Informatik 1

Wintersemester 2006/2007

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 :-)

Voraussetzungen:

Informatik Leistungkurs:	nützlich, aber nicht nötig	;-)
Kenntnis einer Programmiersprache:		
	nützlich, aber nicht nötig	:-)
Eigener Rechner:	nützlich, aber nicht nötig	:-)

Voraussetzungen:

Informatik Leistungkurs:	nützlich, aber nicht nötig ;-)
Kenntnis einer Programmiersprache:	
	nützlich, aber nicht nötig :-)
Eigener Rechner:	nützlich, aber nicht nötig :-)
Abstraktes Denken:	unbedingt erforderlich !!!
Neugierde, technisches Interesse:	unbedingt erforderlich !!!

Unsere Philosophie:

Schreiben ist Macht

Eine alte Kulturtechnik:

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

Schreiben als politisches Instrument:

⇒ um zu bestimmen, was Realität ist;

⇒ um das Handeln von Menschen zu beeinflussen ...

(Vorschriften, Gesetze)

Schreiben als praktisches Werkzeug:

$\Rightarrow$ um festzulegen, wie ein Vorgang ablaufen soll:

- Wie soll das Regal zusammen gebaut werden?
- Wie multipliziert man zwei Zahlen?

$\Rightarrow$ um das Verhalten komplexer mit ihrer Umwelt interagierender Systeme zu realisieren

(etwa Motorsteuerungen, Roboter)

$\Longrightarrow$ Programmierung

1 Vom Problem zum Programm

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

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

$\Longrightarrow$

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

Abu Abdallah Muhamed ibn Musa al Chwaritzmi, 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.

Eigenschaften von Algorithmen:

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

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

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

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

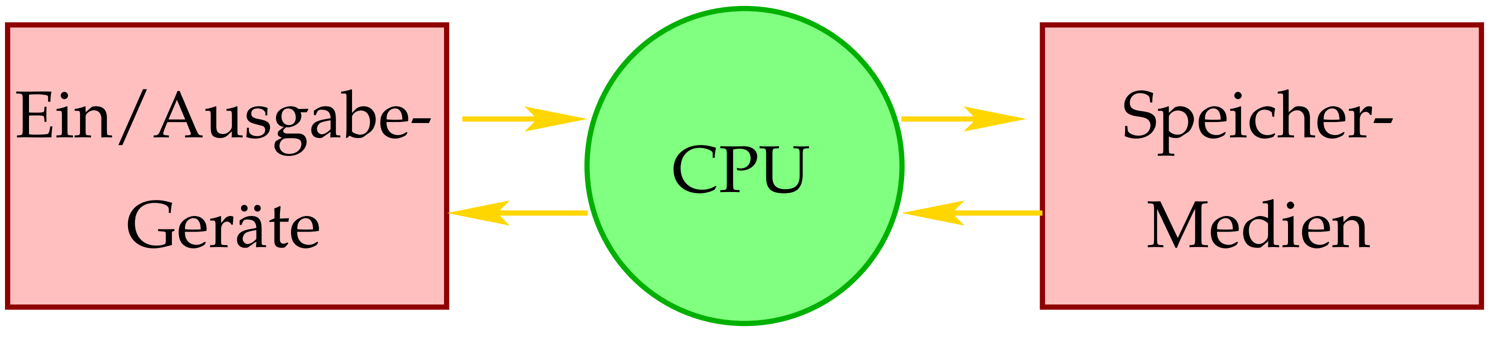
Ein Programm ist die Formulierung eines Algorithmus in einer Programmiersprache.

Die Formulierung gestattet (hoffentlich :-) eine maschinelle 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.

Typischer Aufbau eines Computers:



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

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

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

Hardware == physikalische Bestandteile eines Computers.

Merkmale von Computern:

Geschwindigkeit: schnelle Ausgeführung auch komplexer Programme.

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

Speicherkapazität: riesige Datenmengen speicherbar und schnell zugreifbar.

Kosten: Niedrige laufende Kosten.

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

$\Longrightarrow$ Annahme eines (nicht ganz realistischen, dafür exakt definierten) Maschinenmodells für die Programmierung.

Beliebte Maschinenmodelle:

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

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

λ-Kalkül: eine minimale ↑funktionale Programmiersprache;

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

...

Zur Definition eines Maschinenmodells benötigen wir:

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

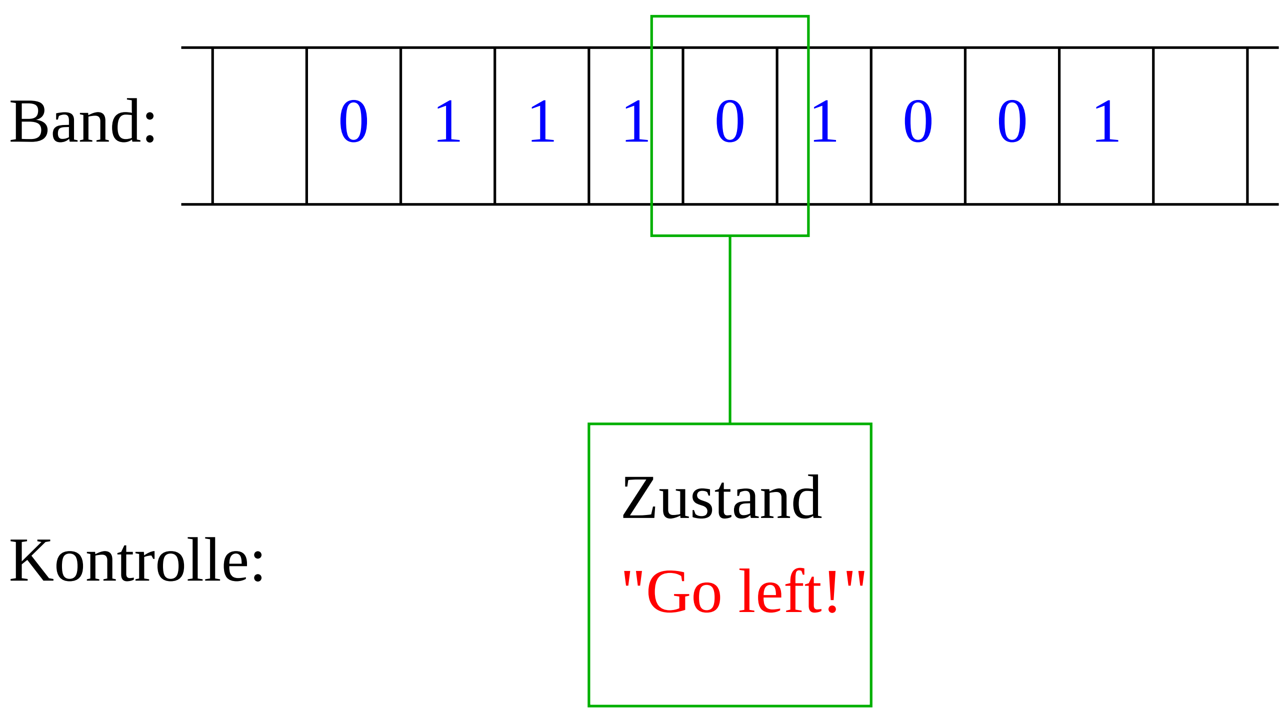
Beispiel 1: Turing-Maschine

Alan Turing, 1912–1954

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

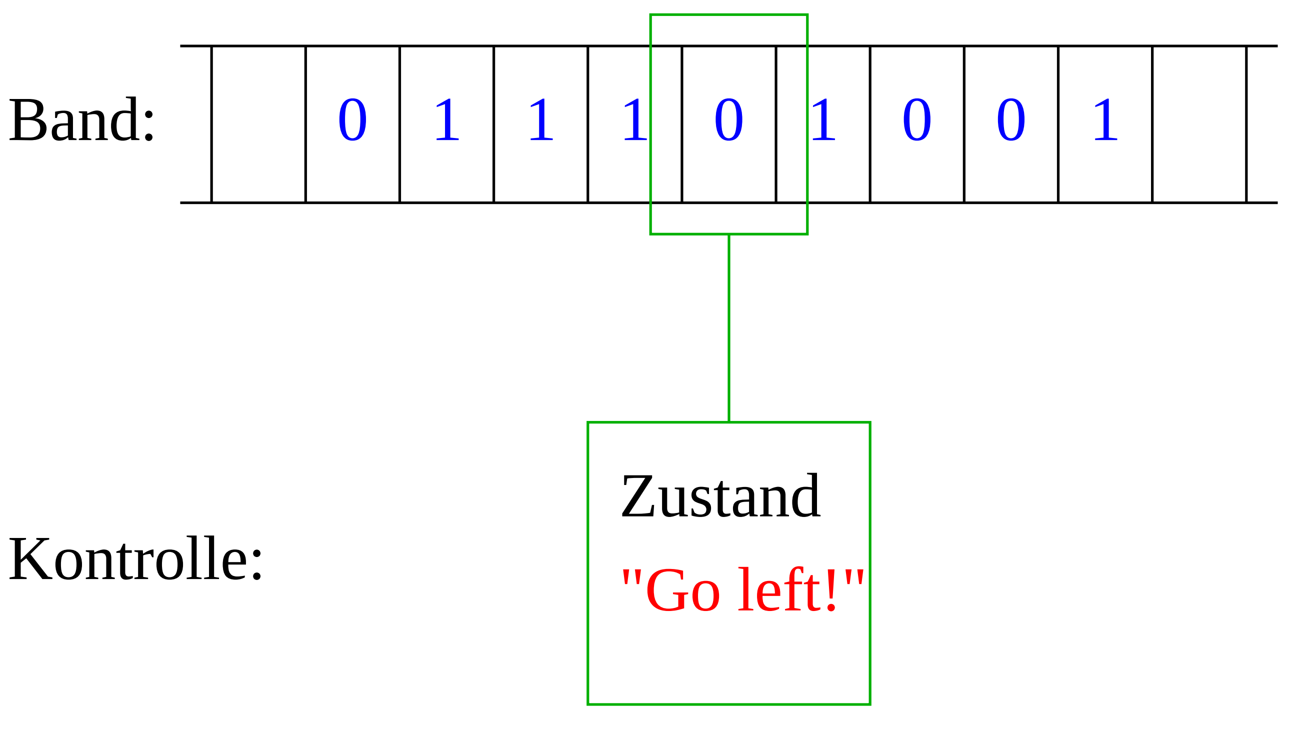
Operationen: Überschreiben des aktuellen Zeichens und Verrücken des Kopfs um eine Position nach rechts oder links;

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



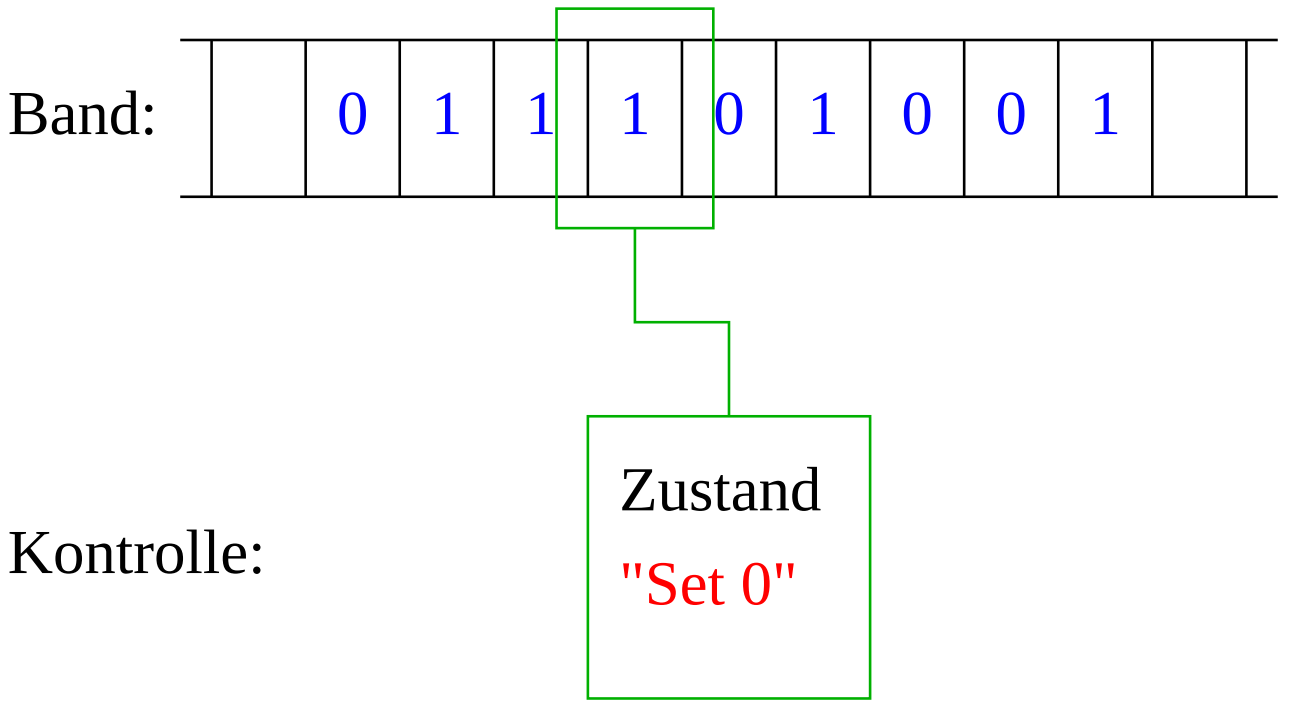
Programm:

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



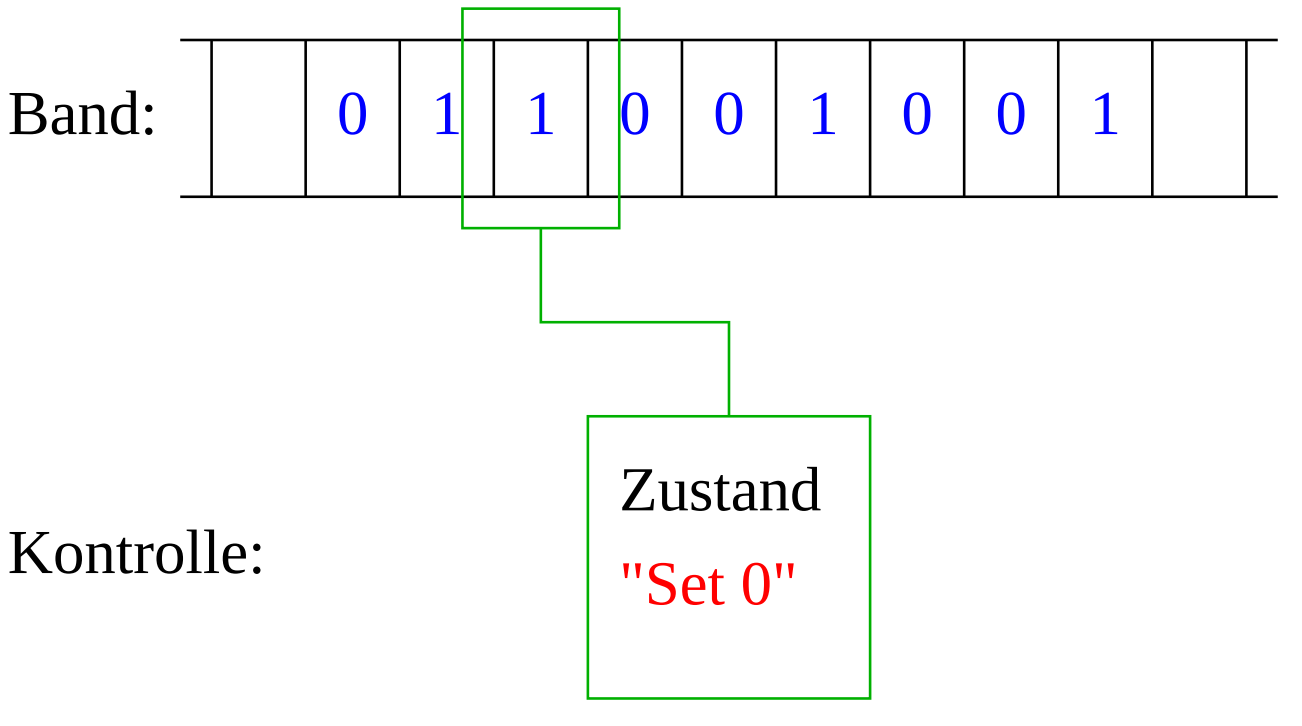
Operation = "Schreibe eine 0 und gehe nach links!"

neuer Zustand = "Set 0"



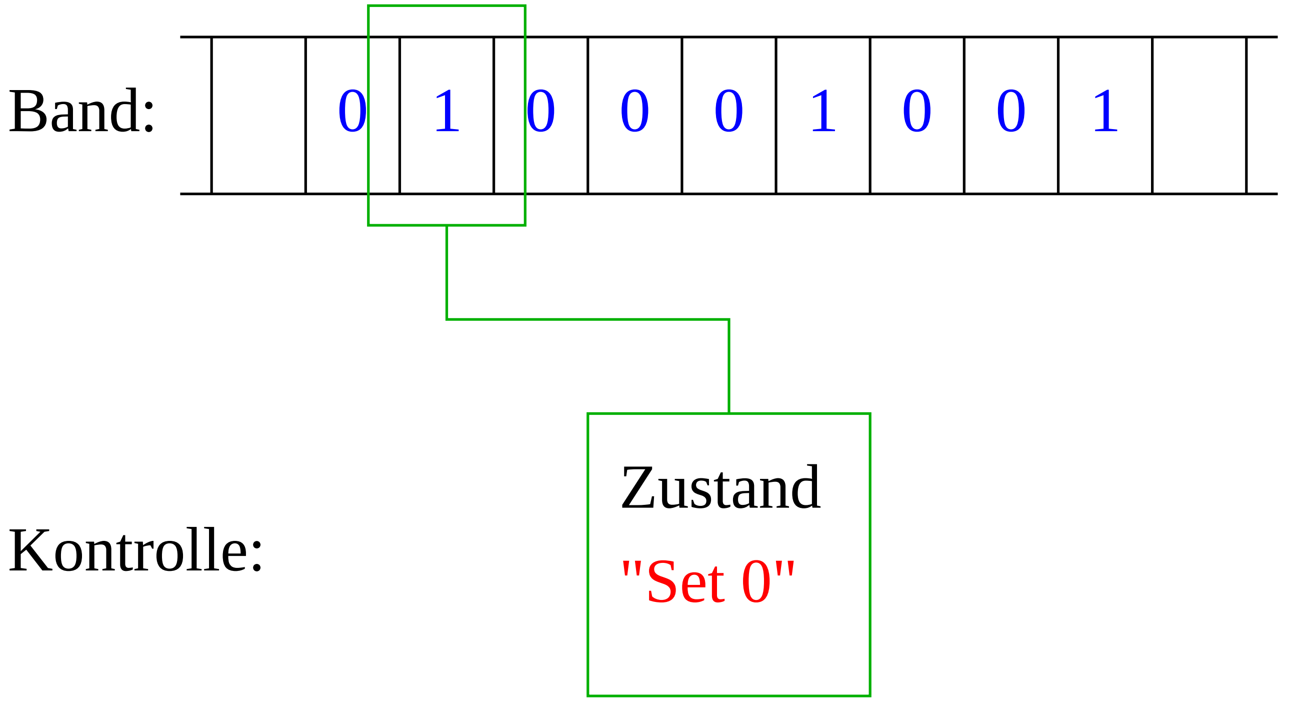
Operation = “Schreibe eine 0 und gehe nach links!”

neuer Zustand = unverändert



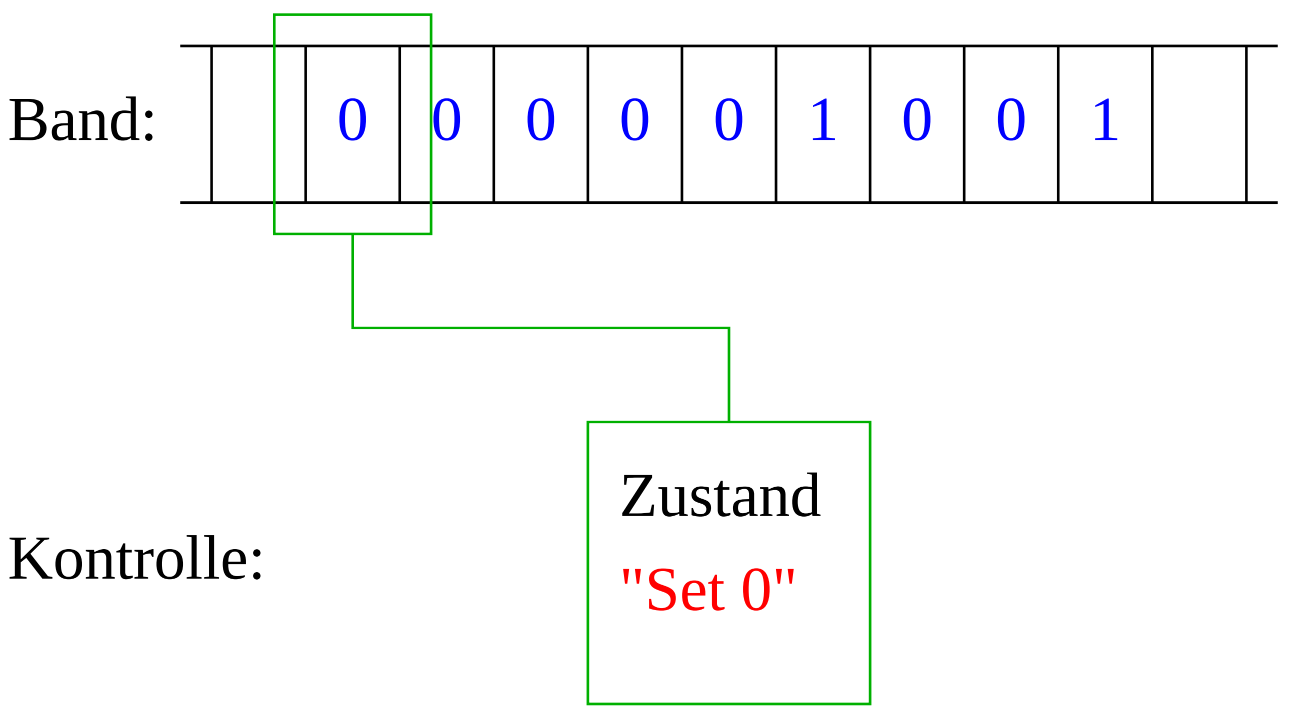
Operation = “Schreibe eine 0 und gehe nach links!”

neuer Zustand = unverändert



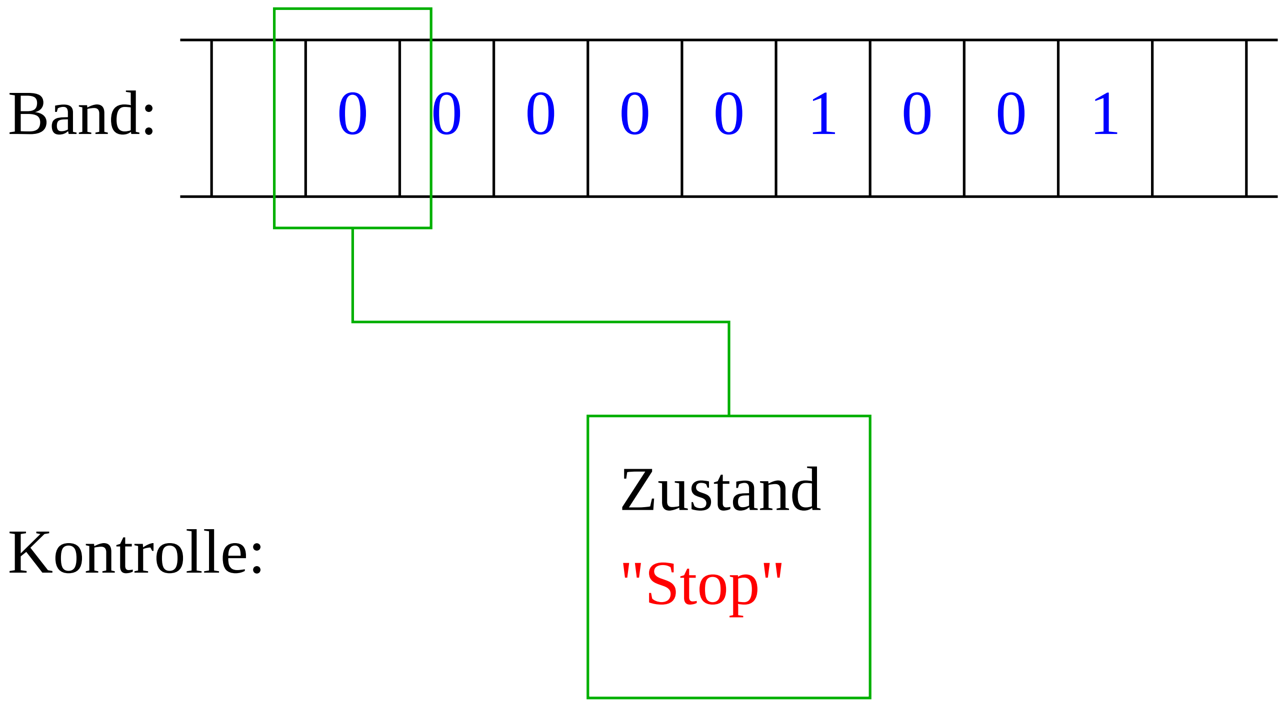
Operation = “Schreibe eine 0 und gehe nach links!”

neuer Zustand = unverändert



Operation = keine

neuer Zustand = “Stop”



Ende der Berechnung.

Fazit:

Die Turing-Maschine ist

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

$\Longrightarrow$ beliebtes Hilfsmittel in der ↑Berechenbarkeitstheorie und in der ↑Komplexitätstheorie.

Beispiel 2: JVM

- minimale Menge von Operationen, Kontroll- sowie Datenstrukturen, um Java-Programme auszuführen.

$\Longrightarrow$ Um Java auf einem Rechner XYZ auszuführen, benötigt man nur einen Simulator für die JVM, der auf XYZ läuft.

$\Longrightarrow$ Portabilität!

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

2 Eine einfache Programmiersprache

Eine Programmiersprache soll

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

Als Beispiel betrachten wir MiniJava.

2.1 Variablen

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

Beispiel:

```
int x, result;
```

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

Erklärung:

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen ("Integers") speichern sollen.

  `int` heißt auch Typ der Variablen `x` und `result`.

- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas "," getrennt.
- Am Ende steht ein Semikolon ";".

2.2 Operationen

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

Beispiele:

- `x = 7;`
  Die Variable `x` erhält den Wert 7.

- `result = x;`
  Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.

- `result = x + 19;`
  Der Wert der Variablen `x` wird ermittelt, 19 dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

- `result = x - 5;`
  Der Wert der Variablen `x` wird ermittelt, 5 abgezogen und dann das Ergebnis der Variablen `result` zugewiesen.

Achtung:

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

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

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

Achtung:

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

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

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

2.3 Kontrollstrukturen

Sequenz:

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

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

$\Longrightarrow$ 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: $\uparrow$ 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:

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

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

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

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

Das Programm `GGT` kann nun übersetzt und dann ausgeführt werden:

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

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

Achtung:

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

... kommt später in der Vorlesung :-)

3 Syntax von Programmiersprachen

Syntax (“Lehre vom Satzbau”):

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

Hilfsmittel bei natürlicher Sprache:

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

Hilfsmittel bei Programmiersprachen:

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

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

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

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

- Kontextbedingungen.

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

$\Longrightarrow$ 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 ($\uparrow$operationelle Semantik); oder
- die definierte Abbildung der Eingaben auf die Ausgaben ($\uparrow$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`

  $\longrightarrow$ Bezeichner für Basis-Typen;

- `if`, `else`, `while`

  $\longrightarrow$ Schlüsselwörter aus Programm-Konstrukten;

- `(,)`, `",'`, `{,}`, `,,;`

  $\longrightarrow$ Sonderzeichen.

3.2 Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten Zeichen:

$$\text{letter} \quad ::= \quad \texttt{\$} \mid \texttt{\_} \mid \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z}$$

$$\text{digit} \quad ::= \quad \texttt{0} \mid \ldots \mid \texttt{9}$$

3.2 Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten Zeichen:

$$\begin{array}{lll} \text{letter} & ::= & \texttt{\$} \mid \texttt{\_} \mid \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z} \\ \text{digit} & ::= & \texttt{0} \mid \ldots \mid \texttt{9} \end{array}$$

- letter und digit bezeichnen Zeichenklassen, d.h. Mengen von Zeichen, die gleich behandelt werden.
- Das Symbol “|” trennt zulässige Alternativen.
- Das Symbol “...” repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen :-)

Schritt 2: Angabe der Anordnung der Zeichen:

$$\text{name} \quad ::= \quad \text{letter} \; (\; \text{letter} \mid \text{digit} \;)^{*}$$

- 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 Zahlen bestehen aus einem Vorzeichen (das evt. auch fehlt)
und einer nichtleeren Folge von Ziffern:

sign	::=	+ | -
number	::=	sign ? digit digit*

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen Konstanten.
Ganze Zahlen bestehen aus einem Vorzeichen (das evt. auch fehlt) und einer nichtleeren Folge von Ziffern:

sign	::=	+ | -
number	::=	sign ? digit digit*

- Der Postfix-Operator “?” besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.
- 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 regulären Ausdrücke[a] (↑Automatentheorie).

[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`)sign? digit digit*
  float ::= sign? digit digit* exp |
  sign? 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.

$$
\begin{array}{lcl}
\text{program} & ::= & \text{decl* stmt*} \\
\text{decl} & ::= & \text{type name (, name)* ;} \\
\text{type} & ::= & \texttt{int}
\end{array}
$$

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.

$$
\begin{array}{lll}
\text{stmt} & ::= & \texttt{;} \mid \texttt{\{}\ \text{stmt}^*\ \texttt{\}} \mid \\
& & \text{name}\ \texttt{=}\ \text{expr}\texttt{;} \mid \text{name}\ \texttt{= read();} \mid \texttt{write(}\ \text{expr}\ \texttt{);} \mid \\
& & \texttt{if (}\ \text{cond}\ \texttt{)}\ \text{stmt} \mid \\
& & \texttt{if (}\ \text{cond}\ \texttt{)}\ \text{stmt}\ \texttt{else}\ \text{stmt} \mid \\
& & \texttt{while (}\ \text{cond}\ \texttt{)}\ \text{stmt}
\end{array}
$$

- 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.

$$
\begin{array}{lcl}
\text{expr} & ::= & \text{number} \mid \text{name} \mid \texttt{(}\ \text{expr}\ \texttt{)} \mid \\
 & & \text{unop expr} \mid \text{expr binop expr} \\
\text{unop} & ::= & \texttt{-} \\
\text{binop} & ::= & \texttt{-} \mid \texttt{+} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}
\end{array}
$$

- 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 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.

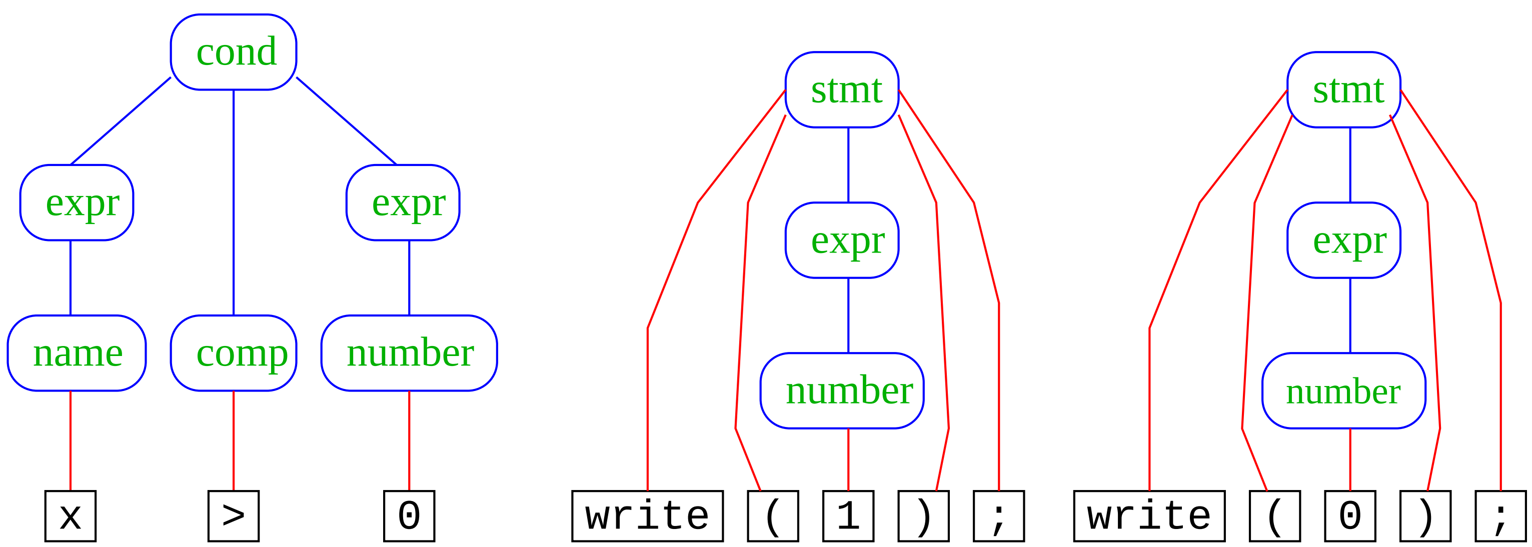
Puh!!! Geschafft ...

Beispiel:

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

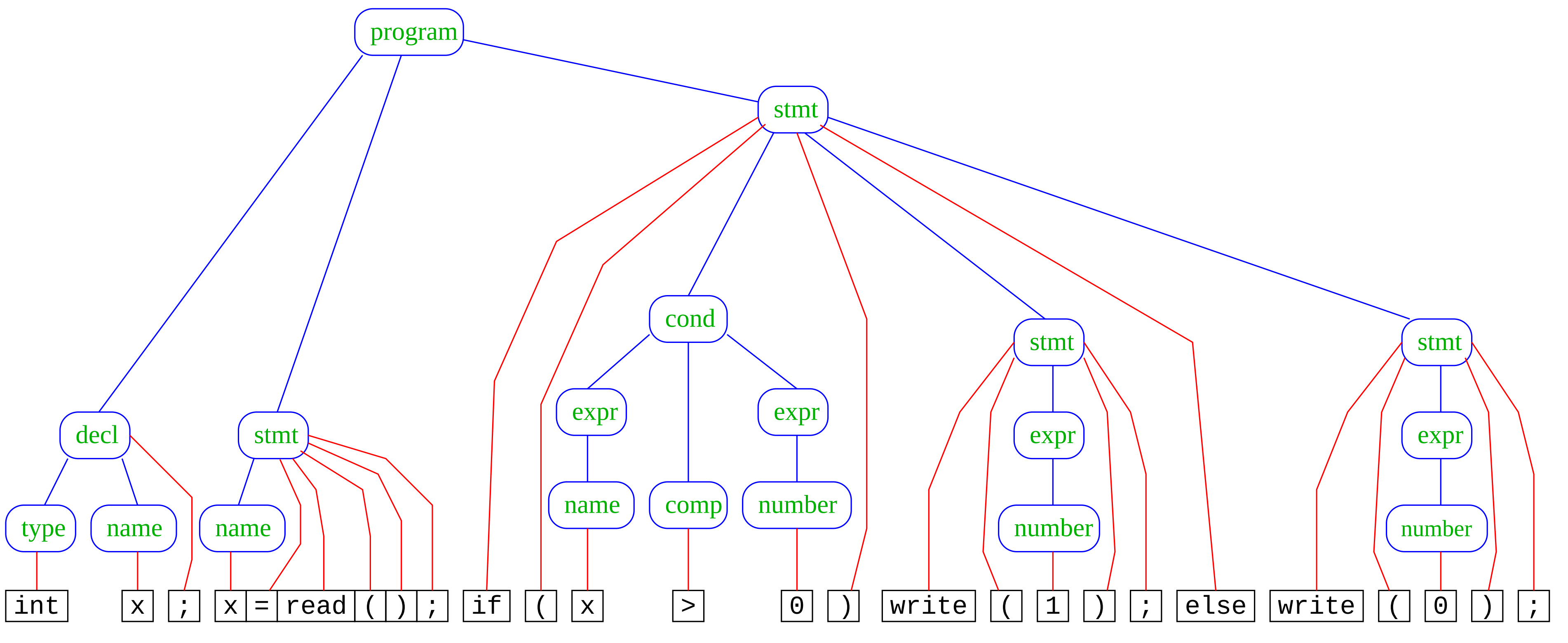
Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume:

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



Blätter: Wörter/Tokens

innere Knoten: Namen von Programm-Bestandteilen



program
decl
stmt
stmt
type
name
name
cond
expr
comp
expr
name
number
stmt
expr
number
stmt
expr
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 (Erfinder von Fortran)

Achtung:

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

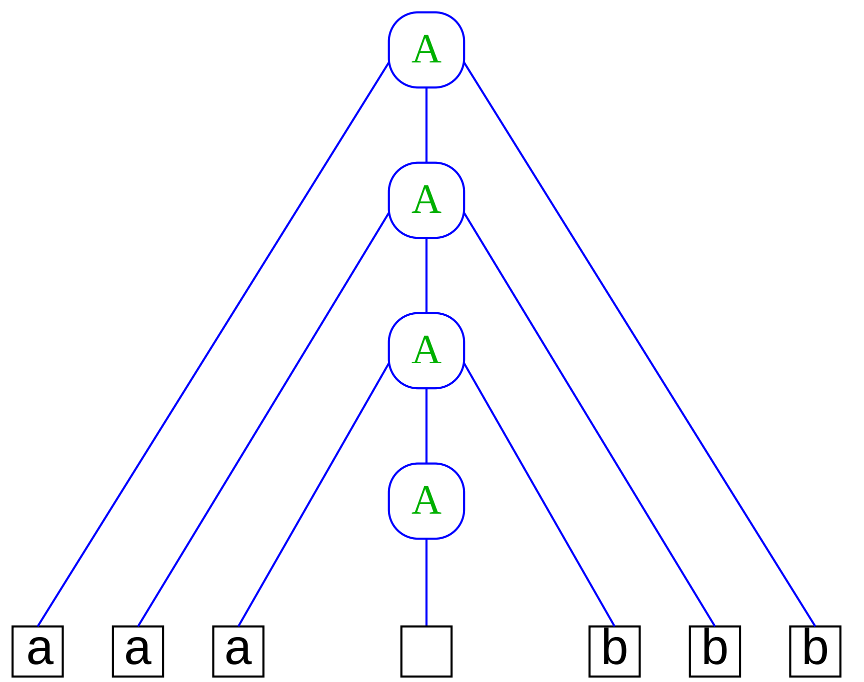
Beispiel:

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

lässt sich mithilfe einer Grammatik beschreiben:

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

Syntax-Baum für das Wort `aaabbb` :



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

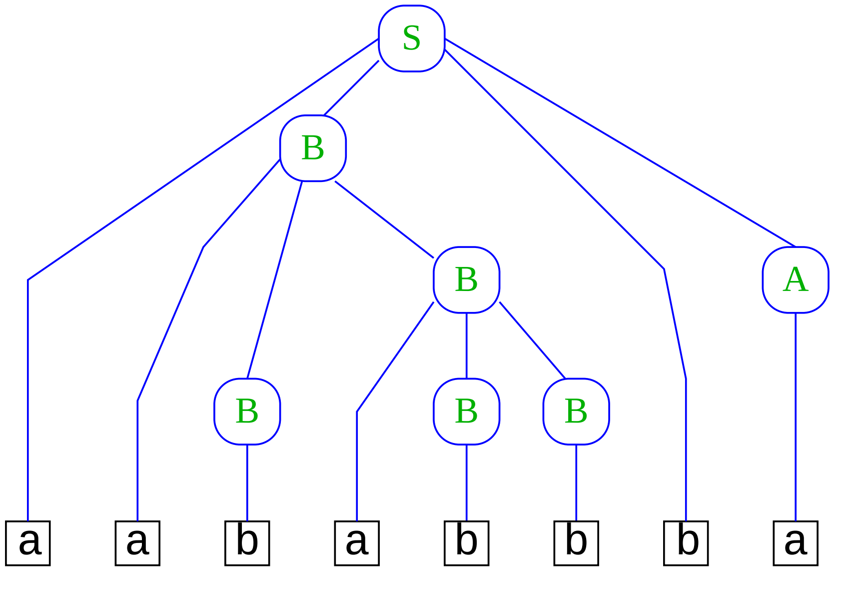
Weiteres Beispiel:

$\mathcal{L}$ = alle Worte mit gleich vielen `a`'s und `b`'s

Zugehörige Grammatik:

$$
\begin{array}{lcl}
S & ::= & (\, \texttt{b}\ A \mid \texttt{a}\ B \,)^* \\
A & ::= & (\, \texttt{b}\ A\ A \mid \texttt{a} \,) \\
B & ::= & (\, \texttt{a}\ B\ B \mid \texttt{b} \,)
\end{array}
$$

Syntax-Baum für das Wort `aababbba` :

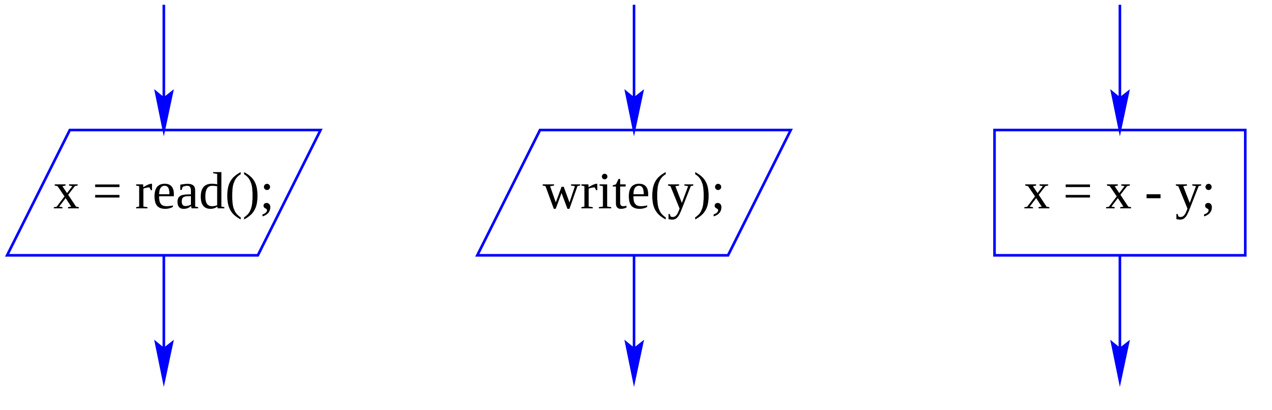


4 Kontrollfluss-Diagramme

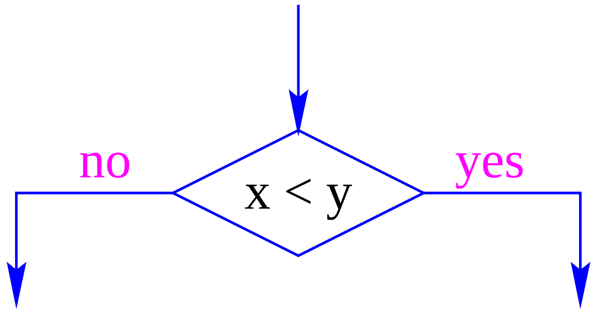
In welcher Weise die Operationen eines Programms nacheinander ausgeführt werden, läßt sich anschaulich mithilfe von Kontrollfluss-Diagrammen darstellen.

Ingredienzien:

Startknoten Endknoten

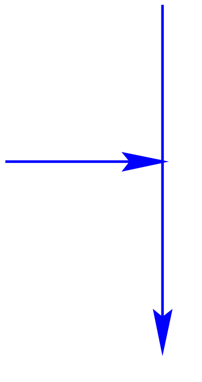


Eingabe Ausgabe Zuweisung



bedingte Verzweigung

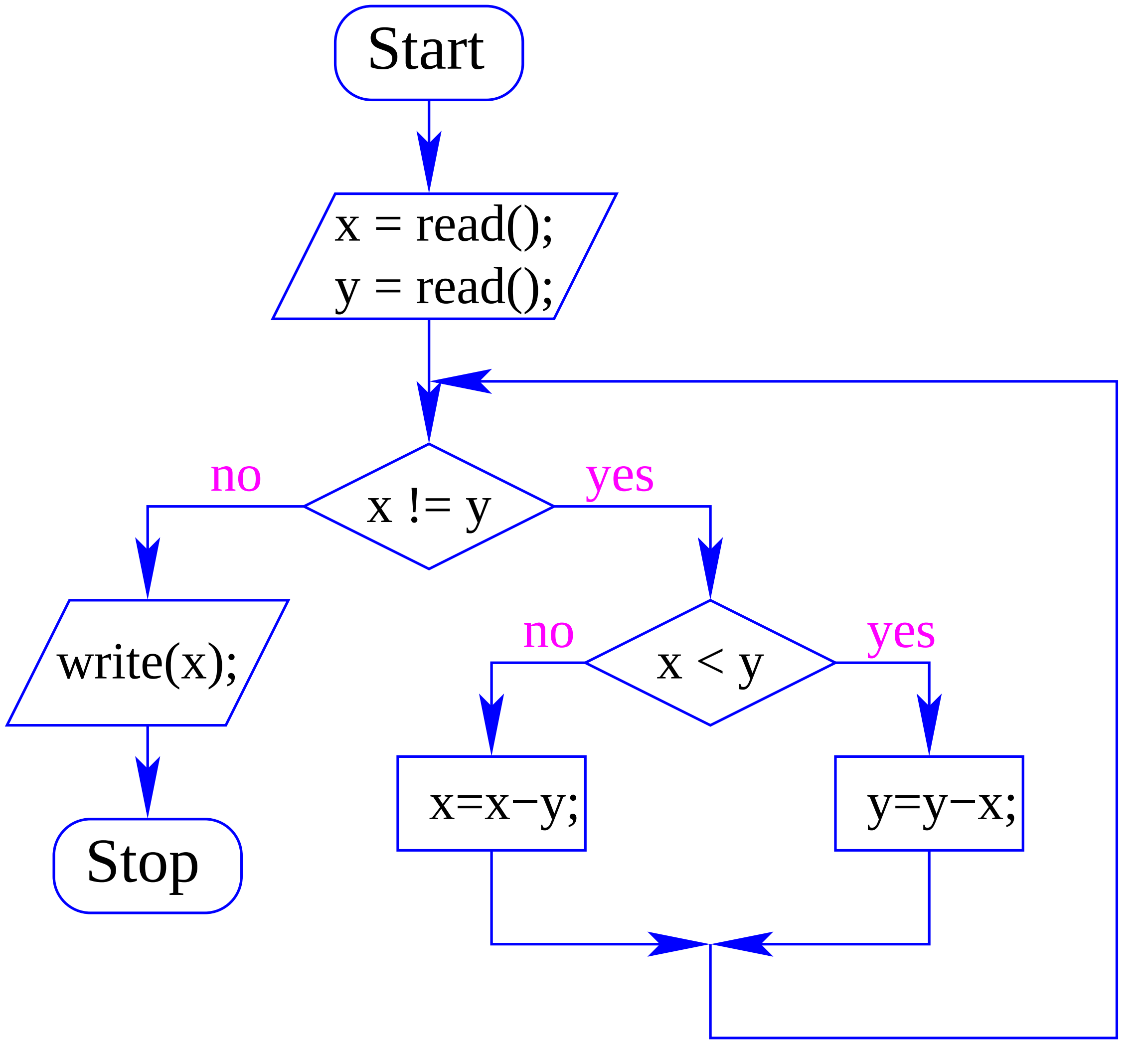
Kante



Zusammenlauf

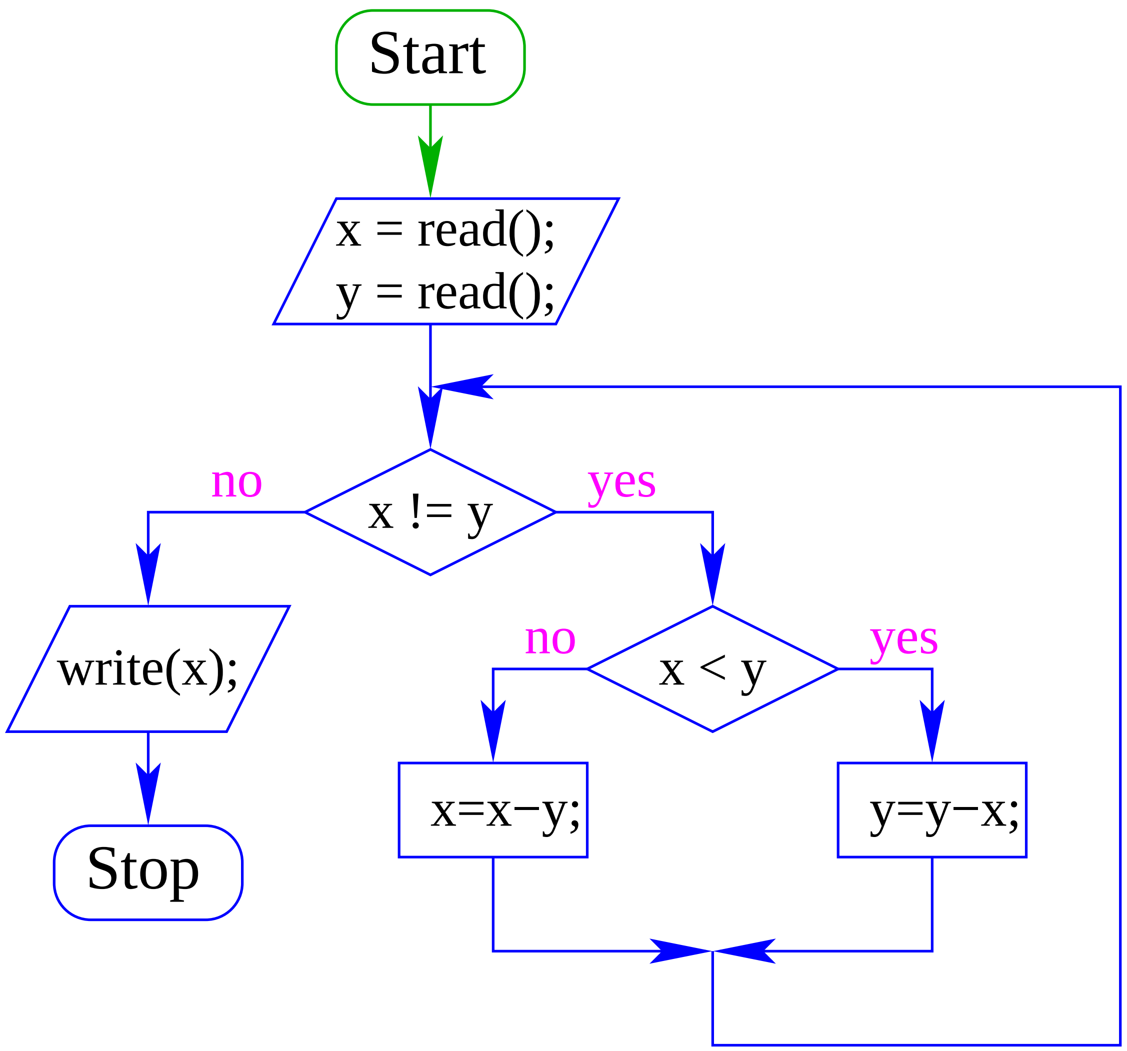
Beispiel:

