Informatik 1

Wintersemester 2011/2012

Helmut Seidl

Institut für Informatik
TU München

0 Allgemeines

Inhalt dieser Vorlesung:

- Einführung in Grundkonzepte der Informatik;
- Einführung in Denkweisen der Informatik;
- Programmieren in Java.

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.

$\Longrightarrow$

Ein Algorithmus beschreibt eine Funktion: $f : E \to A$,
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 $ggT(a, b)$.

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.

Ein Programm ist die Formulierung eines Algorithmus in einer Programmiersprache.

Die Formulierung gestattet (hoffentlich) eine maschinelle Ausgeführung.

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.

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.

$\Longrightarrow$ 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 dekorieren !

MiniJava-Programme sind ausführbares Java.

Man muss sie nur geeignet dekorieren !

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:

- 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
   "uber mehrere Zeilen! */
```

Das Programm `GGT` kann nun übersetzt und dann ausgeführt werden. Auf der Kommandozeile sieht das so aus:

```
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.

$\Longrightarrow$ formalisierter als natürliche Sprache

$\Longrightarrow$ 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).

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`

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:

name ::= 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.

- `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:

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

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen Konstanten.
Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

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

- 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[a] (↑Automatentheorie).

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

[a] Gelegentlich sind auch ϵ, d.h. das "leere Wort" sowie $\emptyset$, d.h. die leere Menge zugelassen.

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

- reservierten Wörtern,
- Namen,
- Konstanten

Ignorierung von

- White Space,
- Kommentaren

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

$\Longrightarrow$ 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`

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, angewand 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.

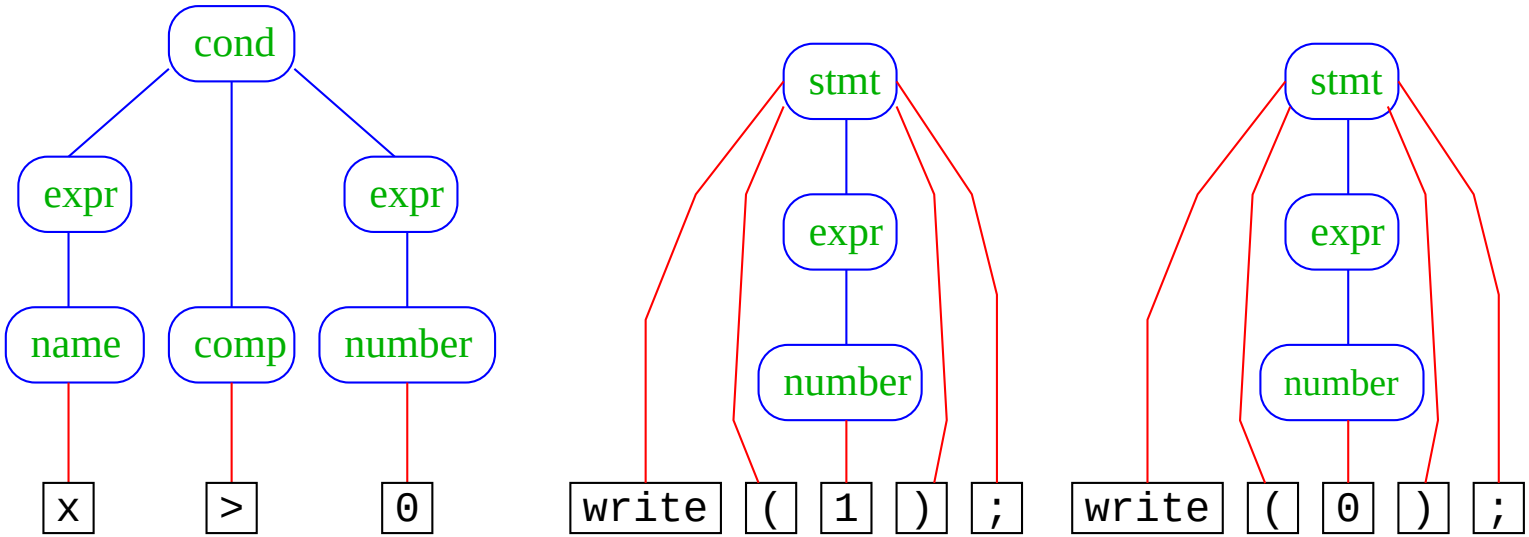
Puh!!! Geschafft ...

Beispiel:

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

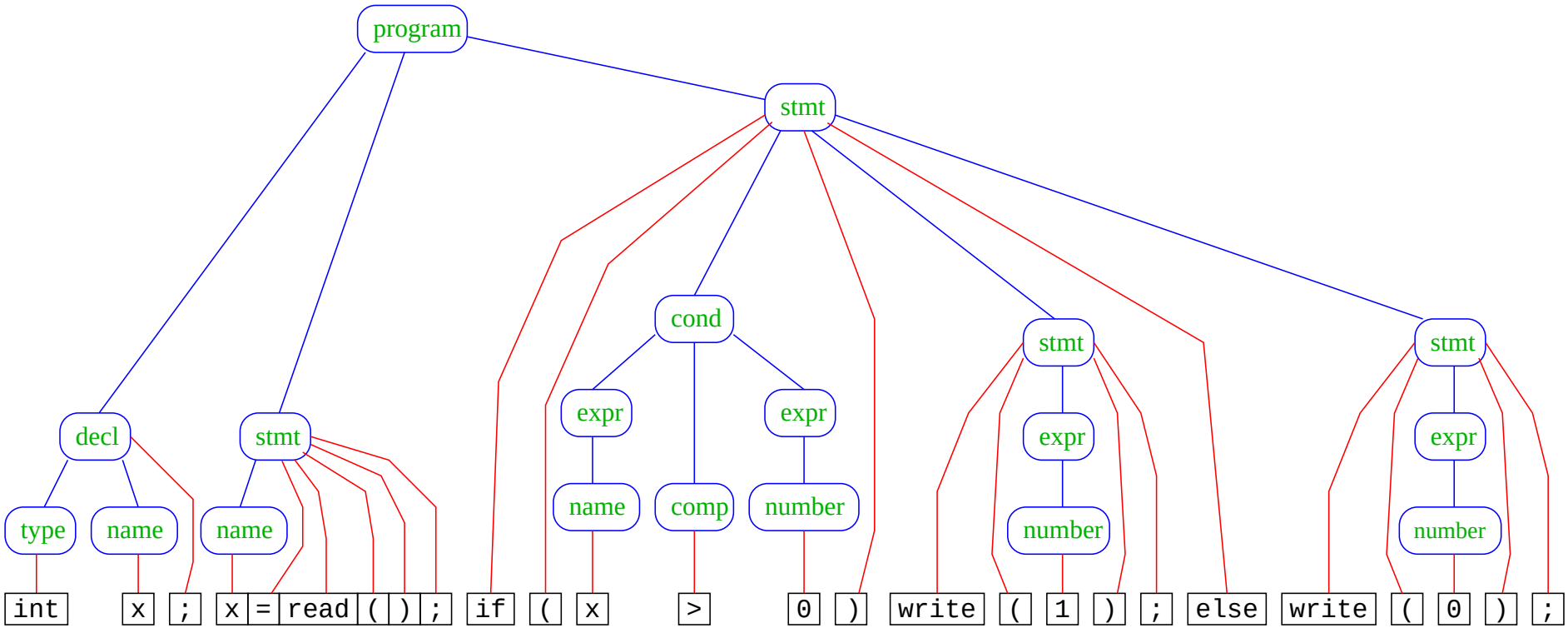
Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume:

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



Blätter: Wörter/Tokens

innere Knoten: Namen von Programm-Bestandteilen


program
stmt
decl
stmt
cond
stmt
stmt
type
name
name
expr
expr
expr
expr
name
comp
number
number
number
int
x
;
x
=
read
(
)
;
if
(
x
>
0
)
write
(
1
)
;
else
write
(
0
)
;

Bemerkungen:

- Die vorgestellte Methode der Beschreibung von Syntax heißt EBNF-Notation (Extended Backus Naur Form 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.

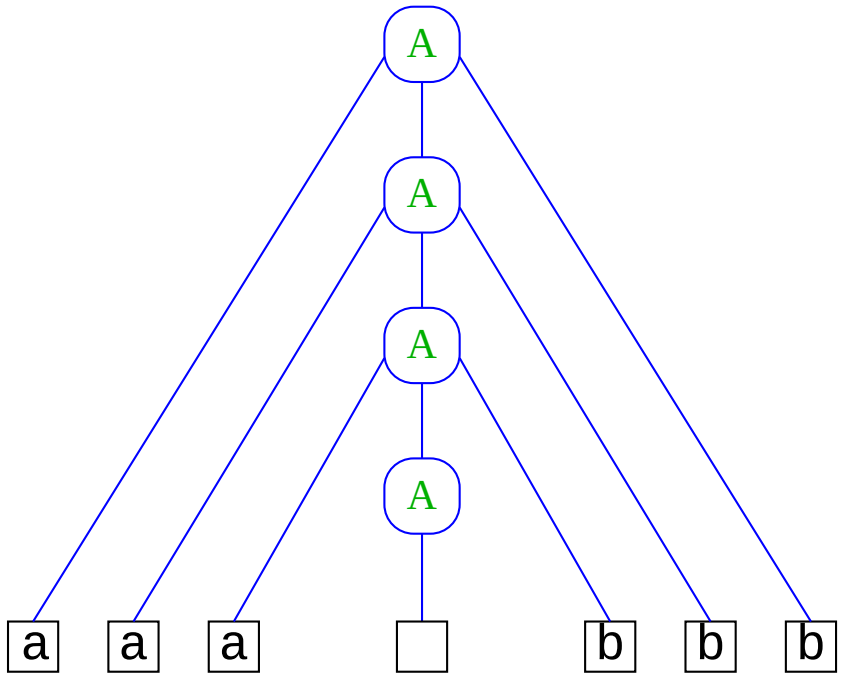
Beispiel:

$$\mathcal{L} = \{\epsilon, \texttt{ab}, \texttt{aabb}, \texttt{aaabbb}, \ldots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

$$\text{A} \quad ::= \quad (\ \texttt{a}\ \text{A}\ \texttt{b}\ )?$$

Syntax-Baum für das Wort `aaabbb` :

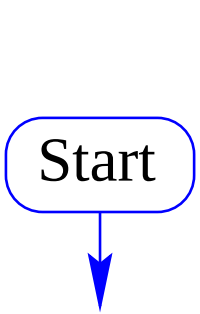


Für $\mathcal{L}$ gibt es aber keinen regulären Ausdruck!!! (↑Automatentheorie)

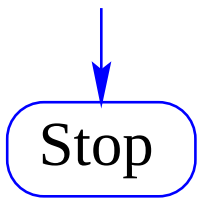
4 Kontrollfluss-Diagramme

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

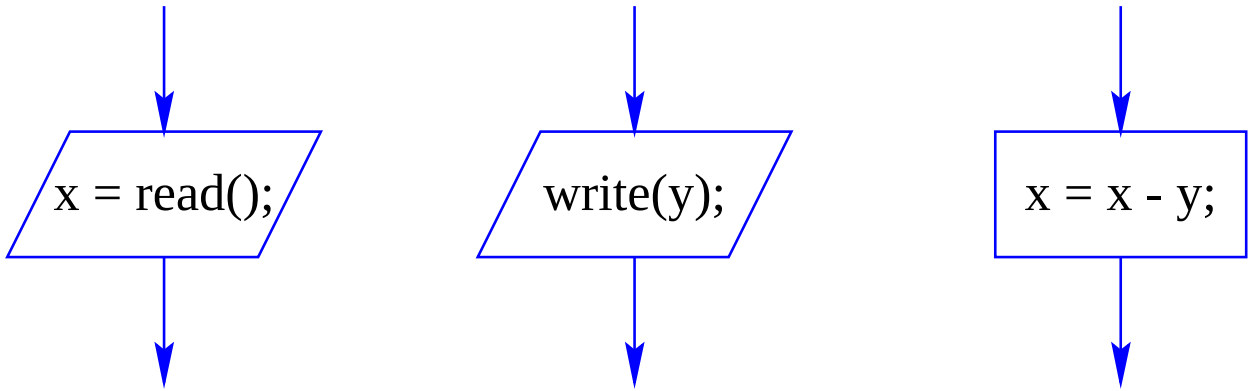
Ingredienzien:



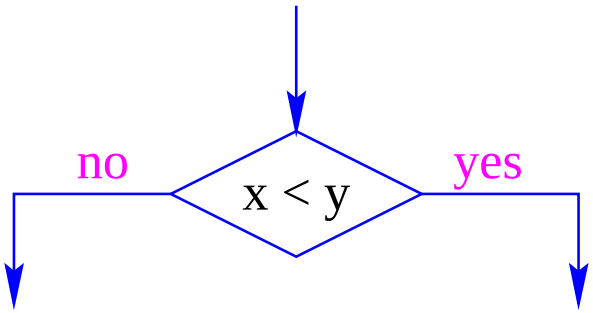
Startknoten



Endknoten

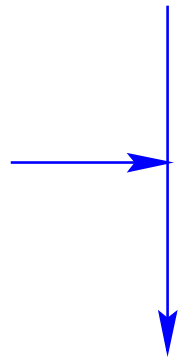


Eingabe Ausgabe Zuweisung



bedingte Verzweigung

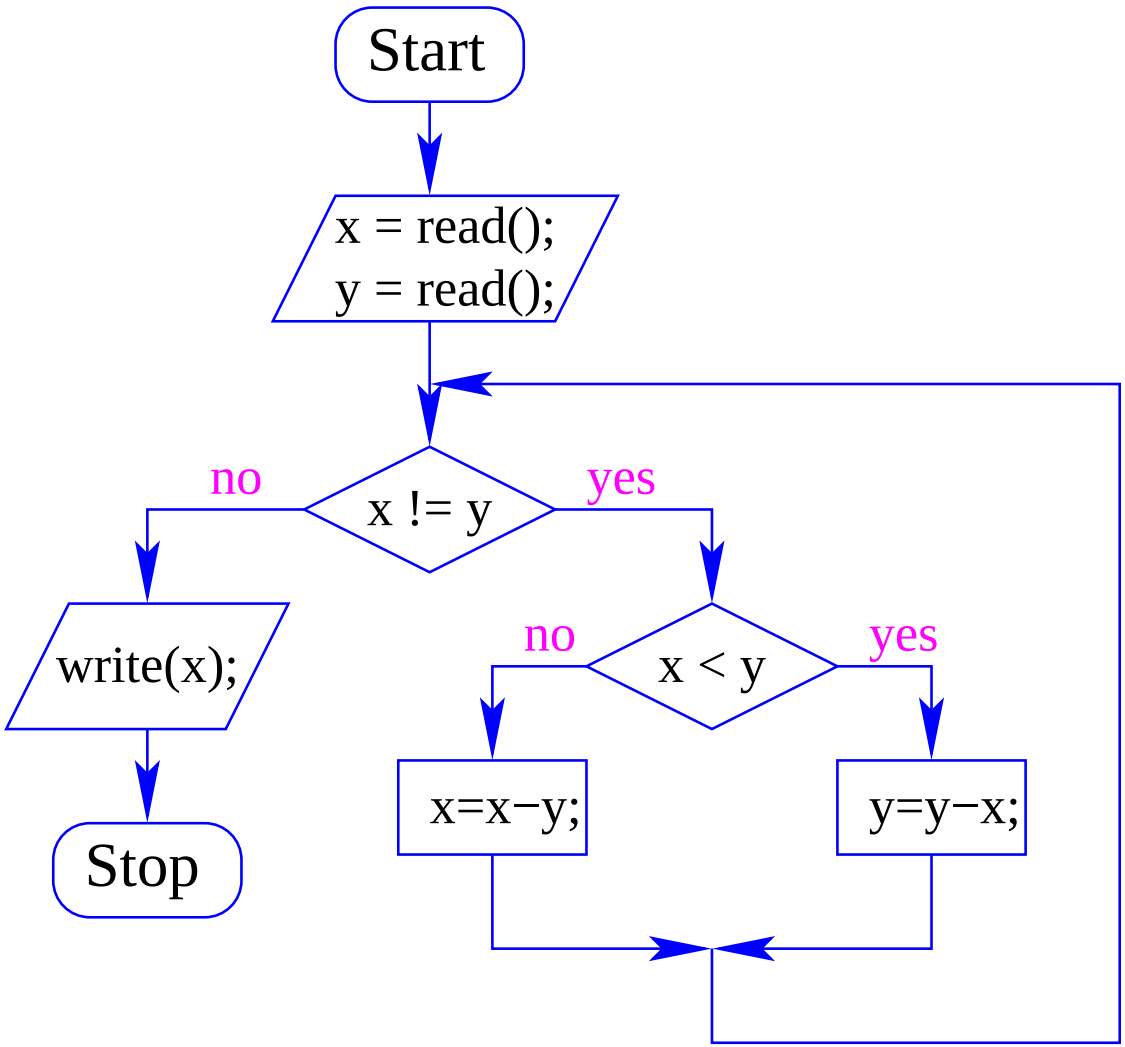
Kante



Zusammenlauf

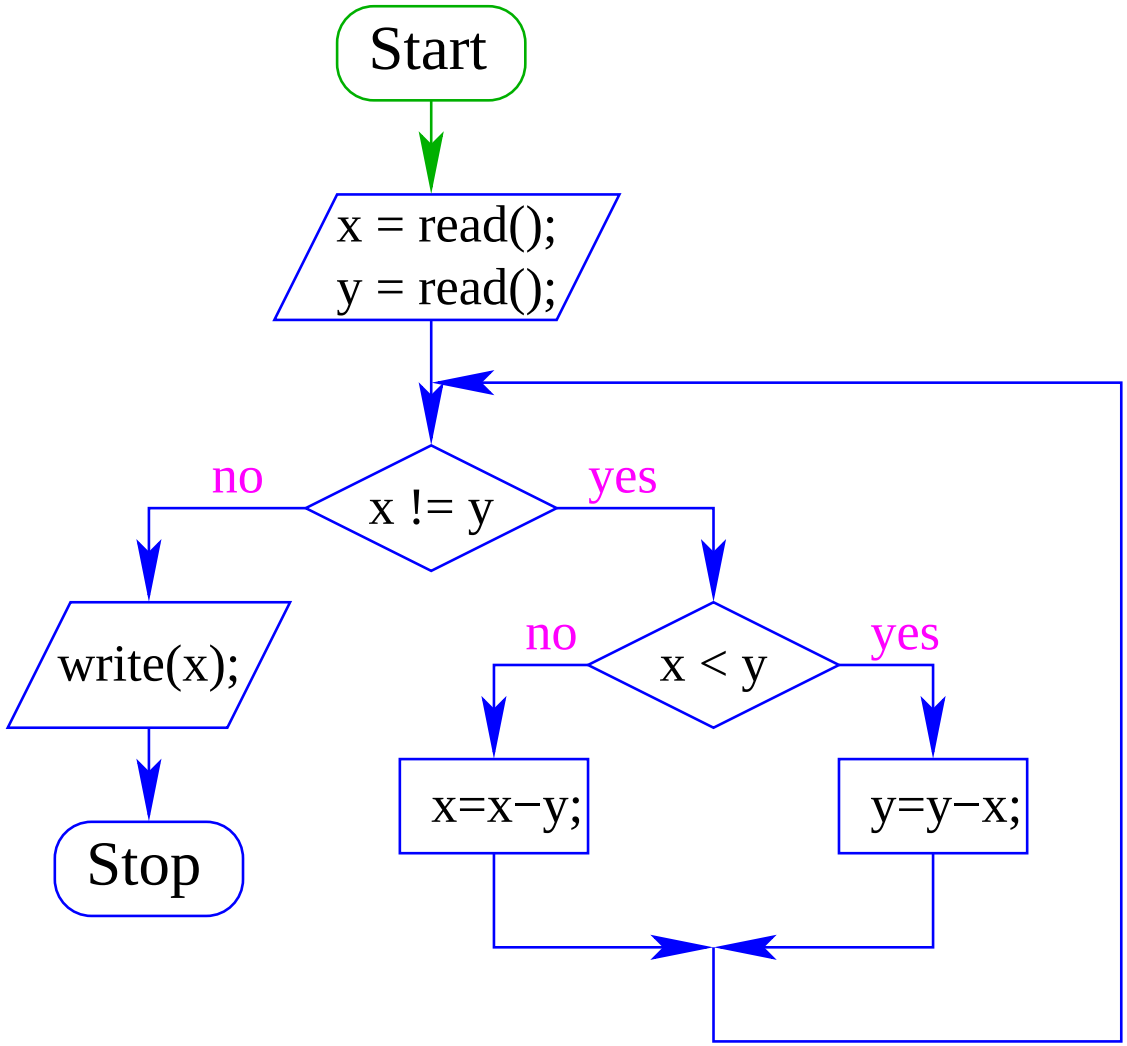
Beispiel:

