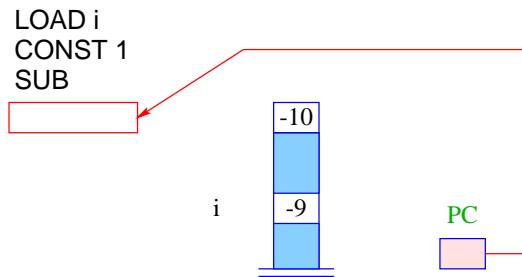
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

Übersetzung von x = LOAD i — x die i -te Variable

Übersetzung von 17 = CONST 17

Übersetzung von $x - 1$ = LOAD i
CONST 1
SUB





Allgemein:

Übersetzung von $- \text{expr}$ = Übersetzung von expr
NEG

Übersetzung von $\text{expr}_1 + \text{expr}_2$ = Übersetzung von expr_1
Übersetzung von expr_2
ADD

... analog für die anderen Operatoren ... **Beispiel:**

Sei expr der Ausdruck: $(x + 7) * (y - 14)$

wobei x und y die 0. bzw. 1. Variable sind.

Dann liefert die Übersetzung:

```
LOAD 0
CONST 7
ADD
LOAD 1
CONST 14
SUB
MUL
```

9.3 Übersetzung von Zuweisungen

Idee:

- Übersetze den Ausdruck auf der rechten Seite.
Das liefert eine Befehlsfolge, die den Wert der rechten Seite oben auf dem Stack ablegt.
- Speichere nun diesen Wert in der Zelle für die linke Seite ab!

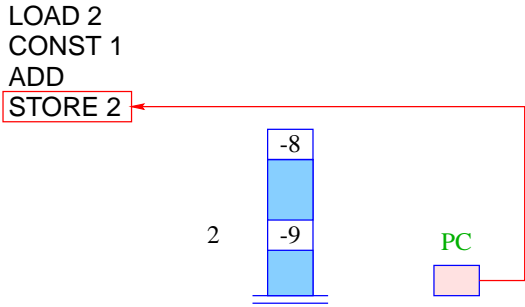
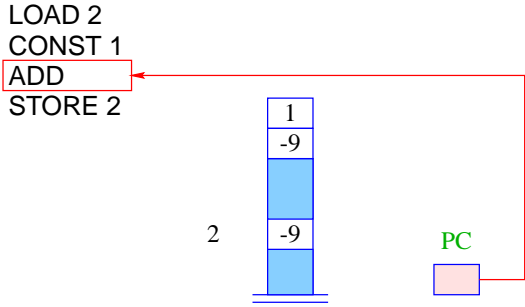
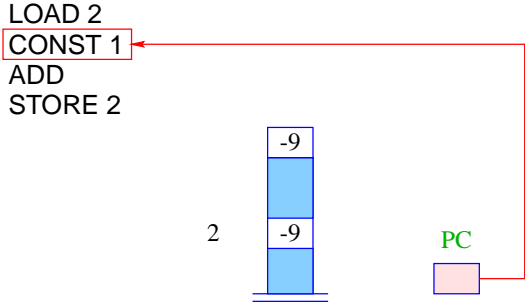
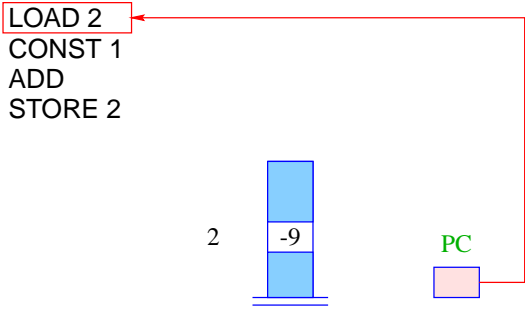
Sei x die Variable Nr. i . Dann ist

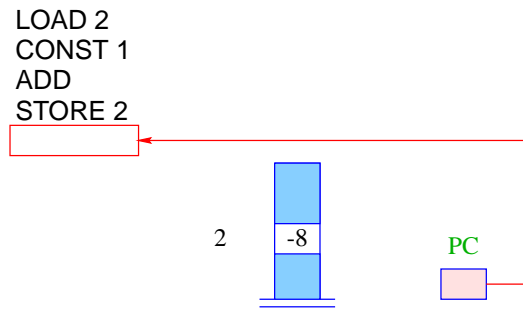
Übersetzung von $x = \text{expr};$ = Übersetzung von expr
STORE i

Beispiel:

Für $x = x + 1$; (x die 2. Variable) liefert das:

```
LOAD 2
CONST 1
ADD
STORE 2
```





Bei der Übersetzung von `x = read();` und `write(expr);` gehen wir analog vor :-
)

Sei x die Variable Nr. i . Dann ist

Übersetzung von `x = read();` = READ
STORE i

Übersetzung von `write(expr);` = Übersetzung von `expr`
WRITE

9.4 Übersetzung von if-Statements

Bezeichne `stmt` das if-Statement

`if (cond) stmt1 else stmt2`

Idee:

- Wir erzeugen erst einmal Befehlsfolgen für `cond`, `stmt1` und `stmt2`.
- Diese ordnen wir hinter einander an.
- Dann fügen wir Sprünge so ein, dass in Abhängigkeit des Ergebnisses der Auswertung der Bedingung jeweils entweder nur `stmt1` oder nur `stmt2` ausgeführt wird.

Folglich (mit A, B zwei neuen Marken):

Übersetzung von `stmt` = Übersetzung von `cond`
FJUMP A
Übersetzung von `stmt1`
JUMP B
A: Übersetzung von `stmt2`
B: ...

- Marke A markiert den Beginn des `else`-Teils.
- Marke B markiert den ersten Befehl hinter dem if-Statement.
- Falls die Bedingung sich zu `false` evaluiert, wird der `then`-Teil übersprungen (mithilfe von FJUMP A).

- Nach Abarbeitung des then-Teils muss in jedem Fall hinter dem gesamten if-Statement fortgefahren werden. Dazu dient JUMP B.

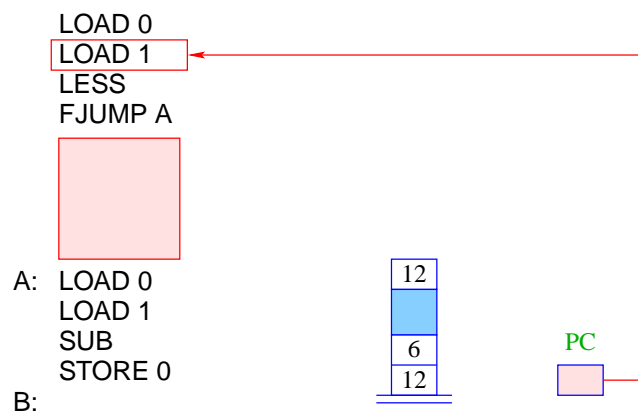
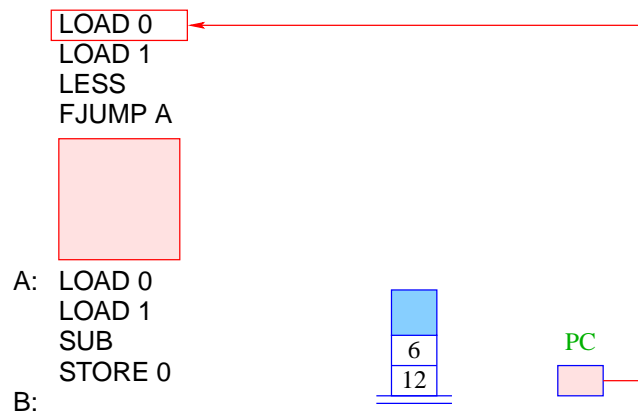
Beispiel:

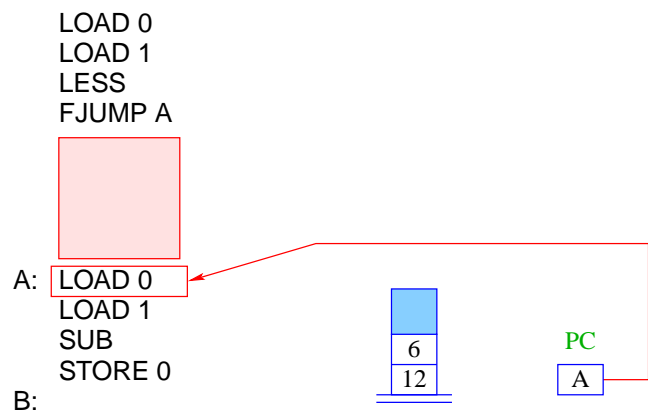
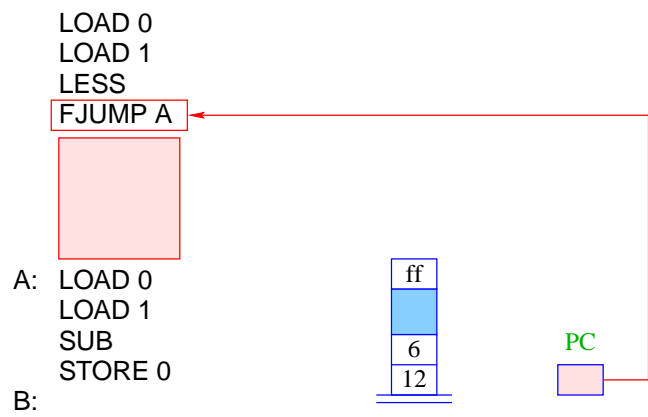
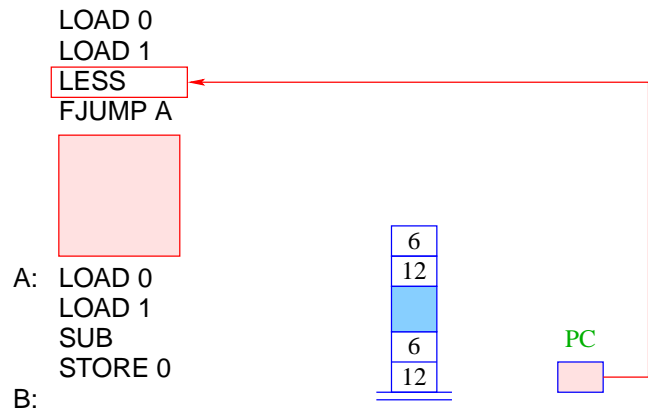
Für das Statement:

```
if (x < y) y = y - x;
else x = x - y;
```

(x und y die 0. bzw. 1. Variable) ergibt das:

LOAD 0	LOAD 1	A: LOAD 0
LOAD 1	LOAD 0	LOAD 1
LESS	SUB	SUB
FJUMP A	STORE 1	STORE 0
	JUMP B	B: ...

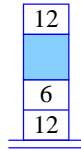




LOAD 0
LOAD 1
LESS
FJUMP A



A: LOAD 0
LOAD 1
SUB
STORE 0
B:



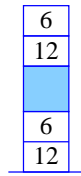
PC



LOAD 0
LOAD 1
LESS
FJUMP A



A: LOAD 0
LOAD 1
SUB
STORE 0
B:



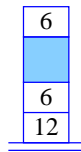
PC



LOAD 0
LOAD 1
LESS
FJUMP A

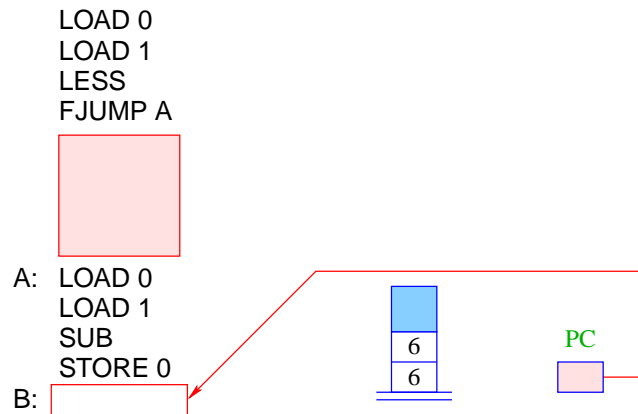


A: LOAD 0
LOAD 1
SUB
STORE 0
B:



PC





9.5 Übersetzung von while-Statements

Bezeichne `stmt` das while-Statement

`while (cond) stmt1`

Idee:

- Wir erzeugen erst einmal Befehlsfolgen für `cond` und `stmt1`.
- Diese ordnen wir hinter einander an.
- Dann fügen wir Sprünge so ein, dass in Abhängigkeit des Ergebnisses der Auswertung der Bedingung entweder hinter das while-Statement gesprungen wird oder `stmt1` ausgeführt wird.
- Nach Ausführung von `stmt1` müssen wir allerdings wieder an den Anfang des Codes zurückspringen :-)

Folglich (mit A, B zwei neuen Marken):

Übersetzung von `stmt` = A: Übersetzung von `cond`
 FJUMP B
 Übersetzung von `stmt1`
 JUMP A
 B: ...

- Marke A markiert den Beginn des while-Statements.
- Marke B markiert den ersten Befehl hinter dem while-Statement.
- Falls die Bedingung sich zu `false` evaluiert, wird die Schleife verlassen (mithilfe von FJUMP B).
- Nach Abarbeitung des Rumpfs muss das while-Statement erneut ausgeführt werden. Dazu dient JUMP A.

Beispiel:

Für das Statement:

```
while (1 < x) x = x - 1;
```

(x die 0. Variable) ergibt das:

A:	CONST 1	LOAD 0
	LOAD 0	CONST 1
	LESS	SUB
	FJUMP B	STORE 0
		JUMP A
		B: ...

9.6 Übersetzung von Statement-Folgen

Idee:

- Wir erzeugen zuerst Befehlsfolgen für die einzelnen Statements in der Folge.
- Dann konkatenieren wir diese.

Folglich:

Übersetzung von $stmt_1 \dots stmt_k$ = Übersetzung von $stmt_1$
...
Übersetzung von $stmt_k$

Beispiel:

Für die Statement-Folge

```
y = y * x;  
x = x - 1;
```

(x und y die 0. bzw. 1. Variable) ergibt das:

LOAD 1	LOAD 0
LOAD 0	CONST 1
MUL	SUB
STORE 1	STORE 0

9.7 Übersetzung ganzer Programme

Nehmen wir an, das Programm `prog` bestehe aus einer Deklaration von n Variablen, gefolgt von der Statement-Folge `ss`.

Idee:

- Zuerst allokiert man Platz für die deklarierten Variablen.
- Dann kommt der Code für `ss`.
- Dann HALT.