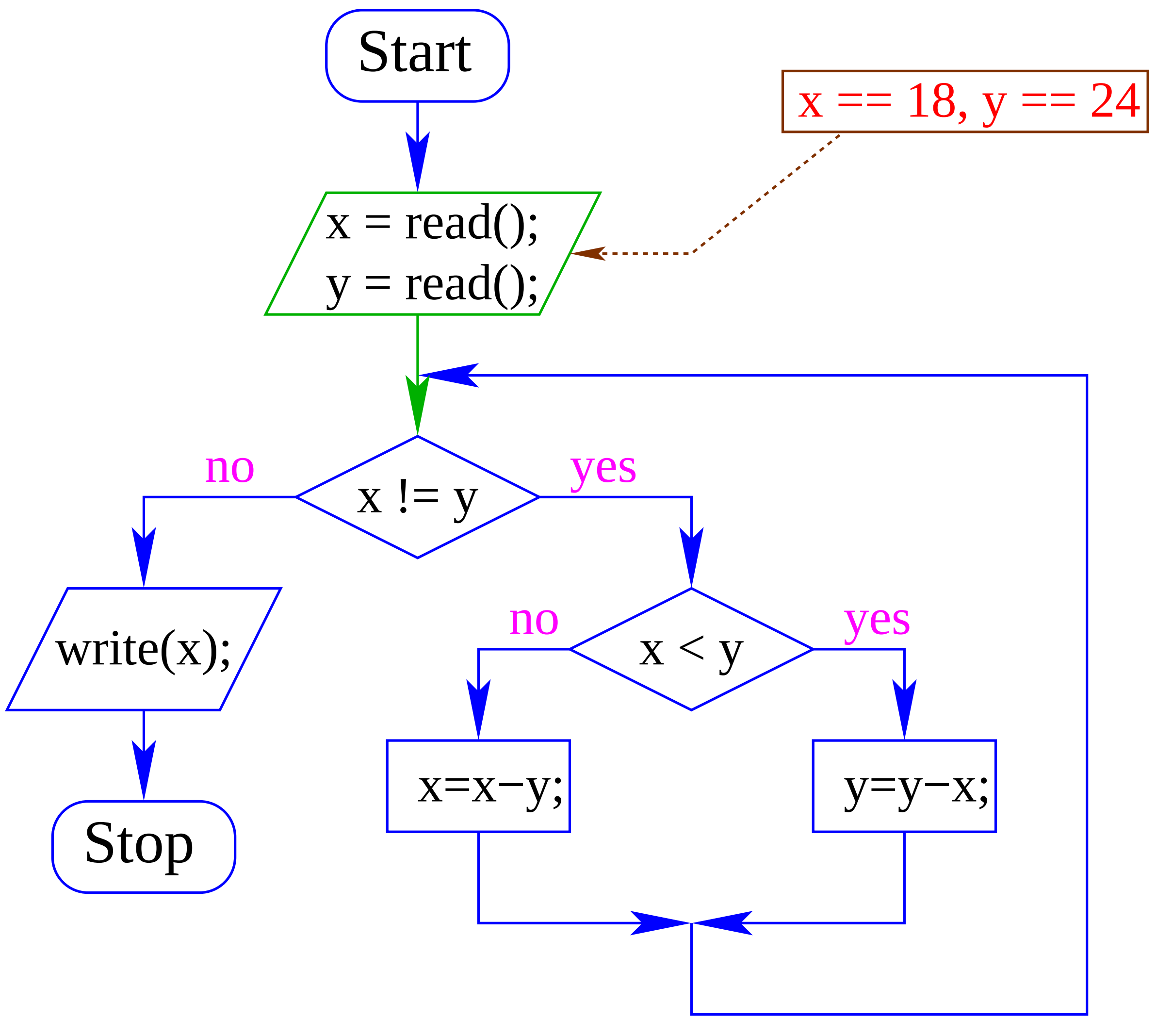
```
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```

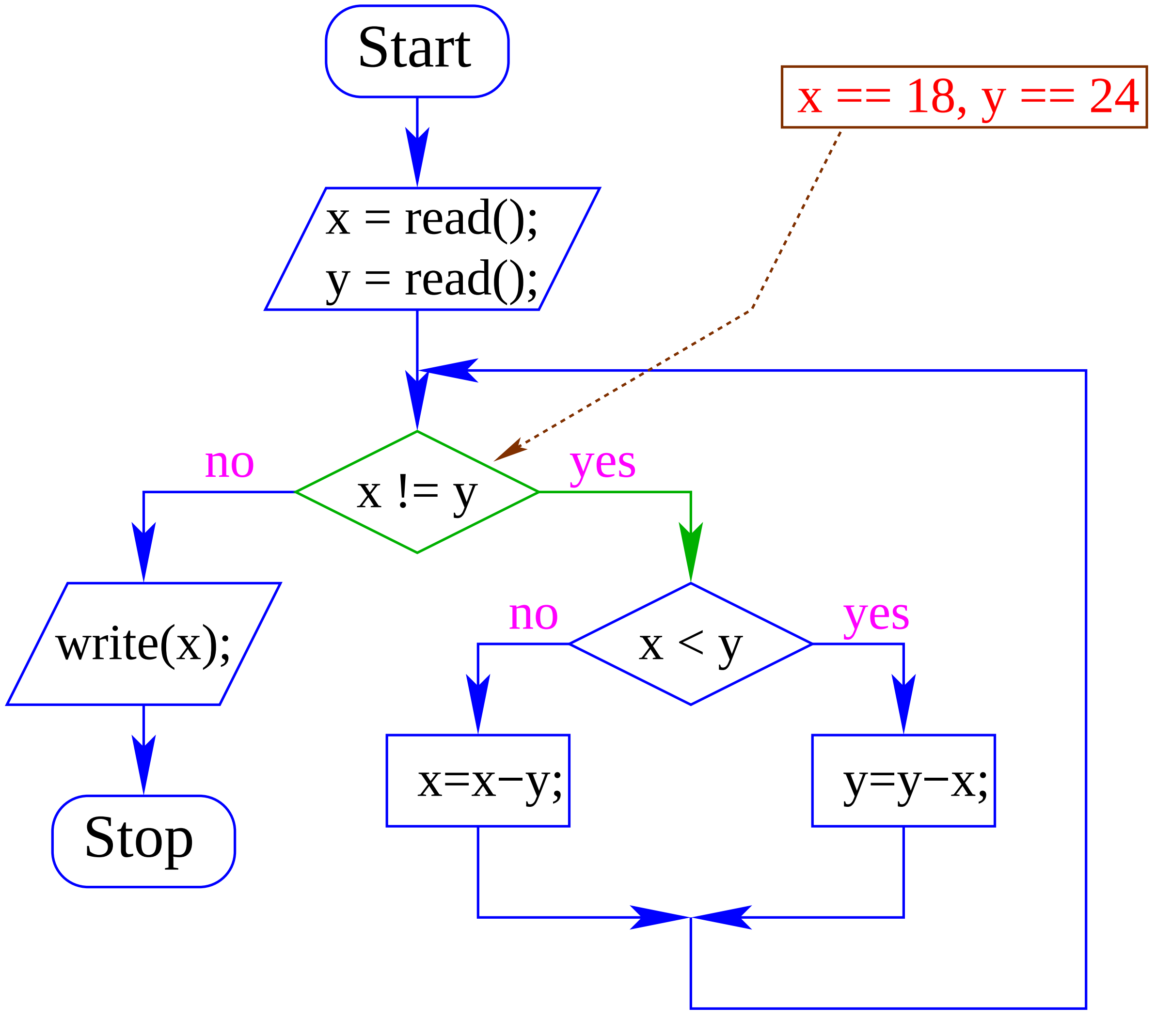


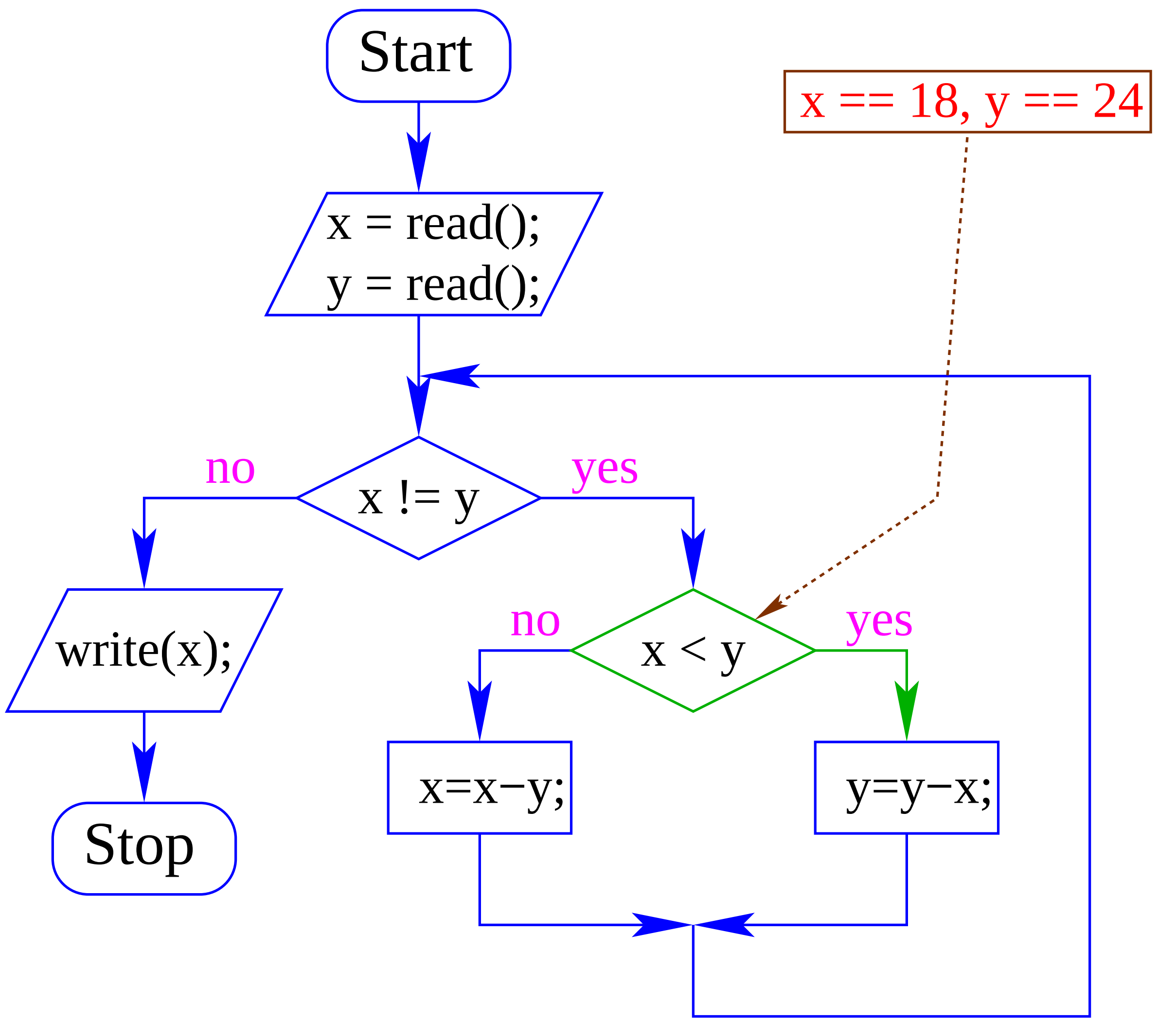
- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollfluss-Diagramm vom Startknoten zum Endknoten.
- Die Deklarationen von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer dem Start- und Endknoten) sind die dabei auszuführenden Operationen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

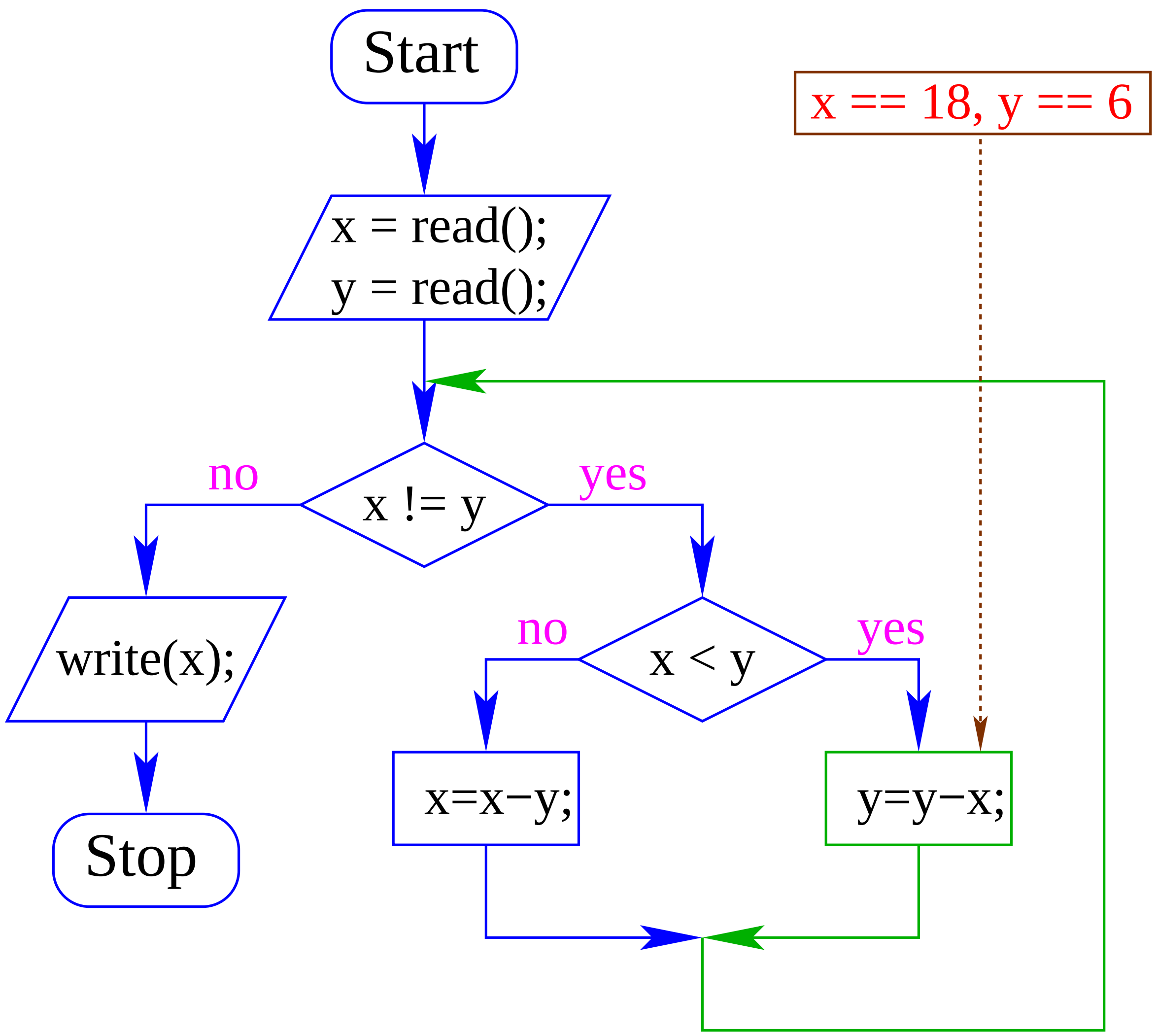
$\Longrightarrow$ operationelle Semantik

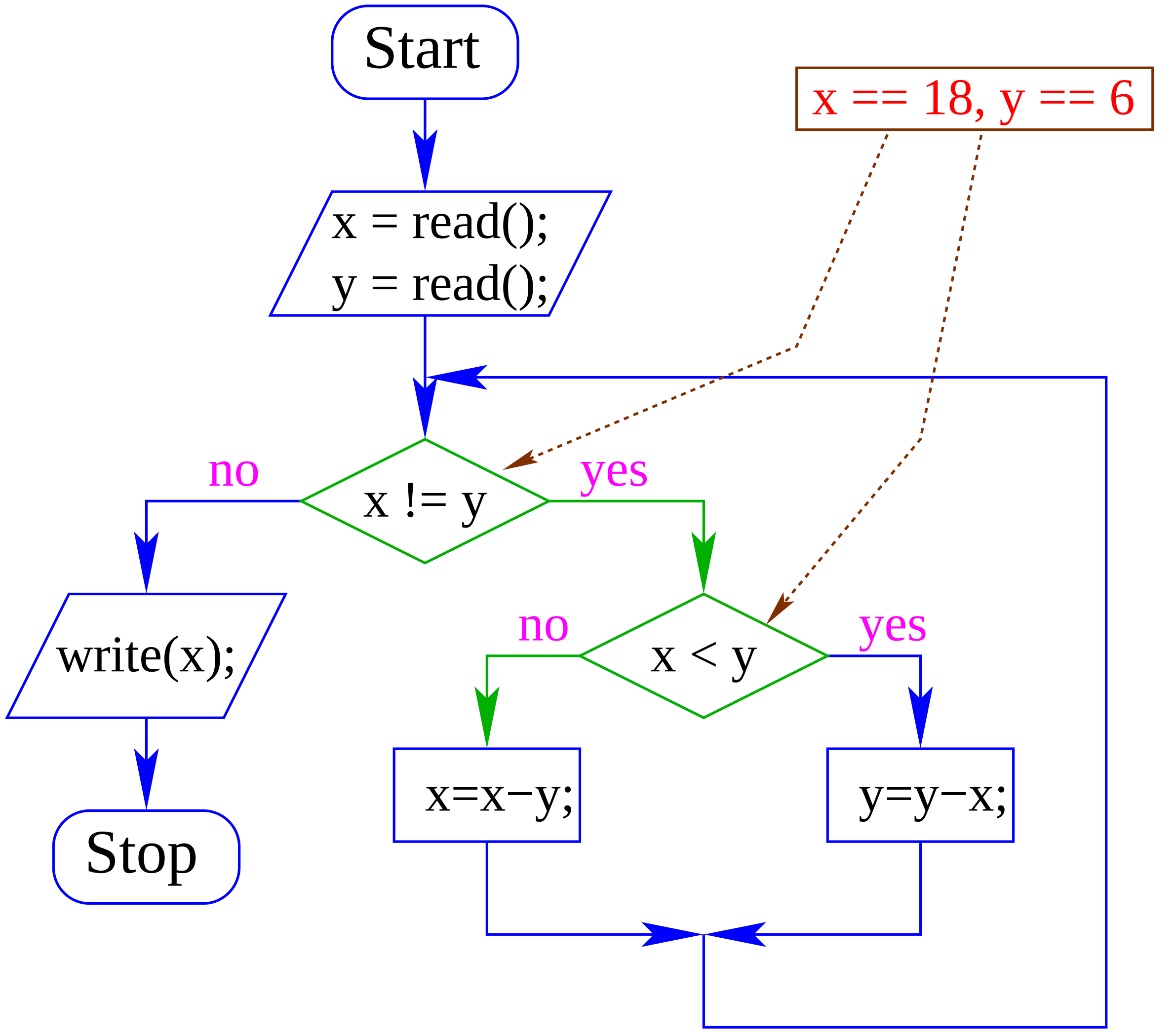

Start
x = read();
y = read();
no
x != y
yes
write(x);
Stop
no
x < y
yes
x=x−y;
y=y−x;

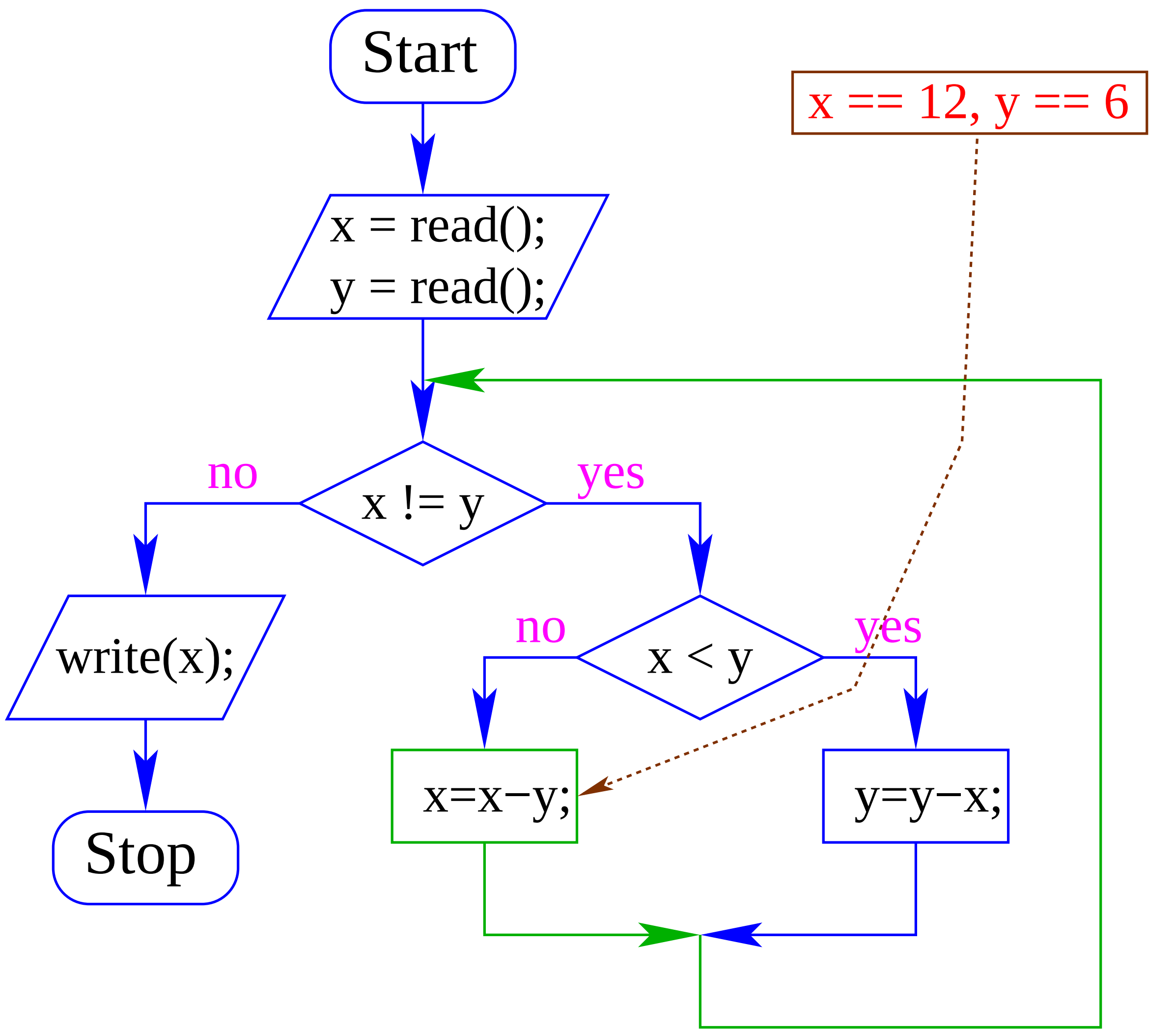

Start
x == 18, y == 24
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

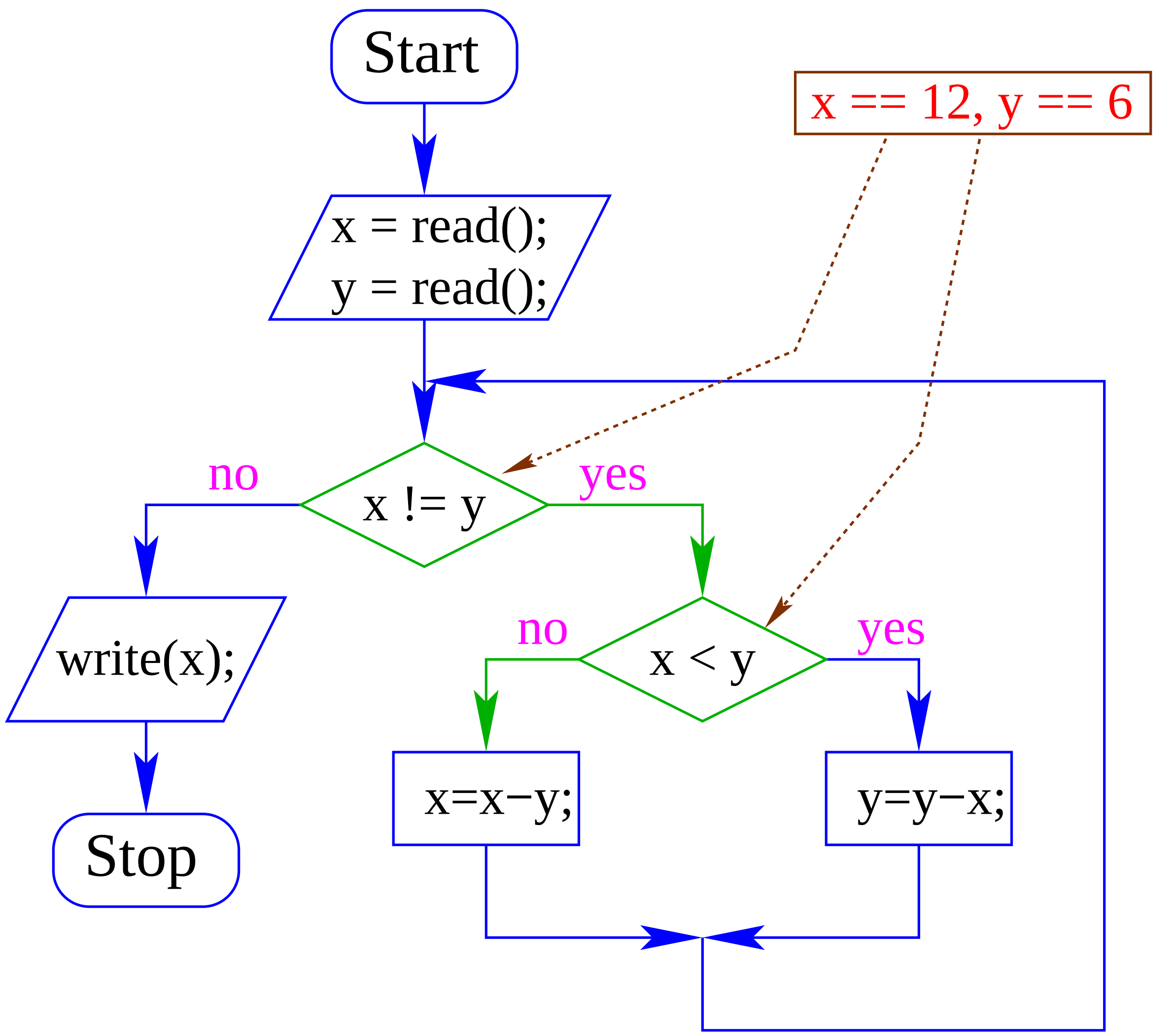

Start
x == 18, y == 24
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
x=x−y;
y=y−x;
Stop

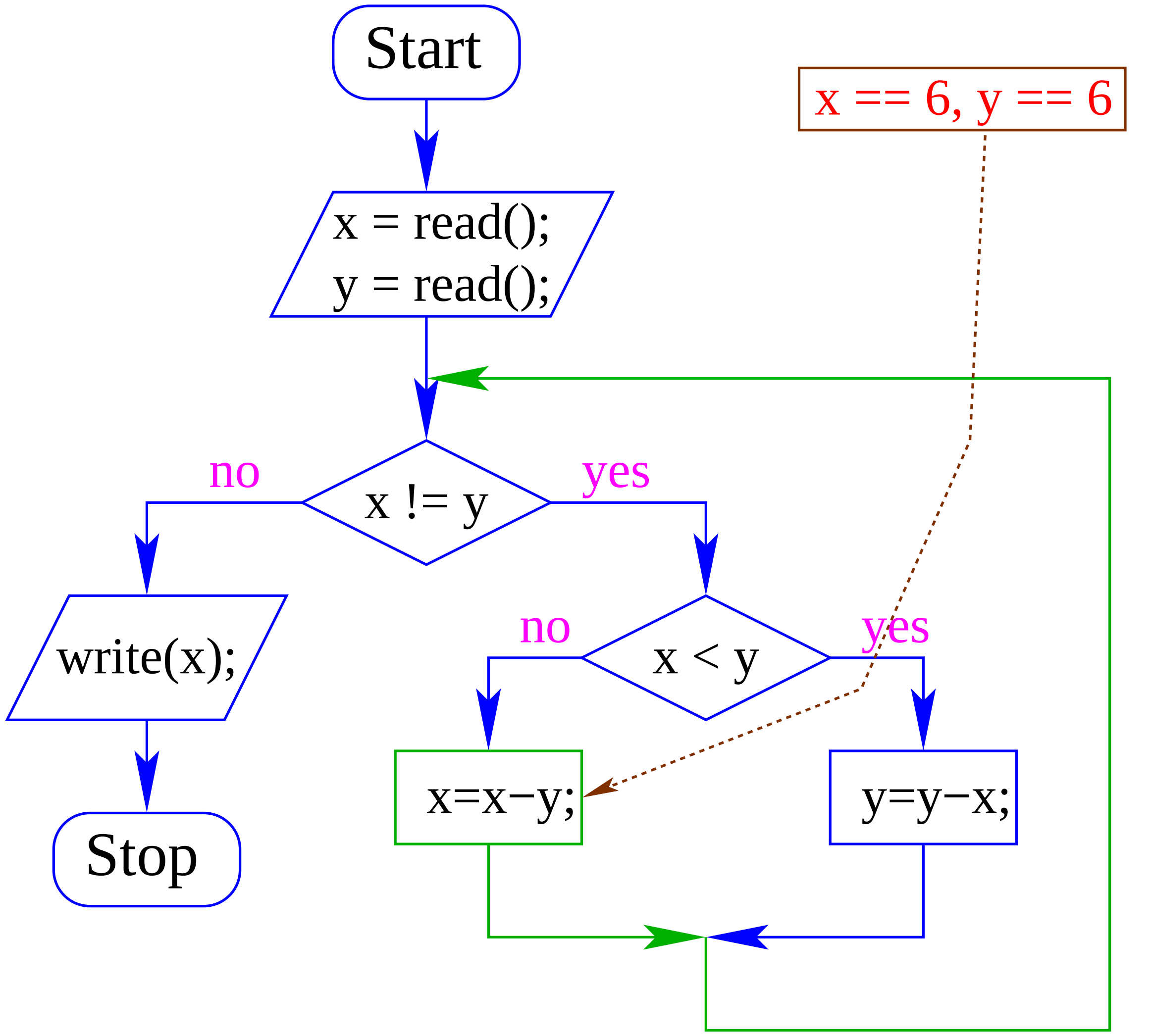

Start
x = read();
y = read();
x == 18, y == 24
no
x != y
yes
write(x);
Stop
no
x < y
yes
x=x−y;
y=y−x;

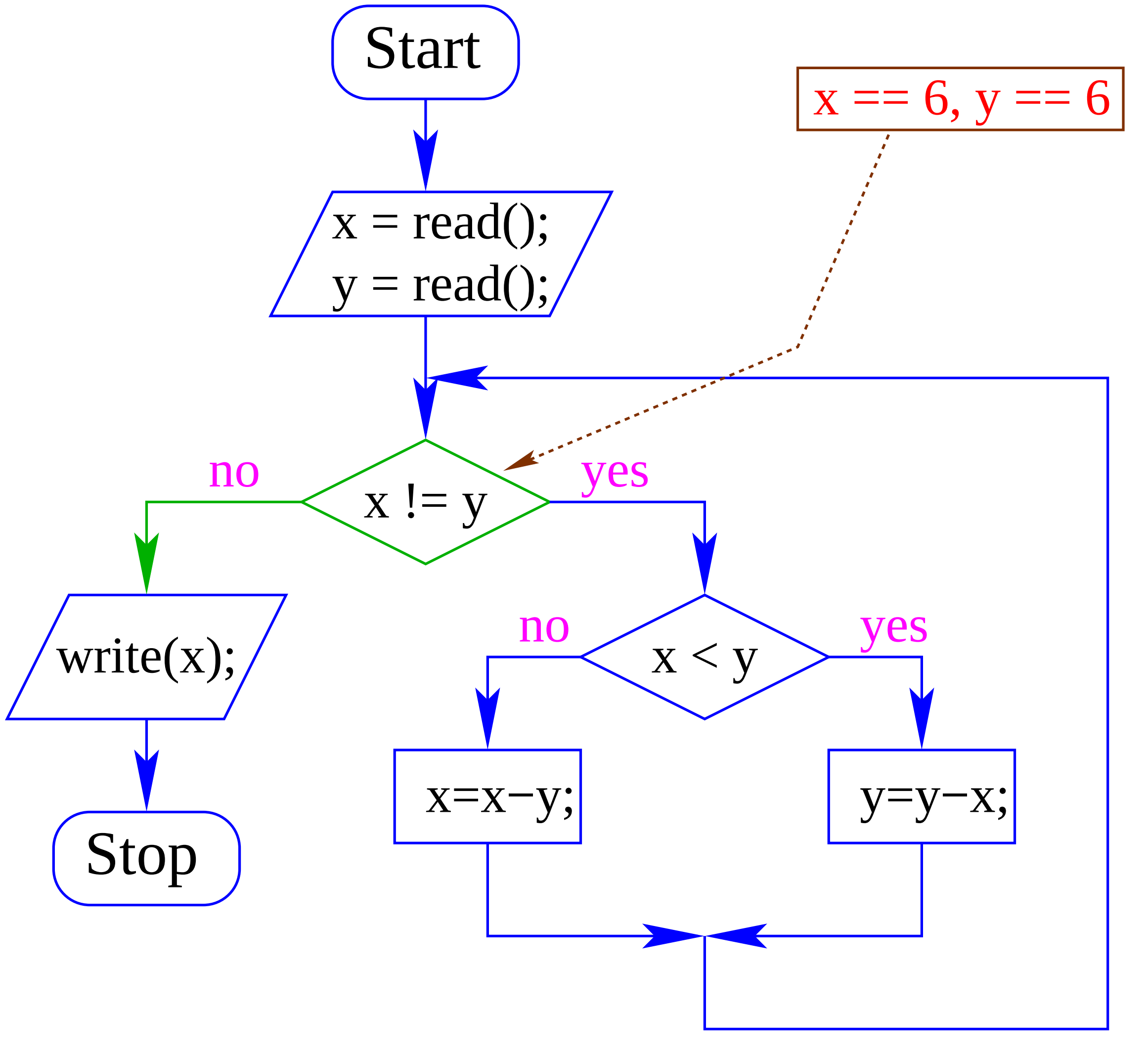

Start
x = read();
y = read();
x == 18, y == 6
no
x != y
yes
write(x);
no
x < y
yes
x=x−y;
y=y−x;
Stop

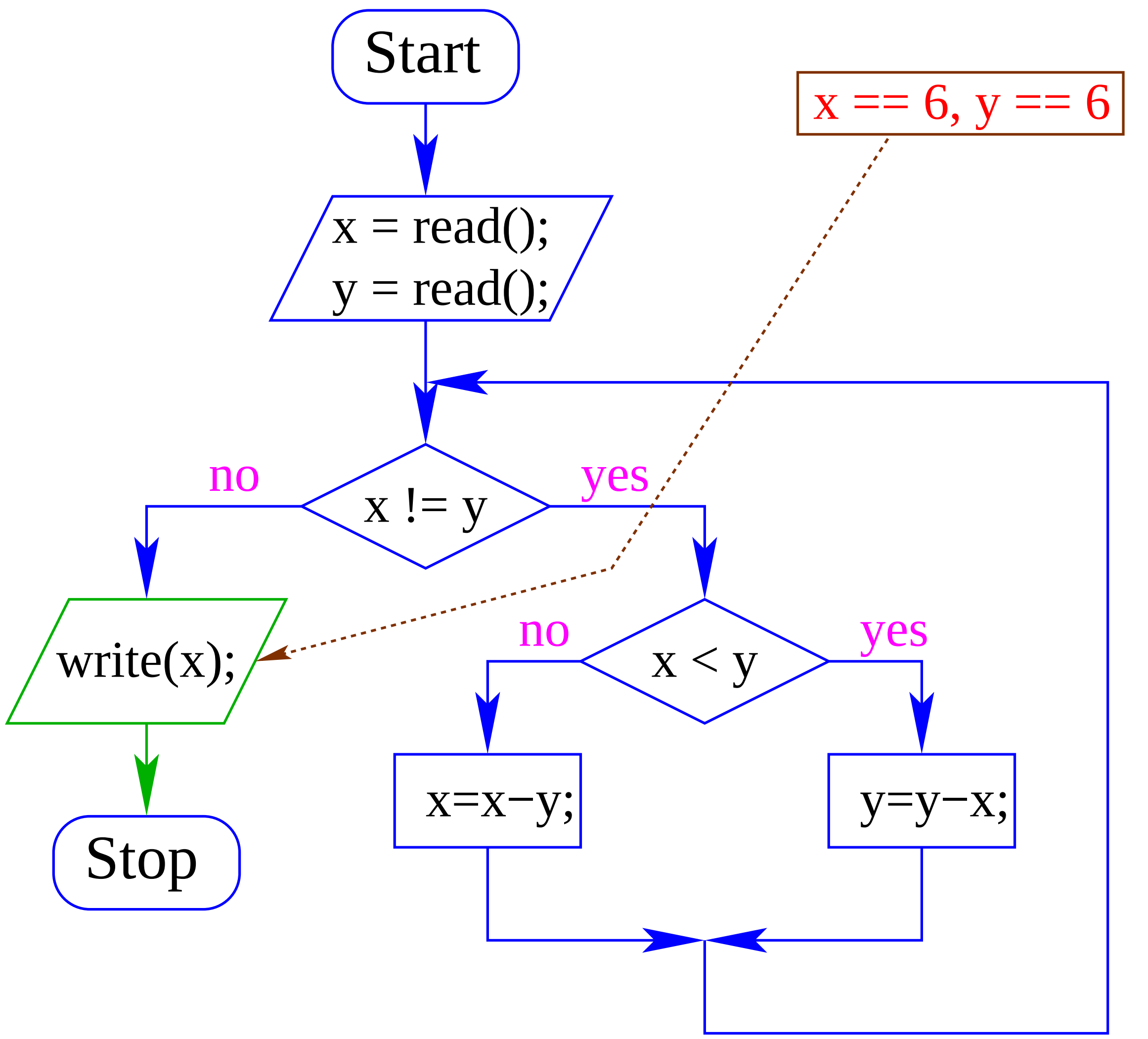

Start
x == 18, y == 6
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

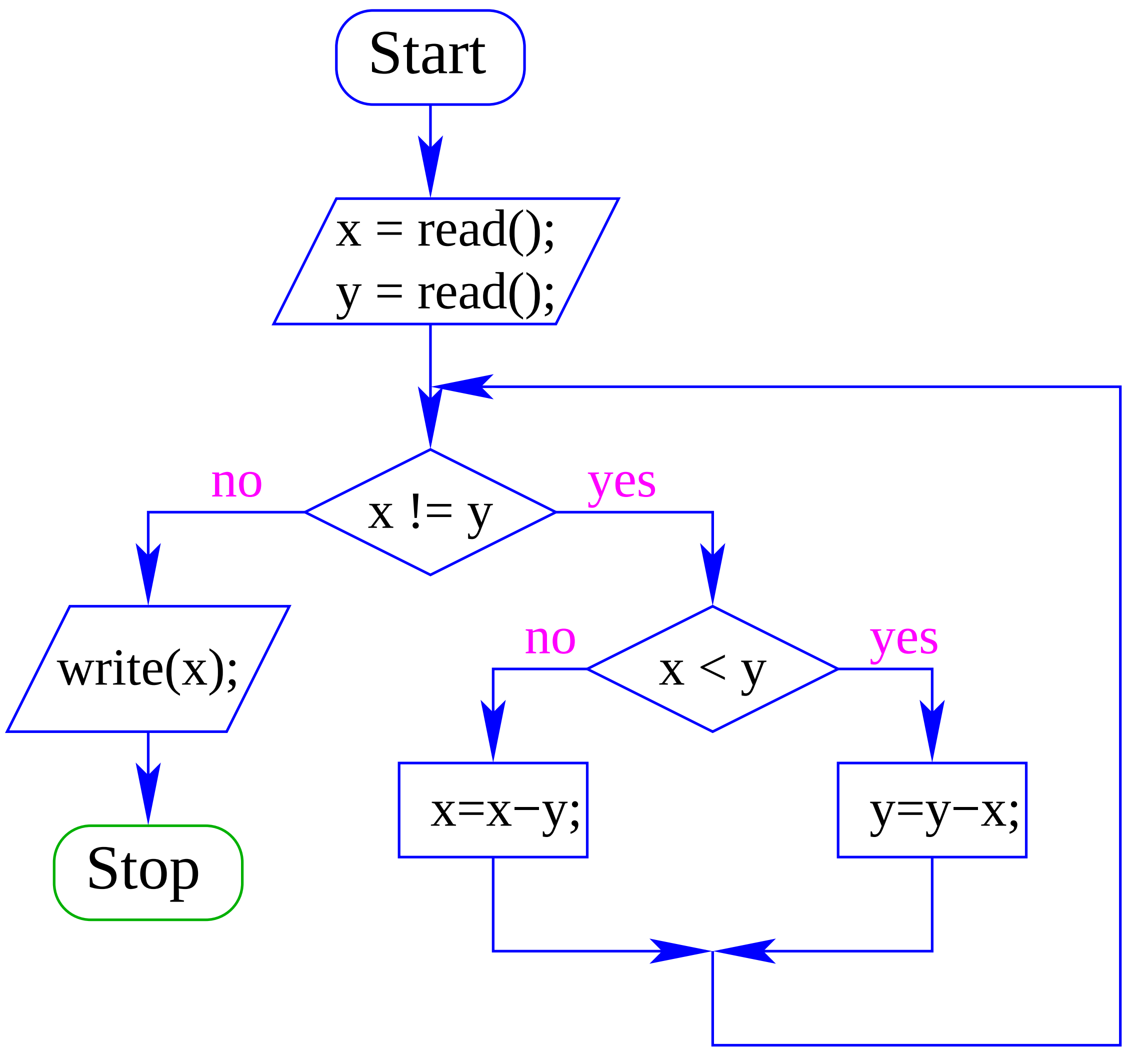

Start
x = read();
y = read();
x == 12, y == 6
no
x != y
yes
write(x);
no
x < y
yes
x=x−y;
y=y−x;
Stop


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


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


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

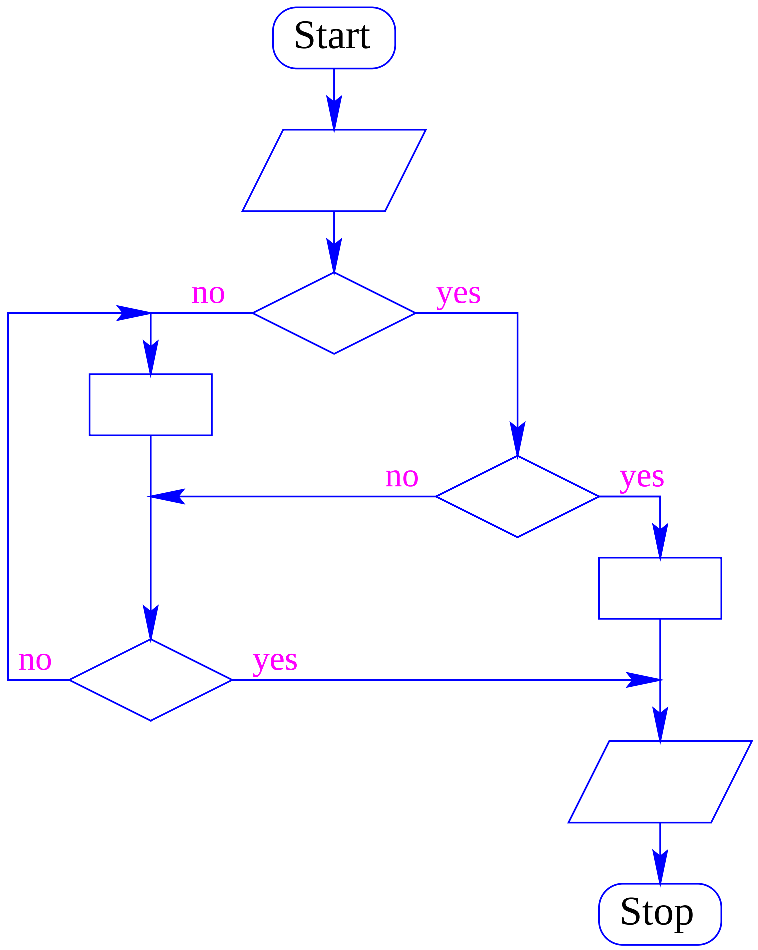

Start
x == 6, y == 6
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;


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

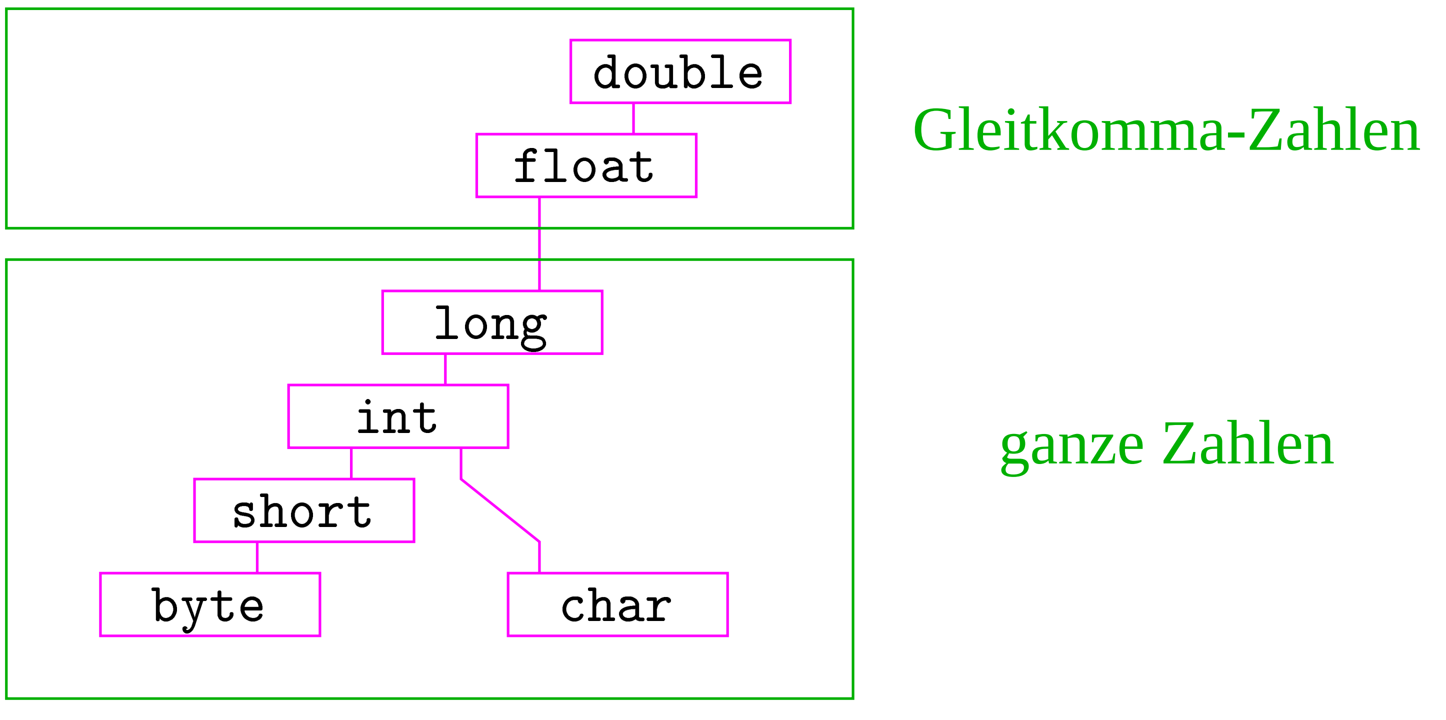
- die Zeichen unserer Tastatur (inklusive Umlaute);
- die chinesischen Schriftzeichen;
- die ägyptischen Hieroglyphen ...

`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) ...


double
float
Gleitkomma-Zahlen
long
int
short
byte
char
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, d.h. ein `double` liefern.
- Das Ergebnis einer Operation auf Basistypen für ganze Zahlen kann einen Wert aus einem größeren ganzzahligen Basistyp liefern (mindestens aber `int`).
- Wird das Ergebnis einer Variablen zugewiesen, sollte deren Typ dies zulassen :-)
- Mithilfe von expliziten Type Casts lässt sich das (evt. unter Verlust von Information) stets bewerkstelligen.

Beispiele:

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

5.3 Strings

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

Hier behandeln wir nur drei Eigenschaften:

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

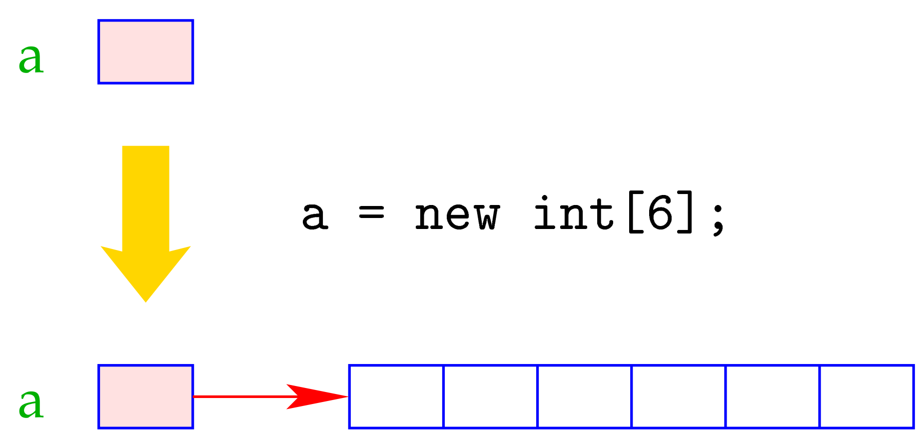
- Lege sie konsekutiv ab!
- Greife auf einzelne Werte über ihren Index zu!

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

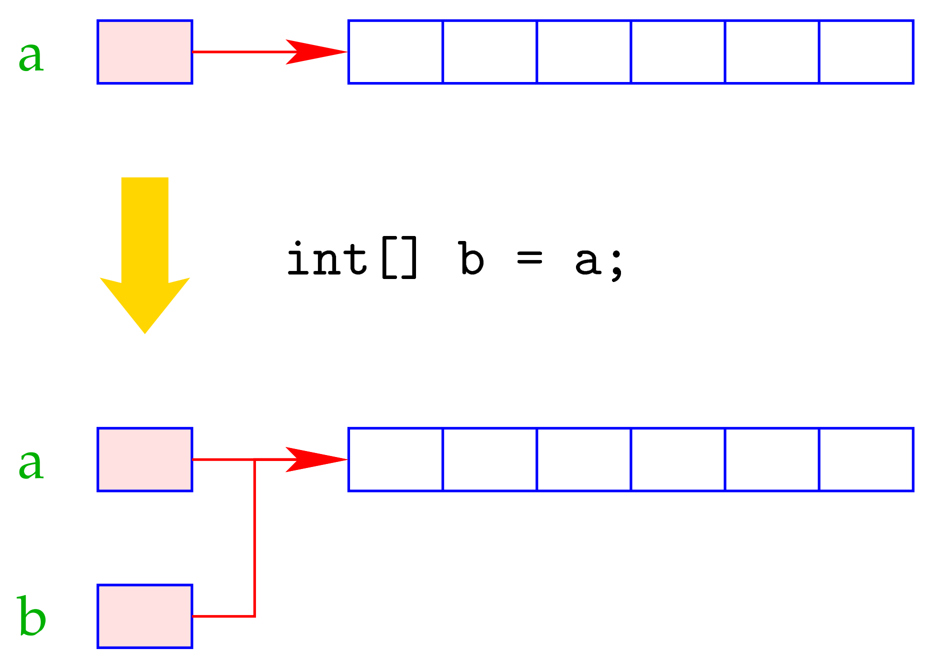
Beispiel: Einlesen eines Felds

```
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:



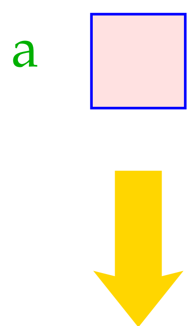
- Der Wert einer Feld-Variable ist also ein Verweis.
- `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



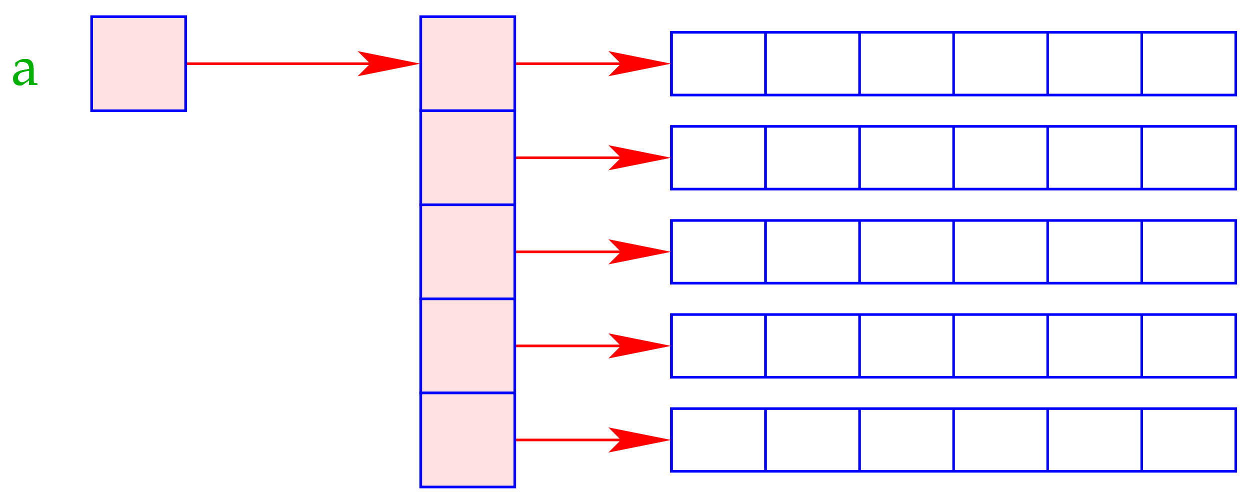
- Die Elemente eines Felds sind von 0 an durchnumeriert.
- Die Anzahl der Elemente des Felds name ist `name.length`.
- Auf das *i*-te Element des Felds name greift man mittels `name[i]` zu.
- Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall $\{0, \ldots,$ `name.length-1`$\}$ 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 ...




```
a = new int[5][6];
```



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.

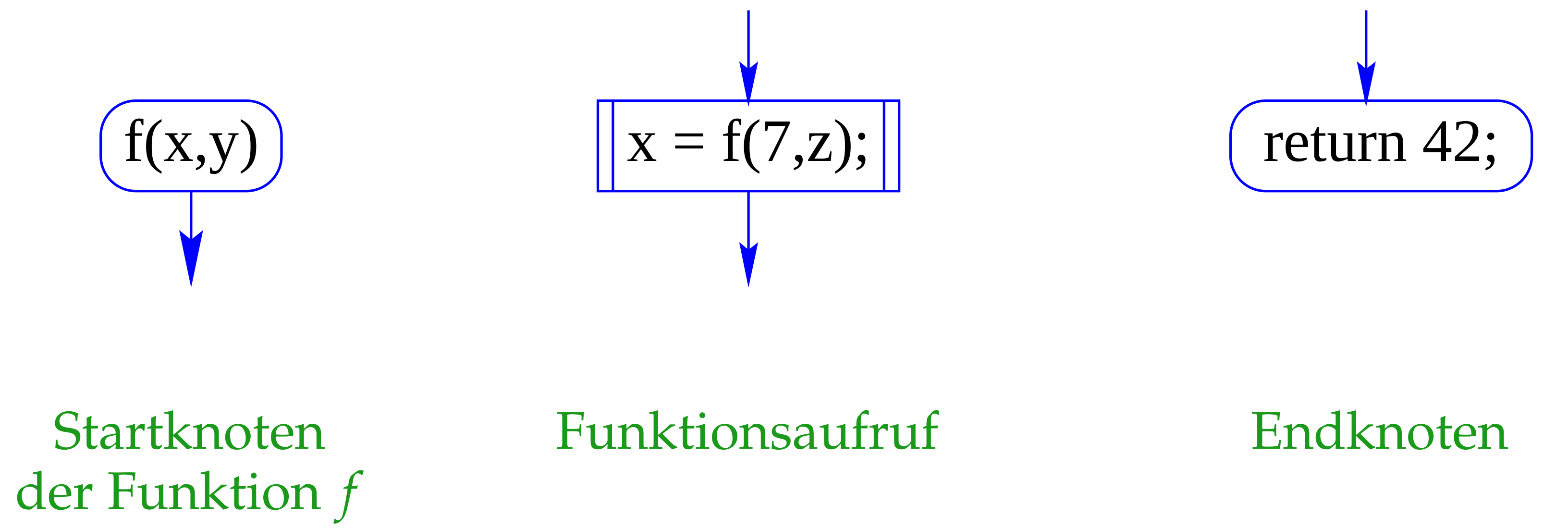
```
public class Test extends MiniJava {
   public static void main (String [] args) {
      write(args[0]+args[1]);
      }
}  // end of class Test
```

Dann liefert der Aufruf:

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

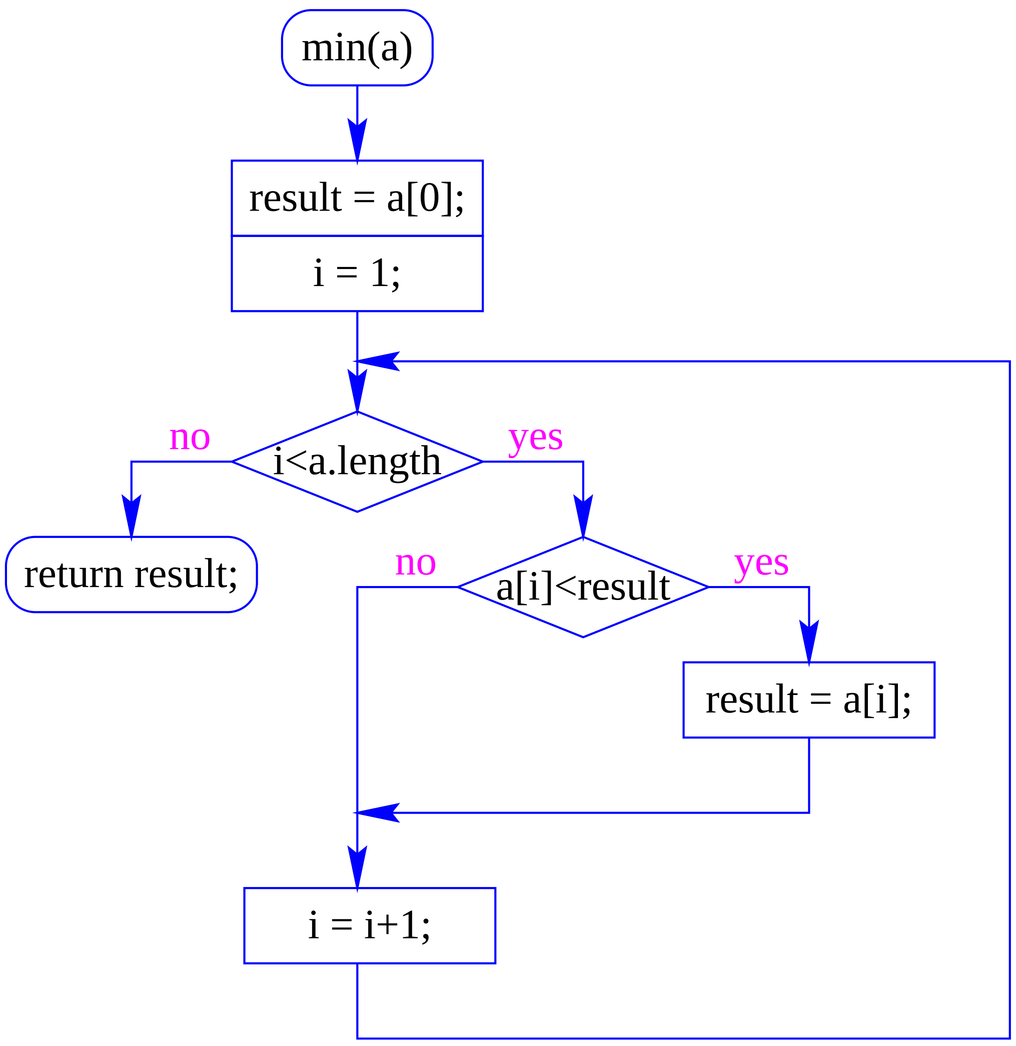
... die Ausgabe: Hello World!