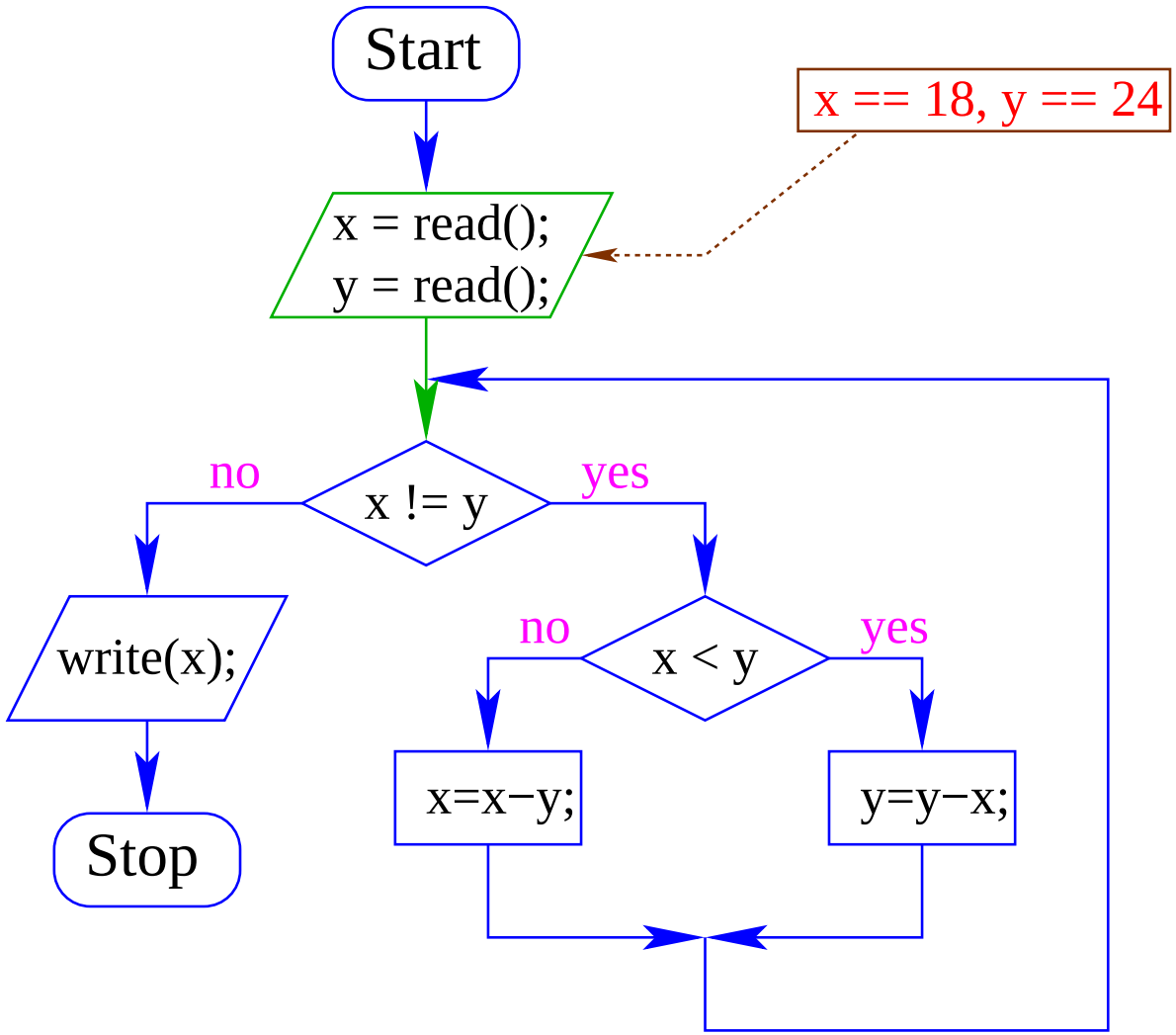
```
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```

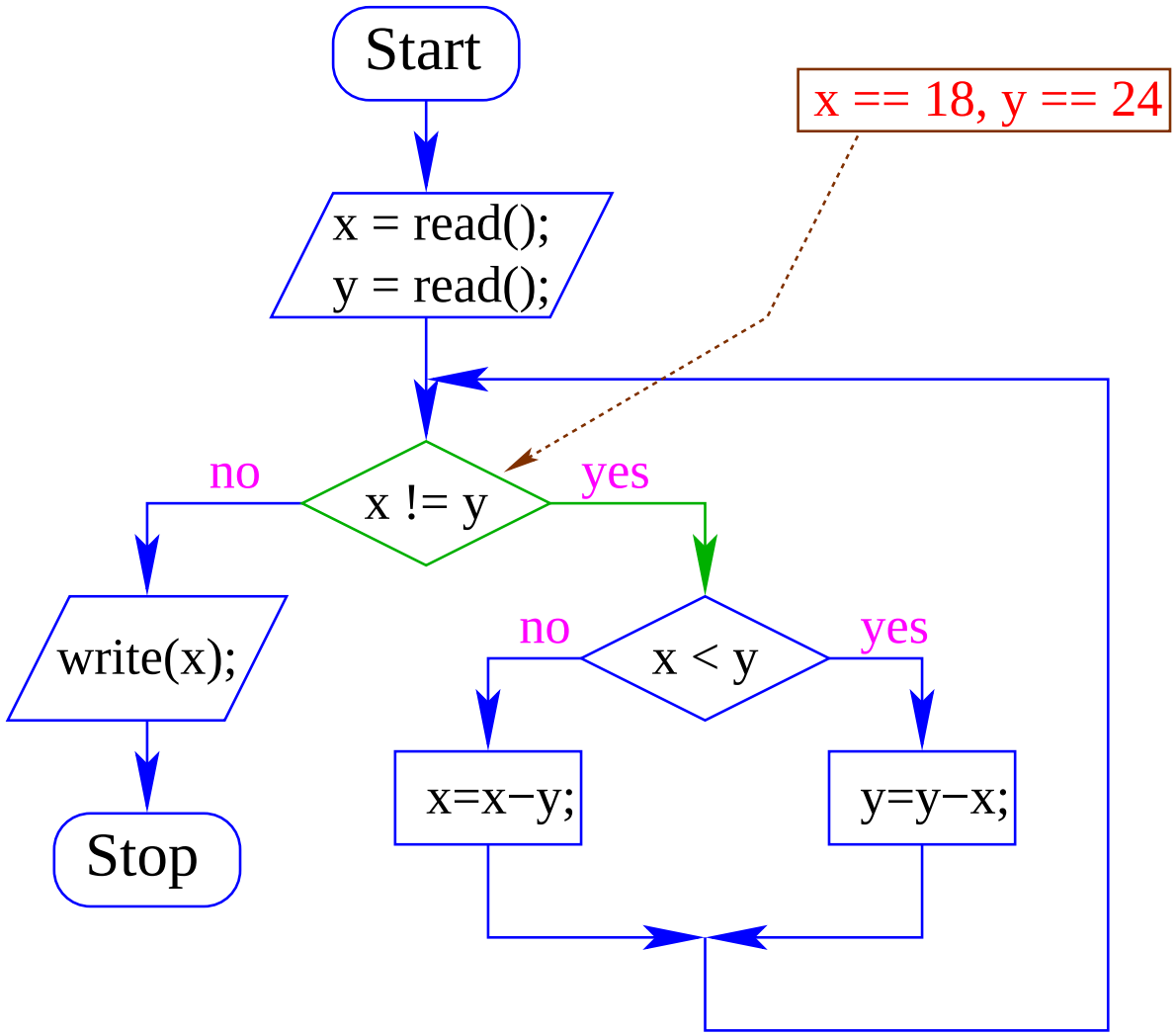


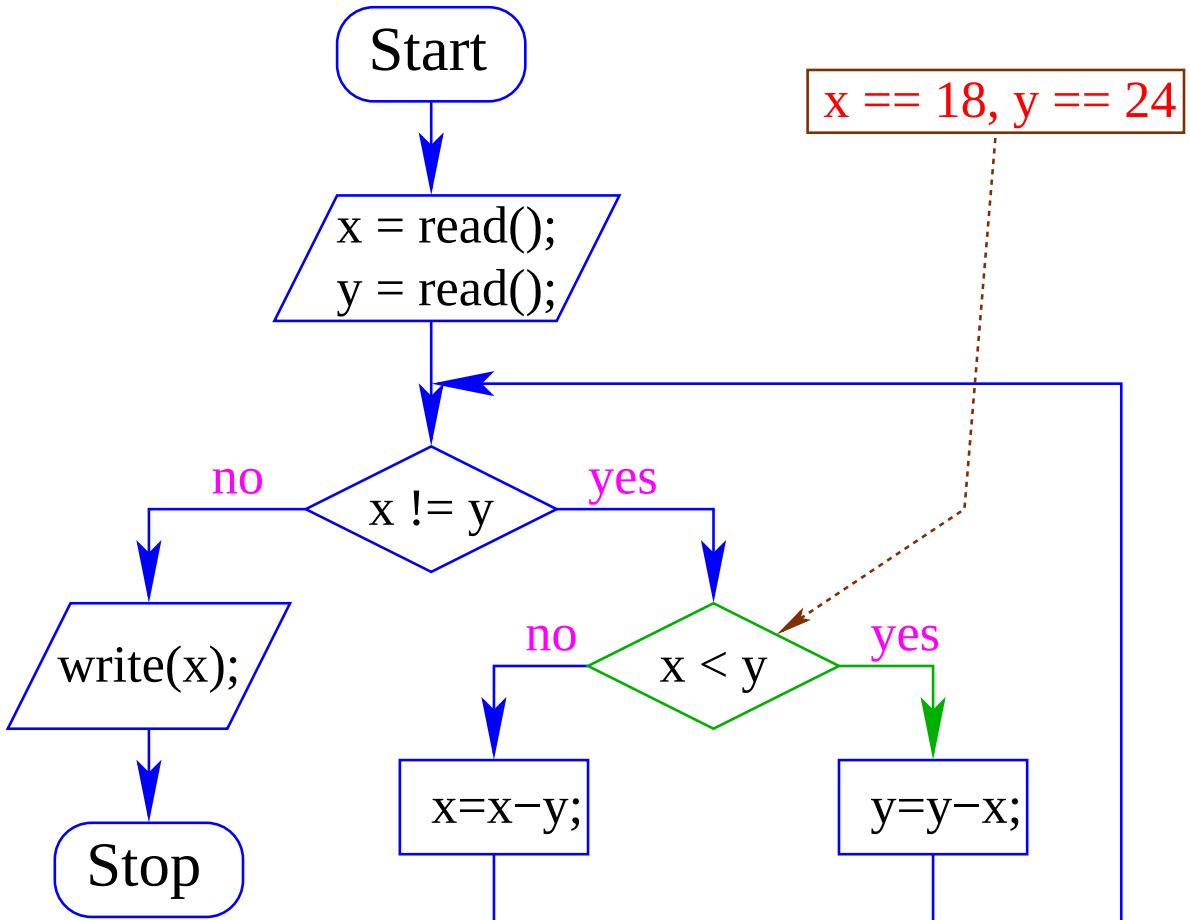
- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollfluss-Diagramm vom Startknoten zum Endknoten.
- Die Deklarationen von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer dem Start- und Endknoten) sind die dabei auszuführenden Operationen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

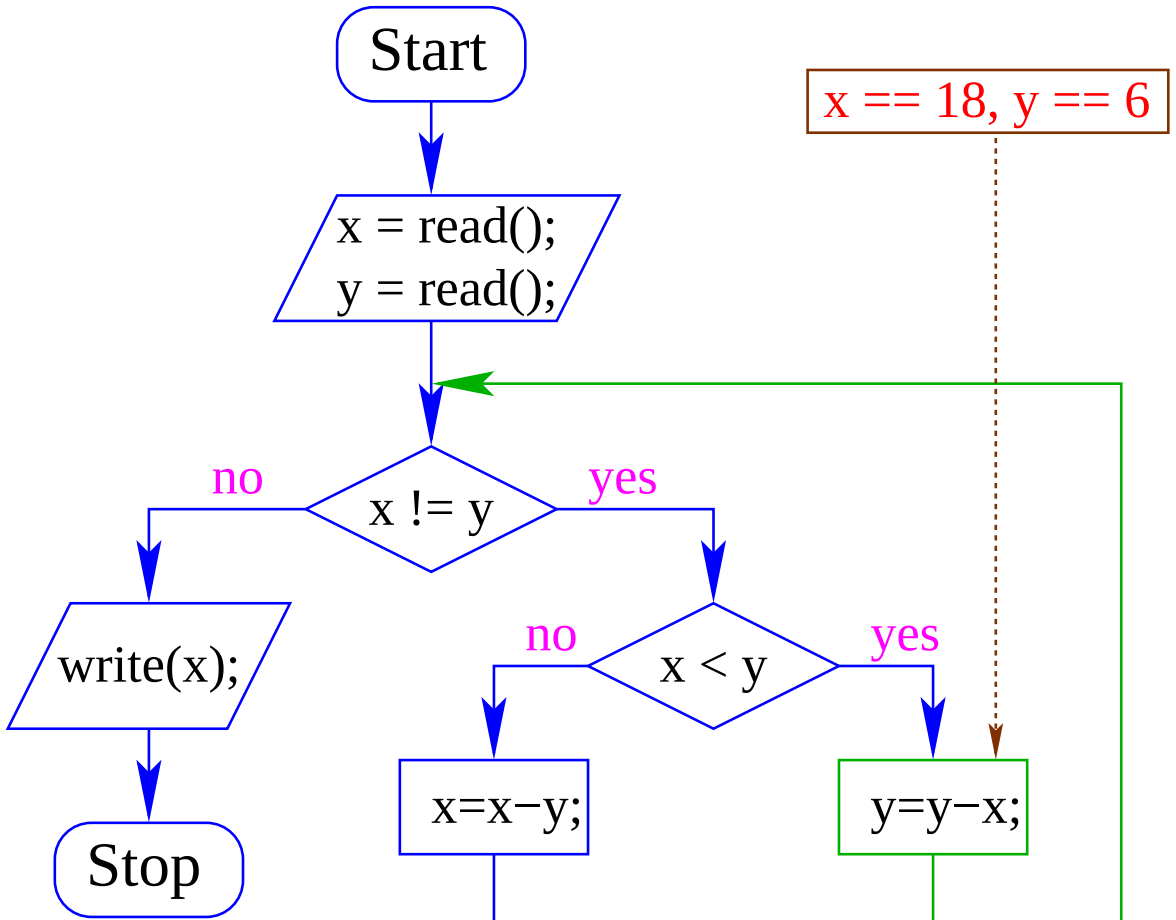
$\Longrightarrow$ operationelle Semantik

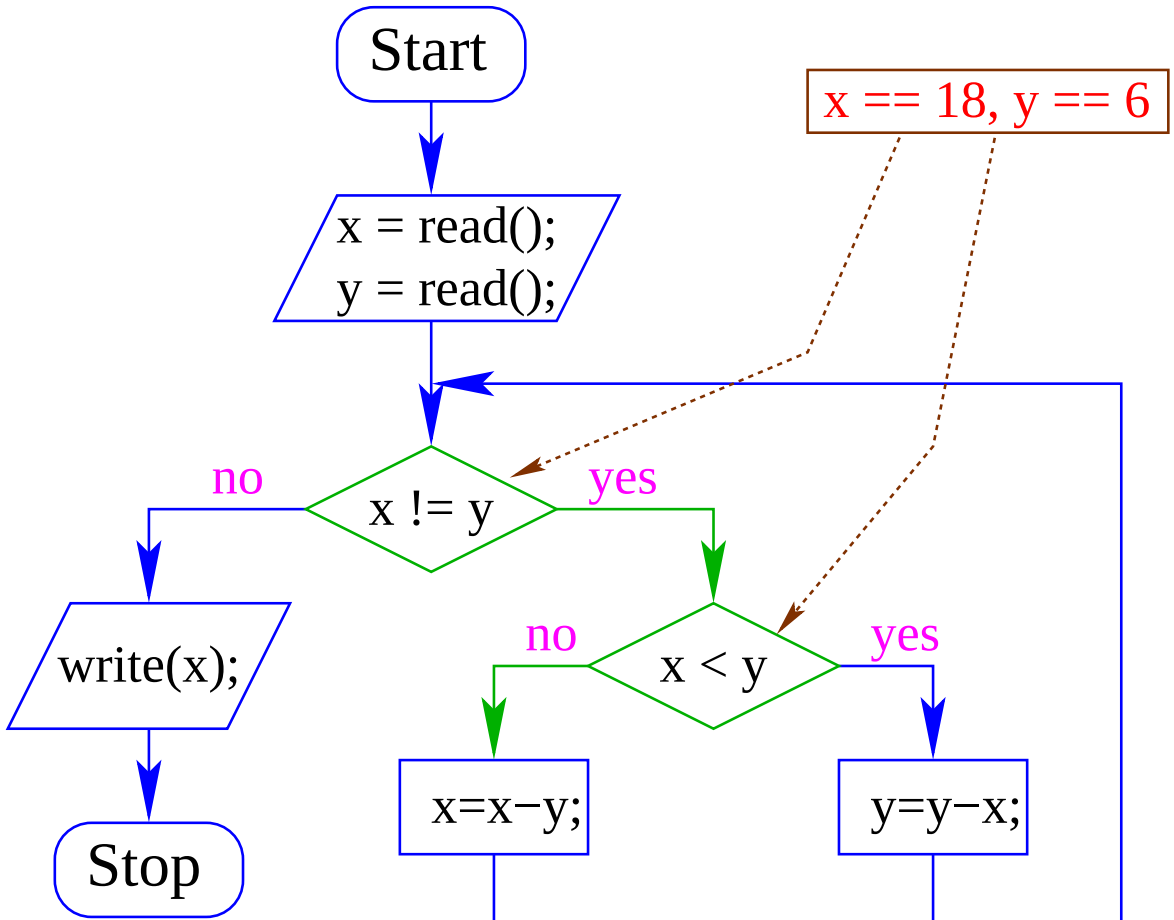

Start
x = read();
y = read();
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

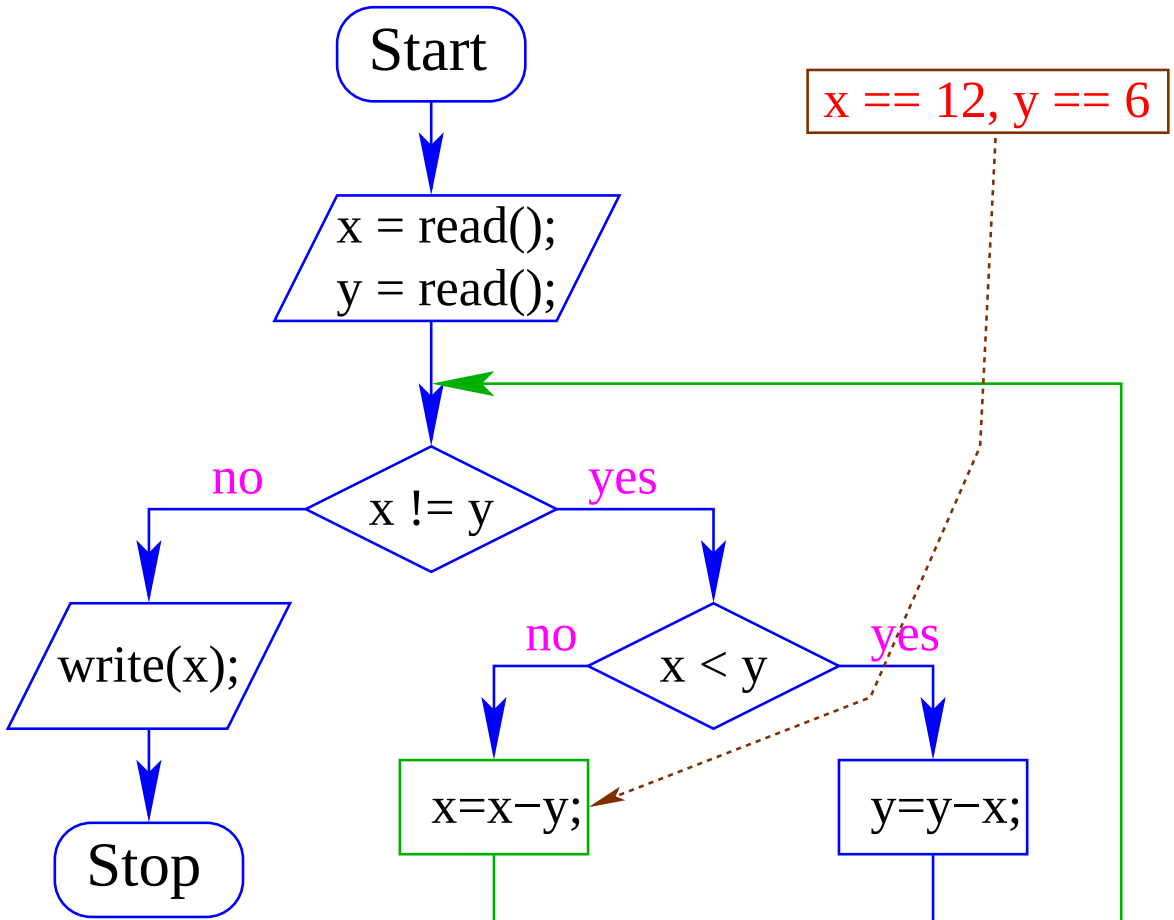

Start
x == 18, y == 24
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

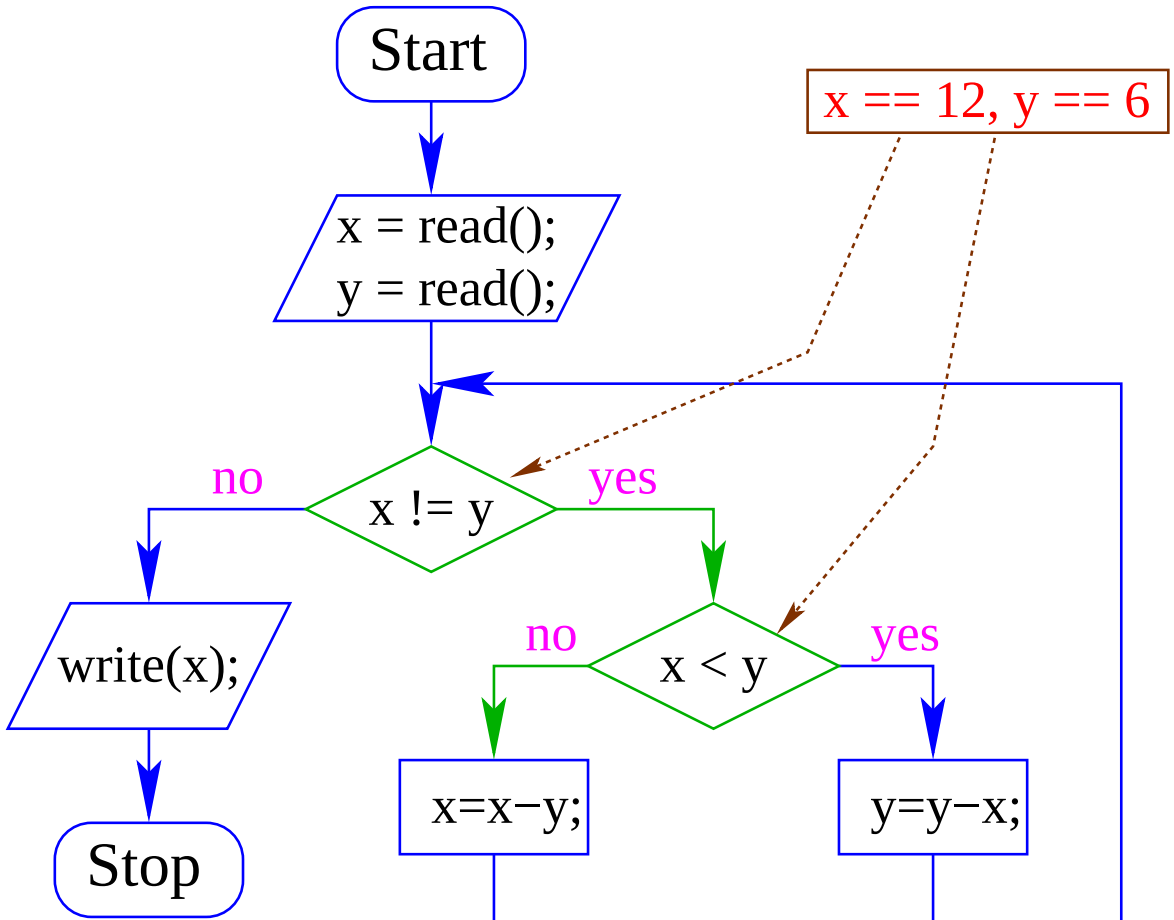

Start
x = read();
y = read();
x == 18, y == 24
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

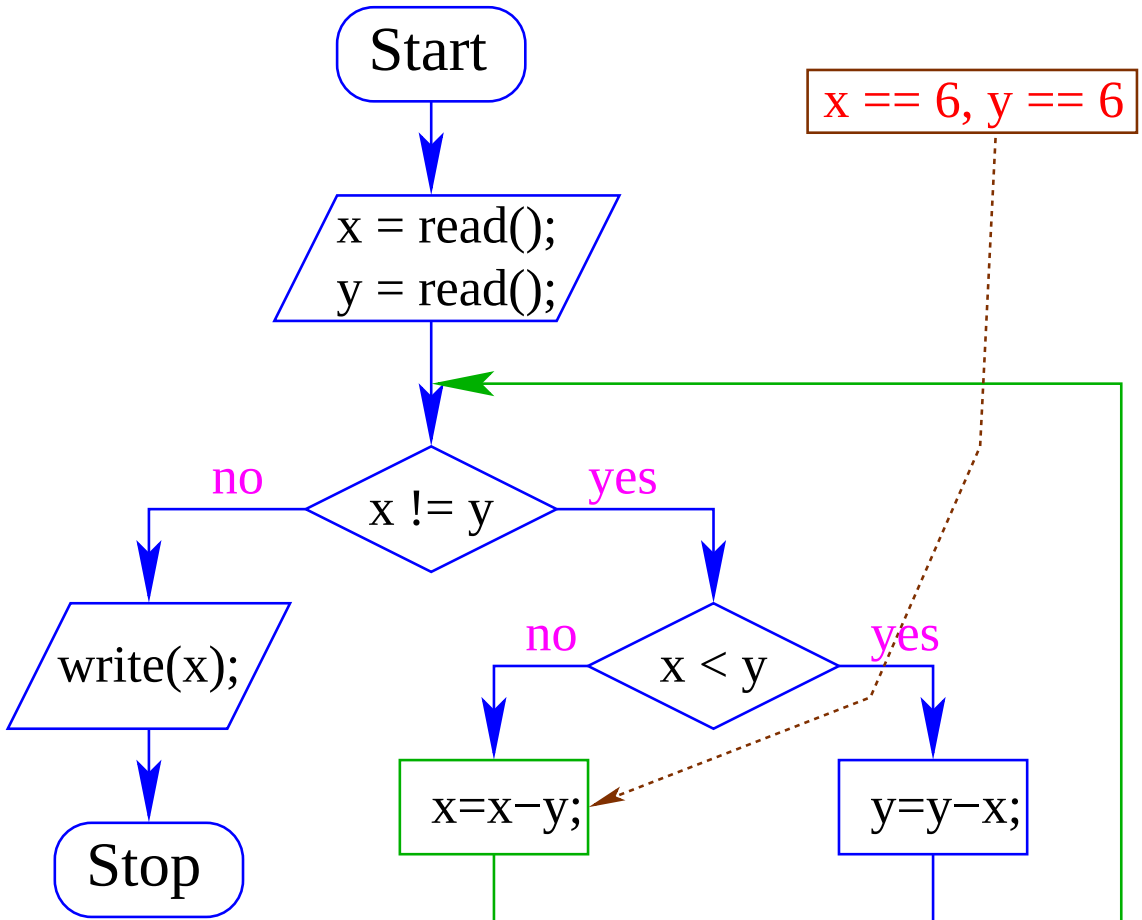

Start
x = read();
y = read();
x == 18, y == 24
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

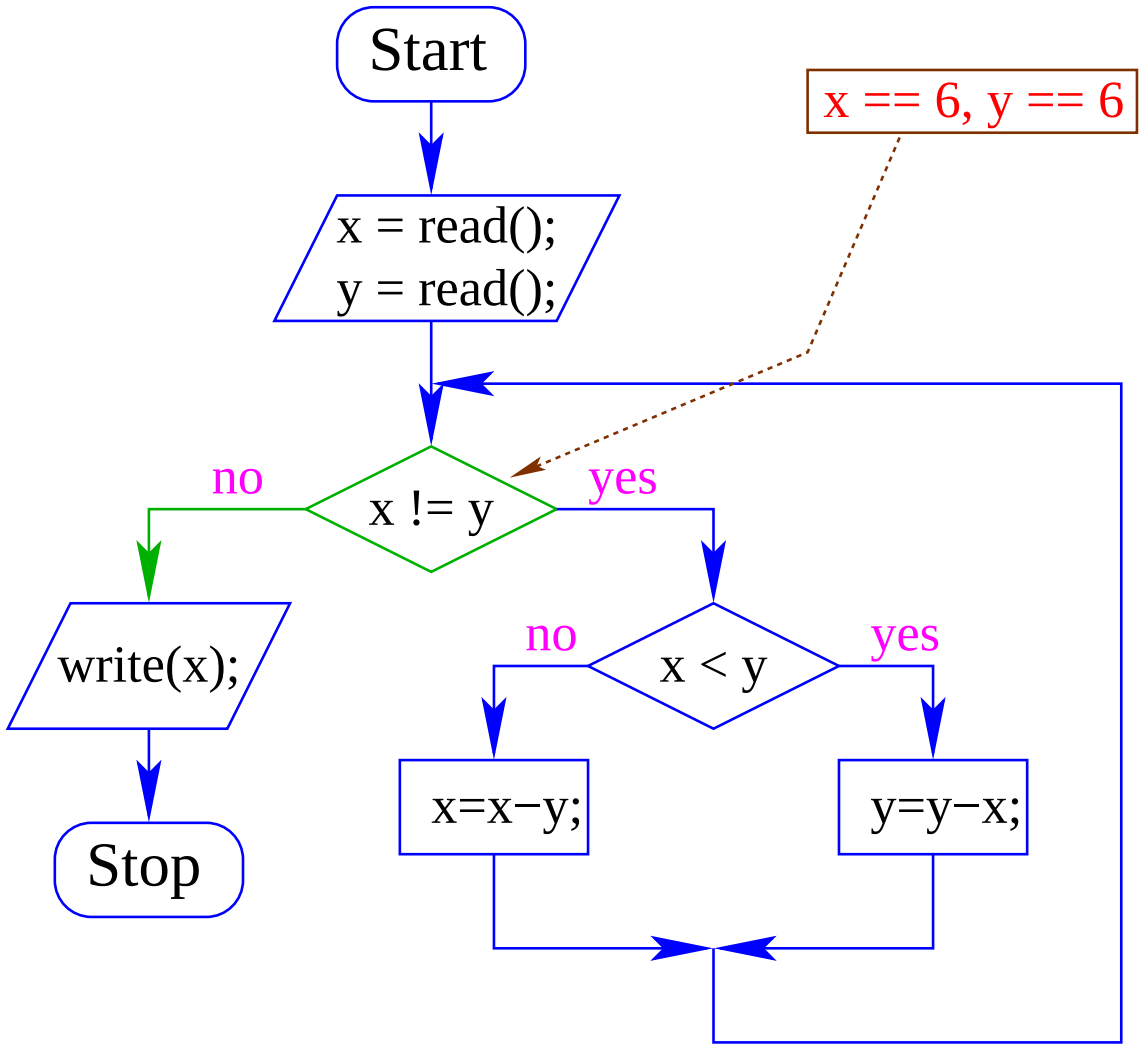

Start
x = read();
y = read();
x == 18, y == 6
no
x != y
yes
write(x);
Stop
no
x < y
yes
x=x−y;
y=y−x;

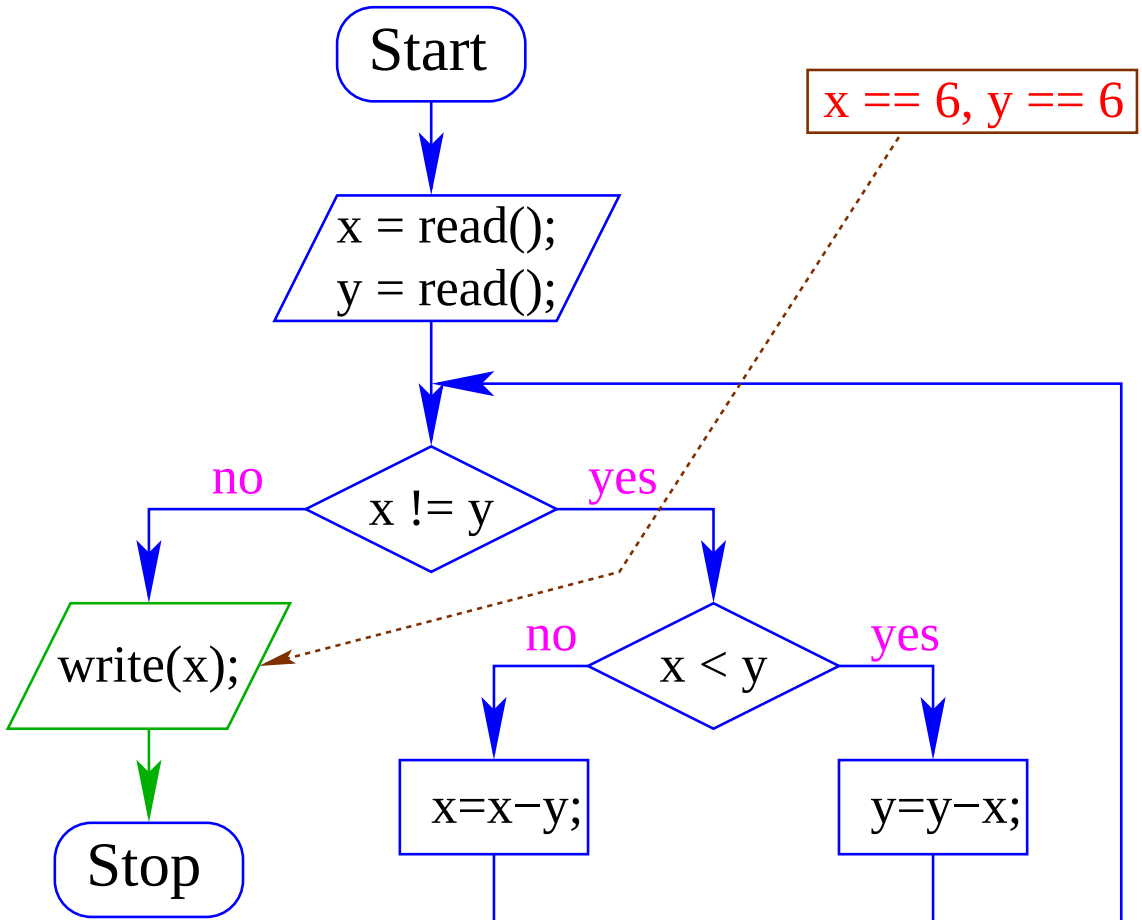

Start
x = read();
y = read();
x == 18, y == 6
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

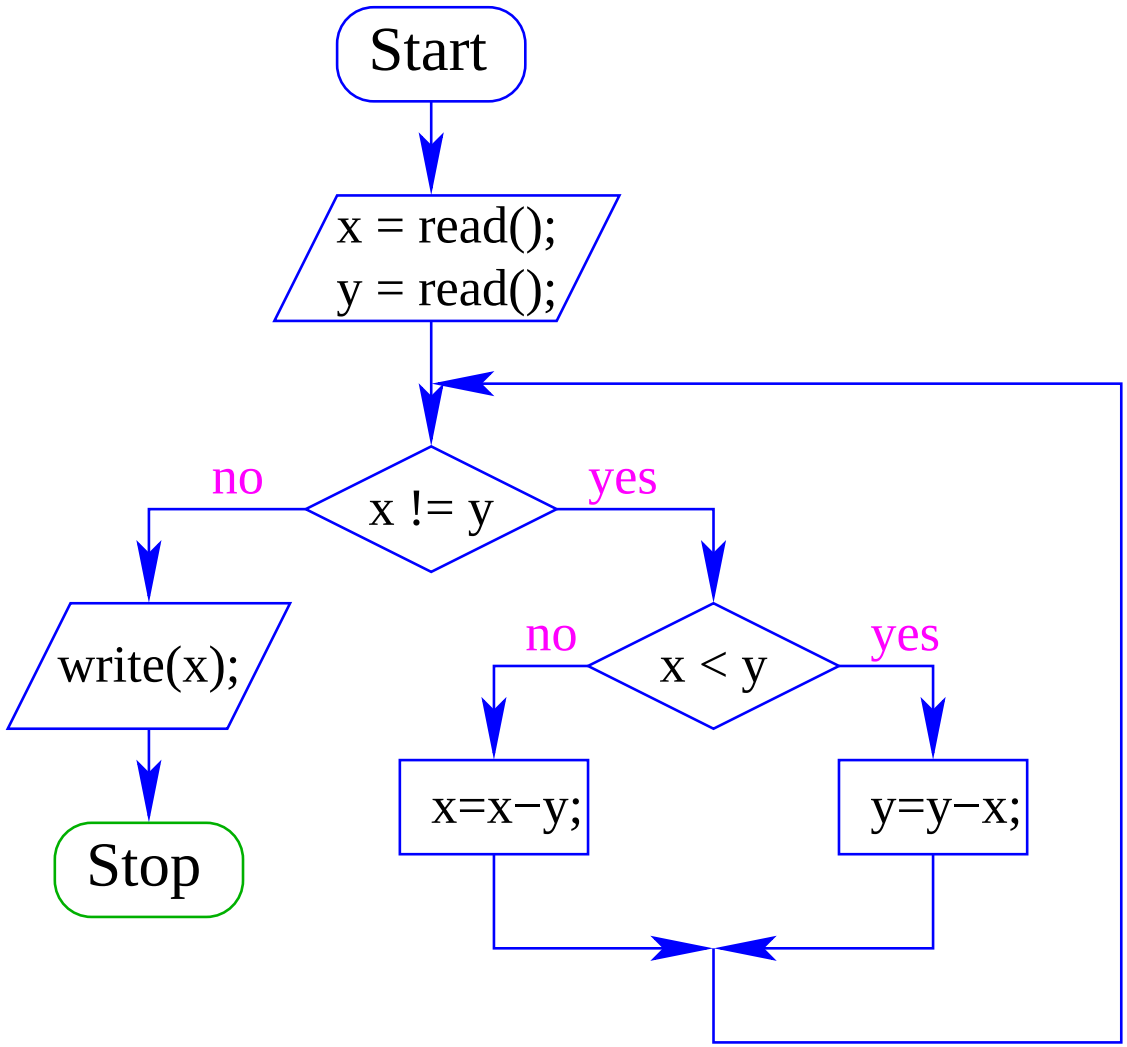

Start
x = read();
y = read();
x == 12, y == 6
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;


Start
x = read();
y = read();
x == 12, y == 6
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;


Start
x = read();
y = read();
x == 6, y == 6
no
x != y
yes
write(x);
Stop
no
x < y
yes
x=x−y;
y=y−x;


Start
x = read();
y = read();
x == 6, y == 6
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

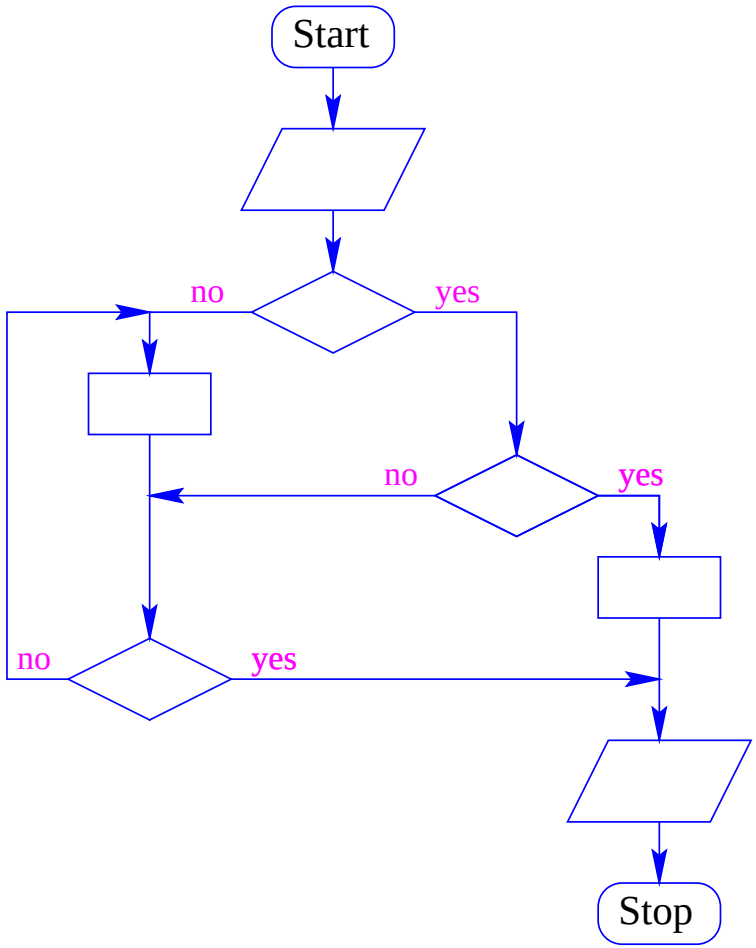

Start
x = read();
y = read();
x == 6, y == 6
no
x != y
yes
write(x);
Stop
no
x < y
yes
x=x−y;
y=y−x;


Start
x = read();
y = read();
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

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.

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: Java warnt nicht vor Überlauf/Unterlauf !!

Beispiel:

```
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	`true`, `false`
`char`	16	alle Unicode-Zeichen

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

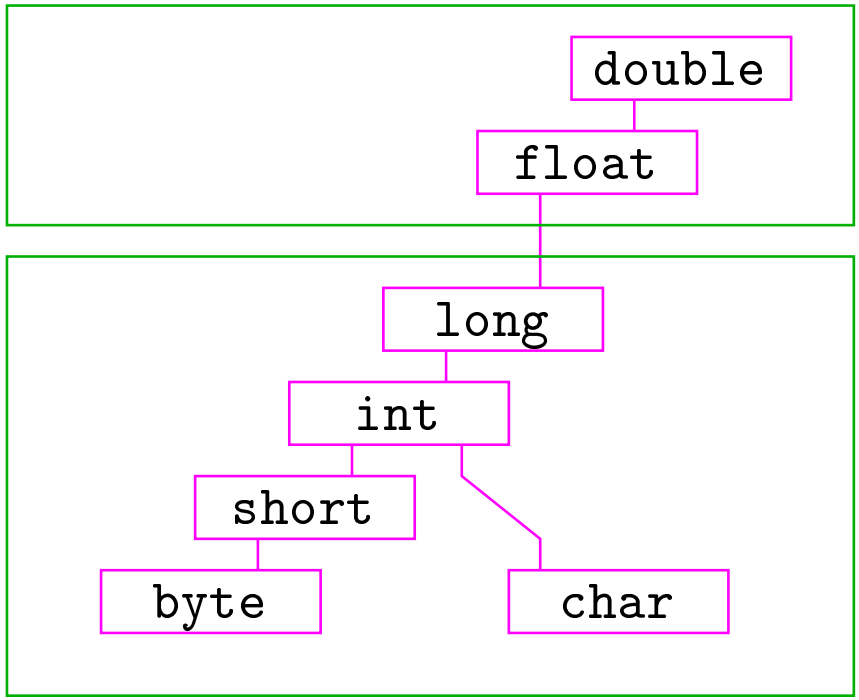
- die Zeichen unserer Tastatur (inklusive Umlaute);
- die chinesischen Schriftzeichen;
- die ägyptischen Hieroglyphen ...

`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) ...



Gleitkomma-Zahlen

ganze Zahlen

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` ...

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 Gleitkomma-Zahlen 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.

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 "`+`" konkatenieren.

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 ...

$\Longrightarrow$ ... 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!

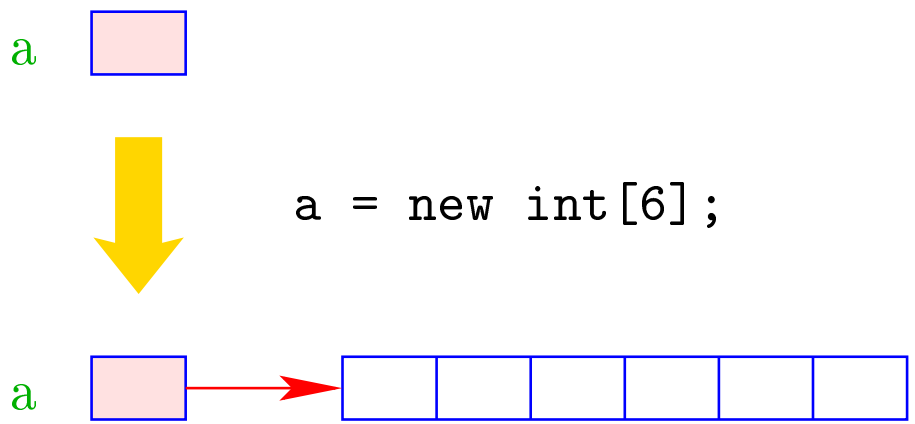
Feld:	17	3	-2	9	0	1
Index:	0	1	2	3	4	5

Beispiel: Einlesen eines Felds

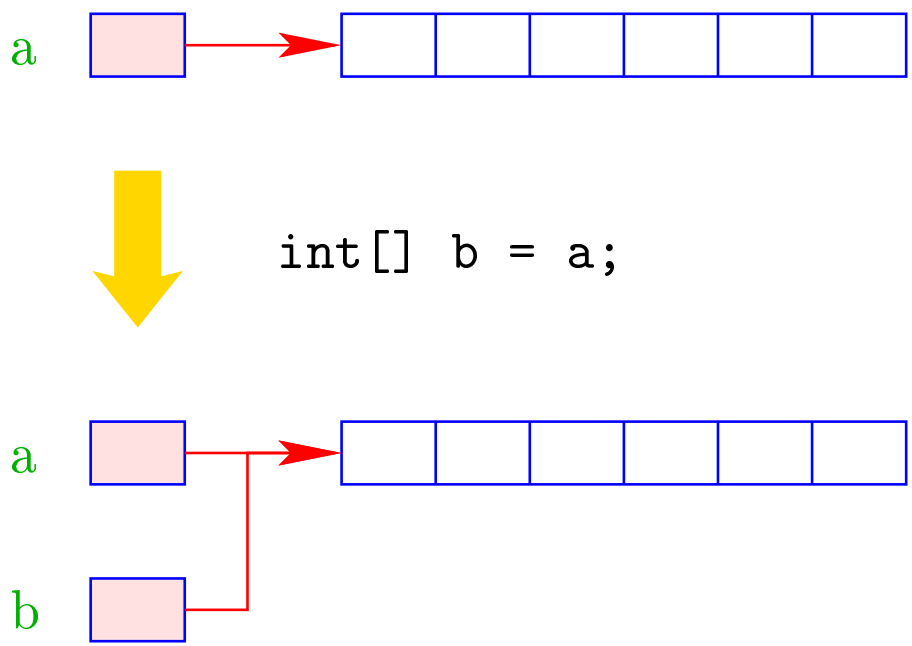
```
int[] a; // Deklaration
int n = read();

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:



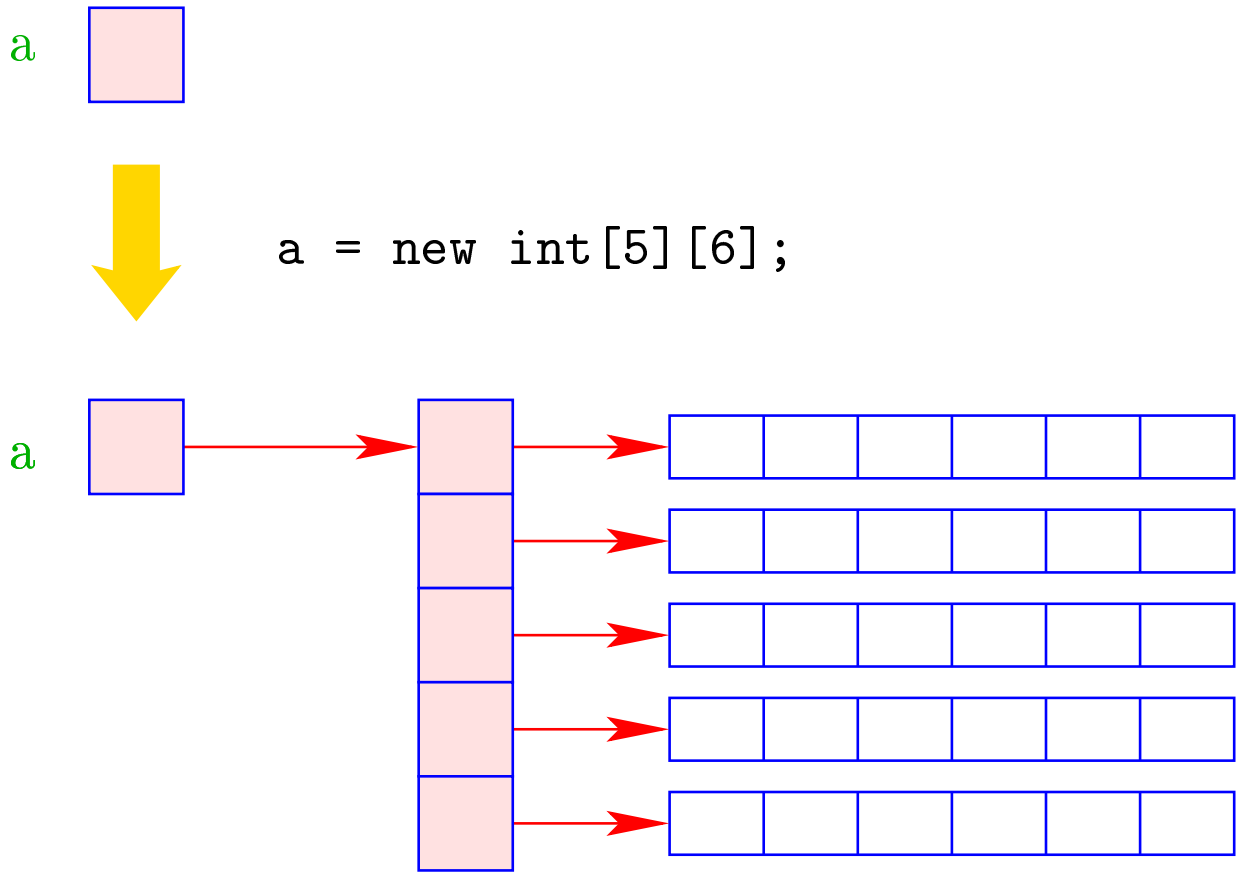
- Der Wert einer Feld-Variable ist also ein Verweis.
- `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



- Die Elemente eines Felds sind von 0 an durchnumeriert.
- 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 $\{0, \ldots,$ 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.

$\Longrightarrow$ ... 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;

$\Longrightarrow$ 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.

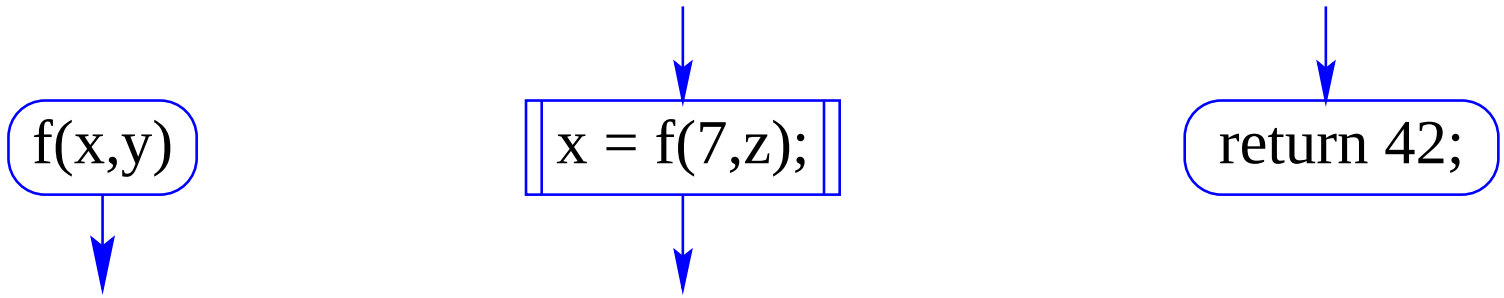
```
public class Test extends MiniJava {
   public static void main (String [] args) {
      write(args[0]+args[1]);
      }
}  // end of class Test
```

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:



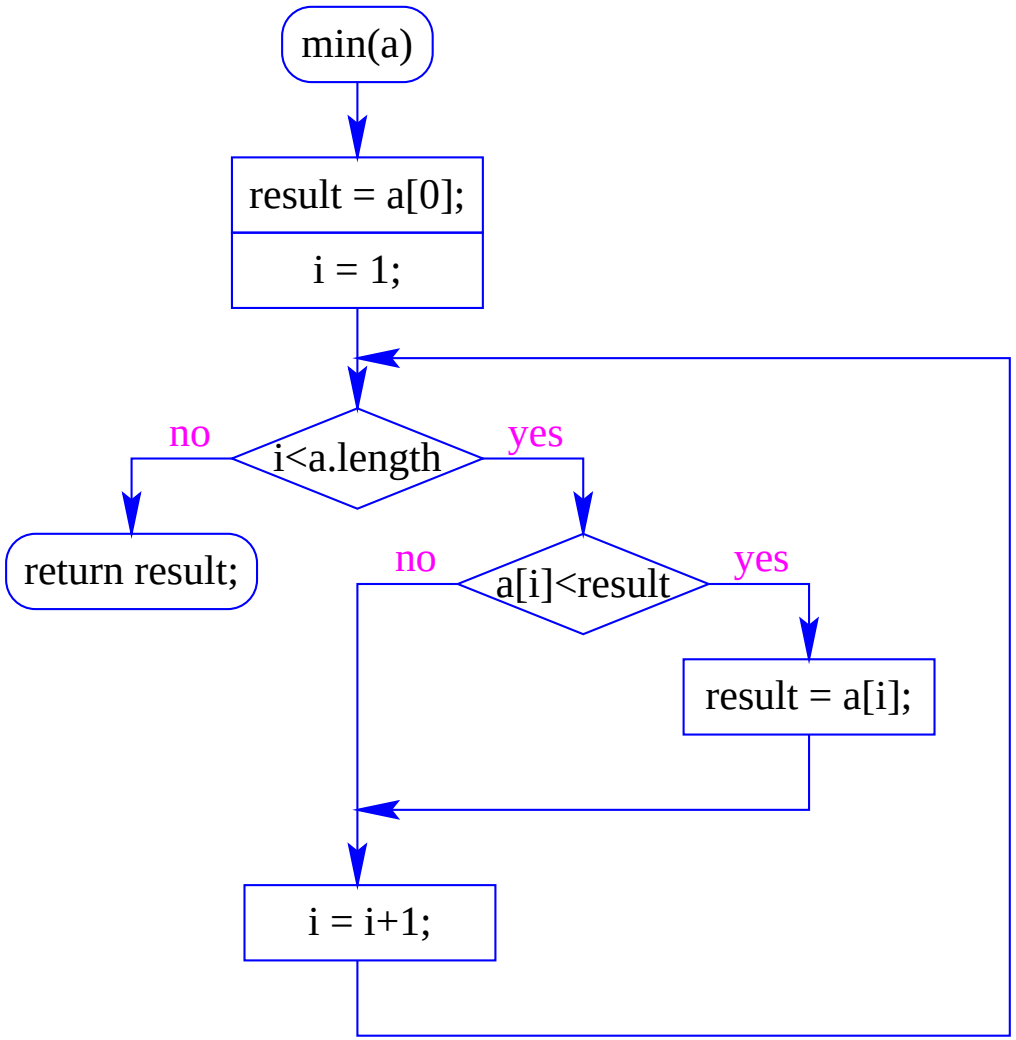
Startknoten der Funktion f

Funktionsaufruf

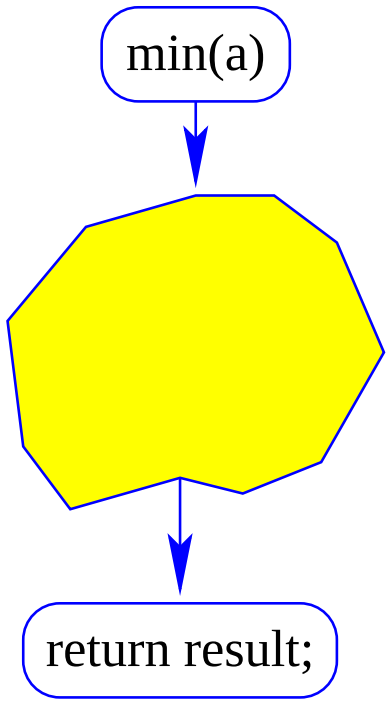
Endknoten

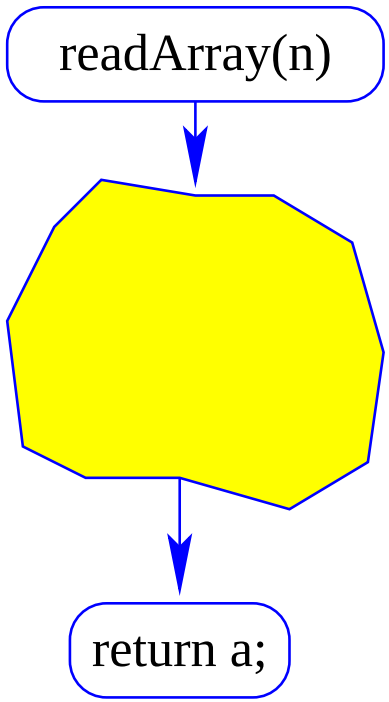
- Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- Ein Aufrufknoten repäsentiert eine Teilberechnung der aufgerufenen Funktion.

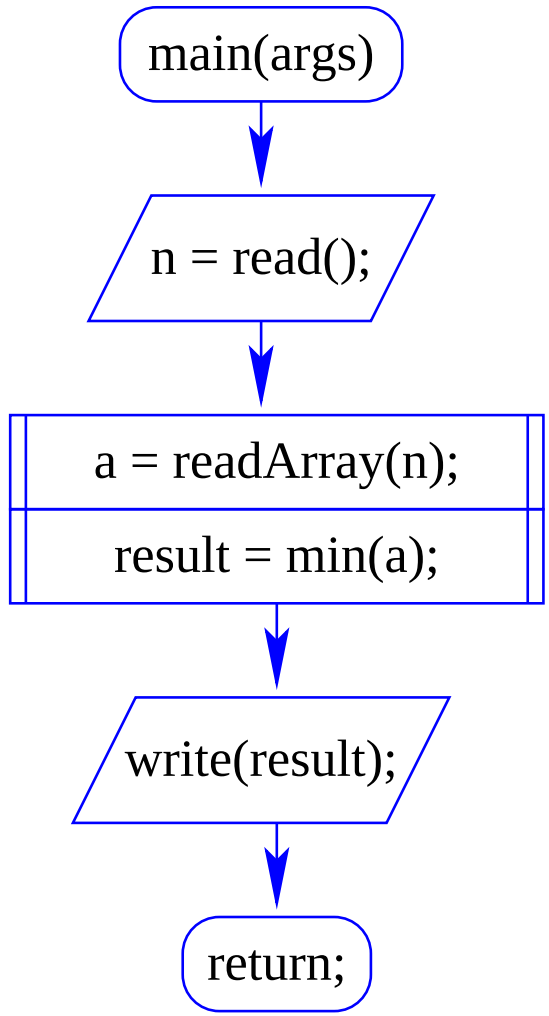
Teildiagramm für die Funktion `min()`:

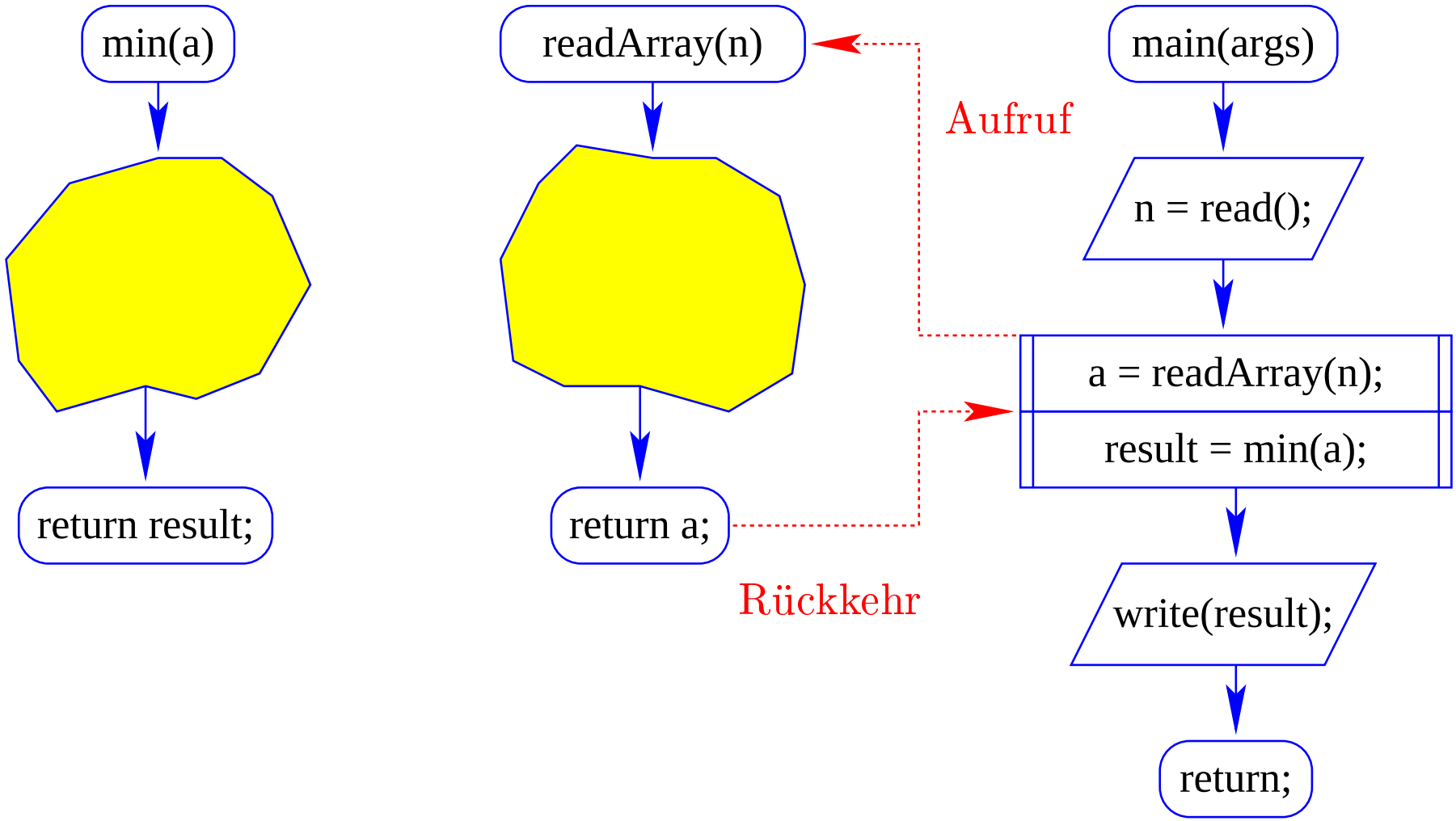


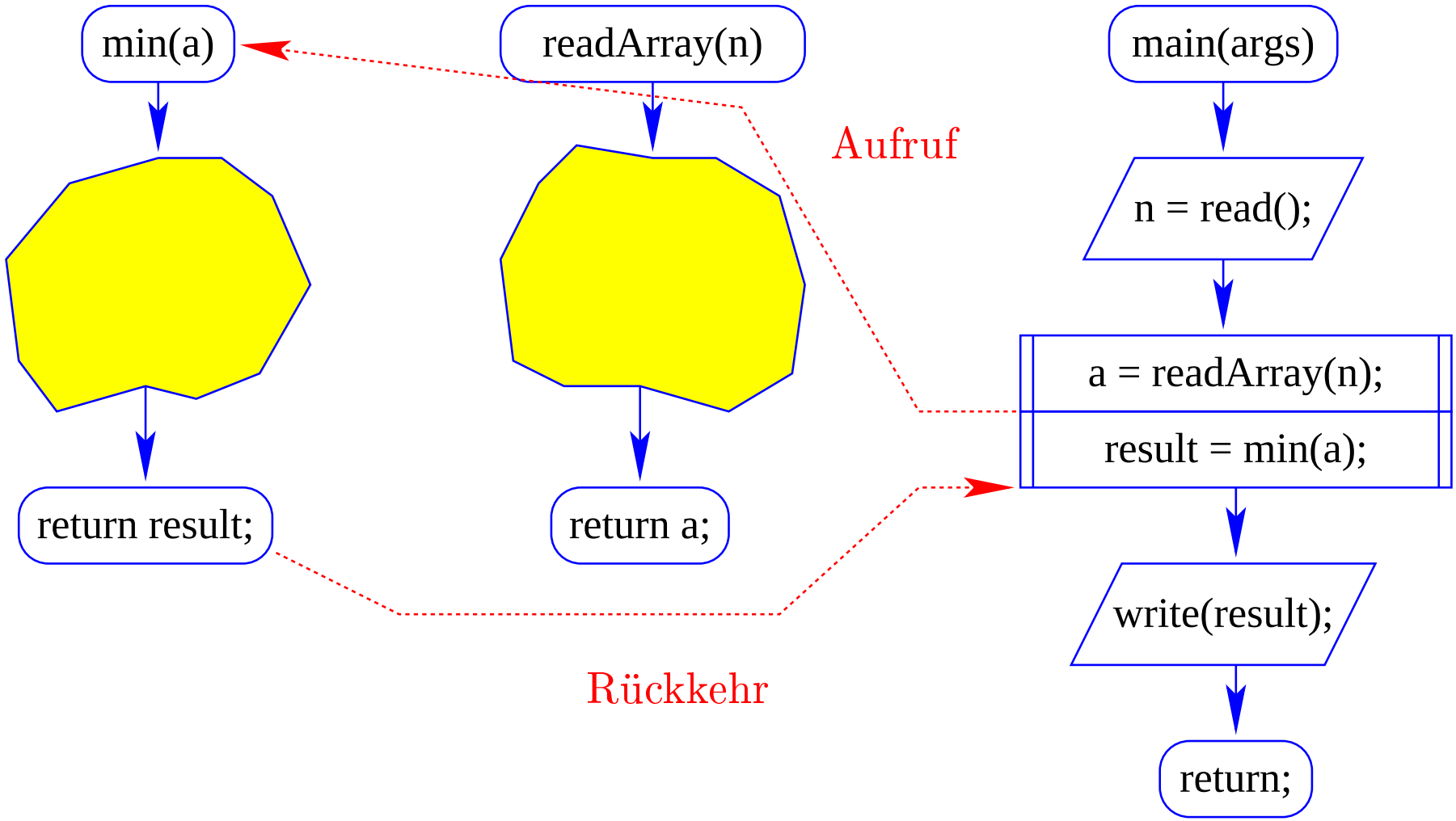
Insgesamt erhalten wir:








min(a)
return result;
readArray(n)
return a;
Aufruf
Rückkehr
main(args)
n = read();
a = readArray(n);
result = min(a);
write(result);
return;


min(a)
readArray(n)
main(args)
Aufruf
n = read();
a = readArray(n);
result = min(a);
return result;
return a;
write(result);
Rückkehr
return;

6 Eine erste Anwendung: Sortieren

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

6 Eine erste Anwendung: Sortieren

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

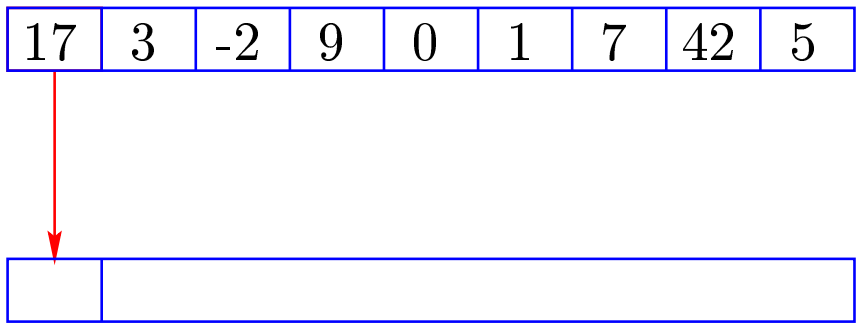
Idee:

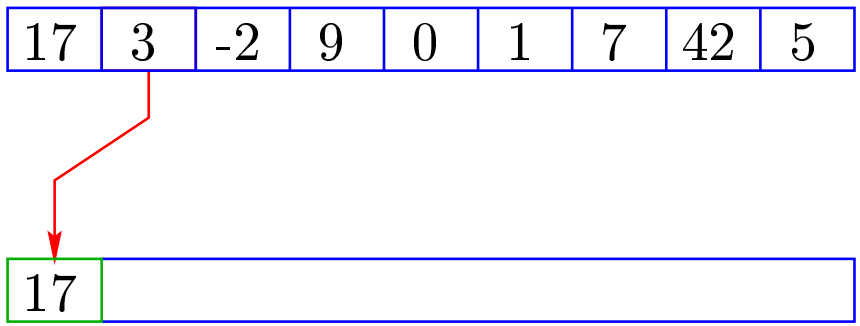
- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

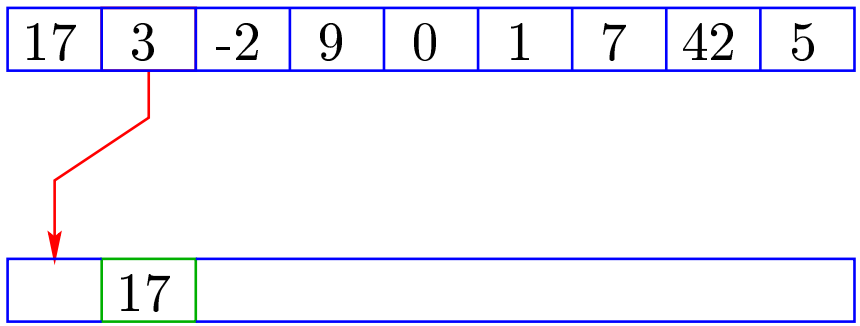
$\Longrightarrow$ Sortieren durch Einfügen ...

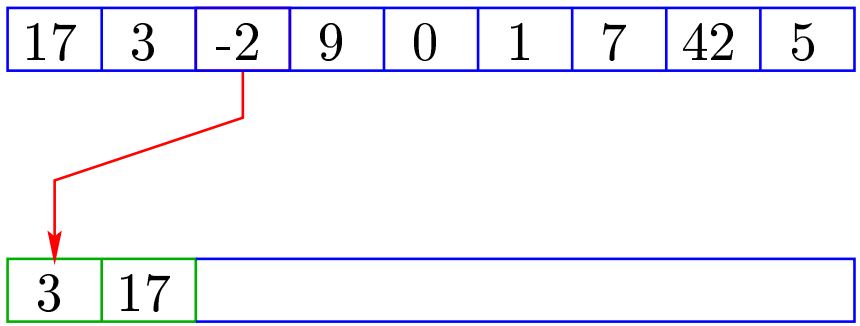
```
public static int[] sort (int[] a) {
   int n = a.length;
   int[] b = new int[n];
   for (int i = 0; i < n; ++i)
      insert (b, a[i], i);
         // b    = Feld, in das eingefügt wird
         // a[i] = einzufügendes Element
         // i    = Anzahl von Elementen in b
   return b;
   } // end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