Folglich:

Übersetzung von `prog` = ALLOC n
Übersetzung von `ss`
HALT

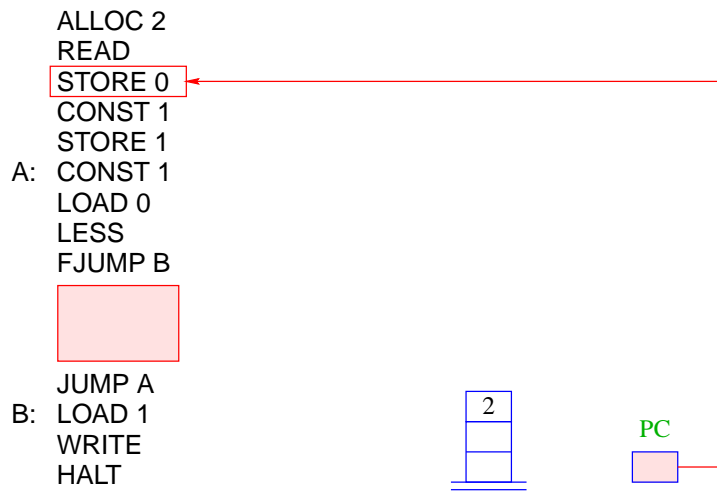
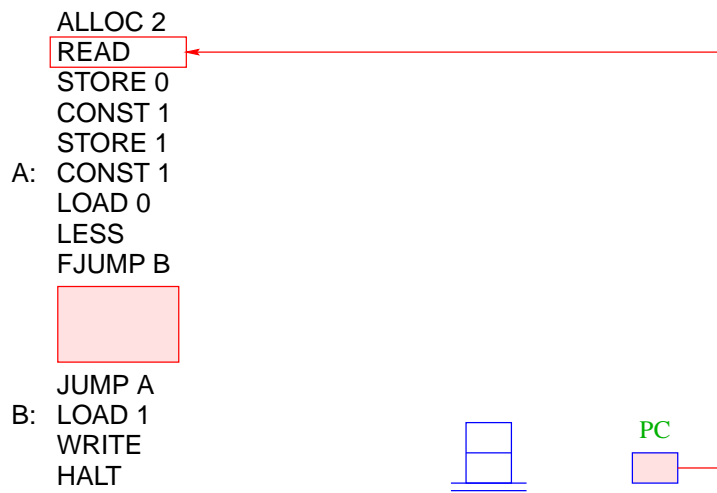
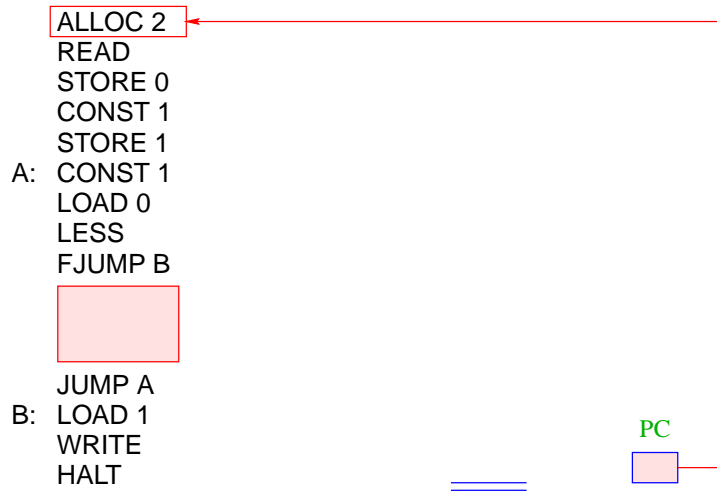
Beispiel:

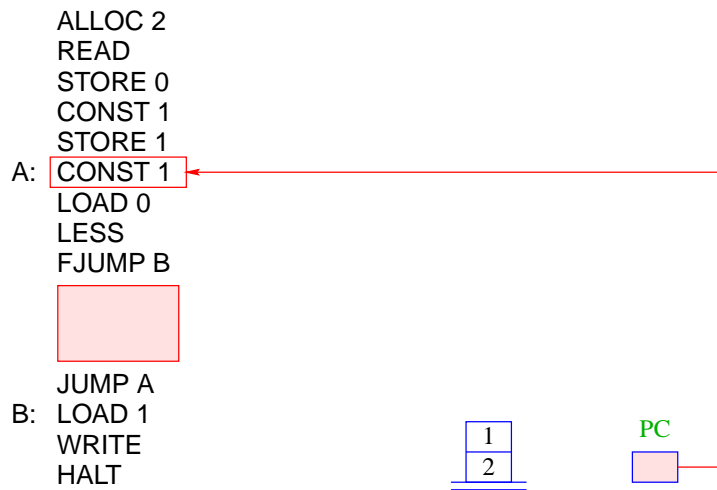
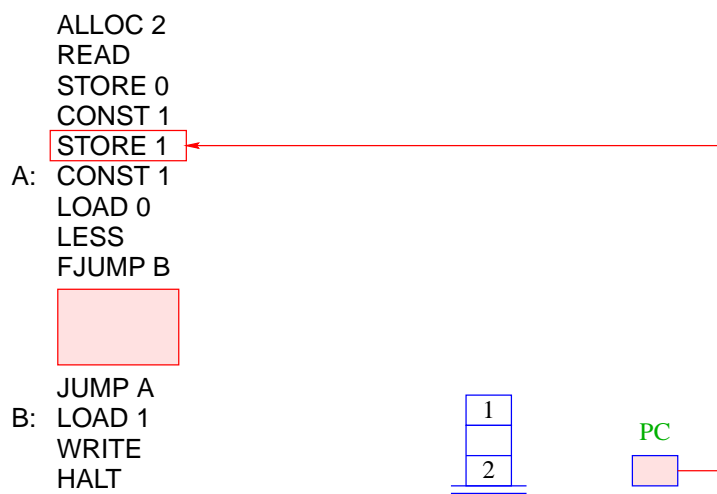
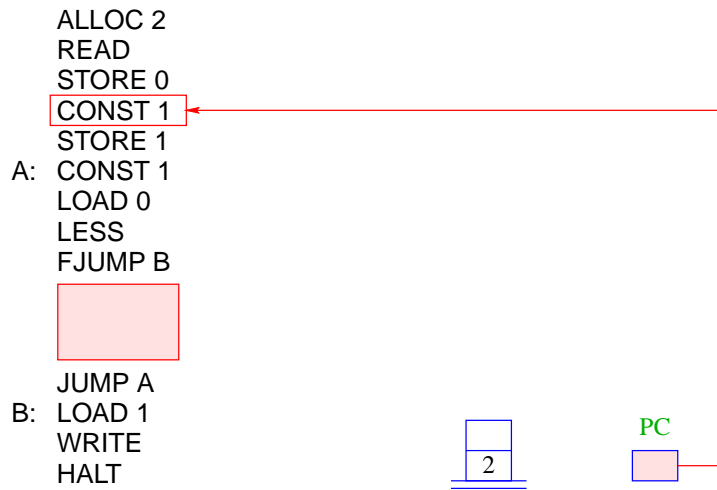
Für das Programm

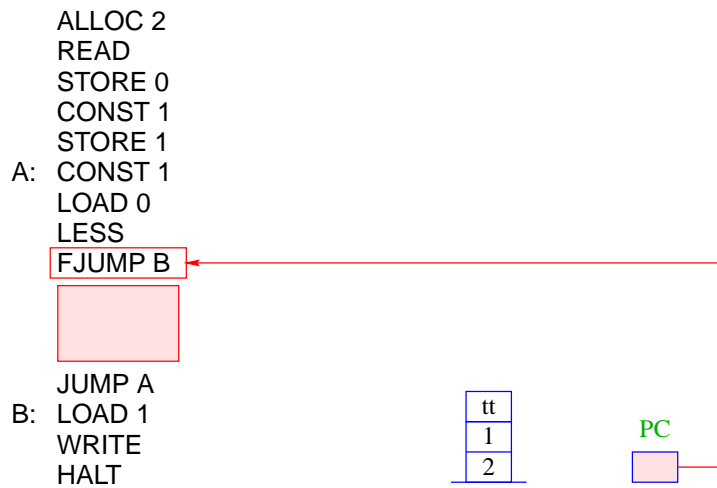
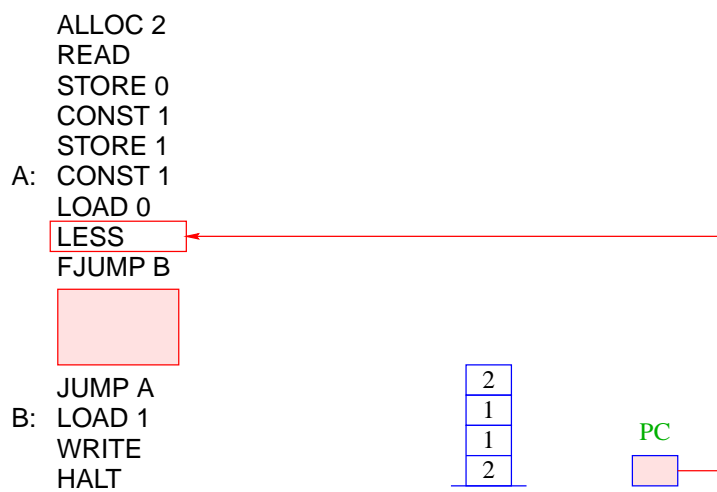
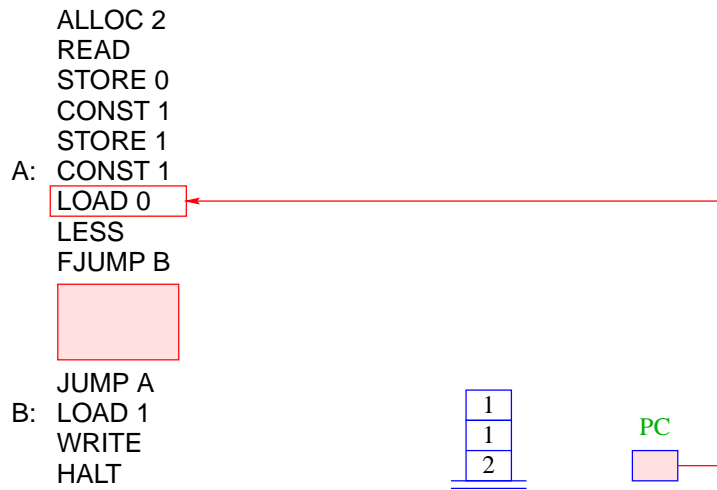
```
int x, y;  
x = read();  
y = 1;  
while (1 < x) {  
    y = y * x;  
    x = x - 1;  
}  
write(y);
```

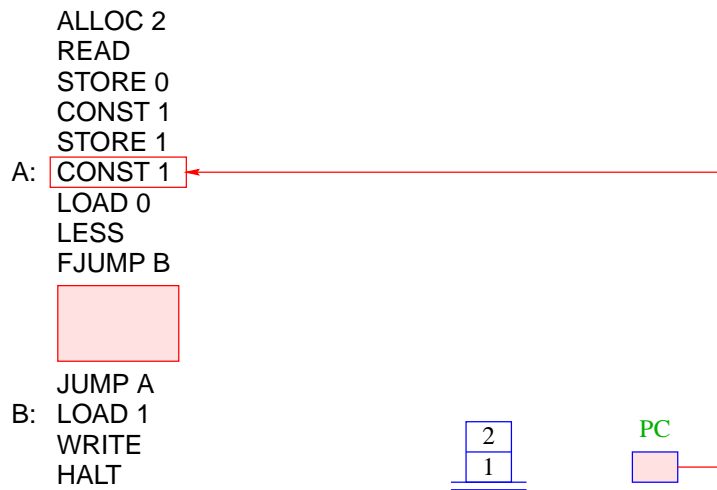
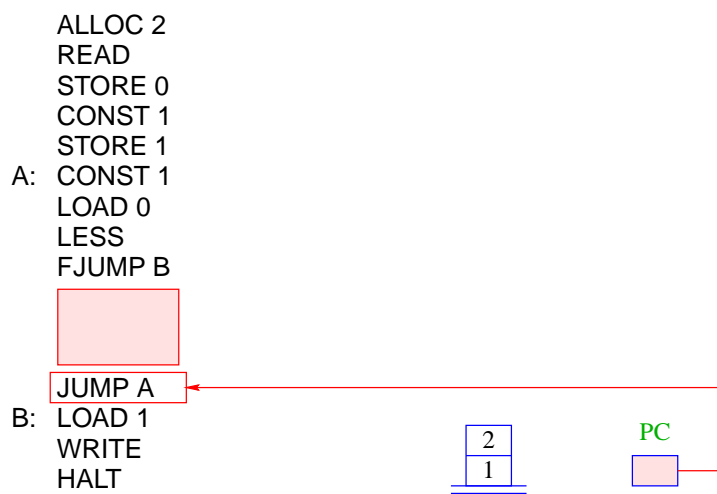
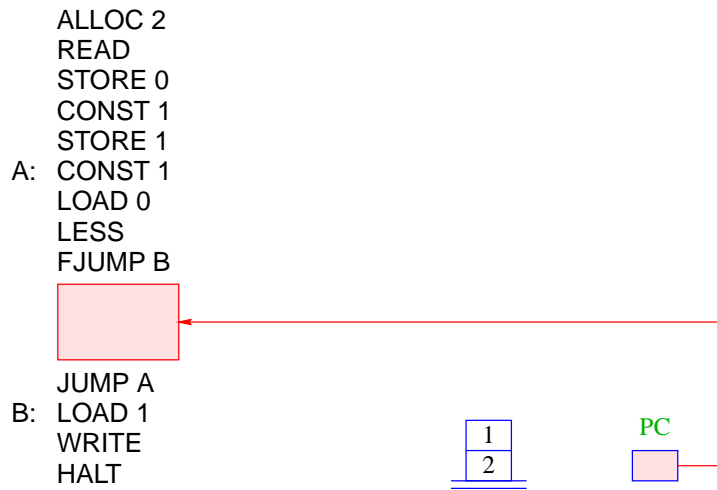
ergibt das (x und y die 0. bzw. 1. Variable) :

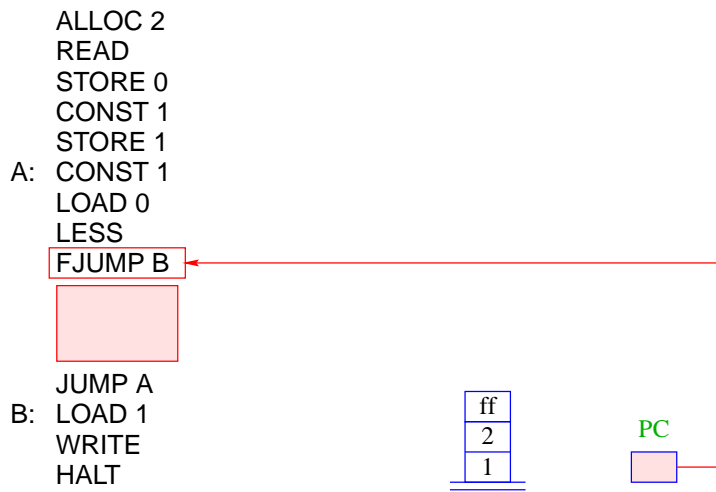
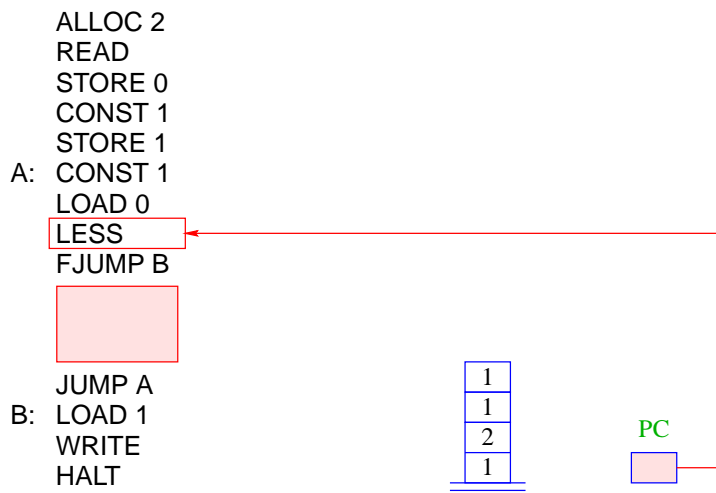
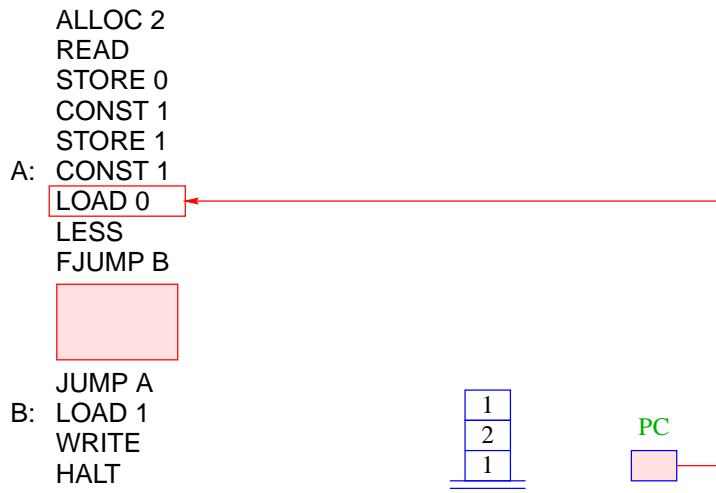
ALLOC 2	A: CONST 1	
READ	LOAD 0	
STORE 0	LESS	
CONST 1	FJUMP B	
STORE 1		
LOAD 1	LOAD 0	B: LOAD 1
LOAD 0	CONST 1	WRITE
MUL	SUB	HALT
STORE 1	STORE 0	
	JUMP A	