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

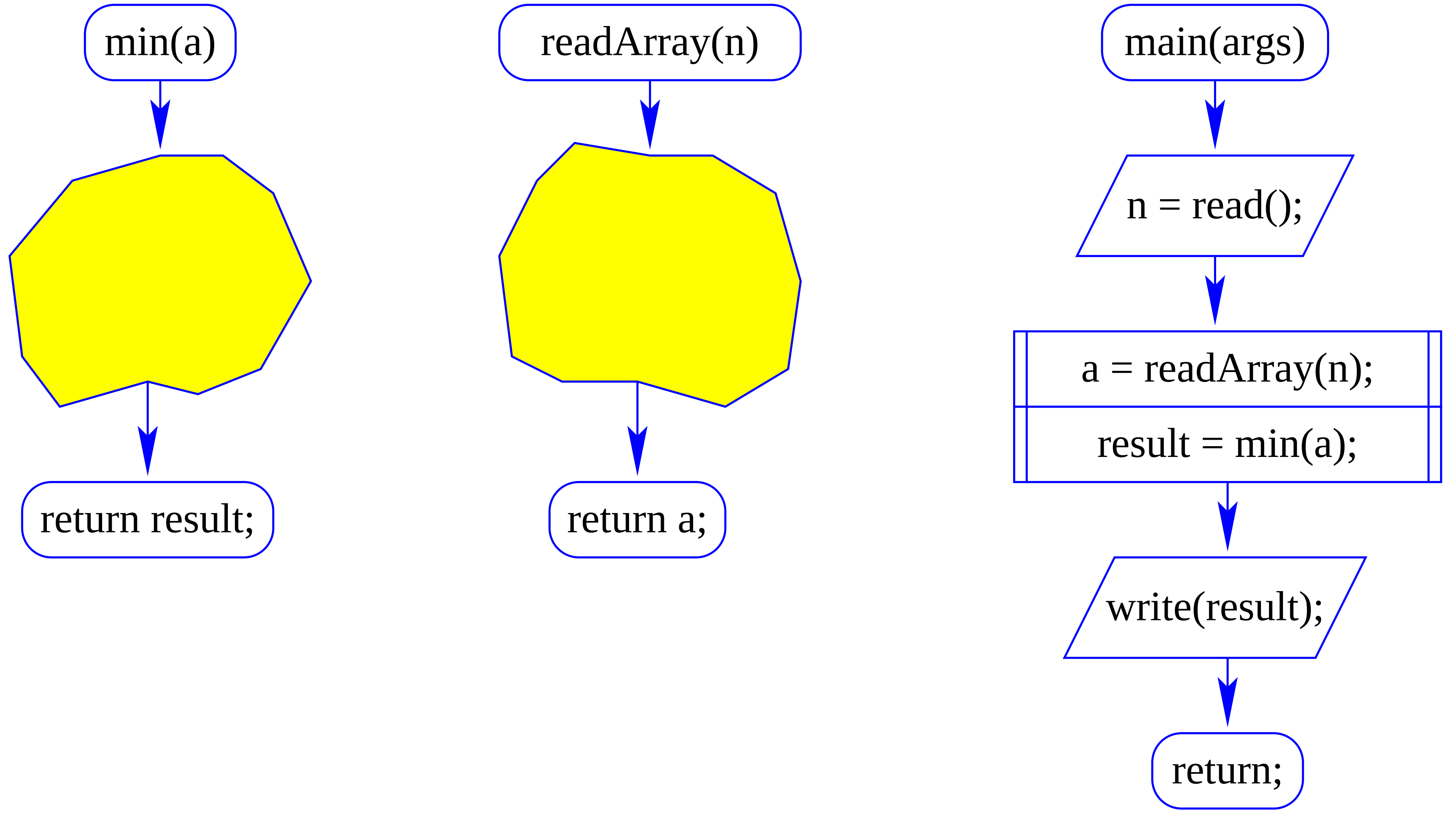


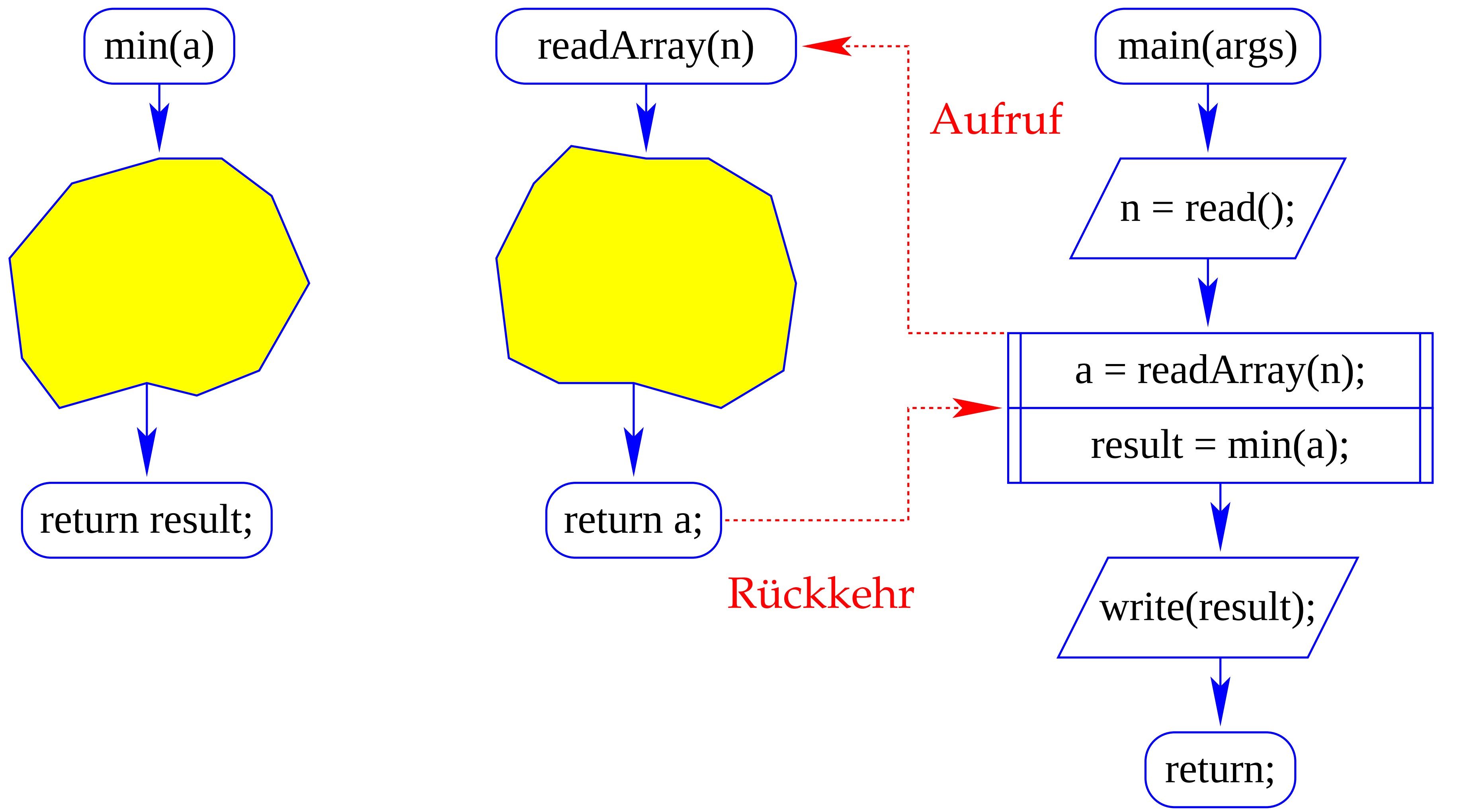
- Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- Ein Aufrufknoten repäsentiert eine Teilberechnung der aufgerufenen Funktion.

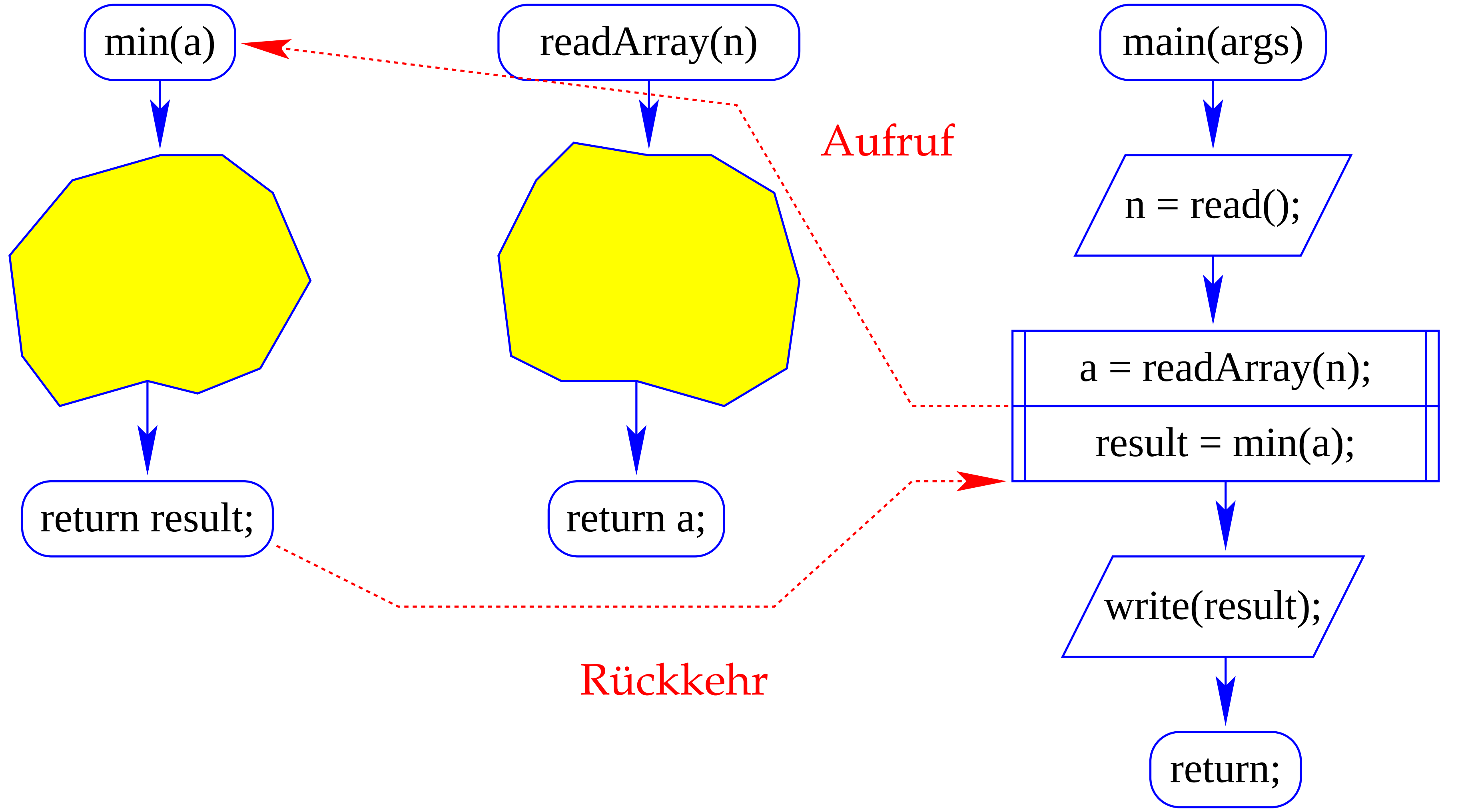
Teildiagramm für die Funktion `min()`:



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.

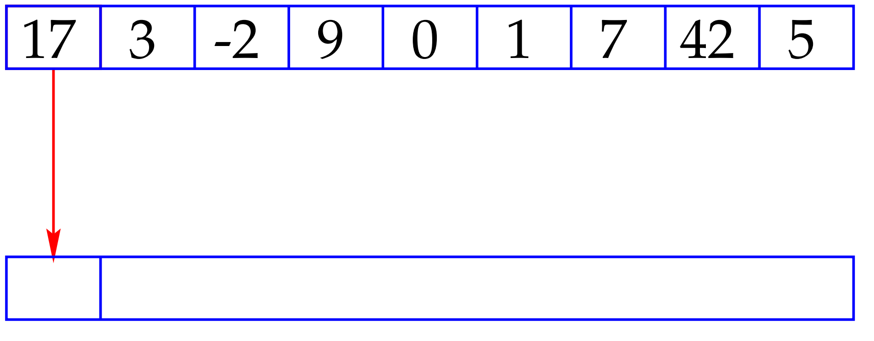
Idee:

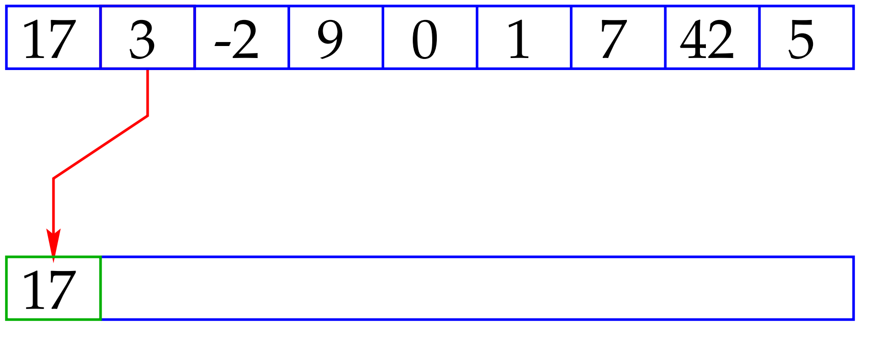
- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

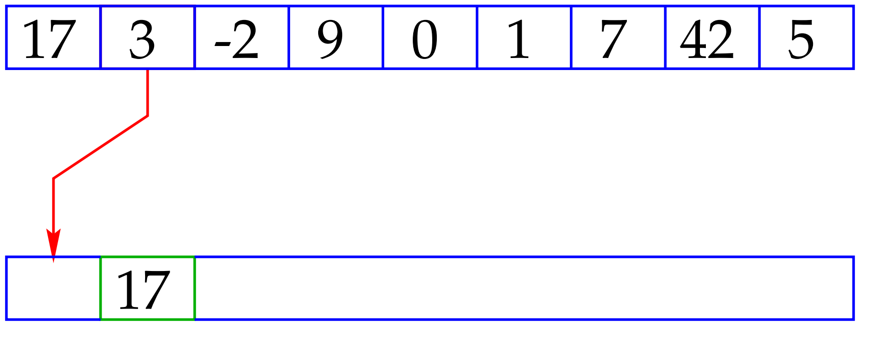
$\Longrightarrow$ Sortieren durch Einfügen ...

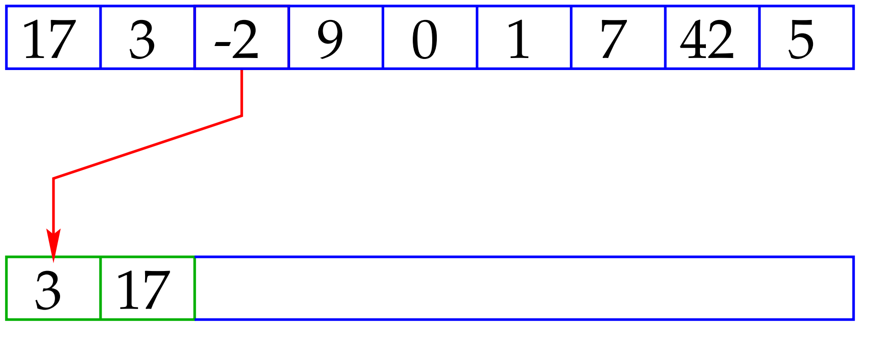
```
public static int[] sort (int[] a) {
    int n = a.length;
    int[] b = new int[n];
    for (int i = 0; i < n; ++i)
        insert (b, a[i], i);
            // b    = Feld, in das eingefügt wird
            // a[i] = einzufügendes Element
            // i    = Anzahl von Elementen in b
    return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

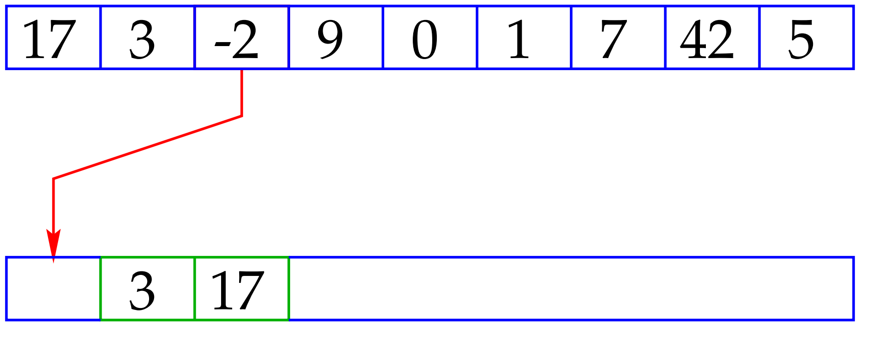
Teilproblem: Wie fügt man ein ???

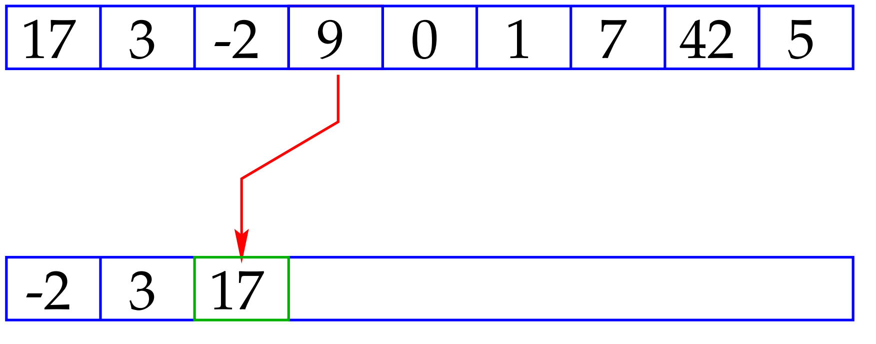

17
3
-2
9
0
1
7
42
5

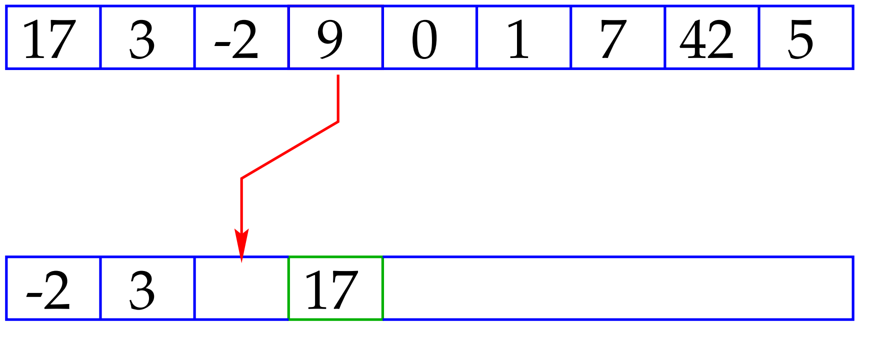

17
3
-2
9
0
1
7
42
5
17

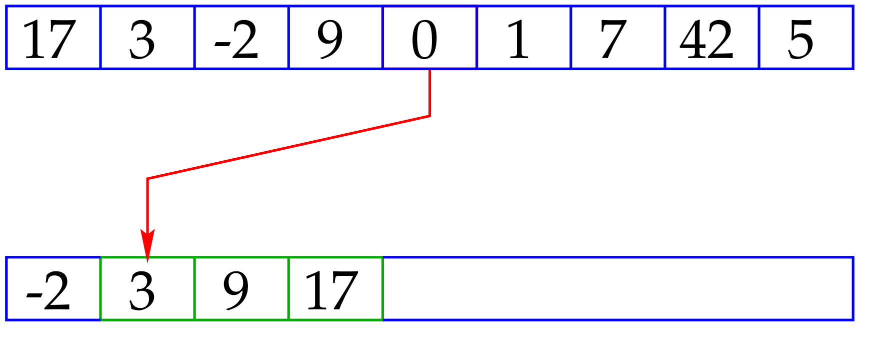

17
3
-2
9
0
1
7
42
5
17

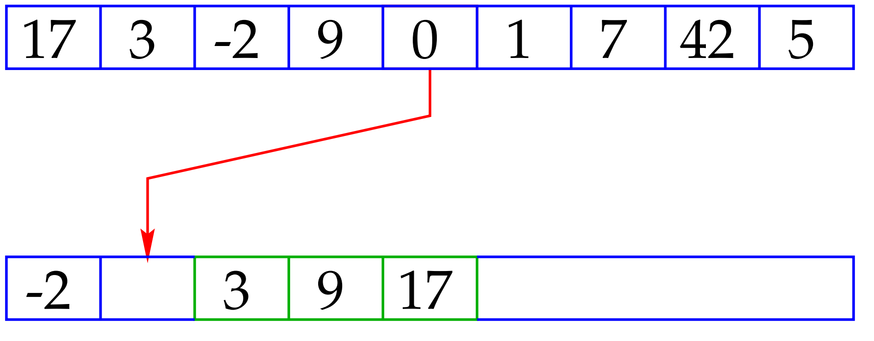

17
3
-2
9
0
1
7
42
5
3
17

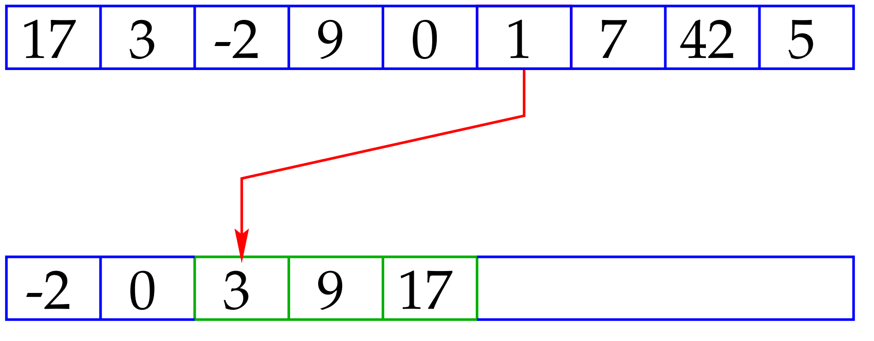

17
3
-2
9
0
1
7
42
5
3
17

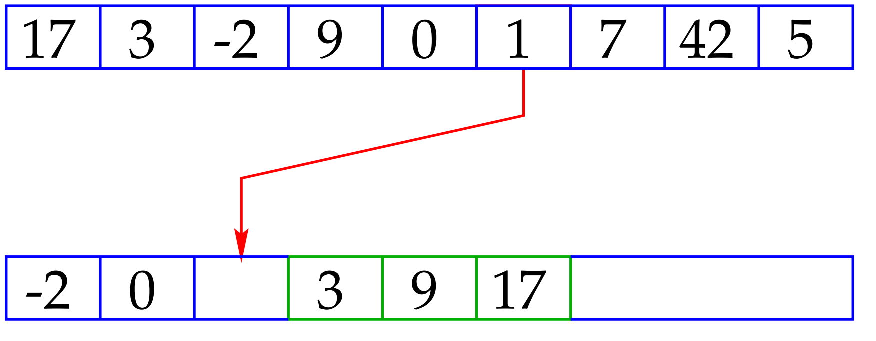

17 3 -2 9 0 1 7 42 5
-2 3 17

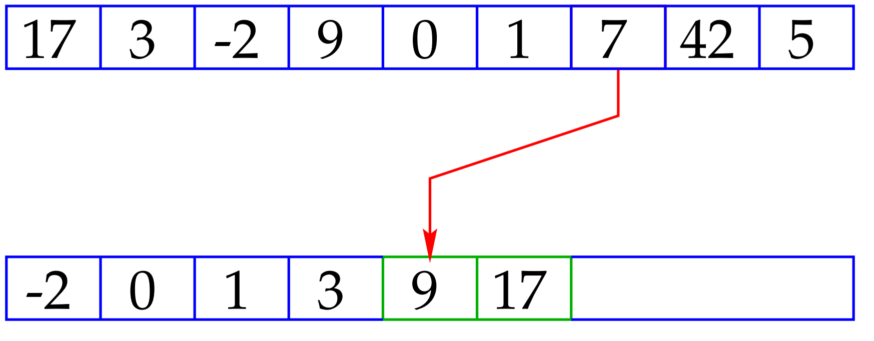

17
3
-2
9
0
1
7
42
5
-2
3
17

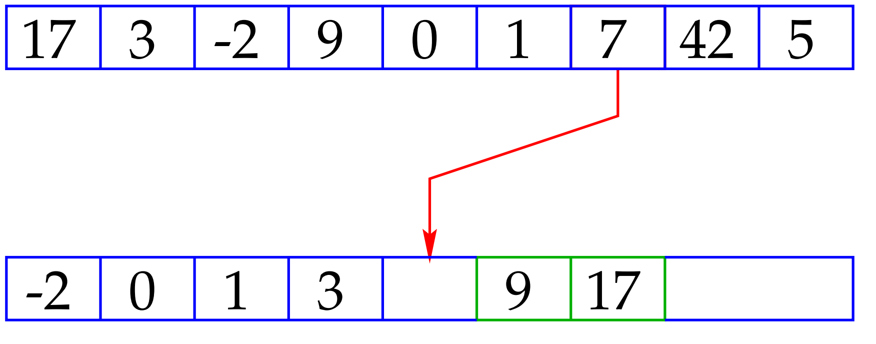

17 3 -2 9 0 1 7 42 5
-2 3 9 17

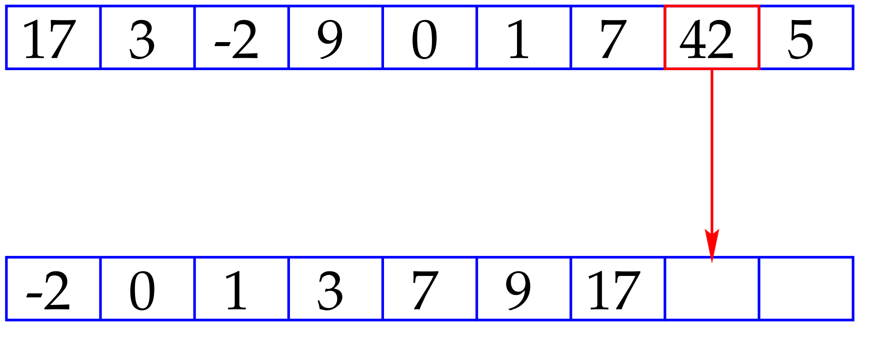

17
3
-2
9
0
1
7
42
5
-2
3
9
17

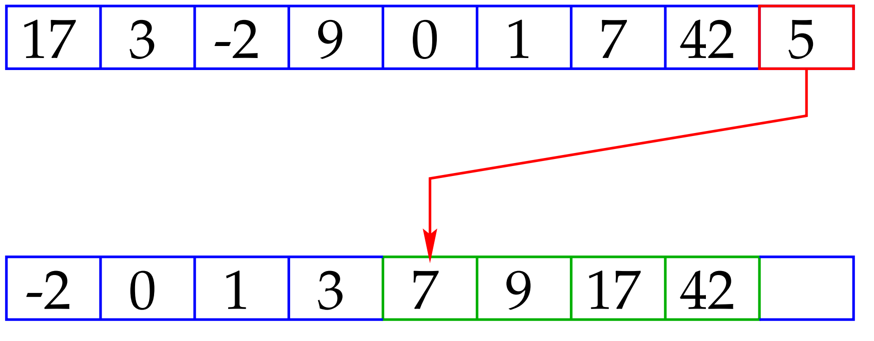

17
3
-2
9
0
1
7
42
5
-2
0
3
9
17

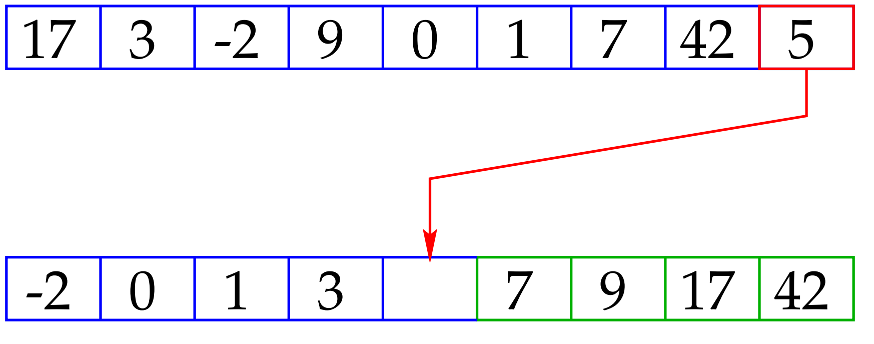

17 3 -2 9 0 1 7 42 5
-2 0 3 9 17


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).

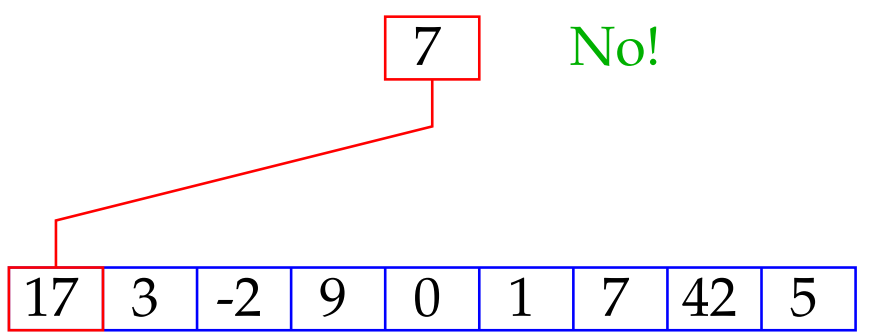
7 Eine zweite Anwendung: Suchen

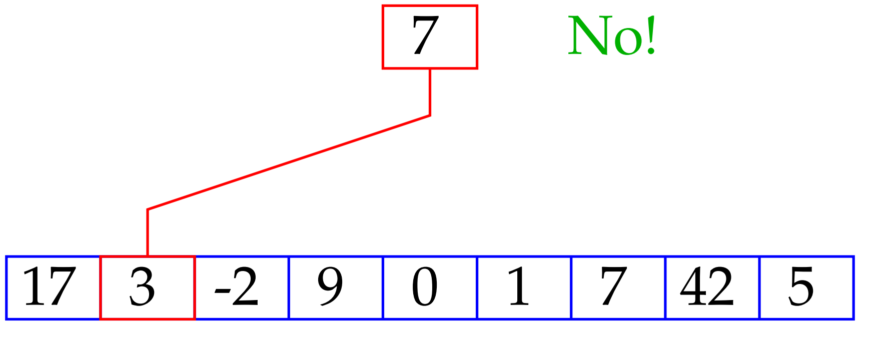
Nehmen wir an, wir wollen herausfinden, ob das Element 7 in unserem Feld `a` enthalten ist.

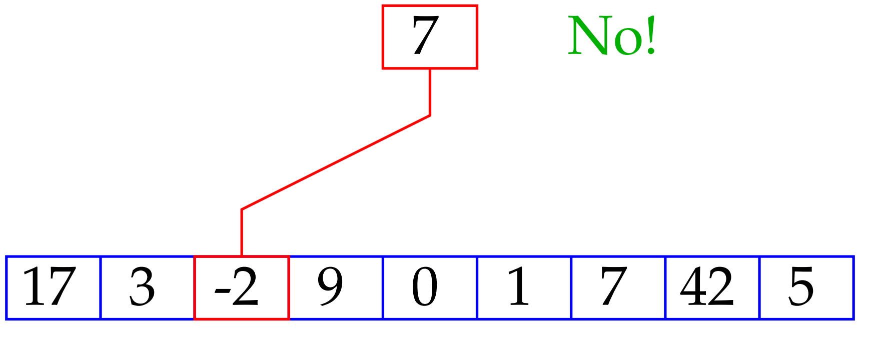
Naives Vorgehen:

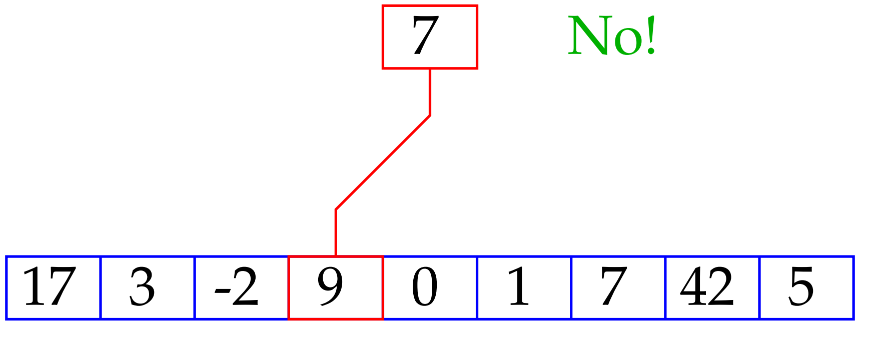
- Wir vergleichen 7 der Reihe nach mit den Elementen `a[0]`, `a[1]`, usw.
- Finden wir ein `i` mit `a[i] == 7`, geben wir `i` aus.
- Andernfalls geben wir `-1` aus: "Sorry, gibt's leider nicht :-("