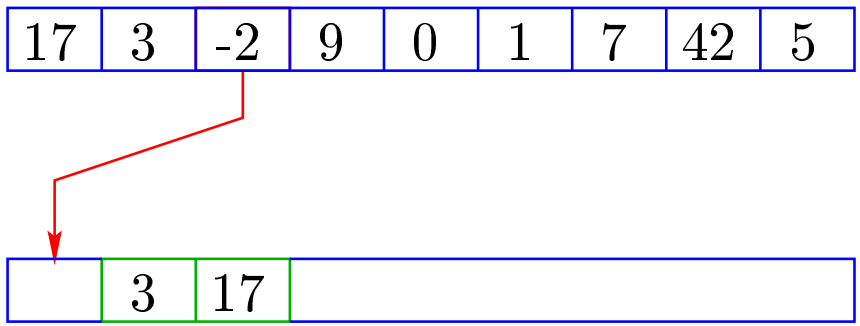
Teilproblem: Wie fügt man ein ???

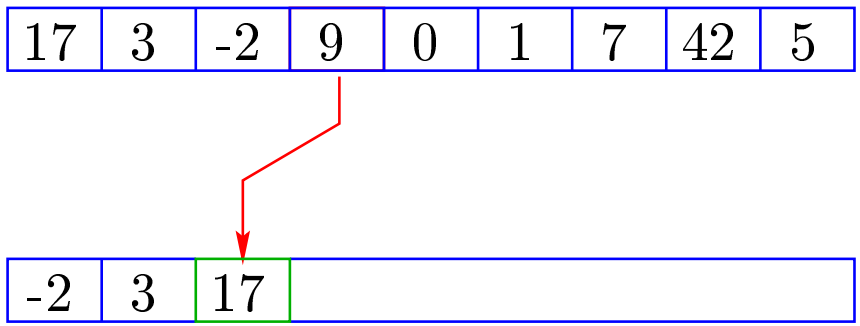

17
3
-2
9
0
1
7
42
5

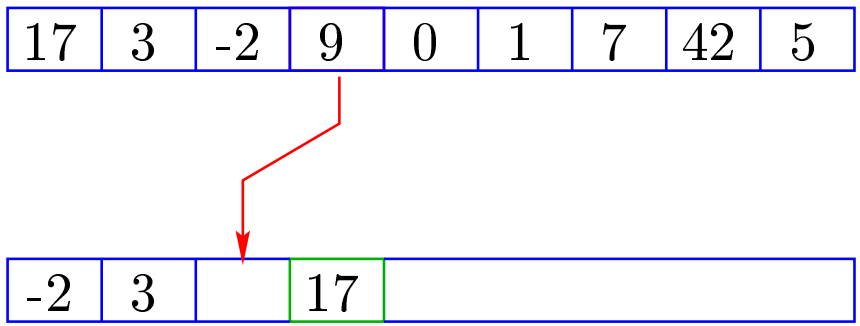

17
3
-2
9
0
1
7
42
5
17

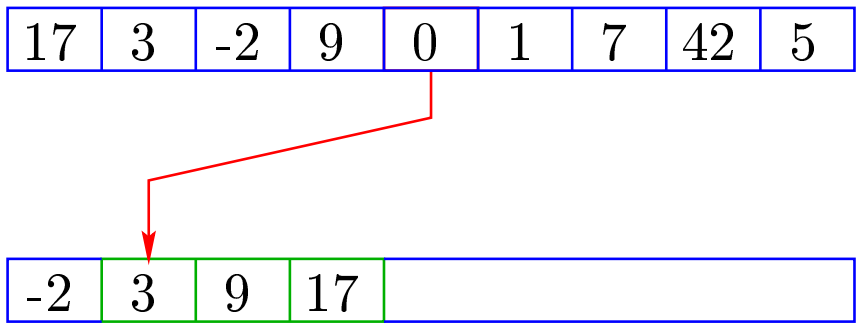

17
3
-2
9
0
1
7
42
5
17

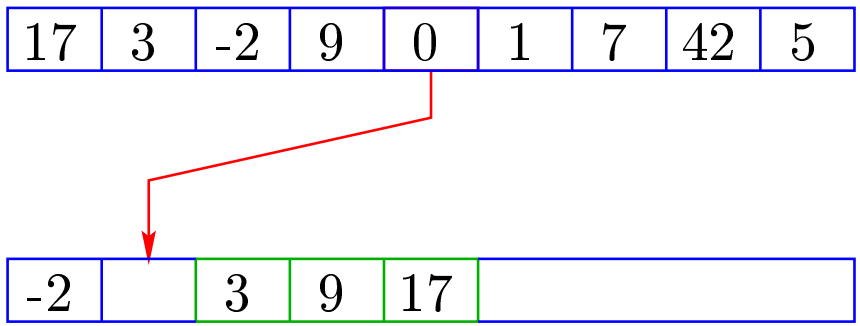

17
3
-2
9
0
1
7
42
5
3
17

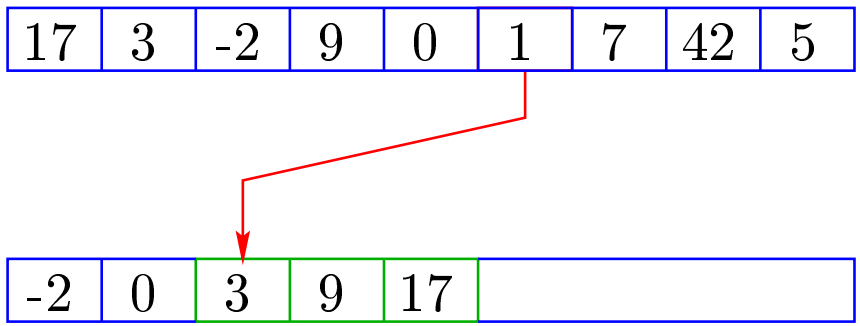

17
3
-2
9
0
1
7
42
5
3
17

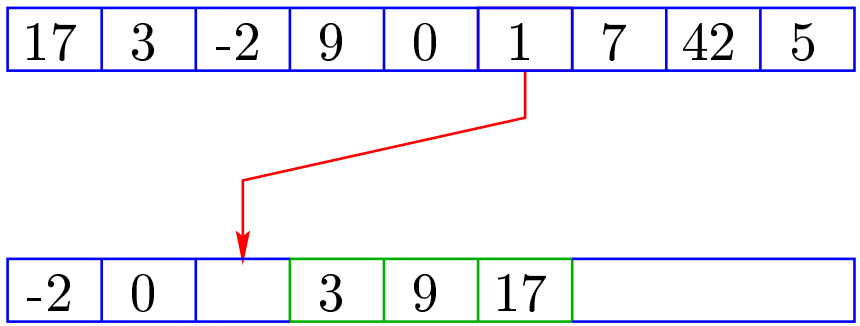

17
3
-2
9
0
1
7
42
5
-2
3
17

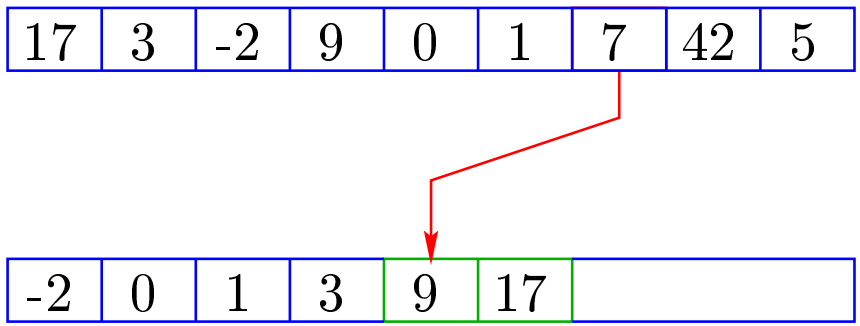

17
3
-2
9
0
1
7
42
5
-2
3
17

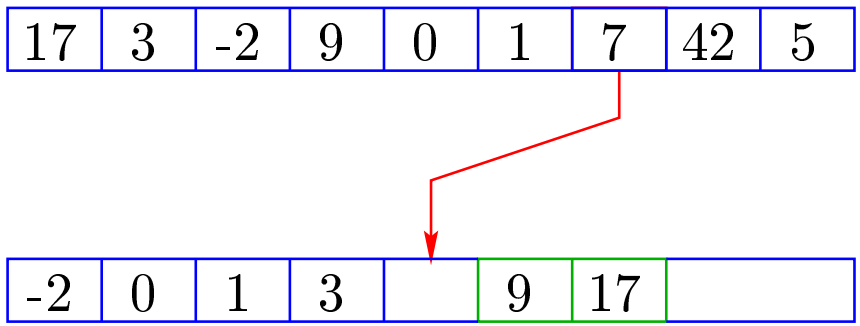

17
3
-2
9
0
1
7
42
5
-2
3
9
17

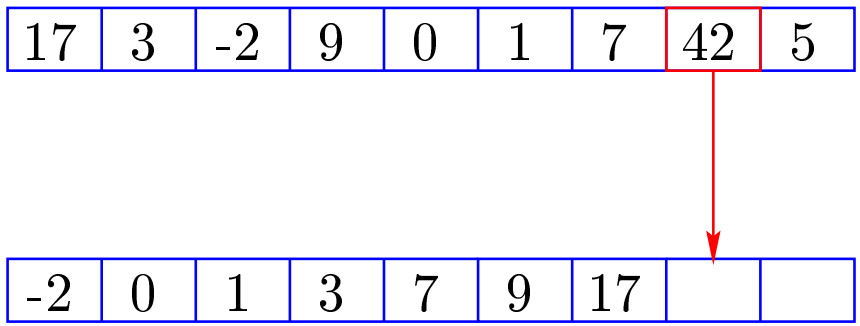

17
3
-2
9
0
1
7
42
5
-2
3
9
17

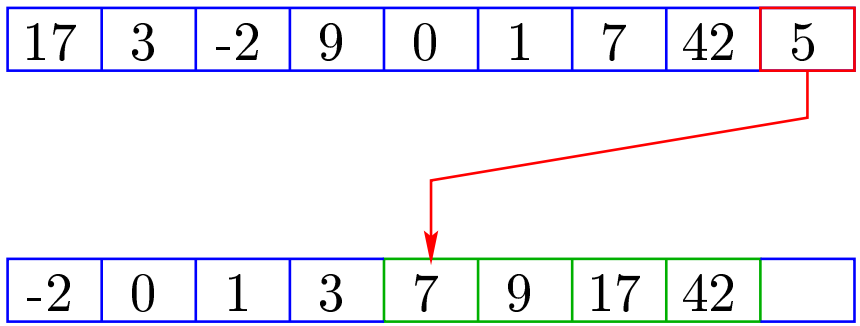

17
3
-2
9
0
1
7
42
5
-2
0
3
9
17

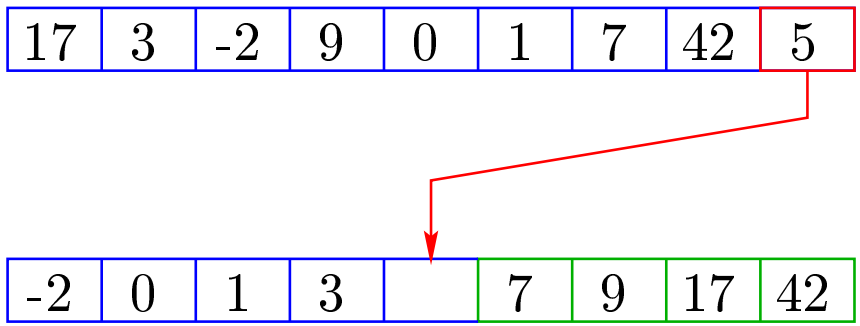

17
3
-2
9
0
1
7
42
5
-2
0
3
9
17

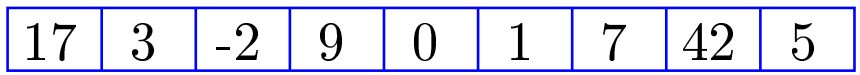

17
3
-2
9
0
1
7
42
5
-2
0
1
3
9
17

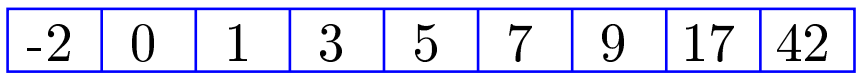

17
3
-2
9
0
1
7
42
5
-2
0
1
3
9
17


17 3 -2 9 0 1 7 42 5
-2 0 1 3 7 9 17


17 3 -2 9 0 1 7 42 5
-2 0 1 3 7 9 17 42


17
3
-2
9
0
1
7
42
5
-2
0
1
3
7
9
17
42


17 3 -2 9 0 1 7 42 5


-2 0 1 3 5 7 9 17 42

```
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` ??
- Gücklicherweise gibt es Sortier-Verfahren, die eine bessere Laufzeit haben (↑Algorithmen und Datenstrukturen).

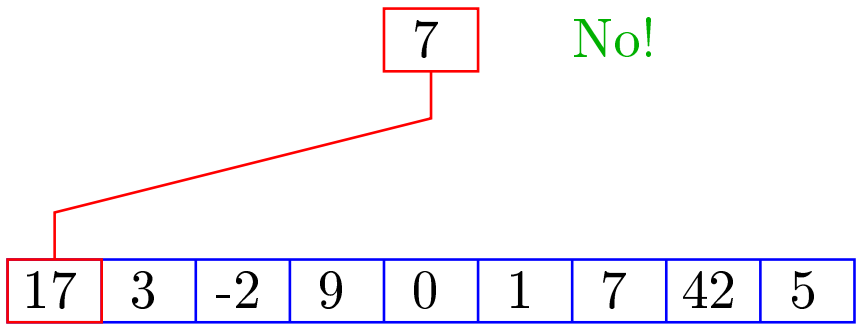
7 Eine zweite Anwendung: Suchen

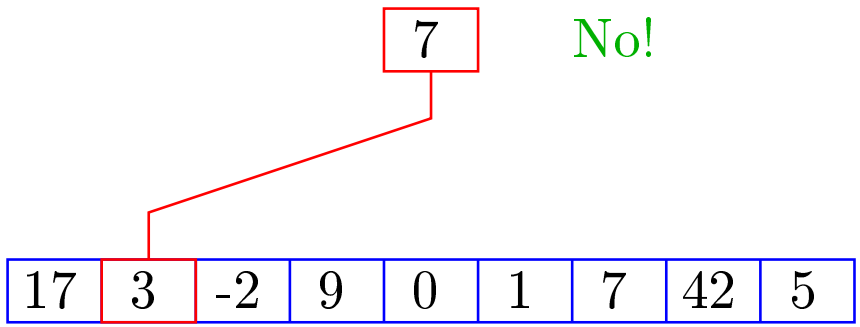
Nehmen wir an, wir wollen herausfinden, ob das Element 7 in unserem Feld `a` enthalten ist.

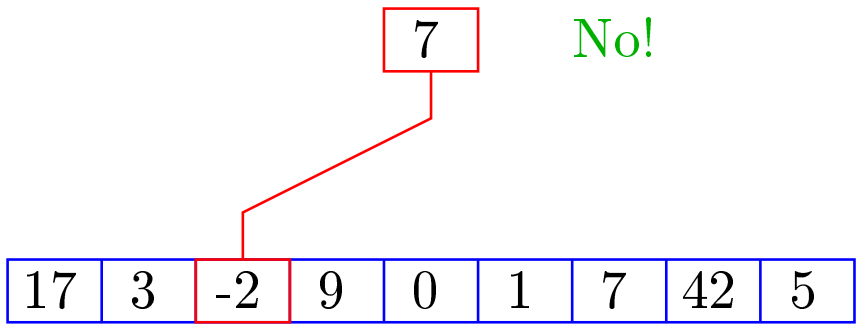
Naives Vorgehen:

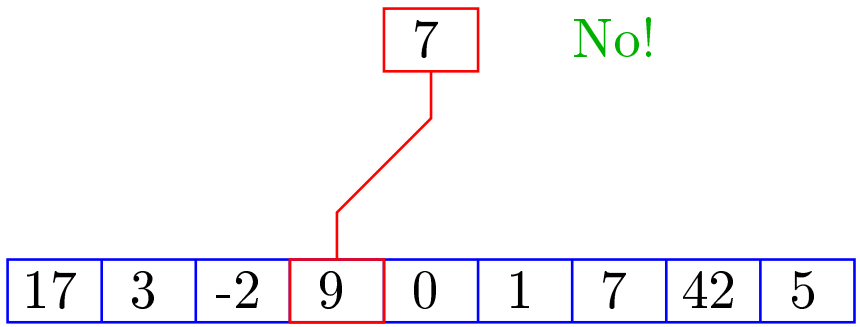
- Wir vergleichen 7 der Reihe nach mit den Elementen `a[0]`, `a[1]`, usw.
- Finden wir ein `i` mit `a[i] == 7`, geben wir `i` aus.
- Andernfalls geben wir `-1` aus: "Sorry, gibt's leider nicht !"

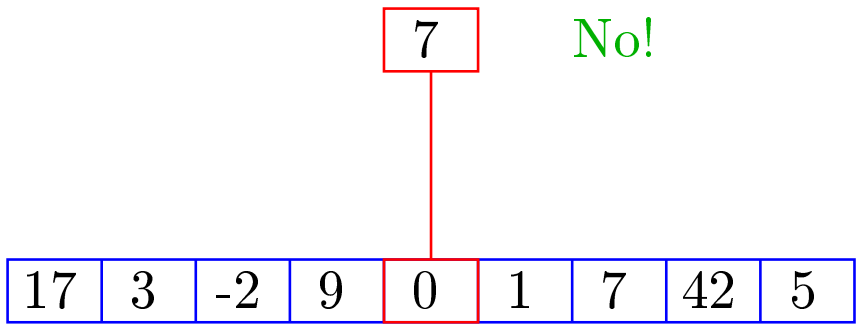
```
public static int find (int[] a, int x) {
   int i = 0;
   while (i < a.length && a[i] != x)
      ++i;
   if (i == a.length)
      return -1;
   else
      return i;
   }
```

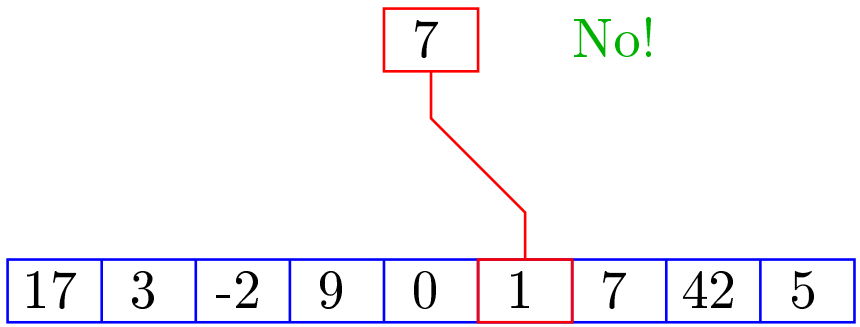

7
No!
17
3
-2
9
0
1
7
42
5

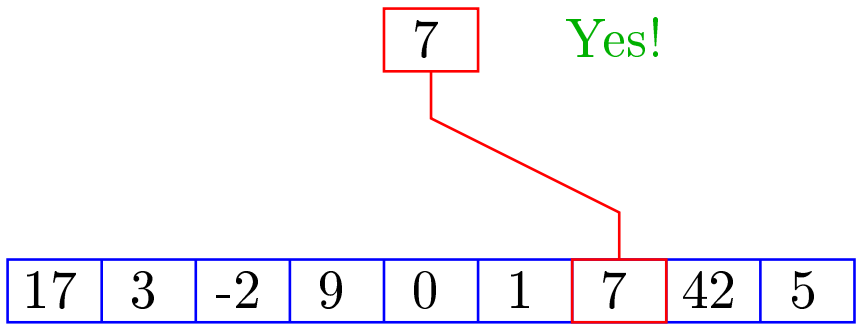

7
No!
17
3
-2
9
0
1
7
42
5


7
No!
17
3
-2
9
0
1
7
42
5


7
No!
17
3
-2
9
0
1
7
42
5


7
No!
17
3
-2
9
0
1
7
42
5


7
No!
17
3
-2
9
0
1
7
42
5


7
Yes!
17
3
-2
9
0
1
7
42
5

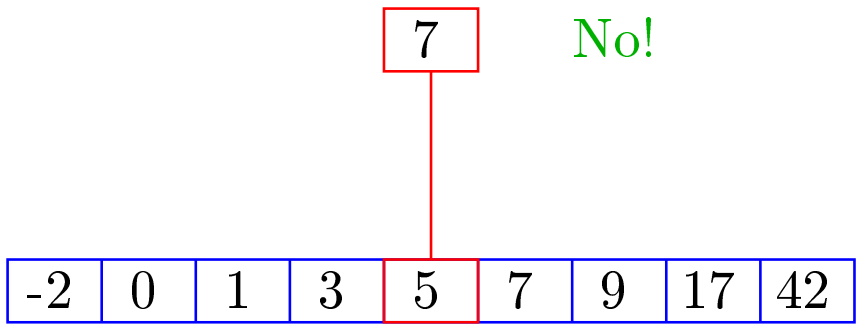
- Im Beispiel benötigen wir 7 Vergleiche.
- Im schlimmsten Fall benötigen wir bei einem Feld der Länge n sogar n Vergleiche ??
- Kommt 7 tatsächlich im Feld vor, benötigen wir selbst im Durchschnitt $(n+1)/2$ viele Vergleiche ??

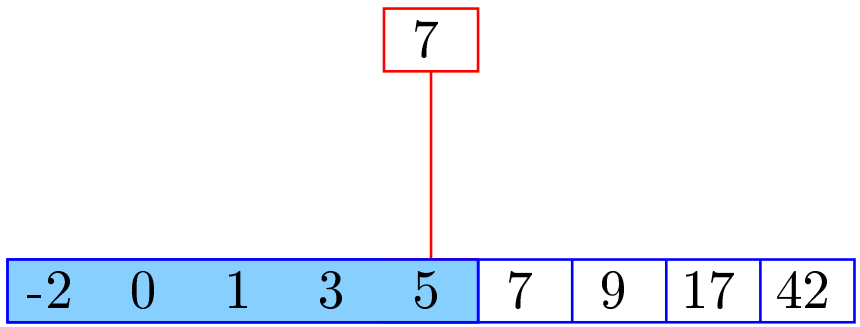
Geht das nicht besser ???

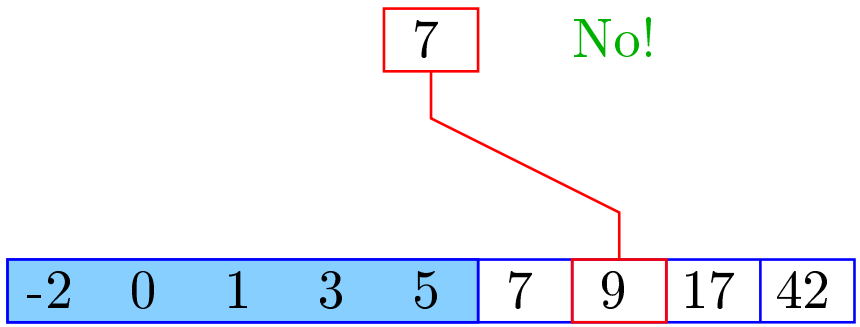
Idee:

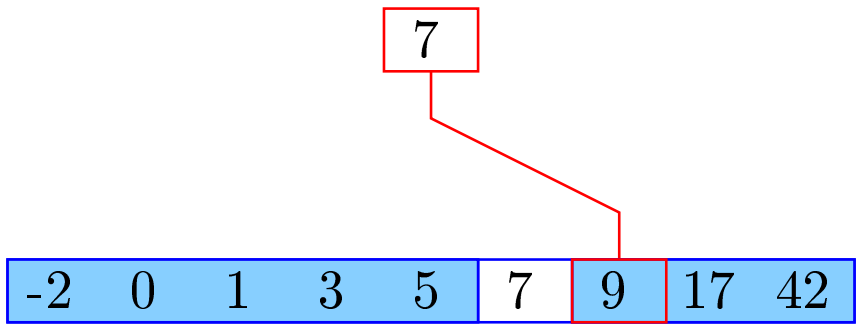
- Sortiere das Feld.
- Vergleiche 7 mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist 7 kleiner, brauchen wir nur noch links weitersuchen.
- Ist 7 größer, brauchen wir nur noch rechts weiter suchen.

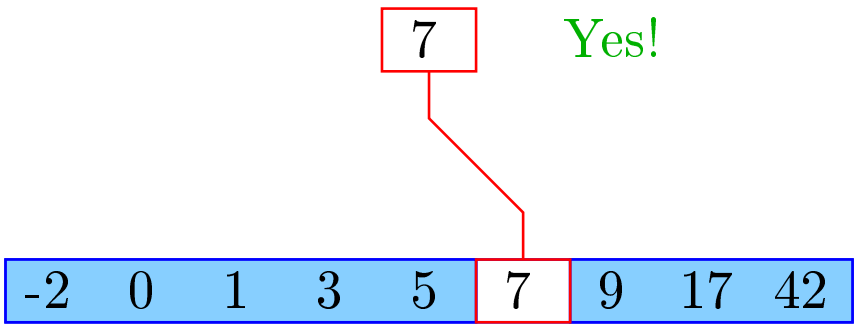
$\Longrightarrow$ binäre Suche ...


7
No!
-2
0
1
3
5
7
9
17
42


7
-2
0
1
3
5
7
9
17
42


7
No!
-2
0
1
3
5
7
9
17
42


7
-2
0
1
3
5
7
9
17
42


7
Yes!
-2
0
1
3
5
7
9
17
42

- 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

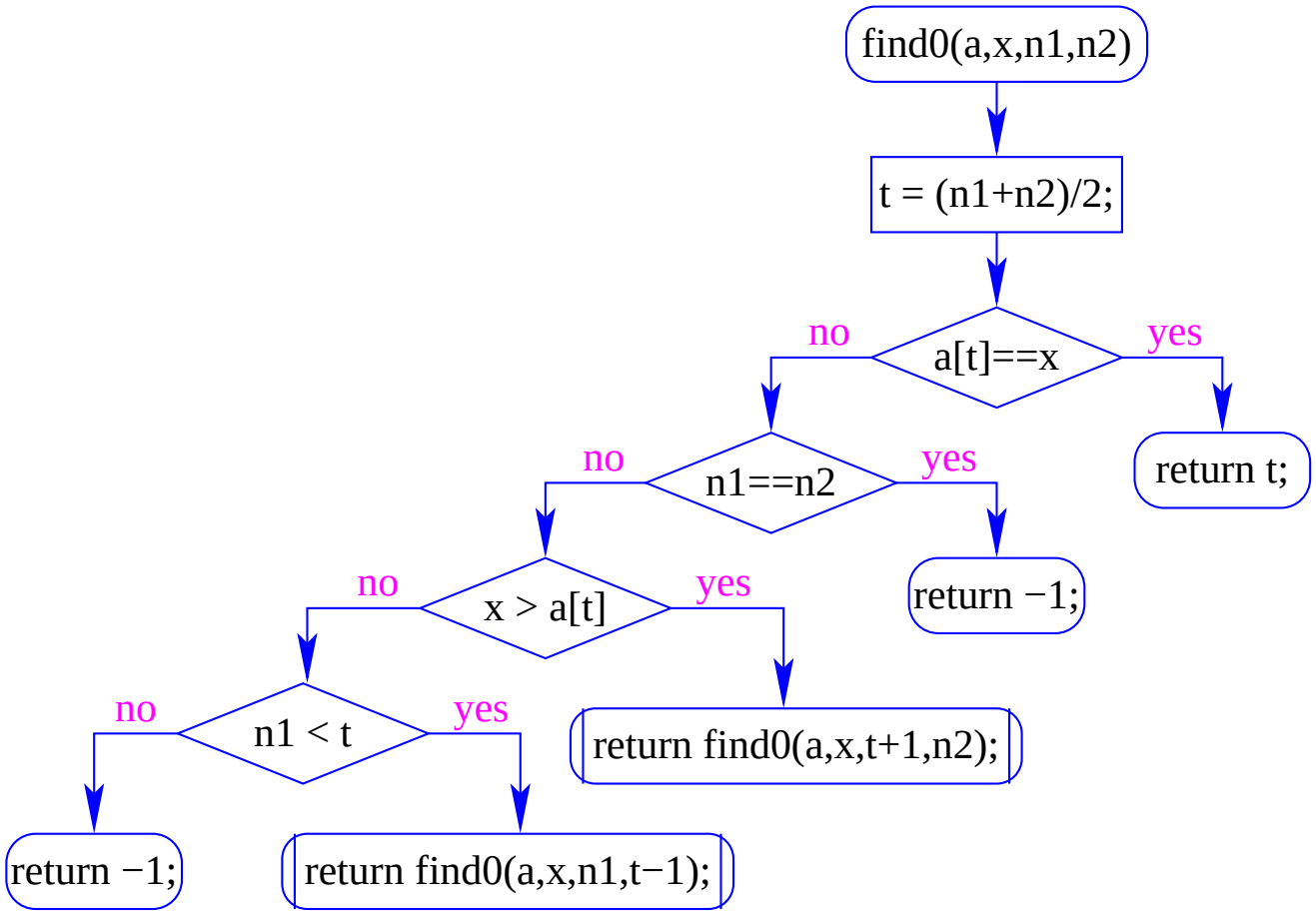
```
public static int find0 (int[] a, int x, int n1, int n2)
```

ein, die im Intervall $[\texttt{n1}, \texttt{n2}]$ 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;
   }
```

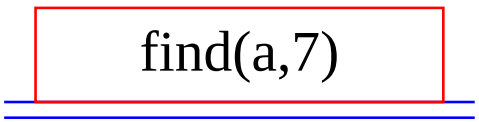
Das Kontrollfluss-Diagramm für `find0()`:



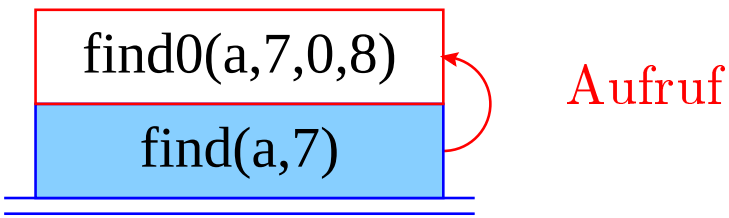
Achtung:

- 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.