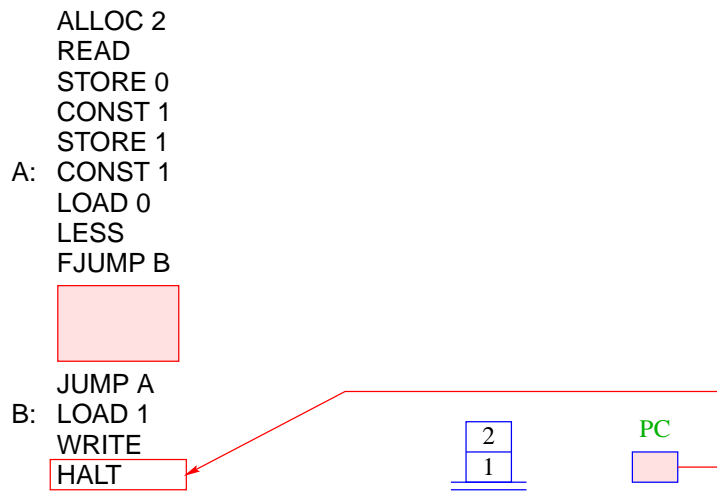
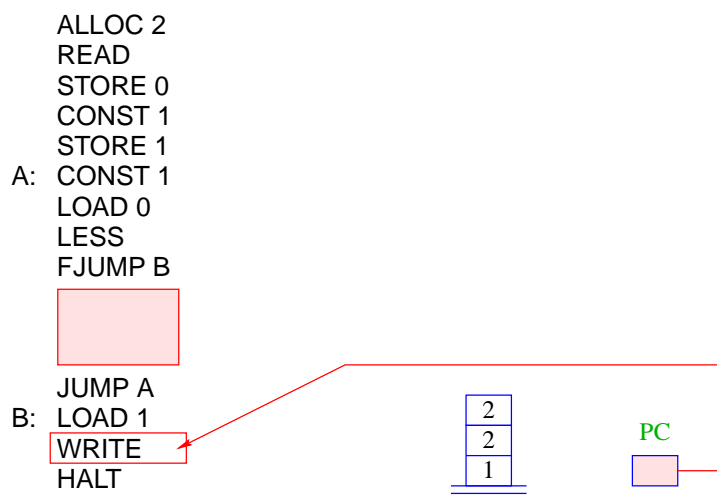
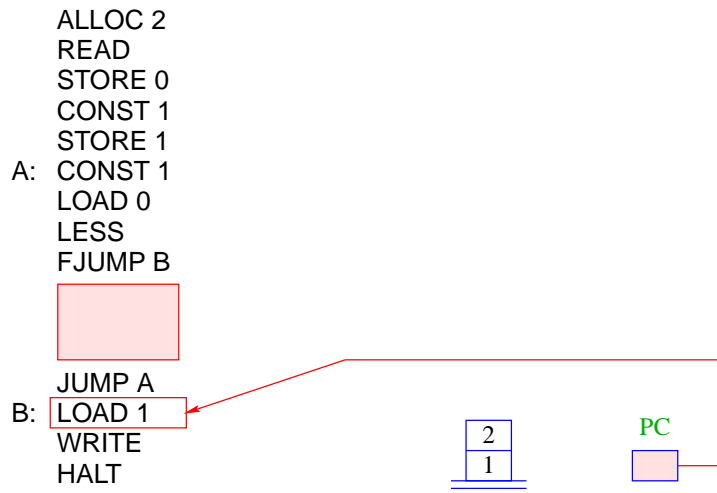












Bemerkungen:

- Die Übersetzungsfunktion, die für ein **MiniJava**-Programm **JVM**-Code erzeugt, arbeitet rekursiv auf der Struktur des Programms.
- Im Prinzip lässt sie sich zu einer Übersetzungsfunktion von ganz **Java** erweitern.
- Zu lösende Übersetzungs-Probleme:
 - mehr Datentypen;
 - Prozeduren;
 - Klassen und Objekte.

↑ **Compilerbau**

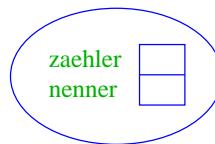
10 Klassen und Objekte

Datentyp	=	Spezifikation von Datenstrukturen
Klasse	=	Datentyp + Operationen
Objekt	=	konkrete Datenstruktur

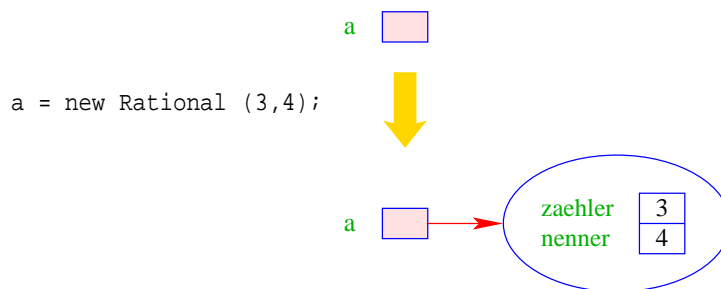
Beispiel: Rationale Zahlen

- Eine rationale Zahl $q \in \mathbb{Q}$ hat die Form $q = \frac{x}{y}$, wobei $x, y \in \mathbb{Z}$.
- x und y heißen Zähler und Nenner von q .
- Ein Objekt vom Typ Rational sollte deshalb als Komponenten int-Variablen zaehler und nenner enthalten:

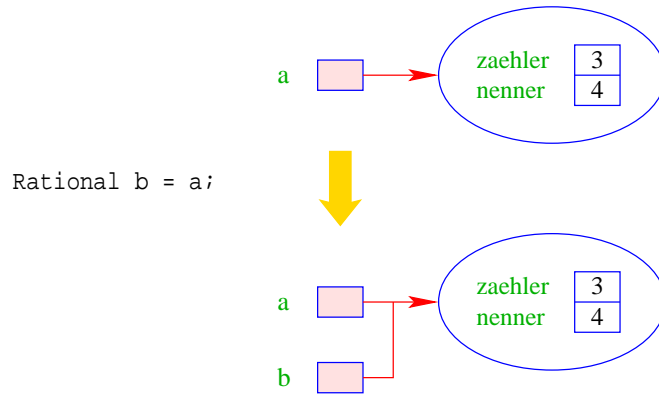
Objekt:



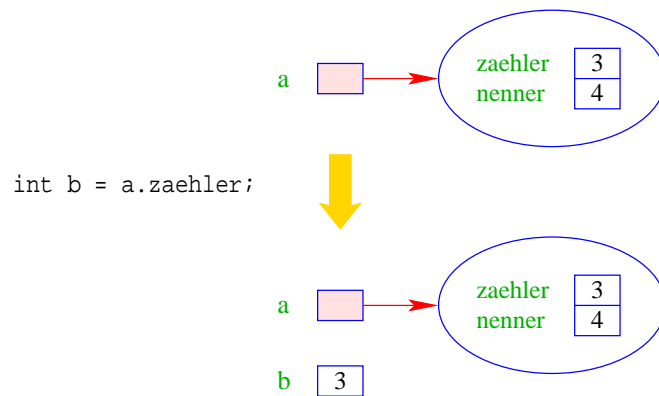
- Die Daten-Komponenten eines Objekts heißen **Objekt-Attribute** oder (kurz) **Attribute**.
- Rational name ; deklariert eine Variable für Objekte der Klasse Rational.
- Das Kommando new Rational(...) legt das Objekt an, ruft einen **Konstruktor** für dieses Objekt auf und liefert das neue Objekt zurück:



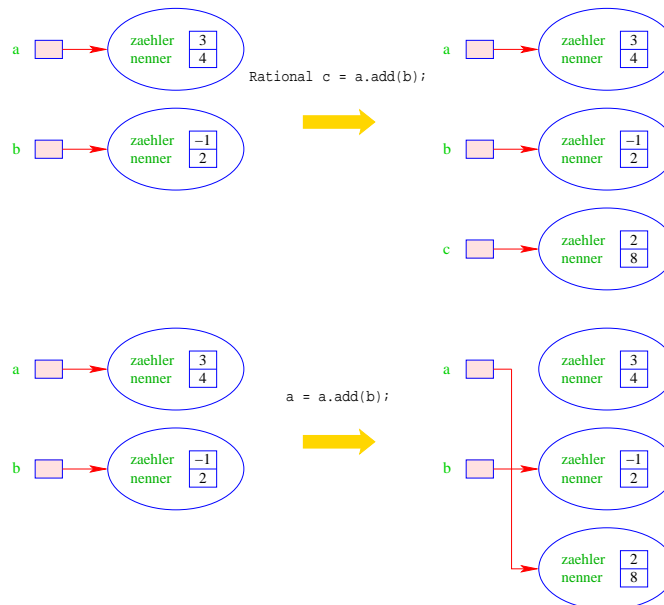
- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objekts initialisieren kann.
- Der Wert einer Rational-Variable ist ein **Verweis** auf einen Speicherbereich.
- Rational b = a; kopiert den Verweis aus a in die Variable b:



- a.zaehler liefert den Wert des Attributs zaehler des Objekts a:



- a.add(b) ruft die Operation add für a mit dem zusätzlichen aktuellen Parameter b auf:



- Die Operationen auf Objekten einer Klasse heißen auch **Methoden**, genauer: **Objekt-Methoden**.

Zusammenfassung:

Eine Klassen-Deklaration besteht folglich aus Deklarationen von:

- **Attributen** für die verschiedenen Wert-Komponenten der Objekte;
- **Konstruktoren** zur Initialisierung der Objekte;
- **Methoden**, d.h. Operationen auf Objekten.

Diese Elemente heißen auch **Members** der Klasse.

```

public class Rational {
    // Attribute:
    private int zaehler, nenner;
    // Konstruktoren:
    public Rational (int x, int y) {
        zaehler = x;
        nenner = y;
    }
    public Rational (int x) {
        zaehler = x;
        nenner = 1;
    }
    ...

    // Objekt-Methoden:
    public Rational add (Rational r) {
        int x = zaehler * r.nenner + r.zaehler * nenner;
        int y = nenner * r.nenner;
        return new Rational (x,y);
    }
    public boolean equals (Rational r) {
        return (zaehler * r.nenner == r.zaehler * nenner);
    }
    public String toString() {
        if (nenner == 1) return "" + zaehler;
        if (nenner > 0) return zaehler + "/" + nenner;
        return (-zaehler) + "/" + (-nenner);
    }
} // end of class Rational

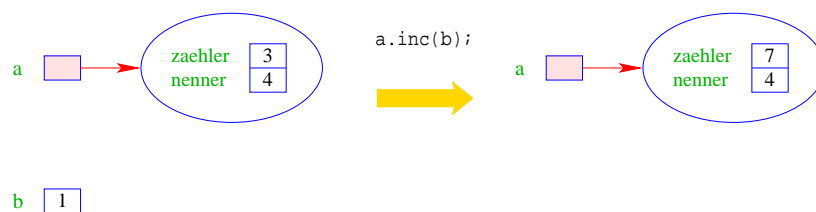
```

Bemerkungen:

- Jede Klasse **sollte** in einer separaten Datei des entsprechenden Namens stehen.
- Die Schlüsselworte `private` bzw. `public` klassifizieren, für wen die entsprechenden Members sichtbar, d.h. zugänglich sind.
- `private` heißt: nur für Members der gleichen Klasse sichtbar.
- `public` heißt: innerhalb des gesamten Programms sichtbar.
- Nicht klassifizierte Members sind nur innerhalb des aktuellen \uparrow **Package** sichtbar.
- Konstruktoren haben den gleichen Namen wie die Klasse.
- Es kann mehrere geben, sofern sie sich im Typ ihrer Argumente unterscheiden.
- Konstruktoren haben **keine** Rückgabewerte und darum auch keinen Rückgabotyp.
- Methoden haben dagegen **stets** einen Rückgabe-Typ, evt. `void`.

```
public void inc (int b) {  
    zaehler = zaehler + b * nenner;  
}
```

- Die Objekt-Methode `inc()` modifiziert das Objekt, für das sie aufgerufen wurde.



- Die Objekt-Methode `equals()` ist nötig, da der Operator “==” bei Objekten die **Identität** der Objekte testet, d.h. die Gleichheit der Referenz !!!
- Die Objekt-Methode `toString()` liefert eine `String`-Darstellung des Objekts.
- Sie wird implizit aufgerufen, wenn das Objekt als Argument für die Konkatenation “+” auftaucht.
- Innerhalb einer Objekt-Methode/eines Konstruktors kann auf die Attribute des Objekts **direkt** zugegriffen werden.
- `private`-Klassifizierung bezieht sich auf die Klasse nicht das Objekt: die Attribute **aller** `Rational`-Objekte sind für `add` sichtbar !!

Eine graphische Visualisierung der Klasse **Rational**, die nur die wesentliche Funktionalität berücksichtigt, könnte so aussehen:

Rational
- zaehler : int - nenner : int
+ add (y : Rational) : Rational + equals (y : Rational) : boolean + toString () : String

Diskussion und Ausblick:

- Solche Diagramme werden von der **UML**, d.h. der **Unified Modelling Language** bereitgestellt, um Software-Systeme zu entwerfen (↑**Software Engineering**)
- Für eine einzelne Klasse lohnt sich ein solches Diagramm nicht wirklich :-)
- Besteht ein System aber aus **sehr vielen** Klassen, kann man damit die **Beziehungen** zwischen verschiedenen Klassen verdeutlichen :-))

Achtung:

UML wurde nicht speziell für **Java** entwickelt. Darum werden Typen abweichend notiert. Auch lassen sich manche Ideen nicht eindeutig oder nur schlecht modellieren :-((

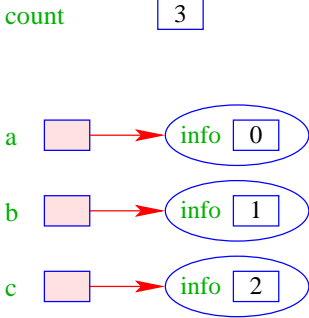
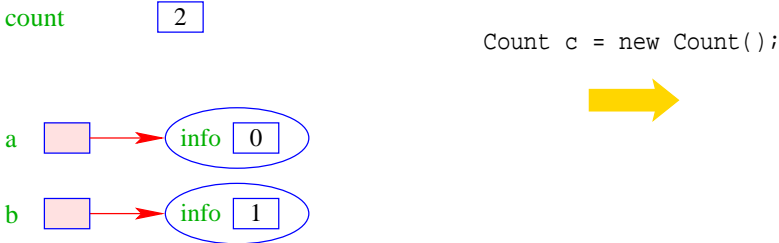
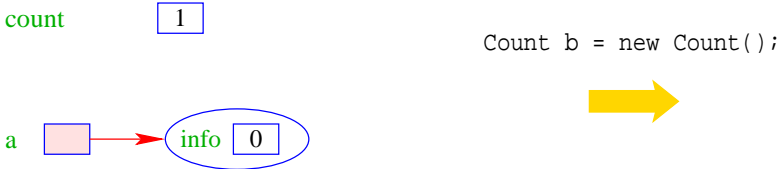
10.1 Klassen-Attribute

- Objekt-Attribute werden für jedes Objekt neu angelegt,
- **Klassen-Attribute** einmal für die gesamte Klasse :-)
- Klassen-Attribute erhalten die Qualifizierung `static`.

```
public class Count {
    private static int count = 0;
    private int info;
    // Konstruktor
    public Count() {
        info = count; count++;
    } ...
} // end of class Count
```

Modellierung:

- Die Zusatzinformation, dass das Attribut count statisch ist, wird in in spitzen Klammern im Diagramm vermerkt.
- Solche Annotationen heißen **Stereotype**.



- Das Klassen-Attribut `count` zählt hier die Anzahl der bereits erzeugten Objekte.
- Das Objekt-Attribut `info` enthält für jedes Objekt eine eindeutige Nummer.
- Außerhalb der Klasse `Class` kann man auf eine öffentliche Klassen-Variable `name` mithilfe von `Class.name` zugreifen.
- Objekt-Methoden werden stets mit einem Objekt aufgerufen ...
- dieses Objekt fungiert wie ein weiteres Argument `:-)`
- Funktionen und Prozeduren der Klasse `ohne` dieses implizite Argument heißen `Klassen-Methoden` und werden durch das Schlüsselwort `static` kenntlich gemacht.

In `Rational` könnten wir definieren:

```
public static Rational[] intToRationalArray(int[] a) {
    Rational[] b = new Rational[a.length];
    for(int i=0; i < a.length; ++i)
        b[i] = new Rational (a[i]);
    return b;
}
```

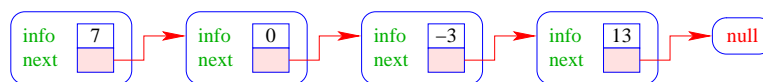
- Die Funktion erzeugt für ein Feld von `int`'s ein entsprechendes Feld von `Rational`-Objekten.
- Außerhalb der Klasse `Class` kann die öffentliche Klassen-Methode `meth()` mithilfe von `Class.meth(...)` aufgerufen werden.

11 Ein konkreter Datentyp: Listen

Nachteil von Feldern:

- feste Größe;
- Einfügen neuer Elemente nicht möglich;
- Streichen ebenfalls nicht :-)

Idee: Listen



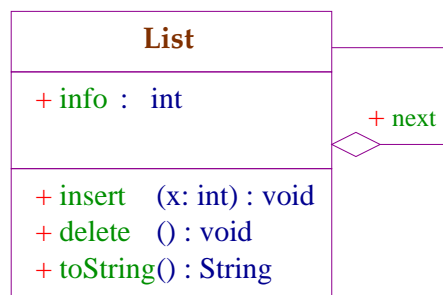
... das heißt:

- info == Element der Liste;
- next == Verweis auf das nächste Element;
- null == leeres Objekt.