```
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;
   }
```

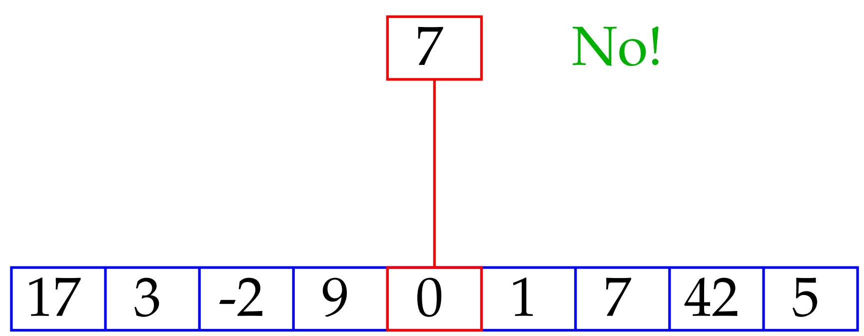

7
No!
17
3
-2
9
0
1
7
42
5

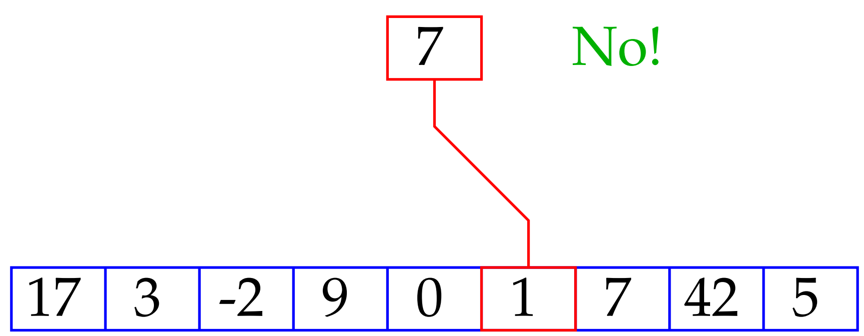

7
No!
17 3 -2 9 0 1 7 42 5

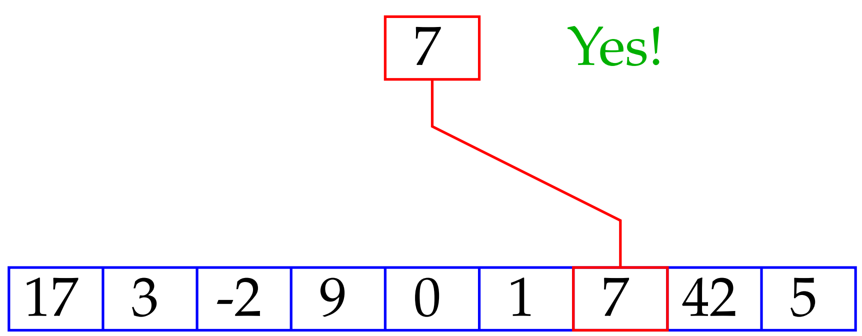

7
No!
17
3
-2
9
0
1
7
42
5



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

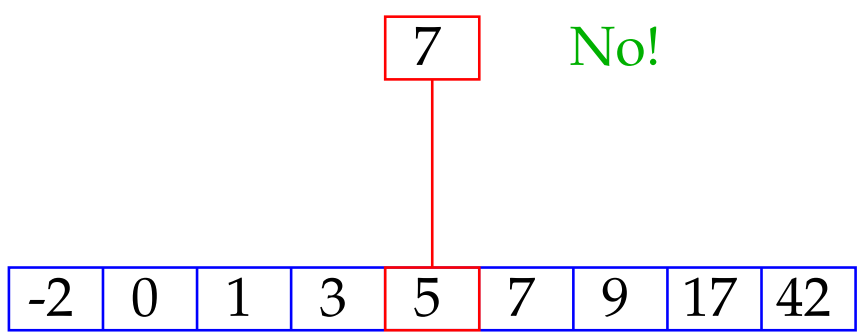
- Im Beispiel benötigen wir 7 Vergleiche.
- Im schlimmsten Fall benötigen wir bei einem Feld der Länge n sogar n Vergleiche :-(
- Kommt 7 tatsächlich im Feld vor, benötigen wir selbst im Durchschnitt $(n+1)/2$ viele Vergleiche :-((

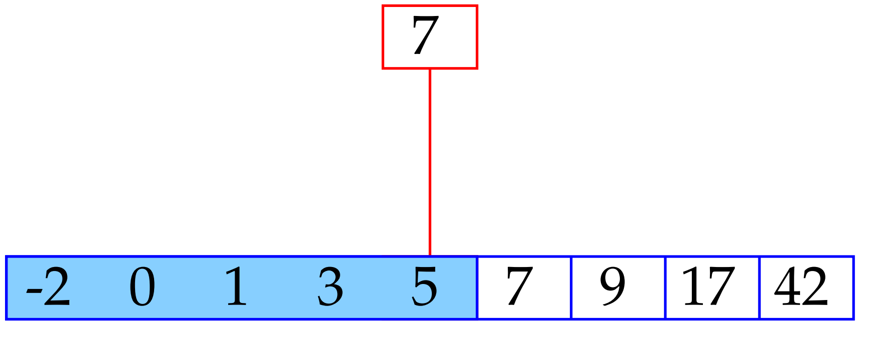
Geht das nicht besser ???

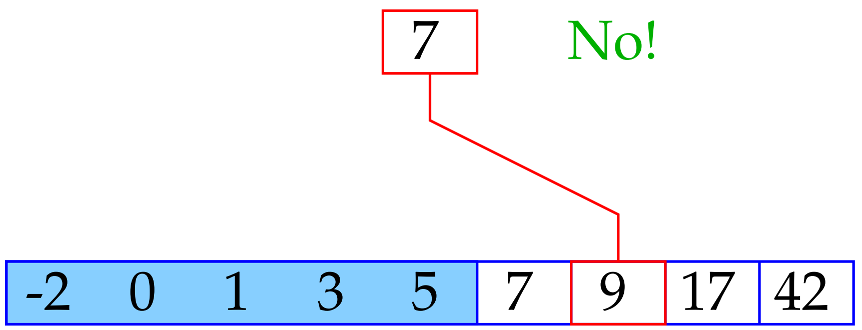
Idee:

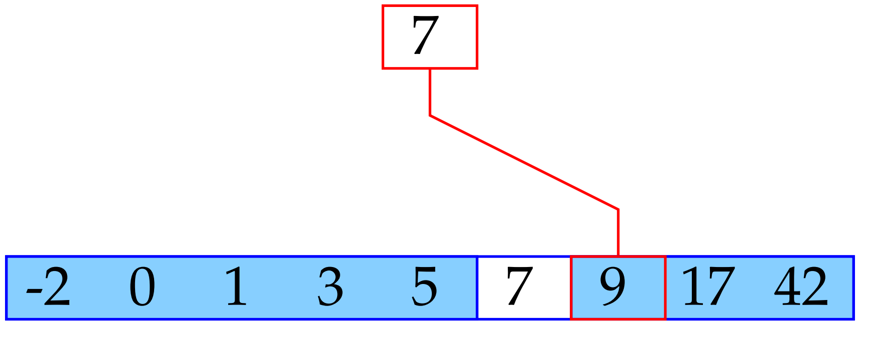
- Sortiere das Feld.
- Vergleiche 7 mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist 7 kleiner, brauchen wir nur noch links weitersuchen.
- Ist 7 größer, brauchen wir nur noch rechts weiter suchen.

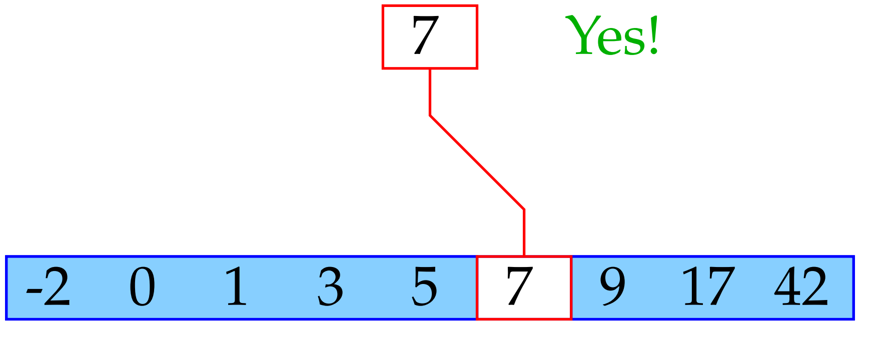
$\Longrightarrow$ binäre Suche ...


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

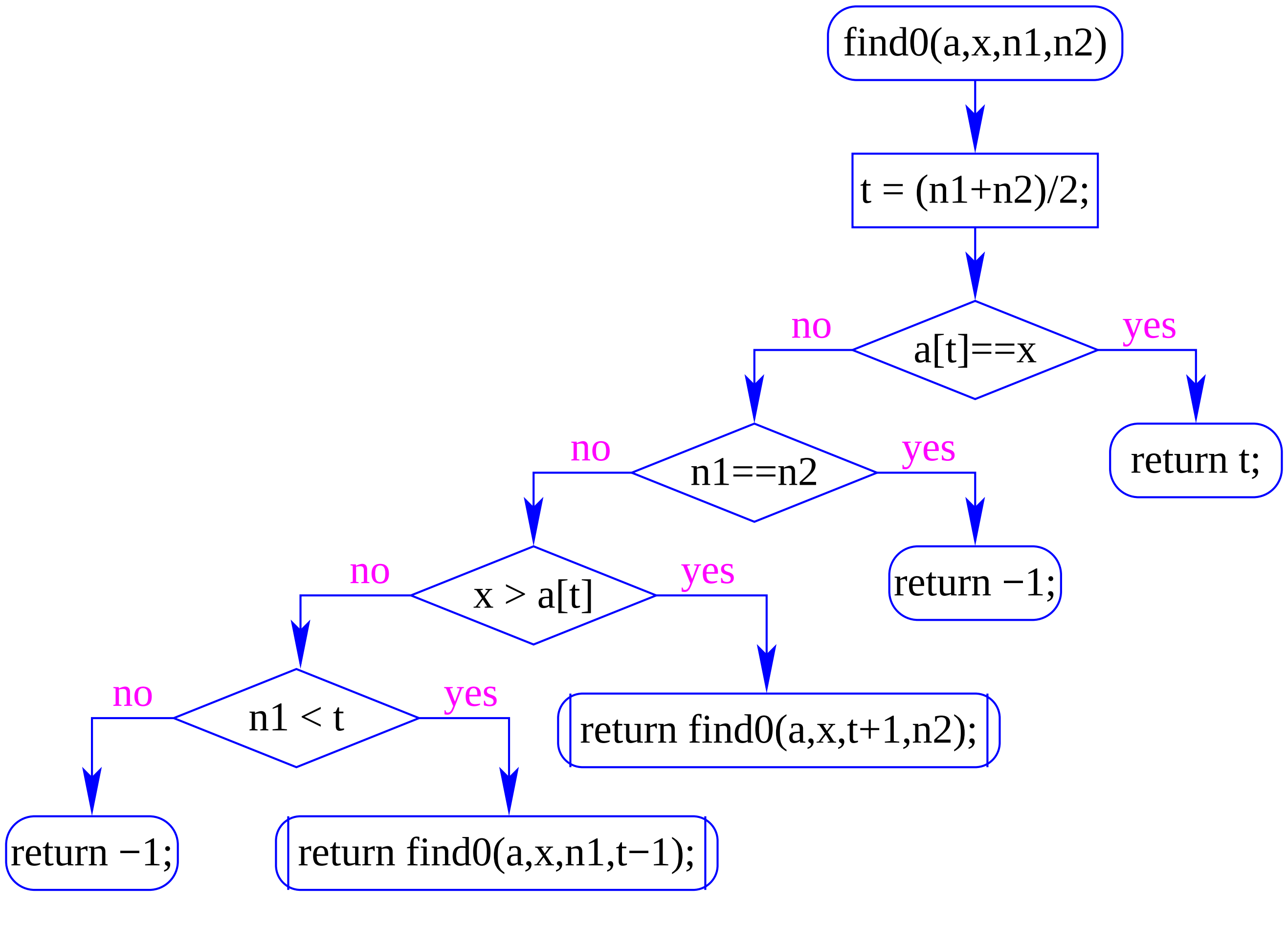
```
public static int find0 (int[] a, int x, int n1, int n2)
```

ein, die im Intervall $[\texttt{n1,n2}]$ 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()`:



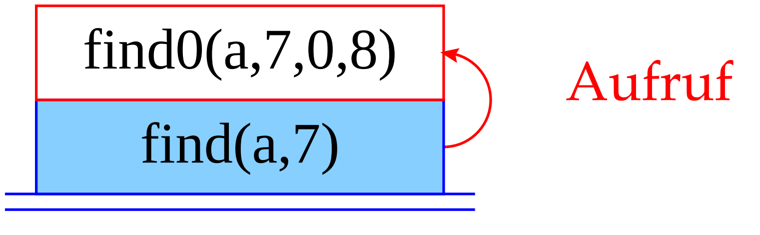
Achtung:

- 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.

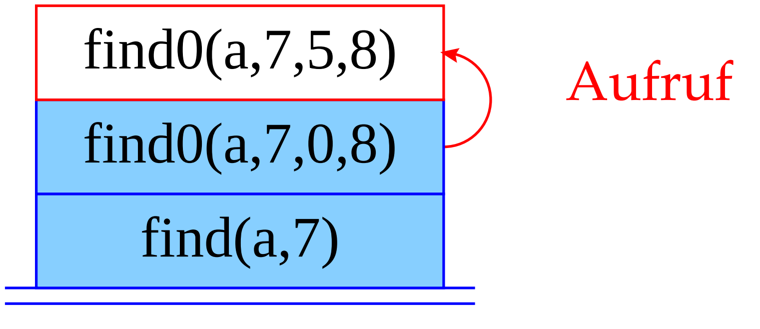
Ausführung:

find(a,7)

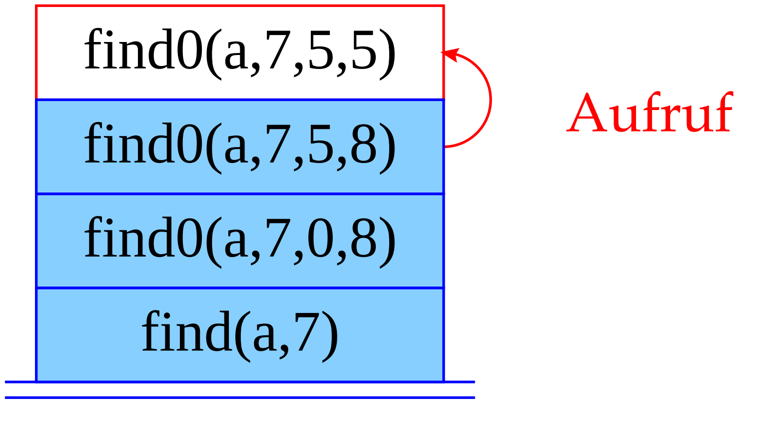
Ausführung:



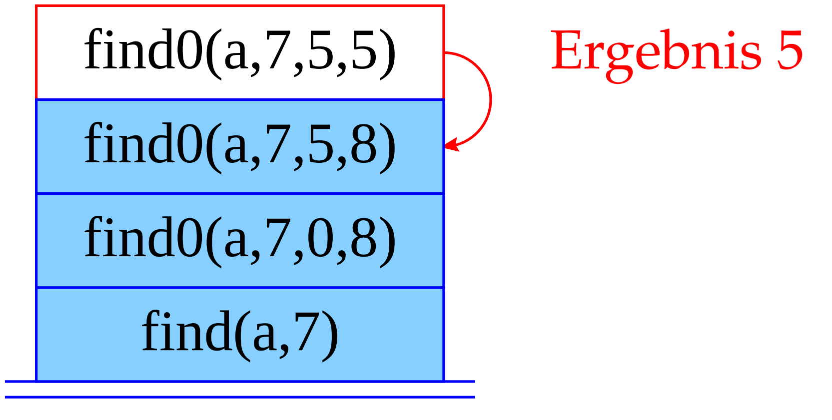
Ausführung:



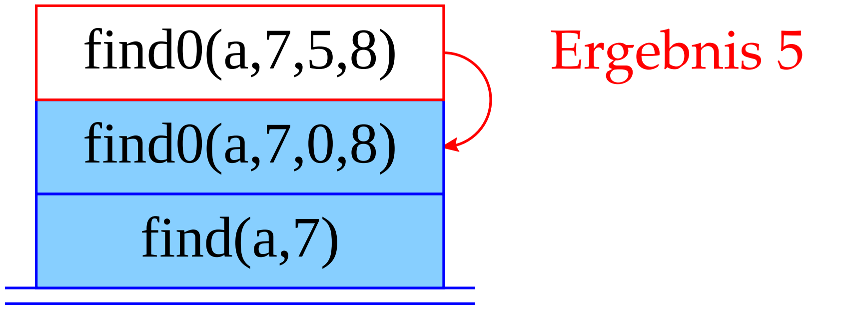
Ausführung:



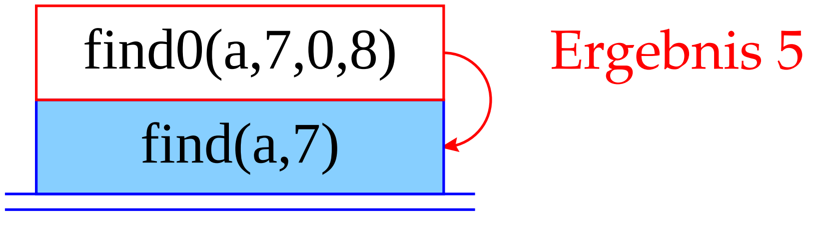
Ausführung:



Ausführung:



Ausführung:



Ausführung:

$\boxed{\text{find(a,7)}}$ 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.

Beobachtung:

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

Verbesserte Ausführung:

find(a,7)

Verbesserte Ausführung:

$\boxed{\text{find0(a,7,0,8)}}$

Verbesserte Ausführung:

$$\boxed{\text{find0(a,7,5,8)}}$$

Verbesserte Ausführung:

find0(a,7,5,5)

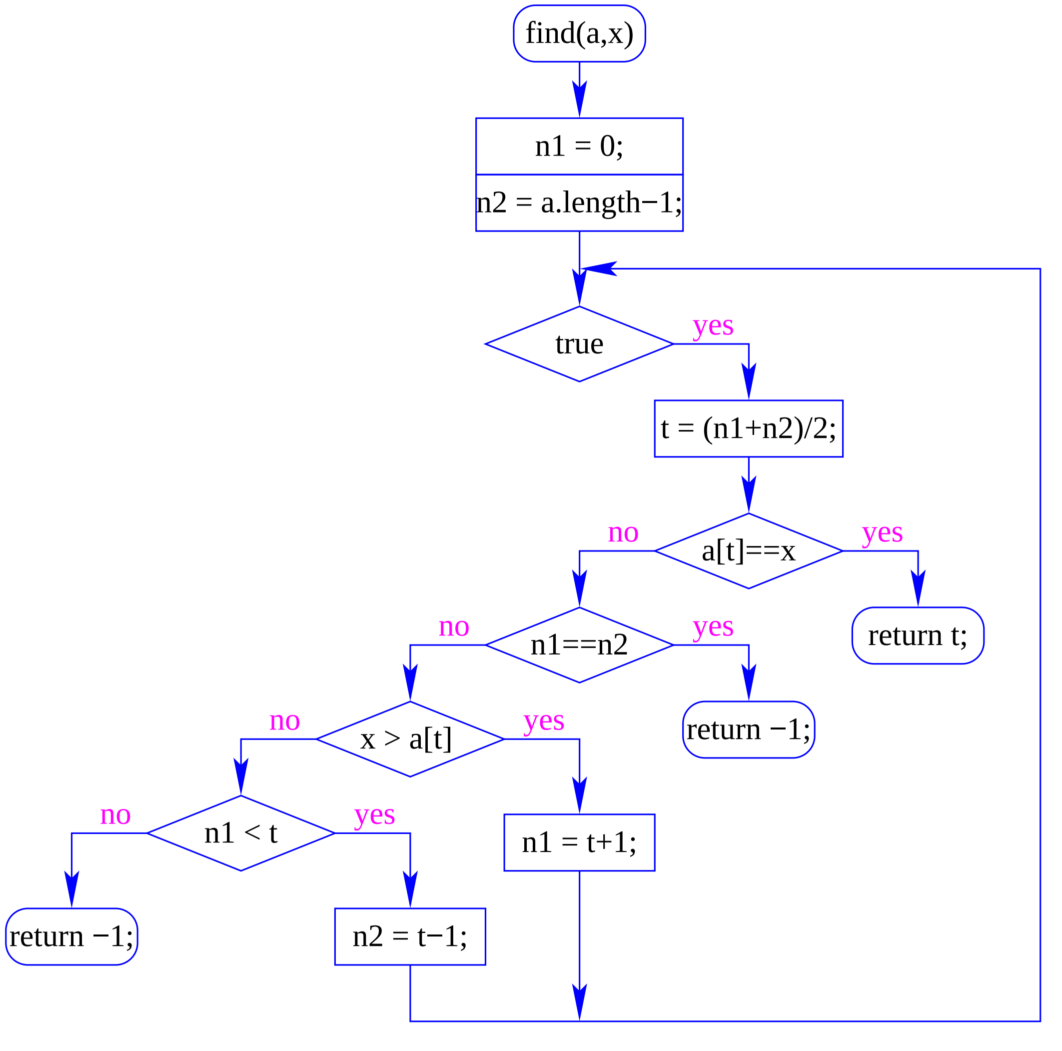
Verbesserte Ausführung:

$\boxed{\text{find0(a,7,5,5)}}$ Ergebnis: 5

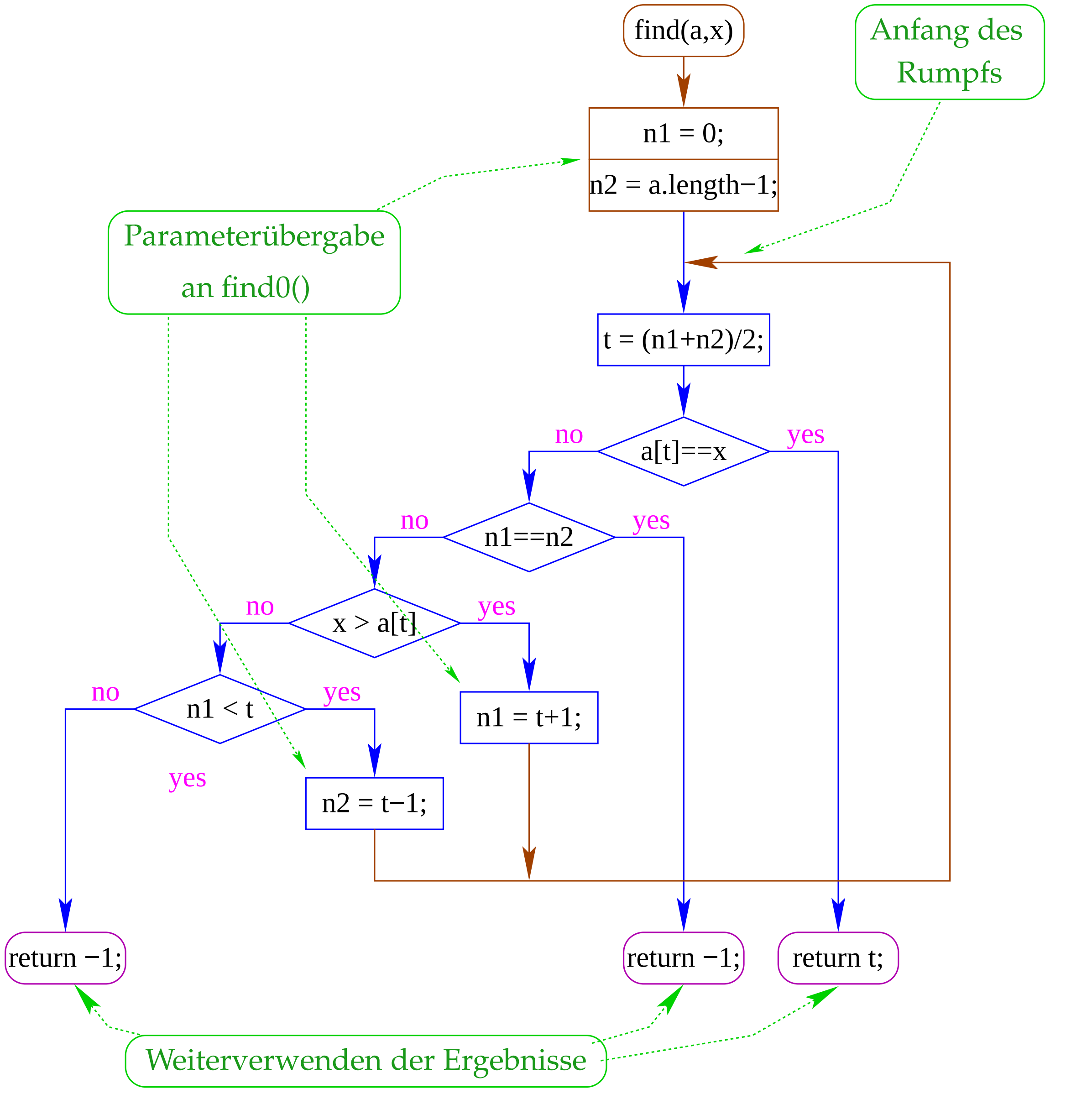
$\Longrightarrow$ end-Rekursion kann durch Iteration (d.h. eine normale Schleife) ersetzt werden ...

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

Das Kontrollfluss-Diagramm:



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

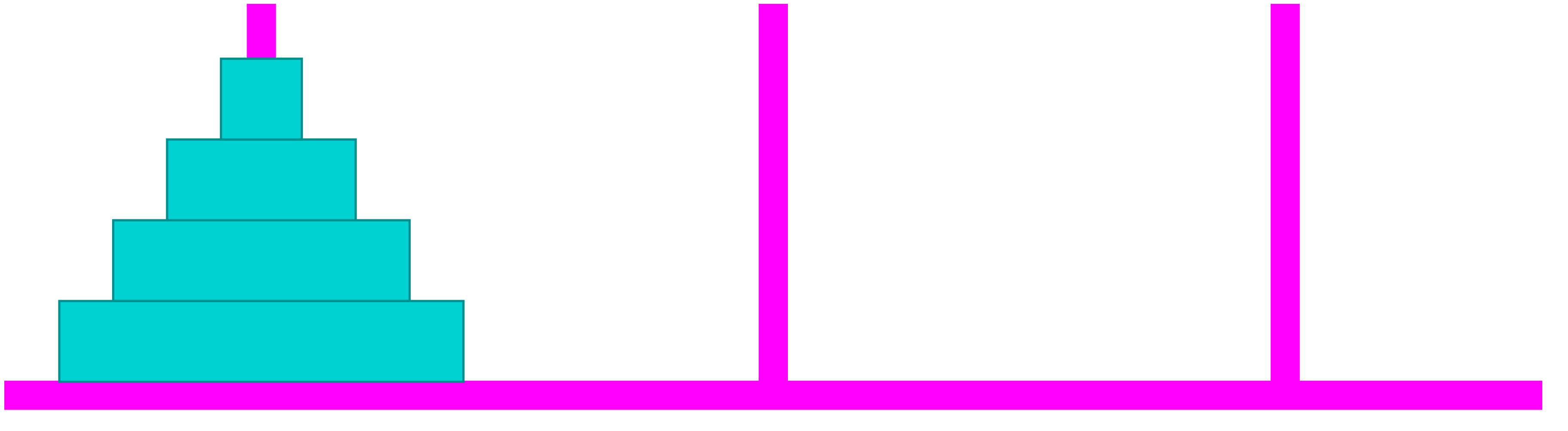

find(a,x)
Anfang des Rumpfs
n1 = 0;
n2 = a.length−1;
Parameterübergabe an find0()
t = (n1+n2)/2;
no
a[t]==x
yes
no
n1==n2
yes
no
x > a[t]
yes
no
n1 < t
yes
n1 = t+1;
yes
n2 = t−1;
return −1;
return −1;
return t;
Weiterverwenden der Ergebnisse

Bemerkung:

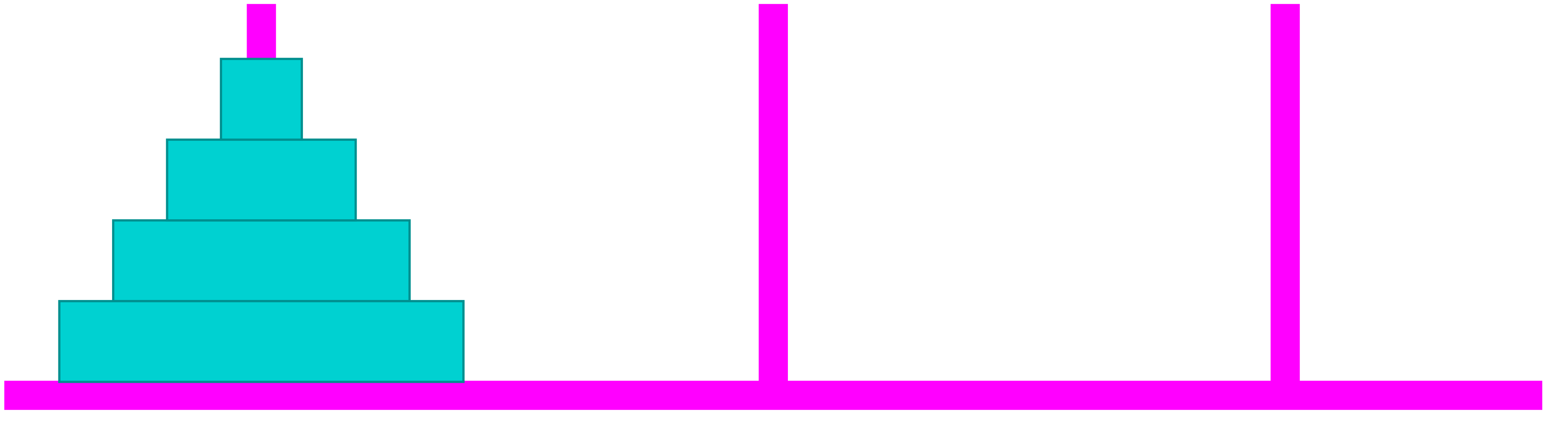
- Jede Rekursion lässt sich beseitigen, indem man den Aufruf-Keller explizit verwaltet.
- Nur im Falle von End-Rekursion kann man auf den Keller verzichten.
- Rekursion ist trotzdem nützlich, weil rekursive Programme oft leichter zu verstehen sind als äquivalente Programme ohne Rekursion ...

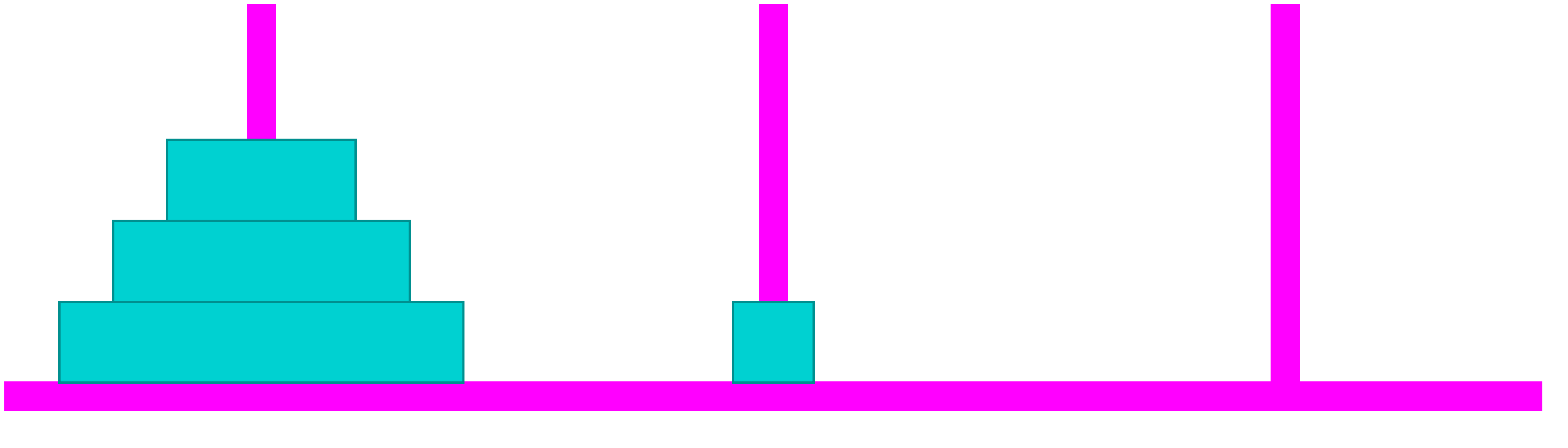
8 Die Türme von Hanoi

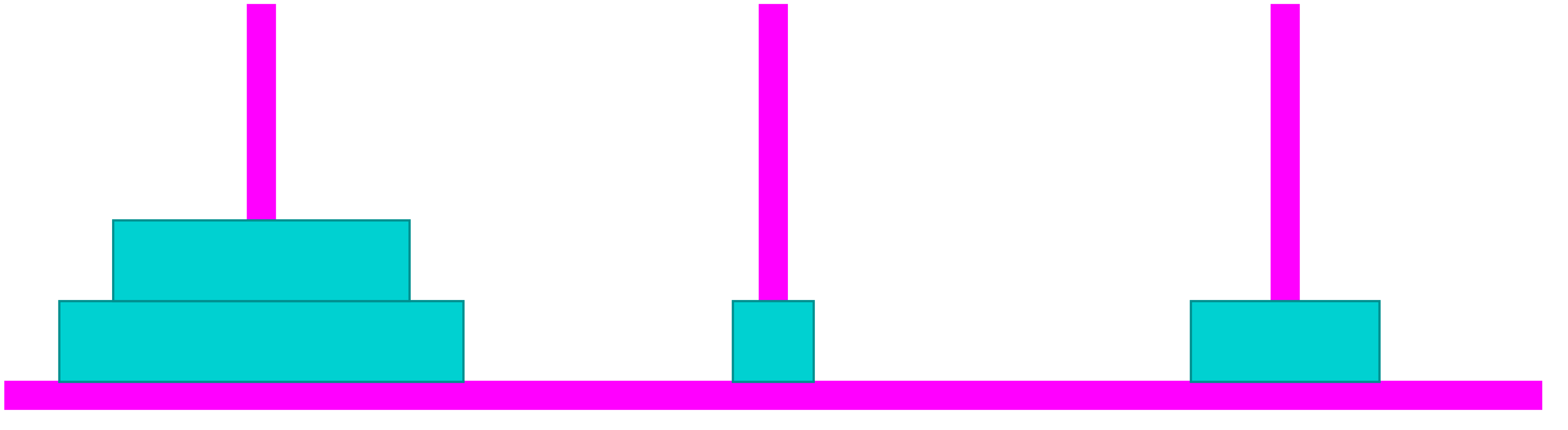
Problem:

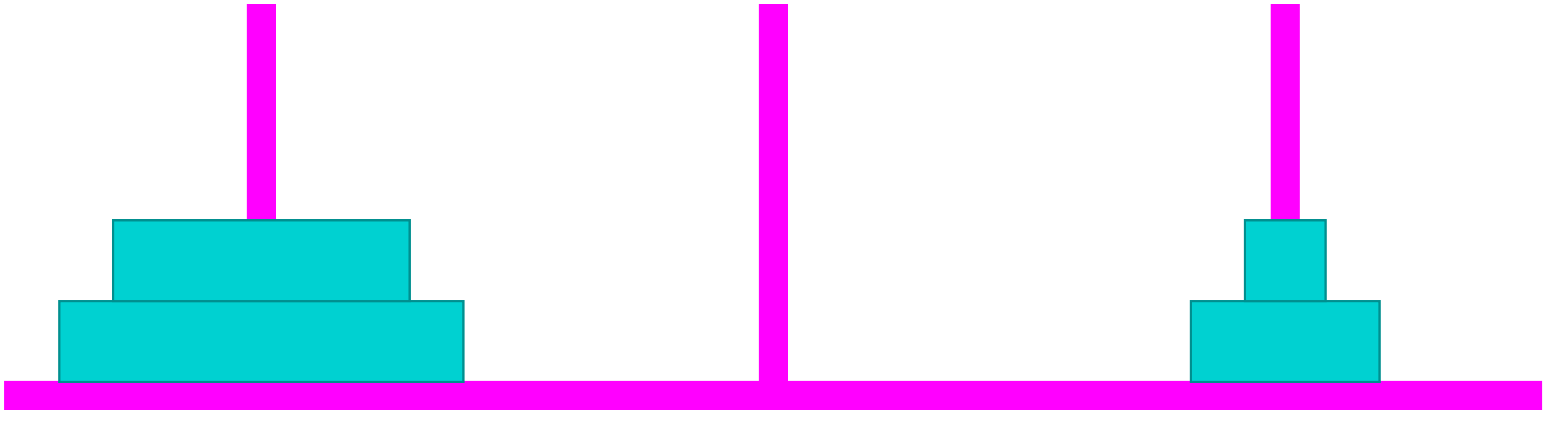


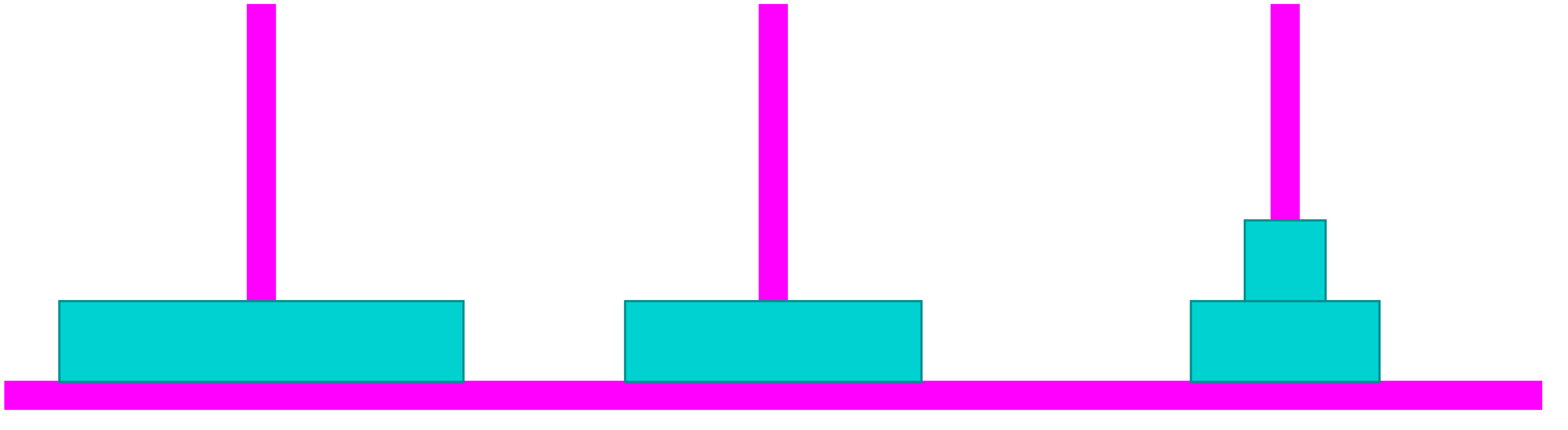
- Bewege den Stapel von links nach rechts!
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer Ring auf einen kleineren gelegt werden.

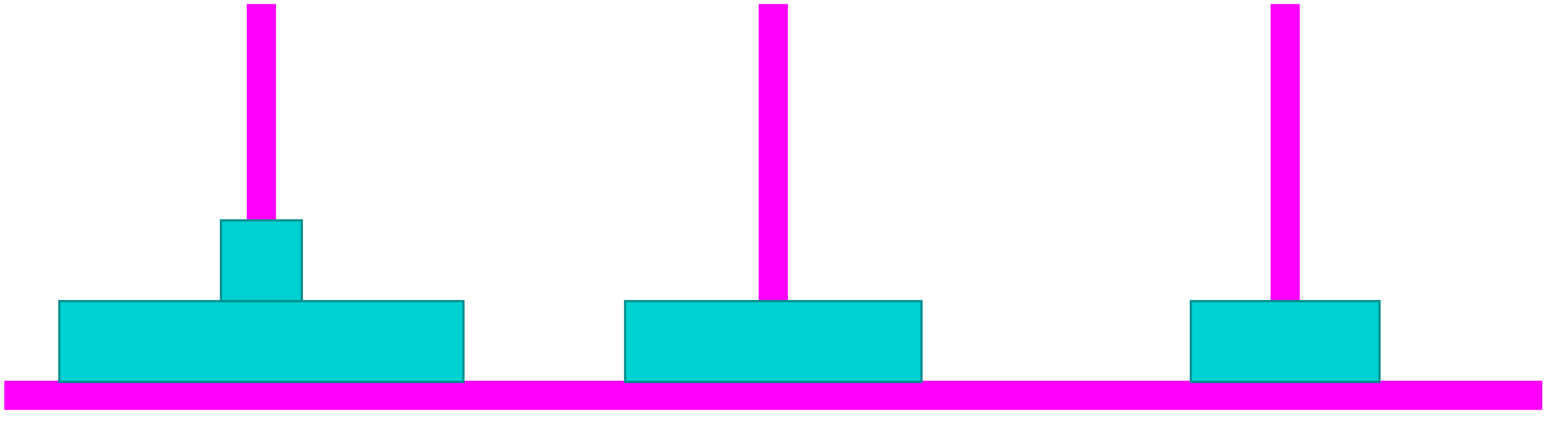


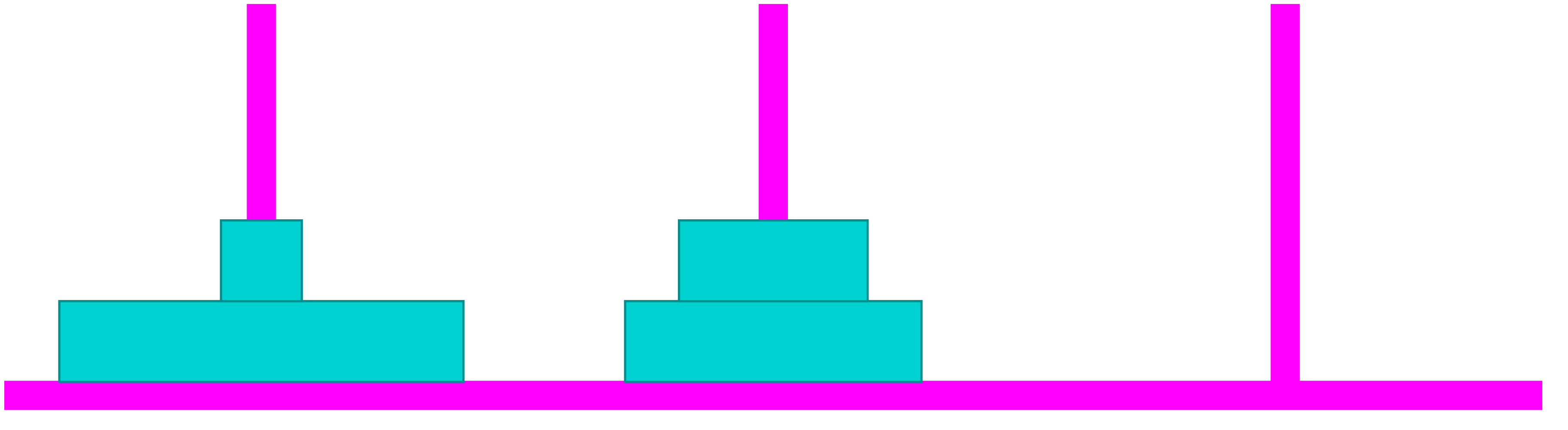


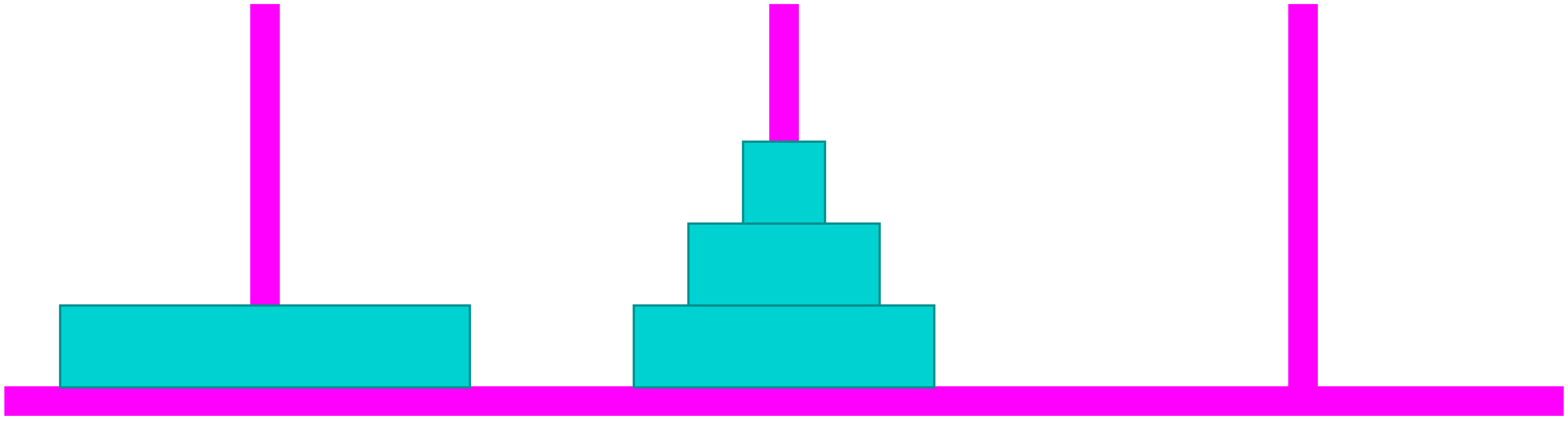


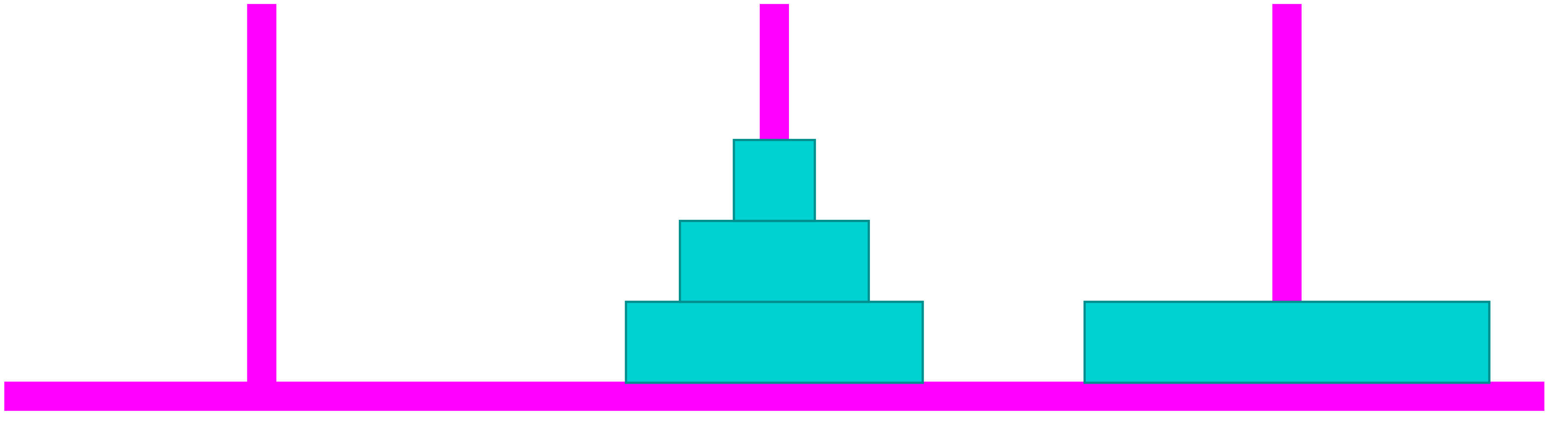


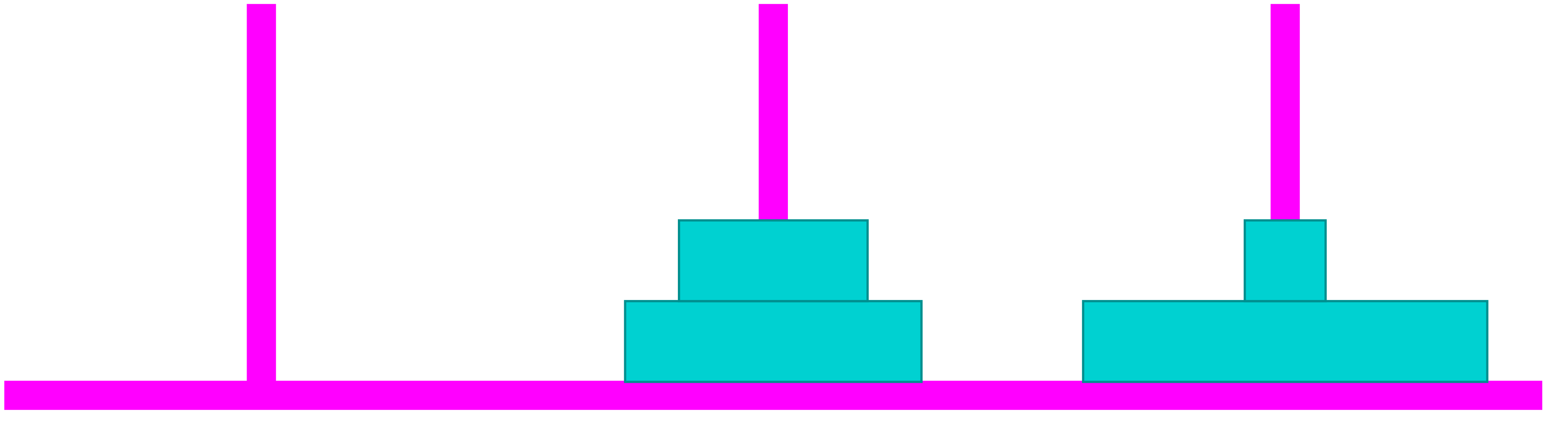


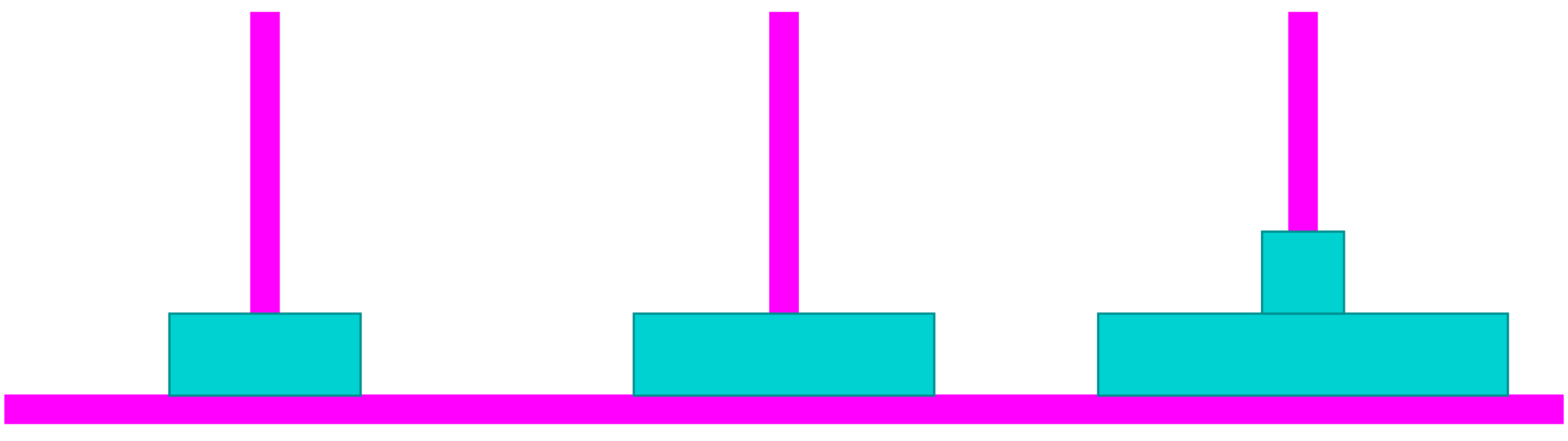


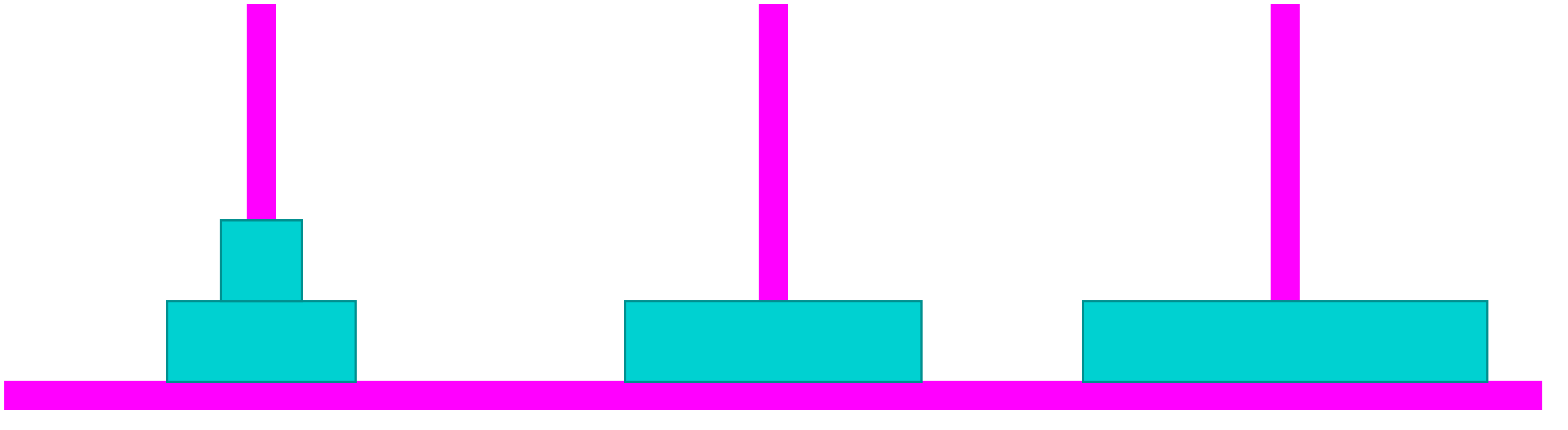


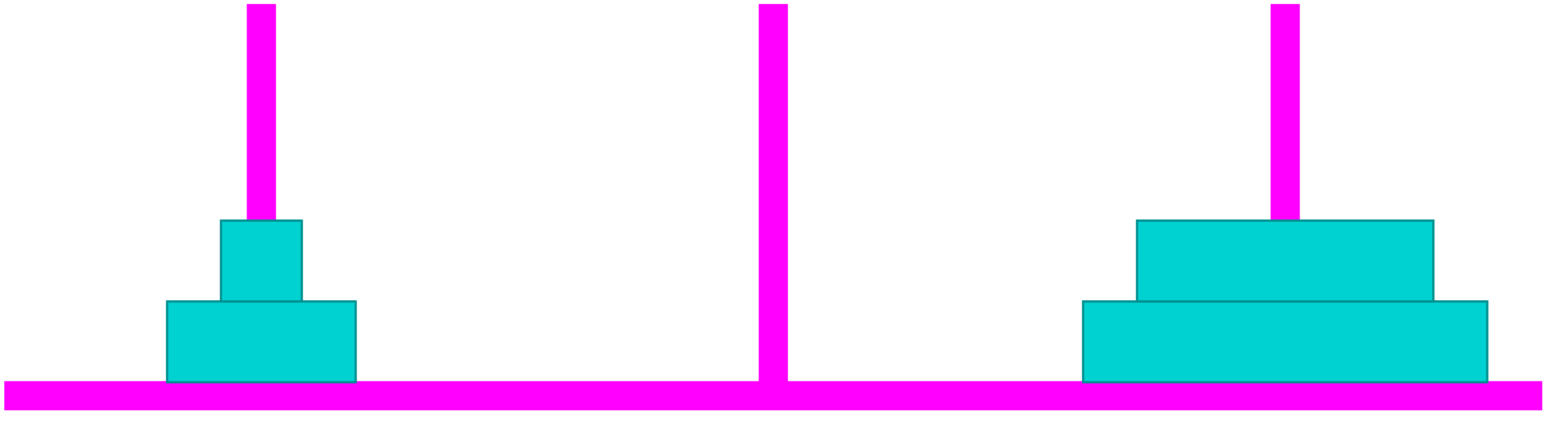


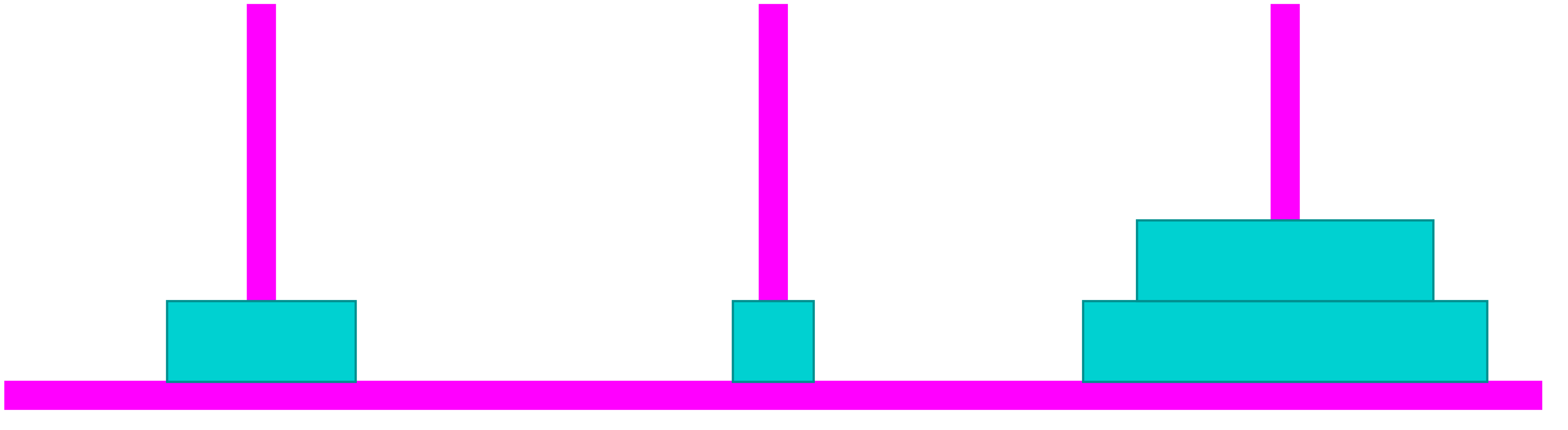


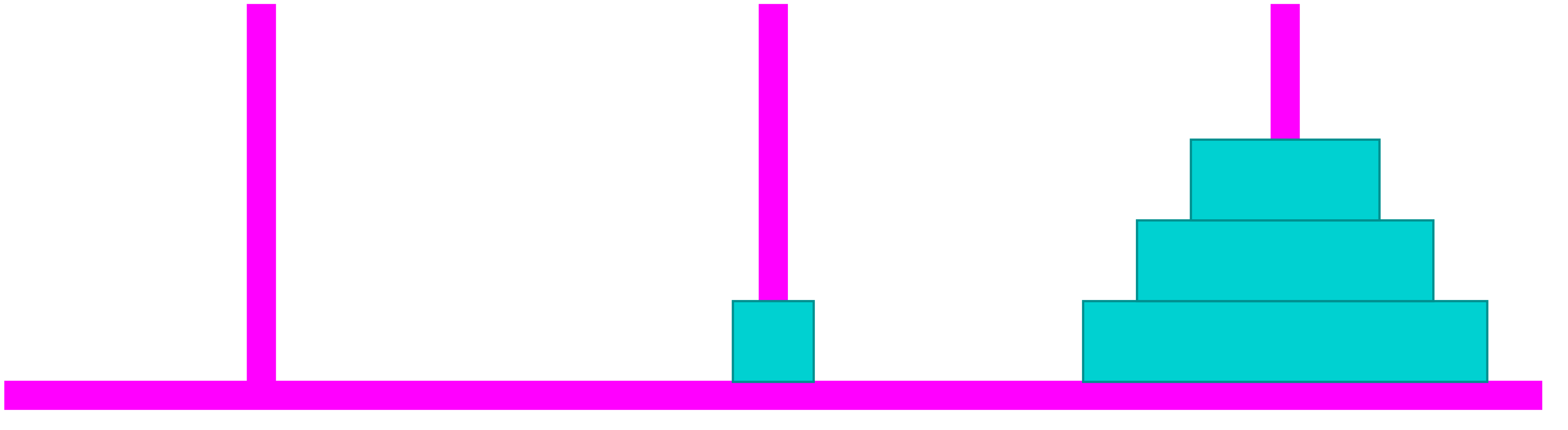


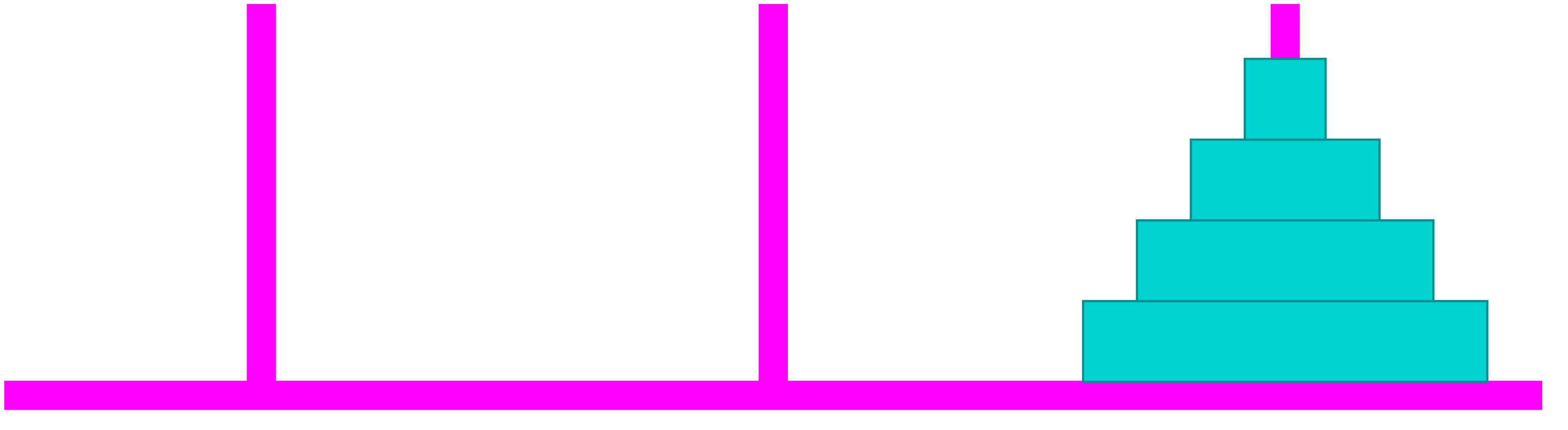












Idee:

- Versetzen eines Turms der Höhe $h = 0$ ist einfach: wir tun nichts.
- Versetzen eines Turms der Höhe $h > 0$ von Position a nach Position b zerlegen wir in drei Teilaufgaben:

  1. Versetzen der oberen $h - 1$ Scheiben auf den freien Platz;
  2. Versetzen der untersten Scheibe auf die Zielposition;
  3. Versetzen der zwischengelagerten Scheiben auf die Zielposition.

- Versetzen eines Turms der Höhe $h > 0$ erfordert also zweimaliges Versetzen eines Turms der Höhe $h - 1$.

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

Bleibt die Ermittlung des freien Platzes ...

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

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

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

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

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

Allgemeine Form eines `switch`-Statements:

$$
\begin{array}{l}
\texttt{switch (}\ \mathit{expr}\ \texttt{) \{} \\
\texttt{case}\ \mathrm{const}_0\ \texttt{:} \quad \mathrm{ss}_0\ (\ \texttt{break;}\ )\,? \\
\texttt{case}\ \mathrm{const}_1\ \texttt{:} \quad \mathrm{ss}_1\ (\ \texttt{break;}\ )\,? \\
\quad\quad \ldots \\
\texttt{case}\ \mathrm{const}_{k-1}\ \texttt{:} \quad \mathrm{ss}_{k-1}\ (\ \texttt{break;}\ )\,? \\
(\ \texttt{default:} \quad \mathrm{ss}_k\ )\,? \\
\texttt{\}}
\end{array}
$$

- expr sollte eine ganze Zahl (oder ein `char`) sein.
- Die const_i sind ganz-zahlige Konstanten.
- Die ss_i sind die alternativen Statement-Folgen.

- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren :-)

- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren :-)

Eine einfachere Lösung in unserem Fall ist :

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

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

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

Bemerkungen:

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

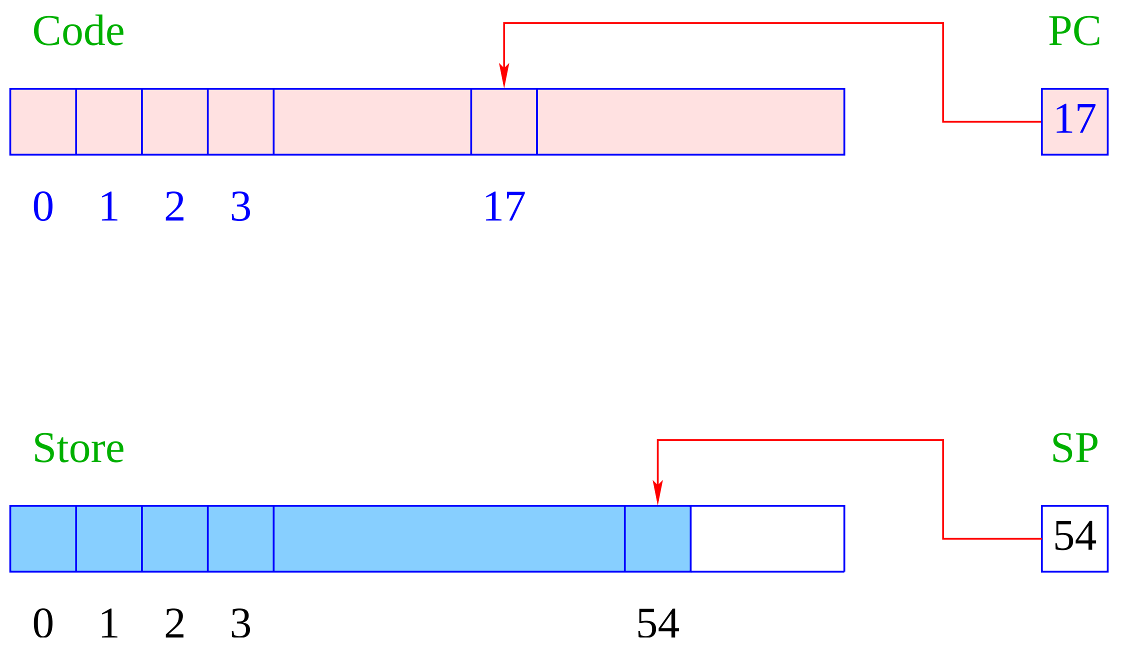
$$\begin{aligned} N(0) &= 0 \qquad\qquad \text{und für } h > 0, \\ N(h) &= 1 + 2 \cdot N(h-1) \end{aligned}$$

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

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

9 Von MiniJava zur JVM

Architektur der JVM:

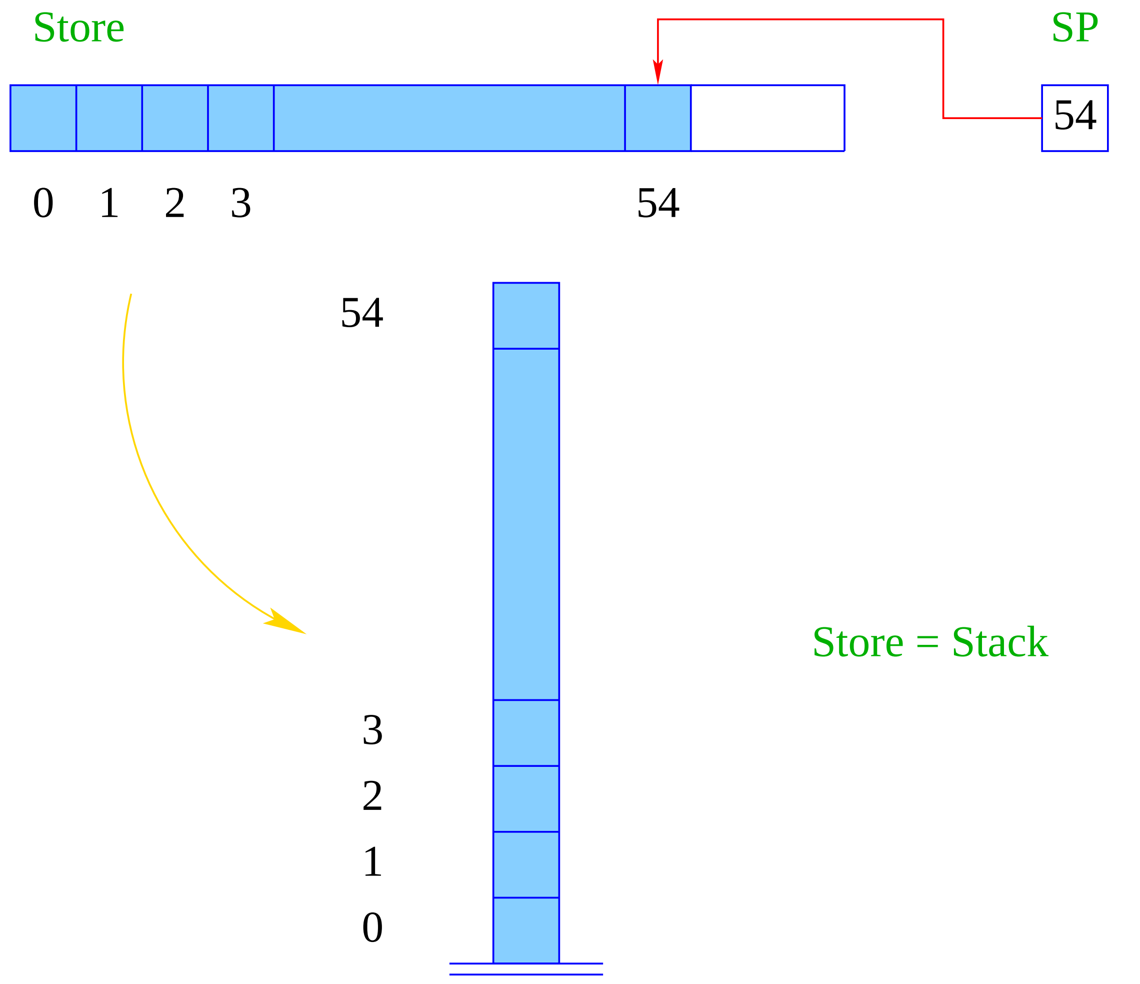


Code	=	enthält JVM-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter –
		zeigt auf nächsten auszuführenden Befehl;
Store	=	Speicher für Daten;
		jede Zelle kann einen Wert aufnehmen;
SP	=	Stack-Pointer –
		zeigt auf oberste belegte Zelle.

Achtung:

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

Konvention:



Befehle der JVM:

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

Ein Beispiel-Programm:

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

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

$$\begin{array}{lcr} \mathsf{A} & = & 5 \\ \mathsf{B} & = & 18 \\ \mathsf{C} & = & 22 \\ \mathsf{D} & = & 23 \end{array}$$

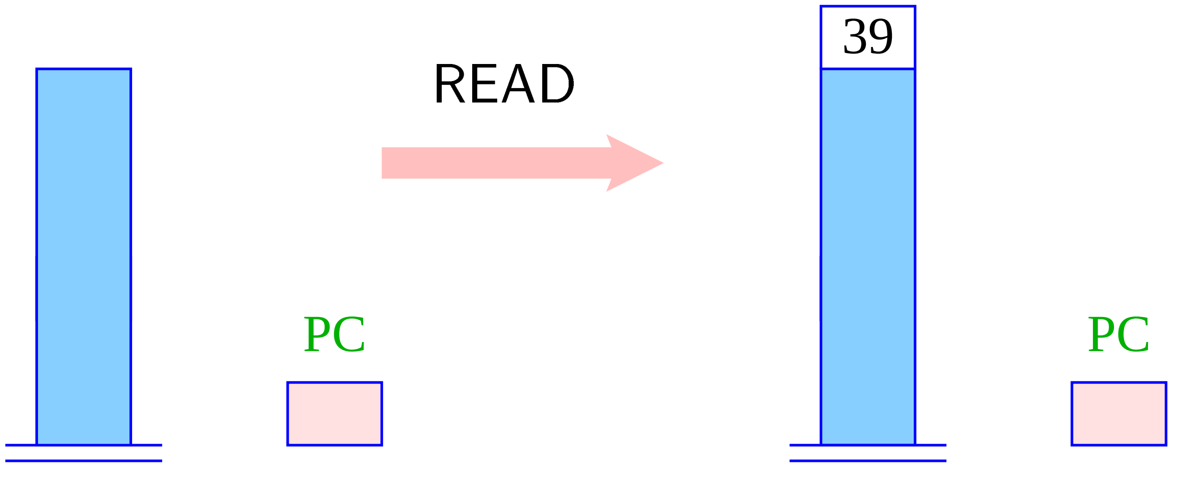
- ... können vom Compiler leicht in die entsprechenden Adressen umgesetzt werden (wir benutzen sie aber, um uns besser im Programm zurechtzufinden :-)

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

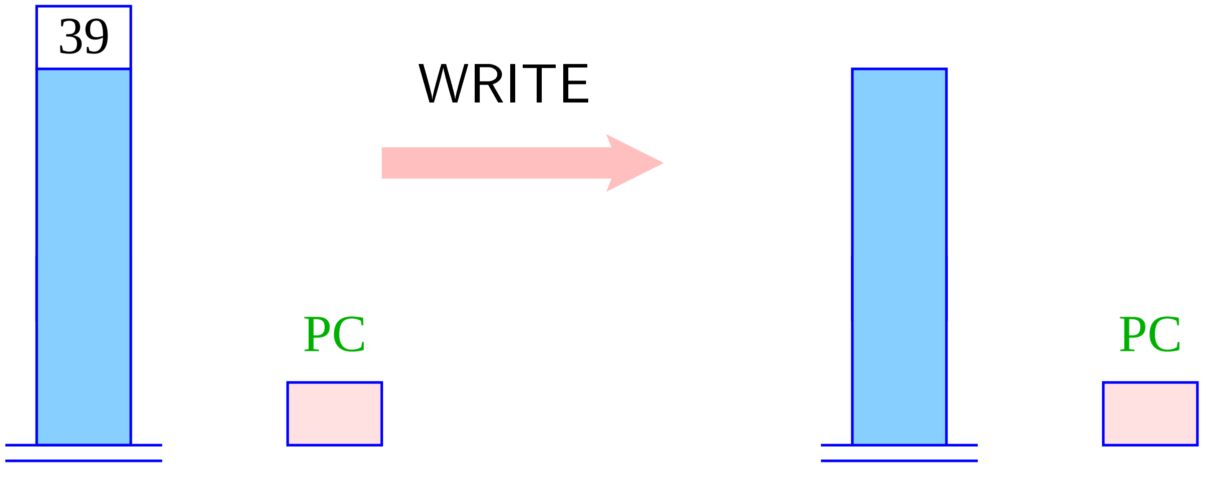
Idee:

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

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



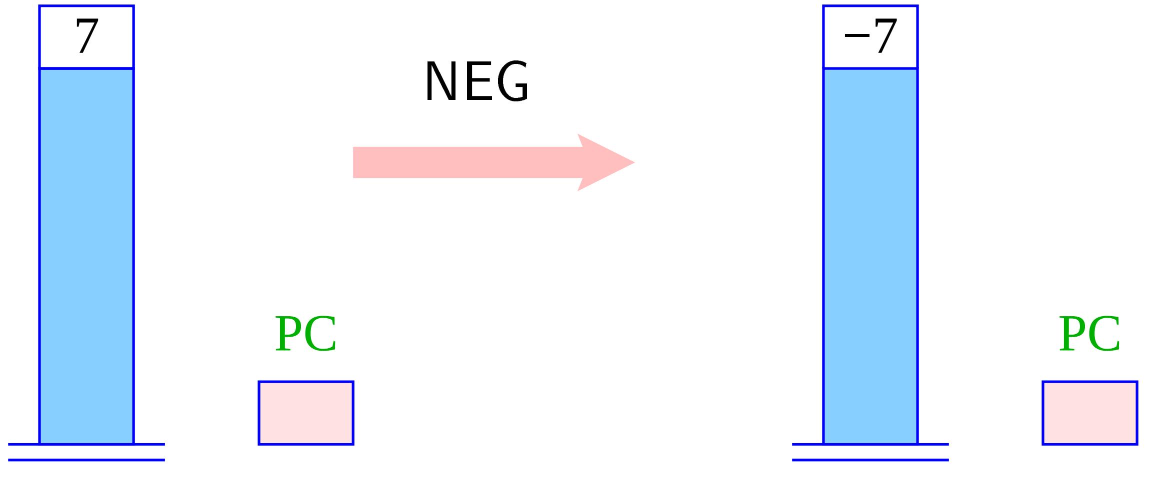
... falls 39 eingegeben wurde

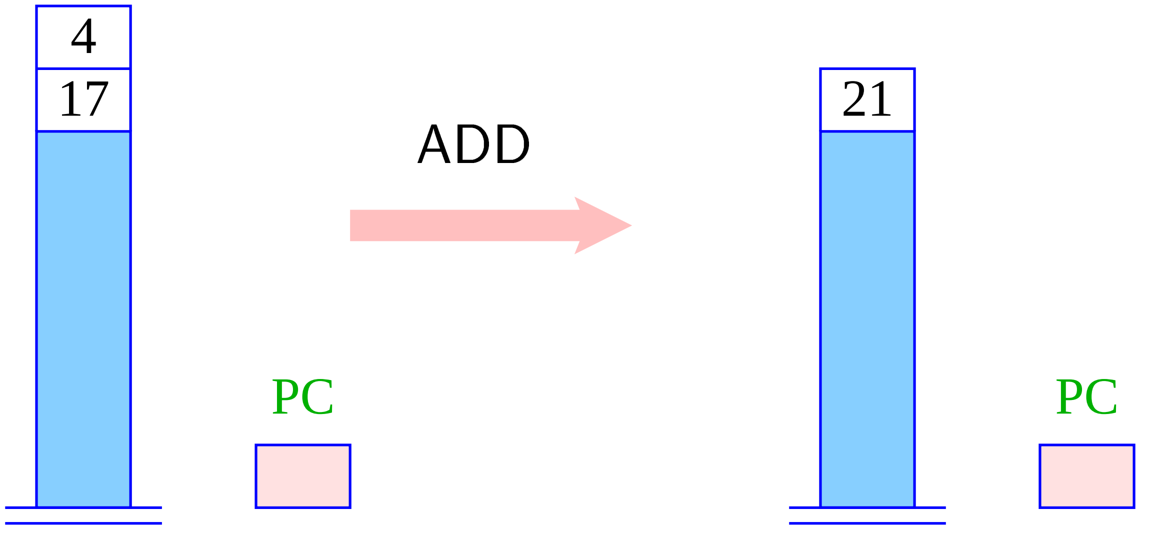


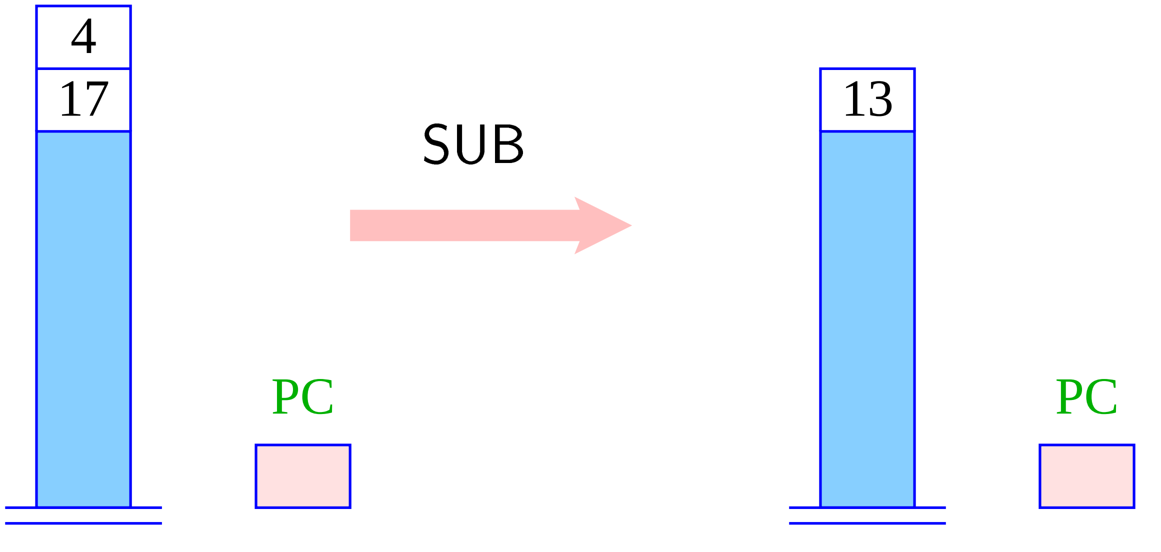
... wobei 39 ausgegeben wird

Arithmetik

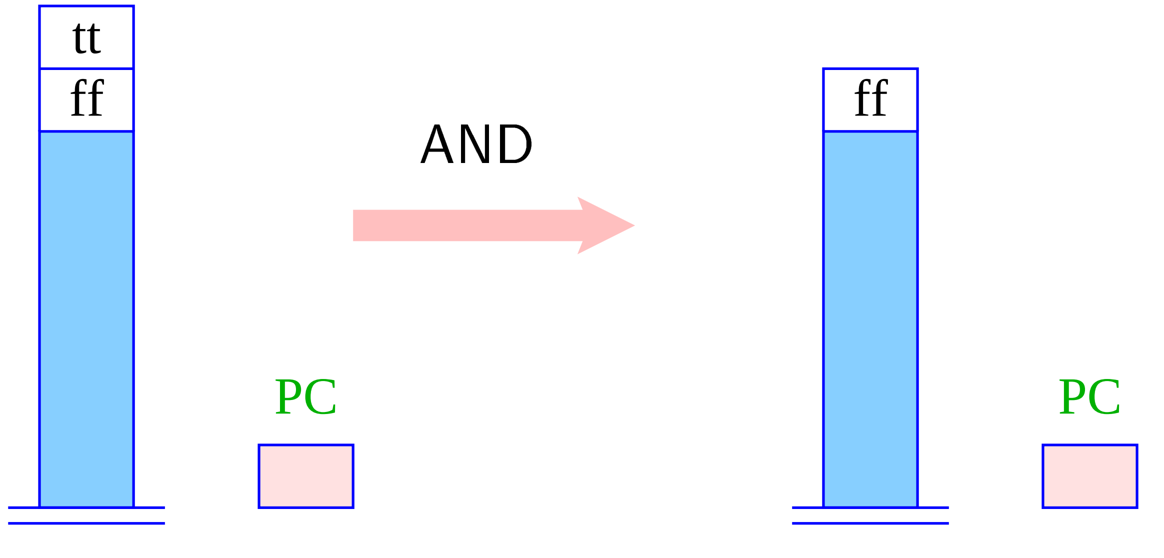
- Unäre Operatoren modifizieren die oberste Zelle.
- Binäre Operatoren verkürzen den Stack.

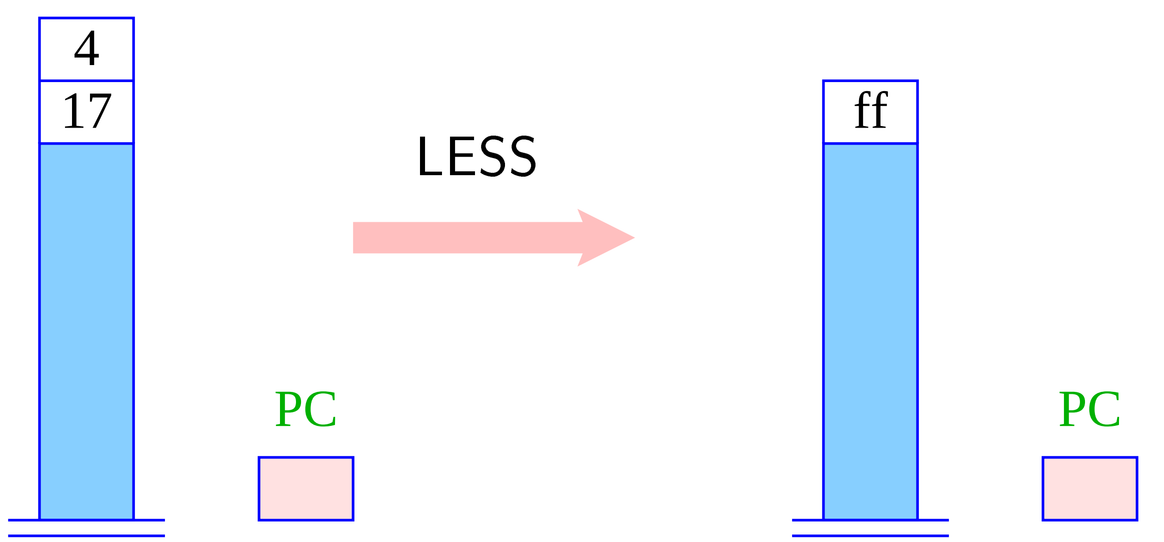

7
NEG
−7
PC
PC


4
17
ADD
21
PC
PC


4
17
SUB
13
PC
PC

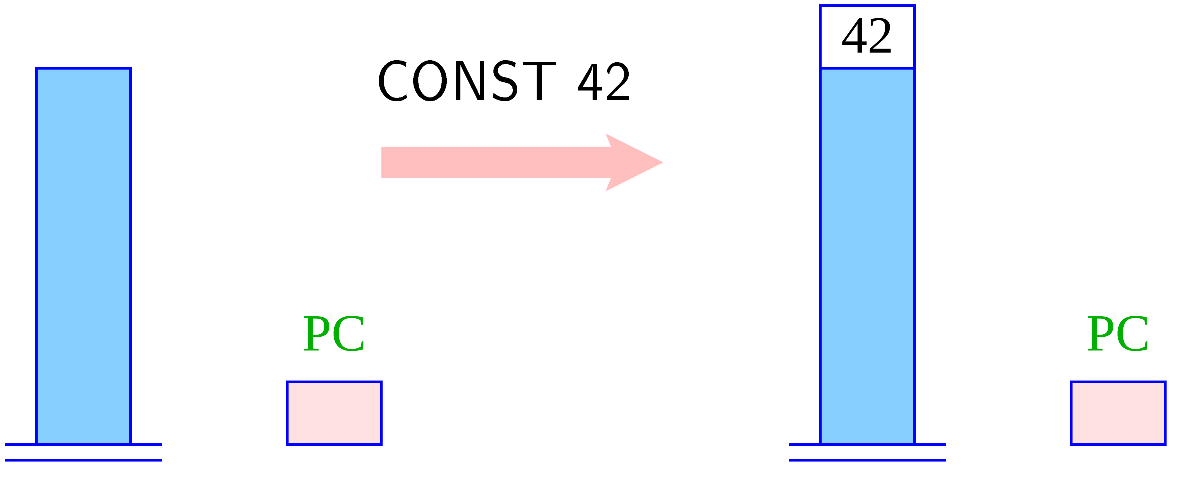
- Die übrigen arithmetischen Operationen MUL, DIV, MOD funktionieren völlig analog.
- Die logischen Operationen NOT, AND, OR ebenfalls – mit dem Unterschied, dass sie statt mit ganzen Zahlen, mit Intern-Darstellungen von `true` und `false` arbeiten (hier: “tt” und “ff”).
- Auch die Vergleiche arbeiten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

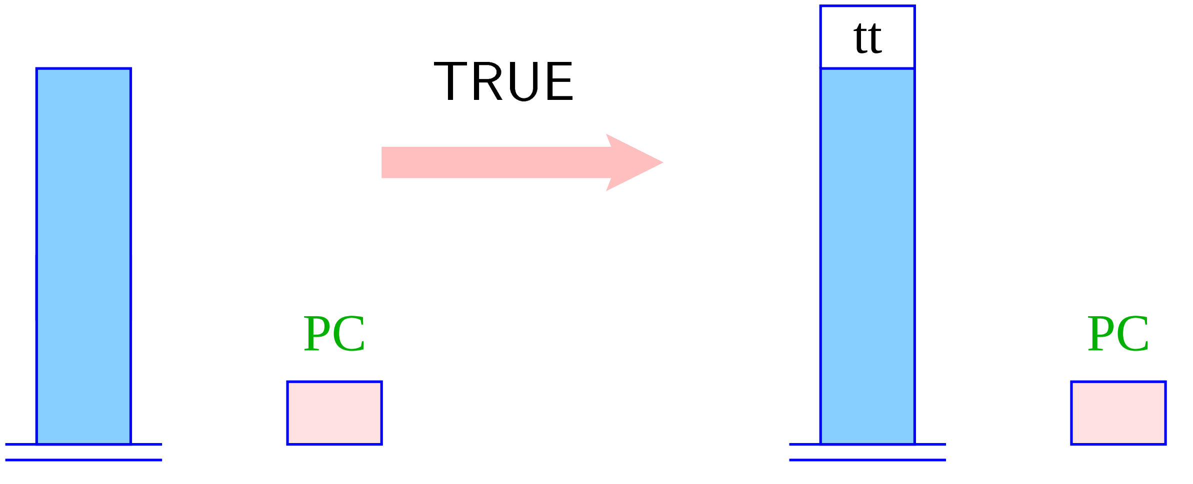

tt
ff
AND
ff
PC
PC

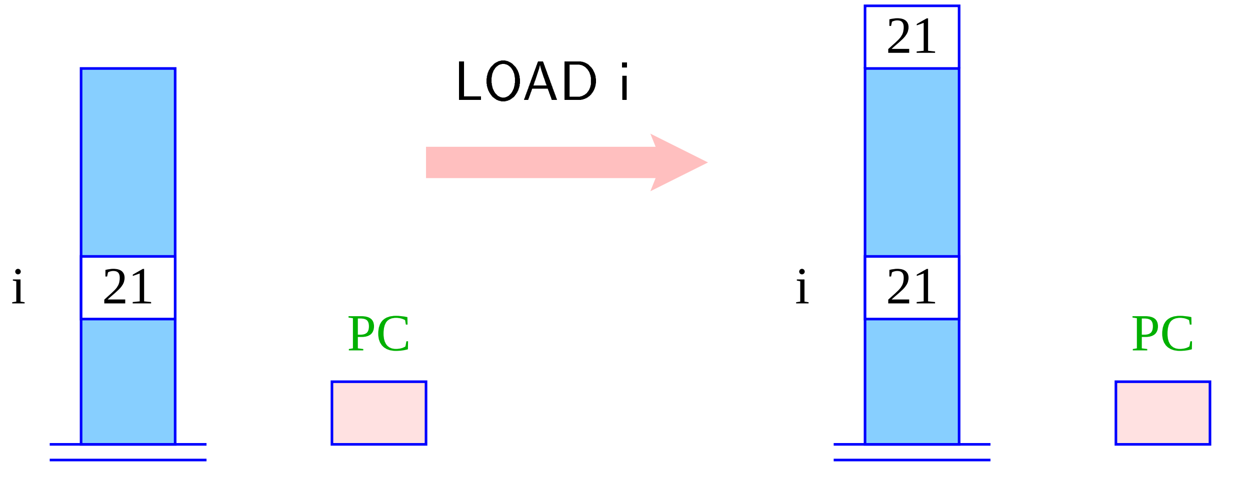

4
17
LESS
ff
PC
PC

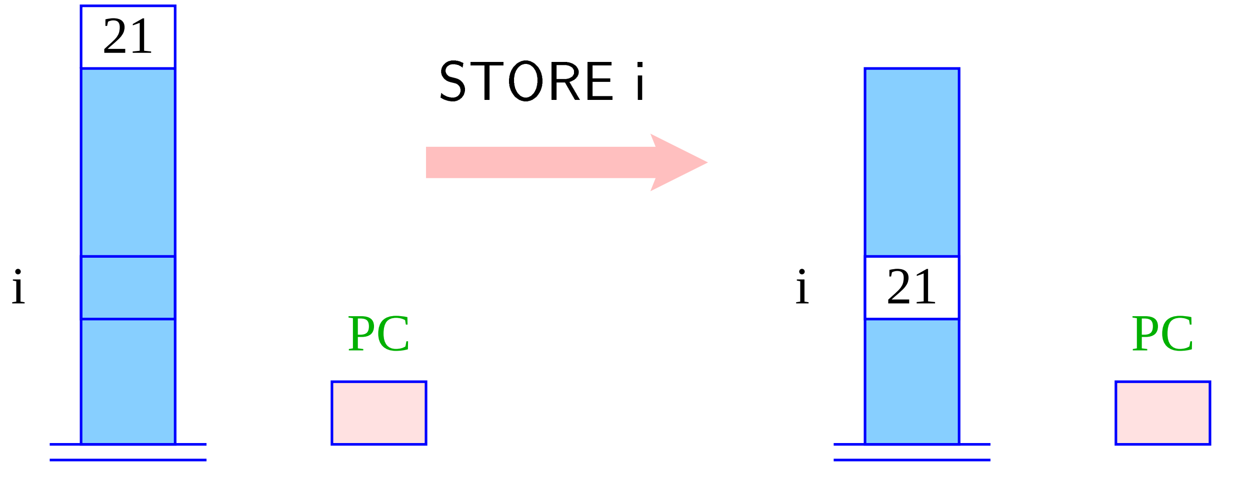
Laden und Speichern

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


CONST 42
42
PC
PC

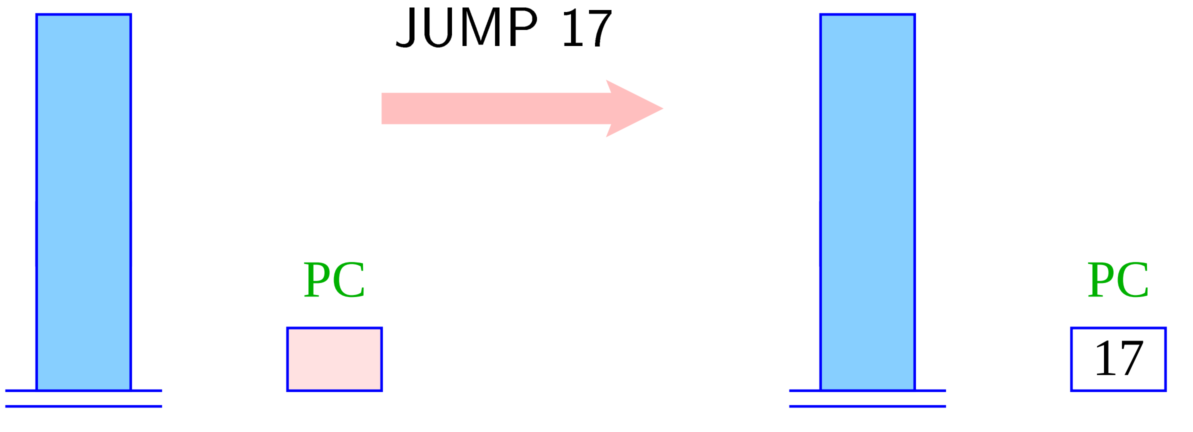

TRUE
tt
PC
PC

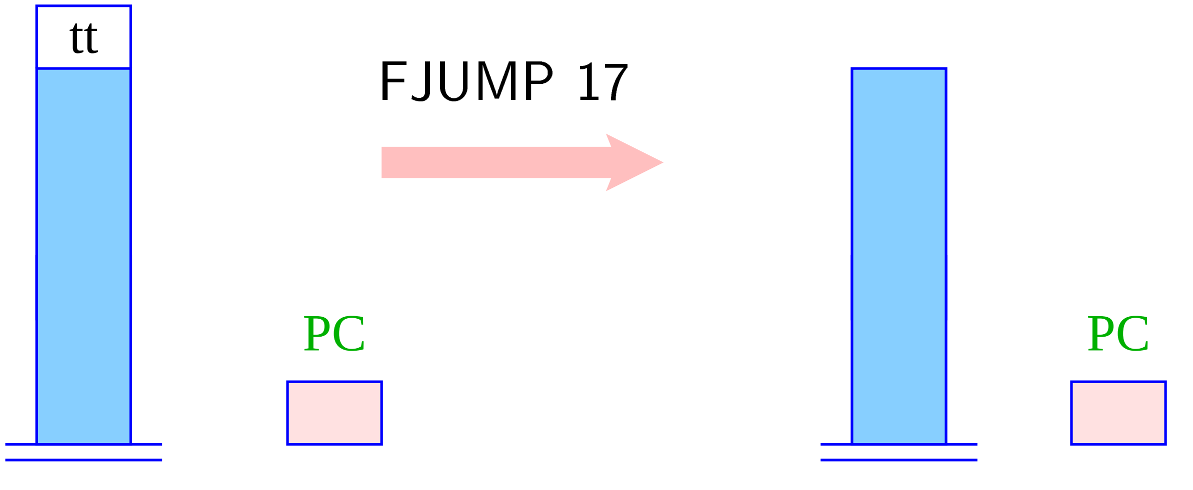

LOAD i
i
21
PC
21
i
21
PC

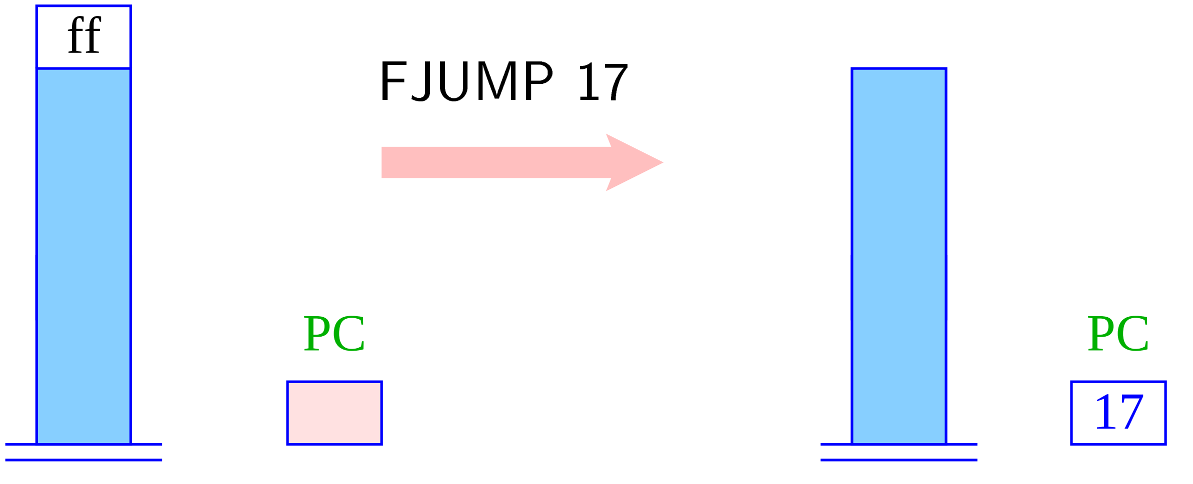

21
i
PC
STORE i
i
21
PC

Sprünge

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

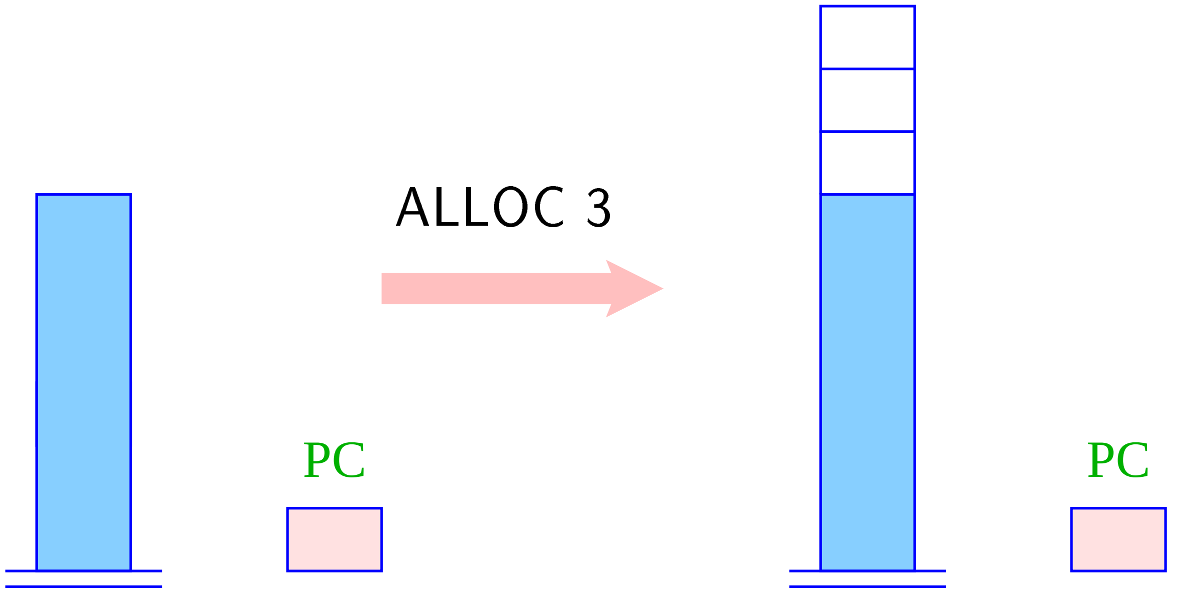

JUMP 17
PC
PC
17


tt
FJUMP 17
PC
PC


ff
FJUMP 17
PC
PC
17

Allokierung von Speicherplatz

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


ALLOC 3
PC
PC

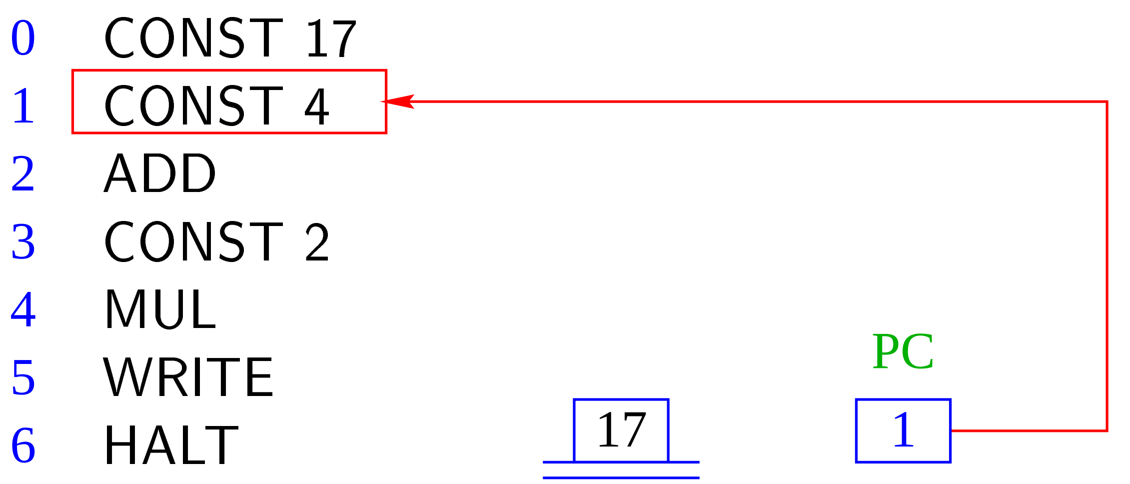
Ein Beispiel-Programm:

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

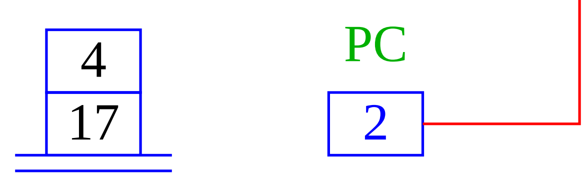
0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT

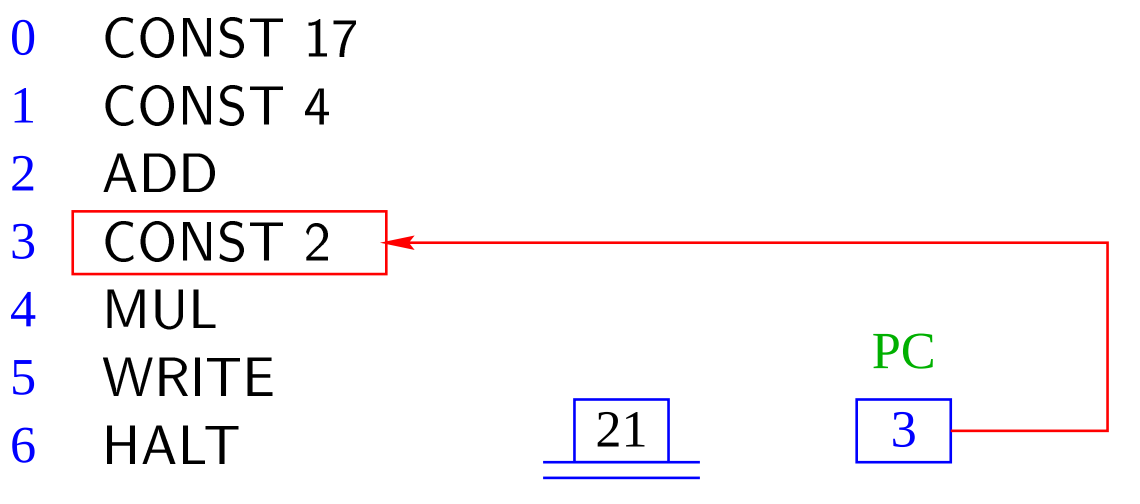
PC

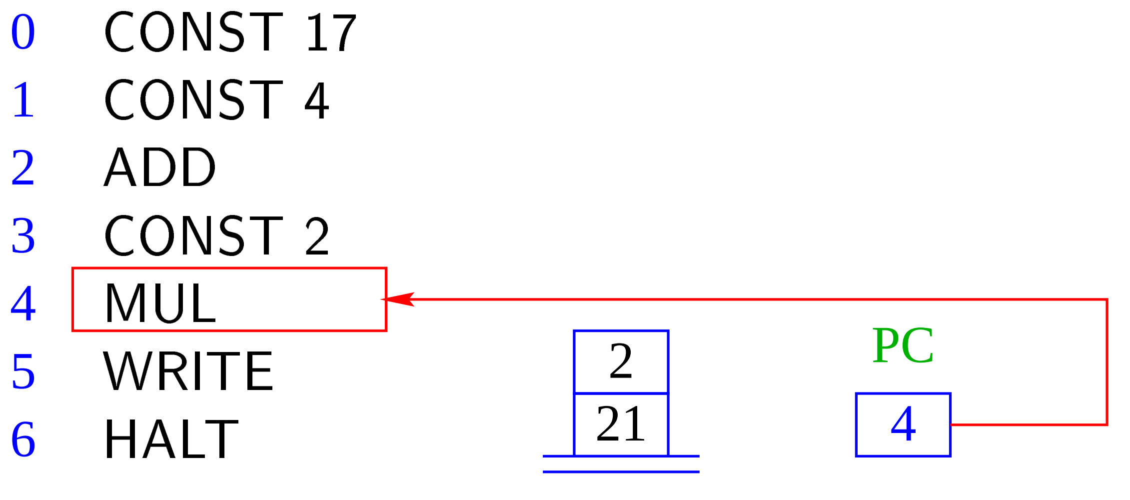
0


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
17
PC
1

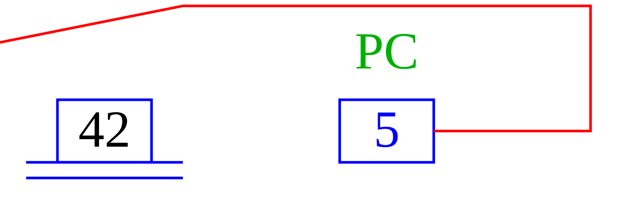
```
0   CONST 17
1   CONST 4
2   ADD
3   CONST 2
4   MUL
5   WRITE
6   HALT
```




0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
21
PC
3


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
2
21
PC
4