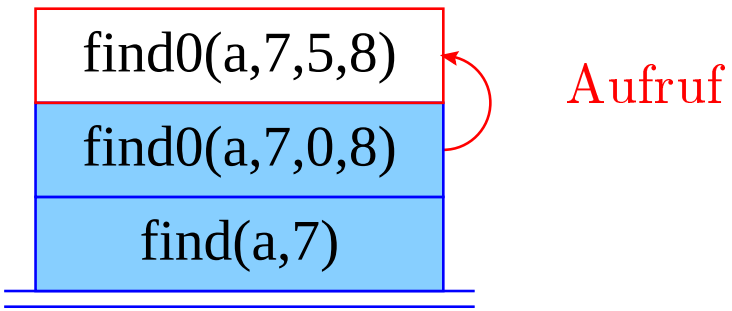
Ausführung:



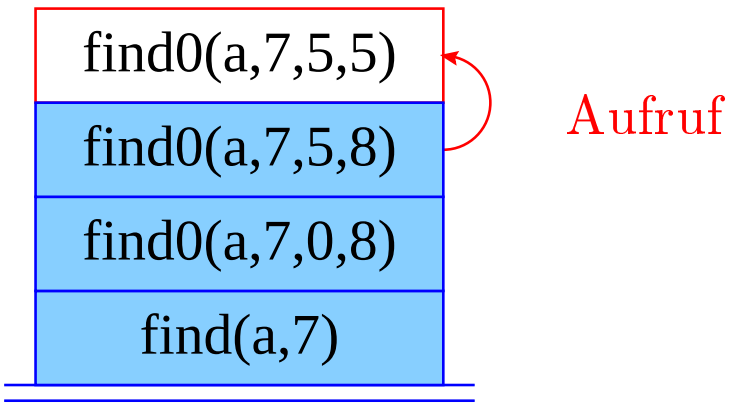
Ausführung:



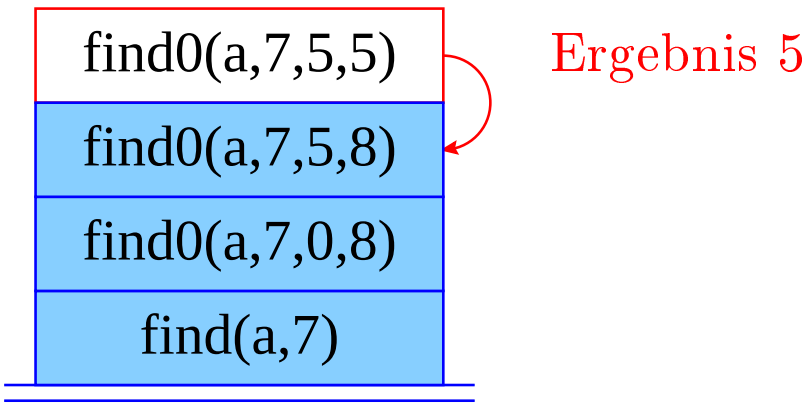
Ausführung:



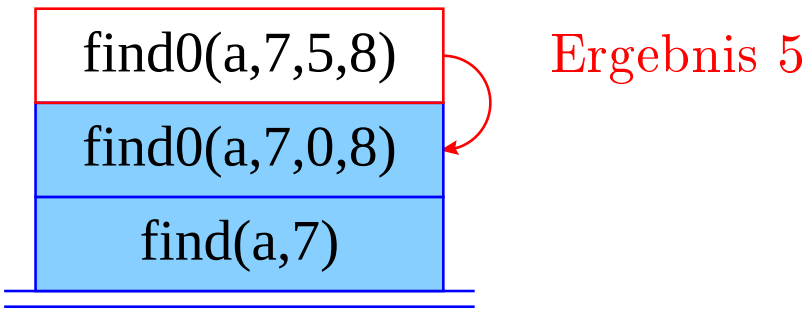
Ausführung:



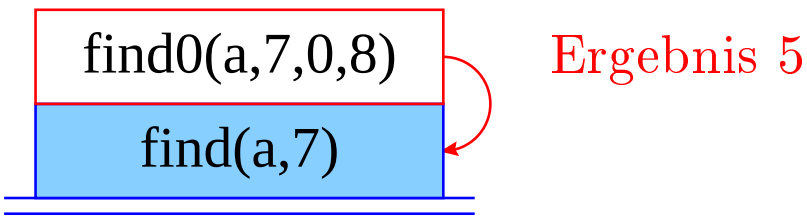
Ausführung:



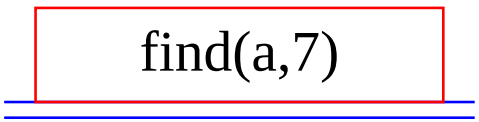
Ausführung:



Ausführung:



Ausführung:



Ergebnis 5

- Die Verwaltung der Funktionsaufrufe erfolgt nach dem LIFO-Prinzip (Last-In-First-Out).
- 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.

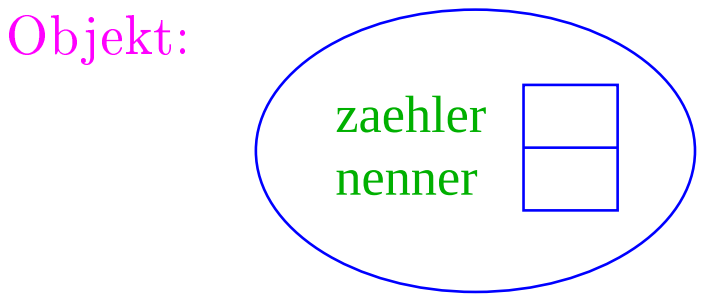
$\Longrightarrow$ ähnliche Technik wird auch für andere rekursive Funktionen angewandt.

8 Klassen und Objekte

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

Beispiel: Rationale Zahlen

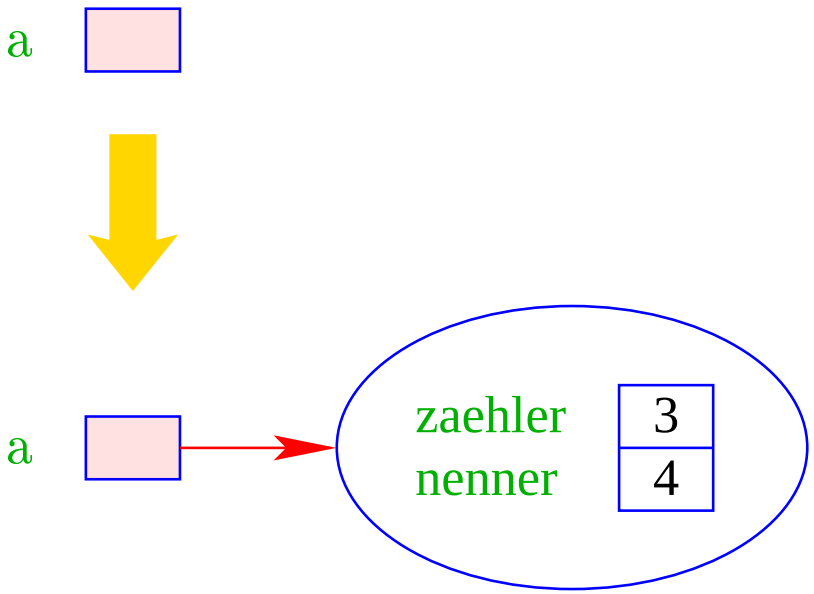
- Eine rationale Zahl $q \in \mathbb{Q}$ hat die Form $q = \frac{x}{y}$, wobei $x, y \in \mathbb{Z}$.
- x und y heißen Zähler und Nenner von q.
- Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` enthalten:



- Die Daten-Komponenten eines Objekts heißen Objekt-Attriute 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:

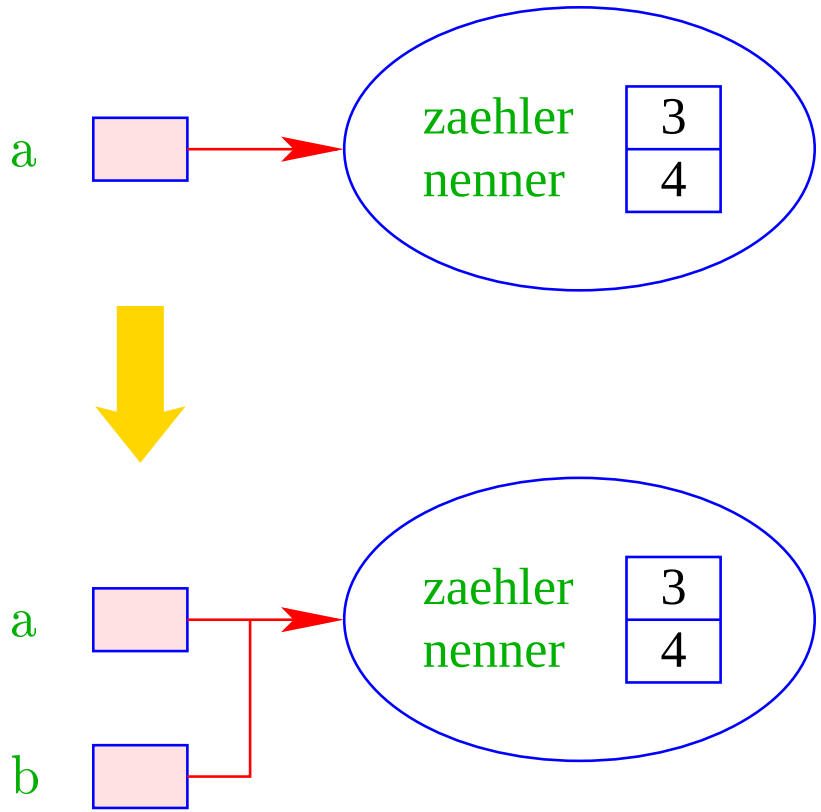
```
a = new Rational (3,4);
```



- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objekts initialisieren kann.

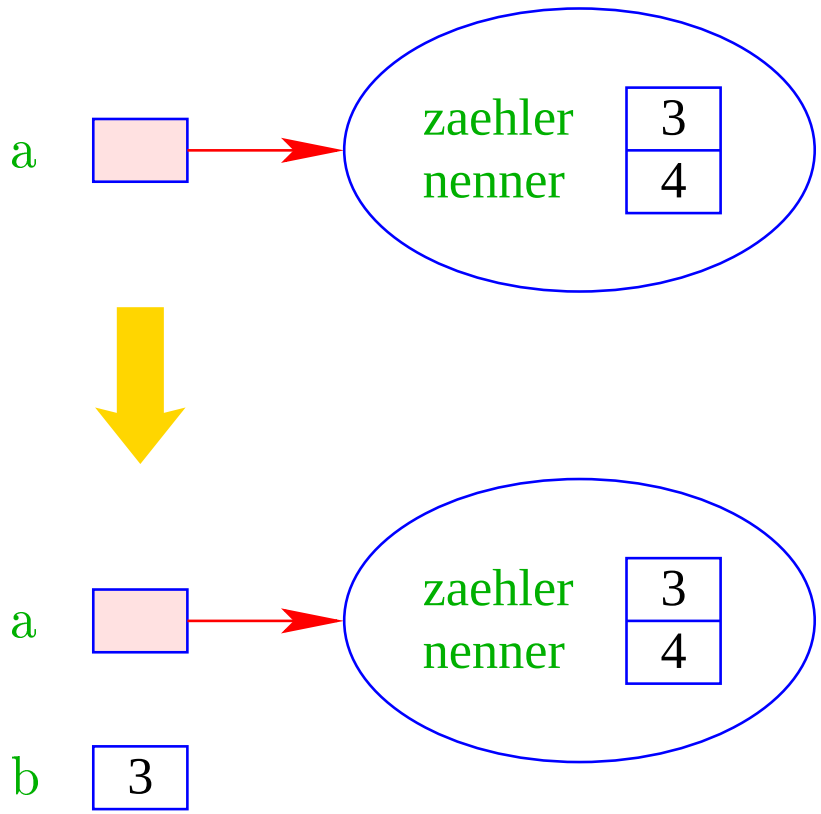
- Der Wert einer `Rational`-Variable ist ein Verweis auf einen Speicherbereich.
- `Rational b = a;` kopiert den Verweis aus `a` in die Variable `b`:

```
Rational b = a;
```

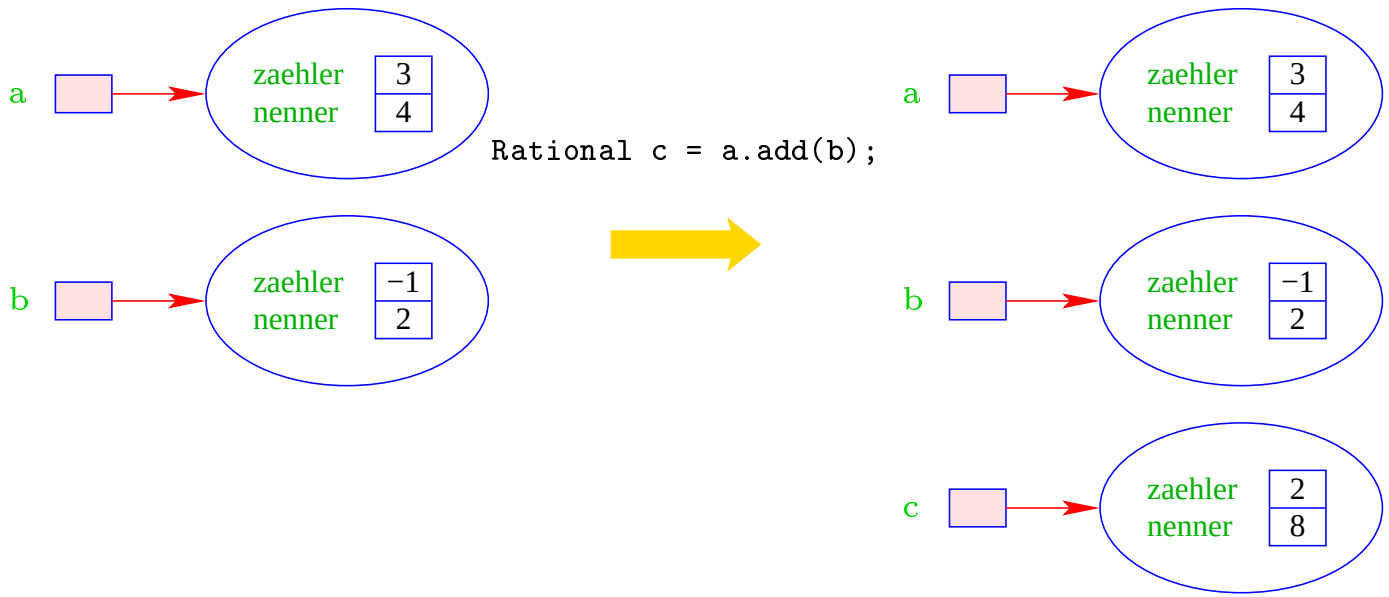


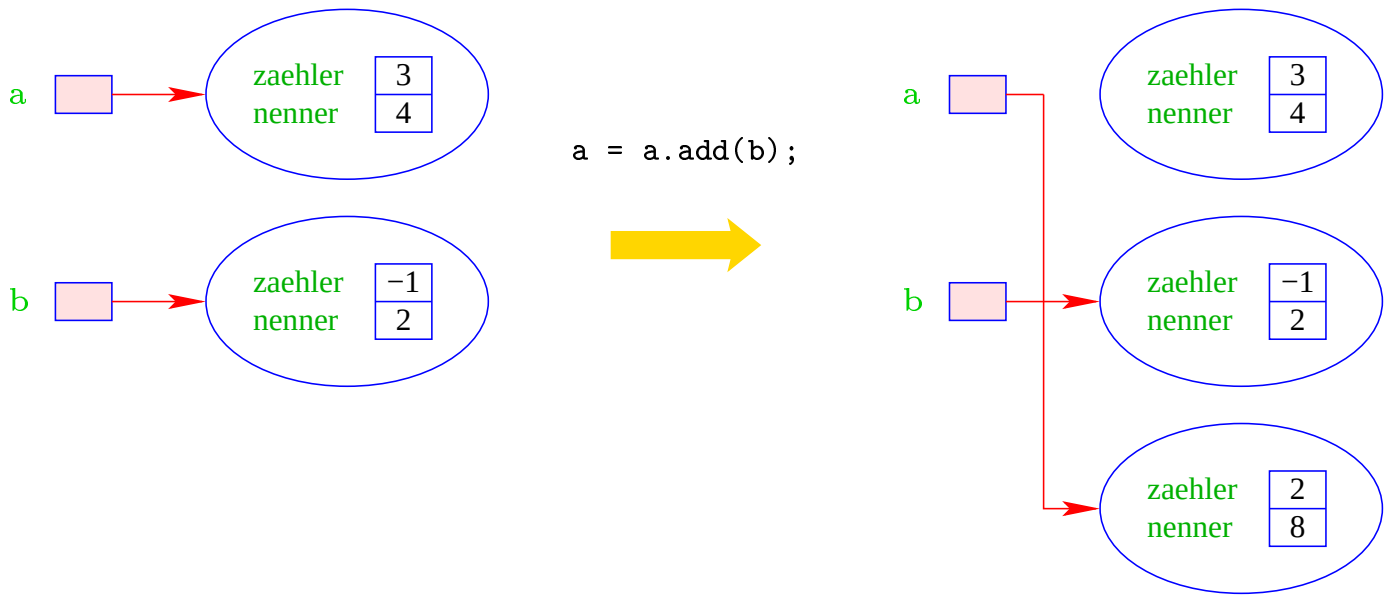
- `a.zaehler` liefert den Wert des Attributs `zaehler` des Objekts `a`:

```
int b = a.zaehler;
```



- `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
```

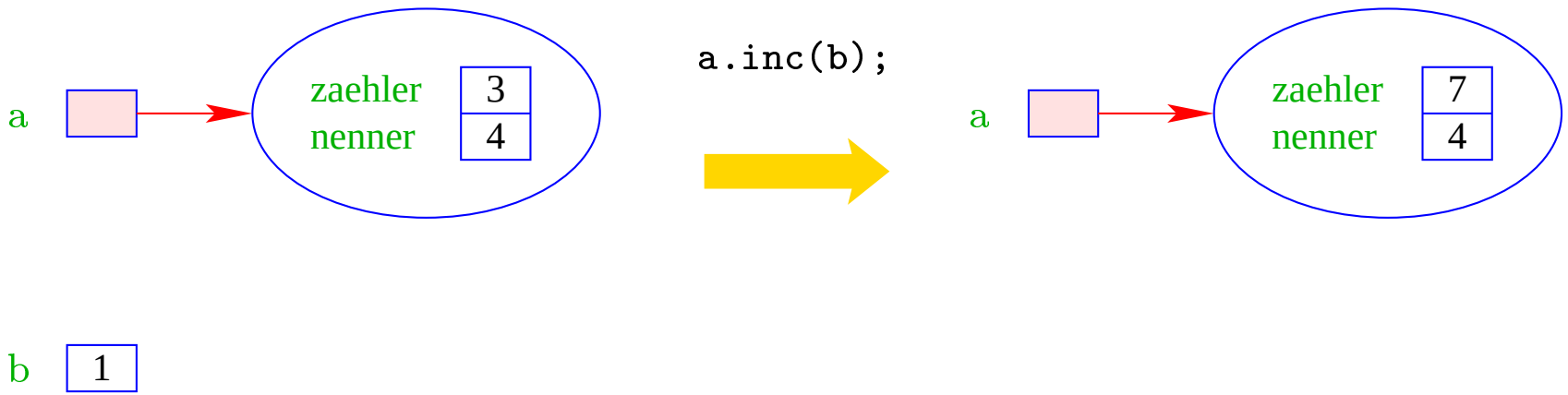
Bemerkungen:

- 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 ↑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 Rückgabetyp.
- Methoden haben dagegen stets einen Rückgabe-Typ, evt. `void`.

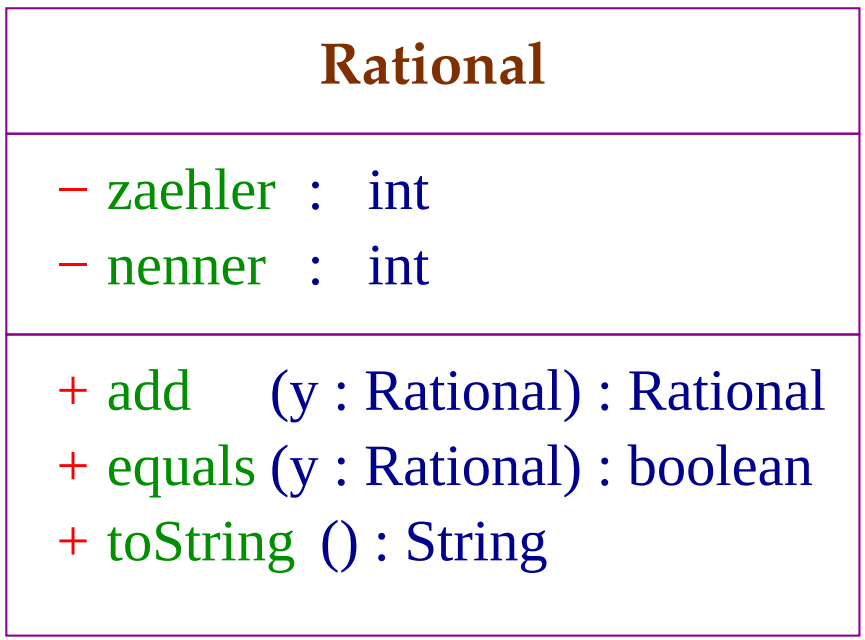
```
public void inc (int b) {
   zaehler = zaehler + b * nenner;
   }
```

- 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 Konkantenation "+" 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:



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.

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:

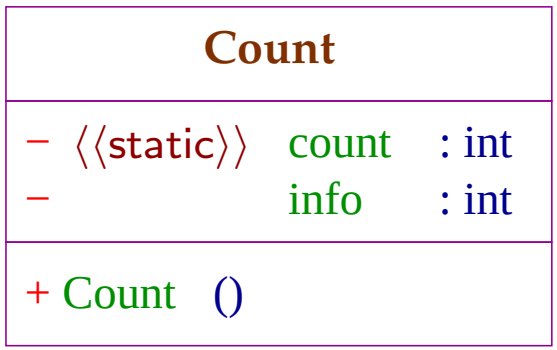
UML wurde nicht speziell für Java entwickelt. Darum werden Typen abweichend notiert. Auch lassen sich manche Ideen nicht eindeutig oder nur schlecht modellieren.

8.1 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
```

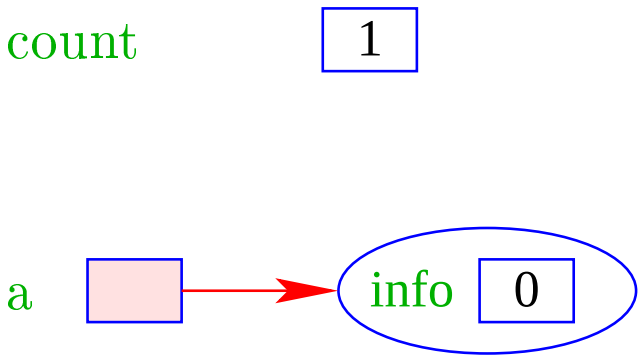
Modellierung:



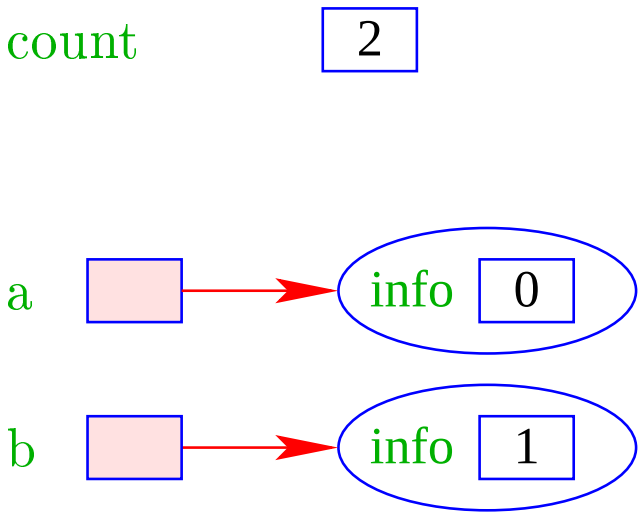
- Die Zusatzinformation, dass das Attribut `count` statisch ist, wird in in spitzen Klammern im Diagramm vermerkt.
- Solche Annotationen heißen Stereotype.

count 0

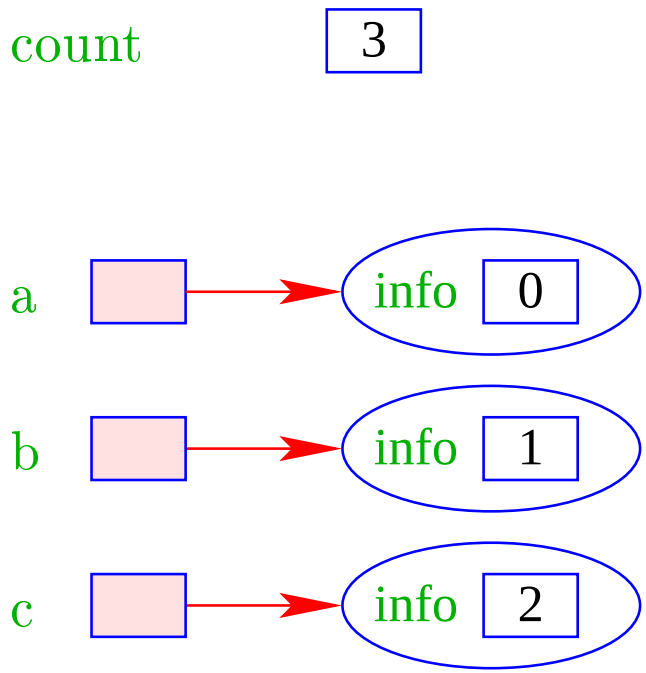
```
Count a = new Count();
```




```
Count b = new Count();
```




```
Count c = new Count();
```


count
3
a
info
0
b
info
1
c
info
2

- 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 zugegreifen.

- 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` zugegreifen.

- 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;
   }
```

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;
   }
```

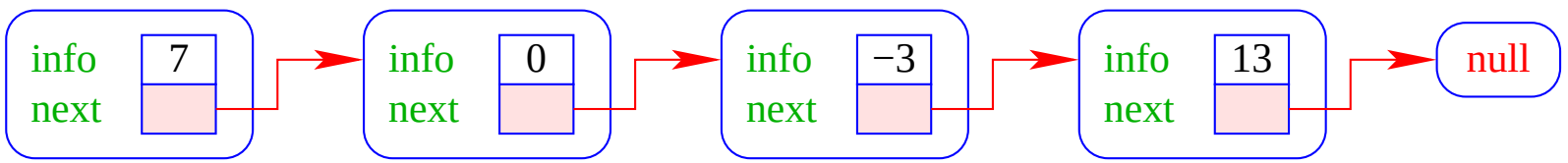
- Die Funktion erzeugt für ein Feld von `int`'s ein entsprechendes Feld von `Rational`-Objekten.
- Außerhalb der Klasse Class kann die öffentliche Klassen-Methode meth() mithilfe von `Class.meth(...)` aufgerufen werden.

9 Ein konkreter Datentyp: Listen

Nachteil von Feldern:

- feste Größe;
- Einfügen neuer Elemente nicht möglich;
- Streichen ebenfalls nicht.

Idee: Listen



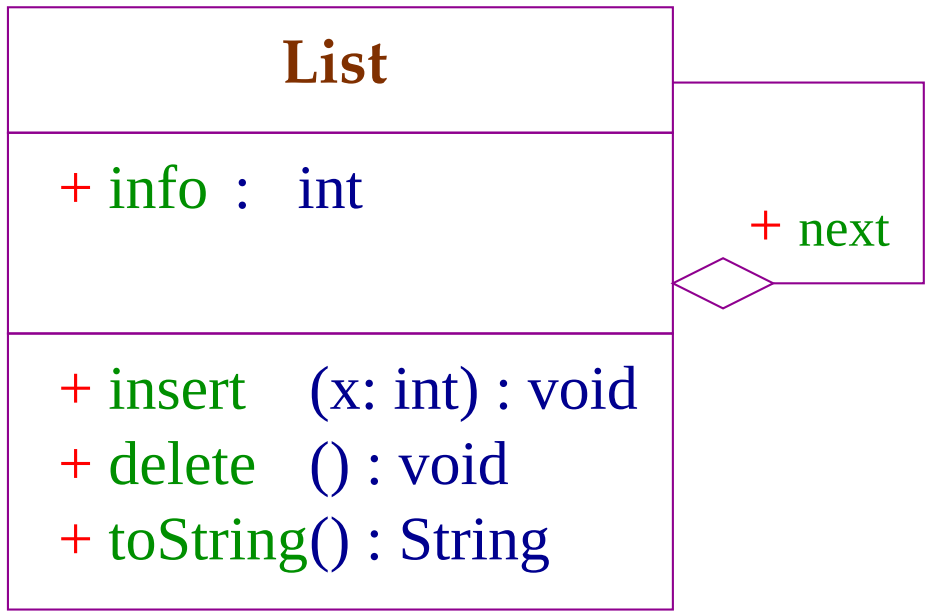
... das heißt:

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

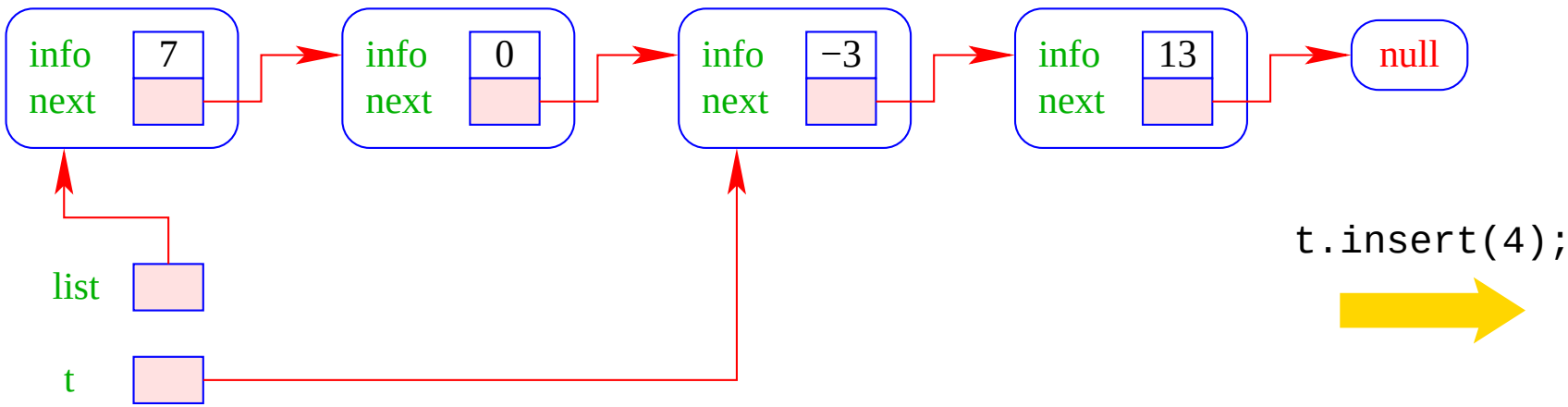
Operationen:

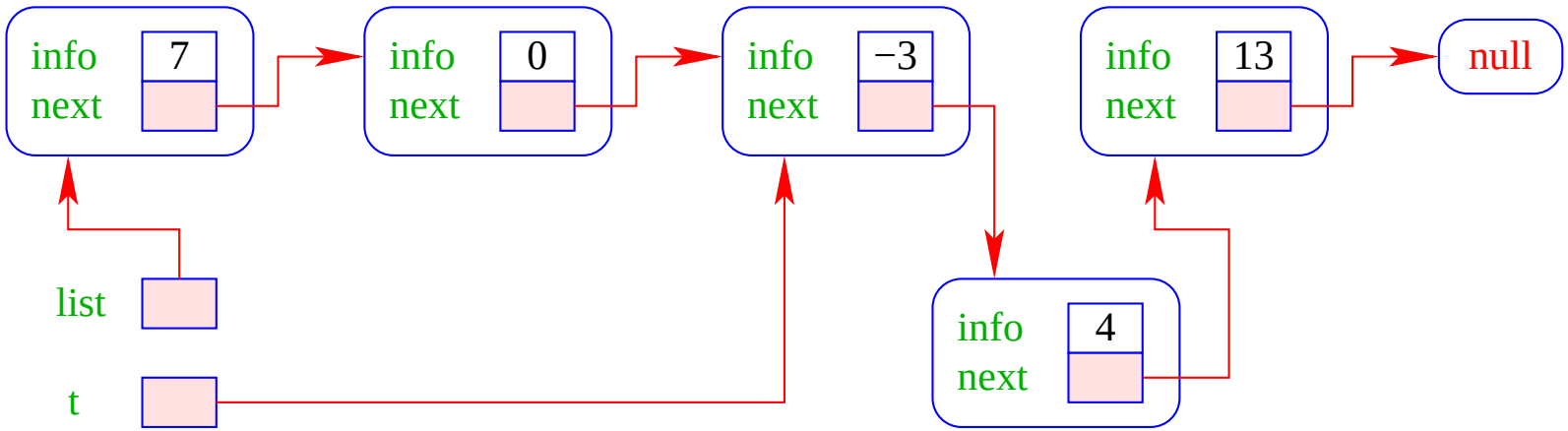
`void insert(int x)`	:	fügt neues x hinter dem aktuellen Element ein;
`void delete()`	:	entfernt Knoten hinter dem aktuellen Element;
`String toString()`	:	liefert eine String-Darstellung.

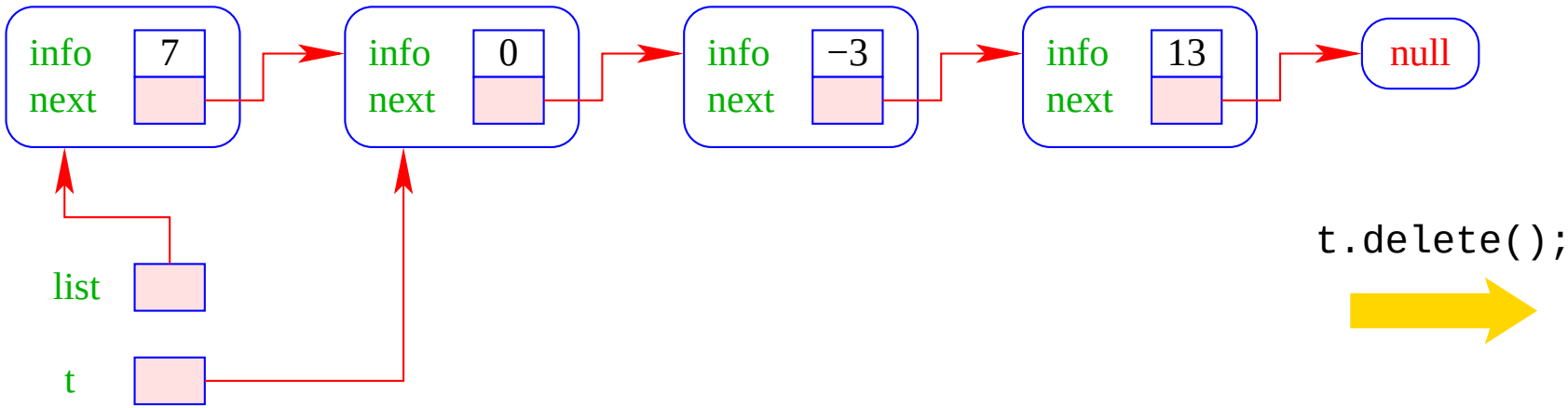
Modellierung:

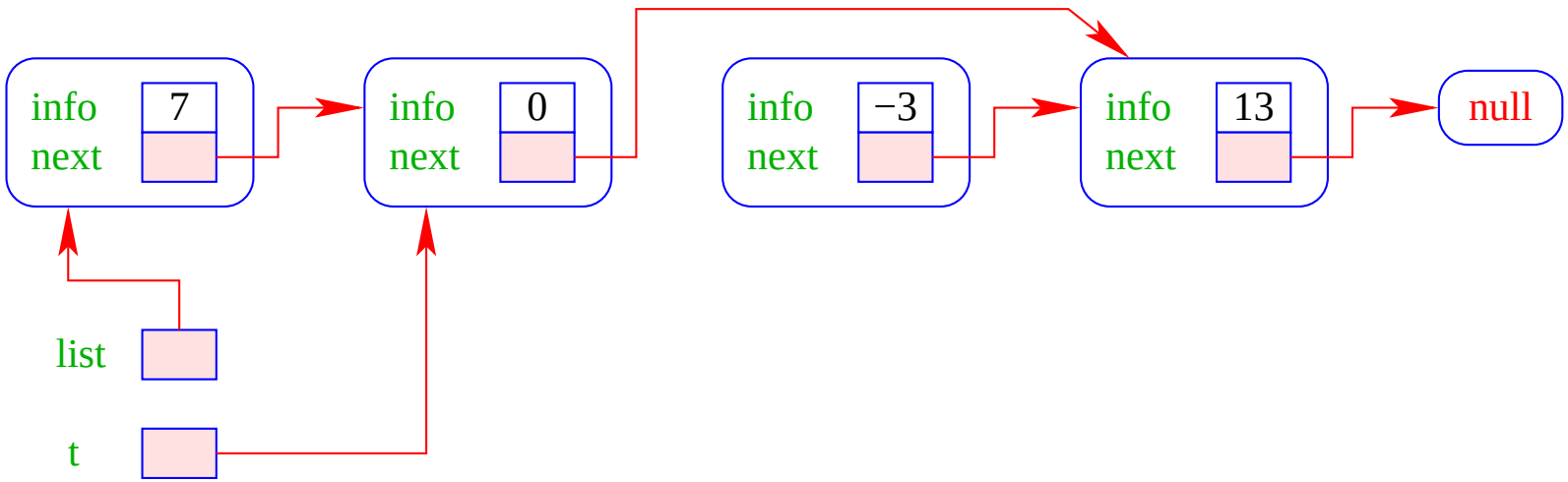


Die Rauten-Verbindung heißt auch Aggregation.


info
next
7
info
next
0
info
next
−3
info
next
13
null
list
t
t.insert(4);


info
next
7
info
next
0
info
next
−3
info
next
13
null
info
next
4
list
t


info
next
7
info
next
0
info
next
−3
info
next
13
null
list
t
t.delete();


info
next
7
info
next
0
info
next
−3
info
next
13
null
list
t

Weiterhin sollte man

- ... eine Liste auf Leerheit testen können;

- ... 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.