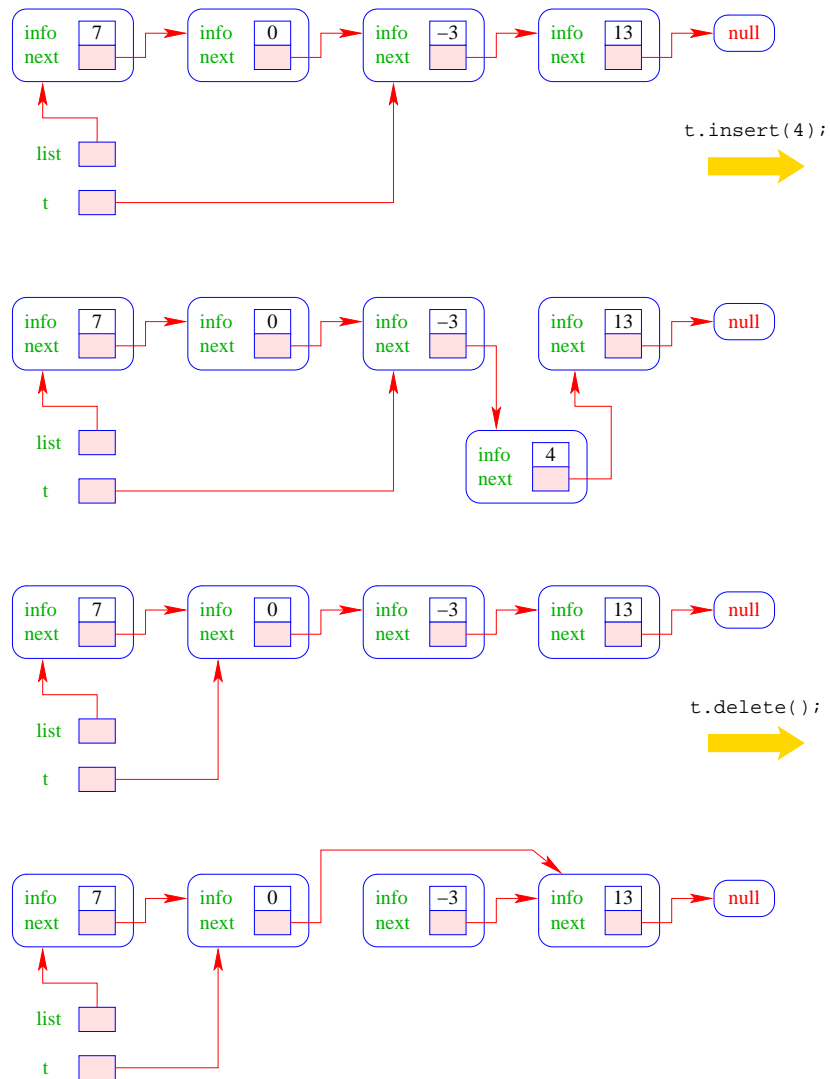
Operationen:

`void insert(int x)` : fügt neues x hinter dem aktuellen Element ein;
`void delete()` : entfernt Knoten hinter dem aktuellen Element;
`String toString()` : liefert eine String-Darstellung.

Modellierung:



Die Rauten-Verbindung heißt auch **Aggregation**.



Weiterhin sollte man

- ... eine Liste auf Leerheit testen können;

Achtung:

das null-Objekt versteht **keinerlei** Objekt-Methoden!!!

- ... neue Listen bauen können, d.h. etwa:
 - ... eine ein-elementige Liste anlegen können;
 - ... eine Liste um ein Element verlängern können;
- ... Listen in Felder und Felder in Listen umwandeln können.

```

public class List {
    public int info;
    public List next;
    // Konstruktoren:
    public List (int x, List l) {
        info = x;
        next = l;
    }
    public List (int x) {
        info = x;
        next = null;
    }
    ...

    // Objekt-Methoden:
    public void insert(int x) {
        next = new List(x,next);
    }
    public void delete() {
        if (next != null)
            next = next.next;
    }
    public String toString() {
        String result = "["+info;
        for(List t=next; t!=null; t=t.next)
            result = result+", "+t.info;
        return result+"]";
    }
    ...
}

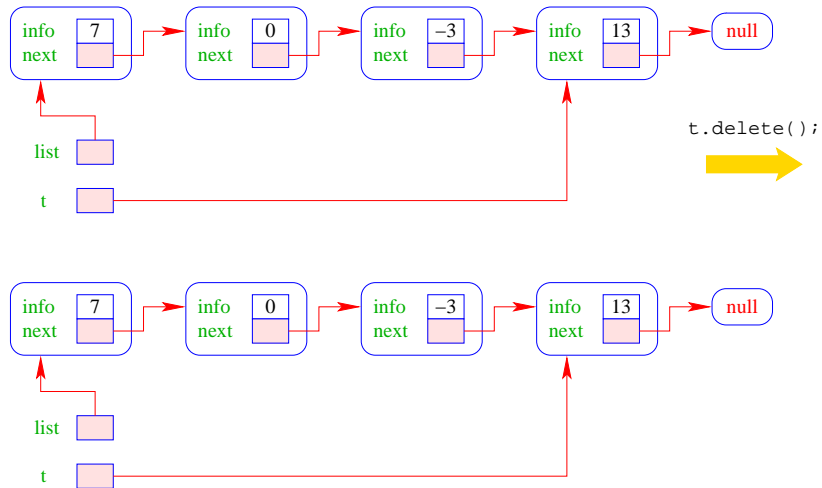
```

- Die Attribute sind public und daher beliebig einsehbar und modifizierbar \implies sehr flexibel, sehr fehleranfällig.
- insert() legt einen neuen Listenknoten an fügt ihn hinter dem aktuellen Knoten ein.
- delete() setzt den aktuellen next-Verweis auf das übernächste Element um.

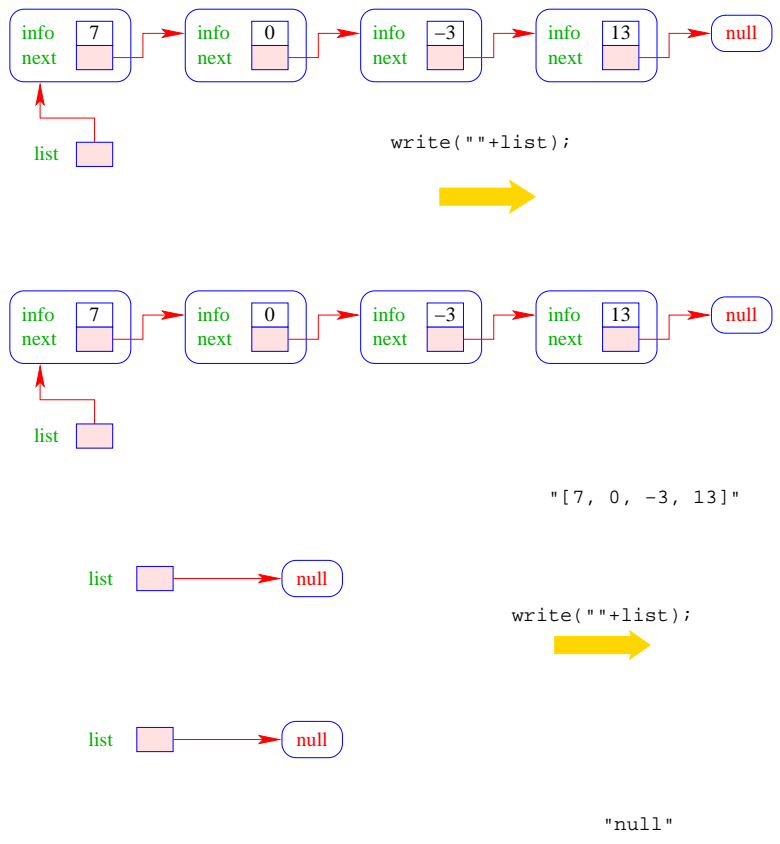
Achtung:

Wenn delete() mit dem letzten Element der Liste aufgerufen wird, zeigt next auf null.

\implies Wir tun dann nix.



- Weil Objekt-Methoden nur für von null verschiedene Objekte aufgerufen werden können, kann die leere Liste nicht mittels toString() als String dargestellt werden.
- Der Konkatenations-Operator “+” ist so schlau, vor Aufruf von toString() zu überprüfen, ob ein null-Objekt vorliegt. Ist das der Fall, wird “null” ausgegeben.
- Wollen wir eine andere Darstellung, benötigen wir eine Klassen-Methode String toString(List l).



```

// Klassen-Methoden:
public static boolean isEmpty(List l) {
    if (l == null)
        return true;
    else
        return false;
}
public static String toString(List l) {
    if (l == null)
        return "[]";
    else
        return l.toString();
}
...

public static List arrayToList(int[] a) {
    List result = null;
    for(int i = a.length-1; i>=0; --i)
        result = new List(a[i],result);
    return result;
}
public int[] listToArray() {
    List t = this;
    int n = length();
    int[] a = new int[n];
    for(int i = 0; i < n; ++i) {
        a[i] = t.info;
        t = t.next;
    }
    return a;
}
...

```

- Damit das erste Element der Ergebnis-Liste a[0] enthält, beginnt die Iteration in arrayToList() beim **größten** Element.
- listToArray() ist als Objekt-Methode realisiert und funktioniert darum nur für **nicht-leere** Listen :-)
- Um eine Liste in ein Feld umzuwandeln, benötigen wir seine Länge.


```

private int length() {
    int result = 1;
    for(List t = next; t!=null; t=t.next)
        result++;
    return result;
}
} // end of class List

```

- Weil length() als private deklariert ist, kann es nur von den Methoden der Klasse List benutzt werden.
- Damit length() auch für null funktioniert, hätten wir analog zu toString() auch noch eine Klassen-Methode int length(List l) definieren können.
- Diese Klassen-Methode würde uns ermöglichen, auch eine Klassen-Methode static int [] listToArray (List l) zu definieren, die auch für leere Listen definiert ist.

Anwendung: Mergesort – Sortieren durch Mischen

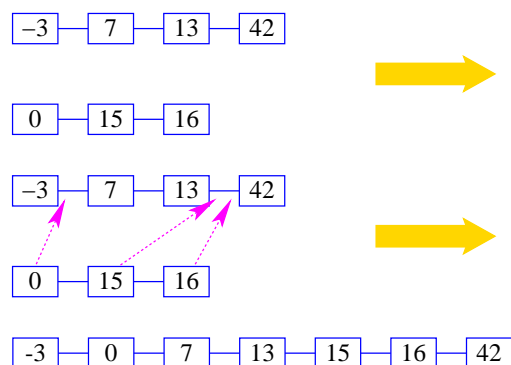


John von Neumann (1945)

Mischen:

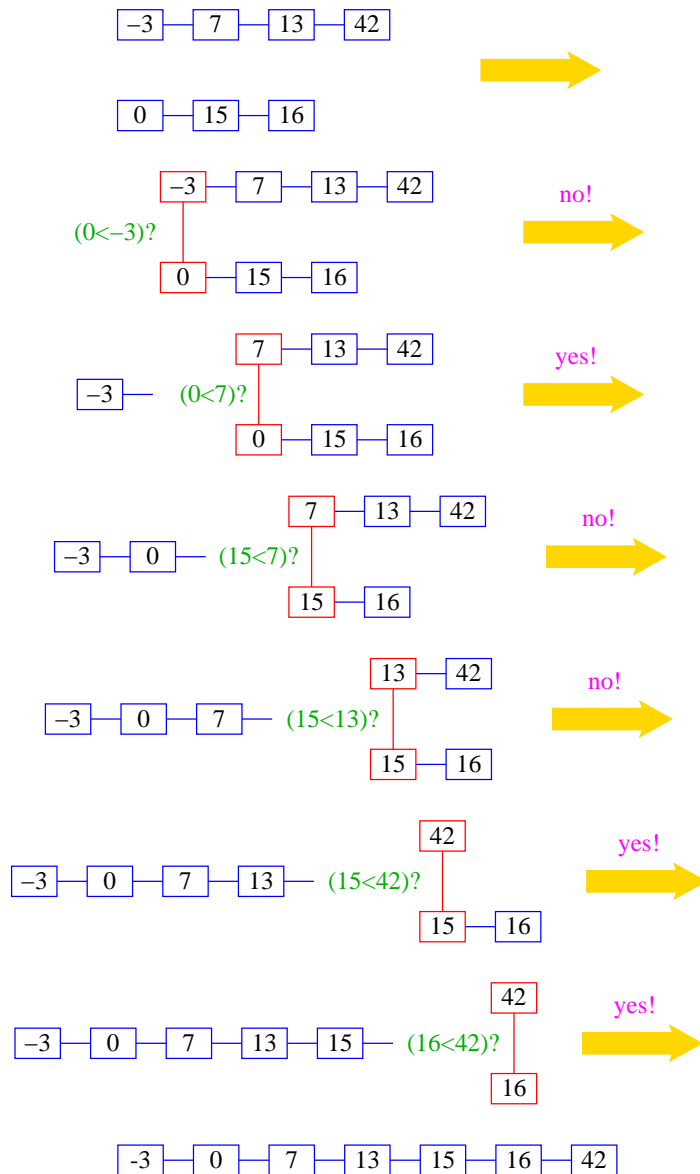
Eingabe: zwei sortierte Listen;

Ausgabe: eine gemeinsame sortierte Liste.



Idee:

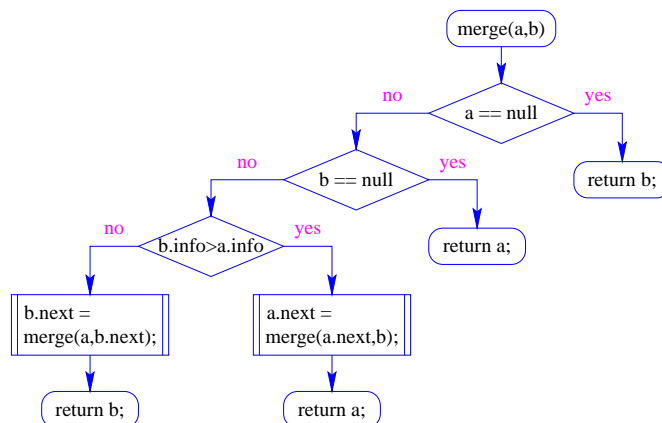
- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Input-Listen.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig :-)



Rekursive Implementierung:

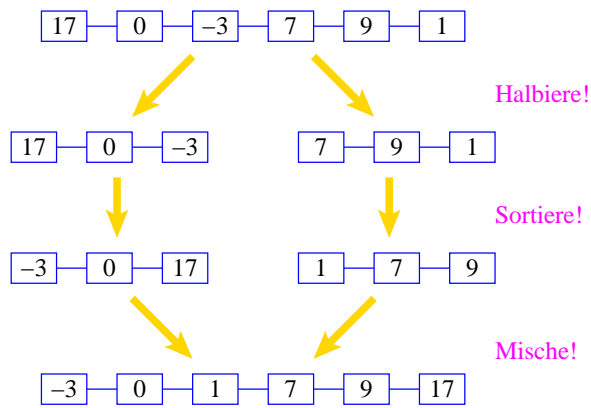
- Falls eine der beiden Listen a und b leer ist, geben wir die andere aus :-)
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente erhalten ...

```
public static List merge(List a, List b) {  
    if (b == null)  
        return a;  
    if (a == null)  
        return b;  
    if (b.info > a.info) {  
        a.next = merge(a.next, b);  
        return a;  
    } else {  
        b.next = merge(a, b.next);  
        return b;  
    }  
}
```



Sortieren durch Mischen:

- Teile zu sortierende Liste in zwei Teil-Listen;
- sortiere jede Hälfte für sich;
- mische die Ergebnisse!