```
0   CONST 17
1   CONST 4
2   ADD
3   CONST 2
4   MUL
5   WRITE
6   HALT
```



```
0  CONST 17
1  CONST 4
2  ADD
3  CONST 2
4  MUL
5  WRITE
6  HALT
```

PC

6

Ausführung eines JVM-Programms:

```
PC = 0;
IR = Code[PC];
while (IR != HALT) {
   PC = PC + 1;
   execute(IR);
   IR = Code[PC];
}
```

- IR = Instruction Register, d.h. eine Variable, die den nächsten auszuführenden Befehl enthält.
- `execute(IR)` führt den Befehl in IR aus.
- `Code[PC]` liefert den Befehl, der in der Zelle in Code steht, auf die PC zeigt.

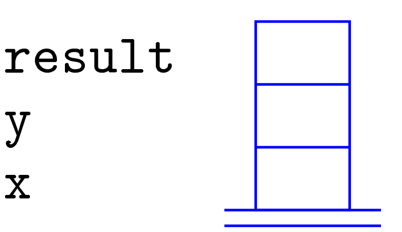
9.1 Übersetzung von Deklarationen

Betrachte Deklaration

```
int x, y, result;
```

Idee:

Wir reservieren der Reihe nach für die Variablen Zellen im Speicher:



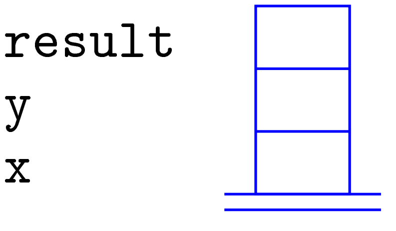
9.1 Übersetzung von Deklarationen

Betrachte Deklaration

```
int x, y, result;
```

Idee:

Wir reservieren der Reihe nach für die Variablen Zellen im Speicher:



$\Longrightarrow$

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

9.2 Übersetzung von Ausdrücken

Idee:

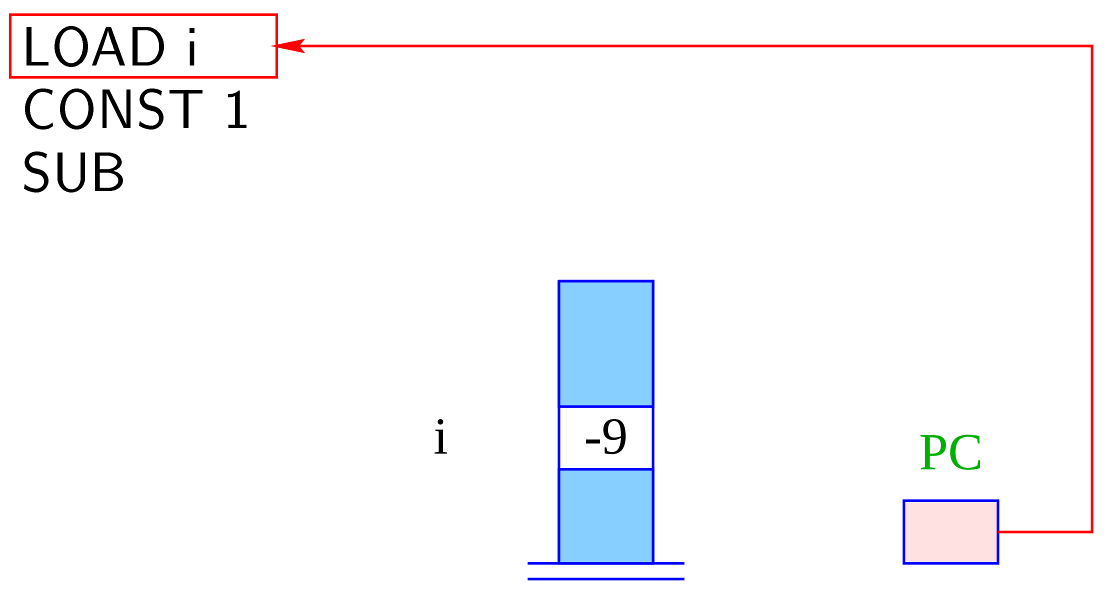
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

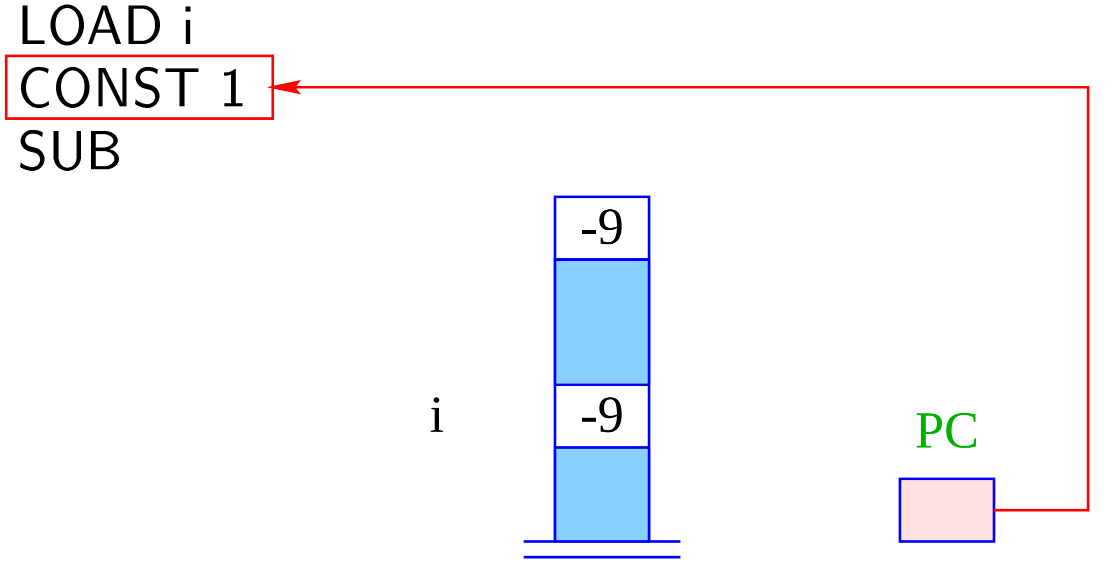
9.2 Übersetzung von Ausdrücken

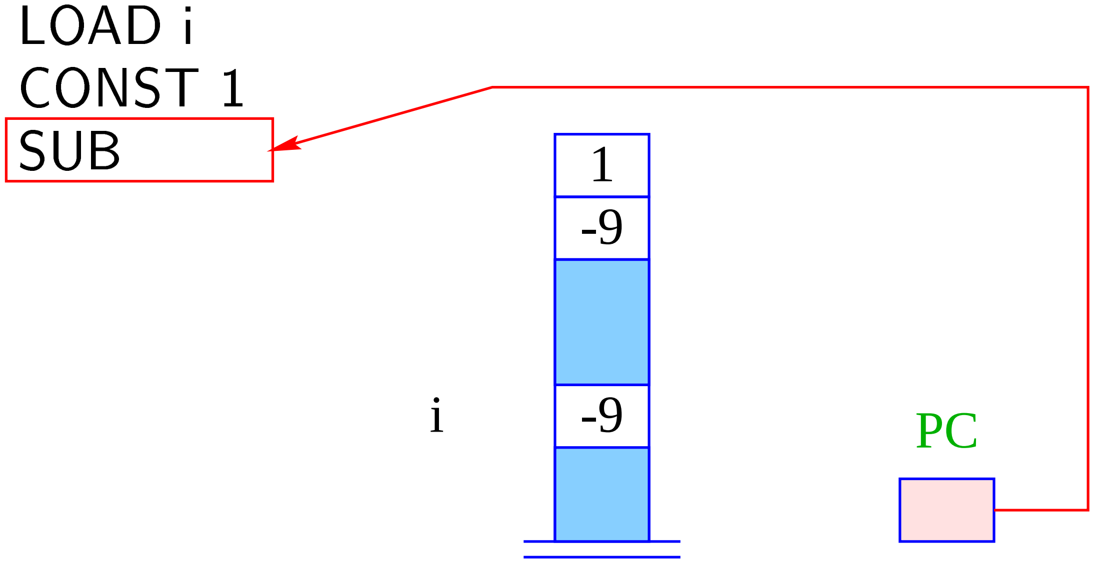
Idee:

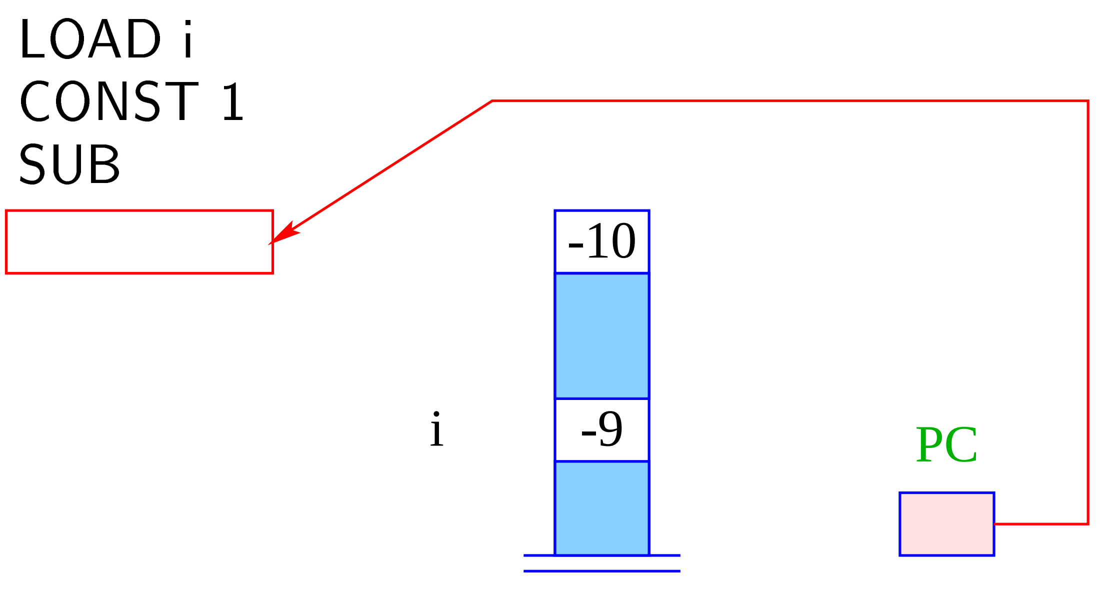
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

Übersetzung von	`x`	=	LOAD i	— `x` die *i*-te Variable
Übersetzung von	`17`	=	CONST 17	
Übersetzung von	`x - 1`	=	LOAD i	
			CONST 1	
			SUB	


LOAD i
CONST 1
SUB
i
-9
PC


LOAD i
CONST 1
SUB
-9
i
-9
PC


LOAD i
CONST 1
SUB
1
-9
i
-9
PC


LOAD i
CONST 1
SUB
-10
i
-9
PC

Allgemein:

Übersetzung von	- expr	=	Übersetzung von expr
			NEG
Übersetzung von	$expr_1$ + $expr_2$	=	Übersetzung von $expr_1$
			Übersetzung von $expr_2$
			ADD

... analog für die anderen Operatoren ...

Beispiel:

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

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

Dann liefert die Übersetzung:

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

9.3 Übersetzung von Zuweisungen

Idee:

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

9.4 Übersetzung von Zuweisungen

Idee:

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

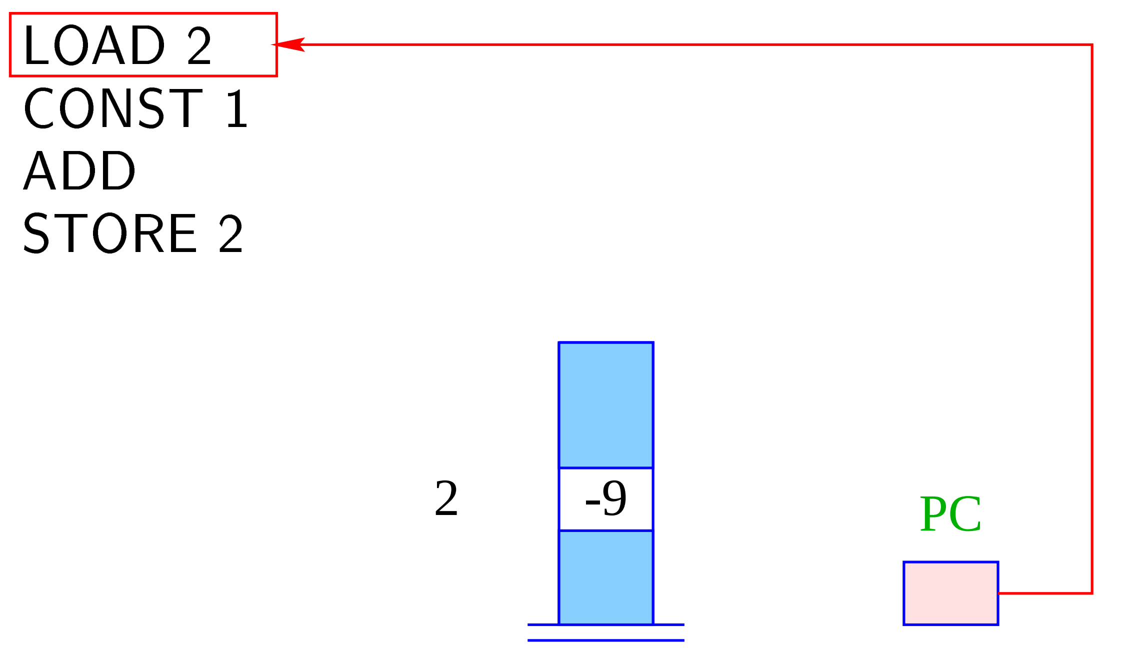
Sei `x` die Variable Nr. i. Dann ist

Übersetzung von	`x = expr;`	=	Übersetzung von	expr
			STORE i	

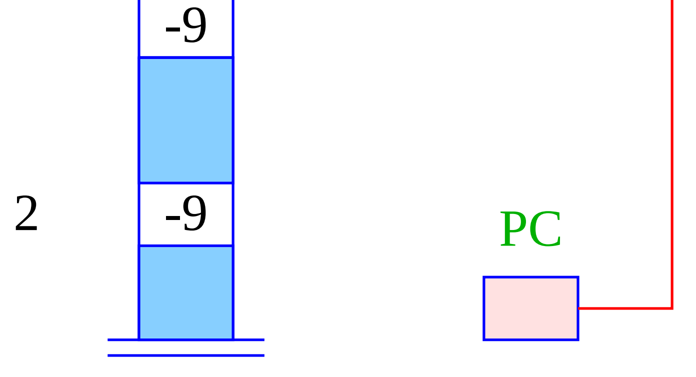
Beispiel:

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

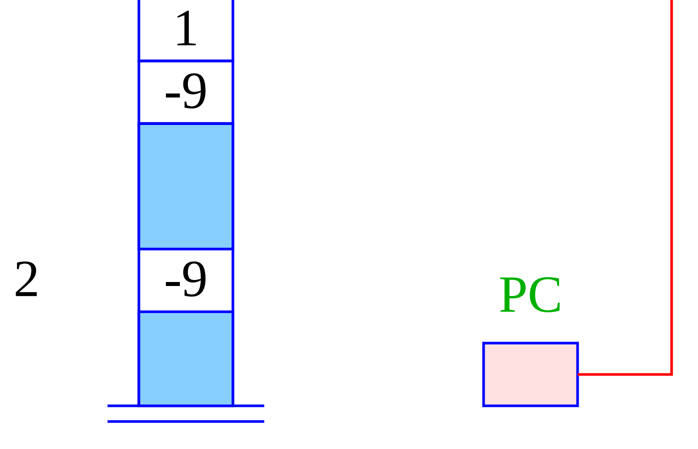
```
LOAD 2
CONST 1
ADD
STORE 2
```


LOAD 2
CONST 1
ADD
STORE 2
2
-9
PC

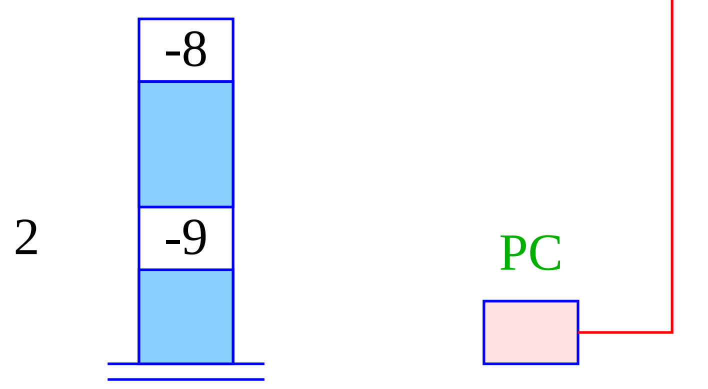
```
LOAD 2
CONST 1
ADD
STORE 2
```



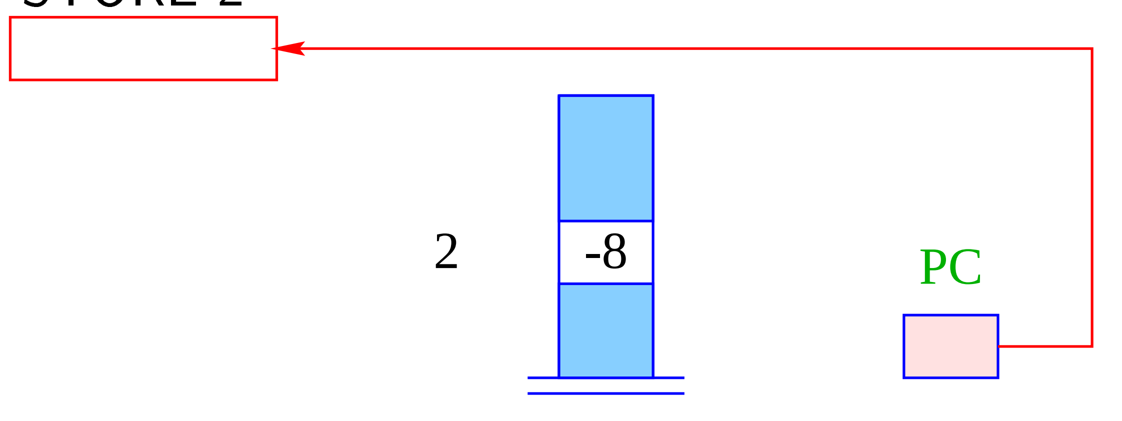
```
LOAD 2
CONST 1
ADD
STORE 2
```



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



LOAD 2
CONST 1
ADD
STORE 2



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

Sei `x` die Variable Nr. i. Dann ist

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

9.5 Übersetzung von `if`-Statements

Bezeichne stmt das `if`-Statement

$$\texttt{if (} \mathit{cond} \texttt{) } \mathit{stmt}_1 \texttt{ else } \mathit{stmt}_2$$

Idee:

- Wir erzeugen erst einmal Befehlsfolgen für cond, stmt_1 und stmt_2.
- Diese ordnen wir hinter einander an.
- Dann fügen wir Sprünge so ein, dass in Abhängigkeit des Ergebnisses der Auswertung der Bedingung jeweils entweder nur stmt_1 oder nur stmt_2 ausgeführt wird.

Folglich (mit A, B zwei neuen Marken):

Übersetzung von stmt =		Übersetzung von	cond
		FJUMP A	
		Übersetzung von	$stmt_1$
		JUMP B	
	A:	Übersetzung von	$stmt_2$
	B:	...	

- Marke A markiert den Beginn des else-Teils.
- Marke B markiert den ersten Befehl hinter dem `if`-Statement.
- Falls die Bedingung sich zu `false` evaluiert, wird der `then`-Teil übersprungen (mithilfe von FJUMP A).
- Nach Abarbeitung des `then`-Teils muss in jedem Fall hinter dem gesamten `if`-Statement fortgefahren werden. Dazu dient JUMP B.

Beispiel:

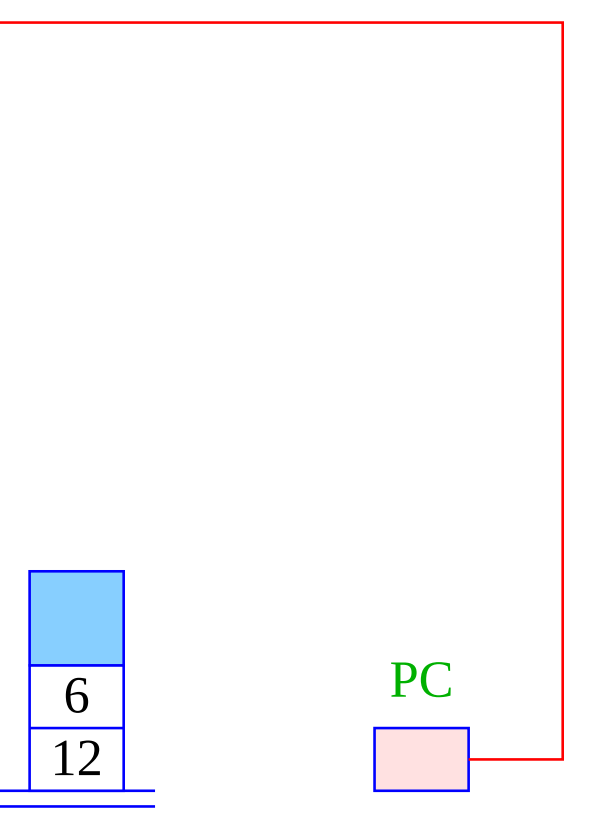
Für das Statement:

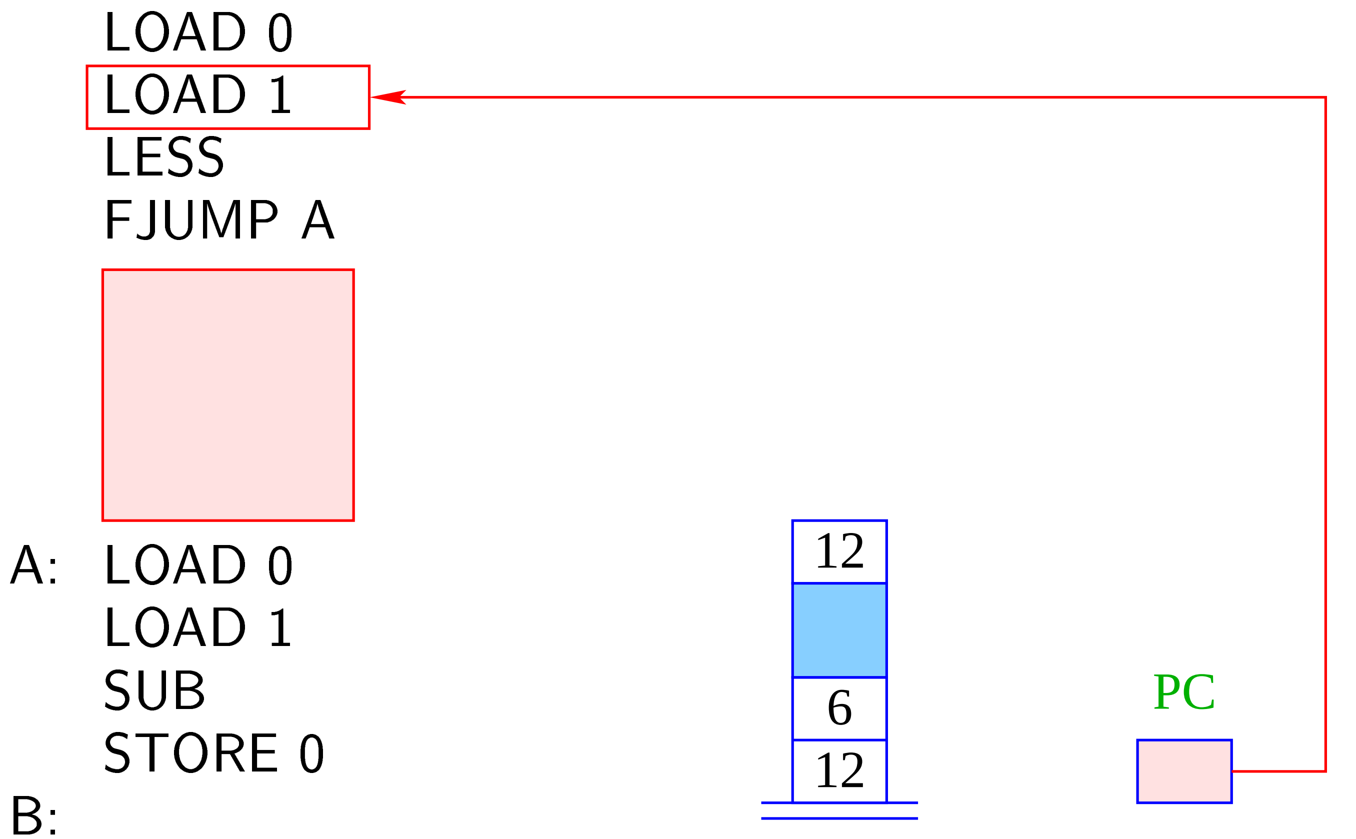
```
if (x < y) y = y - x;
else x = x - y;
```

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

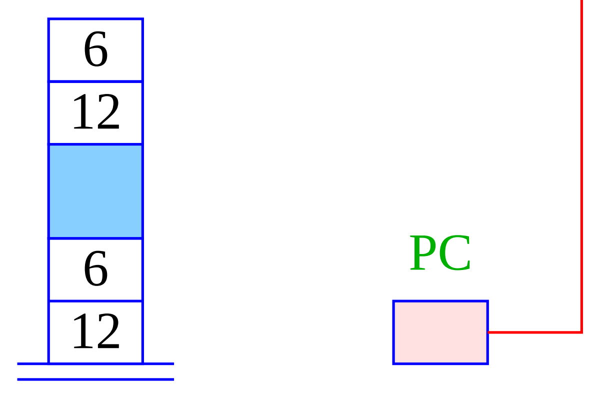
LOAD 0	LOAD 1	A:	LOAD 0
LOAD 1	LOAD 0		LOAD 1
LESS	SUB		SUB
FJUMP A	STORE 1		STORE 0
	JUMP B	B:	...