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;
      }
   ...
```

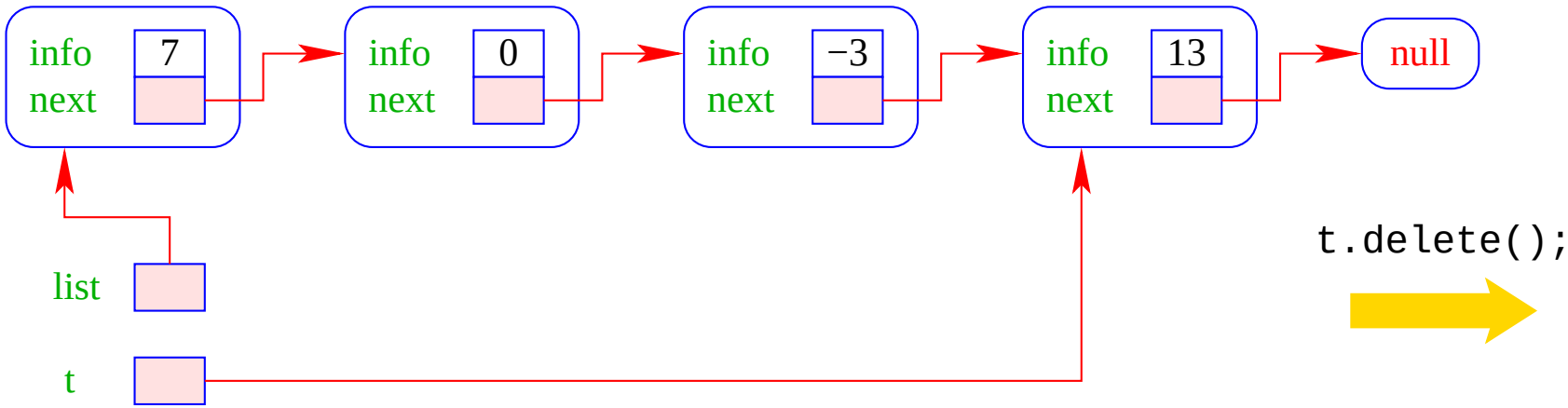
```
// Objekt-Methoden:
   public void insert(int x) {
      next = new List(x,next);
      }
   public void delete() {
      if (next != null)
         next = next.next;
      }
   public String toString() {
      String result = "["+info;
      for(List t=next; t!=null; t=t.next)
         result =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

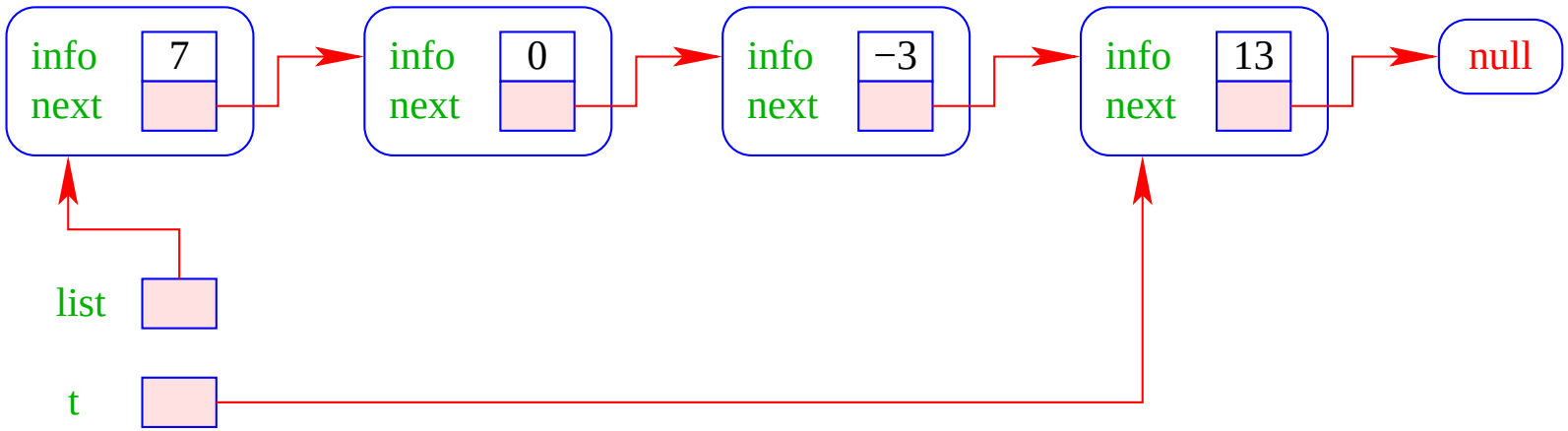
- Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar $\Longrightarrow$ sehr flexibel, sehr fehleranfällig.
- `insert()` legt einen neuen Listenknoten an fügt ihn hinter dem aktuellen Knoten ein.
- `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

Achtung:

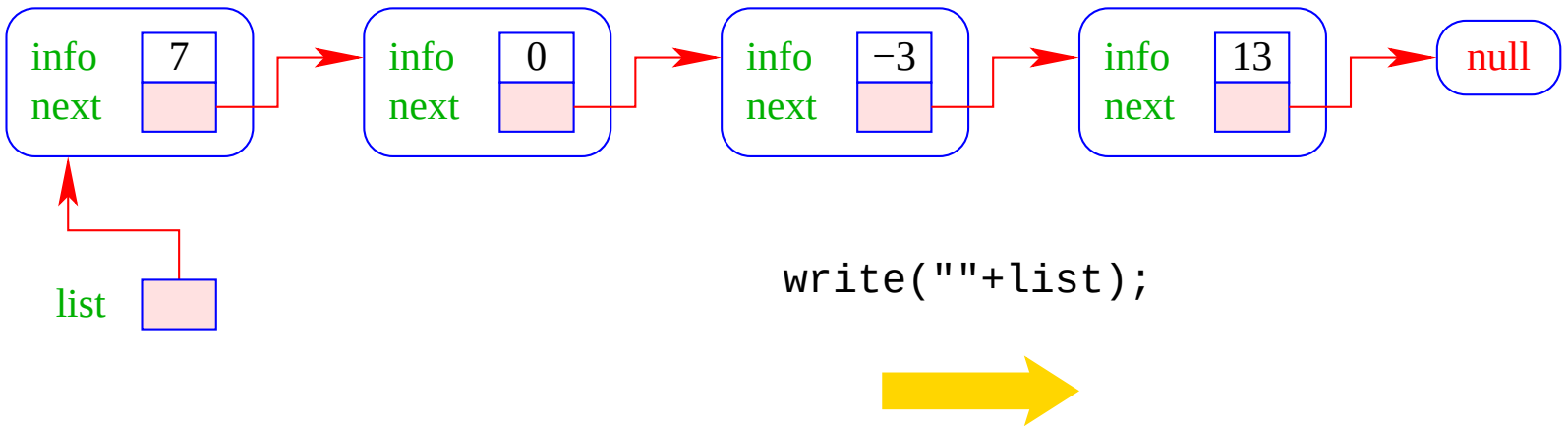
Wenn `delete()` mit dem letzten Element der Liste aufgerufen wird, zeigt `next` auf `null`.

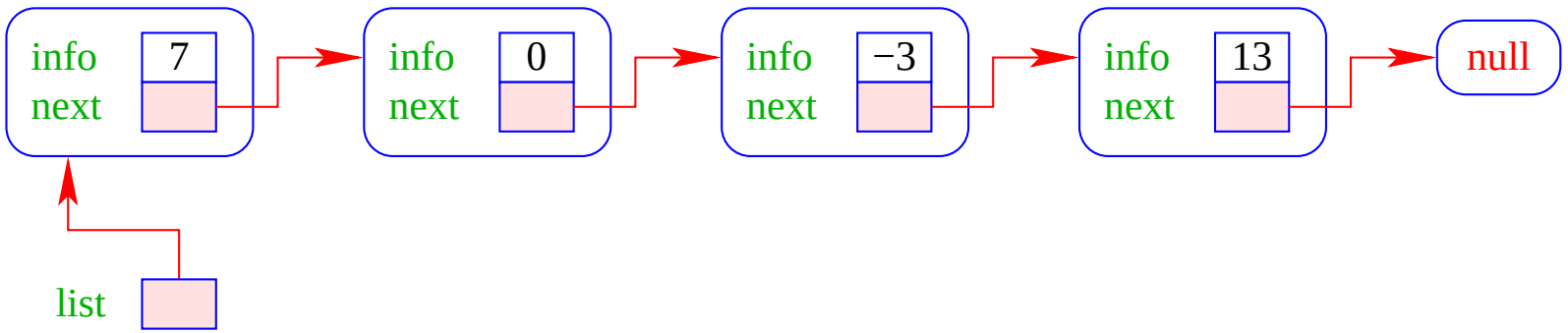
$\Longrightarrow$ Wir tun dann nix.


info
next
7
info
next
0
info
next
−3
info
next
13
null
list
t
t.delete();


info
next
7
info
next
0
info
next
−3
info
next
13
null
list
t

- 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)`.


info
next
7
info
next
0
info
next
−3
info
next
13
null
list
write(""+list);




```
"[7, 0, −3, 13]"
```


```
write(""+list);
```

`"null"`

```
// 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.

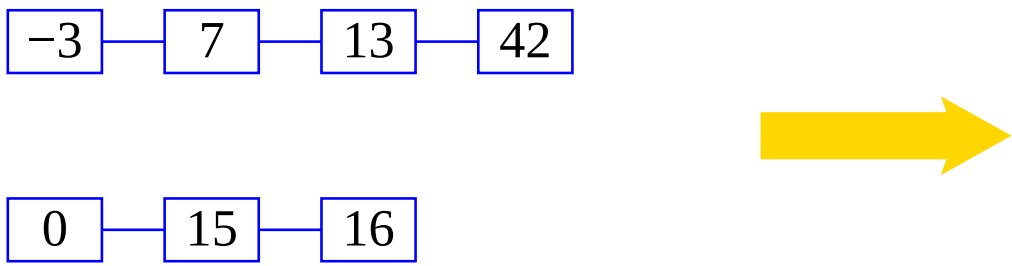
Anwendung: Mergesort – Sortieren durch Mischen

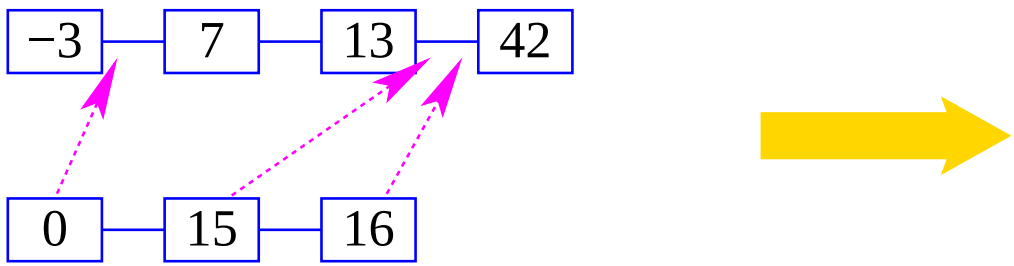
John von Neumann (1945)

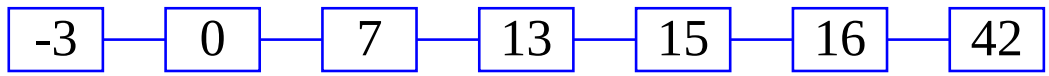
Mischen:

Eingabe: zwei sortierte Listen;

Ausgabe: eine gemeinsame sortierte Liste.

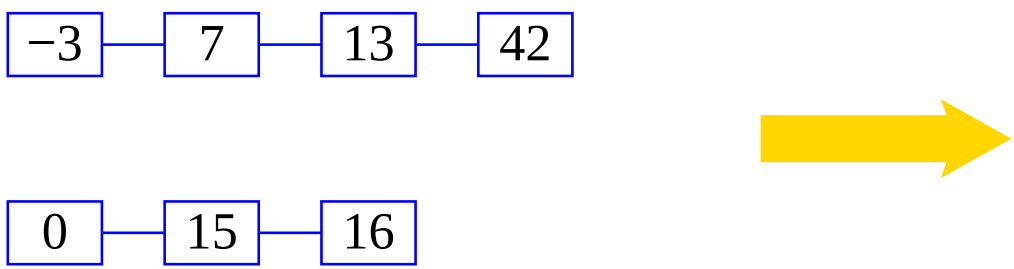

−3
7
13
42
0
15
16

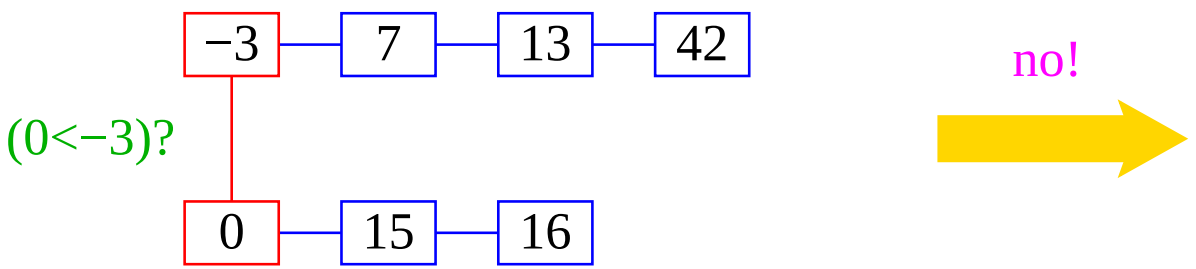

−3
7
13
42
0
15
16

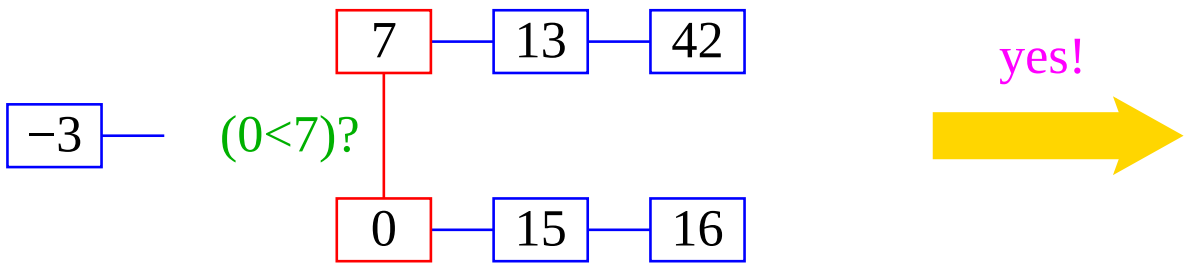

-3
0
7
13
15
16
42

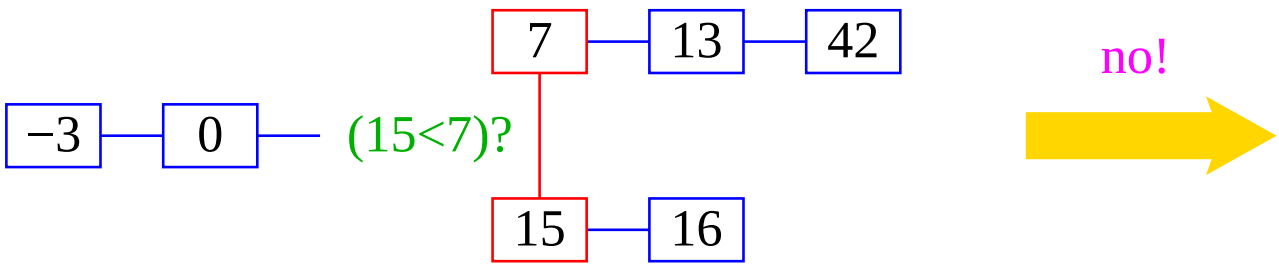
Idee:

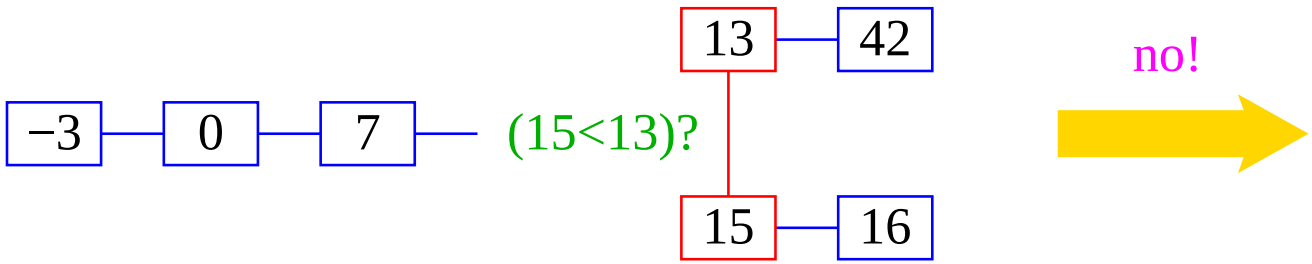
- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Input-Listen.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche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

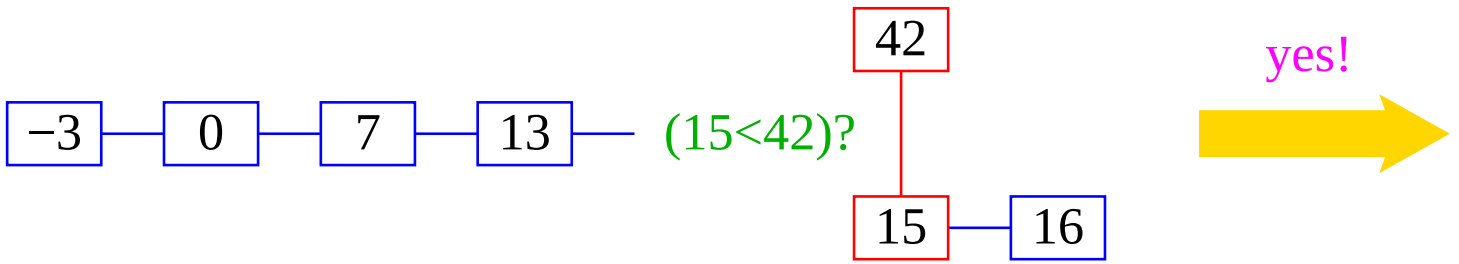

−3
7
13
42
0
15
16

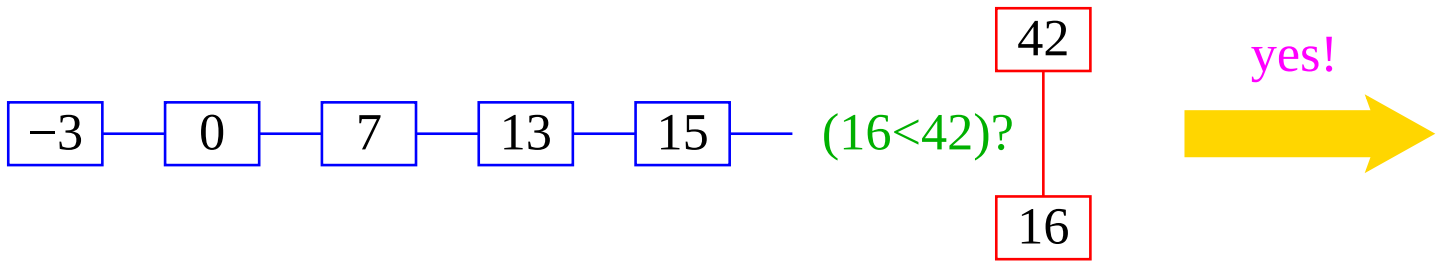

−3
7
13
42
(0<−3)?
no!
0
15
16

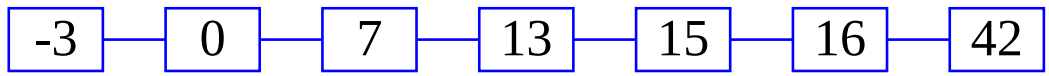

7
13
42
−3
(0<7)?
yes!
0
15
16


7
13
42
−3
0
(15<7)?
15
16
no!


13
42
−3
0
7
(15<13)?
15
16
no!


42
yes!
−3
0
7
13
(15<42)?
15
16

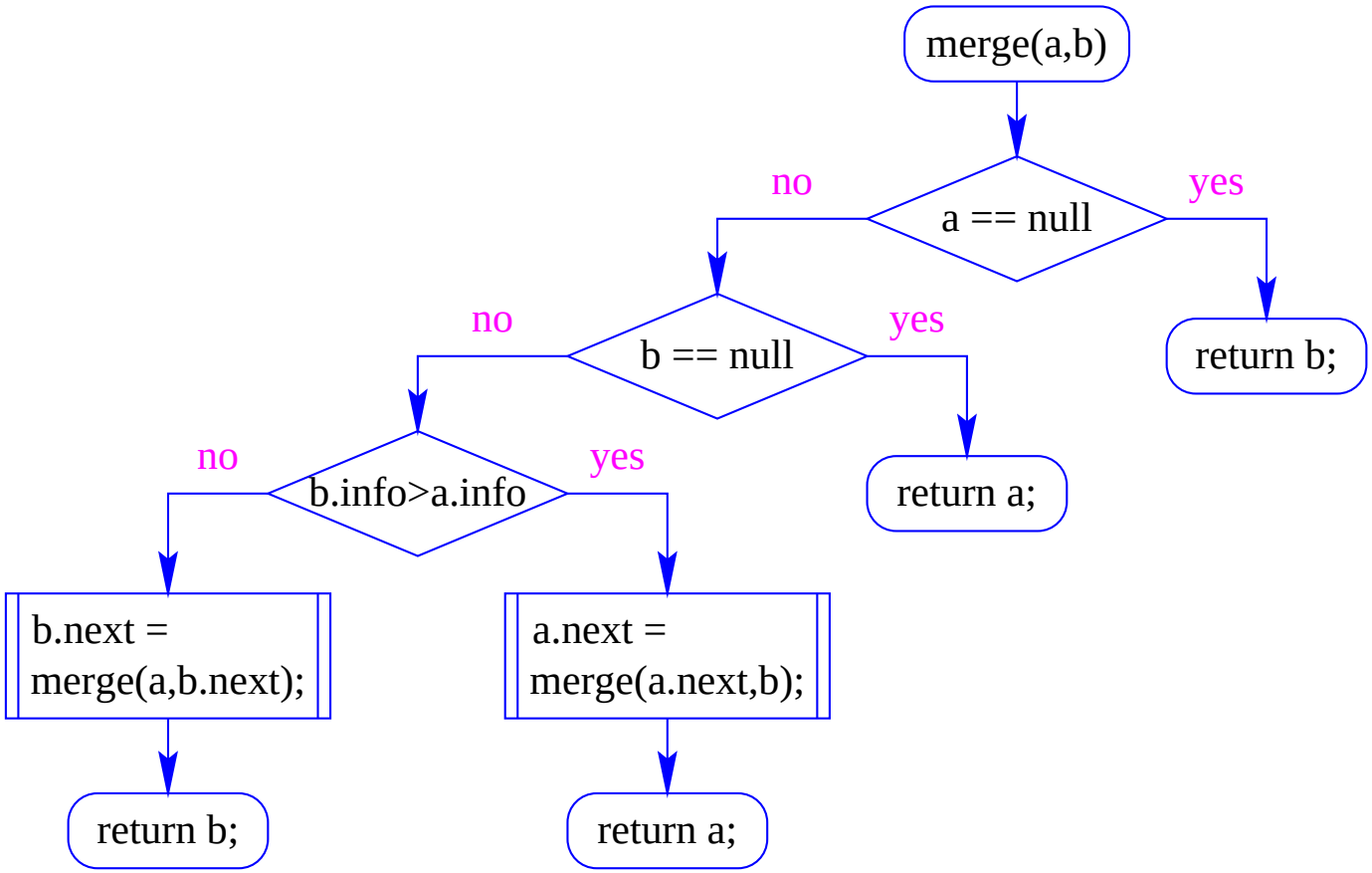

−3
0
7
13
15
(16<42)?
42
16
yes!


-3
0
7
13
15
16
42

Rekursive Implementierung:

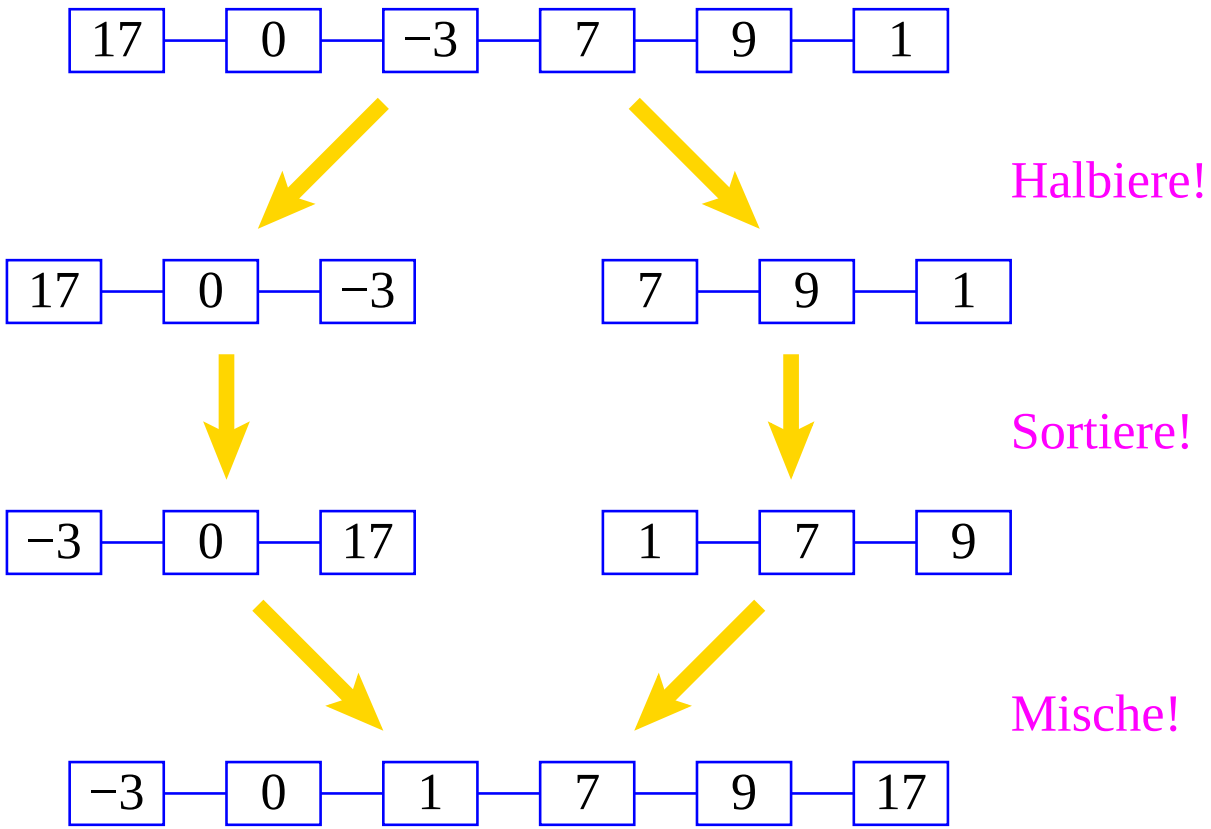
- Falls eine der beiden Listen `a` und `b` leer ist, geben wir die andere aus.
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente erhalten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

```
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;
      }
   }
```


merge(a,b)
a == null
no
yes
return b;
b == null
no
yes
return a;
b.info>a.info
no
yes
b.next =
merge(a,b.next);
a.next =
merge(a.next,b);
return b;
return a;

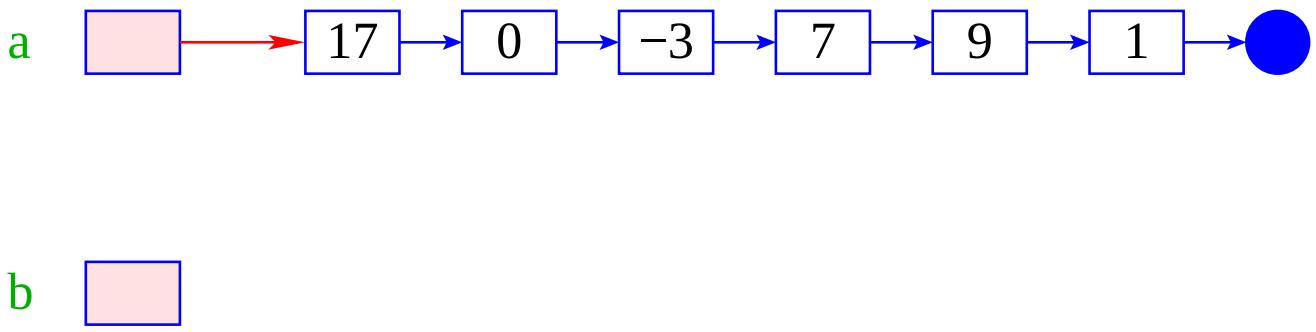
Sortieren durch Mischen:

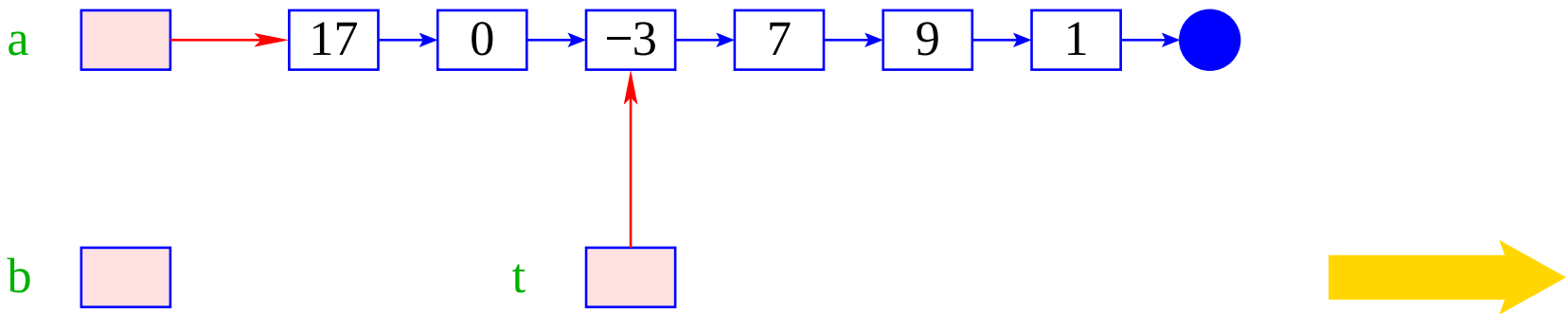
- Teile zu sortierende Liste in zwei Teil-Listen;
- sortiere jede Hälfte für sich;
- mische die Ergebnisse!

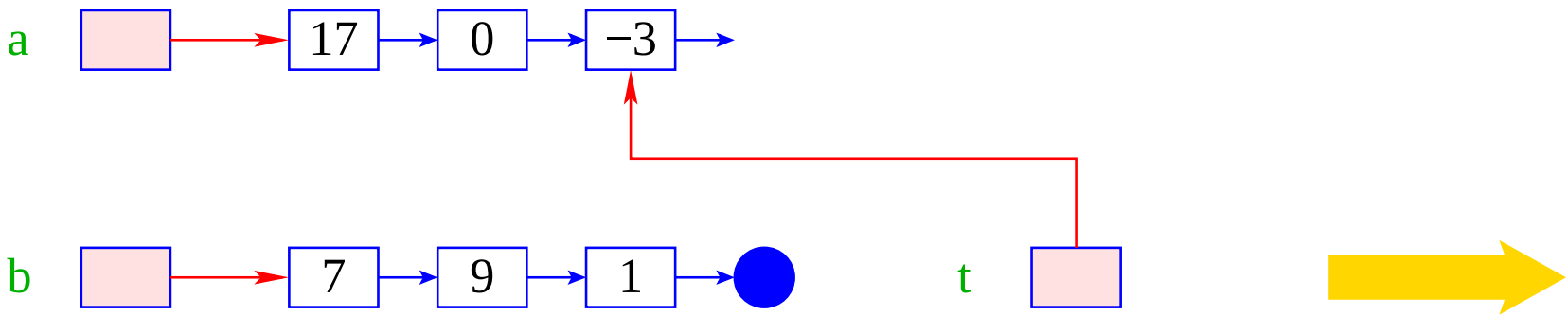

17 0 −3 7 9 1
Halbiere!
17 0 −3
7 9 1
Sortiere!
−3 0 17
1 7 9
Mische!
−3 0 1 7 9 17

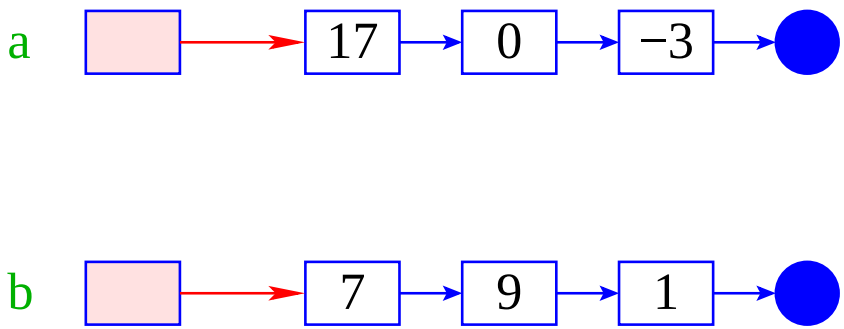
```
public static List sort(List a) {
   if (a == null || a.next == null)
      return a;
   List b = a.half(); // Halbiere!
   a = sort(a);
   b = sort(b);
   return merge(a,b);
   }
```

```
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;
   }
```


a
17
0
−3
7
9
1
b
b = a.half();


a
17
0
−3
7
9
1
b
t


a
17
0
−3
b
7
9
1
t


a
17
0
−3
b
7
9
1

Diskussion:

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

  Dann gilt:

$$\begin{aligned} V(1) &= 0 \\ V(2n) &\leq 2 \cdot V(n) + 2 \cdot n \end{aligned}$$

- 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 impliementieren (wie?)

10 Abstrakte Datentypen

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

$\Longrightarrow$ 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).

10.1 Beispiel 1: Keller (Stacks)

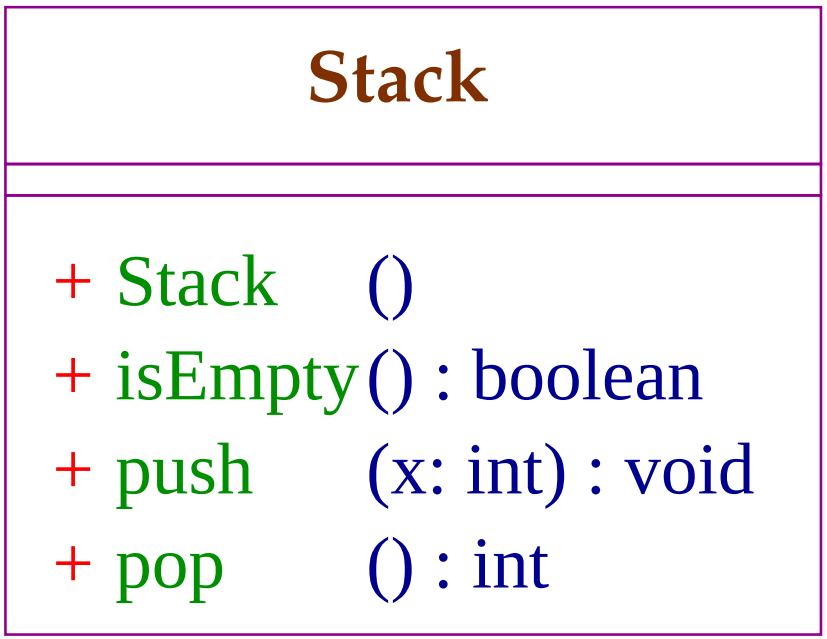
Operationen:

`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.

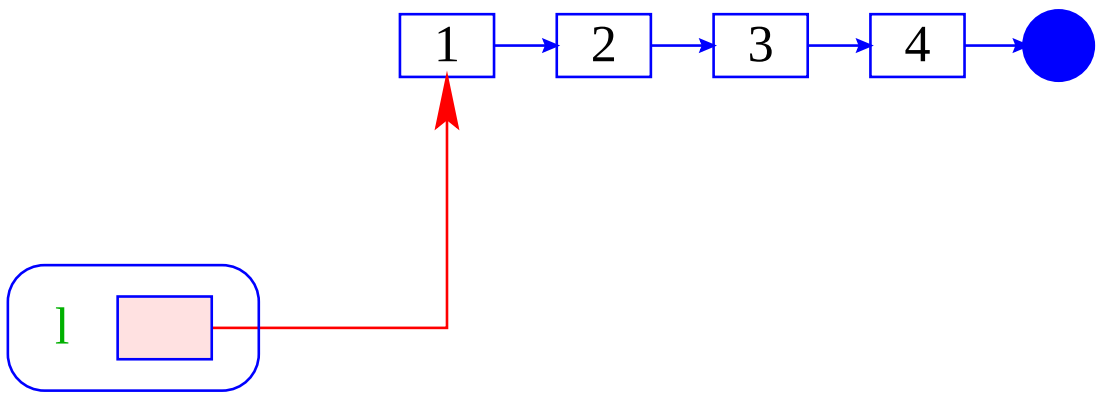
Friedrich Ludwig Bauer, TUM

Modellierung:



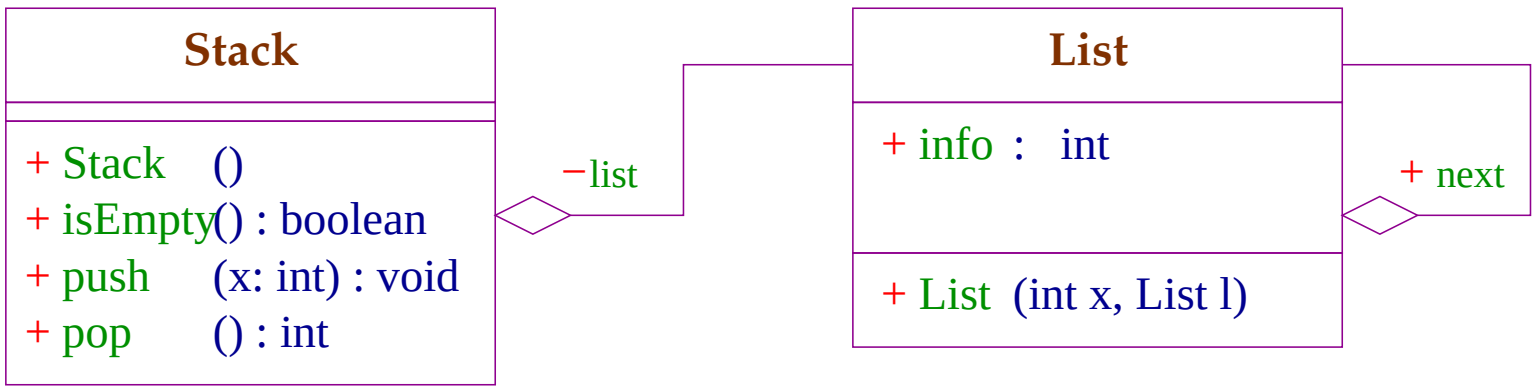
Erste Idee:

- Realisiere Keller mithilfe einer Liste!



- Das Attribut `l` 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;

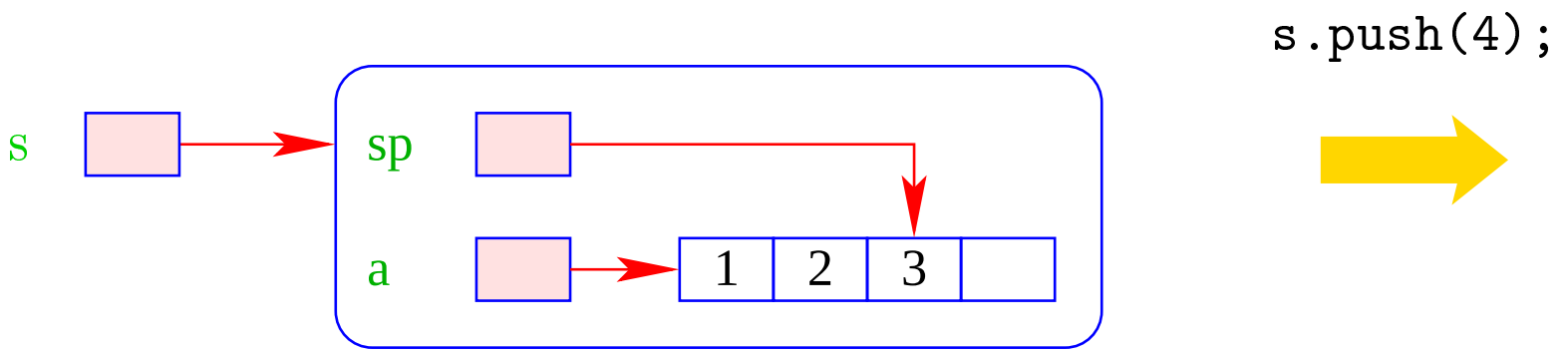
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms!

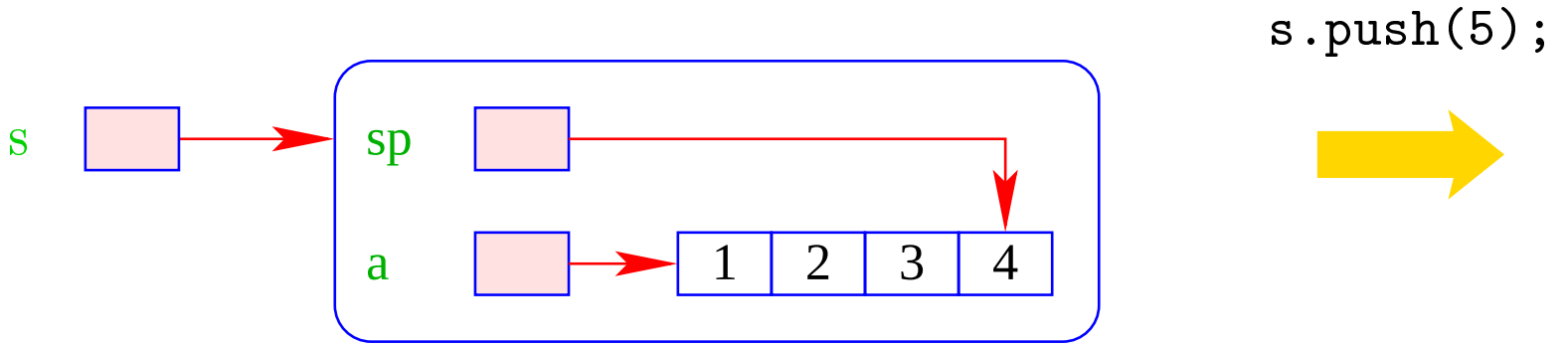
- Die Implementierung ist sehr einfach;
- ... nutzte gar nicht alle Features von `List` aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut;

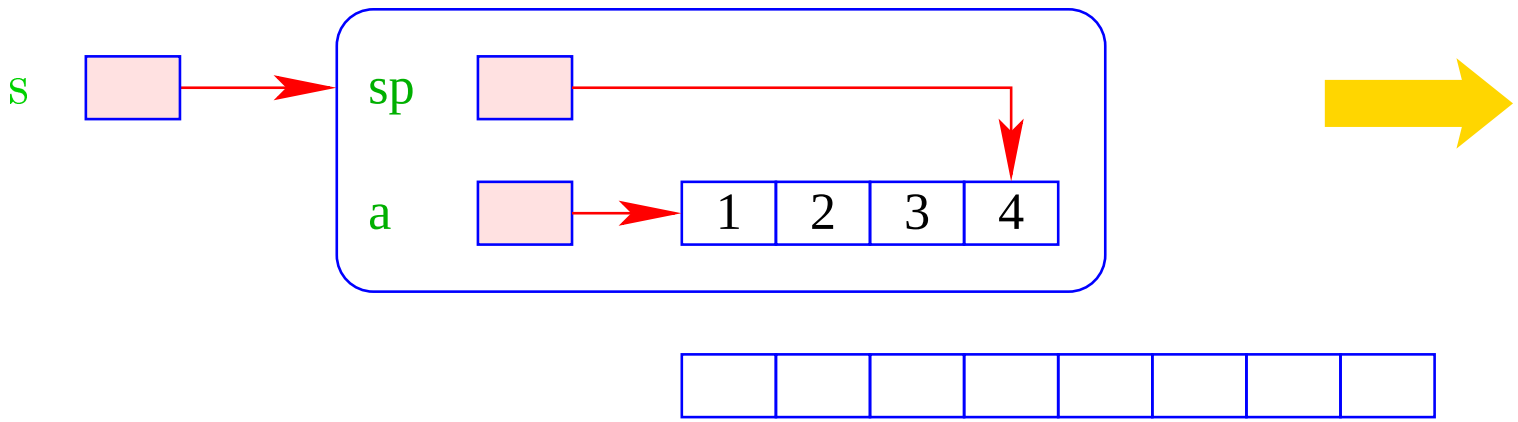
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms!

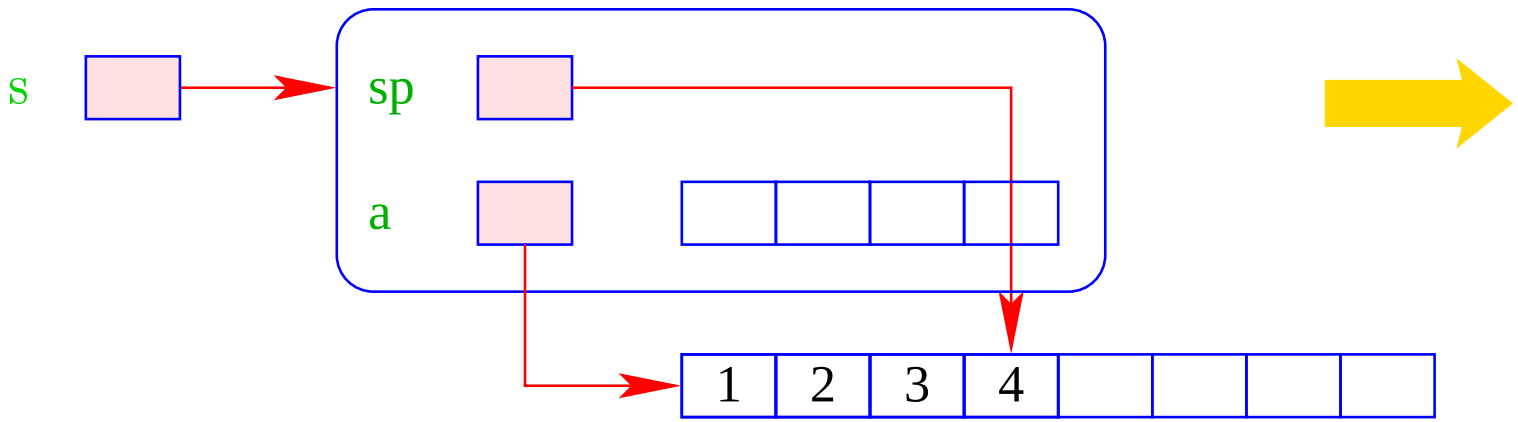
Zweite Idee:

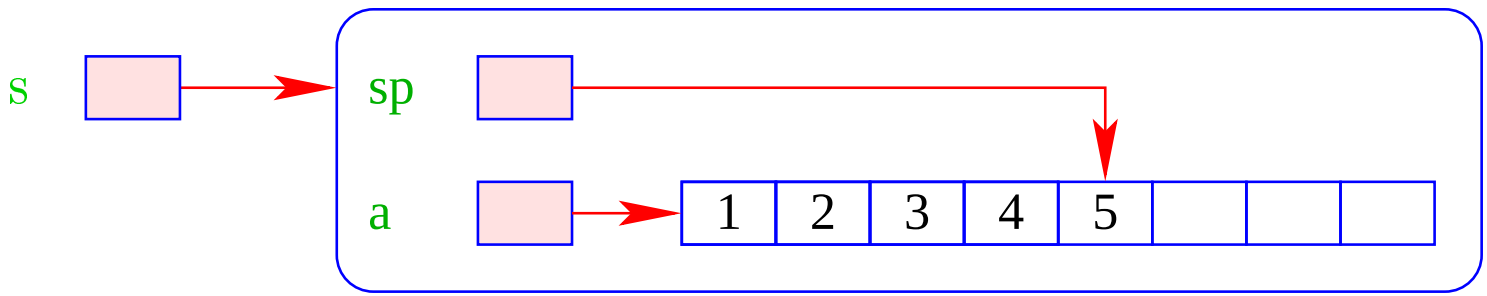
- Realisiere den Keller mithilfe eines Felds und eines Stackpointers, der auf die oberste belegte Zelle zeigt.
- Läuft das Feld über, ersetzen wir es durch ein größeres.


s
sp
a
1
2
3
s.push(4);

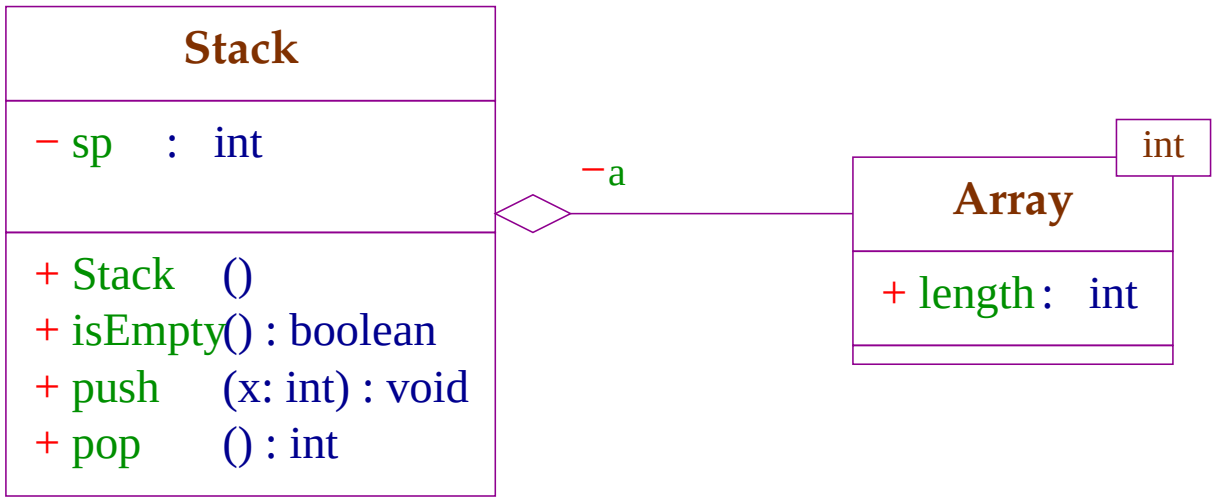

s
sp
a
1
2
3
4
s.push(5);


s
sp
a
1
2
3
4


s
sp
a
1
2
3
4


s
sp
a
1
2
3
4
5

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);
      }
   ...
```