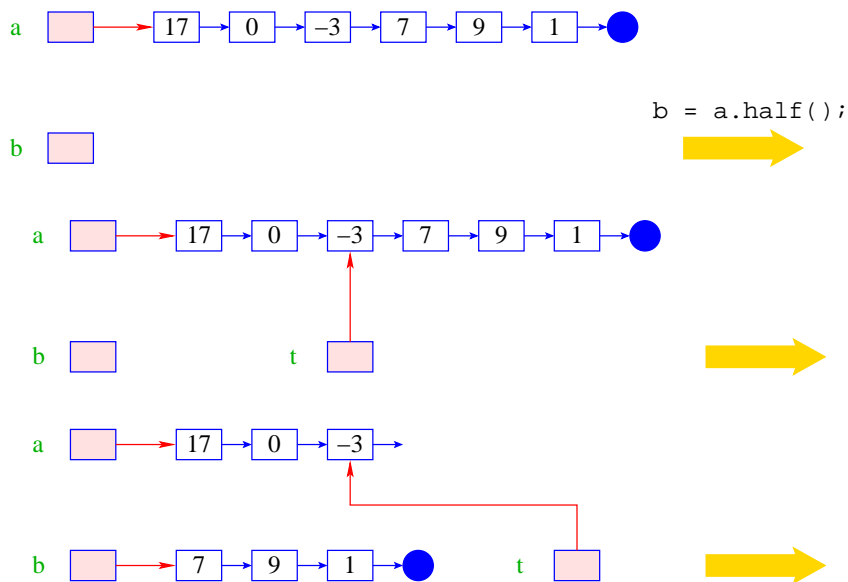


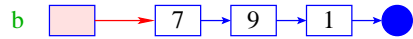
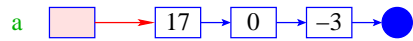
```

public static List sort(List a) {
    if (a == null || a.next == null)
        return a;
    List b = a.half(); // Halbiere!
    a = sort(a);
    b = sort(b);
    return merge(a,b);
}

public List half() {
    int n = length();
    List t = this;
    for(int i=0; i<n/2-1; i++)
        t = t.next;
    List result = t.next;
    t.next = null;
    return result;
}

```





Diskussion:

- Sei $V(n)$ die Anzahl der Vergleiche, die Mergesort maximal zum Sortieren einer Liste der Länge n benötigt.

Dann gilt:

$$V(1) = 0$$

$$V(2n) \leq 2 \cdot V(n) + 2 \cdot n$$

- Für $n = 2^k$, sind das dann nur $k \cdot n$ Vergleiche !!!

Achtung:

- Unsere Funktion `sort()` zerstört ihr Argument ?!
- Alle Listen-Knoten der Eingabe werden weiterverwendet :-)
- Die Idee des Sortierens durch Mischen könnte auch mithilfe von Feldern realisiert werden (wie ?-)
- Sowohl das Mischen wie das Sortieren könnte man statt rekursiv auch iterativ implementieren (wie ?-)

12 Abstrakte Datentypen

- Spezifiziere nur die Operationen!
- Verberge Details
 - der Datenstruktur;
 - der Implementierung der Operationen.

\implies Information Hiding

Sinn:

- Verhindern illegaler Zugriffe auf die Datenstruktur;

- **Entkopplung** von Teilproblemen für
 - Implementierung, aber auch
 - Fehlersuche und
 - Wartung;
- leichter **Austausch** von Implementierungen (↑**rapid prototyping**).

12.1 Beispiel 1: Keller (Stacks)

Operationen:

```
boolean isEmpty() : testet auf Leerheit;
int pop()         : liefert oberstes Element;
void push(int x)  : legt x oben auf dem Keller ab;
String toString() : liefert eine String-Darstellung.
```

Weiterhin müssen wir einen leeren Keller anlegen können.



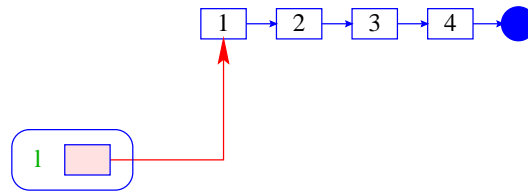
Friedrich Ludwig Bauer, TUM

Modellierung:

Stack	
+	Stack ()
+	isEmpty () : boolean
+	push (x: int) : void
+	pop () : int

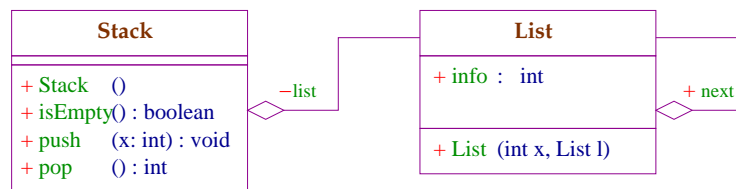
Erste Idee:

- Realisiere Keller mithilfe einer Liste!



- Das Attribut 1 zeigt auf das oberste Element.

Modellierung:



Die gefüllte Raute besagt, dass die Liste nur von Stack aus zugreifbar ist :-)

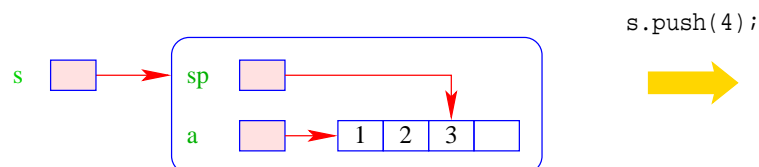
Implementierung:

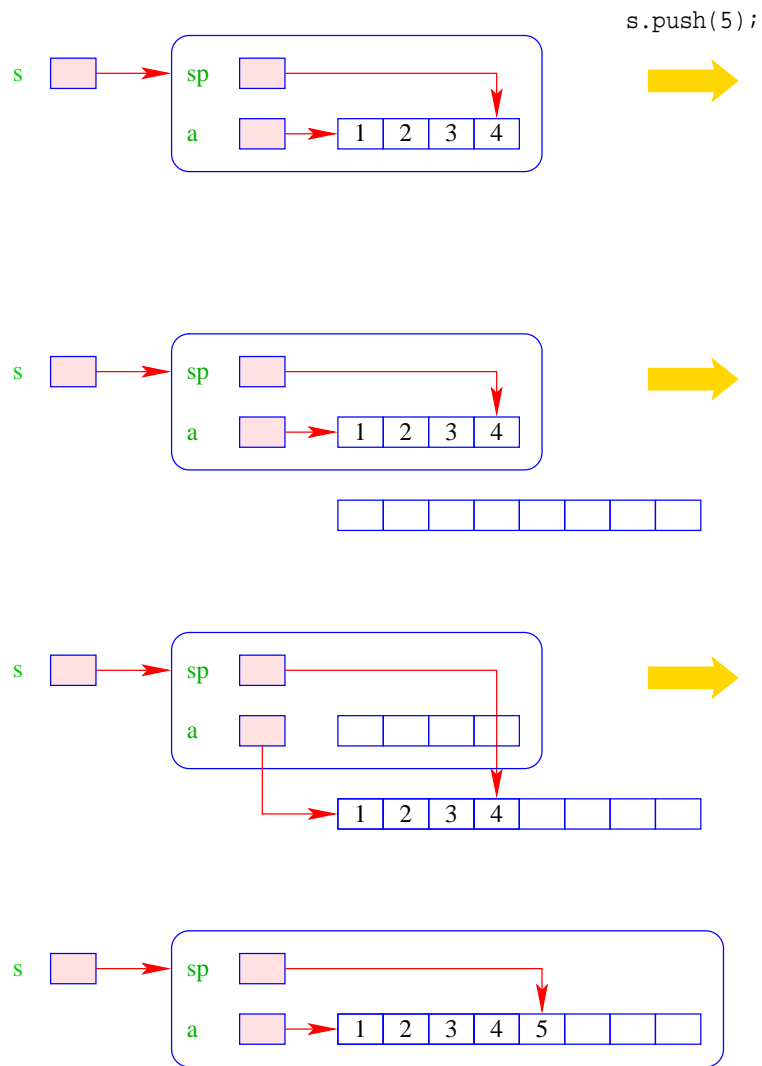
```
public class Stack {
    private List list;
    // Konstruktor:
    public Stack() {
        list = null;
    }
    // Objekt-Methoden:
    public boolean isEmpty() {
        return list==null;
    }
    ...
    public int pop() {
        int result = list.info;
        list = list.next;
        return result;
    }
    public void push(int a) {
        list = new List(a,list);
    }
    public String toString() {
        return List.toString(list);
    }
} // end of class Stack
```

- Die Implementierung ist sehr einfach;
- ... nutzte gar nicht alle Features von List aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut;
⇒ führt zu schlechtem ↑Cache-Verhalten des Programms!

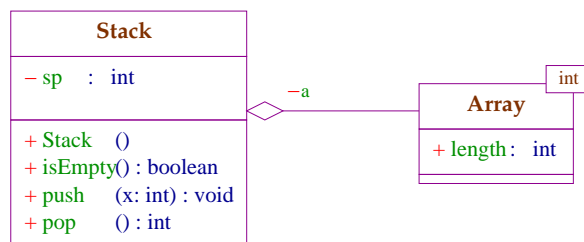
Zweite Idee:

- Realisiere den Keller mithilfe eines Felds und eines Stackpointers, der auf die oberste belegte Zelle zeigt.
- Lläuft das Feld über, ersetzen wir es durch ein größeres :-)





Modellierung:



Implementierung:

```
public class Stack {
    private int sp;
    private int[] a;
    // Konstruktoren:
    public Stack() {
        sp = -1; a = new int[4];
    }
    // Objekt-Methoden:
    public boolean isEmpty() {
        return (sp < 0);
    }
    ...

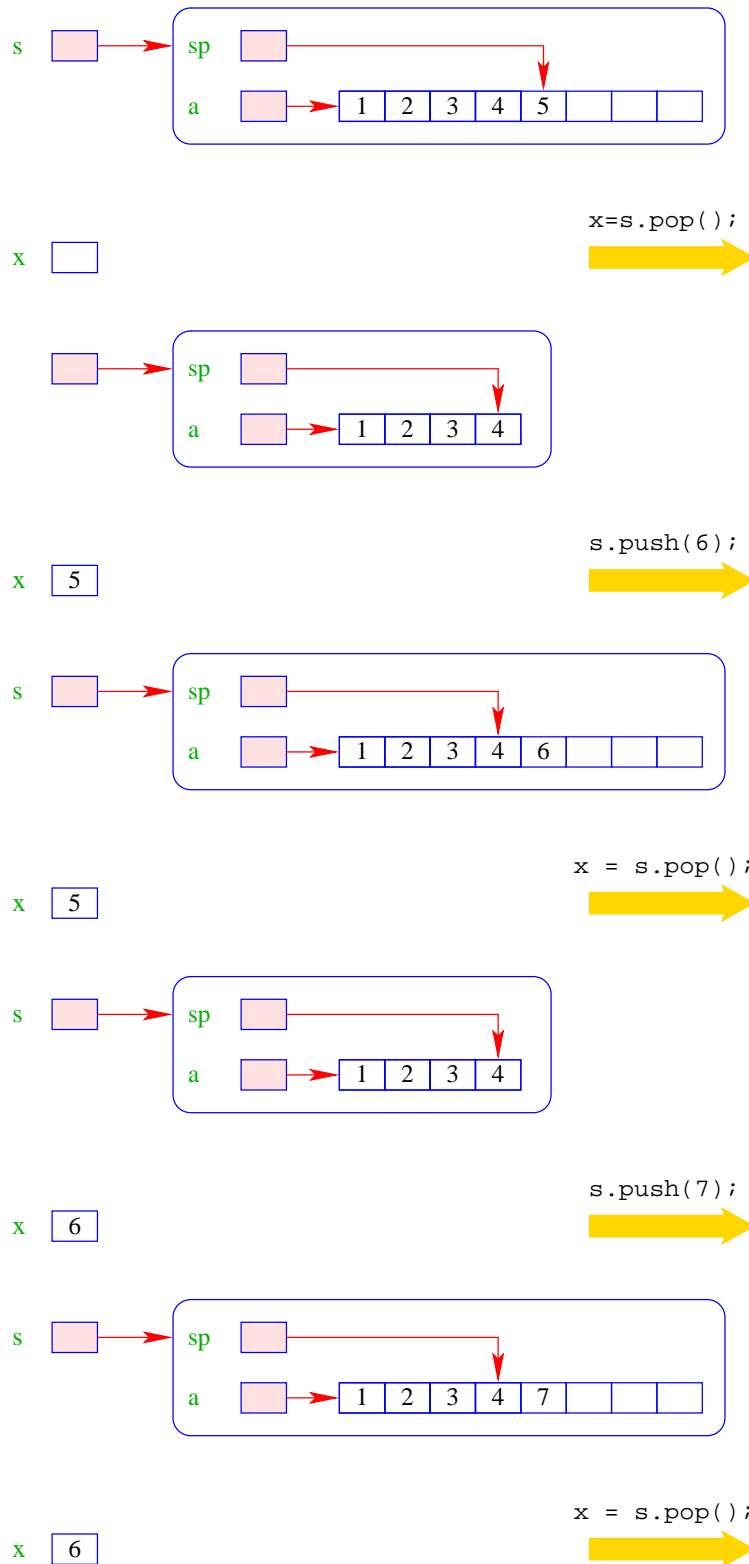
    public int pop() {
        return a[sp--];
    }
    public void push(int x) {
        ++sp;
        if (sp == a.length) {
            int[] b = new int[2*sp];
            for(int i=0; i<sp; ++i) b[i] = a[i];
            a = b;
        }
        a[sp] = x;
    }
    public toString() {...}
} // end of class Stack
```

Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben :-)

Erste Idee:

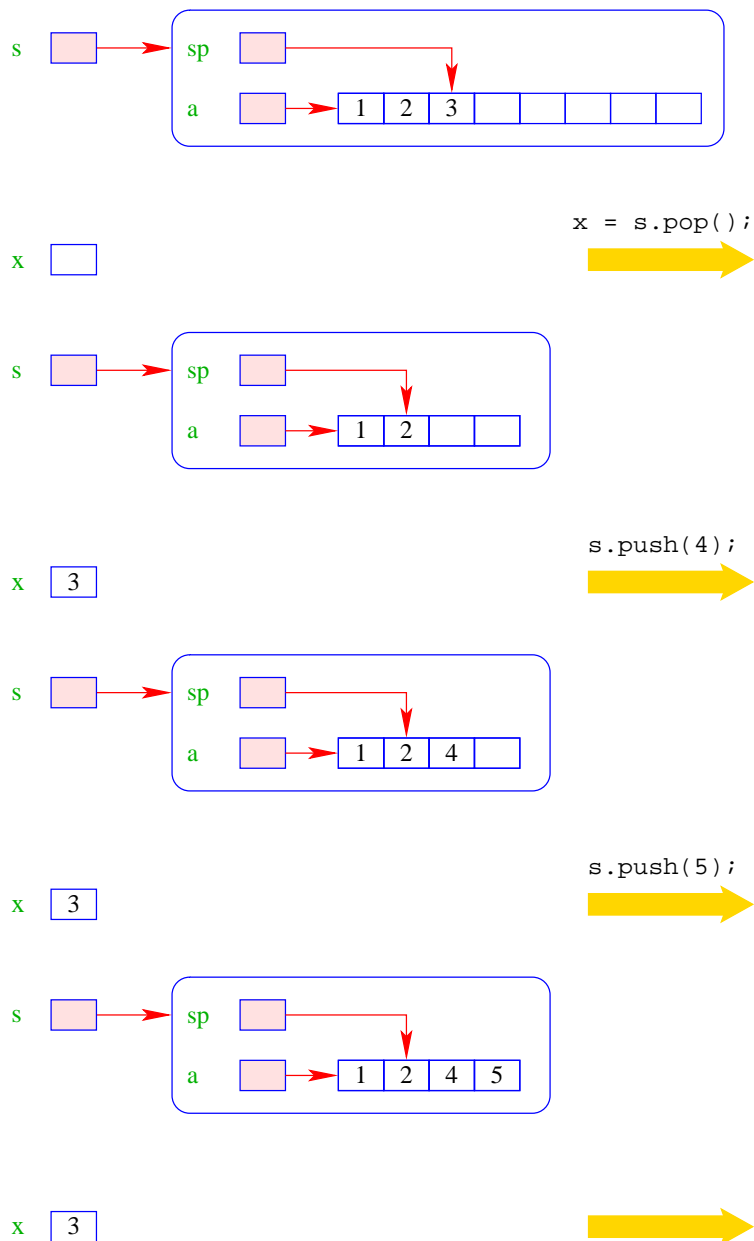
- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei ...



- Im schlimmsten Fall müssen bei **jeder** Operation sämtliche Elemente kopiert werden :-(-

Zweite Idee:

- Wir geben erst frei, wenn der Pegel auf **ein Viertel** fällt – und dann auch nur die Hälfte !



- Vor jedem Kopieren werden **mindestens** halb so viele Operationen ausgeführt, wie Elemente kopiert werden :-)
- Gemittelt über die gesamte Folge von Operationen werden pro Operation maximal zwei Zahlen kopiert ↑ **amortisierte Aufwandsanalyse**.

```

public int pop() {
    int result = a[sp];
    if (sp == a.length/4 && sp>=2) {
        int[] b = new int[2*sp];
        for(int i=0; i < sp; ++i)
            b[i] = a[i];
        a = b;
    }
    sp--;
    return result;
}

```

12.2 Beispiel 2: Schlangen (Queues)

(Warte-) Schlangen verwalten ihre Elemente nach dem **FIFO**-Prinzip (**First-In-First-Out**).