```
     LOAD 0
     LOAD 1
     LESS
     FJUMP A

A:   LOAD 0
     LOAD 1
     SUB
     STORE 0
B:
```

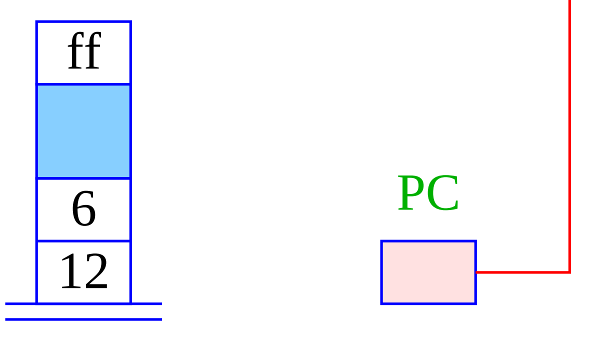



LOAD 0
LOAD 1
LESS
FJUMP A
A: LOAD 0
LOAD 1
SUB
STORE 0
B:
12
6
12
PC

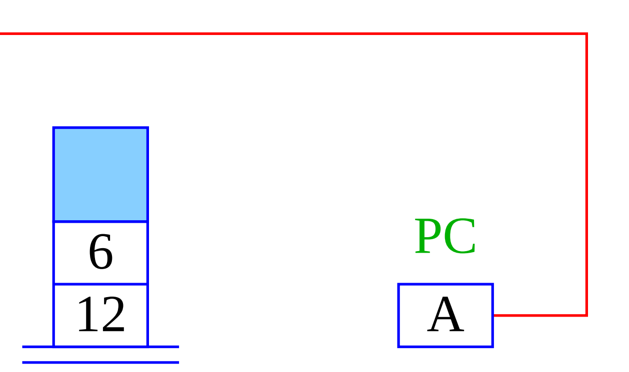
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



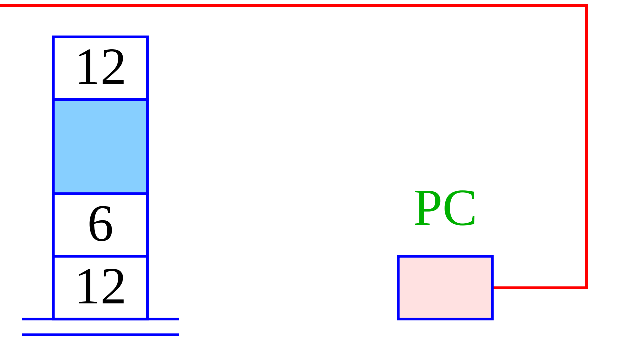
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



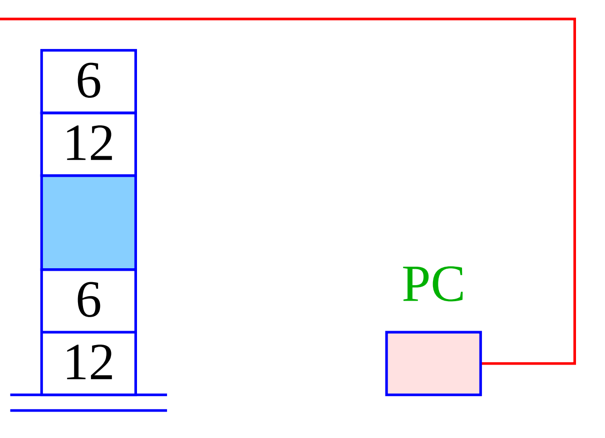
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



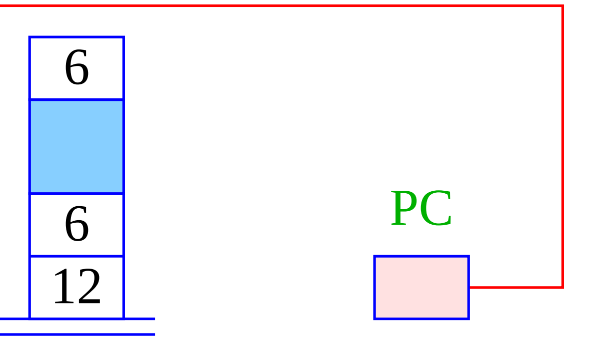
```
     LOAD 0
     LOAD 1
     LESS
     FJUMP A

A:   LOAD 0
     LOAD 1
     SUB
     STORE 0
B:
```



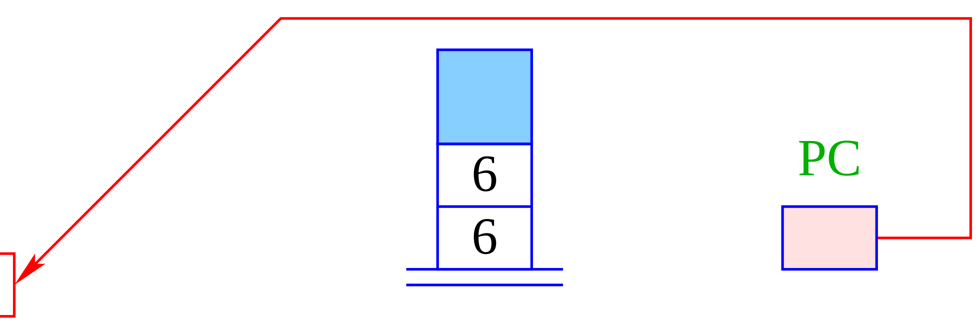
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



9.6 Übersetzung von `while`-Statements

Bezeichne stmt das `while`-Statement

$$\texttt{while} \; (\text{cond}) \; \text{stmt}_1$$

Idee:

- Wir erzeugen erst einmal Befehlsfolgen für cond und stmt_1.
- Diese ordnen wir hinter einander an.
- Dann fügen wir Sprünge so ein, dass in Abhängigkeit des Ergebnisses der Auswertung der Bedingung entweder hinter das `while`-Statement gesprungen wird oder stmt_1 ausgeführt wird.
- Nach Ausführung von stmt_1 müssen wir allerdings wieder an den Anfang des Codes zurückspringen :-)

Folglich (mit A, B zwei neuen Marken):

Übersetzung von	stmt	=	A:	Übersetzung von cond
				FJUMP B
				Übersetzung von $stmt_1$
				JUMP A
			B:	...

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

Beispiel:

Für das Statement:

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

(`x` die 0. Variable) ergibt das:

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

9.7 Übersetzung von Statement-Folgen

Idee:

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

9.8 Übersetzung von Statement-Folgen

Idee:

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

Folglich:

Übersetzung von	$\text{stmt}_1 \ldots \text{stmt}_k$	=	Übersetzung von	stmt_1
			...	
			Übersetzung von	stmt_k

Beispiel:

Für die Statement-Folge

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

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

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

9.9 Übersetzung ganzer Programme

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

Idee:

- Zuerst allokieren wir Platz für die deklarierten Variablen.
- Dann kommt der Code für ss.
- Dann HALT.

9.9 Übersetzung ganzer Programme

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

Idee:

- Zuerst allokieren wir Platz für die deklarierten Variablen.
- Dann kommt der Code für ss.
- Dann HALT.

Folglich:

Übersetzung von	prog	=	ALLOC n
			Übersetzung von ss
			HALT

Beispiel:

Für das Programm

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

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

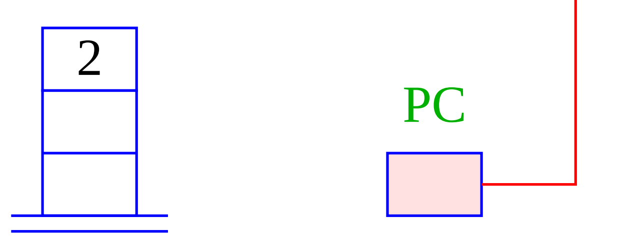
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1

A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    LOAD 1
    LOAD 0
    MUL
    STORE 1

    LOAD 0
    CONST 1
    SUB
    STORE 0
    JUMP A

B:  LOAD 1
    WRITE
    HALT
```

```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```

PC

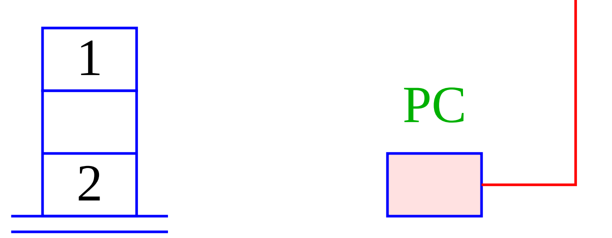
```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B

     JUMP A
B:   LOAD 1
     WRITE
     HALT
```

```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



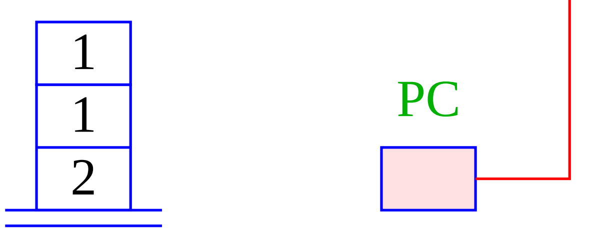
```
       ALLOC 2
       READ
       STORE 0
       CONST 1
       STORE 1
A:     CONST 1
       LOAD 0
       LESS
       FJUMP B

       JUMP A
B:     LOAD 1
       WRITE
       HALT
```

```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B

     JUMP A
B:   LOAD 1
     WRITE
     HALT
```

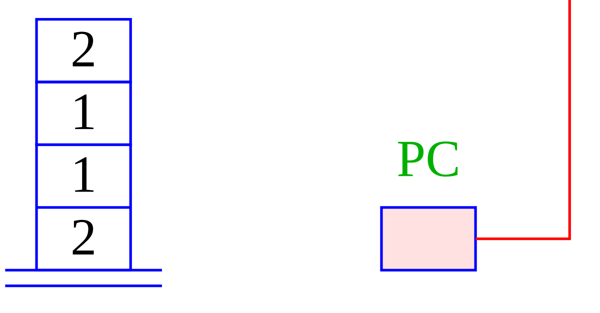


```
       ALLOC 2
       READ
       STORE 0
       CONST 1
       STORE 1
A:     CONST 1
       LOAD 0
       LESS
       FJUMP B

       JUMP A
B:     LOAD 1
       WRITE
       HALT
```

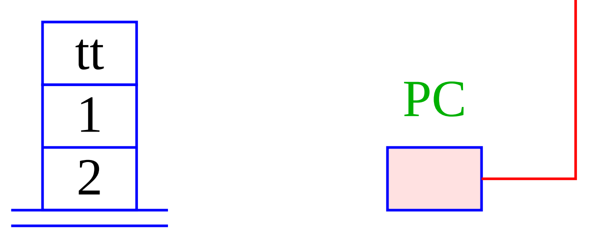
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```

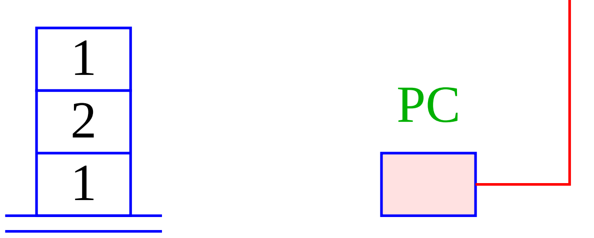


```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```

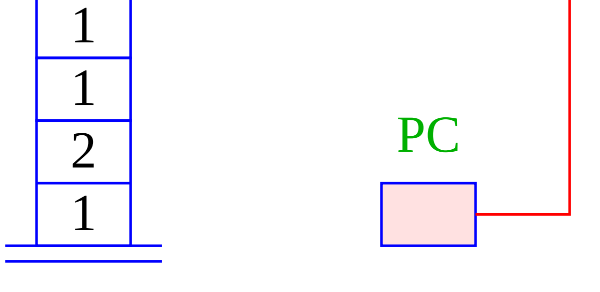
```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B

     JUMP A
B:   LOAD 1
     WRITE
     HALT
```

```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```

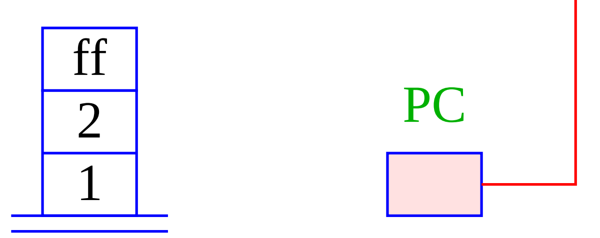
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



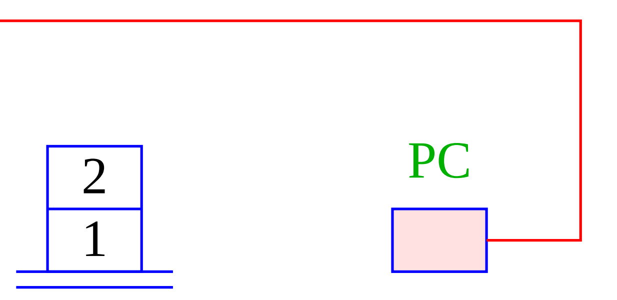
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



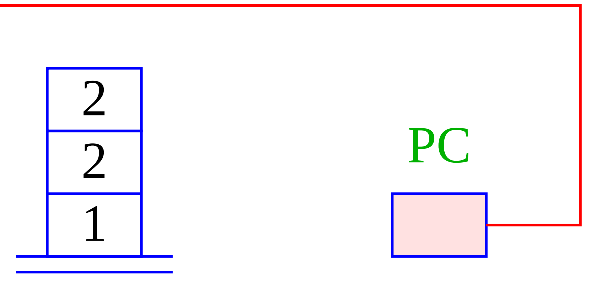
```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B

     JUMP A
B:   LOAD 1
     WRITE
     HALT
```



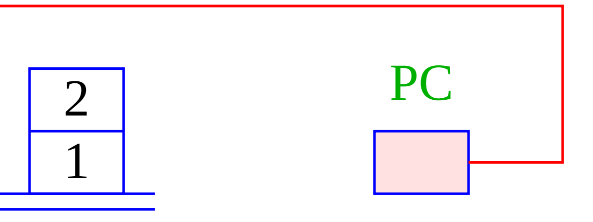
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B

     JUMP A
B:   LOAD 1
     WRITE
     HALT
```



```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B

     JUMP A
B:   LOAD 1
     WRITE
     HALT
```



Bemerkungen:

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

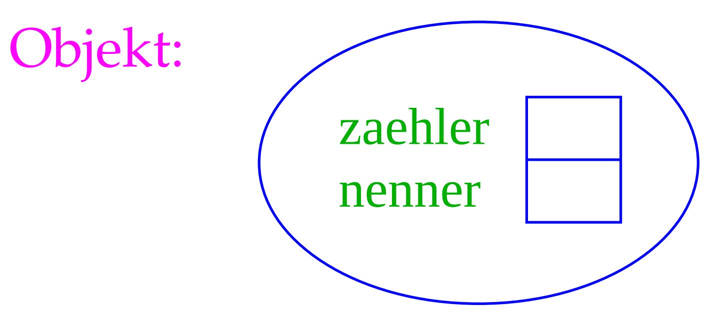
↑Compilerbau

10 Klassen und Objekte

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

Beispiel: Rationale Zahlen

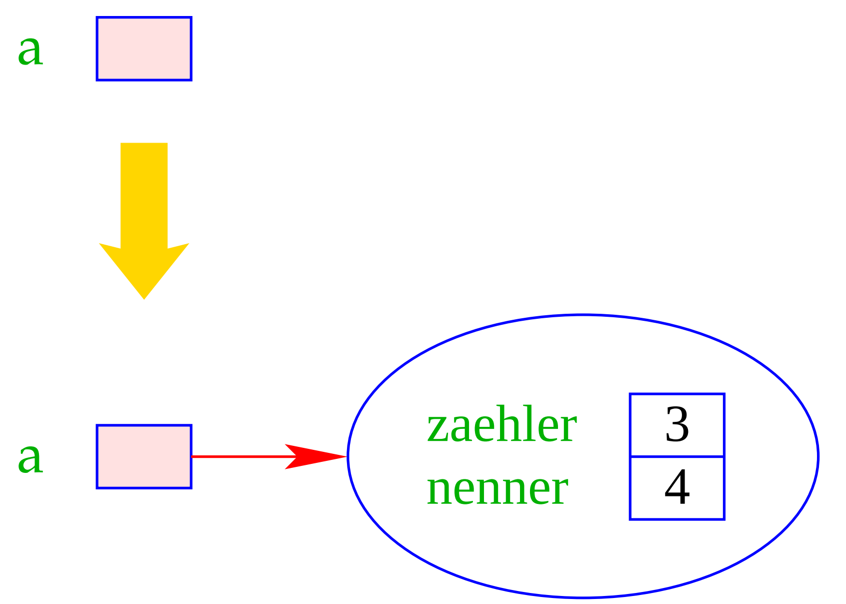
- Eine rationale Zahl $q \in \mathbb{Q}$ hat die Form $q = \frac{x}{y}$, wobei $x, y \in \mathbb{Z}$.
- x und y heißen Zähler und Nenner von q.
- Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` enthalten:



- Die Daten-Komponenten eines Objekts heißen Instanz-Variablen oder 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:

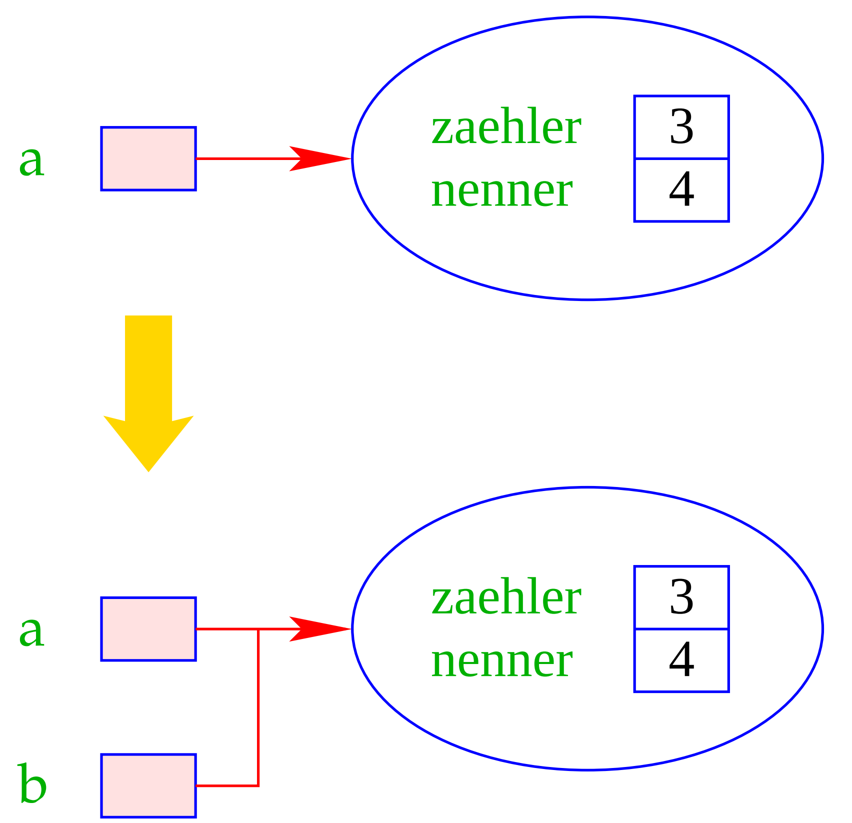
```
a = new Rational (3,4);
```



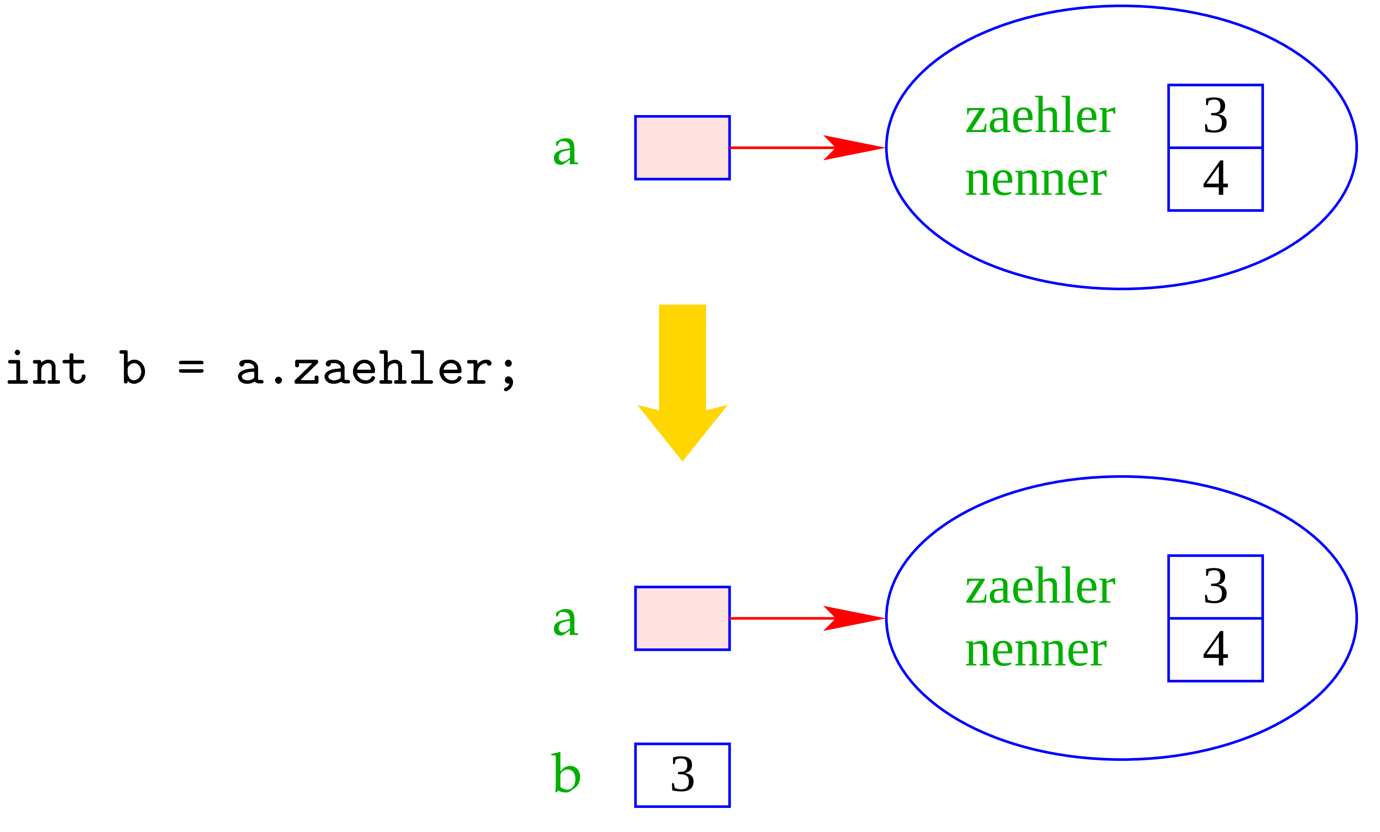
- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objekts initialisieren kann.

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

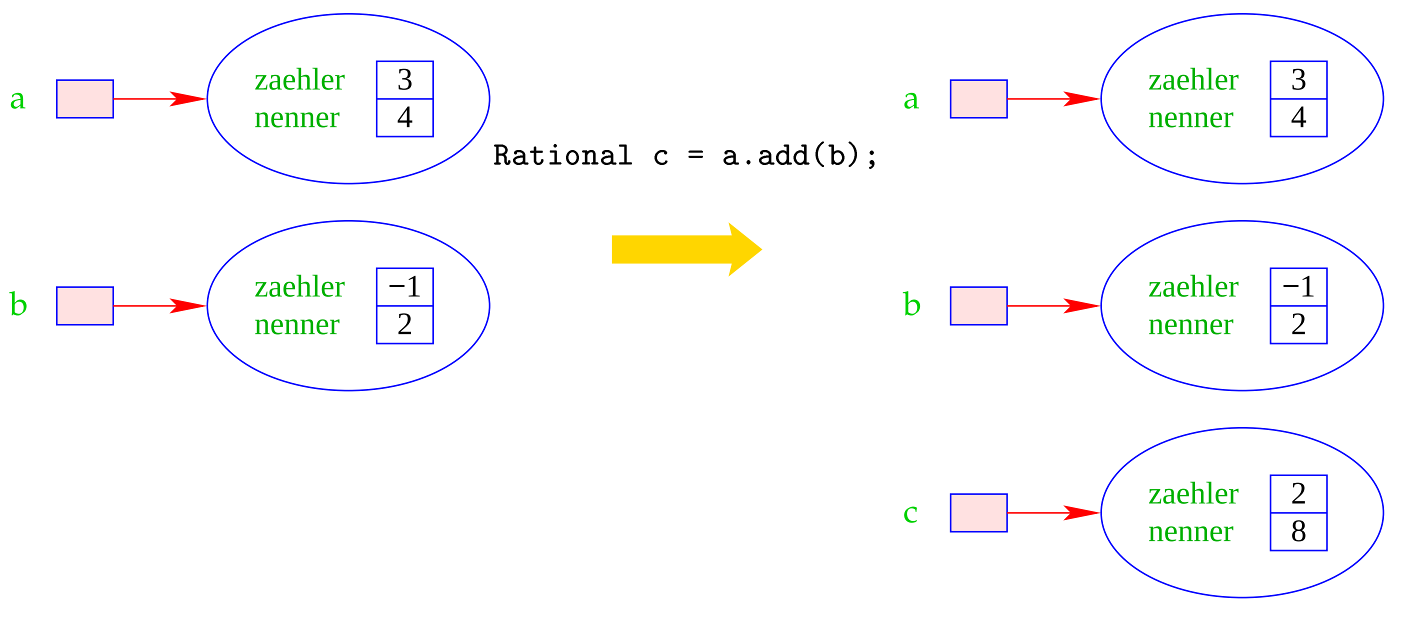
`Rational b = a;`

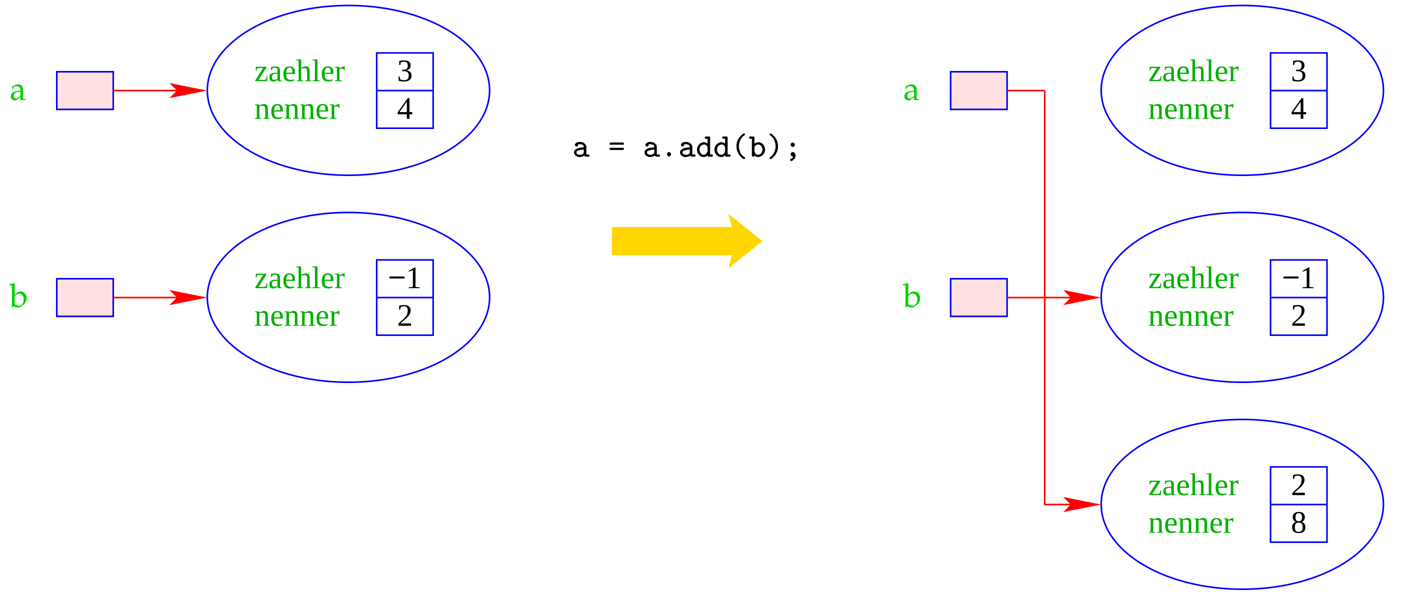


- `a.zaehler` liefert den Wert des Attributs `zaehler` des Objekts `a`:



- `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
```

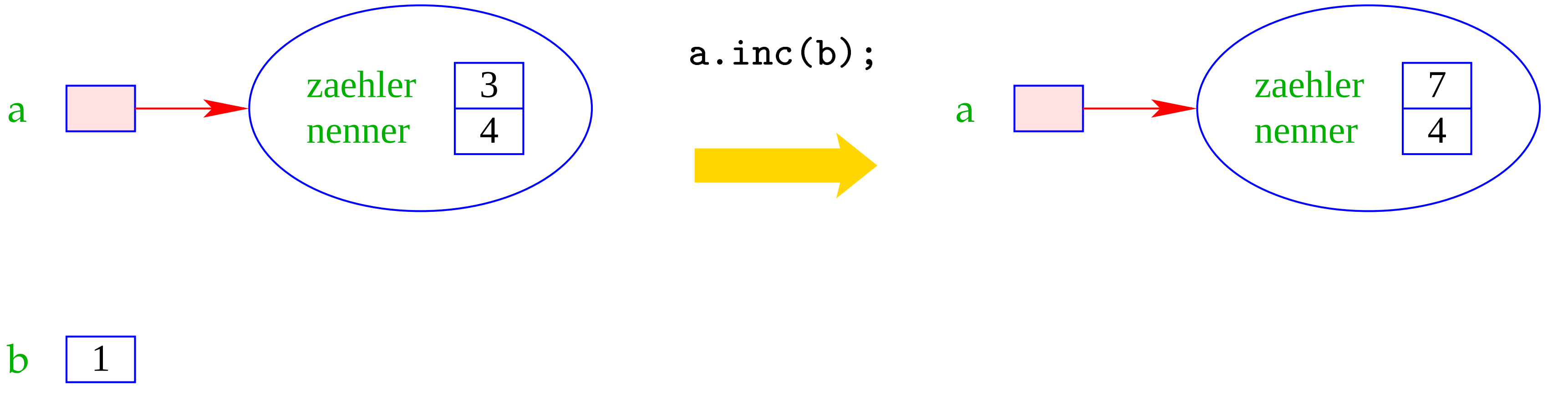
Bemerkungen:

- 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:

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

Diskussion und Ausblick:

- Solche Diagramme werden von der UML, d.h. der Unified Modelling Language bereitgestellt, um Software-Systeme zu entwerfen (↑Software Engineering)
- Für eine einzelne Klasse lohnen 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 lohnen 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 oder nur schlecht modellieren :-(

10.1 Selbst-Referenzen

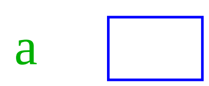
```
public class Cyclic {
   private int info;
   private Cyclic ref;
   // Konstruktor
   public Cyclic() {
      info = 17;
      ref = this;
      }
      ...
   } // end of class Cyclic
```

10.1 Selbst-Referenzen

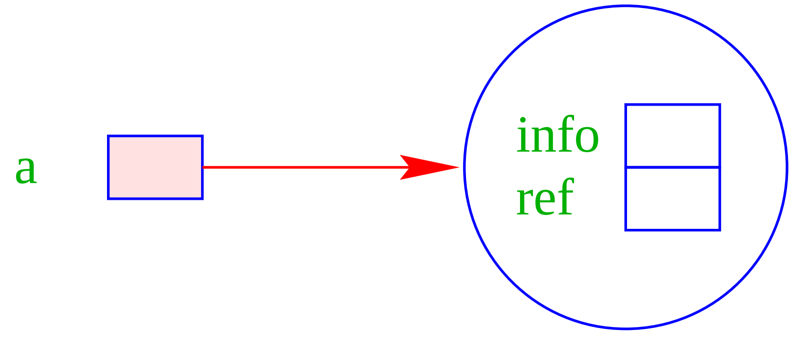
```
public class Cyclic {
   private int info;
   private Cyclic ref;
   // Konstruktor
   public Cyclic() {
      info = 17;
      ref = this;
      }
      ...
   } // end of class Cyclic
```

Innerhalb eines Members kann man mithilfe von `this` auf das aktuelle Objekt selbst zugreifen :-)

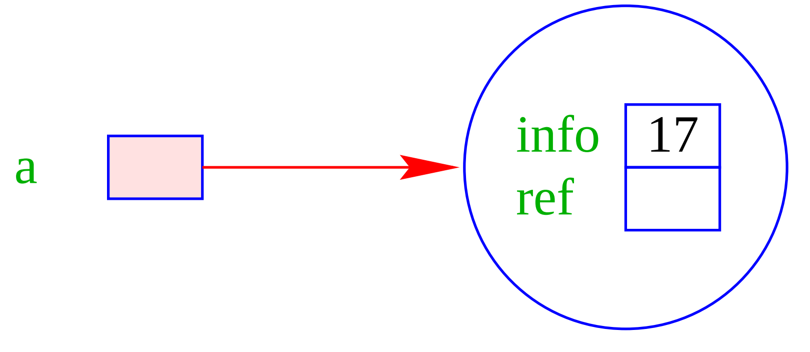
Für `Cyclic a = new Cyclic();` ergibt das:



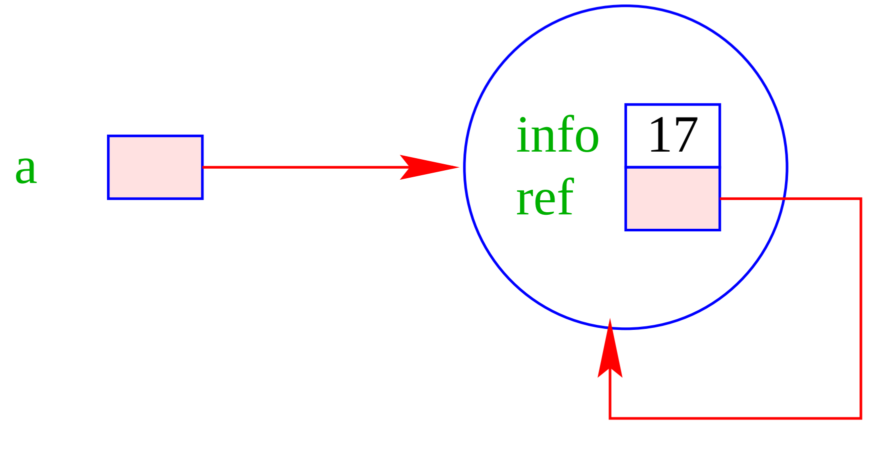
Für `Cyclic a = new Cyclic();` ergibt das:



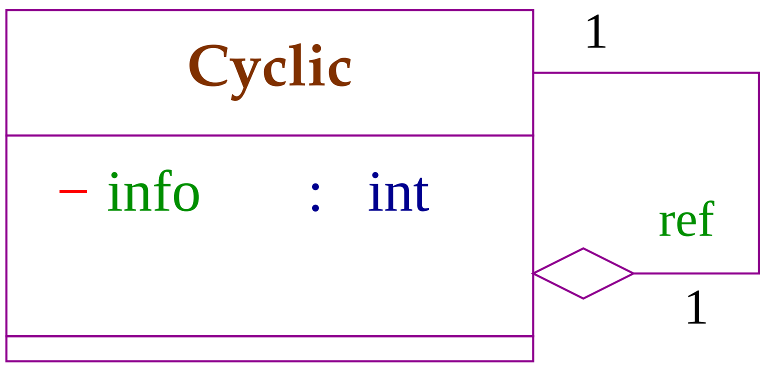
Für `Cyclic a = new Cyclic();` ergibt das:



Für `Cyclic a = new Cyclic();` ergibt das:



Modellierung einer Selbst-Referenz:



Die Rauten-Verbindung heißt auch Aggregation.

Das Klassen-Diagramm vermerkt, dass jedes Objekt der Klasse `Cyclic` einen Verweis mit dem Namen ref auf ein weiteres Objekt der Klasse `Cyclic` enthält :-)

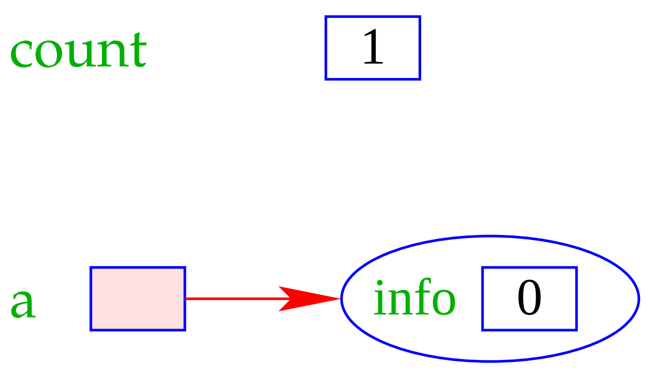
10.2 Klassen-Attribute

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

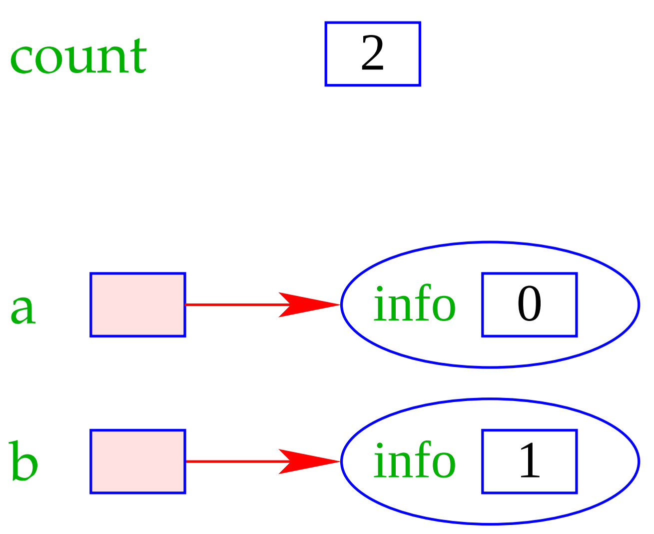
```
public class Count {
   private static int count = 0;
   private int info;
   // Konstruktor
   public Count() {
      info = count; count++;
      } ...
   } // end of class Count
```

count $\boxed{0}$

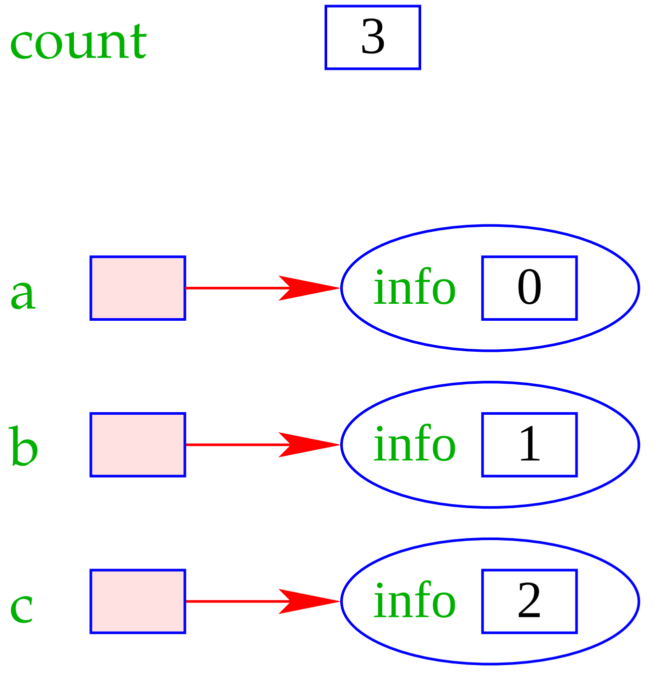
```
Count a = new Count();
```




```
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.

11 Abstrakte Datentypen

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

$\Longrightarrow$ Information Hiding

Sinn:

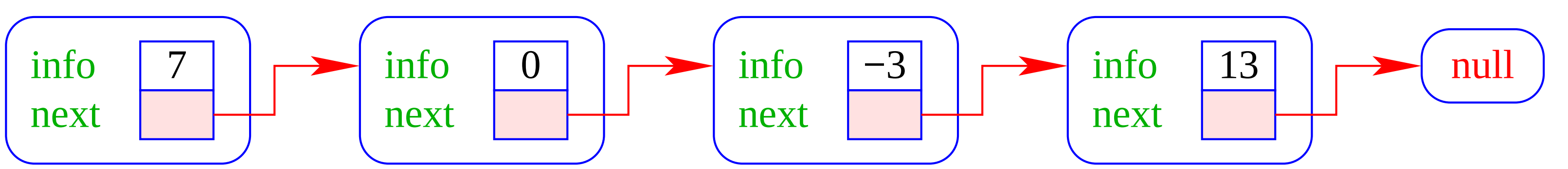
- Verhindern illegaler Zugriffe auf die Datenstruktur;
- Entkopplung von Teilproblemen für
  - Implementierung, aber auch
  - Fehlersuche und
  - Wartung;
- leichter Austausch von Implementierungen (↑rapid prototyping).

11.1 Ein konkreter Datentyp: Listen

Nachteil von Feldern:

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

Idee: Listen



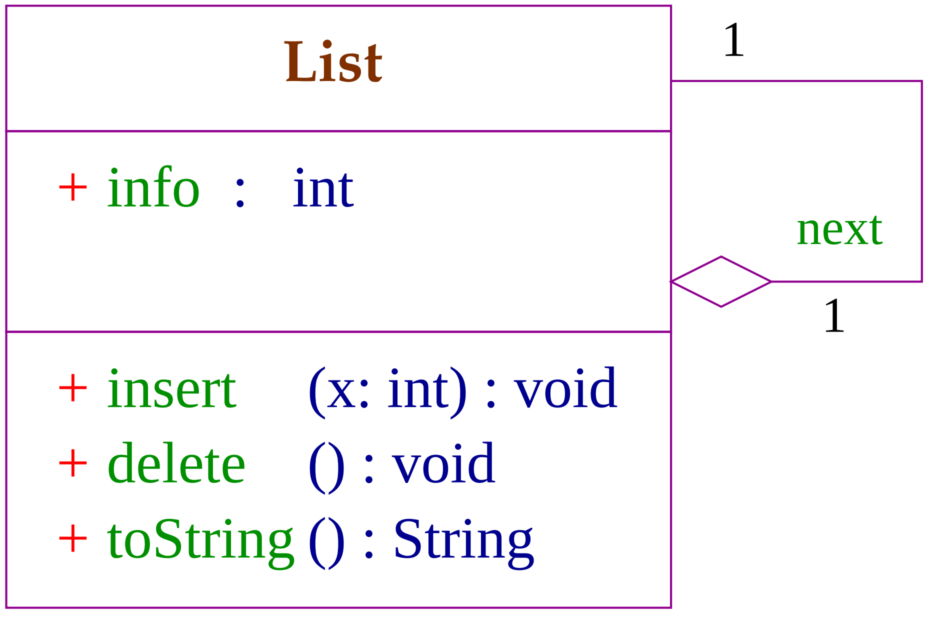
... das heißt:

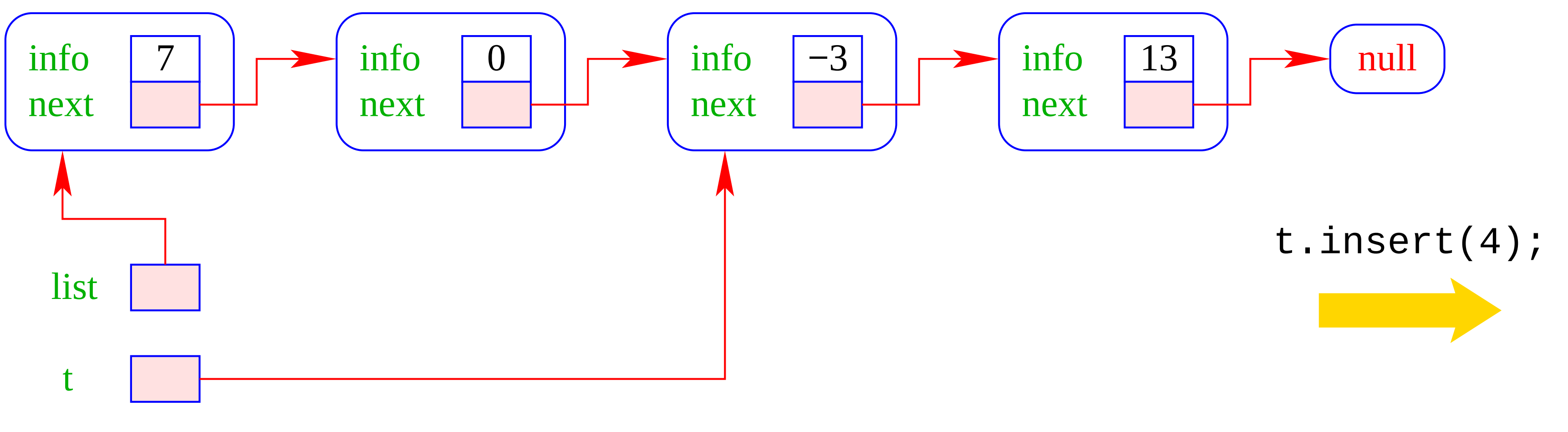
- `info` $==$ Element der Liste;
- `next` $==$ Verweis auf das nächste Element;
- `null` $==$ leeres Objekt.

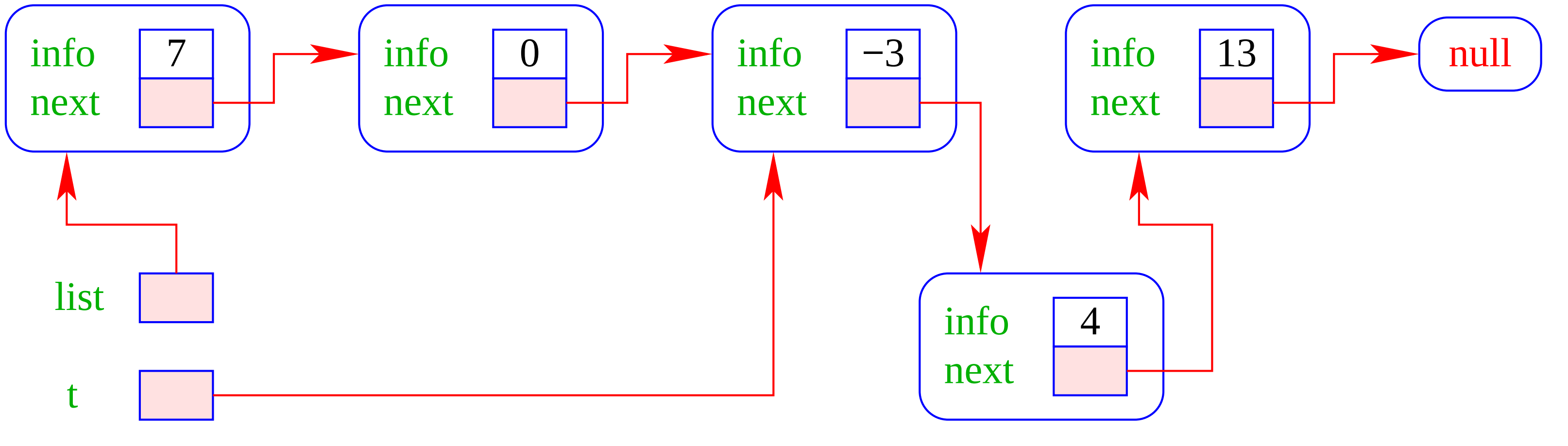
Operationen:

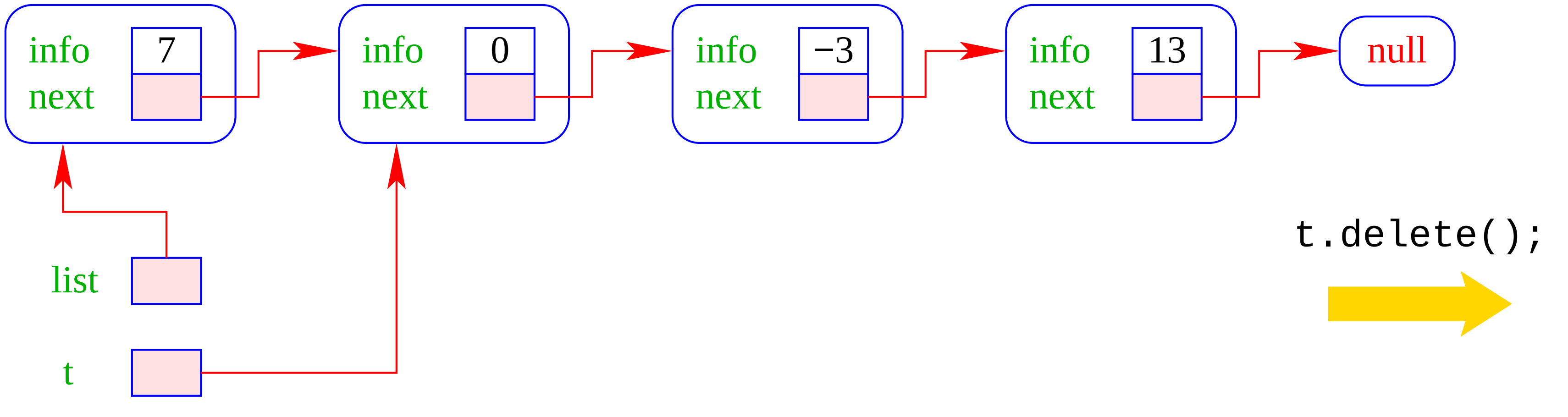
`void insert(int x)`	:	fügt neues x hinter dem aktuellen Element ein;
`void delete()`	:	entfernt Knoten hinter dem aktuellen Element;
`String toString()`	:	liefert eine String-Darstellung.

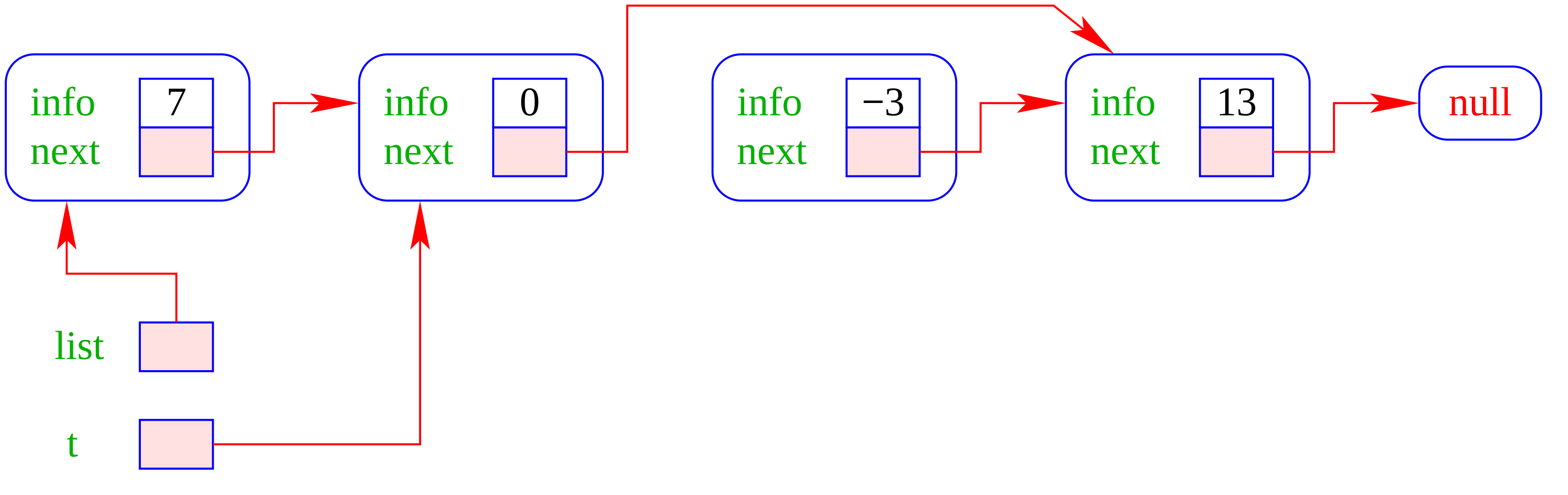
Modellierung:




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


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


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;
      }
   ...
```