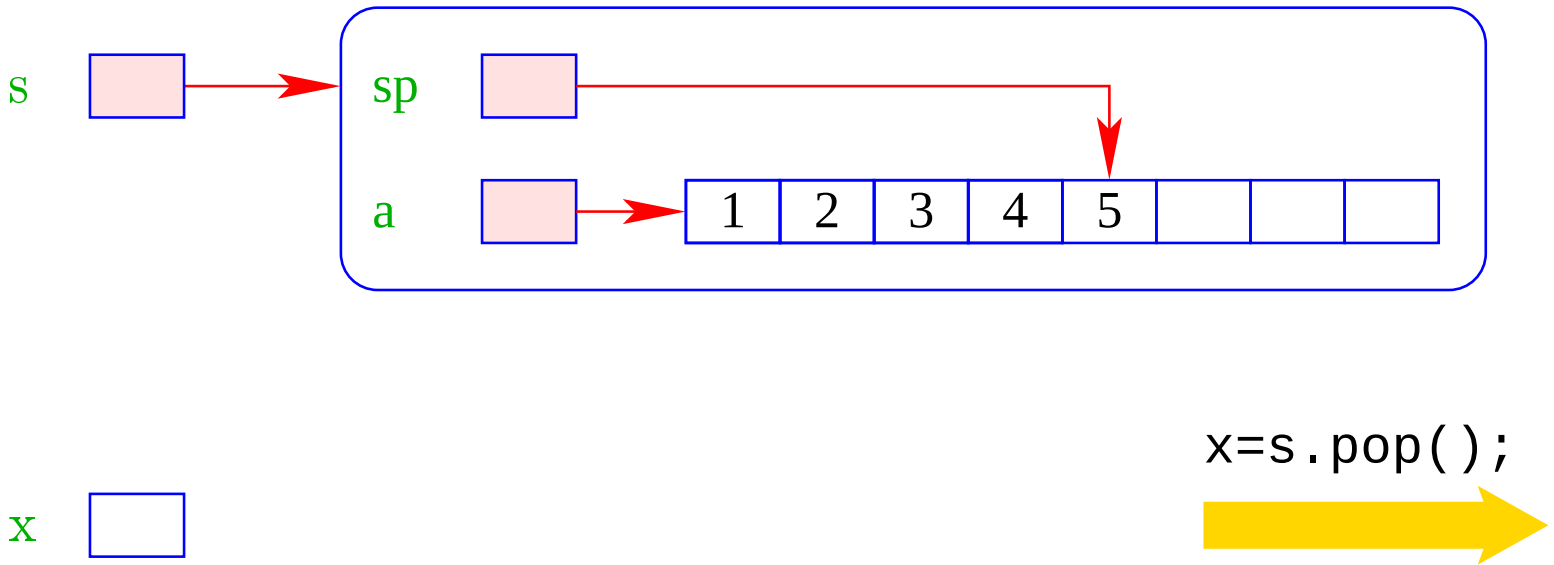
```
   public int pop() {
      return a[sp--];
      }
   public void push(int x) {
      ++sp;
      if (sp == a.length) {
         int[] b = new int[2*sp];
         for(int i=0; i<sp; ++i) b[i] = a[i];
         a = b;
         }
      a[sp] = x;
      }
   public toString() {...}
} // end of class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

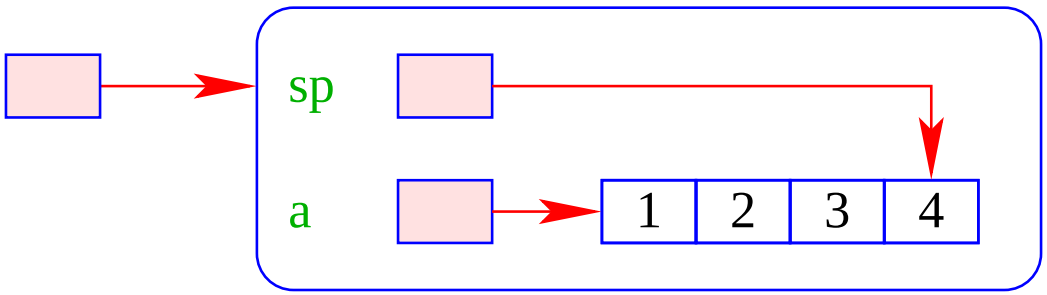
Nachteil:

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

Erste Idee:

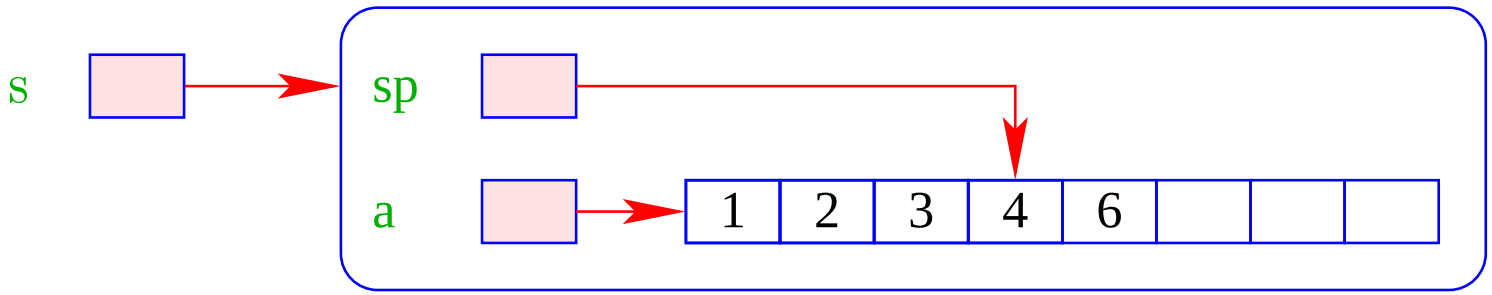
- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei ...


s
sp
a
1
2
3
4
5
x
x=s.pop();



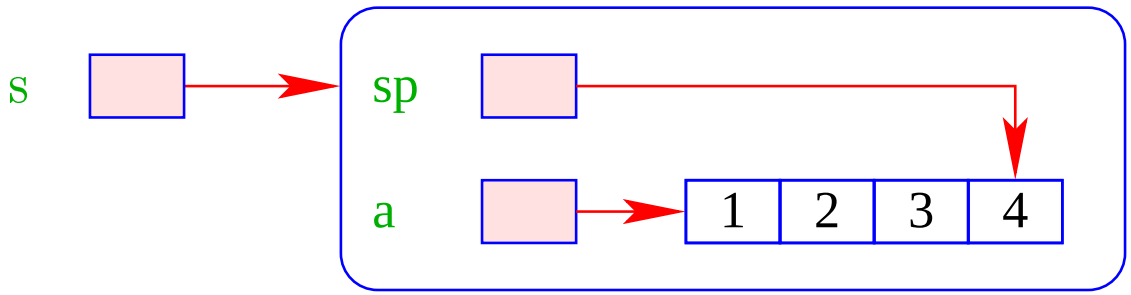
x 5

```
s.push(6);
```



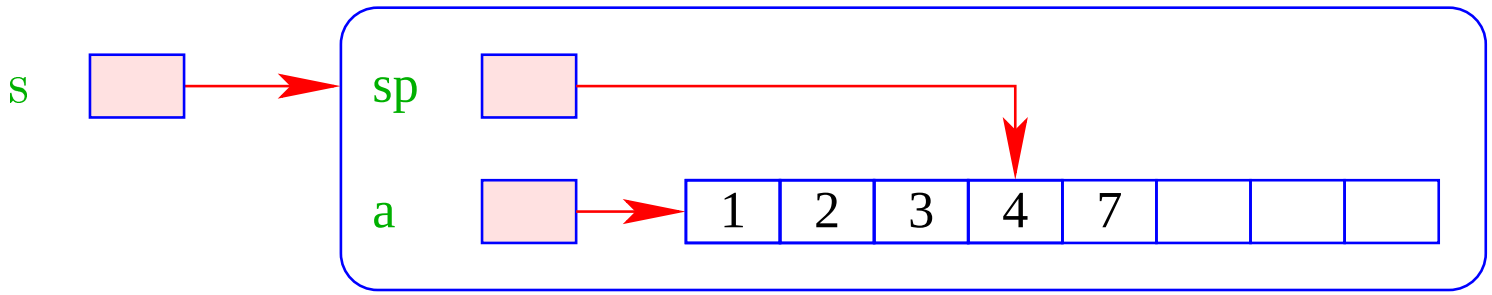

```
x = s.pop();
```


x 5



x 6

```
s.push(7);
```



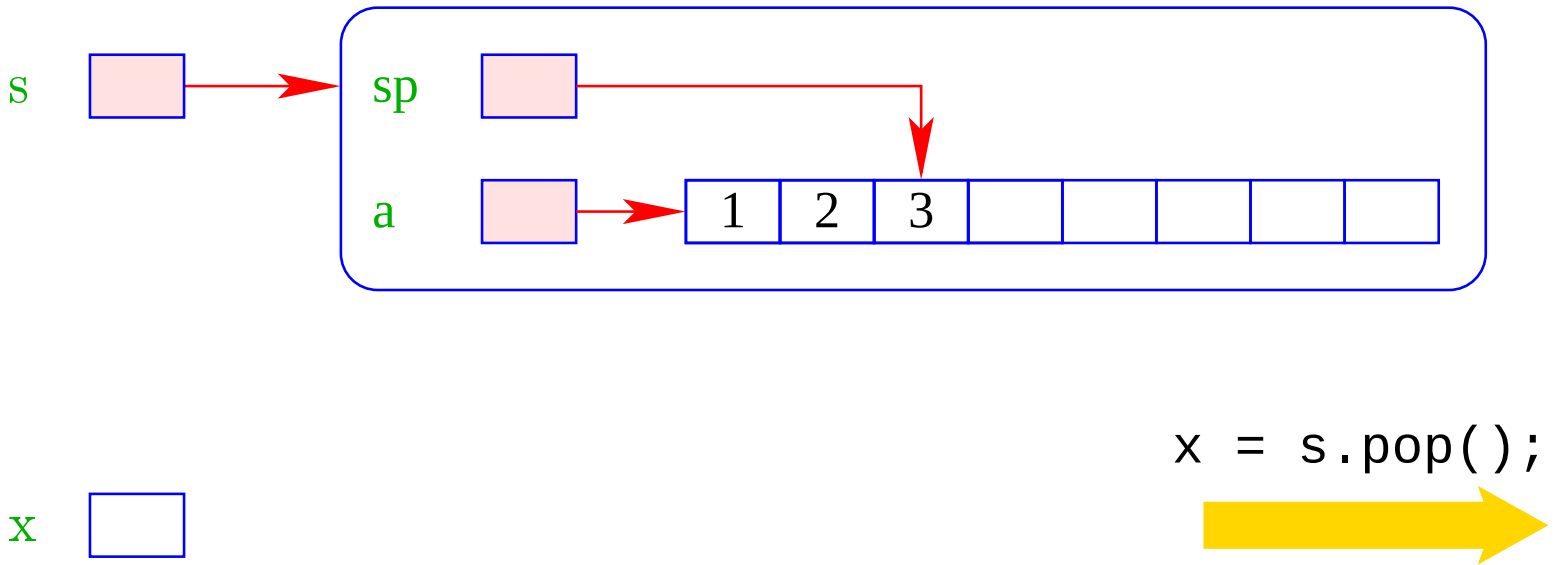

```
x = s.pop();
```

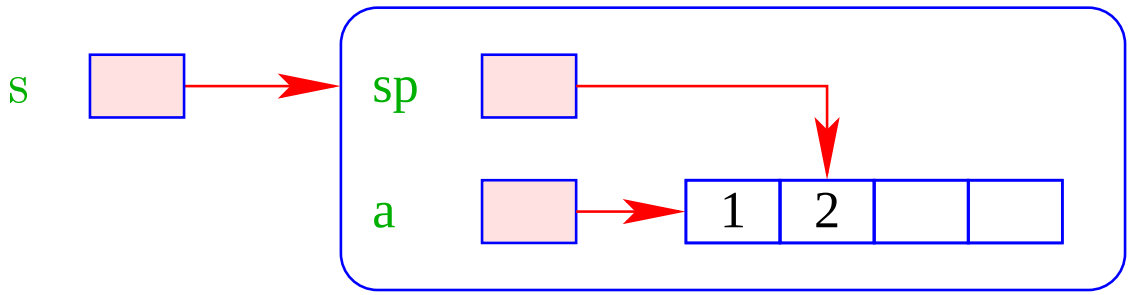

x 6

- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

Zweite Idee:

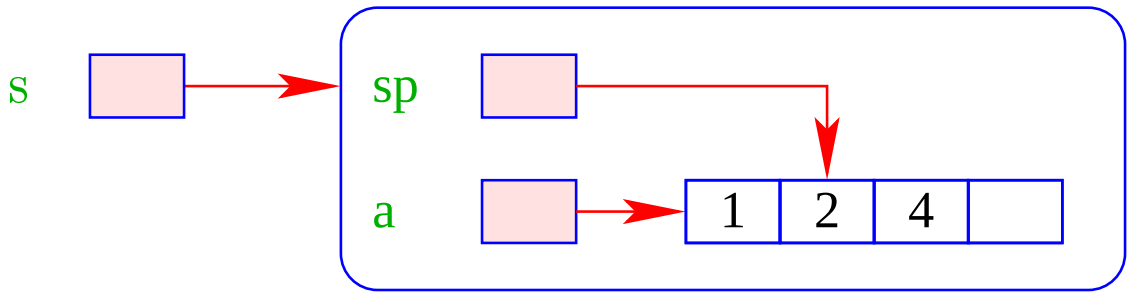
- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte !


s
sp
a
1
2
3
x
x = s.pop();



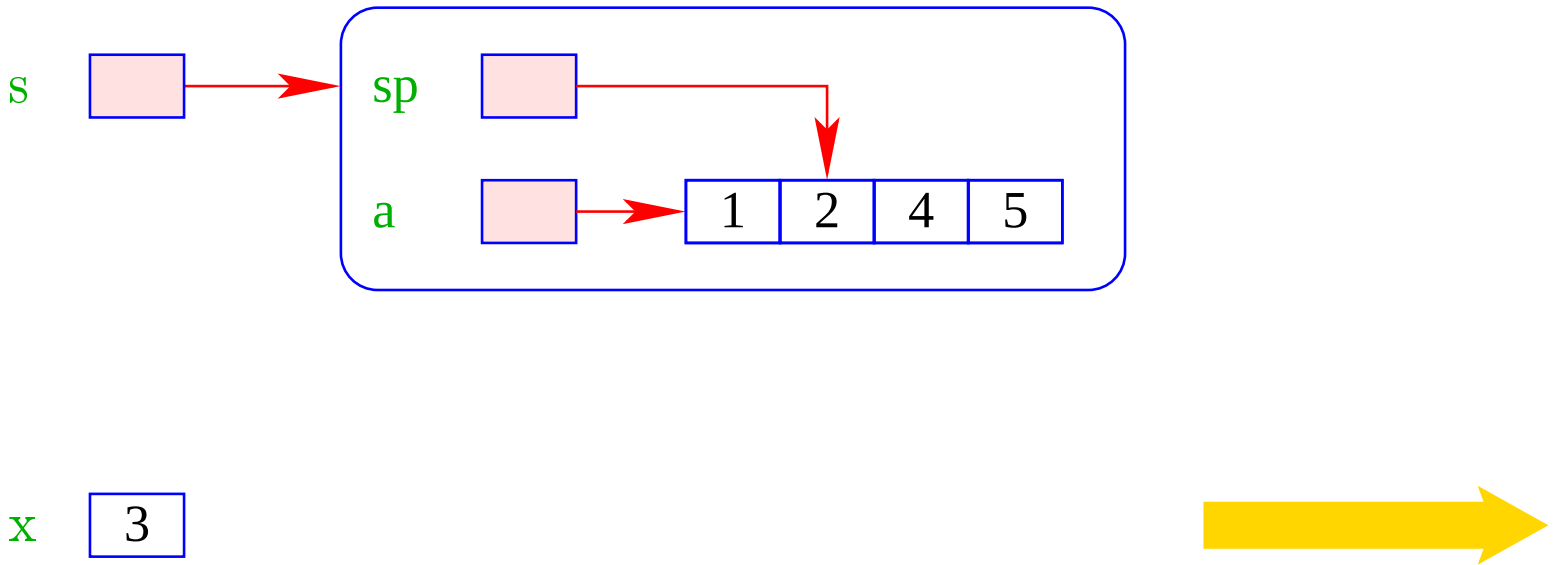
x 3

```
s.push(4);
```



x 3

```
s.push(5);
```


s
sp
a
1
2
4
5
x
3

- 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;
   }
```

10.2 Beispiel 2: Schlangen (Queues)

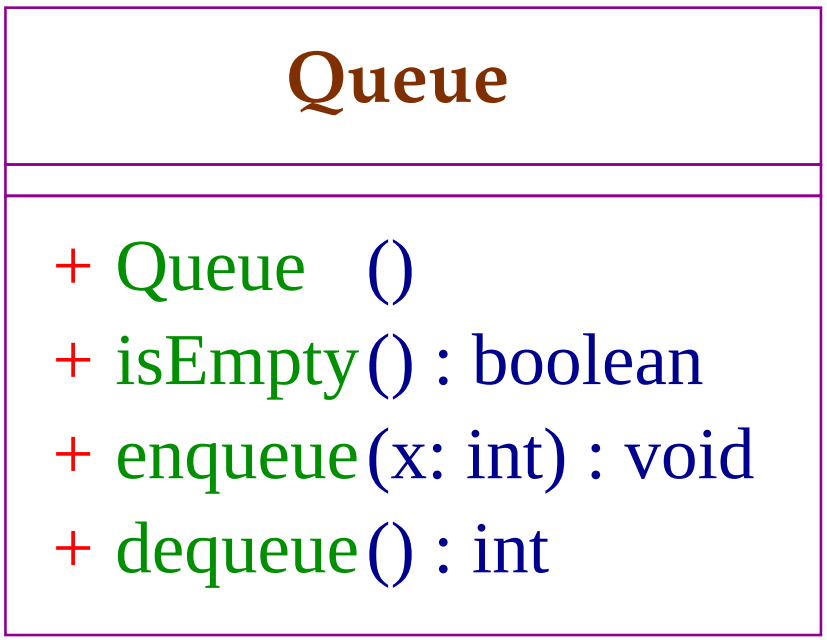
(Warte-) Schlangen verwalten ihre Elemente nach dem FIF0-Prinzip (First-In-First-Out).

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.

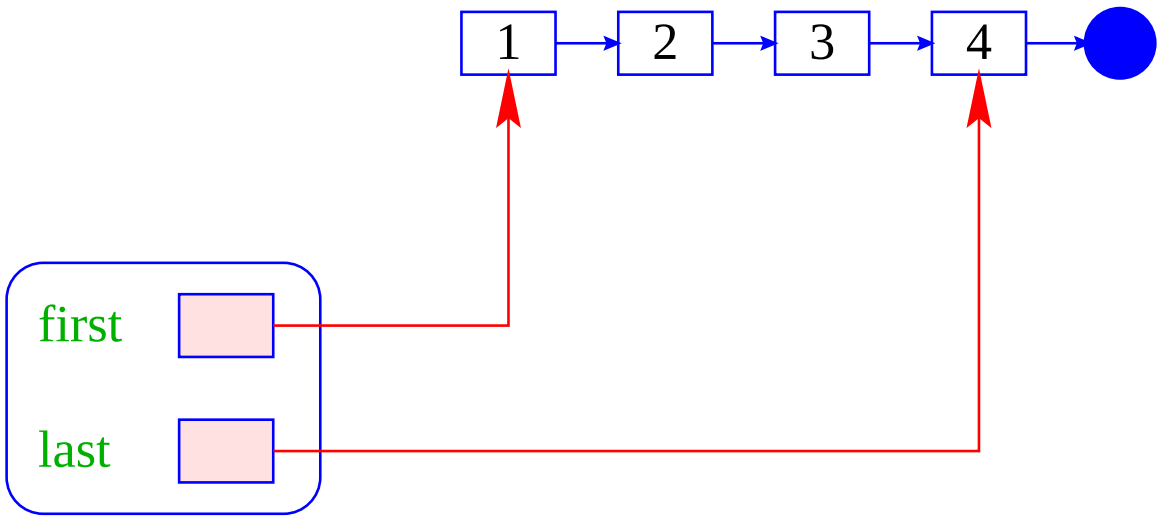
Weiterhin müssen wir eine leere Schlange anlegen können.

Modellierung:



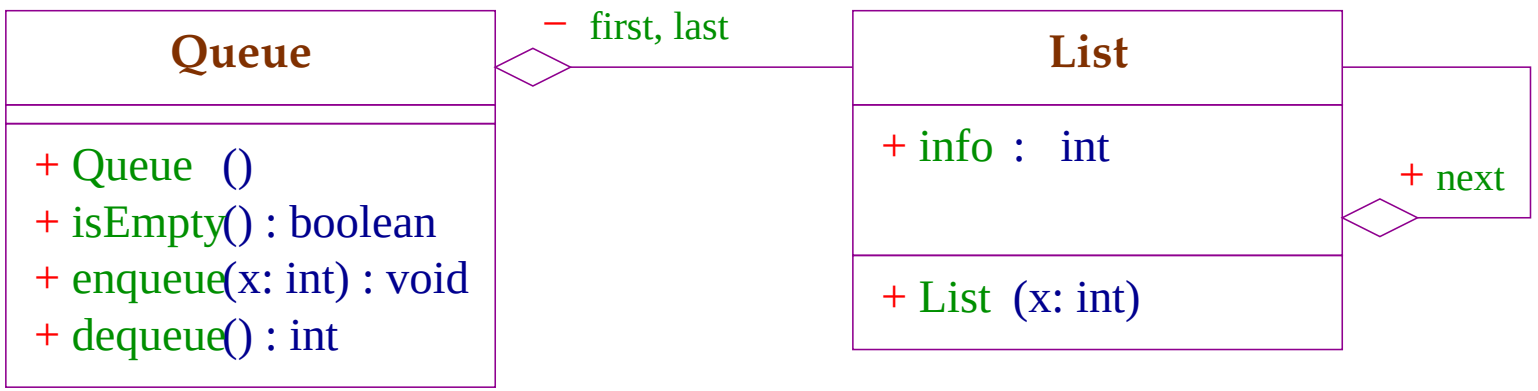
Erste Idee:

- 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:

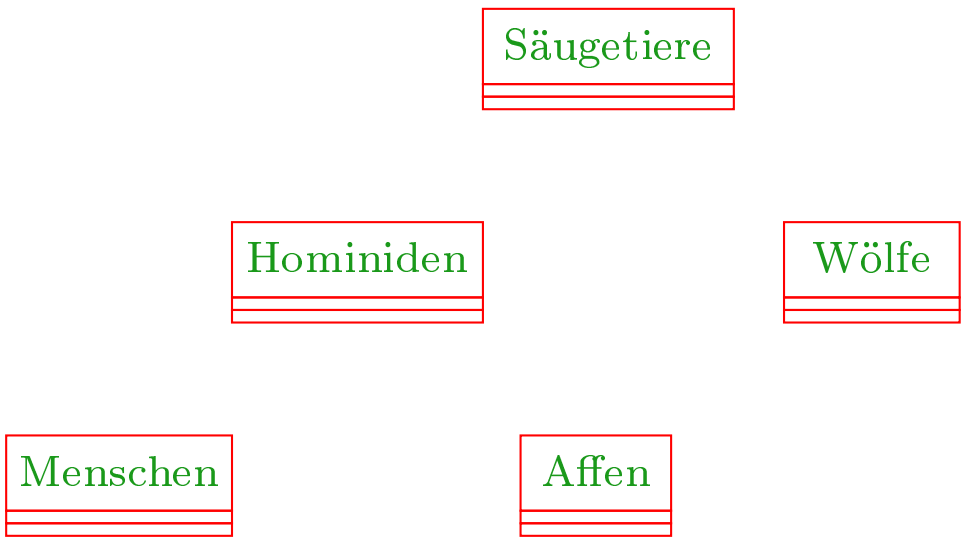
```
public class Queue {
   private List first, last;
   // Konstruktor:
   public Queue () {
      first = last = null;
      }
   // Objekt-Methoden:
   public boolean isEmpty() {
      return first==null;
      }
   ...
```

```
    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
```

11 Vererbung

Beobachtung:

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

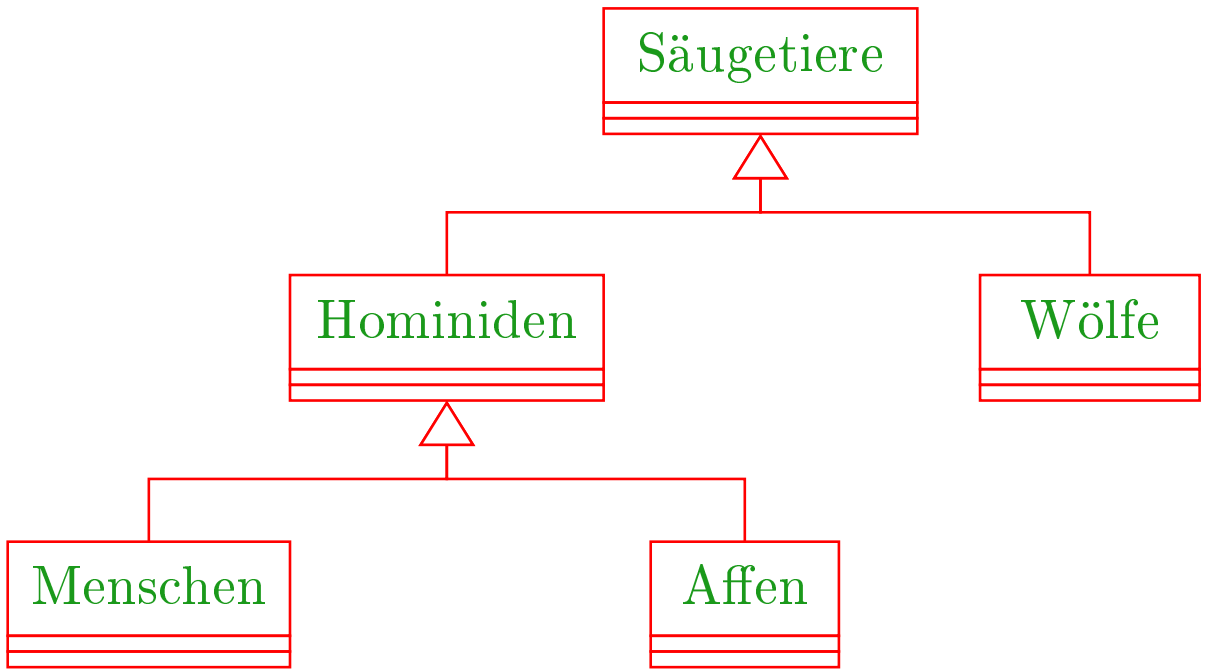


Idee:

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

$\Longrightarrow$ inkrementelles Programmieren

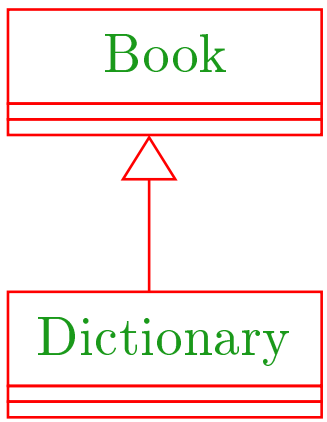
$\Longrightarrow$ Software Reuse


Säugetiere
Hominiden
Wölfe
Menschen
Affen

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:



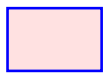
Implementierung:

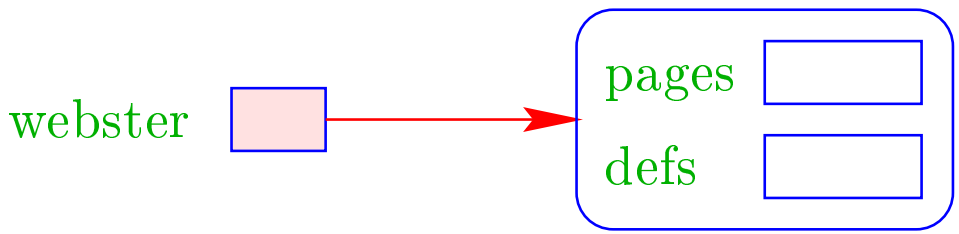
```
public class Book {
   protected int pages;
   public Book() {
      pages = 150;
   }
   public void page_message() {
      System.out.print("Number of pages:\t"+pages+"\n");
      }
   } // end of class Book
   ...
```

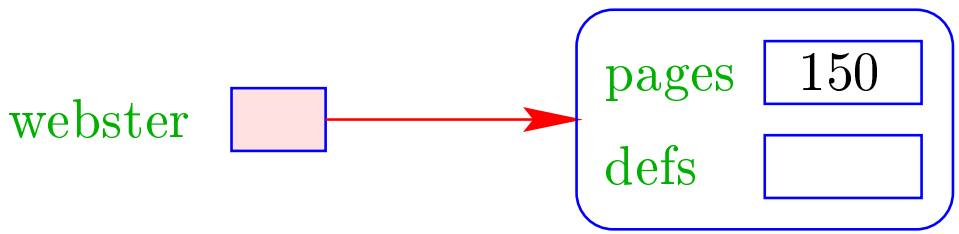
```
public class Dictionary extends Book {
   private int defs;
   public Dictionary(int x) {
      pages = 2*pages;
      defs = x;
      }
   public void defs_message() {
      System.out.print("Number of defs:\t\t"+defs+"\n");
      System.out.print("Defs per page:\t\t"+defs/pages+"\n");
      }
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

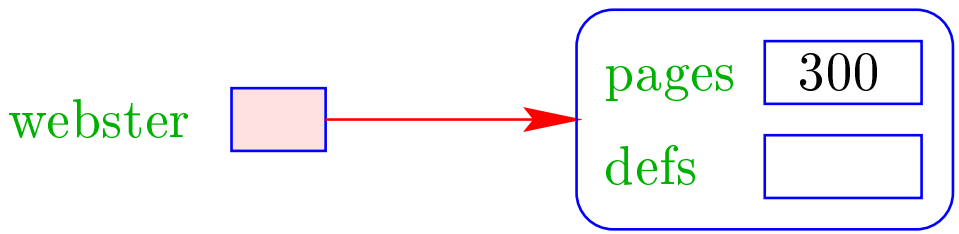
- `class` A `extends` B `{ ... }` deklariert die Klasse A als Unterklasse der Klasse B.
- Alle Members von B stehen damit automatisch auch der Klasse A zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse sichtbar.
- Als `private` deklarierte Members können dagegen in 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.

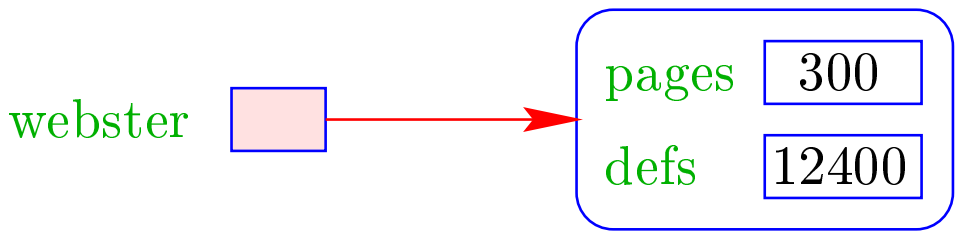
`Dictionary webster = new Dictionary(12400);` liefert:

webster 


webster
pages
defs


webster
pages
150
defs


webster
pages
300
defs


webster
pages
300
defs
12400

```
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
```

11.1 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.

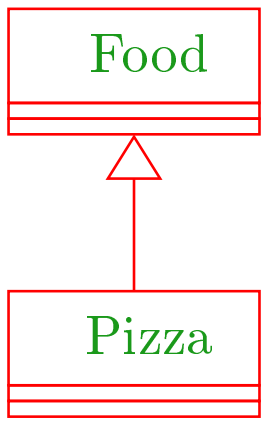
```
public class Words {
   public static void main(String[] args) {
      Dictionary webster = new Dictionary(540,36600);
      webster.message();
   } // end of main
} // end of class Words
```

- 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
```

11.2 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.