Operationen:

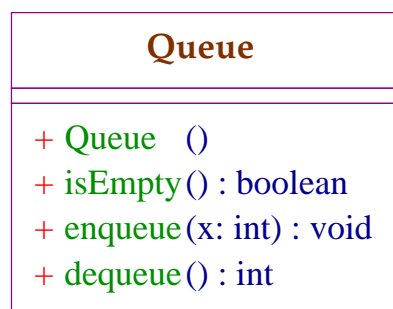
```

boolean isEmpty()    : testet auf Leerheit;
int dequeue()        : liefert erstes Element;
void enqueue(int x)  : reiht x in die Schlange ein;
String toString()    : liefert eine String-Darstellung.

```

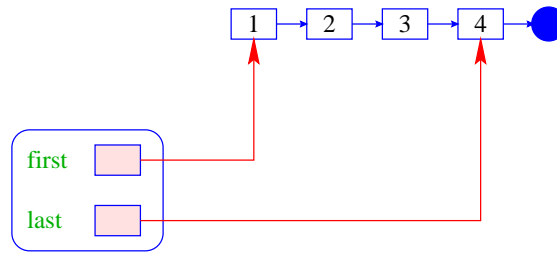
Weiterhin müssen wir eine leere Schlange anlegen können :-)

Modellierung:



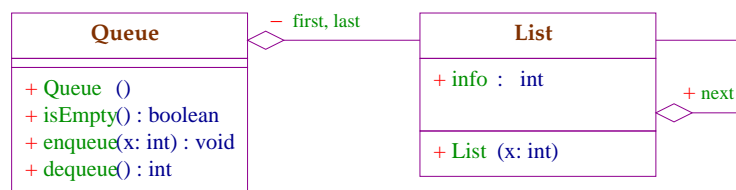
Erste Idee:

- Realisiere Schlange mithilfe einer Liste :



- first zeigt auf das nächste zu entnehmende Element;
- last zeigt auf das Element, hinter dem eingefügt wird.

Modellierung:



Objekte der Klasse Queue enthalten **zwei** Verweise auf Objekte der Klasse List :-)

Implementierung:

```

public class Queue {
    private List first, last;
    // Konstruktor:
    public Queue () {
        first = last = null;
    }
    // Objekt-Methoden:
    public boolean isEmpty() {
        return first==null;
    }
    ...
}
  
```

```

public int dequeue () {
    int result = first.info;
    if (last == first) last = null;
    first = first.next;
    return result;
}

public void enqueue (int x) {
    if (first == null) first = last = new List(x);
    else { last.next = new List(x); last = last.next; }
}

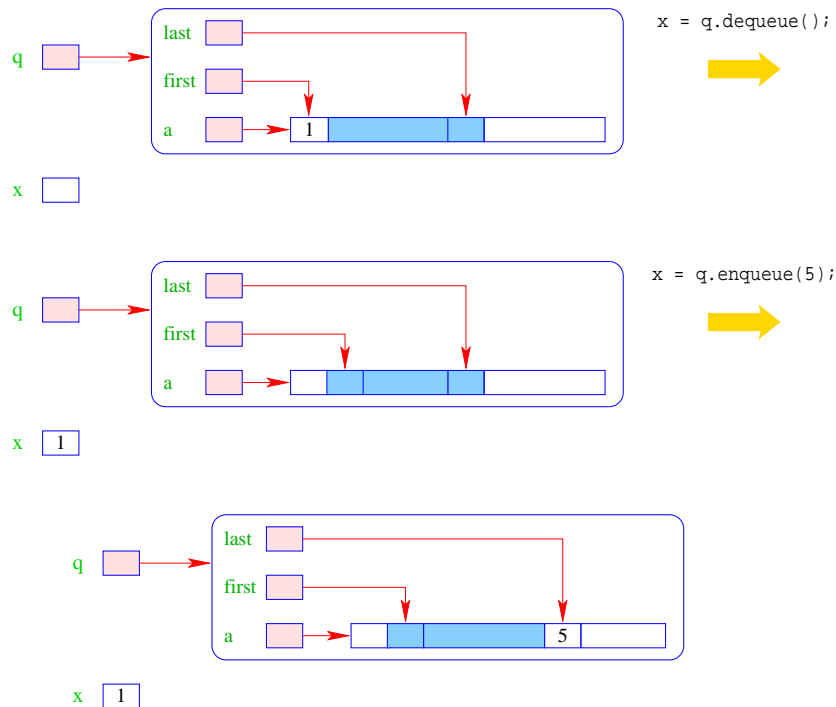
public String toString() {
    return List.toString(first);
}
} // end of class Queue

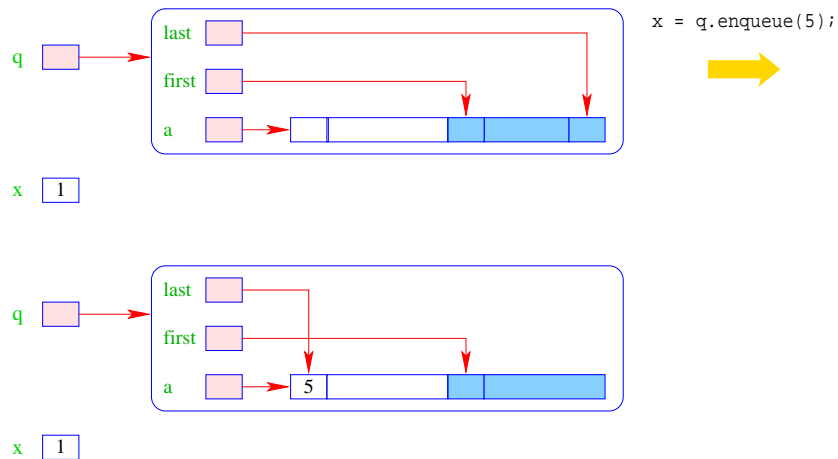
```

- Die Implementierung ist wieder sehr einfach :-)
- ... nutzt ebenfalls kaum Features von List aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut
 ⇒ führt zu schlechtem ↑Cache-Verhalten des Programms :-)

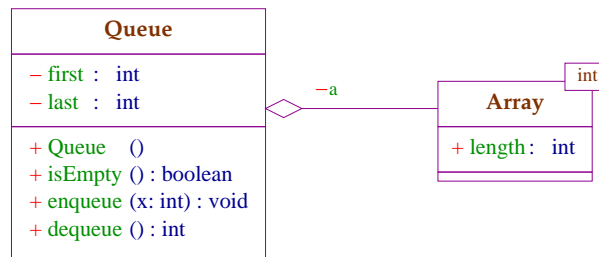
Zweite Idee:

- Realisiere die Schlange mithilfe eines Felds und **zweier** Pointer, die auf das erste bzw. letzte Element der Schlange zeigen.
- Läuft das Feld über, ersetzen wir es durch ein größeres.





Modellierung:



Implementierung:

```

public class Queue {
    private int first, last;
    private int[] a;
    // Konstruktor:
    public Queue () {
        first = last = -1;
        a = new int[4];
    }
    // Objekt-Methoden:
    public boolean isEmpty() { return first==-1; }
    public String toString() {...}
    ...
}
  
```

Implementierung von enqueue():

- Falls die Schlange leer war, muss first und last auf 0 gesetzt werden.
- Andernfalls ist das Feld a genau dann voll, wenn das Element x an der Stelle first eingetragen werden sollte.

- In diesem Fall legen wir ein Feld doppelter Größe an.
Die Elemente `a[first]`, `a[first+1]`, ..., `a[a.length-1]`, `a[0]`, `a[1]`, ..., `a[first-1]` kopieren wir nach `b[0]`, ..., `b[a.length-1]`.
- Dann setzen wir `first = 0`; `last = a.length` und `a = b`;
- Nun kann `x` an der Stelle `a[last]` abgelegt werden.

```
public void enqueue (int x) {
    if (first==-1) {
        first = last = 0;
    } else {
        int n = a.length;
        last = (last+1)%n;
        if (last == first) {
            int[] b = new int[2*n];
            for (int i=0; i<n; ++i) {
                b[i] = a[(first+i)%n];
            } // end for
            first = 0; last = n; a = b;
        } // end if and else
        a[last] = x;
    }
}
```

Implementierung von `dequeue()`:

- Falls nach Entfernen von `a[first]` die Schlange leer ist, werden `first` und `last` auf `-1` gesetzt.
- Andernfalls wird `first` um 1 (modulo der Länge von `a`) inkrementiert ...

```
public int dequeue () {
    int result = a[first];
    if (first == last) first = last = -1;
    else first = (first+1) % a.length;
    return result;
}
```

Diskussion:

- In dieser Implementierung von `dequeue()` wird der Platz für die Schlange nie verkleinert ...
- Fällt die Anzahl der Elemente in der Schlange unter ein Viertel der Länge des Felds `a`, können wir aber (wie bei Kellern) das Feld durch ein halb so großes ersetzen :-)

Achtung:

Die Elemente in der Schlange müssen aber jetzt nicht mehr nur am Anfang von a liegen
!!!

Zusammenfassung:

- Der Datentyp `List` ist nicht sehr **abstrakt**, dafür extrem flexibel
 \implies gut geeignet für **rapid prototyping**.
- Für die **nützlichen** (eher **:-)** abstrakten Datentypen `Stack` und `Queue` lieferten wir zwei Implementierungen:

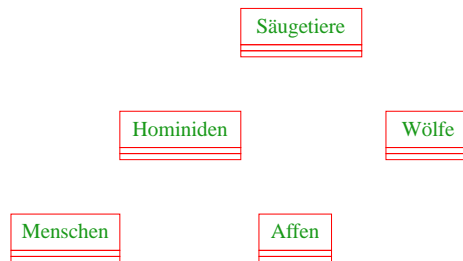
Technik	Vorteil	Nachteil
<code>List</code>	einfach	nicht-lokal
<code>int[]</code>	lokal	etwas komplexer

- **Achtung:** oft werden bei diesen Datentypen noch weitere Operationen zur Verfügung gestellt **:-)**

13 Vererbung

Beobachtung:

- Oft werden mehrere Klassen von Objekten benötigt, die zwar ähnlich, aber doch verschieden sind.

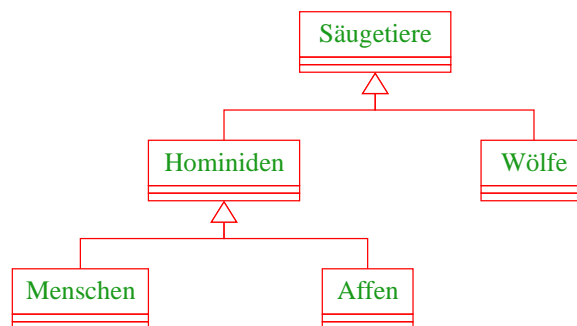


Idee:

- Finde Gemeinsamkeiten heraus!
- Organisiere in einer Hierarchie!
- Implementiere zuerst was allen gemeinsam ist!
- Implementiere dann nur noch den Unterschied!

⇒ inkrementelles Programmieren

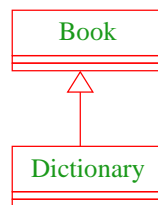
⇒ Software Reuse



Prinzip:

- Die Unterklasse verfügt über die Members der Oberklasse und eventuell auch noch über weitere.
- Das Übernehmen von Members der Oberklasse in die Unterklasse nennt man **Vererbung** (oder **inheritance**).

Beispiel:



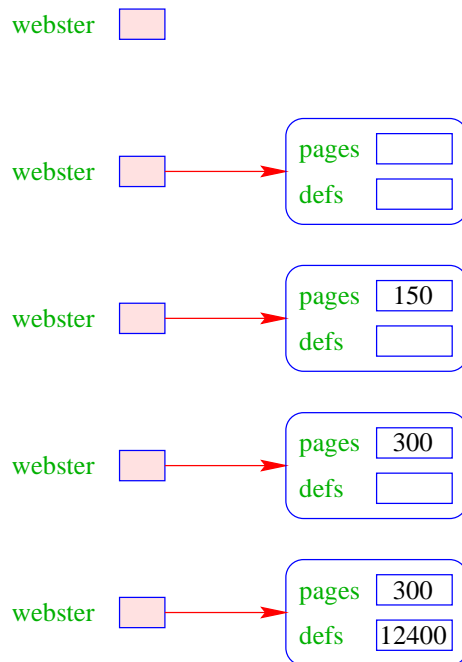
Implementierung:

```
public class Book {
    protected int pages;
    public Book() {
        pages = 150;
    }
    public void page_message() {
        System.out.print("Number of pages:\t"+pages+"\n");
    }
} // end of class Book
...

public class Dictionary extends Book {
    private int defs;
    public Dictionary(int x) {
        pages = 2*pages;
        defs = x;
    }
    public void defs_message() {
        System.out.print("Number of defs:\t\t"+defs+"\n");
        System.out.print("Defs per page:\t\t"+defs/pages+"\n");
    }
} // end of class Dictionary
```

- class **A** extends **B** { ... } deklariert die Klasse **A** als Unterklasse der Klasse **B**.
- Alle Members von **B** stehen damit automatisch auch der Klasse **A** zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse **sichtbar**.
- Als `private` deklarierte Members können dagegen in der Unterklasse **nicht** direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- Wenn ein Konstruktor der Unterklasse **A** aufgerufen wird, wird **implizit** zuerst der Konstruktor **B()** der Oberklasse aufgerufen.

Dictionary webster = new Dictionary(12400); liefert:



```
public class Words {  
    public static void main(String[] args) {  
        Dictionary webster = new Dictionary(12400);  
        webster.page_message();  
        webster.defs_message();  
    } // end of main  
} // end of class Words
```

- Das neue Objekt webster enthält die Attribute pages und defs, sowie die Objekt-Methoden page_message() und defs_message().
- Kommen in der Unterklasse nur weitere Members hinzu, spricht man von einer **is_a**-Beziehung. (Oft müssen aber Objekt-Methoden der Oberklasse in der Unterklasse **umdefiniert** werden.)
- Die Programm-Ausführung liefert:

```
Number of pages:      300  
Number of defs:      12400  
Defs per page:       41
```

13.1 Das Schlüsselwort `super`

- Manchmal ist es erforderlich, in der Unterklasse **explizit** die Konstruktoren oder Objekt-Methoden der Oberklasse aufzurufen. Das ist der Fall, wenn
 - Konstruktoren der Oberklasse aufgerufen werden sollen, die Parameter besitzen;
 - Objekt-Methoden oder Attribute der Oberklasse und Unterklasse gleiche Namen haben.
- Zur Unterscheidung der aktuellen Klasse von der Oberklasse dient das Schlüsselwort `super`.

... im Beispiel:

```
public class Book {
    protected int pages;
    public Book(int x) {
        pages = x;
    }
    public void message() {
        System.out.print("Number of pages:\t"+pages+"\n");
    }
} // end of class Book
...
public class Dictionary extends Book {
    private int defs;
    public Dictionary(int p, int d) {
        super(p);
        defs = d;
    }
    public void message() {
        super.message();
        System.out.print("Number of defs:\t\t"+defs+"\n");
        System.out.print("Defs per page:\t\t"+defs/pages+"\n");
    }
} // end of class Dictionary
```

- `super(...);` ruft den entsprechenden Konstruktor der Oberklasse auf.
- Analog gestattet `this(...);` den entsprechenden Konstruktor der eigenen Klasse aufzurufen **:-)**
- Ein solcher expliziter Aufruf muss stets ganz am Anfang eines Konstruktors stehen.
- Deklariert eine Klasse **A** einen Member **memb** gleichen Namens wie in einer Oberklasse, so ist nur noch der Member **memb** aus **A** sichtbar.

- Methoden mit unterschiedlichen Argument-Typen werden als verschieden angesehen :-)
- `super.memb` greift für das aktuelle Objekt `this` auf Attribute oder Objekt-Methoden `memb` der Oberklasse zu.
- Eine andere Verwendung von `super` ist **nicht gestattet**.