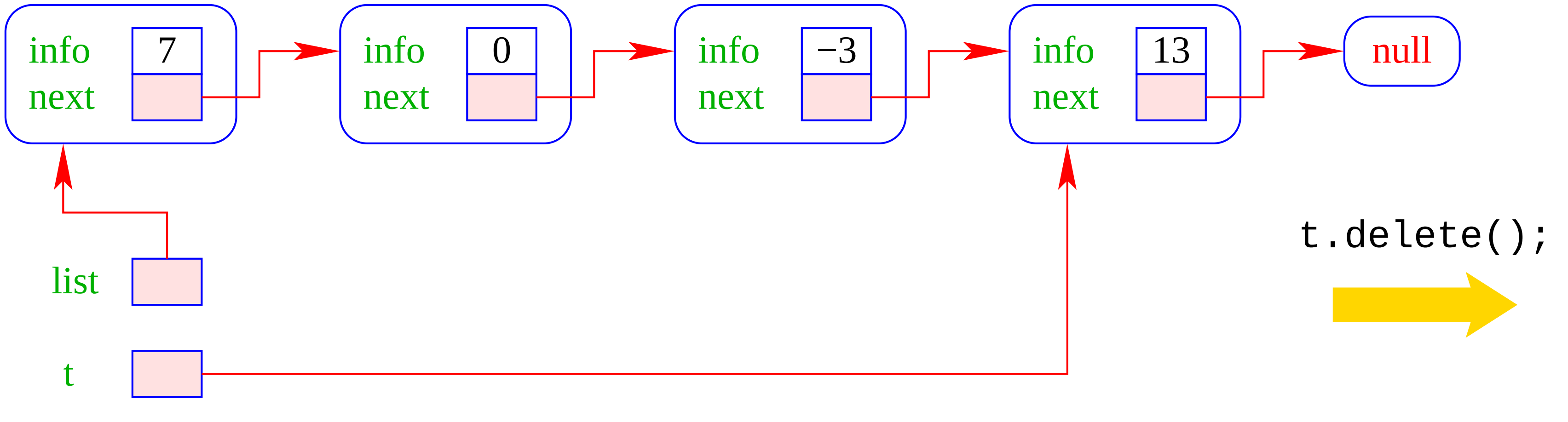
```
// Objekt-Methoden:
   public void insert(int x) {
      next = new List(x,next);
      }
   public void delete() {
      if (next != null)
         next = next.next;
      }
   public String toString() {
      String result = "["+info;
      for(List t=next; t!=null; t=t.next)
         result =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

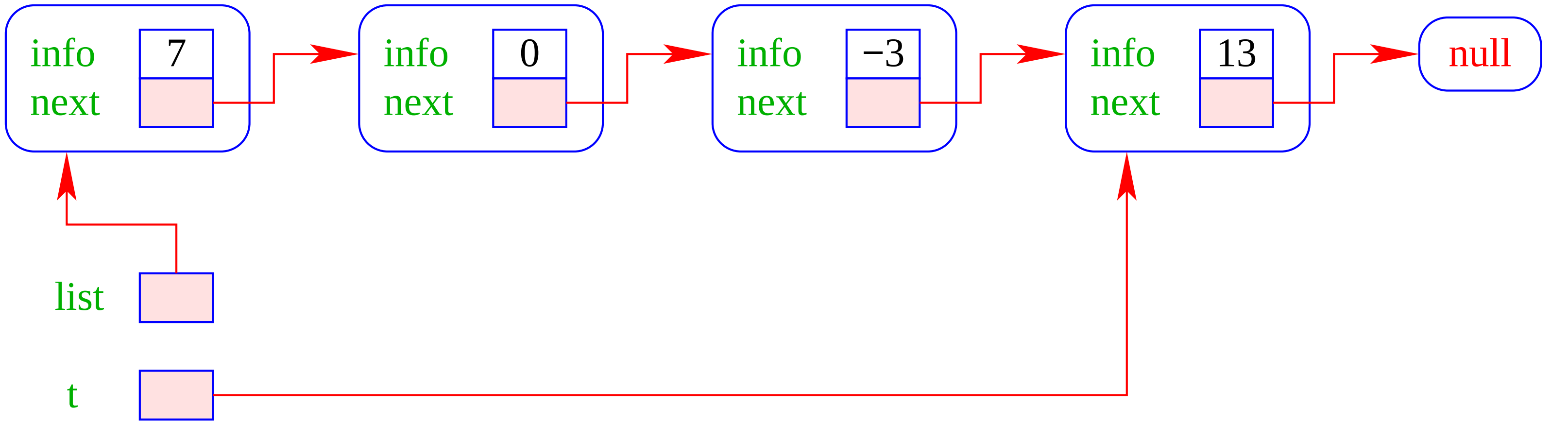
- Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar $\Longrightarrow$ sehr flexibel, sehr fehleranfällig.
- `insert()` legt einen neuen Listenknoten an fügt ihn hinter dem aktuellen Knoten ein.
- `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

Achtung:

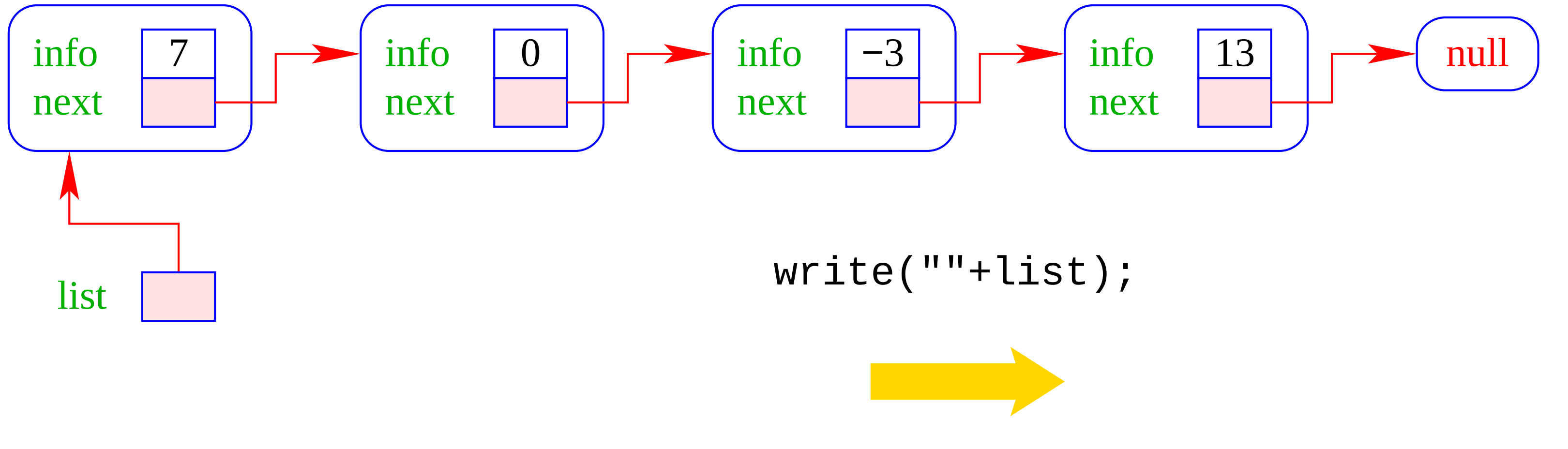
Wenn `delete()` mit dem letzten Element der Liste aufgerufen wird, zeigt `next` auf `null`.

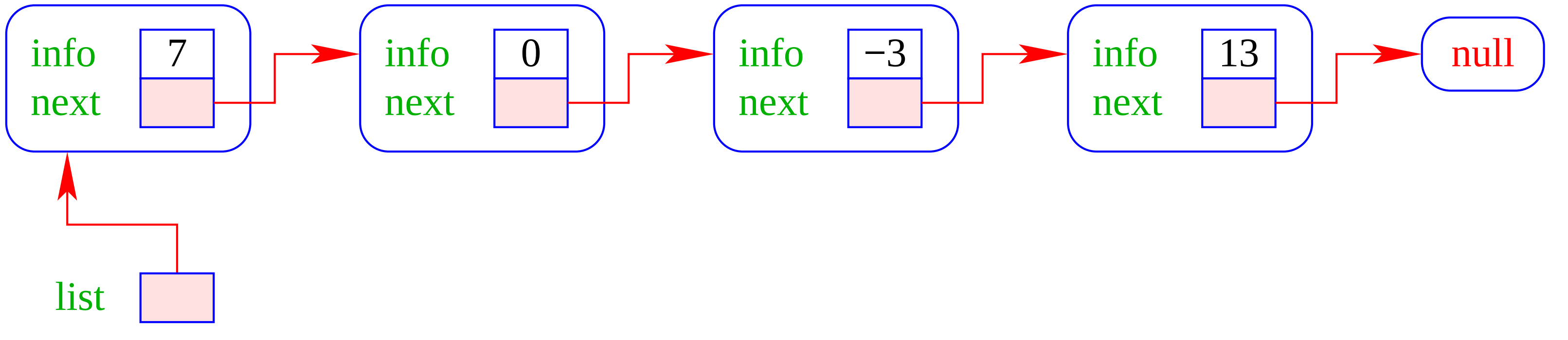
$\Longrightarrow$ Wir tun dann nix.


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


info
next
7
0
−3
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
7
next
info
0
next
info
−3
next
info
13
next
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
```

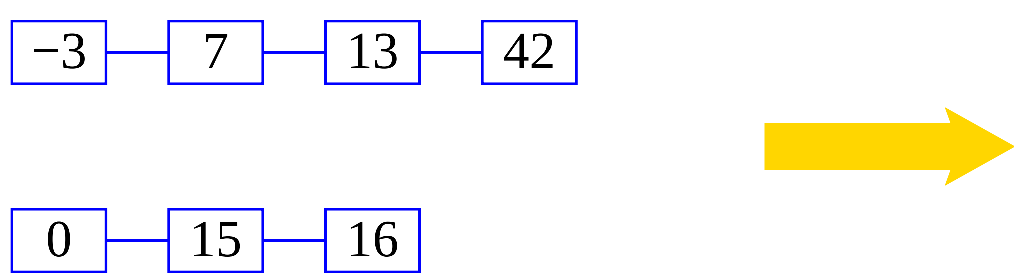
- Weil `length()` als `private` deklariert ist, kann es nur von den Methoden der Klasse `List` benutzt werden.
- Damit `length()` auch für `null` funktioniert, hätten wir analog zu toString() auch noch eine Klassen-Methode `int length(List l)` definieren können.
- Diese Klassen-Methode würde uns ermöglichen,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

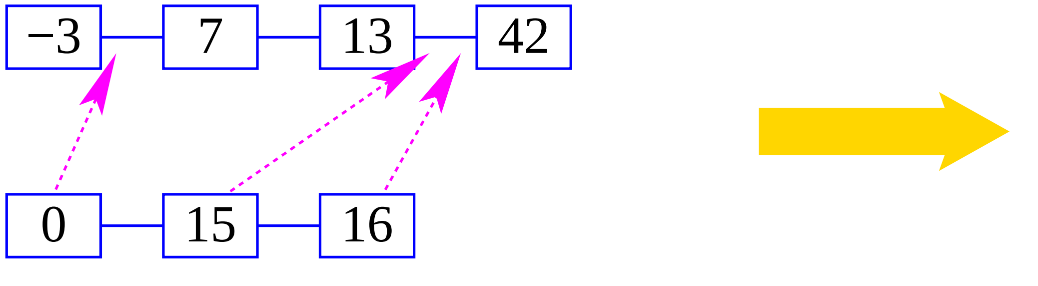
Anwendung: Mergesort – Sortieren durch Mischen

Mischen:

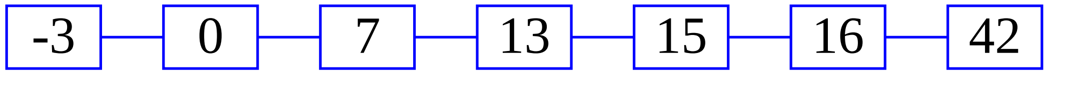
Eingabe: zwei sortierte Listen;

Ausgabe: eine gemeinsame sortierte Liste.


−3
7
13
42
0
15
16

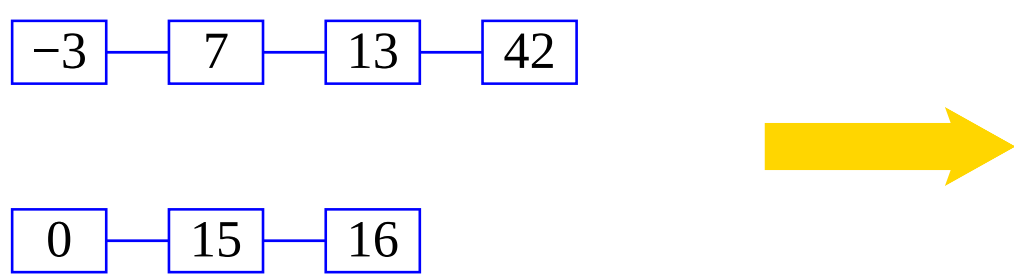


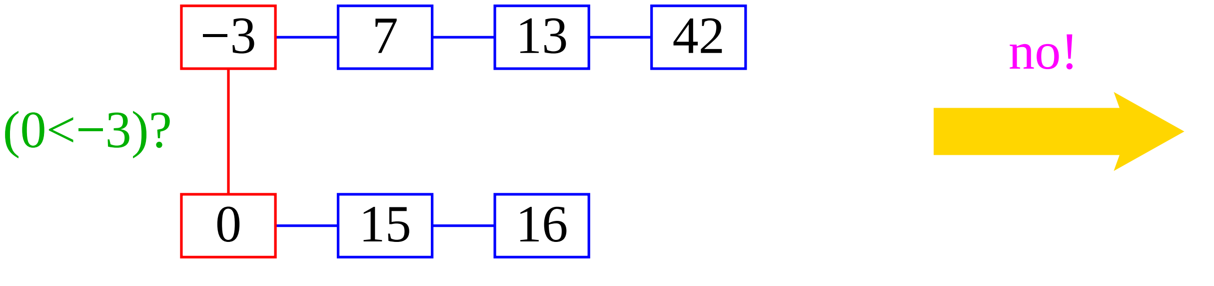
−3
7
13
42
0
15
16

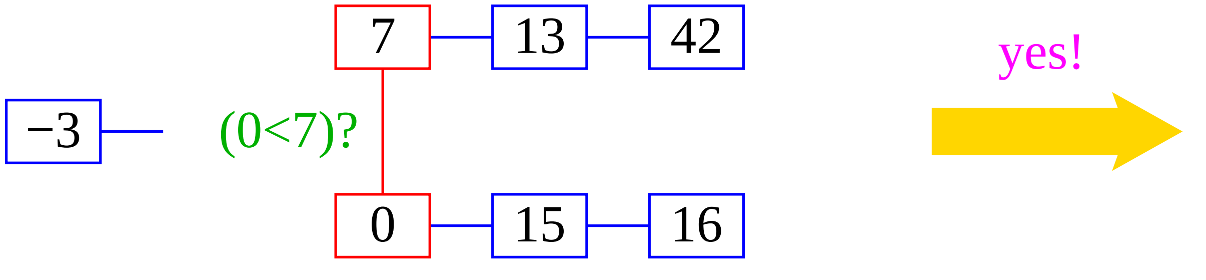

-3
0
7
13
15
16
42

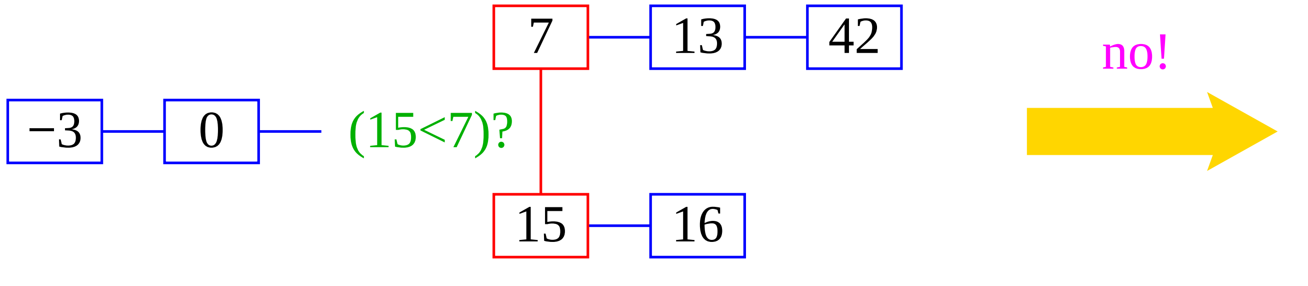
Idee:

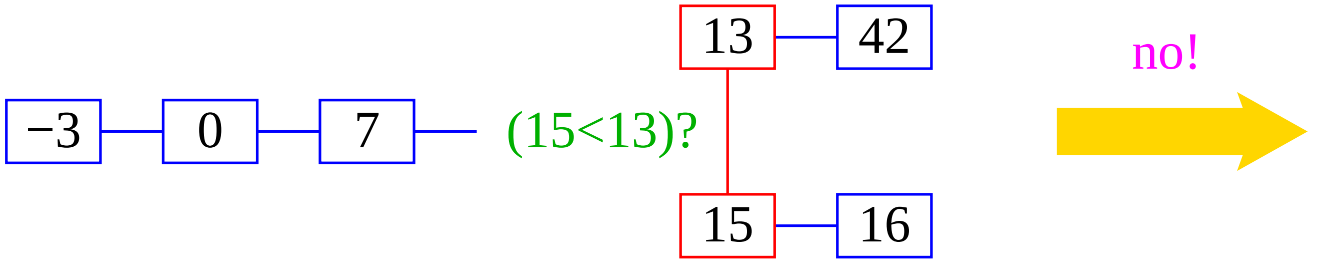
- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Input-Listen.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche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

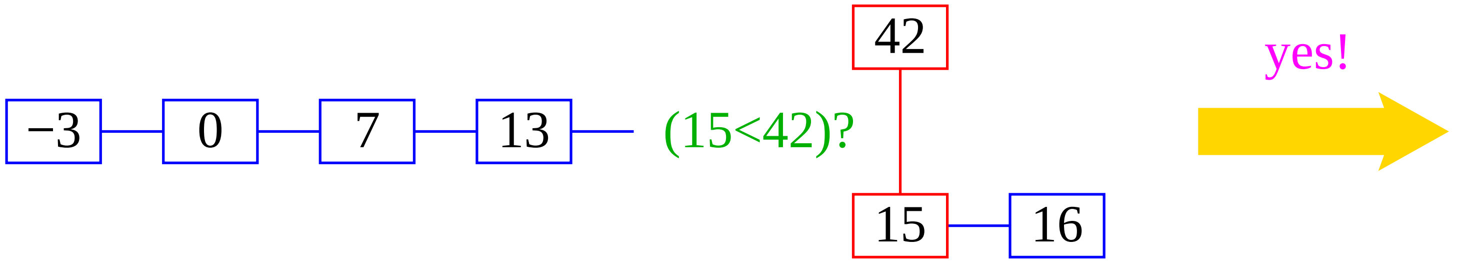

−3
7
13
42
0
15
16

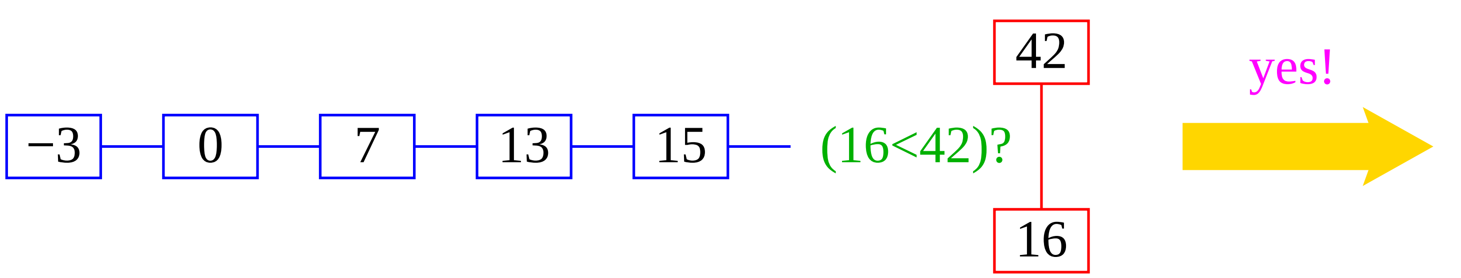

−3
7
13
42
(0<−3)?
0
15
16
no!

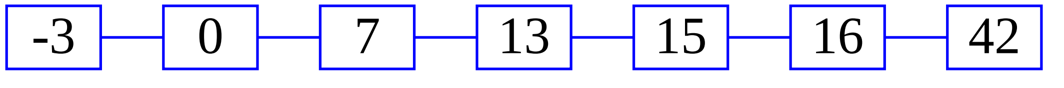

7
13
42
−3
(0<7)?
0
15
16
yes!


7
13
42
no!
−3
0
(15<7)?
15
16


−3
0
7
(15<13)?
13
42
15
16
no!


42
yes!
−3
0
7
13
(15<42)?
15
16

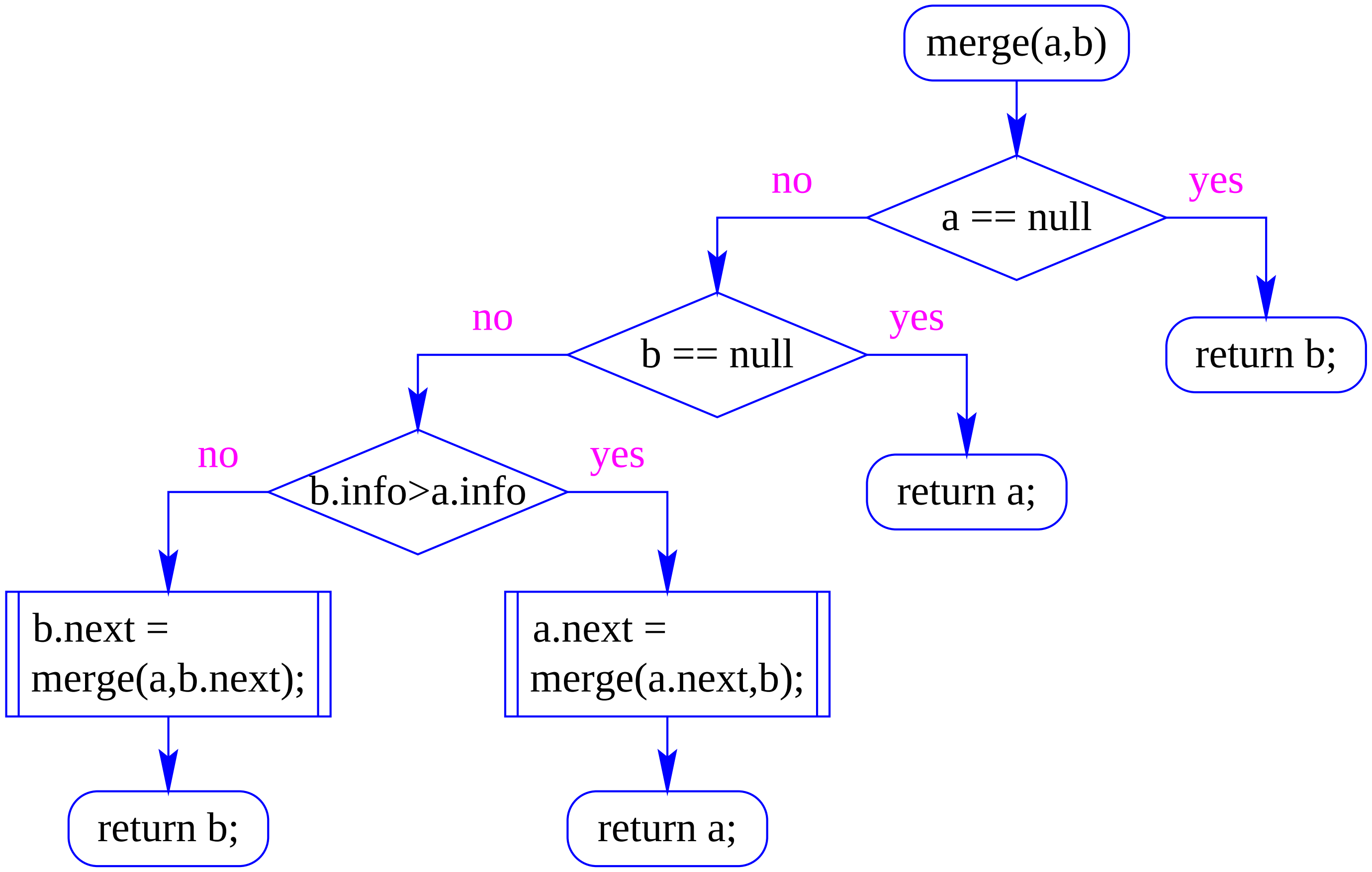

42
yes!
−3
0
7
13
15
(16<42)?
16


-3
0
7
13
15
16
42

Rekursive Implementierung:

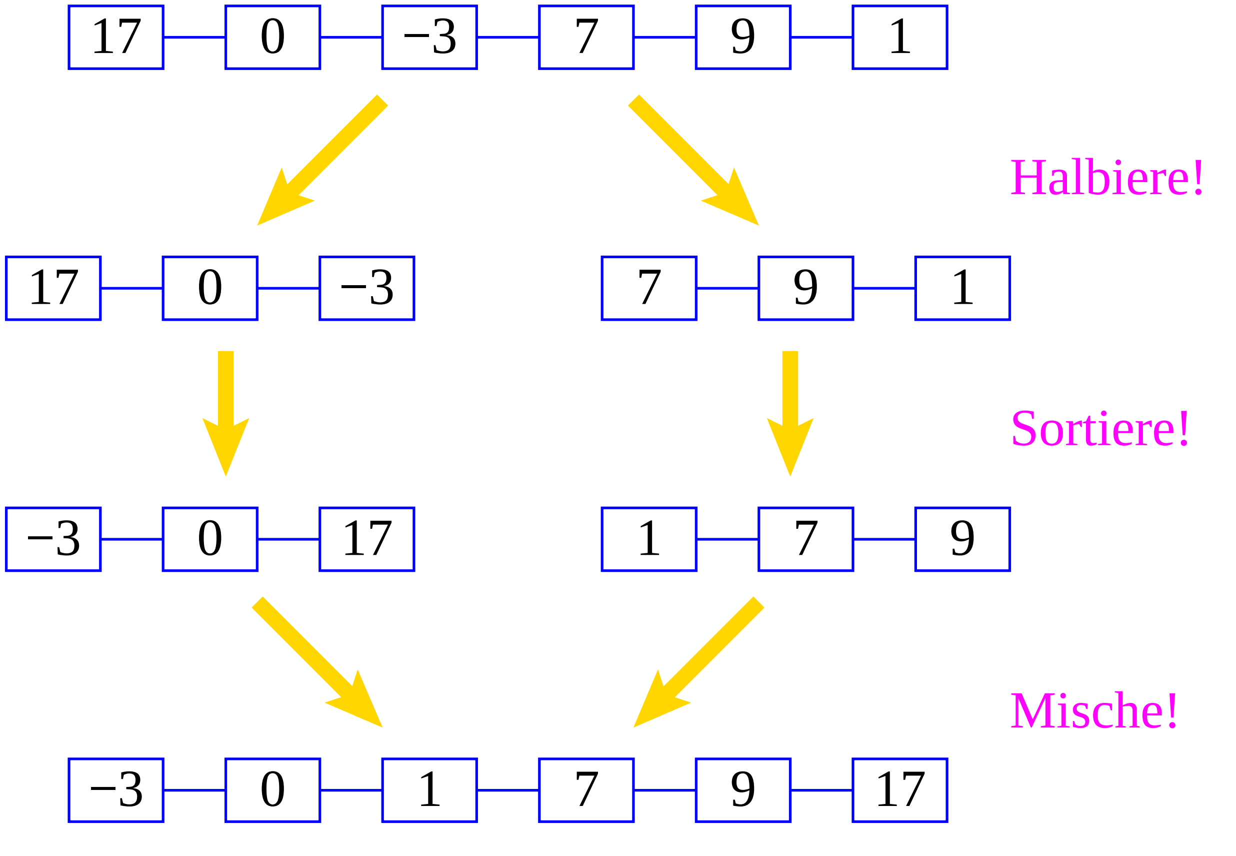
- Falls eine der beiden Listen `a` und `b` leer ist, geben wir die andere aus :-)
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente 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)
no
a == null
yes
return b;
no
b == null
yes
return a;
no
b.info>a.info
yes
b.next =
merge(a,b.next);
a.next =
merge(a.next,b);
return b;
return a;

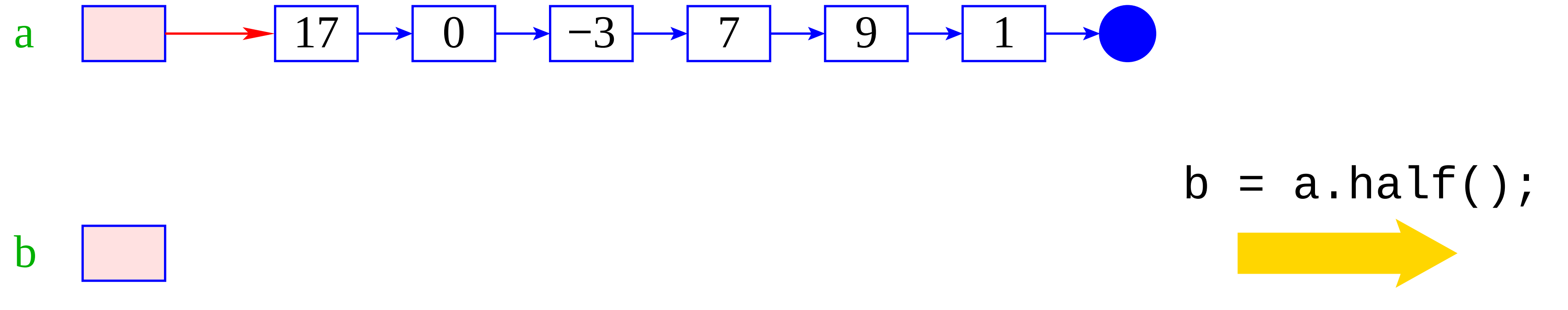
Sortieren durch Mischen:

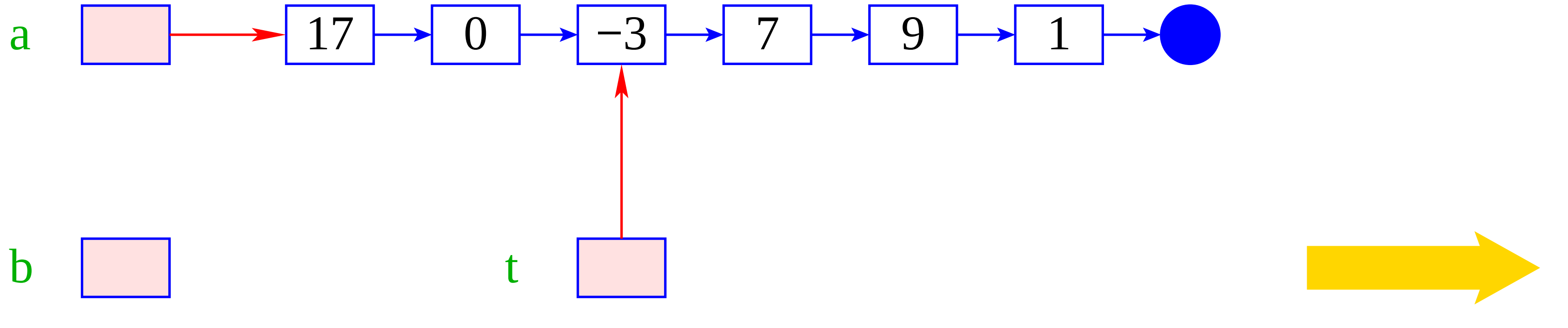
- Teile zu sortierende Liste in zwei Teil-Listen;
- sortiere jede Hälfte für sich;
- mische die Ergebnisse!

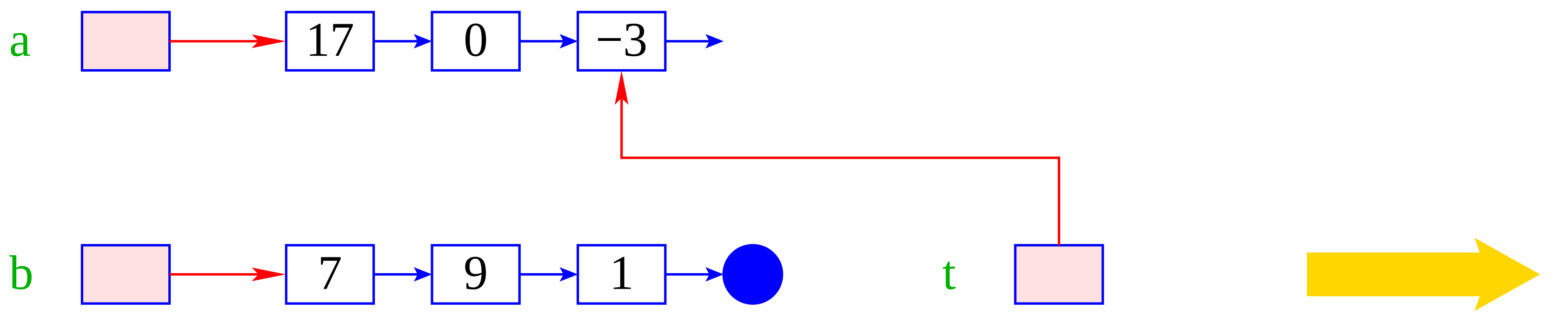

17
0
−3
7
9
1
Halbiere!
17
0
−3
7
9
1
Sortiere!
−3
0
17
1
7
9
Mische!
−3
0
1
7
9
17

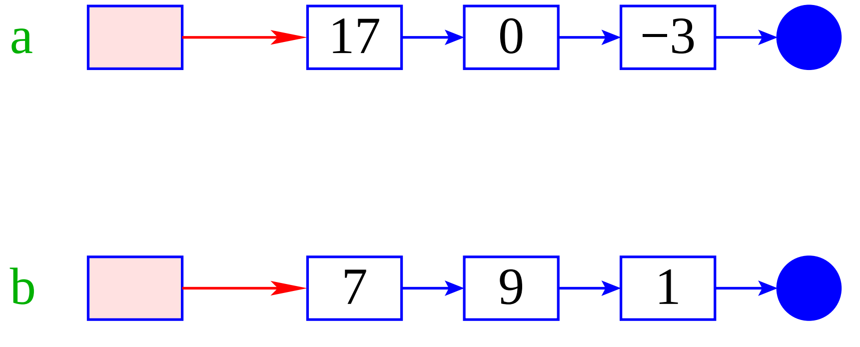
```
public static List sort(List a) {
   if (a == null || a.next == null)
      return a;
   List b = a.half(); // Halbiere!
   a = sort(a);
   b = sort(b);
   return merge(a,b);
   }
```

```
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:

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

- 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 ?-))

11.2 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.

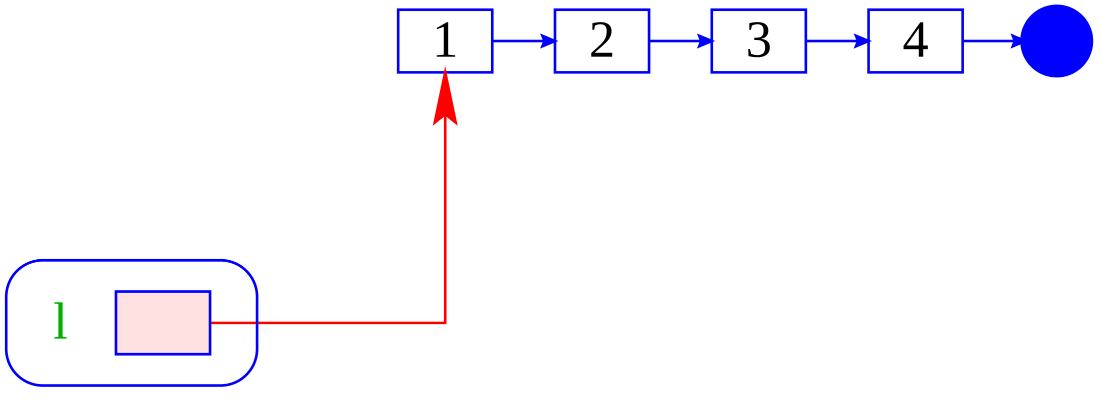
Weiterhin müssen wir einen leeren Keller anlegen können.

Modellierung:

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

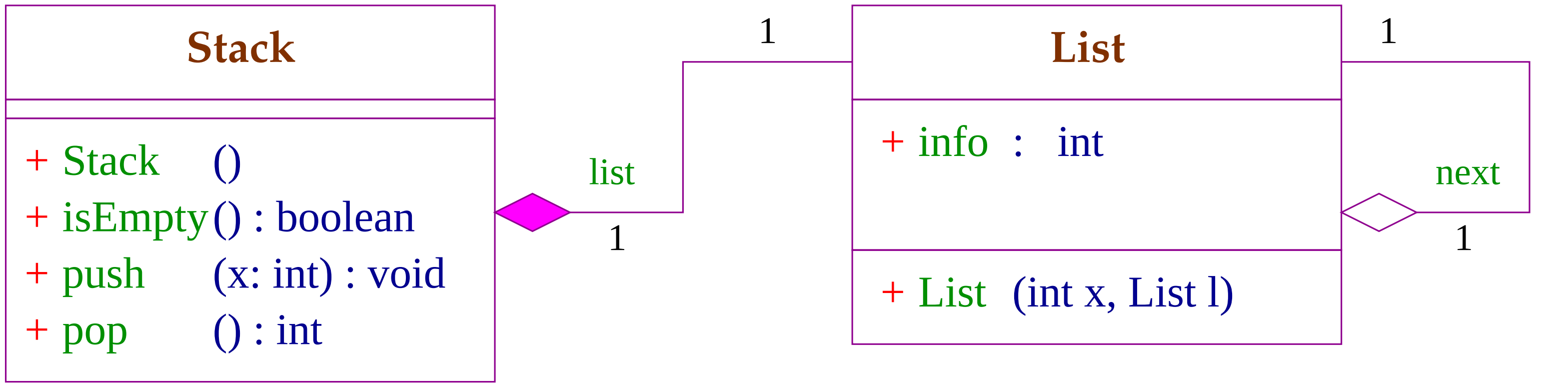
Erste Idee:

- 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;

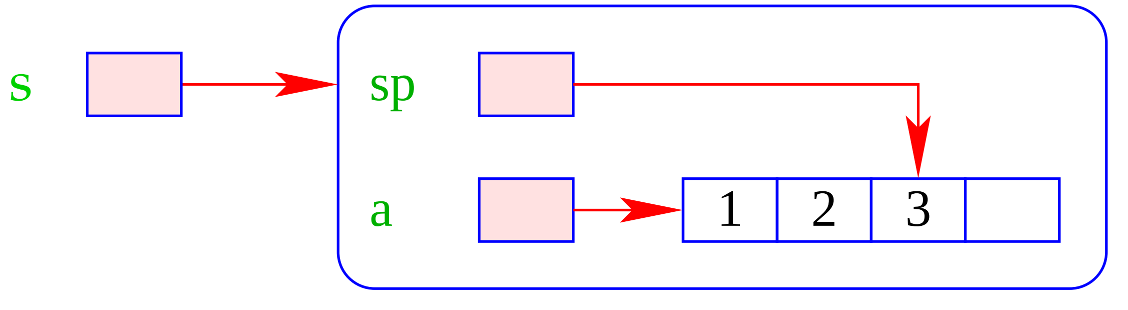
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms!

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

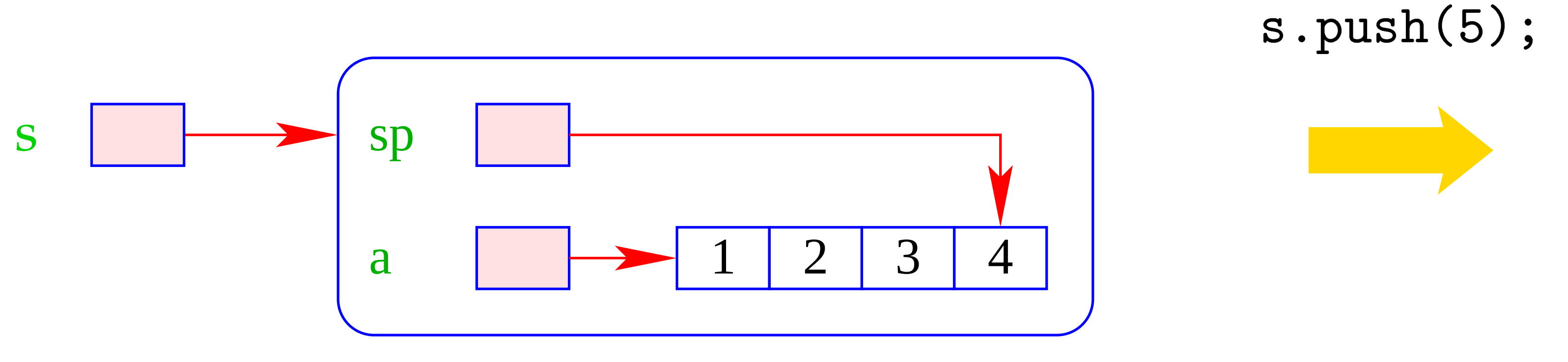
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms!

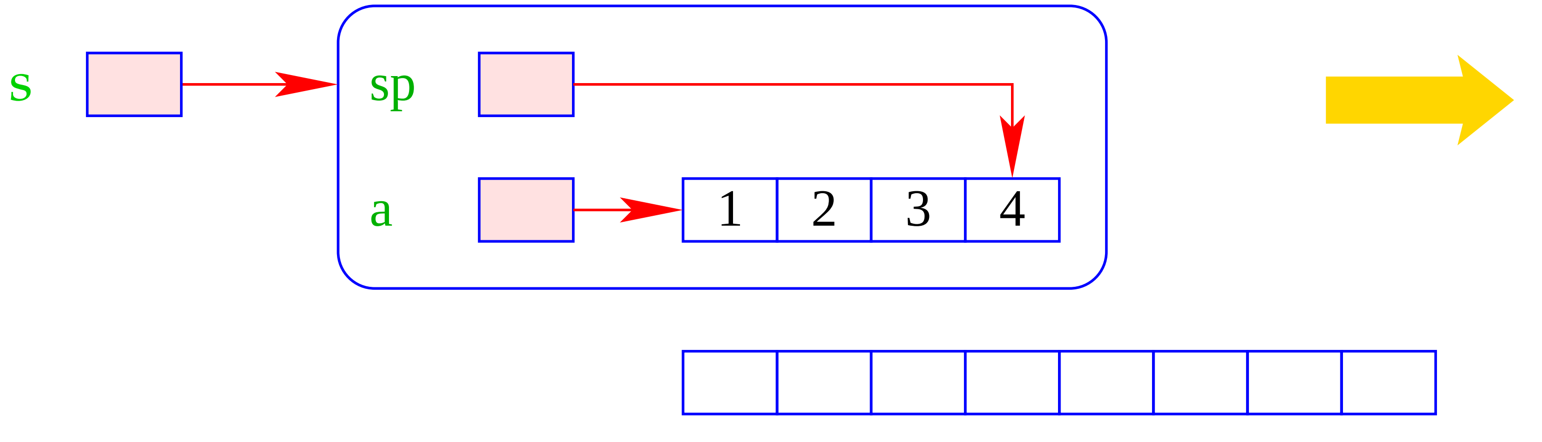
Zweite Idee:

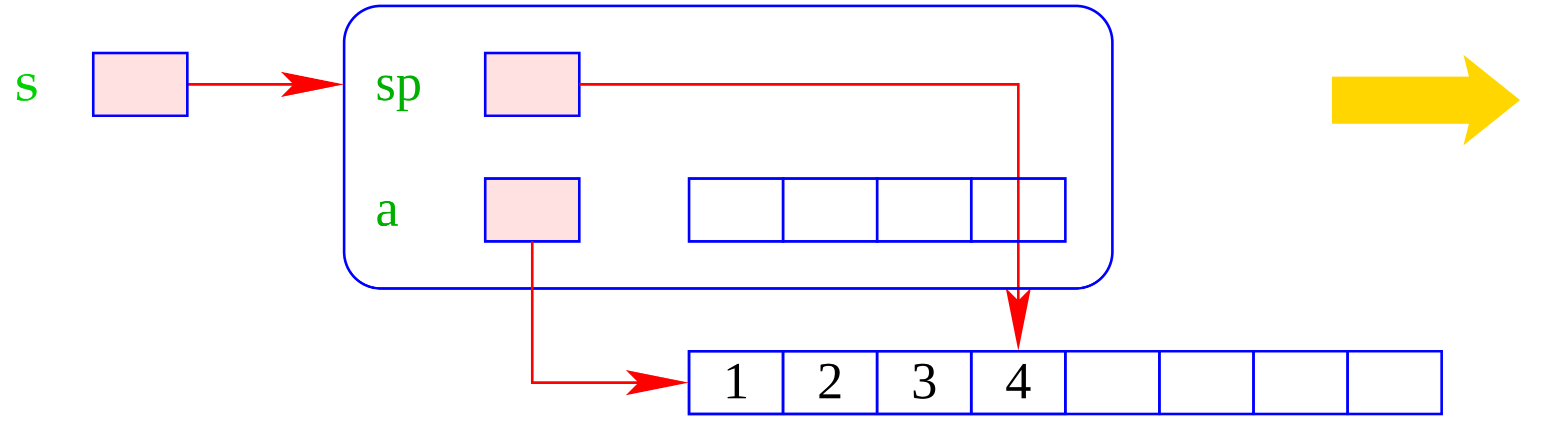
- Realisiere den Keller mithilfe eines Felds und eines Stackpointers, der auf die oberste belegte Zelle zeigt.
- Läuft das Feld über, ersetzen wir es durch ein größeres :-)



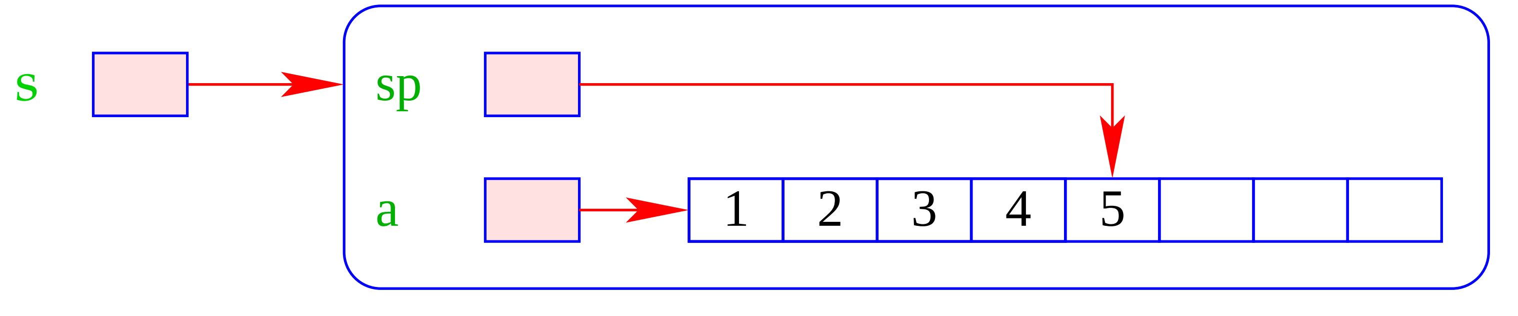

```
s.push(4);
```


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

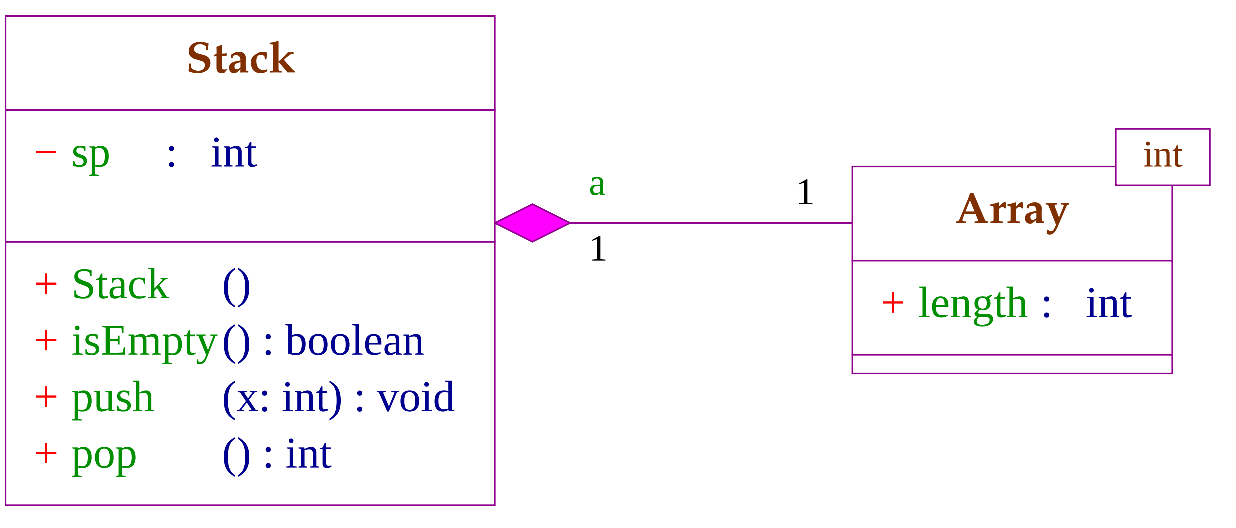

s
sp
a
1
2
3
4



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);
      }
   ...
```