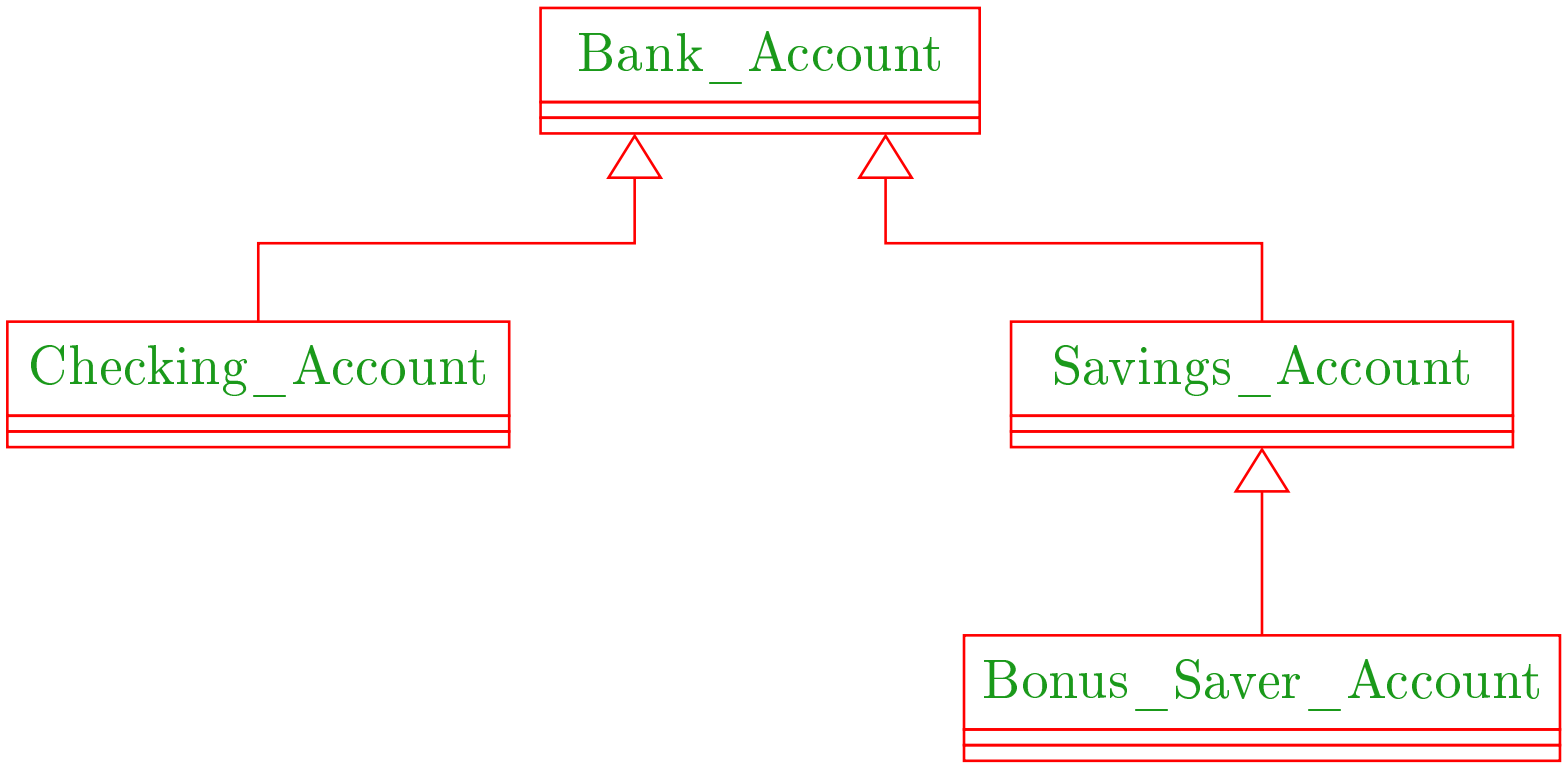
```
public class Pizza extends Food {
   public Pizza (int amount_fat) {
      super (amount_fat,8);
      }
   } // end of class Pizza
```

- 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`

11.3 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
```

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
```

12 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: Auswertung von Ausdrücken

```
public abstract class Expression {
   private int value;
   private boolean evaluated = false;
   public int getValue() {
      if (evaluated) return value;
      else {
         value = evaluate();
         evaluated = true;
         return value;
      }
   }
   abstract protected int evaluate();
} // end of class Expression
```

- Die Unterklassen von `Expression` repräsentieren die verschiedenen Arten von Ausdrücken.
- Allen Unterklassen gemeinsam ist eine Objekt-Methode `evaluate()` – immer mit einer anderen Implementierung.

- 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: `int getValue()`.

- Die Methode `evaluate()` soll den Ausdruck auswerten.
- Die Methode `getValue()` speichert das Ergebnis in dem Attribut `value` ab und vermerkt, dass der Ausdruck bereits ausgewertet wurde.

Beispiel für einen Ausdruck:

```
public final class Const extends Expression {
   private int n;
   public Const(int x) { n=x; }
   protected int evaluate() {
      return n;
   }  // end of evaluate()
}     // end of class Const
```

- Der Ausdruck 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 $\Longrightarrow$ Konstanten.

... andere Ausdrücke:

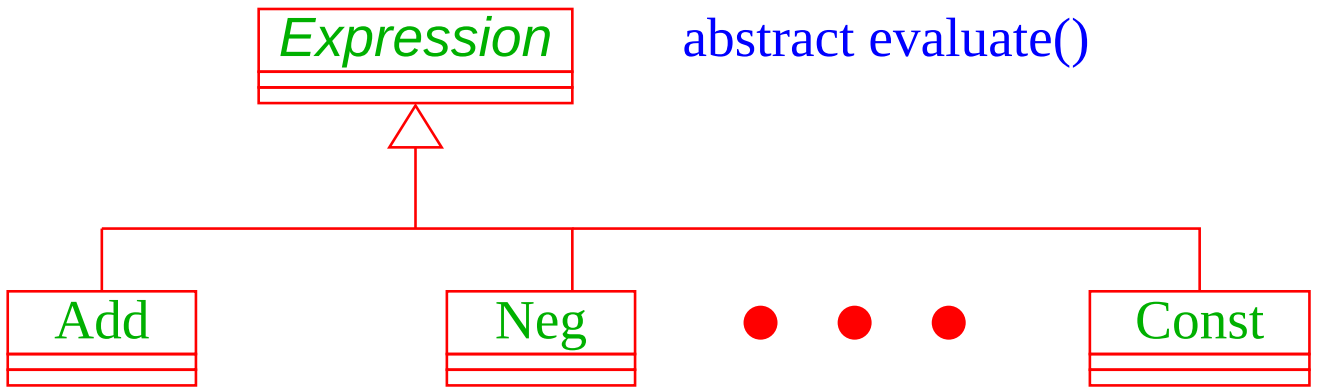
```
public final class Add extends Expression {
   private Expression left, right;
   public Add(Expression l, Expression r) {
      left = l; right = r;
   }
   protected int evaluate() {
      return left.getValue() + right.getValue();
   }  // end of evaluate()
}     // end of class Add
public final class Neg extends Expression {
   private Expression arg;
   public Neg(Expression a) { arg = a; }
   protected int evaluate() { return -arg.getValue(); }
}     // end of class Neg
```

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

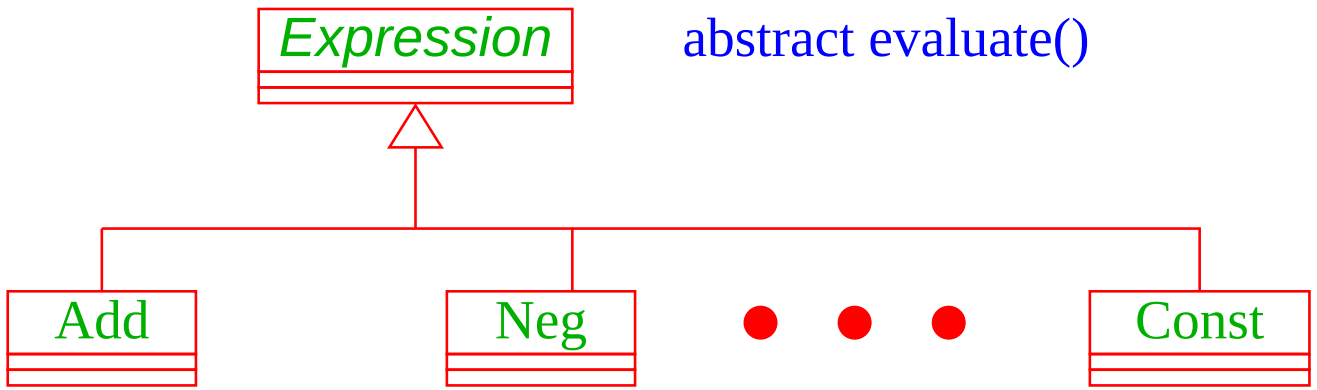
```
public static void main(String[] args) {
   Expression e = new Add (
      new Neg (new Const(8)),
      new Const(16));
   System.out.println(e.getValue())
   }
```

- Die Methode `getValue()` ruft eine Methode `evaluate()` sukzessive für jeden Teilausdruck von `e` auf.
- Welche konkrete Implementierung dieser Methode dabei jeweils gewählt wird, hängt von der konkreten Klasse des jeweiligen Teilausdrucks ab, d.h. entscheidet sich erst zur Laufzeit.
- Das nennt man auch dynamische Bindung.

Die abstrakte Klasse *Expression*:

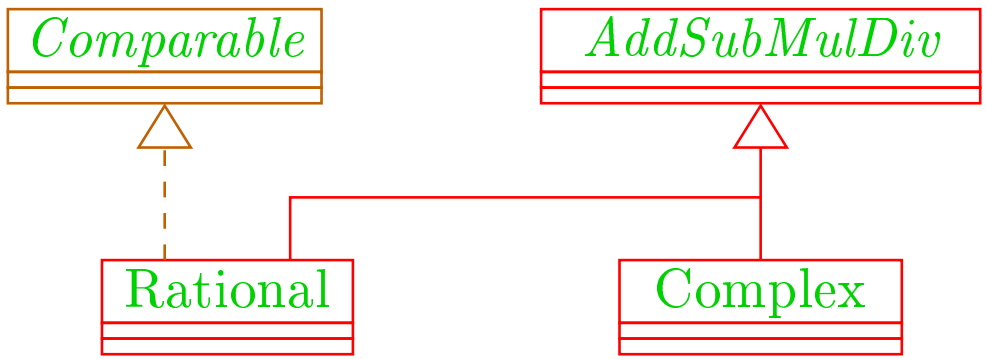


Die abstrakte Klasse *Expression*:



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

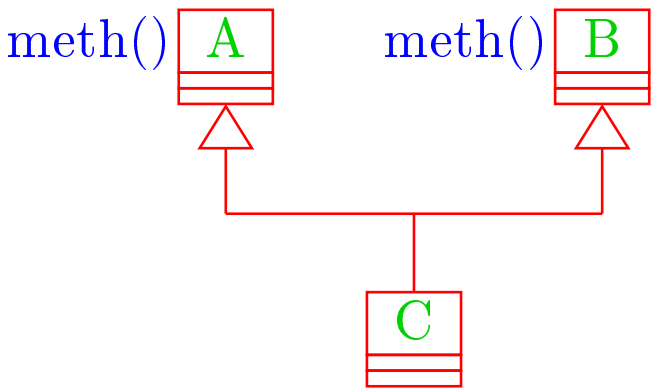
Beispiel:



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 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.

- 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).

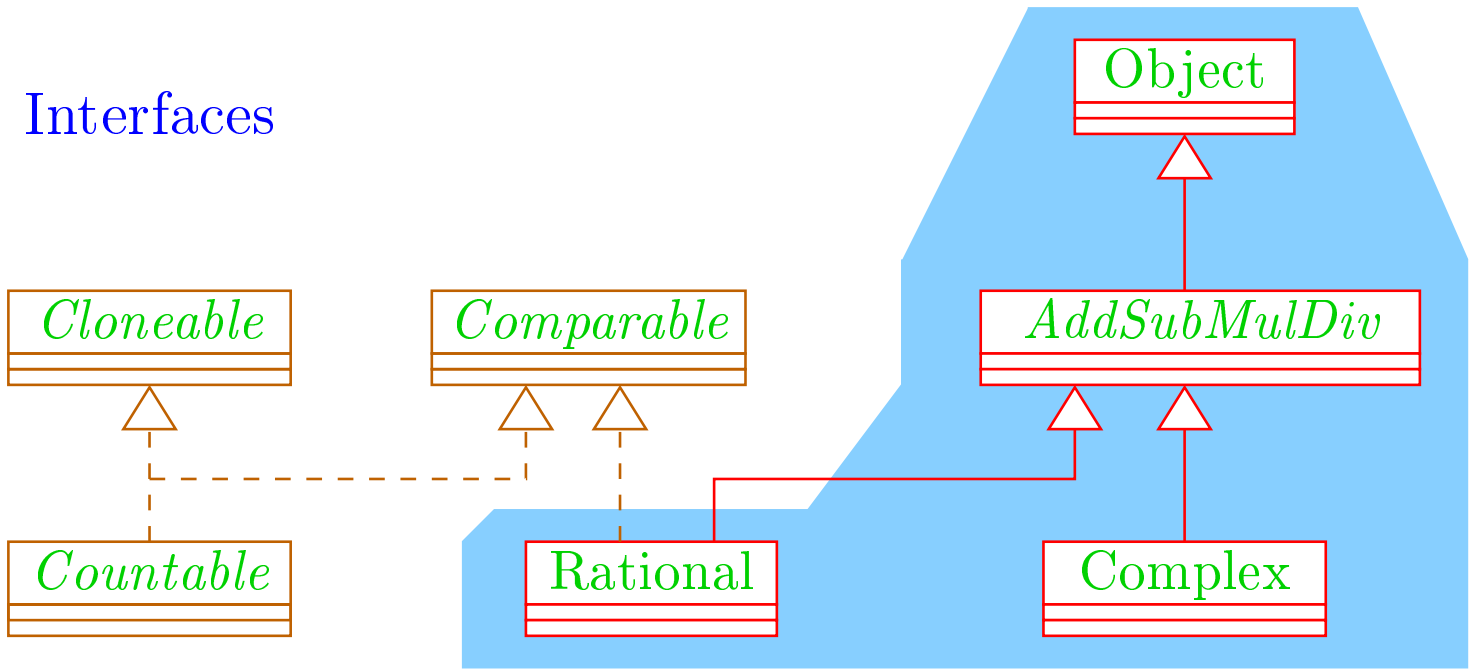
- Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- Erweiternde Interfaces können Konstanten umdefinieren...
- Kommt eine Konstante gleichen Namens const in verschiedenen implementierten Interfaces A und B vor, kann man sie durch A.const und B.const unterscheiden.

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:


Interfaces
Object
Cloneable
Comparable
AddSubMulDiv
Countable
Rational
Complex
Klassen-Hierarchie

13 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.

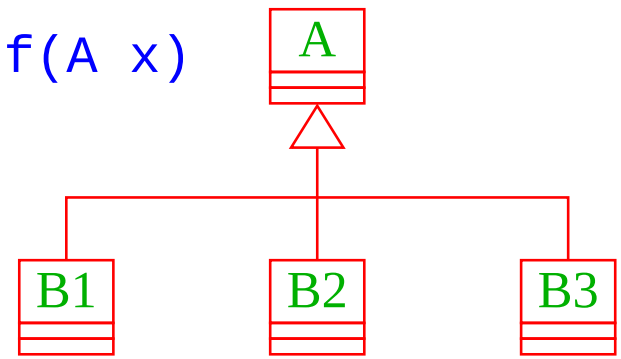
13.1 Unterklassen-Polymorphie

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:

... 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.

Betrachte einen Aufruf $e_0.f(e_1, \ldots, e_k)$.

Ermittlung der aufgerufenen Methode:

- Bestimme die statischen Typen $T_0, \ldots, T_k$ der Ausdrücke $e_0, \ldots, e_k$.
- Suche in einer Oberklasse von T_0 nach einer Methode mit Namen f, deren Liste von Argumenttypen bestmöglich zu der Liste $T_1, \ldots, T_k$ passt.

  Der Typ I dieser rein statisch gefundenen Methode ist die Signatur der Methode f an dieser Aufrufstelle im Programm.
- Der dynamische Typ D des Objekts, zu dem sich e_0 auswertet, gehört zu einer Unterklasse von T_0.
- Die Methode f wird nun aufgerufen, deren Typ I ist und die in der nächsten Oberklasse von D implementiert wird.

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 ???

13.2 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
```

```
   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, weil ihre Werte durch den Aufruf eines Konstruktors festgelet 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; }
}
```

Bemerkungen (Forts.):

- Typparameter können auch lokal für eine Methode eingesetzt werden. Eine Deklaration:

```
public static <T> List<T> create () {
   return new List<T>();
}
```

könnte in jeder anderen Klasse stehen. Der Typparameter `T` wird dann an der Aufrufstelle festgelegt. Der Aufruf

```
List<String> list = create ();
```

instantiiert z.B. den Parameter `T` mit `String`.

13.3 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.

13.3 Wrapper-Klassen

... bleibt ein Problem:

- Der Datentyp `String` ist eine Klasse;
- Felder sind Klassen; aber
- Basistypen wie `int`, `boolean`, `double` sind keine Klassen!

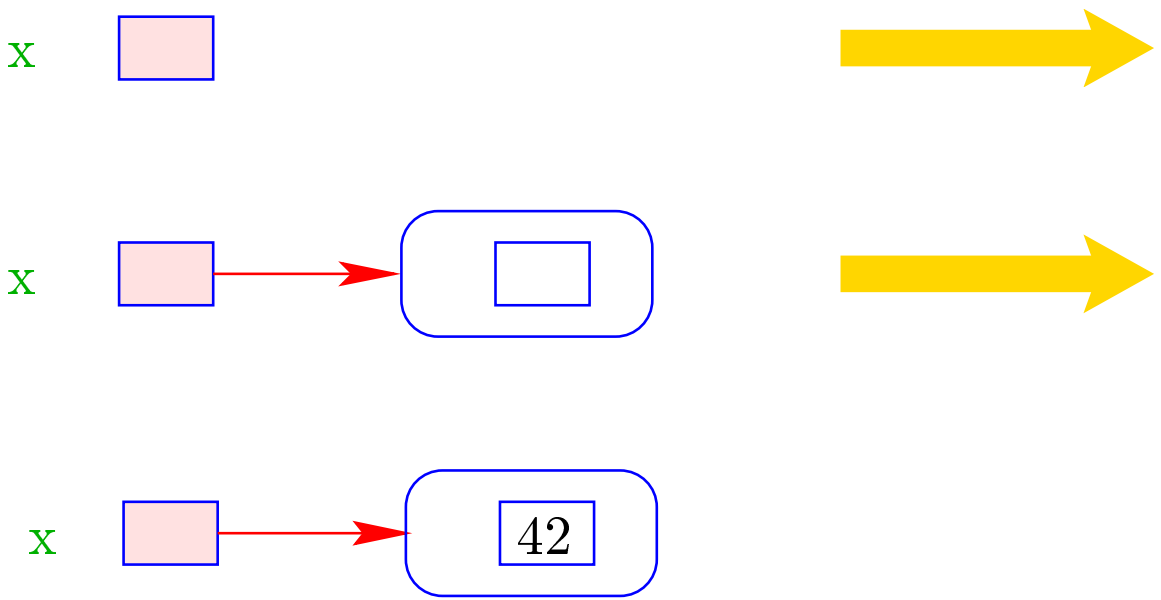
  (Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl.

Ausweg:

- Wickle die Werte eines Basis-Typs in ein Objekt ein!

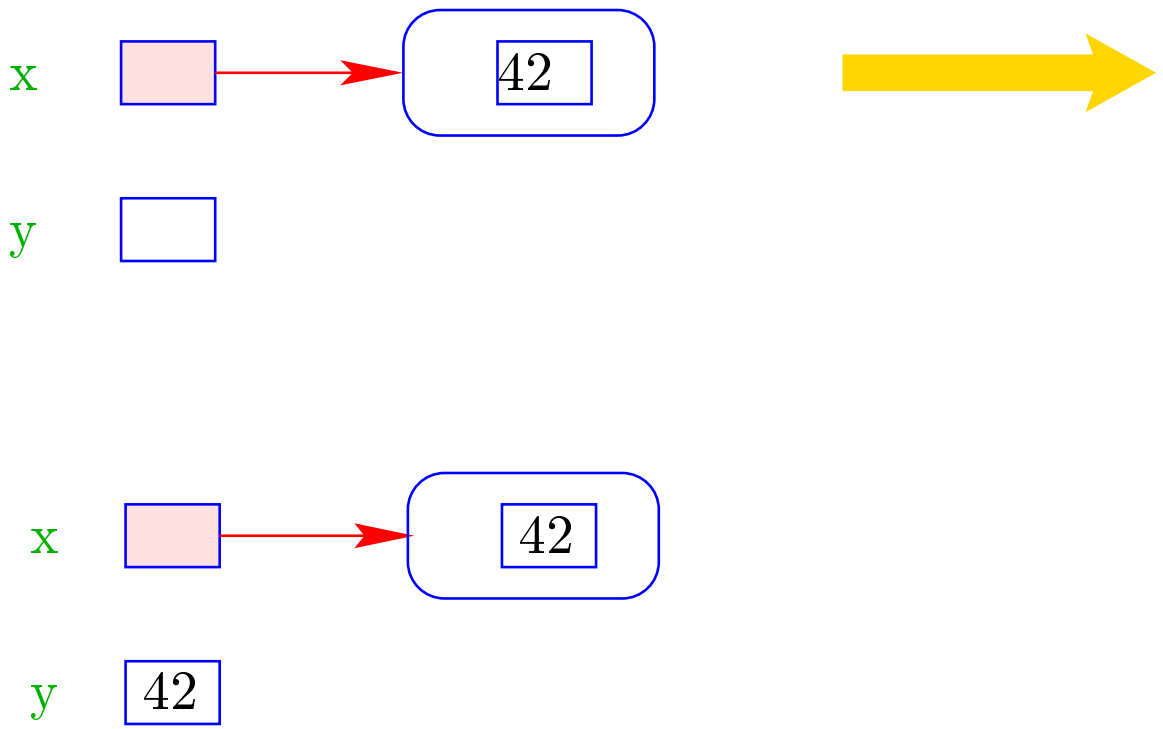
  $\Longrightarrow$ Wrapper-Objekte aus Wrapper-Klassen.

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

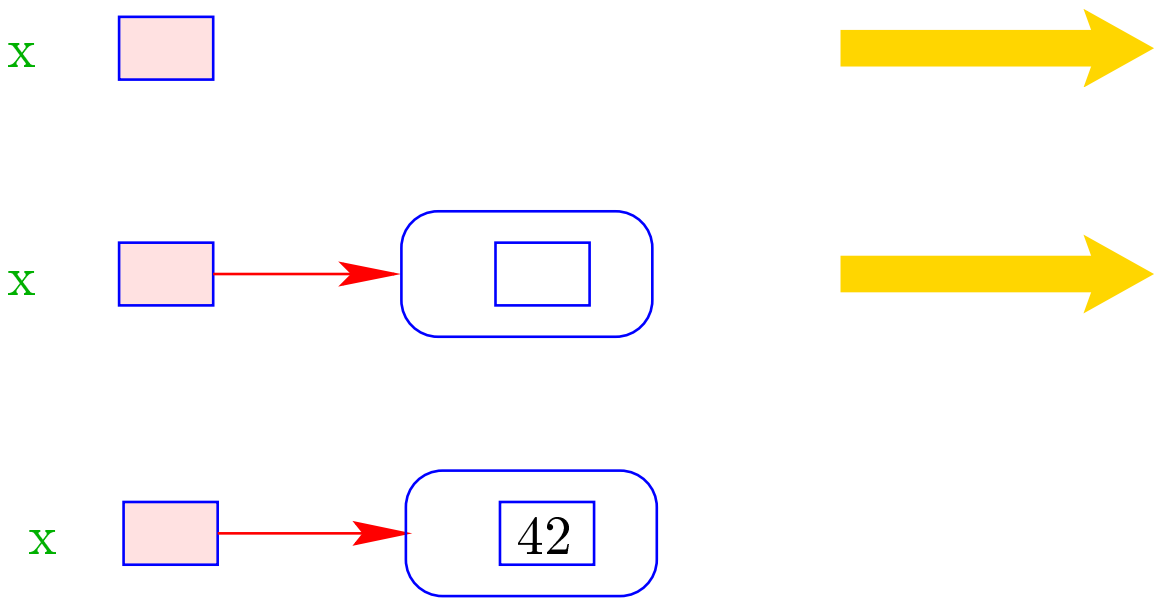


Eingewickelte Werte können auch wieder ausgewickelt werden.

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:

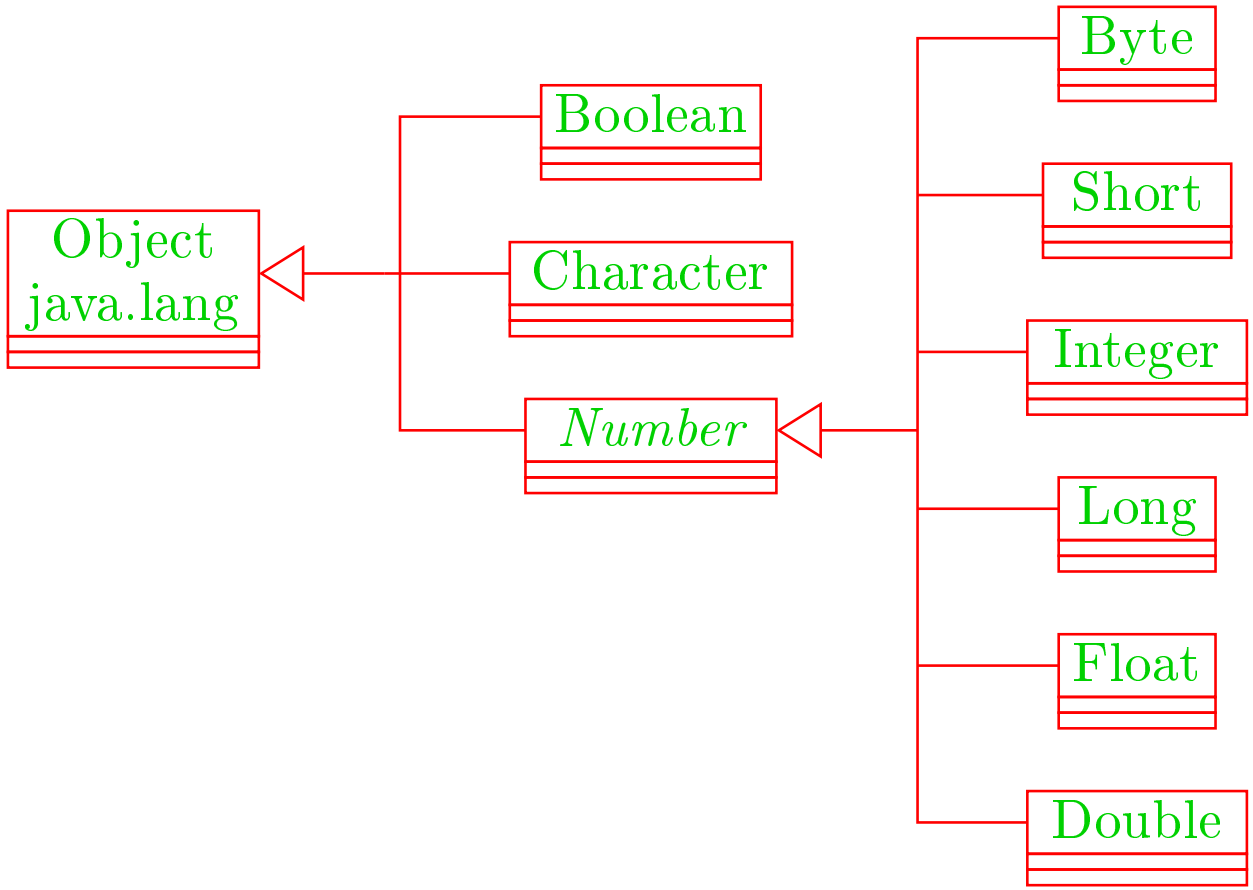
- Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- `public boolean equals(Object obj);` liefert `true` genau dann wenn `obj` den gleichen `int`-Wert enthält.

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:

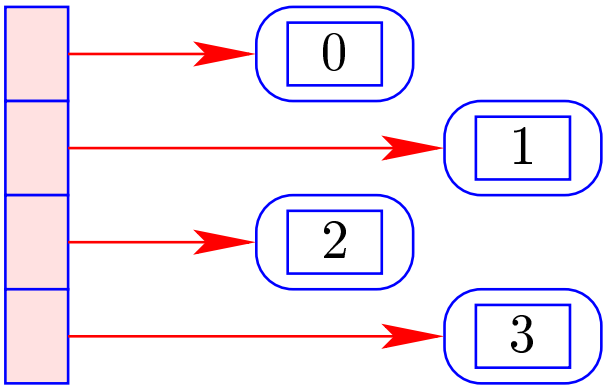
- `Double` und `Float` enthalten zusätzlich die Konstanten

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

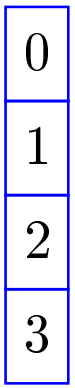
- Zusätzlich gibt es die Tests
  - ```
    public static boolean isInfinite(double v);
    public static boolean isNaN(double v);
    ```
    (analog für `float`)
  - ```
    public boolean isInfinite();
    public boolean isNaN();
    ```

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

Vergleich `Integer` mit `int`:



`Integer []`



`int []`

\+ `Integer`s 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 $\Longrightarrow$ schlechteres Cache-Verhalten.

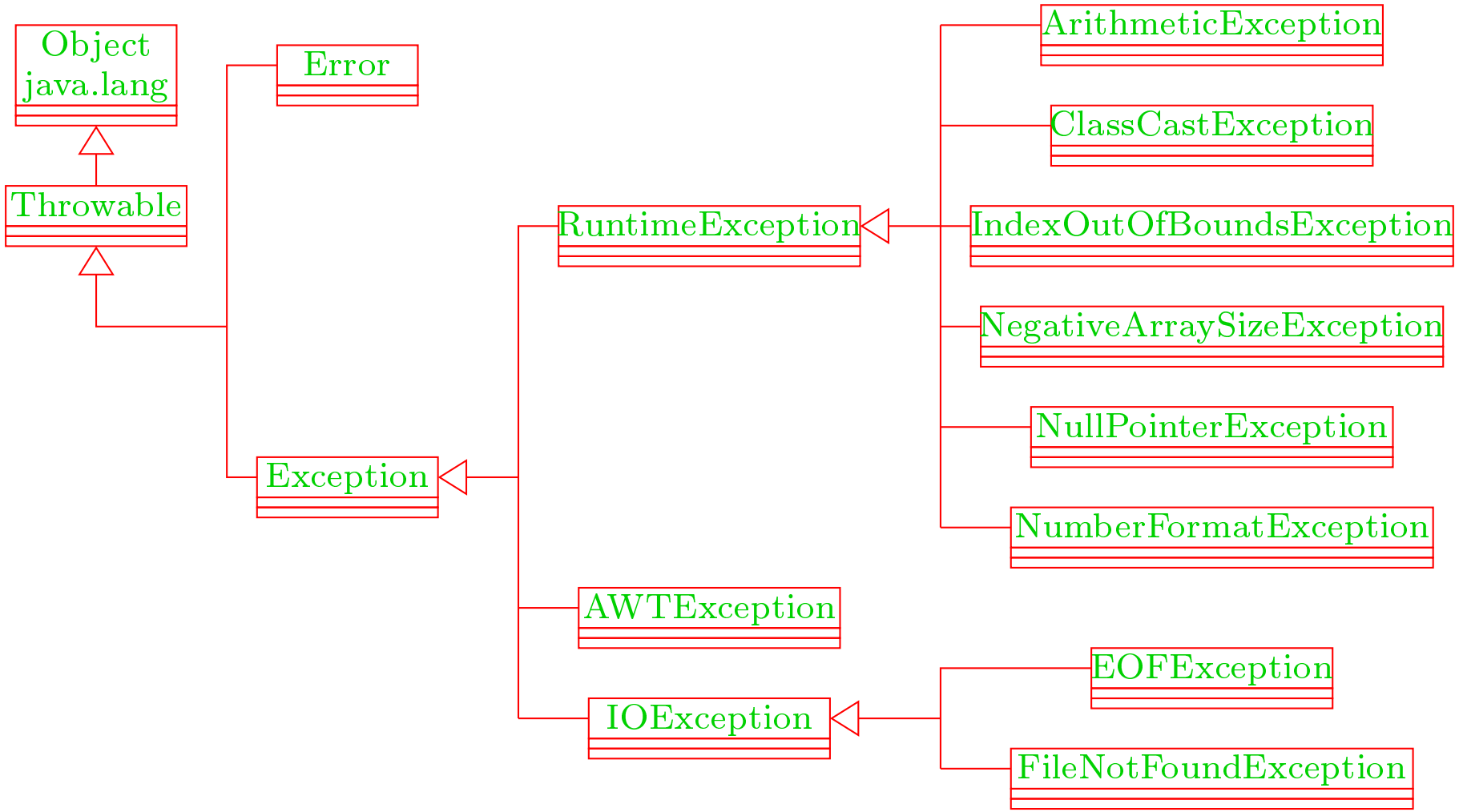
14 Fehler-Objekte: Werfen, Fangen, Behandeln

- Tritt während der Programm-Ausführung ein Fehler auf, wird die normale Programm-ausführung abgebrochen und ein Fehler-Objekt erzeugt (geworfen).
- Die Klasse `Throwable` fasst alle Arten von Fehlern zusammen.
- Ein Fehler-Objekt kann gefangen und geeignet behandelt werden.

Idee: Explizite Trennung von

- normalem Programm-Ablauf (der effizient und übersichtlich sein sollte); und
- Behandlung von Sonderfällen (wie illegalen Eingaben, falscher Benutzung, Sicherheitsattacken, ...)

Einige der vordefinierten Fehler-Klassen:



Die direkten Unterklassen von `Throwable` sind:

- `Error` – für fatale Fehler, die zur Beendigung des gesamten Programms führen, und
- `Exception` – für bewältigbare Fehler oder Ausnahmen.

Ausnahmen der Klasse `Exception`, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode aufgelistet werden !!!

Achtung:

- Die Unterklasse `RuntimeException` der Klasse `Exception` fasst die bei normaler Programm-Ausführung evt. auftretenden Ausnahmen zusammen.
- Eine `RuntimeException` kann jederzeit auftreten ...
- Sie braucht darum nicht im Kopf von Methoden deklariert zu werden.
- Sie kann, muss aber nicht abgefangen werden.

Achtung:

- Die Unterklasse `RuntimeException` der Klasse `Exception` fasst die bei normaler Programm-Ausführung evt. auftretenden Ausnahmen zusammen.
- Eine `RuntimeException` kann jederzeit auftreten ...
- Sie braucht darum nicht im Kopf von Methoden deklariert zu werden.
- Sie kann, muss aber nicht abgefangen werden.

Arten der Fehler-Behandlung:

- Ignorieren;
- Abfangen und Behandeln dort, wo sie entstehen;
- Abfangen und Behandeln an einer anderen Stelle.

Tritt ein Fehler auf und wird nicht behandelt, bricht die Programm-Ausführung ab.

Beispiel:

```
public class Zero {
   public static main(String[] args) {
      int x = 10;
      int y = 0;
      System.out.println(x/y);
   }  // end of main()
}     // end of class Zero
```

Das Programm bricht wegen Division durch `(int)0` ab und liefert die Fehler-Meldung:

```
Exception in  thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Compiled Code)
```

Die Fehlermeldung besteht aus drei Teilen:

1. der ↑Thread, in dem der Fehler auftrat;
2. `System.err.println(toString());` d.h. dem Namen der Fehlerklasse, gefolgt von einer Fehlermeldung, die die Objekt-Methode `getMessage()` liefert, hier: "`/ by zero`".
3. `printStackTrace(System.err);` d.h. der Funktion, in der der Fehler auftrat, genauer: der Angabe sämtlicher Aufrufe im Rekursions-Stack.

Soll die Programm-Ausführung nicht beendet werden, muss der Fehler abgefangen werden.

Beispiel: `NumberFormatException`

```
public class Adding extends MiniJava {
   public static void main(String[] args) {
      int x = getInt("1. Zahl:\t");
      int y = getInt("2. Zahl:\t");
      write("Summe:\t\t"+ (x+y));
   }  // end of main()
   public static int getInt(String str) {
      ...
```

- Das Programm liest zwei `int`-Werte ein und addiert sie.
- Bei der Eingabe können möglicherweise Fehler auftreten:
  - ... weil keine syntaktisch korrekte Zahl eingegeben wird;
  - ... weil sonstige unvorhersehbare Ereignisse eintreffen.
- Die Behandlung dieser Fehler ist in der Funktion `getInt()` verborgen ...

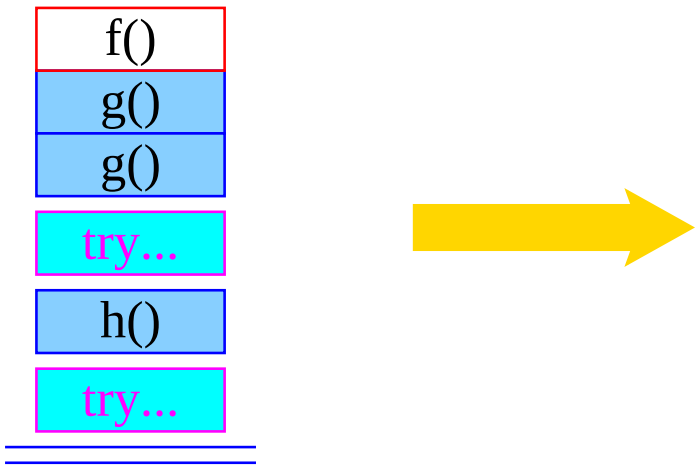
```
        String s;
        while (true) {
           try {
              s = readString(str);
              return Integer.parseInt(s);
           } catch (NumberFormatException e) {
              System.out.println("Falsche Eingabe! ...");
           } catch (IOException e) {
              System.out.println("Eingabeproblem: Ende ...");
              System.exit(0);
           }
        }  // end of while
     }     // end of getInt()
}          // end of class Adding
```

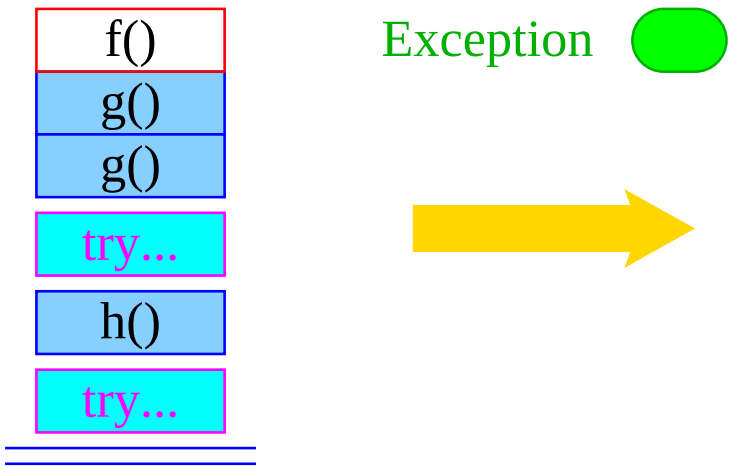
... ermöglicht folgenden Dialog:

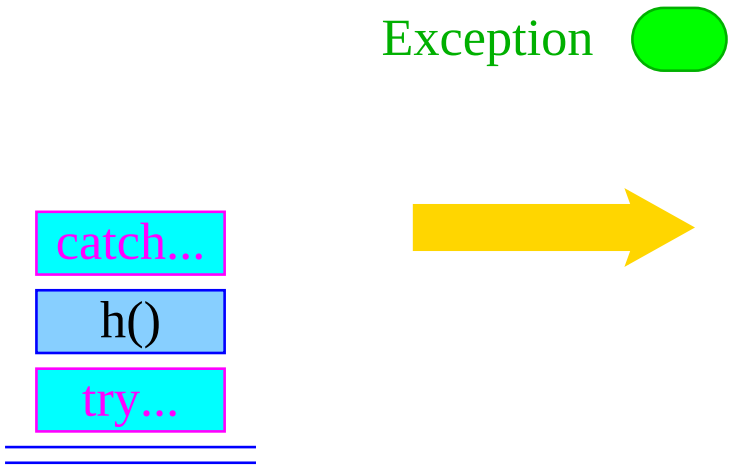
```
> java Adding
1. Zahl:          abc
Falsche Eingabe! ...
1. Zahl:          0.3
Falsche Eingabe! ...
1. Zahl:          17
2. Zahl:          25
Summe:            42
```

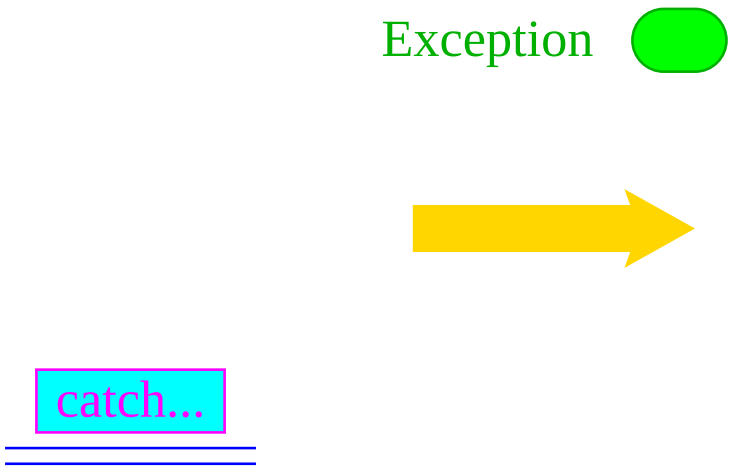
- Ein Exception-Handler besteht aus einem `try`-Block `try {` ss `}` , in dem der Fehler möglicherweise auftritt; gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge ss kein Fehler-Objekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.
- Wird eine Exception ausgelöst, durchsucht der Handler mithilfe des geworfenen Fehler-Objekts sequentiell die `catch`-Regeln.

- Jede `catch`-Regel ist von der Form: `catch (Exc e) {...}` wobei Exc eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehler-Objekt aus (einer Unterklasse) von Exc stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.


f()
g()
g()
try...
h()
try...


f()
g()
g()
try...
h()
try...
Exception


Exception
catch...
h()
try...


Exception
catch...

- Auslösen eines Fehlers verlässt abrupt die aktuelle Berechnung.
- Damit das Programm trotz Auftretens des Fehlers in einem geordneten Zustand bleibt, ist oft Aufräumarbeit erforderlich – z.B. das Schließen von I/O-Strömen.
- Dazu dient `finally { ss }` nach einem `try`-Statement.

- Auslösen eines Fehlers verlässt abrupt die aktuelle Berechnung.
- Damit das Programm trotz Auftretens des Fehlers in einem geordneten Zustand bleibt, ist oft Aufräumarbeit erforderlich – z.B. das Schließen von I/O-Strömen.
- Dazu dient `finally { ss }` nach einem `try`-Statement.

Achtung:

- Die Folge ss von Statements wird auf jeden Fall ausgeführt.
- Wird kein Fehler im `try`-Block geworfen, wird sie im Anschluss an den `try`-Block ausgeführt.
- Wird ein Fehler geworfen und mit einer `catch`-Regel behandelt, wird sie nach dem Block der `catch`-Regel ausgeführt.
- Wird der Fehler von keiner `catch`-Regel behandelt, wird ss ausgeführt, und dann der Fehler weitergereicht.

Beispiel: `NullPointerException`

```
public class Kill {
   public static void kill() {
      Object x = null; x.hashCode ();
   }
   public static void main(String[] args) {
      try { kill();
      } catch (ClassCastException b) {
         System.out.println("Falsche Klasse!!!");
      } finally {
         System.out.println("Leider nix gefangen ...");
      }
   }  // end of main()
}     // end of class Kill
```

... liefert:

```
> java Kill
Leider nix gefangen ...
Exception in thread "main" java.lang.NullPointerException
        at Kill.kill(Compiled Code)
        at Kill.main(Compiled Code)
```

Exceptions können auch

- selbst definiert und
- selbst geworfen werden.

Beispiel:

```
public class Killed extends Exception {
   Killed() {}
   Killed(String s) {super(s);}
} // end of class Killed
public class Kill {
   public static void kill() throws Killed {
      throw new Killed();
   }
   ...
```

```
    public static void main(String[] args) {
       try {
          kill();
       } catch (RuntimeException r) {
          System.out.println("RunTimeException "+ r +"\n");
       } catch (Killed b) {
          System.out.println("Killed It!");
          System.out.println(b);
          System.out.println(b.getMessage());
       }
    } // end of main
}    // end of class Kill
```

- Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden !
- Die Klasse `Exception` verfügt über die Konstruktoren
  `public Exception();` `public Exception(String str);`
  (`str` ist die evt. auszugebende Fehlermeldung).
- `throw` exc löst den Fehler exc aus – sofern sich der Ausdruck exc zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen.
- Ausgabe:

```
Killed It!
Killed
Null
```

Fazit:

- Fehler in Java sind Objekte und können vom Programm selbst behandelt werden.
- `try ... catch ... finally` gestattet, die Fehlerbehandlung deutlich von der normalen Programmausführung zu trennen.
- Die vordefinierten Fehlerarten reichen oft aus.
- Werden spezielle neue Fehler/Ausnahmen benötigt, können diese in einer Vererbungshierarchie organisiert werden.

Warnung:

- Der Fehler-Mechanismus von Java sollte auch nur zur Fehler-Behandlung eingesetzt werden:
  - Installieren eines Handlers ist billig; fangen einer `Exception` dagegen teuer.
  - Ein normaler Programm-Ablauf kann durch eingesetzte `Exceptions` bis zur Undurchsichtigkeit verschleiert werden.
  - Was passiert, wenn `catch`- und `finally`-Regeln selbst wieder Fehler werfen?
- Fehler sollten dort behandelt werden, wo sie auftreten.
- Es ist besser spezifischere Fehler zu fangen als allgemeine – z.B. mit `catch (Exception e) { }`

15 Hashing und die Klasse `String`

- Die Klasse `String` stellt Wörter von (Unicode-) Zeichen dar.
- Objekte dieser Klasse sind stets konstant, d.h. können nicht verändert werden.
- Veränderbare Wörter stellt die Klasse ↑`StringBuffer` zur Verfügung.

Beispiel:

```
String str = "abc";
```

... ist äquivalent zu:

```
char[] data = new char[] {'a', 'b', 'c'};
String str = new String(data);
```

Weitere Beispiele:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc"+cde);
String c = "abc".substring(2,3);
String d = cde.substring(1,2);
```

- Die Klasse `String` stellt Methoden zur Verfügung, um
  - einzelne Zeichen oder Teilfolgen zu untersuchen,
  - Wörter zu vergleichen,
  - neue Kopien von Wörtern zu erzeugen, die etwa nur aus Klein- (oder Groß-) Buchstaben bestehen.
- Für jede Klasse gibt es eine Methode `String toString()`, die eine `String`-Darstellung liefert.
- Der Konkatenations-Operator "`+`" ist mithilfe der Methode `append()` der Klasse `StringBuffer` implementiert.