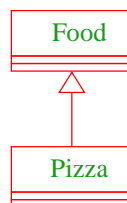
```
public class Words {
    public static void main(String[] args) {
        Dictionary webster = new Dictionary(540,36600);
        webster.message();
    } // end of main
} // end of class Words
```

- Das neue Objekt `webster` enthält die Attribute `pages` wie `defs`.
- Der Aufruf `webster.message()` ruft die Objekt-Methode der Klasse `Dictionary` auf.
- Die Programm-Ausführung liefert:

```
Number of pages:      540
Number of defs:      36600
Defs per page:       67
```

13.2 Private Variablen und Methoden

Beispiel:



Das Programm `Eating` soll die Anzahl der **Kalorien pro Mahlzeit** ausgeben.

```
public class Eating {
    public static void main (String[] args) {
        Pizza special = new Pizza(275);
        System.out.print("Calories per serving: " +
            special.calories_per_serving());
    } // end of main
} // end of class Eating
```

```

public class Food {
    private int CALORIES_PER_GRAM = 9;
    private int fat, servings;
    public Food (int num_fat_grams, int num_servings) {
        fat = num_fat_grams;
        servings = num_servings;
    }
    private int calories() {
        return fat * CALORIES_PER_GRAM;
    }
    public int calories_per_serving() {
        return (calories() / servings);
    }
} // end of class Food

public class Pizza extends Food {
    public Pizza (int amount_fat) {
        super (amount_fat,8);
    }
} // end of class Pizza

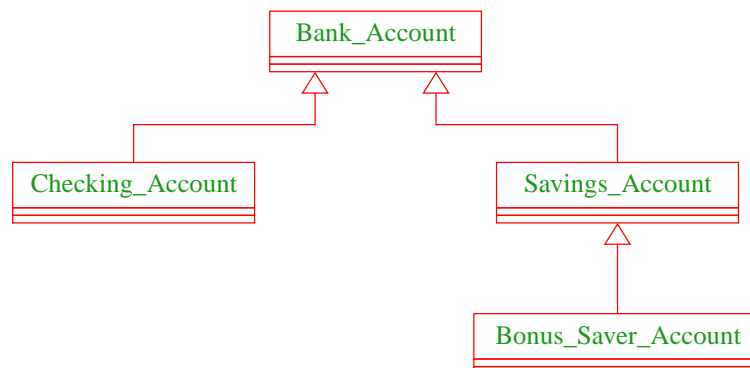
```

- Die Unterklasse Pizza verfügt über alle Members der Oberklasse Food – wenn auch nicht alle **direkt** zugänglich sind.
- Die Attribute und die Objekt-Methode calories() der Klasse Food sind privat, und damit für Objekte der Klasse Pizza verborgen.
- Trotzdem können sie von der public Objekt-Methode calories_per_serving benutzt werden.

... Ausgabe des Programms: Calories per serving: 309

13.3 Überschreiben von Methoden

Beispiel:



Aufgabe:

- Implementierung von einander abgeleiteter Formen von Bank-Konten.
- Jedes Konto kann eingerichtet werden, erlaubt Einzahlungen und Auszahlungen.
- Verschiedene Konten verhalten sich unterschiedlich in Bezug auf Zinsen und Kosten von Konto-Bewegungen.

Einige Konten:

```
public class Bank {
    public static void main(String[] args) {
        Savings_Account savings =
            new Savings_Account (4321, 5028.45, 0.02);
        Bonus_Saver_Account big_savings =
            new Bonus_Saver_Account (6543, 1475.85, 0.02);
        Checking_Account checking =
            new Checking_Account (9876, 269.93, savings);
        ...
    }
}
```

Einige Konto-Bewegungen:

```
savings.deposit (148.04);
big_savings.deposit (41.52);
savings.withdraw (725.55);
big_savings.withdraw (120.38);
checking.withdraw (320.18);
} // end of main
} // end of class Bank
```

Fehlt nur noch die Implementierung der Konten selbst :-)

```

public class Bank_Account {
    // Attribute aller Konten-Klassen:
    protected int account;
    protected double balance;
    // Konstruktor:
    public Bank_Account (int id, double initial) {
        account = id; balance = initial;
    }
    // Objekt-Methoden:
    public void deposit(double amount) {
        balance = balance+amount;
        System.out.print("Deposit into account "+account+"\n"
            +"Amount:\t\t"+amount+"\n"
            +"New balance:\t"+balance+"\n\n");
    }
    ...
}

```

- Anlegen eines Kontos `Bank_Account` speichert eine (hoffentlich neue) Konto-Nummer sowie eine Anfangs-Einlage.
- Die zugehörigen Attribute sind `protected`, d.h. können nur von Objekt-Methoden der Klasse bzw. ihrer Unterklassen modifiziert werden.
- die Objekt-Methode `deposit` legt Geld aufs Konto, d.h. modifiziert den Wert von `balance` und teilt die Konto-Bewegung mit.

```

    public boolean withdraw(double amount) {
        System.out.print("Withdrawal from account "+ account +"\n"
            +"Amount:\t\t"+ amount +"\n");
        if (amount > balance) {
            System.out.print("Sorry, insufficient funds...\n\n");
            return false;
        }
        balance = balance-amount;
        System.out.print("New balance:\t"+ balance +"\n\n");
        return true;
    }
} // end of class Bank_Account

```

- Die Objekt-Methode `withdraw()` nimmt eine Auszahlung vor.
- Falls die Auszahlung scheitert, wird eine Mitteilung gemacht.
- Ob die Auszahlung erfolgreich war, teilt der Rückgabewert mit.
- Ein `Checking_Account` verbessert ein normales Konto, indem im Zweifelsfall auf die Rücklage eines Sparkontos zurückgegriffen wird.

Ein Giro-Konto:

```
public class Checking_Account extends Bank_Account {
    private Savings_Account overdraft;
    // Konstruktor:
    public Checking_Account(int id, double initial,
                            Savings_Account savings) {
        super (id, initial);
        overdraft = savings;
    }
    ...
// modifiziertes withdraw():
public boolean withdraw(double amount) {
    if (!super.withdraw(amount)) {
        System.out.print("Using overdraft...\n");
        if (!overdraft.withdraw(amount-balance)) {
            System.out.print("Overdraft source insufficient.\n\n");
            return false;
        } else {
            balance = 0;
            System.out.print("New balance on account "+ account +": 0\n\n");
        } }
    return true;
}
} // end of class Checking_Account
```

- Die Objekt-Methode withdraw wird neu definiert, die Objekt-Methode deposit wird übernommen.
- Der Normalfall des Abhebens erfolgt (als Seiteneffekt) beim Testen der ersten if-Bedingung.
- Dazu wird die withdraw-Methode der Oberklasse aufgerufen.
- Scheitert das Abheben mangels Geldes, wird der Fehlbetrag vom Rücklagen-Konto abgehoben.
- Scheitert auch das, erfolgt keine Konto-Bewegung, dafür eine Fehlermeldung.
- Andernfalls sinkt der aktuelle Kontostand auf 0 und die Rücklage wird verringert.

Ein Sparbuch:

```
public class Savings_Account extends Bank_Account {
    protected double interest_rate;
    // Konstruktor:
    public Savings_Account (int id, double init, double rate) {
        super(id,init); interest_rate = rate;
    }
    // zusaetzliche Objekt-Methode:
    public void add_interest() {
        balance = balance * (1+interest_rate);
        System.out.print("Interest added to account: "+ account
            +"\nNew balance:\t"+ balance +"\n\n");
    }
} // end of class Savings_Account
```

- Die Klasse Savings_Account erweitert die Klasse Bank_Account um das zusätzliche Attribut double interest_rate (Zinssatz) und eine Objekt-Methode, die die Zinsen gutschreibt.
- Alle sonstigen Attribute und Objekt-Methoden werden von der Oberklasse geerbt.
- Die Klasse Bonus_Saver_Account erhöht zusätzlich den Zinssatz, führt aber Strafkosten fürs Abheben ein.

Ein Bonus-Sparbuch:

```
public class Bonus_Saver_Account extends Savings_Account {
    private int penalty;
    private double bonus;
    // Konstruktor:
    public Bonus_Saver_Account(int id, double init, double rate) {
        super(id, init, rate); penalty = 25; bonus = 0.03;
    }
    // Modifizierung der Objekt-Methoden:
    public boolean withdraw(double amount) {
        System.out.print("Penalty incurred:\t"+ penalty +"\n");
        return super.withdraw(amount+penalty);
    }
    ...
    public void add_interest() {
        balance = balance * (1+interest_rate+bonus);
        System.out.print("Interest added to account: "+ account
            +"\nNew balance:\t" + balance +"\n\n");
    }
} // end of class Bonus_Safer_Account
```

... als [Ausgabe](#) erhalten wir dann:

Deposit into account 4321
Amount: 148.04
New balance: 5176.49

Deposit into account 6543
Amount: 41.52
New balance: 1517.37

Withdrawal from account 4321
Amount: 725.55
New balance: 4450.94

Penalty incurred: 25
Withdrawal from account 6543
Amount: 145.38
New balance: 1371.9899999999998

Withdrawal from account 9876
Amount: 320.18
Sorry, insufficient funds...

Using overdraft...
Withdrawal from account 4321
Amount: 50.25
New balance: 4400.69

New balance on account 9876: 0

14 Abstrakte Klassen, finale Klassen und Interfaces

- Eine **abstrakte** Objekt-Methode ist eine Methode, für die keine Implementierung bereit gestellt wird.
- Eine Klasse, die abstrakte Objekt-Methoden enthält, heißt ebenfalls **abstrakt**.
- Für eine abstrakte Klasse können offenbar keine Objekte angelegt werden :-)
- Mit abstrakten können wir Unterklassen mit verschiedenen Implementierungen der gleichen Objekt-Methoden zusammenfassen.

Beispiel: Implementierung der **JVM**

```
public abstract class Instruction {
    protected static IntStack stack = new IntStack();
    protected static int pc = 0;
    public boolean halted() { return false; }
    abstract public int execute();
} // end of class Instruction
```

- Die Unterklassen von `Instruction` repräsentieren die Befehle der **JVM**.
- Allen Unterklassen gemeinsam ist eine Objekt-Methode `execute()` – immer mit einer anderen Implementierung :-)
- Die statischen Variablen der Oberklasse stehen sämtlichen Unterklassen zur Verfügung.
- Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- Für die abstrakte Methode muss der vollständige Kopf angegeben werden – inklusive den Parameter-Typen und den (möglicherweise) geworfenen Exceptions.
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier:
`boolean halted()`.
- Die angegebene Implementierung liefert eine **Default**-Implementierung für `boolean halted()`.
- Klassen, die eine andere Implementierung brauchen, können die Standard-Implementierung ja überschreiben :-)
- Die Methode `execute()` soll die Instruktion ausführen und als Rückgabe-Wert den `pc` des nächsten Befehls ausgeben.

Beispiel für eine Instruktion:

```
public final class Const extends Instruction {
    private int n;
    public Const(int x) { n=x; }
    public int execute() {
        stack.push(n);
        return ++pc;
    } // end of execute()
} // end of class Const
```

- Der Befehl **CONST** benötigt ein Argument. Dieses wird dem Konstruktor mitgegeben und in einer privaten Variable gespeichert.
- Die Klasse ist als **final** deklariert.
- Zu als **final** deklarierten Klassen dürfen keine Unterklassen deklariert werden !!!
- Aus Sicherheits- wie Effizienz-Gründen sollten so viele Klassen wie möglich als **final** deklariert werden ...
- Statt ganzer Klassen können auch einzelne Variablen oder Methoden als **final** deklariert werden.
- Finale Members dürfen nicht in Unterklassen umdefiniert werden.
- Finale Variablen dürfen zusätzlich nur initialisiert, aber nicht modifiziert werden ==
⇒ **Konstanten**.

... andere Instruktionen:

```
public final class Sub extends Instruction {
    public int execute() {
        final int y = stack.pop();
        final int x = stack.pop();
        stack.push(x-y); return ++pc;
    } // end of execute()
} // end of class Sub

public final class Halt extends Instruction {
    public boolean halted() {
        pc=0; stack = new IntStack(); return true;
    }
    public int execute() { return 0; }
} // end of class Halt
```

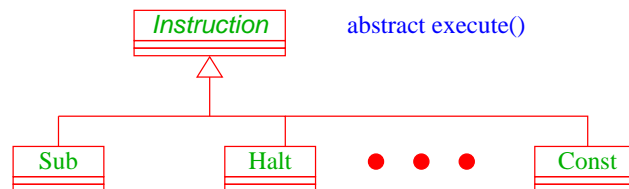
- In der Klasse **Halt** wird die Objekt-Methode **halted()** neu definiert.
- Achtung bei **Sub** mit der Reihenfolge der Argumente!

... die Funktion `main()` einer Klasse `Jvm`:

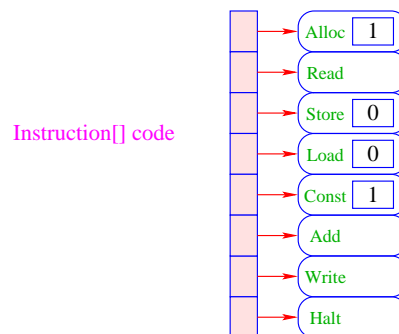
```
public static void main(String[] args) {
    Instruction[] code = getCode();
    Instruction ir = code[0];
    while(!ir.halted())
        ir = code[ir.execute()];
}
```

- Für einen vernünftigen Interpreter müssen wir natürlich auch in der Lage sein, ein **JVM**-Programm einzulesen, d.h. eine Funktion `getCode()` zu implementieren...

Die abstrakte Klasse `Instruction`:



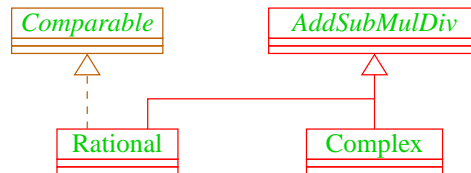
- Jede Unterklasse von `Instruction` verfügt über ihre eigene Methode `execute()`.
- In dem Feld `Instruction[] code` liegen Objekte aus solchen Unterklassen.



- Die Interpreter-Schleife ruft eine Methode `execute()` für die Elemente dieses Felds auf.
- Welche konkrete Methode dabei jeweils aufgerufen wird, hängt von der konkreten Klasse des jeweiligen Objekts ab, d.h. entscheidet sich erst zur Laufzeit.
- Das nennt man auch **dynamische Bindung**.

Leider (zum Glück?) lässt sich nicht die ganze Welt hierarchisch organisieren ...

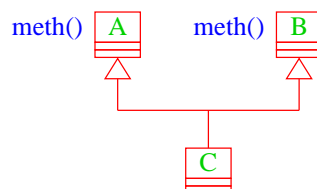
Beispiel:



AddSubMulDiv = Objekte mit Operationen `add()`, `sub()`, `mul()`, und `div()`

Comparable = Objekte, die eine `compareTo()`-Operation besitzen.

- Mehrere direkte Oberklassen einer Klasse führen zu konzeptuellen Problemen:
 - Auf welche Klasse bezieht sich `super` ?
 - Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren ?



- Kein Problem entsteht, wenn die Objekt-Methode `meth()` in allen Oberklassen abstrakt ist :-)
- oder zumindest nur in maximal einer Oberklasse eine Implementierung besitzt :-))

Ein **Interface** kann aufgefasst werden als eine abstrakte Klasse, wobei:

- alle Objekt-Methoden abstrakt sind;
- es keine Klassen-Methoden gibt;
- alle Variablen **Konstanten** sind.

Beispiel:

```
public interface Comparable {
    int compareTo(Object x);
}
```

- **Object** ist die gemeinsame Oberklasse aller Klassen.
- Methoden in Interfaces sind automatisch Objekt-Methoden und `public`.
- Es muss eine **Obermenge** der in Implementierungen geworfenen Exceptions angegeben werden.
- Evt. vorkommende Konstanten sind automatisch `public static`.

Beispiel (Forts.):

```
public class Rational extends AddSubMulDiv
    implements Comparable {
    private int zaehler, nenner;
    public int compareTo(Object cmp) {
        Rational fraction = (Rational) cmp;
        long left = zaehler * fraction.nenner;
        long right = nenner * fraction.zaehler;
        if (left == right) return 0;
        else if (left < right) return -1;
        else return 1;
    } // end of compareTo
    ...
} // end of class Rational
```

- `class A extends B implements B1, B2,...,Bk { ... }` gibt an, dass die Klasse **A** als Oberklasse **B** hat und zusätzlich die Interfaces **B1, B2,...,Bk** unterstützt, d.h. passende Objekt-Methoden zur Verfügung stellt.
- **Java** gestattet maximal eine Oberklasse, aber beliebig viele implementierte Interfaces.
- Die Konstanten des Interface können in implementierenden Klassen **direkt** benutzt werden.
- Interfaces können als Typen für formale Parameter, Variablen oder Rückgabewerte benutzt werden.
- Darin abgelegte Objekte sind dann stets aus einer implementierenden Klasse.
- Expliziter Cast in eine solche Klasse ist möglich (und leider auch oft nötig :-)
- Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.