Einige Konstruktoren:

- `String();`
- `String(char[] value);`
- `String(String s);`
- `String(StringBuffer buffer);`

Einige Objekt-Methoden:

- `char charAt(int index);`
- `int compareTo(String anotherString);`
- `boolean equals(Object obj);`
- `String intern();`
- `int indexOf(int chr);`
- `int indexOf(int chr, int fromIndex);`
- `int lastIndexOf(int chr);`
- `int lastIndexOf(int chr, int fromIndex);`

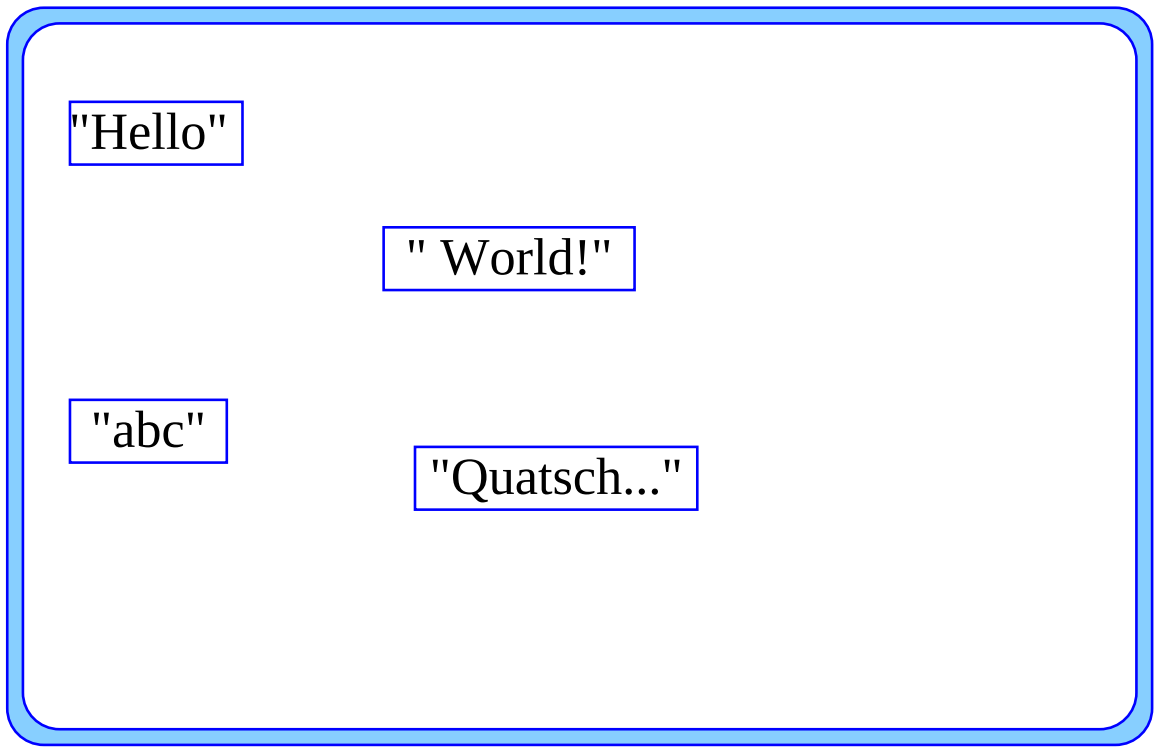
... weitere Objekt-Methoden:

- `int length();`
- `String replace(char oldChar, char newChar);`
- `String substring(int beginIndex);`
- `String substring(int beginIndex, int endIndex);`
- `char[] toCharArray();`
- `String toLowerCase();`
- `String toUpperCase();`
- `String trim();` : beseitigt White Space am Anfang und Ende des Worts.

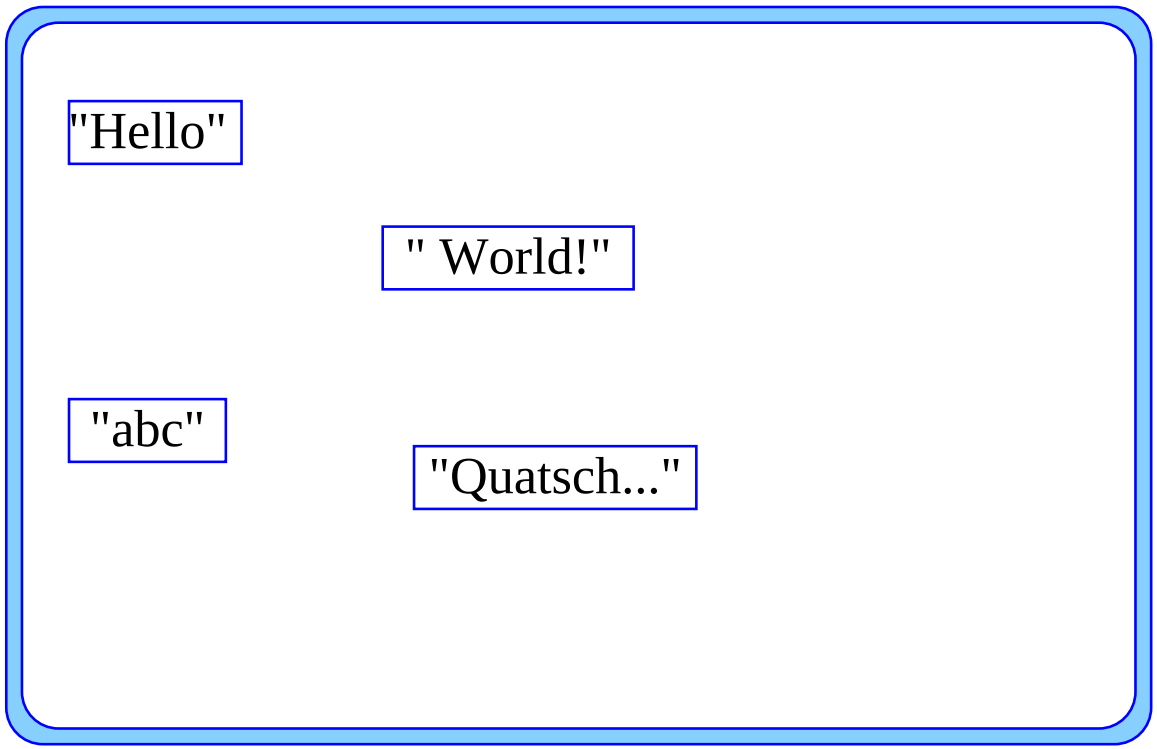
... sowie viele weitere.

Zur Objekt-Methode `intern()`:

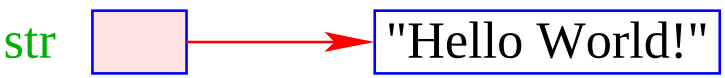
- Die Java-Klasse `String` verwaltet einen privaten String-Pool:

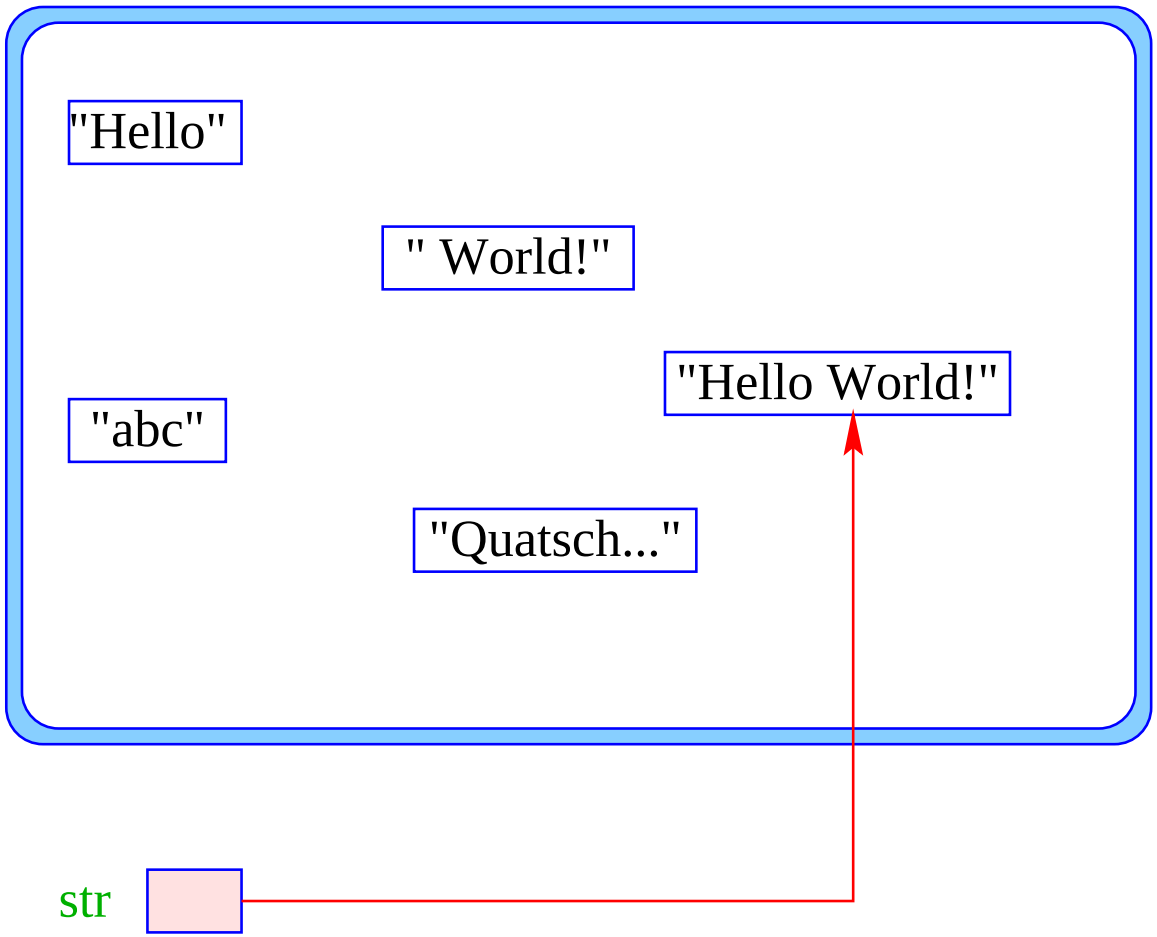


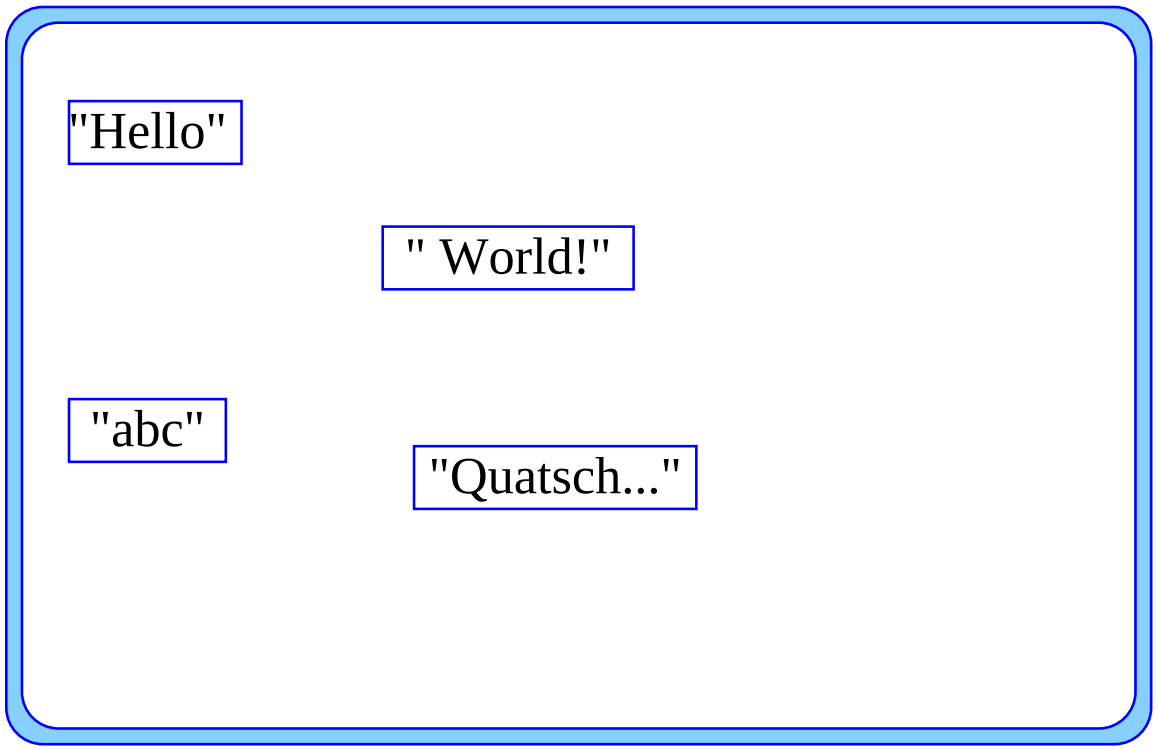
- Alle `String`-Konstanten des Programms werden automatisch in den Pool eingetragen.
- `s.intern();` überprüft, ob die gleiche Zeichenfolge wie `s` bereits im Pool ist.
- Ist dies der Fall, wird ein Verweis auf das Pool-Objekt zurück gegeben.
- Andernfalls wird `s` in den Pool eingetragen und `s` zurück geliefert.



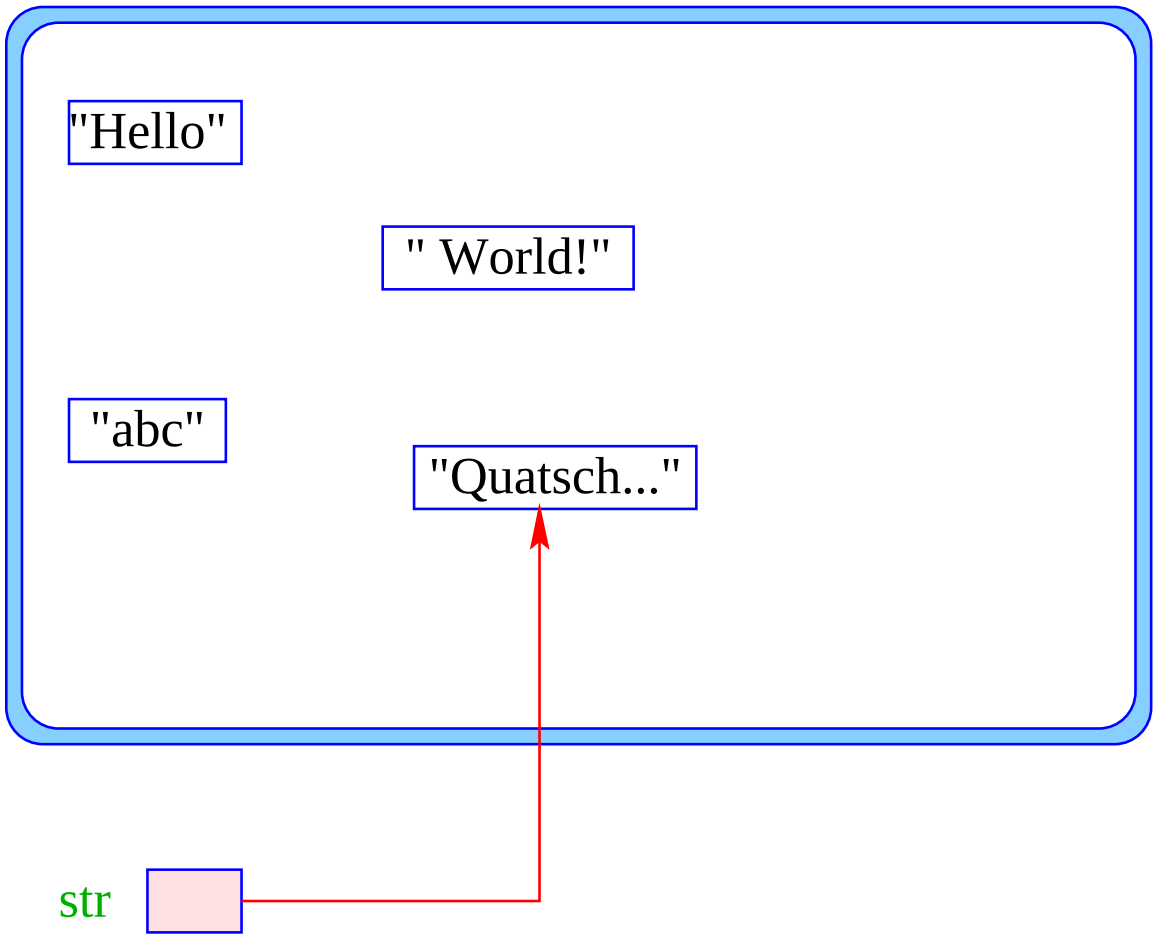

```
str = str.intern();
```


"Hello"
" World!"
"Hello World!"
"abc"
"Quatsch..."
str




```
str = str.intern();
```


"Hello"
" World!"
"abc"
"Quatsch..."
str

Vorteil:

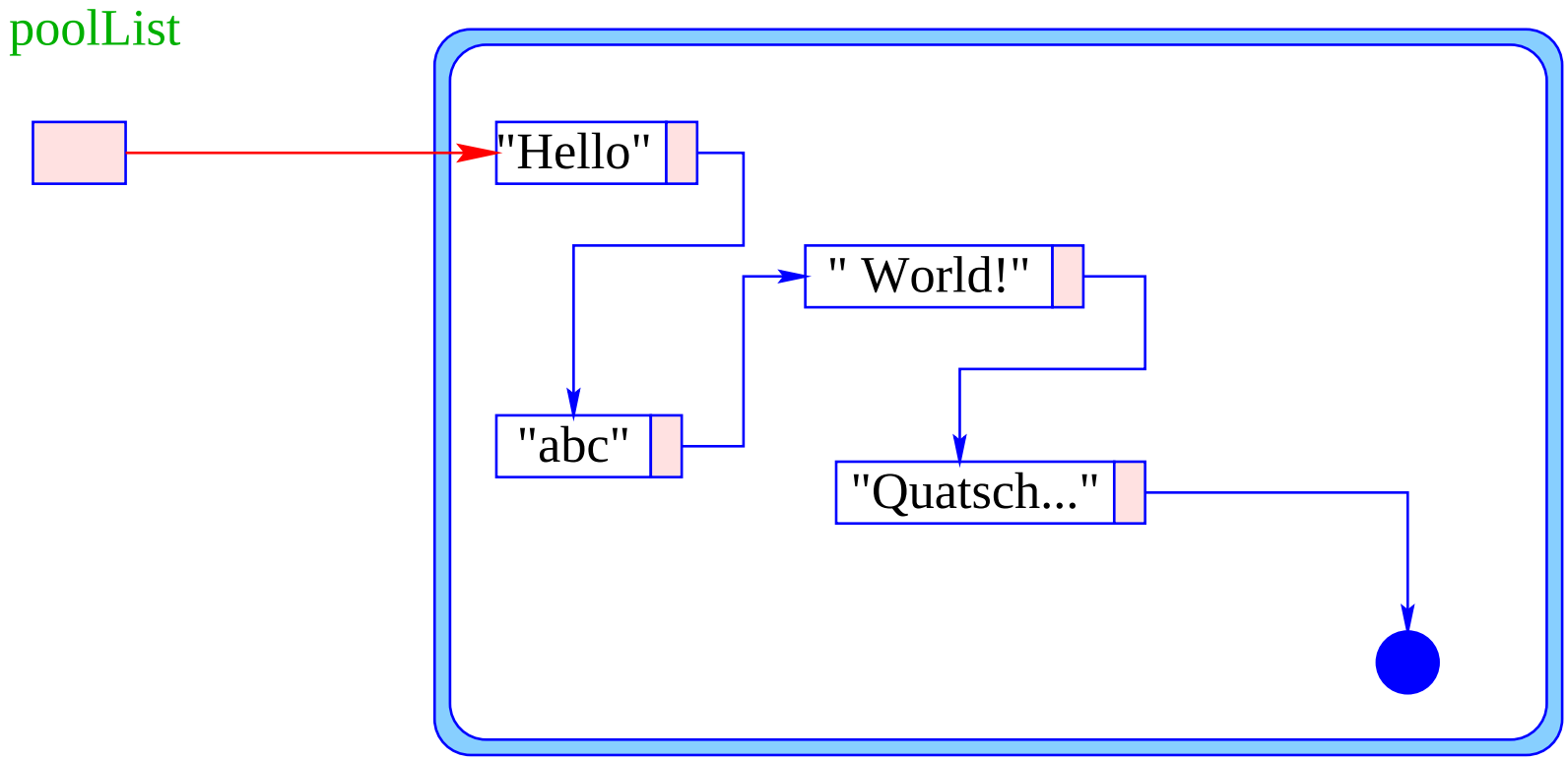
- Internalisierte Wörter existieren nur einmal.
- Test auf Gleichheit reduziert sich zu Test auf Referenz-Gleichheit, d.h. "=="
  $\Longrightarrow$ erheblich effizienter als zeichenweiser Vergleich !!!

... bleibt nur ein Problem:

- Wie findet man heraus, ob ein gleiches Wort im Pool ist ??

1. Idee:

- Verwalte eine Liste der (Verweise auf die) Wörter im Pool;
- implementiere `intern()` als eine `List`-Methode, die die Liste nach dem gesuchten Wort durchsucht.
- Ist das Wort vorhanden, wird ein Verweis darauf zurückgegeben.
- Andernfalls wird das Wort (z.B. vorne) in die Liste eingefügt.


poolList
"Hello"
" World!"
"abc"
"Quatsch..."

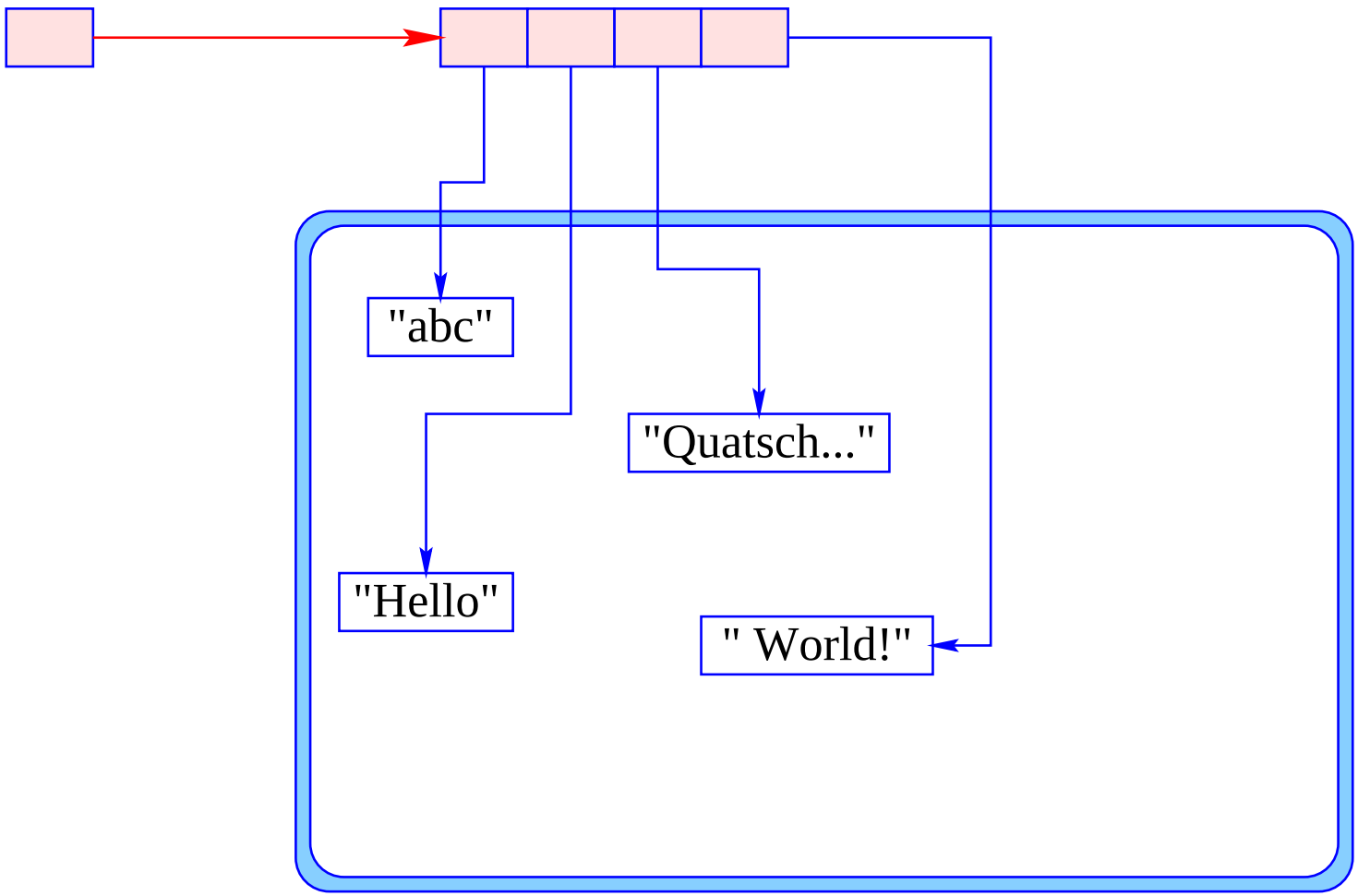
\+ Die Implementierung ist einfach.

\- die Operation `intern()` muss das einzufügende Wort mit jedem Wort im Pool vergleichen $\Longrightarrow$ immens teuer !!!

2. Idee:

- Verwalte ein sortiertes Feld von (Verweisen auf) `String`-Objekte.
- Herausfinden, ob ein Wort bereits im Pool ist, ist dann ganz einfach ...

poolArray



\+ Auffinden eines Worts im Pool ist einfach.

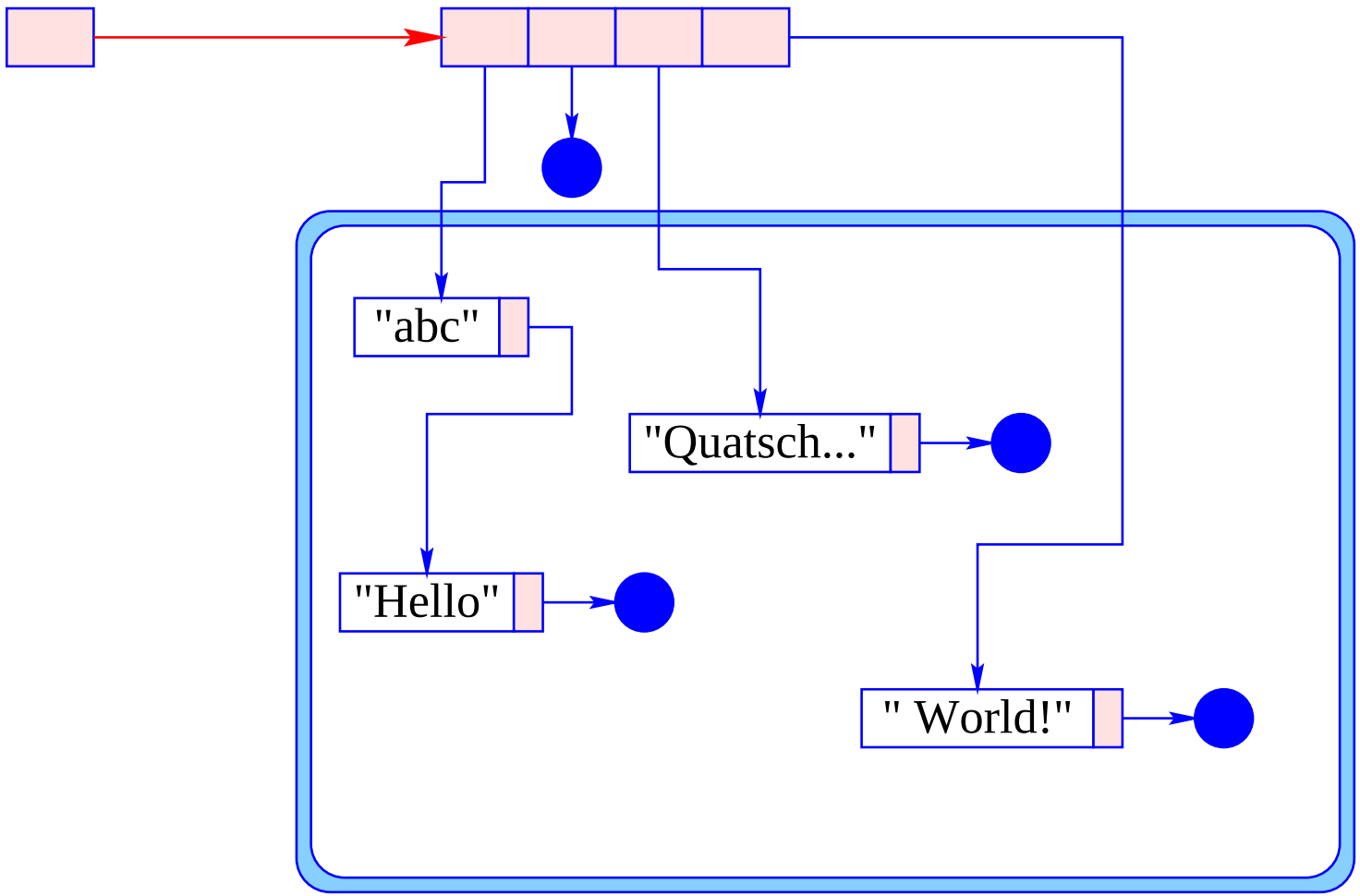
\- Einfügen eines neuen Worts erfordert aber evt. Kopieren aller bereits vorhandenen Verweise ...

$\Longrightarrow$ immer noch sehr teuer !!!

3. Idee: Hashing

- Verwalte nicht eine, sondern viele Listen!
- Verteile die Wörter (ungefähr) gleichmäßig über die Listen.
- Auffinden der richtigen Liste muss schnell möglich sein.
- In der richtigen Liste wird dann sequentiell gesucht.

hashSet



Auffinden der richtigen Liste:

- Benutze eine (leicht zu berechnende) Funktion `hash: String -> int`;
- Eine solche Funktion heißt Hash-Funktion.
- Eine Hash-Funktion ist gut, wenn sie die Wörter (einigermaßen) gleichmäßig verteilt.
- Hat das Feld `hashSet` die Größe m, und gibt es n Wörter im Pool, dann müssen pro Aufruf von `intern();` nur Listen einer Länge ca. n/m durchsucht werden !!!

Sei s das Wort $s_0 s_1 \ldots s_{k-1}$.

Beispiele für Hash-Funktionen:

- $h_0(s) = s_0 + s_{k-1}$;
- $h_1(s) = s_0 + s_1 + \ldots + s_{k-1}$;
- $h_2(s) = (\ldots((s_0 * p) + s_1) * p + \ldots) * p + s_{k-1}$ für eine krumme Zahl p.

(Die `String`-Objekt-Methode `hashCode()` entspricht der Funktion h_2 mit $p = 31$.)

String	h_0	h_1	h_2 $(p = 7)$
alloc	196	523	276109
add	197	297	5553
and	197	307	5623
const	215	551	282083
div	218	323	5753
eq	214	214	820
fjump	214	546	287868
false	203	523	284371
halt	220	425	41297
jump	218	444	42966
less	223	439	42913
leq	221	322	6112
...		...	

String	h_0	h_1	h_2
...		...	
load	208	416	43262
mod	209	320	6218
mul	217	334	6268
neq	223	324	6210
neg	213	314	6200
not	226	337	6283
or	225	225	891
read	214	412	44830
store	216	557	322241
sub	213	330	6552
true	217	448	46294
write	220	555	330879

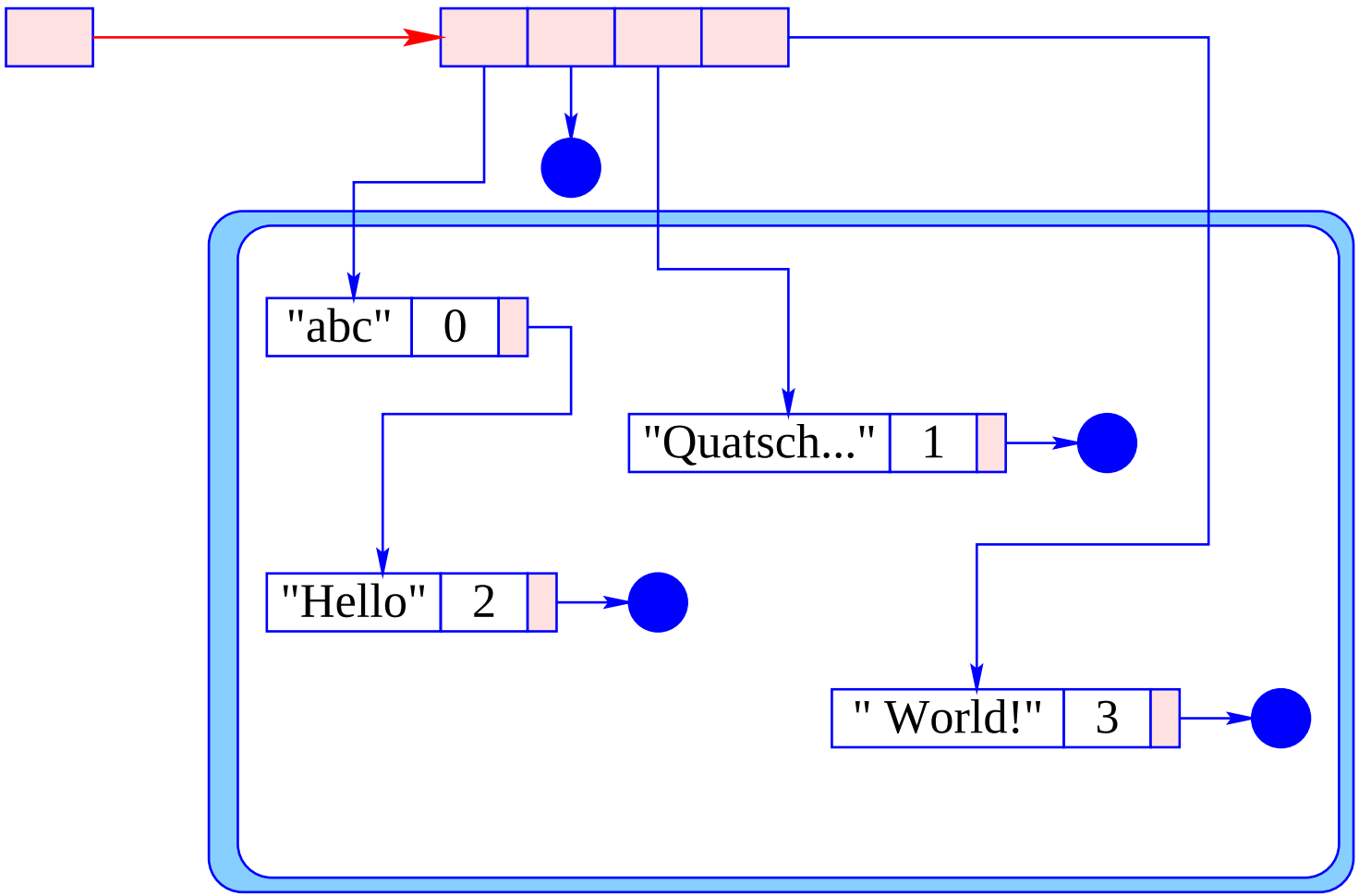
Mögliche Implementierung von `intern()`:

```
public class String {
   private static int n = 1024;
   private static List<String>[] hashSet = new List<String>[n];
   public String intern() {
      int i = (Math.abs(hashCode()%n);
      for (List<String> t=hashSet[i]; t!=null; t=t.next)
         if (equals(t.info)) return t.info;
      hashSet[i] = new List<String>(this, hashSet[i]);
      return this;
   }  // end of intern()
   ...
}     // end of class String
```

- Die Methode `hashCode()` existiert für sämtliche Objekte.
- Folglich können wir (wenn wir Lust haben) ähnliche Pools auch für andere Klassen implementieren.
- Vorsicht! In den Pool eingetragene Objekte können vom Garbage-Collector nicht eingesammelt werden ...
- Statt nur nachzusehen, ob ein Wort `str` (bzw. ein Objekt `obj`) im Pool enthalten ist, könnten wir im Pool auch noch einen Wert hinterlegen

  $\Longrightarrow$ Implementierung von beliebigen Funktionen `String -> type` (bzw. `Object -> type`)

hashTable



Weitere Klassen zur Manipulation von Zeichen-Reihen:

- `StringBuffer` – erlaubt auch destruktive Operationen, z.B. Modifikation einzelner Zeichen, Einfügen, Löschen, Anhängen ...
- `java.util.StringTokenizer` – erlaubt die Aufteilung eines `String`-Objekts in Tokens, d.h. durch Separatoren (typischerweise White-Space) getrennte Zeichen-Teilfolgen.