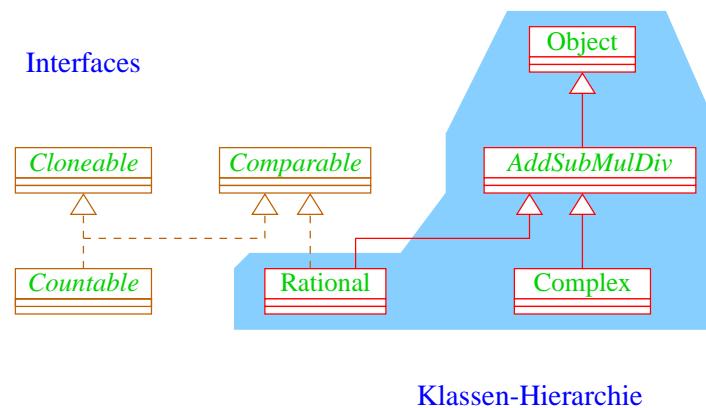
- Erweiternde Interfaces können Konstanten auch umdefinieren...
- (kommen Konstanten gleichen Namens in verschiedenen implementierten Interfaces vor, gibt's einen [Laufzeit-Fehler...](#))

Beispiel (Forts.):

```
public interface Countable extends Comparable, Cloneable {
    Countable next();
    Countable prev();
    int number();
}
```

- Das Interface Countable umfasst die (beide vordefinierten :-)) Interfaces Comparable und Cloneable.
- Das vordefinierte Interface Cloneable verlangt eine Objekt-Methode `public Object clone()` die eine Kopie des Objekts anlegt.
- Eine Klasse, die Countable implementiert, muss über die Objekt-Methoden `compareTo()`, `clone()`, `next()`, `prev()` und `number()` verfügen.

Übersicht:



15 Polymorphie

Problem:




- Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur grade nochmal definieren :-)

15.1 Unterklassen-Polymorphie

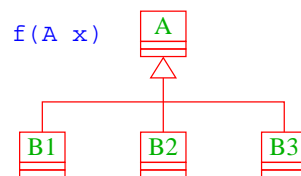
Idee:

- Eine Operation `meth (A x)` lässt sich auch mit einem Objekt aus einer Unterklasse von `A` aufrufen !!!
- Kennen wir eine gemeinsame Oberklasse `Base` für alle möglichen aktuellen Parameter unserer Operation, dann definieren wir `meth` einfach für `Base ...`
- Eine Funktion, die für mehrere Argument-Typen definiert ist, heißt auch **polymorph**.

Statt:

`f (B1 x)`  `f (B2 x)`  `f (B3 x)` 

... besser:



Fakt:

- Die Klasse `Object` ist eine gemeinsame Oberklasse für **alle** Klassen.
- Eine Klasse ohne angegebene Oberklasse ist eine direkte Unterklasse von `Object`.
- Einige nützliche Methoden der Klasse `Object` :
 - `String toString()` liefert (irgendeine) Darstellung als `String`;

- `boolean equals(Object obj)` testet auf **Objekt-Identität** oder Referenz-Gleichheit:

```
public boolean equals(Object obj) {
    return this==obj;
}
```

...

- `int hashCode()` liefert eine eindeutige Nummer für das Objekt.
- ... viele weitere **geheimnisvolle Methoden**, die u.a. mit **↑paralleler Programm-Ausführung** zu tun haben :-)

Achtung:

Object-Methoden können aber (und sollten evt.:-) in Unterklassen durch geeignetere Methoden überschrieben werden.

Beispiel:

```
public class Poly {
    public String toString() { return "Hello"; }
}
public class PolyTest {
    public static String addWorld(Object x) {
        return x.toString()+" World!";
    }
    public static void main(String[] args) {
        Object x = new Poly();
        System.out.print(addWorld(x)+"\n");
    }
}
```

... liefert:

Hello World!

- Die Klassen-Methode `addWorld()` kann auf jedes Objekt angewendet werden.
- Die Klasse `Poly` ist eine Unterklasse von `Object`.
- Einer Variable der Klasse **A** kann ein Objekt **jeder Unterklasse** von **A** zugewiesen werden.
- Darum kann `x` das neue `Poly`-Objekt aufnehmen :-)

Bemerkung:

- Die Klasse Poly enthält keinen explizit definierten Konstruktor.
- Eine Klasse A, die keinen anderen Konstruktor besitzt, enthält **implizit** den trivialen Konstruktor `public A () {}`.

Achtung:

```
public class Poly {
    public String greeting() {
        return "Hello";
    }
}
public class PolyTest {
    public static void main(String[] args) {
        Object x = new Poly();
        System.out.print(x.greeting()+" World!\n");
    }
}
```

... liefert ...

... einen Compiler-Fehler:

```
Method greeting() not found in class java.lang.Object.
    System.out.print(x.greeting()+" World!\n");
                       ^
1 error
```

- Die Variable x ist als Object deklariert.
- Der Compiler weiss nicht, ob der aktuelle Wert von x ein Objekt aus einer Unterklasse ist, in welcher die Objekt-Methode `greeting()` definiert ist.
- Darum lehnt er dieses Programm ab.

Ausweg:

- Benutze einen expliziten `cast` in die entsprechende Unterklasse!

```
public class Poly {
    public String greeting() { return "Hello"; }
}
public class PolyTest {
    public void main(String[] args) {
        Object x = new Poly();
        if (x instanceof Poly)
            System.out.print(((Poly) x).greeting()+" World!\n");
        else
            System.out.print("Sorry: no cast possible!\n");
    }
}
```

Fazit:

- Eine Variable `x` einer Klasse `A` kann Objekte `b` aus sämtlichen Unterklassen `B` von `A` aufnehmen.
- Durch diese Zuweisung vergisst `Java` die Zugehörigkeit zu `B`, da `Java` alle Werte von `x` als Objekte der Klasse `A` behandelt.
- Mit dem Ausdruck `x instanceof B` können wir zur `Laufzeit` die Klassenzugehörigkeit von `x` testen `;-)`
- Sind wir uns sicher, dass `x` aus der Klasse `B` ist, können wir in diesen Typ `casten`.
- Ist der aktuelle Wert der Variablen `x` bei der Überprüfung tatsächlich ein Objekt (einer Unterklasse) der Klasse `B`, liefert der Ausdruck genau dieses Objekt zurück. Andernfalls wird eine `↑Exception` ausgelöst `;-)`

Beispiel: Unsere Listen

```
public class List {
    public Object info;
    public List next;
    public List(Object x, List l) {
        info=x; next=l;
    }
    public void insert(Object x) {
        next = new List(x,next);
    }
    public void delete() {
        if (next!=null) next=next.next;
    }
    ...
    public String toString() {
        String result = "["+info;
        for (List t=next; t!=null; t=t.next)
            result=result+", "+t.info;
        return result+"]";
    }
    ...
} // end of class List
```

- Die Implementierung funktioniert ganz analog zur Implementierung für int.
- Die toString()-Methode ruft implizit die (stets vorhandene) toString()-Methode für die Listen-Elemente auf.

... aber Achtung:

```
...
Poly x = new Poly();
List list = new List (x);
x = list.info;
System.out.print(x+"\n");
...
```

liefert ...

... einen **Compiler-Fehler**, da der Variablen x nur Objekte einer Unterklasse von Poly zugewiesen werden dürfen.

Stattdessen müssen wir schreiben:


```

...
Poly x = new Poly();
List list = new List (x);
x = (Poly) list.info;
System.out.print(x+"\n");
...

```

Das ist hässlich !!! Geht das nicht besser ???

15.2 Generische Klassen

Idee:

- Seit Version 1.5 verfügt Java über generische Klassen ...
- Anstatt das Attribut info als Object zu deklarieren, geben wir der Klasse einen Typ-Parameter T für info mit !!!
- Bei Anlegen eines Objekts der Klasse List bestimmen wir, welchen Typ T und damit info haben soll ...

Beispiel: Unsere Listen

```

public class List<T> {
    public T info;
    public List<T> next;
    public List (T x, List<T> l) {
        info=x; next=l;
    }
    public void insert(t x) {
        next = new List<T> (x,next);
    }
    public void delete() {
        if (next!=null) next=next.next;
    }
    ...

public static void main (String [] args) {
    List<Poly> list = new List<Poly> (new Poly(),null);
    System.out.print (list.info.greeting()+"\n");
}
} // end of class List

```

- Die Implementierung funktioniert ganz analog zur Implementierung für `Object`.
- Der Compiler weiß aber nun in `main`, dass `list` vom Typ `List` ist mit Typ-Parameter `T = Poly`.
- Deshalb ist `list.info` vom Typ `Poly` :-)
- Folglich ruft `list.info.greeting()` die entsprechende Methode der Klasse `Poly` auf :-))

Bemerkungen:

- Typ-Parameter dürfen nur in den Typen von Objekt-Attributen und Objekt-Methoden verwendet werden !!!
- Jede Unterklasse einer parametrisierten Klasse muss mindestens die gleichen Parameter besitzen:

`A<S,T> extends B<T>` ist erlaubt :-)

`A<S> extends B<S,T>` ist verboten :-)

- `Poly()` ist eine Unterklasse von `Object`; aber `List<Poly>` ist **keine** Unterklasse von `List<Object>` !!!

Bemerkungen (Forts.):

- Für einen Typ-Parameter `T` kann man auch eine Oberklasse oder ein Interface angeben, das `T` auf jeden Fall erfüllen soll ...

```
public interface Executable {
    void execute ();
}
public class ExecutableList<E extends Executable> {
    E element;
    ExecutableList<E> next;
    void executeAll () {
        element.execute ();
        if (next == null) return;
        else next.executeAll ();
    }
}
```

Bemerkungen (Forts.):

- Beachten Sie, dass hier ebenfalls das Schlüsselwort `extends` benutzt wird!
- Auch gelten hier weitere Beschränkungen, wie eine parametrisierte Klasse eine Oberklasse sein kann :-)

- Auch Interfaces können parametrisiert werden.
- Insbesondere kann **Comparable** parametrisiert werden – und zwar mit der Klasse, mit deren Objekten man vergleichen möchte ...

```
public class Test implements Comparable<Test> {
    public int compareTo (Test x) { return 0; }
}
```

15.3 Wrapper-Klassen

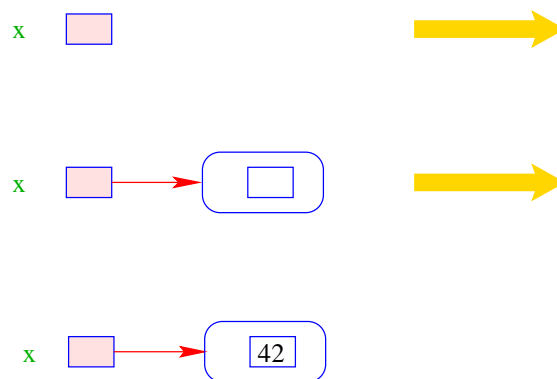
... bleibt ein Problem:

- Der Datentyp `String` ist eine Klasse;
- Felder sind Klassen; **aber**
- **Basistypen** wie `int`, `boolean`, `double` sind keine Klassen!
(Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl :-)

Ausweg:

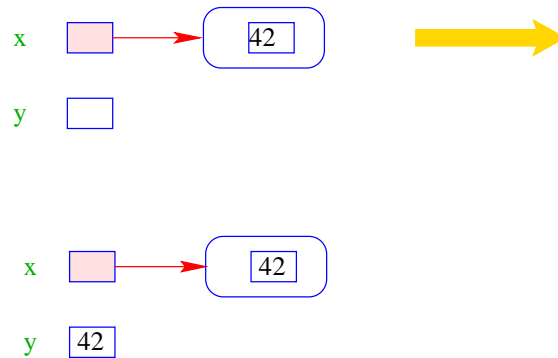
- Wickle die Werte eines Basis-Typs in ein Objekt ein!
⇒ **Wrapper-Objekte** aus **Wrapper-Klassen**.

Die Zuweisung `Integer x = new Integer(42);` bewirkt:

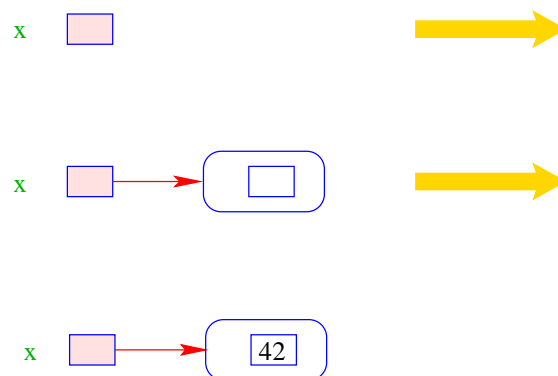


Eingewickelte Werte können auch wieder ausgewickelt werden.

Seit **Java 1.5** erfolgt bei einer Zuweisung `int y = x;` eine **automatische Konvertierung**:



Umgekehrt wird bei Zuweisung eines int-Werts an eine Integer-Variable: `Integer x = 42;` automatisch der Konstruktor aufgerufen:



Gibt es erst einmal die Klasse Integer, lassen sich dort auch viele andere nützliche Dinge ablegen.

Zum Beispiel:

- `public static int MIN_VALUE = -2147483648;` liefert den kleinsten int-Wert;
- `public static int MAX_VALUE = 2147483647;` liefert den größten int-Wert;
- `public static int parseInt(String s) throws NumberFormatException;` berechnet aus dem String-Objekt s die dargestellte Zahl — sofern s einen int-Wert darstellt.

Andernfalls wird eine **exception** geworfen :-)

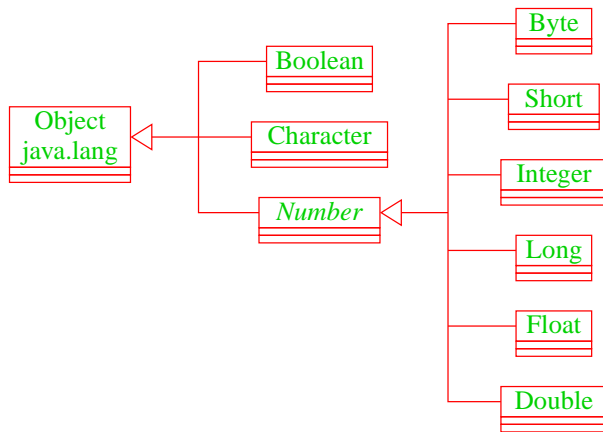
Bemerkungen:

- Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- Dieser Konstruktor liefert zu einem String-Objekt s ein Integer-Objekt, dessen Wert s darstellt.

- `public boolean equals(Object obj);` liefert true genau dann wenn obj den gleichen int-Wert enthält.

Ähnliche Wrapper-Klassen gibt es auch für die übrigen Basistypen ...

Wrapper-Klassen:



- Sämtliche Wrapper-Klassen für Typen `type` (außer char) verfügen über
 - Konstruktoren aus Basiswerten bzw. String-Objekten;
 - eine statische Methode `type parseType(String s);`
 - eine Methode `boolean equals(Object obj)` (auch Character).
- Bis auf Boolean verfügen alle über Konstanten `MIN_VALUE` und `MAX_VALUE`.
- Character enthält weitere Hilfsfunktionen, z.B. um Ziffern zu erkennen, Klein- in Großbuchstaben umzuwandeln ...
- Die numerischen Wrapper-Klassen sind in der gemeinsamen Oberklasse `Number` zusammengefasst.
- Diese Klasse ist `↑abstrakt` d.h. man kann keine `Number`-Objekte anlegen.

Spezialitäten:

- Double und Float enthalten zusätzlich die Konstanten

```

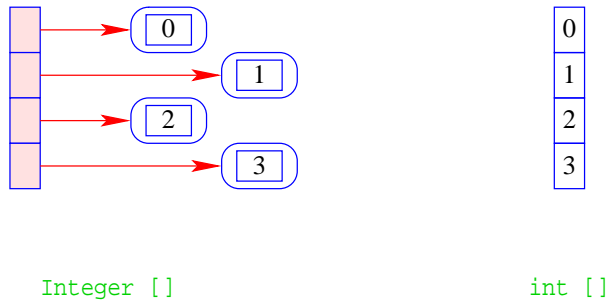
NEGATIVE_INFINITY = -1.0/0
POSITIVE_INFINITY = +1.0/0
NaN                = 0.0/0
  
```

- Zusätzlich gibt es die Tests
 - `public static boolean isInfinite(double v);`
 - `public static boolean isNaN(double v);`
 - (analog für float)

- `public boolean isInfinite();`
`public boolean isNaN();`

mittels derer man auf (Un)Endlichkeit der Werte testen kann.

Vergleich Integer mit int:



- + Integers können in polymorphen Datenstrukturen hausen.
- Sie benötigen mehr als doppelt so viel Platz.
- Sie führen zu vielen kleinen (evt.) über den gesamten Speicher verteilten Objekten \Rightarrow schlechteres Cache-Verhalten.