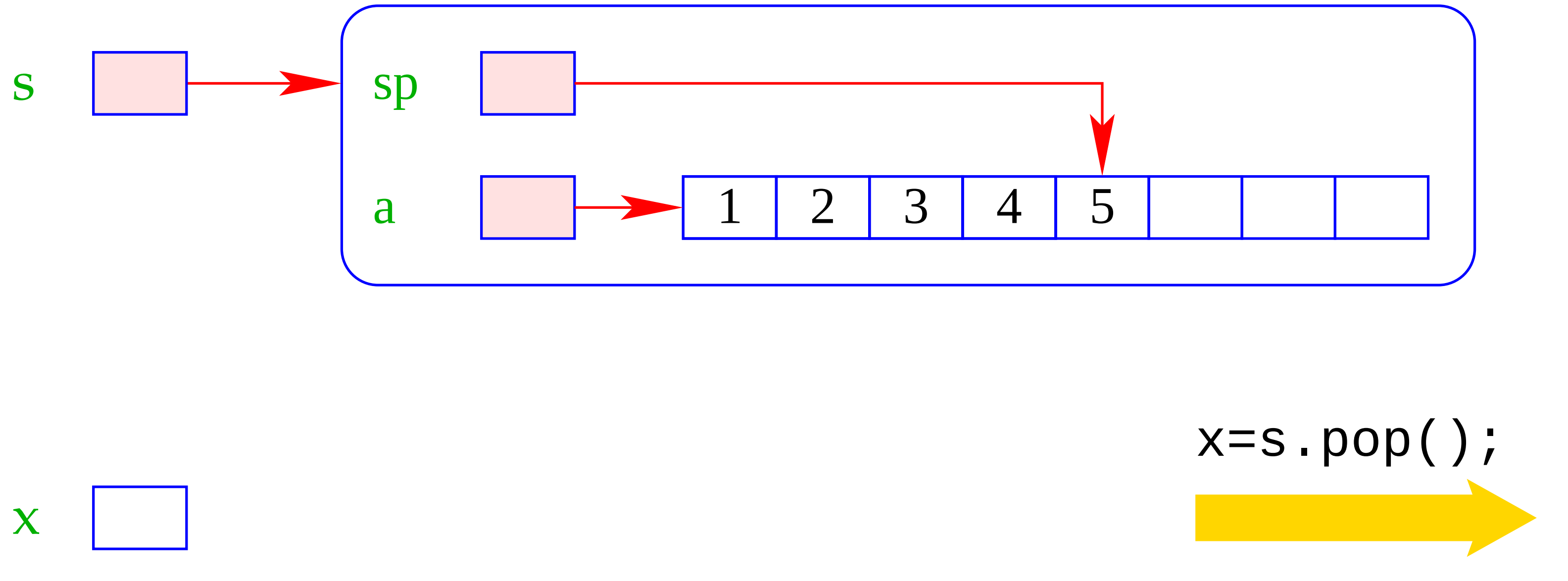
```
   public int pop() {
      return a[sp--];
      }
   public void push(int x) {
      ++sp;
      if (sp == a.length) {
         int[] b = new int[2*sp];
         for(int i=0; i<sp; ++i) b[i] = a[i];
         a = b;
         }
      a[sp] = x;
      }
   public toString() {...}
} // end of class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

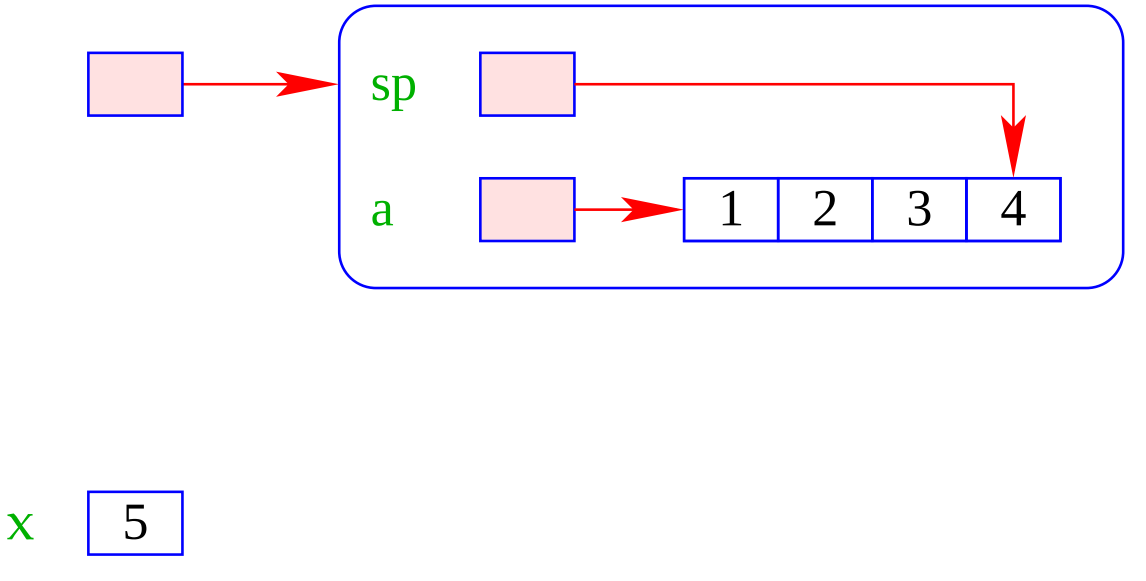
Nachteil:

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

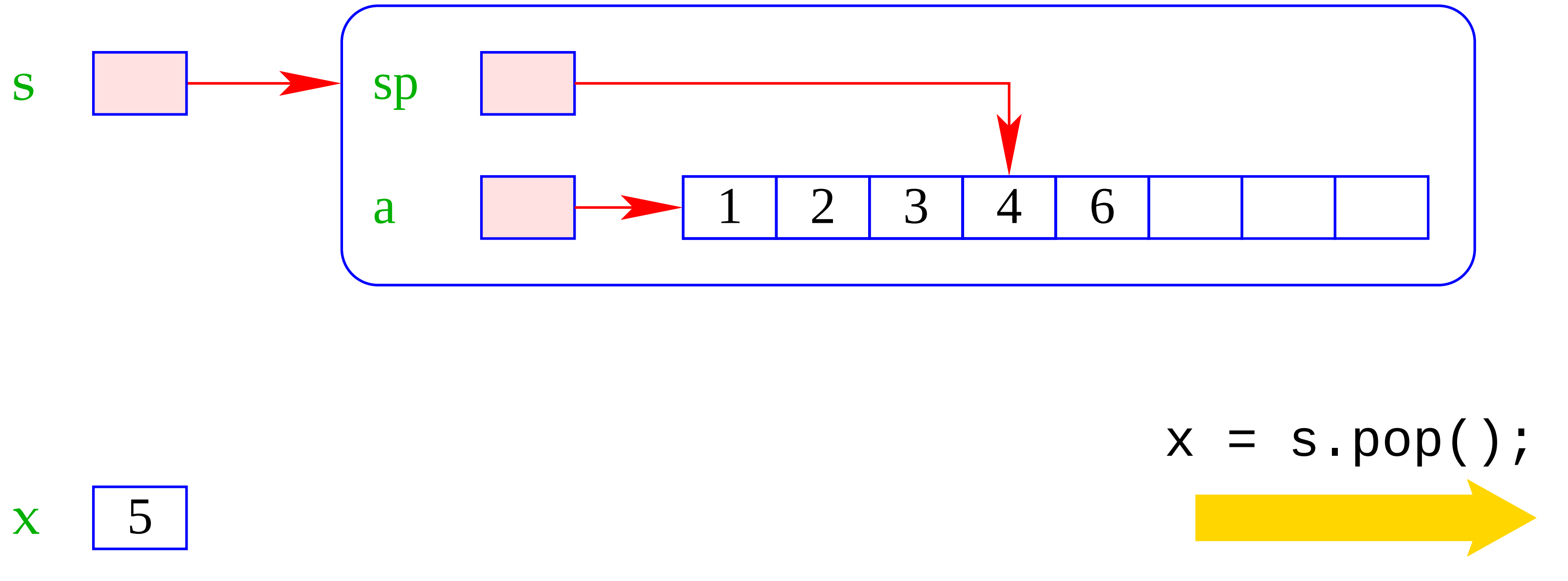
Erste Idee:

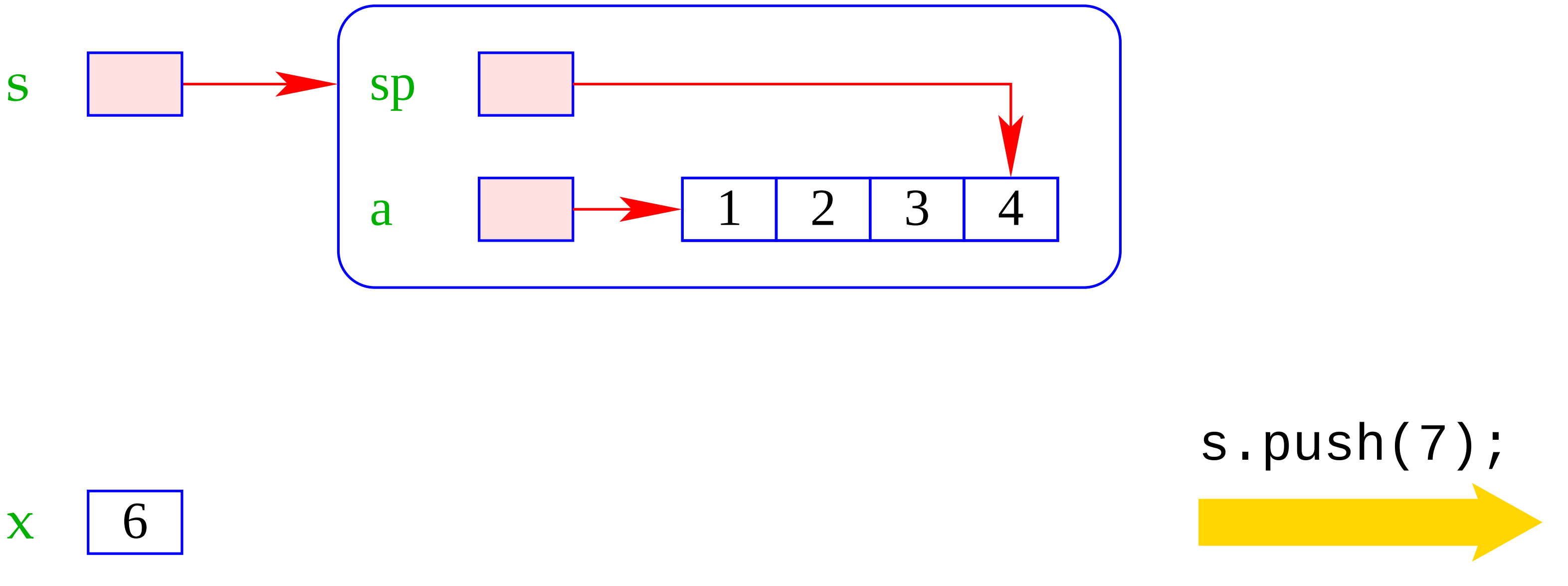
- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei ...

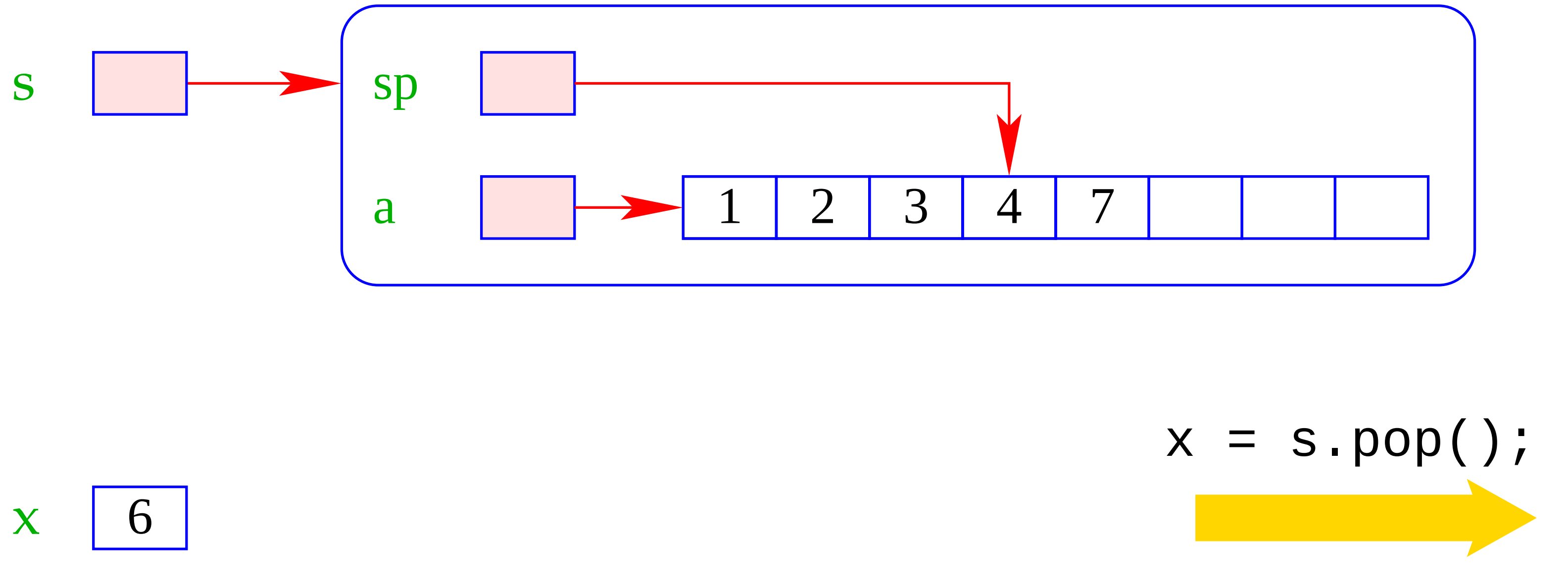

s
sp
a
1
2
3
4
5
x
x=s.pop();



`s.push(6);`


s
sp
a
1
2
3
4
6
x
5
x = s.pop();

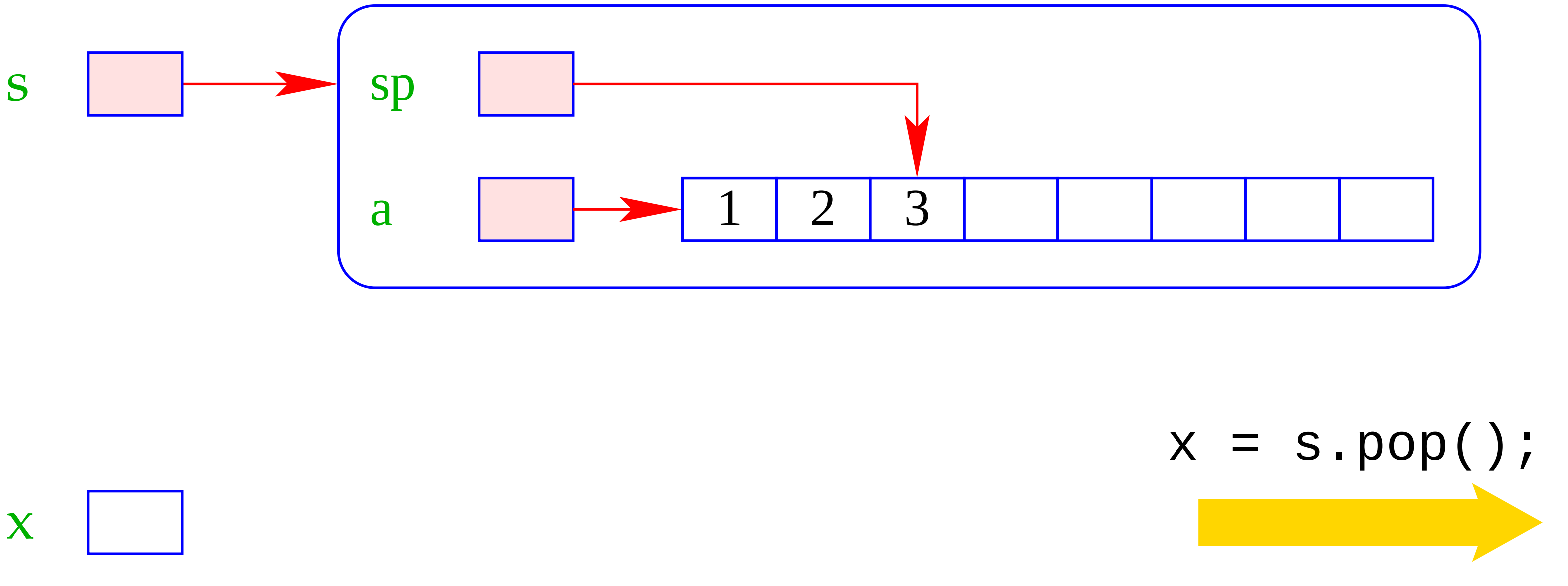

s
sp
a
1
2
3
4
x
6
s.push(7);

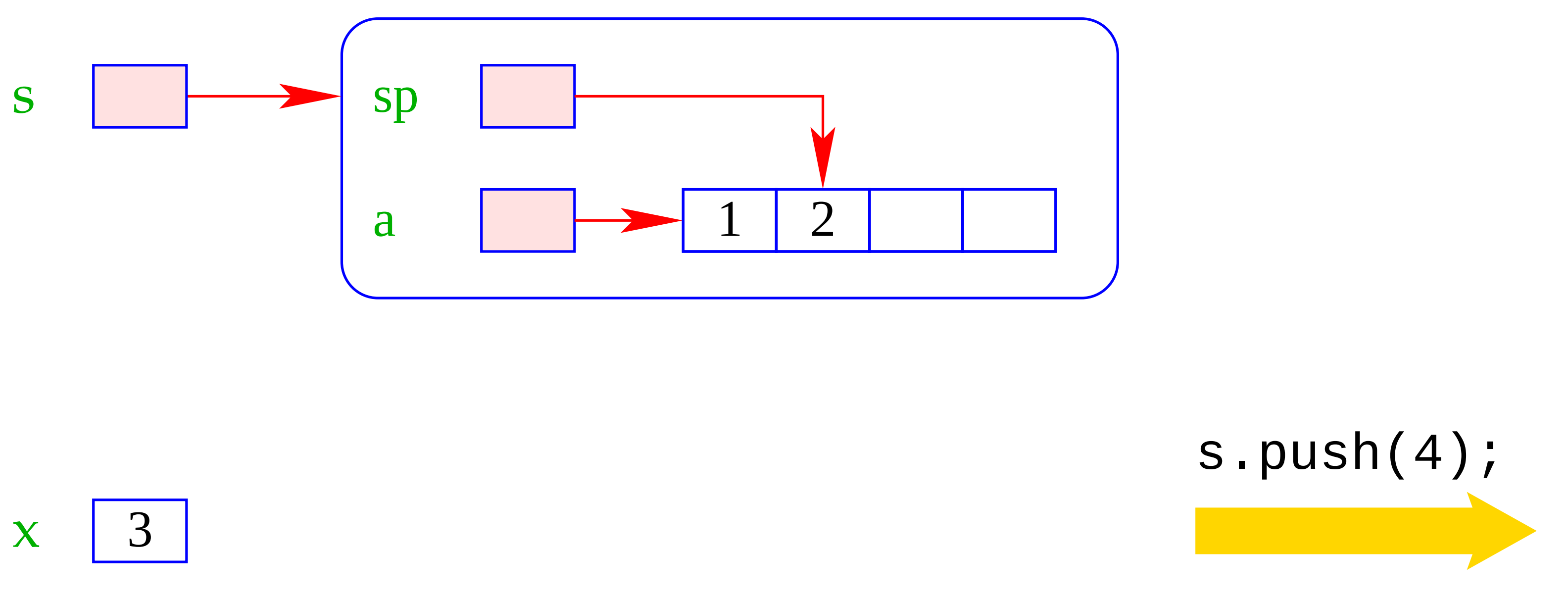

s
sp
a
1
2
3
4
7
x
6
x = s.pop();

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

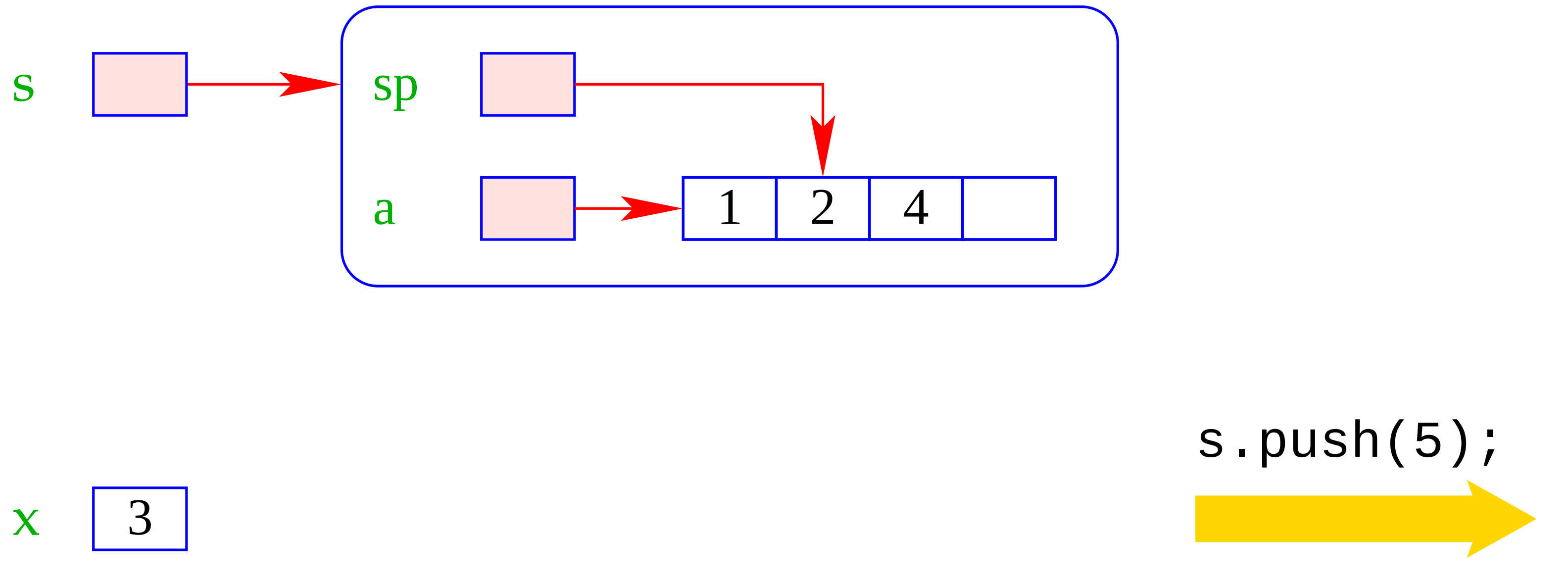
Zweite Idee:

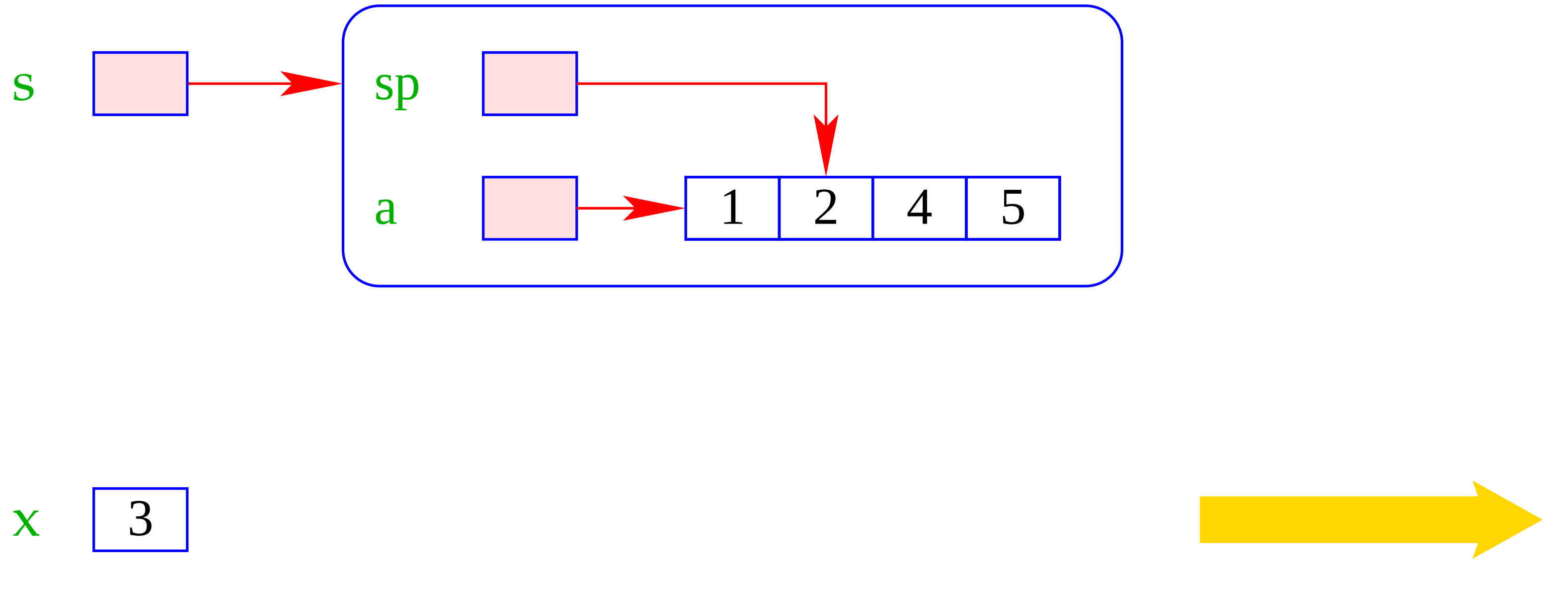
- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte !


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



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


s
sp
a
1
2
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;
   }
```

11.3 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.

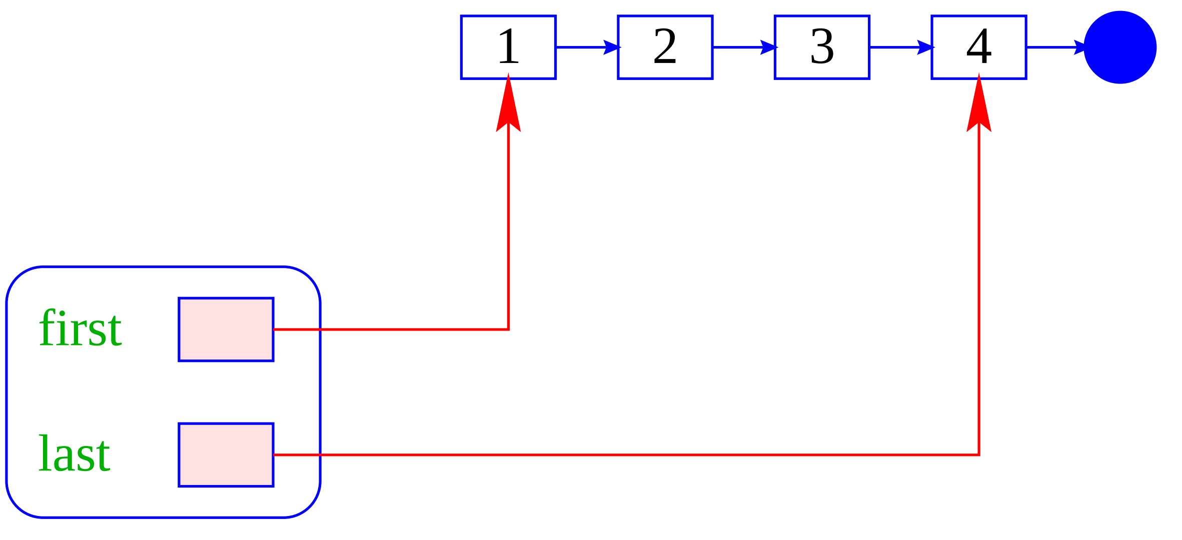
Weiterhin müssen wir eine leere Schlange anlegen können :-)

Modellierung:

Queue
+ Queue ()   + isEmpty() : boolean   + enqueue(x: int) : void   + dequeue() : int

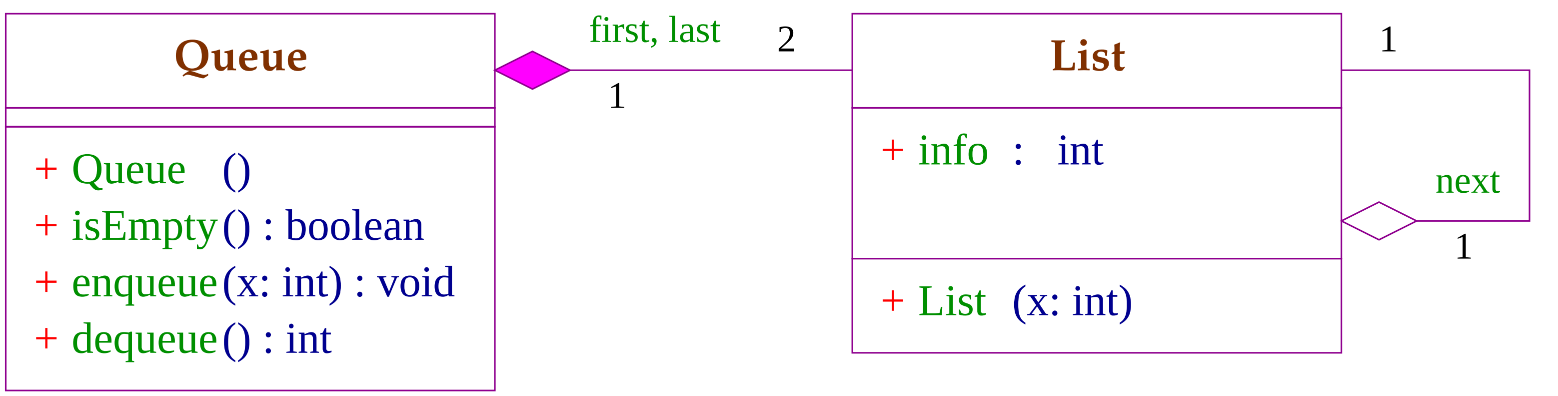
Erste Idee:

- Realisiere Schlange mithilfe einer Liste :



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

Modellierung:



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

Implementierung:

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

```
   public int dequeue () {
      int result = first.info;
      if (last == first) last = null;
      first = first.next;
      return result;
      }
   public void enqueue (int x) {
      if (first == null) first = last = new List(x);
      else { last.next = new List(x); last = last.next; }
      }
   public String toString() {
      return List.toString(first);
      }
} // end of class Queue
```

- Die Implementierung ist wieder sehr einfach :-)
- ... nutzt ebenfalls kaum Features von `List` aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut

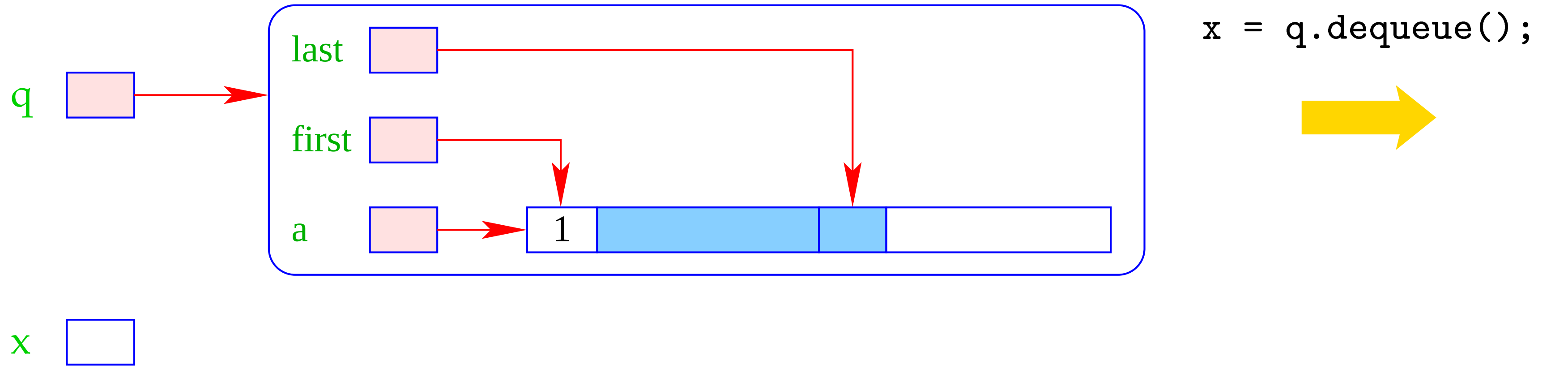
$\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms :-(

- Die Implementierung ist wieder sehr einfach :-)
- ... nutzt ebenfalls kaum Features von `List` aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut

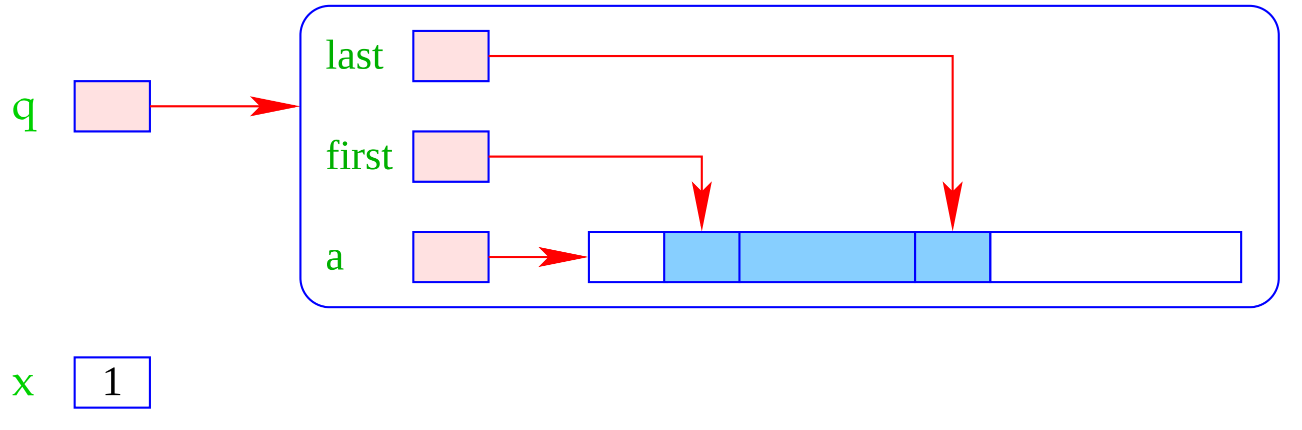
  $\Longrightarrow$ führt zu schlechtem ↑Cache-Verhalten des Programms :-(

Zweite Idee:

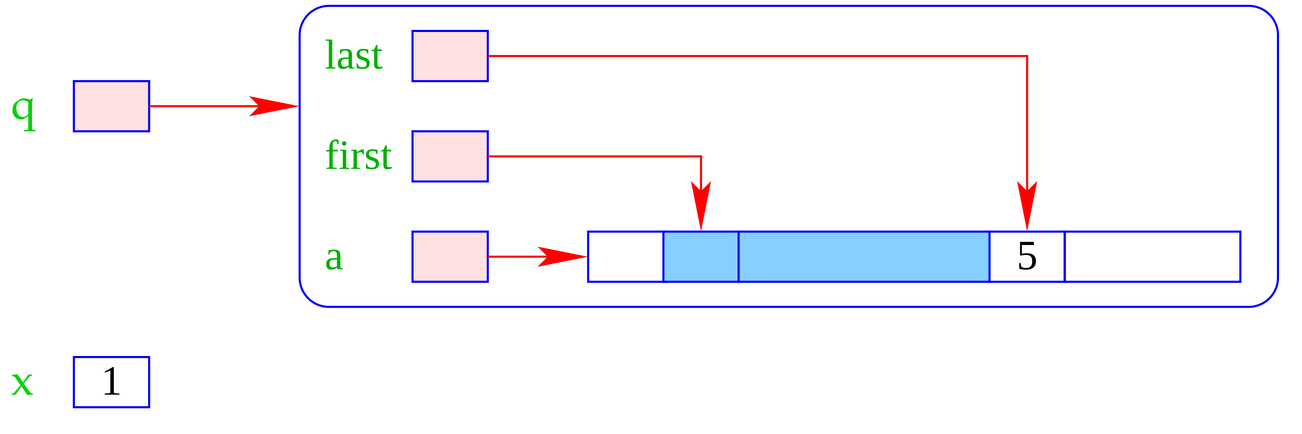
- Realisiere die Schlange mithilfe eines Felds und zweier Pointer, die auf das erste bzw. letzte Element der Schlange zeigen.
- Läuft das Feld über, ersetzen wir es durch ein größeres.

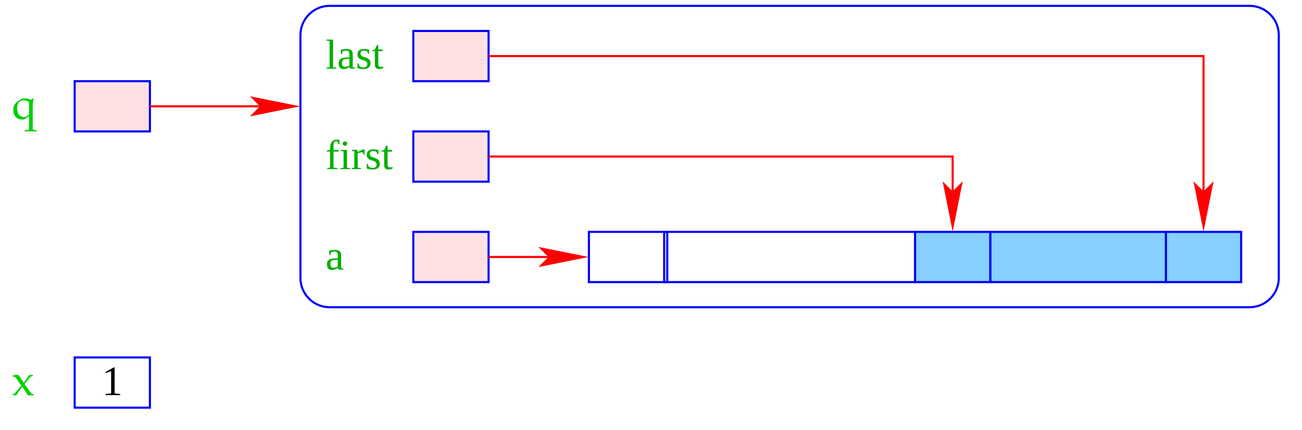


last
q
first
a
1
x
x = q.dequeue();

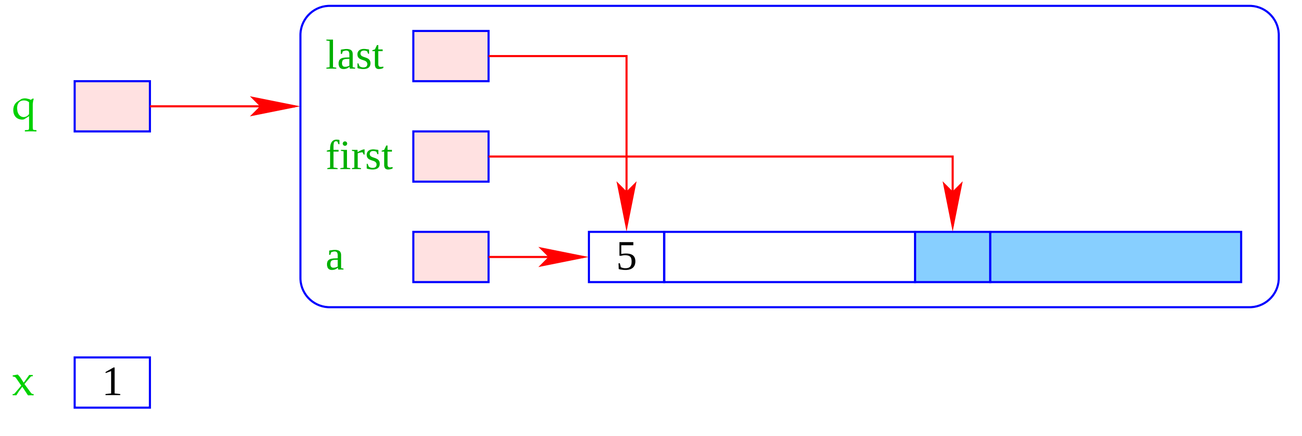



```
x = q.enqueue(5);
```


last
q
first
a
5
x
1

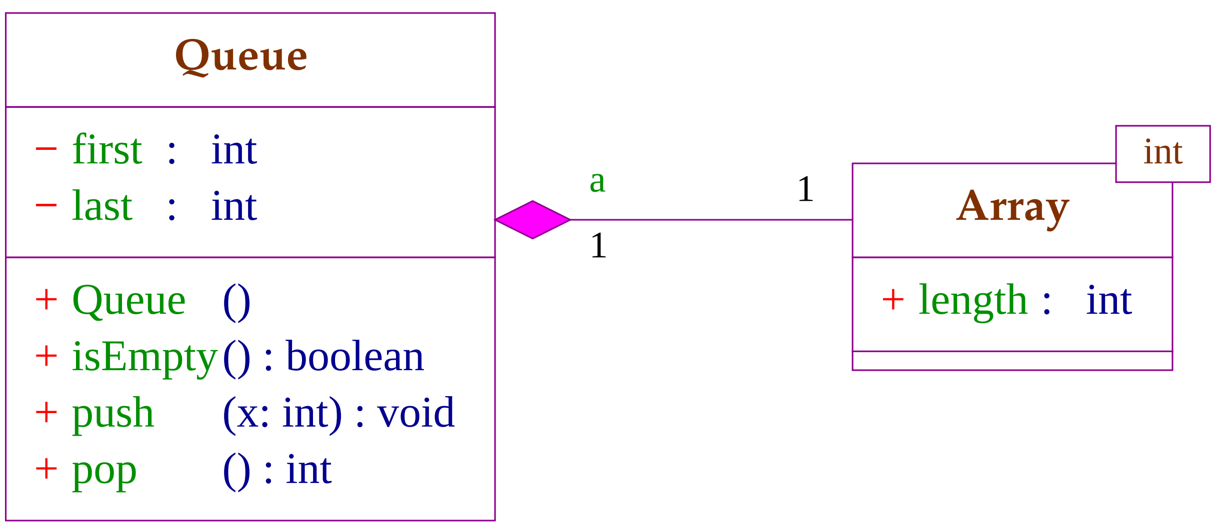


`x = q.enqueue(5);`



last
q
first
a
5
x
1

Modellierung:



Implementierung:

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

Implementierung von `enqueue()`:

- Falls die Schlange leer war, muss `first` und `last` auf `0` gesetzt werden.
- Andernfalls ist das Feld `a` genau dann voll, wenn das Element `x` an der Stelle `first` eingetragen werden sollte.
- In diesem Fall legen wir ein Feld doppelter Größe an.

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

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

Implementierung von `dequeue()`:

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

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

Diskussion:

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

  Achtung:

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

Zusammenfassung:

- Der Datentyp `List` ist nicht sehr abstract, dafür extrem flexibel

  $\Longrightarrow$ gut geeignet für rapid prototyping.

- Für die nützlichen (eher :-) abstrakten Datentypen `Stack` und `Queue` lieferten wir zwei Implementierungen:

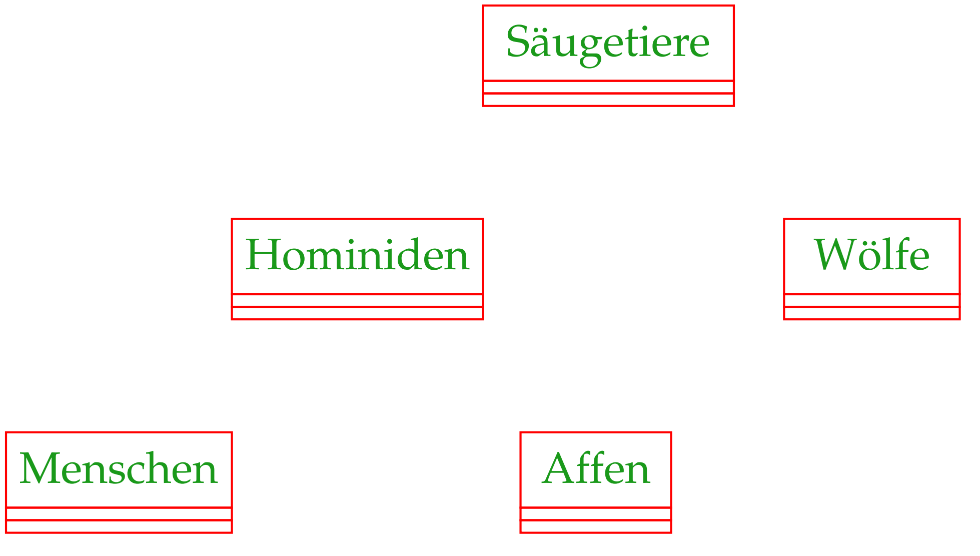
Technik	Vorteil	Nachteil
`List`	einfach	nicht-lokal
int[]	lokal	etwas komplexer

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

12 Vererbung

Beobachtung:

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

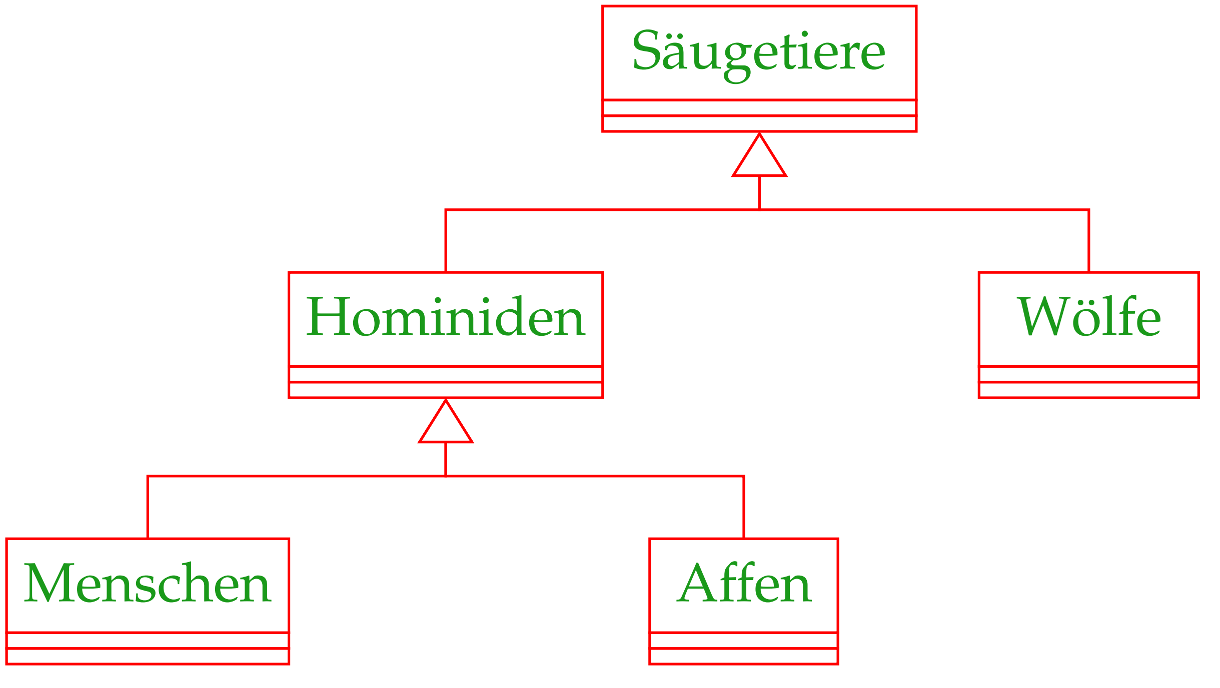


Idee:

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

$\Longrightarrow$ inkrementelles Programmieren

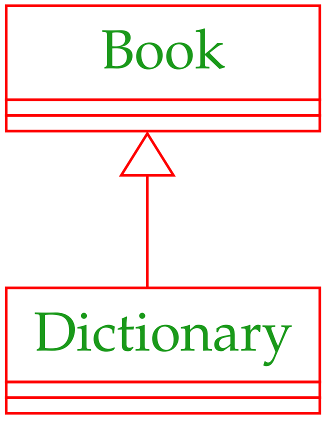
$\Longrightarrow$ Software Reuse


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:

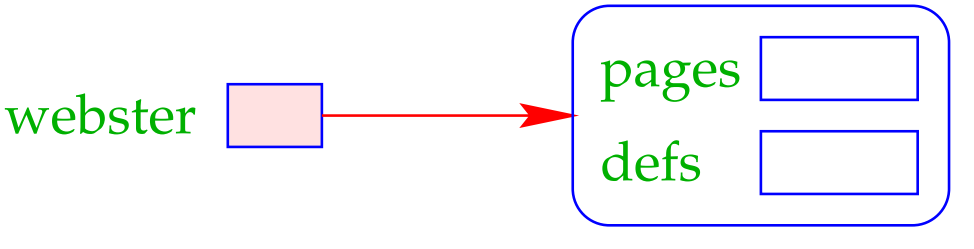
```
public class Book {
   protected int pages;
   public Book() {
      pages = 150;
   }
   public void page_message() {
      System.out.print("Number of pages:\t"+pages+"\n");
      }
   } // end of class Book
   ...
```

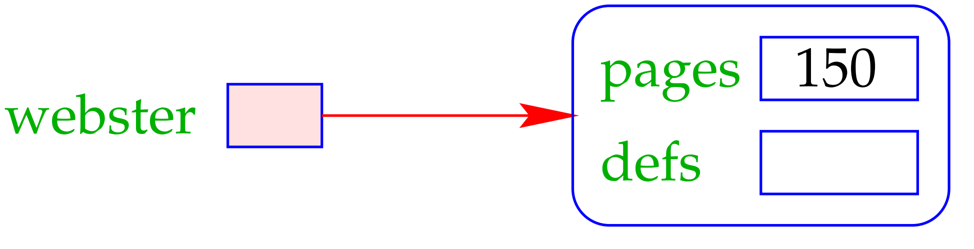
```
public class Dictionary extends Book {
   private int defs;
   public Dictionary(int x) {
      pages = 2*pages;
      defs = x;
      }
   public void defs_message() {
      System.out.print("Number of defs:\t\t"+defs+"\n");
      System.out.print("Defs per page:\t\t"+defs/pages+"\n");
      }
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

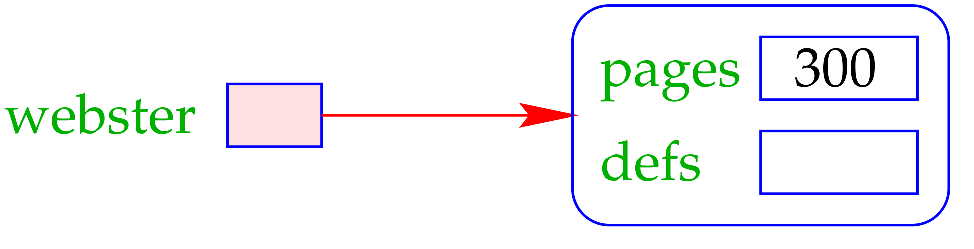
- `class A extends B { ... }` deklariert die Klasse A als Unterklasse der Klasse B.
- Alle Members von B stehen damit automatisch auch der Klasse A zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse sichtbar.
- Als `private` deklarierte Members können dagegen in der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

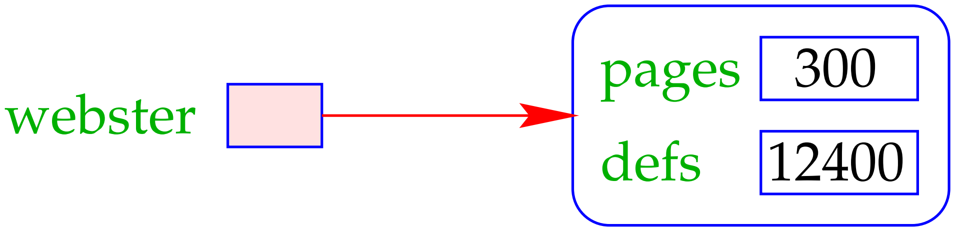
`Dictionary webster = new Dictionary(12400);` liefert:

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
```

12.1 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.

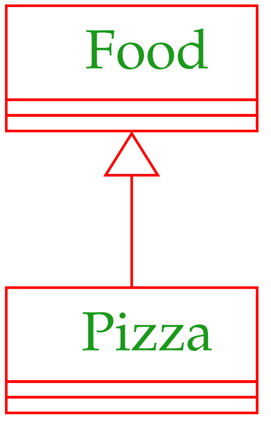
```
public class Words {
   public static void main(String[] args) {
      Dictionary webster = new Dictionary(540,36600);
      webster.message();
   } // end of main
} // end of class Words
```

- 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
```

12.2 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
```

```java
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.

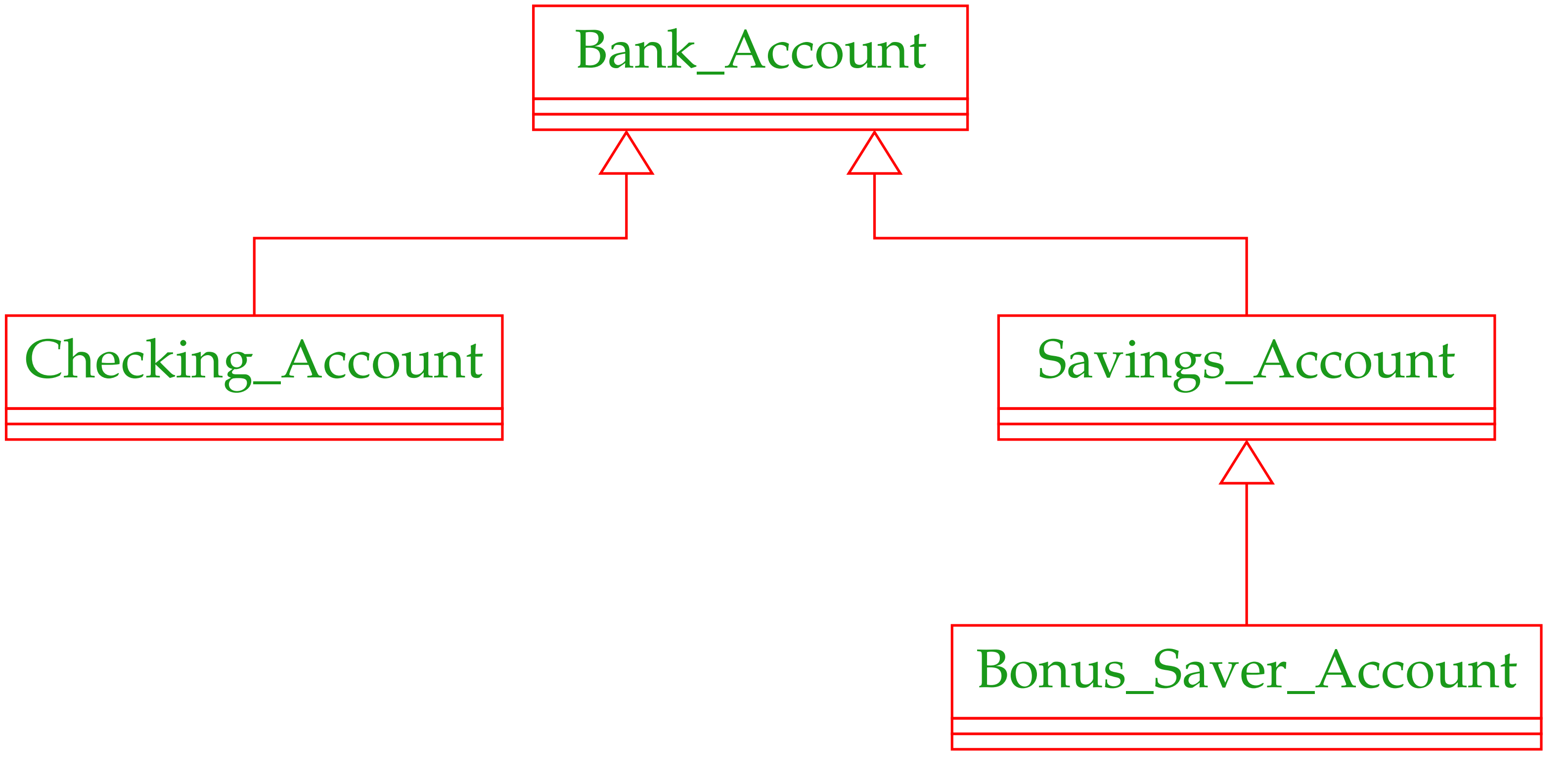
```
public class Pizza extends Food {
   public Pizza (int amount_fat) {
      super (amount_fat,8);
      }
   } // end of class Pizza
```

- 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`

12.3 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
```

13 Abstrakte Klassen, finale Klassen und Interfaces

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

Beispiel: Implementierung der JVM

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

- Die Unterklassen von `Instruction` repräsentieren die Befehle der JVM.
- Allen Unterklassen gemeinsam ist eine Objekt-Methode `execute()` – immer mit einer anderen Implementierung :-)
- Die statischen Variablen der Oberklasse stehen sämtlichen Unterklassen zur Verfügung.

- Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- Für die abstrakte Methode muss der vollständige Kopf angegeben werden – inklusive den Parameter-Typen und den (möglicherweise) geworfenen Exceptions.
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `boolean halted()`.
- Die angebene Implementierung liefert eine Default-Implementierung für `boolean halted()`.
- Klassen, die eine andere Implementierung brauchen, können die Standard-Implementierung ja überschreiben :-)

- Die Methode `execute()` soll die Instruktion ausführen und als Rückgabe-Wert den `pc` des nächsten Befehls ausgeben.

Beispiel für eine Instruktion:

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

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

... andere Instruktionen:

```
public final class Sub extends Instruction {
   public int execute() {
      final int y = stack.pop();
      final int x = stack.pop();
      stack.push(x-y); return ++pc;
   }  // end of execute()
}     // end of class Sub
public final class Halt extends Instruction {
   public boolean halted() {
      pc=0; stack = new IntStack(); return true;
      }
   public int execute() { return 0; }
}     // end of class Halt
```

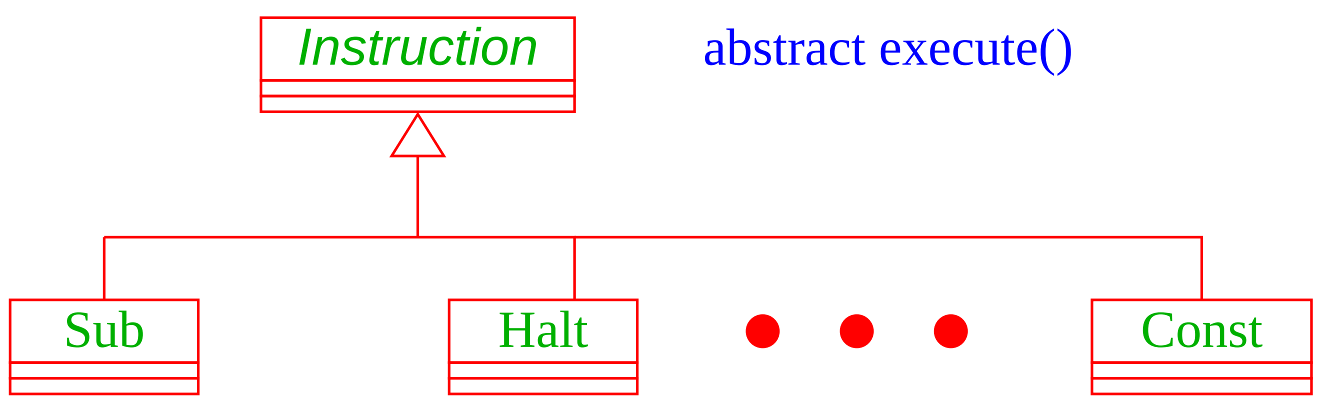
- In der Klasse `Halt` wird die Objekt-Methode `halted()` neu definiert.
- Achtung bei `Sub` mit der Reihenfolge der Argumente!

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

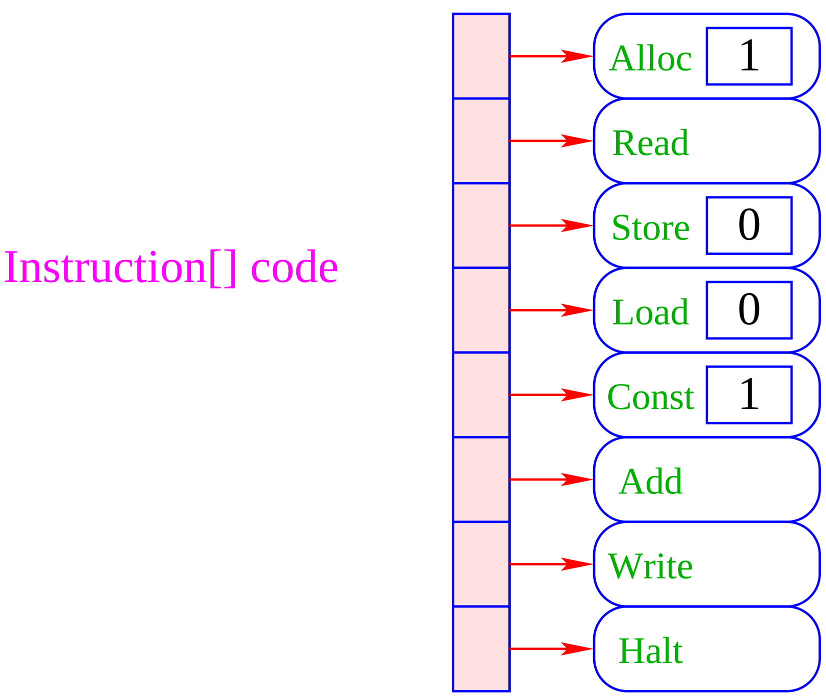
```
public static void main(String[] args) {
   Instruction[] code = getCode();
   Instruction ir = code[0];
   while(!ir.halted())
      ir = code[ir.execute()];
   }
```

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

Die abstrakte Klasse `Instruction`:



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

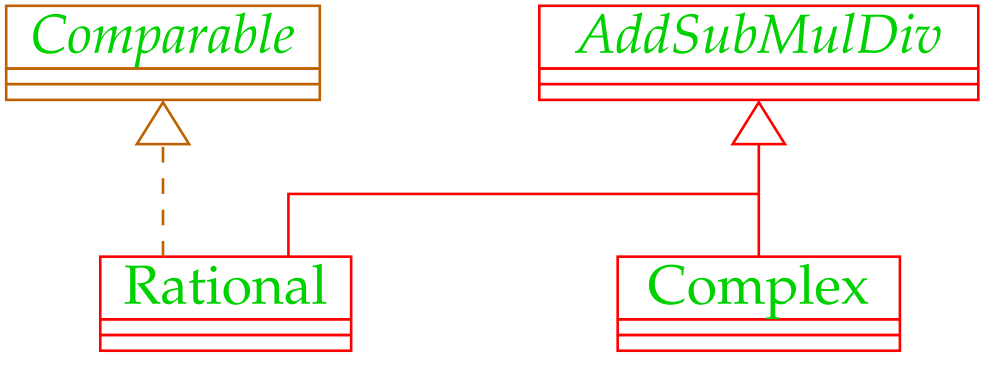


- Die Interpreter-Schleife ruft eine Methode `execute()` für die Elemente dieses Felds auf.
- Welche konkrete Methode dabei jeweils aufgerufen wird, hängt von der konkreten Klasse des jeweiligen Objekts ab, d.h. entscheidet sich erst zur Laufzeit.
- Das nennt man auch dynamische Bindung.

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

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

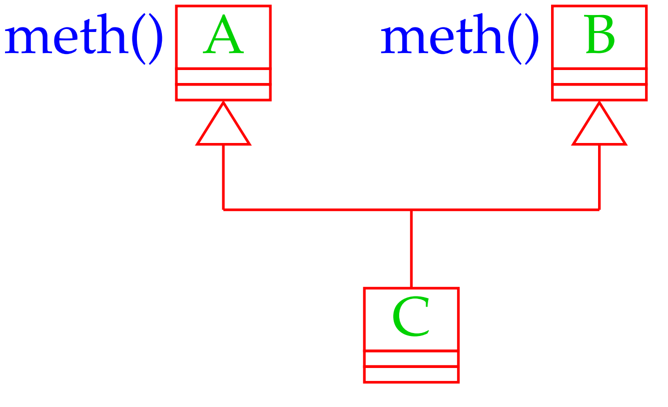
Beispiel:



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

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

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

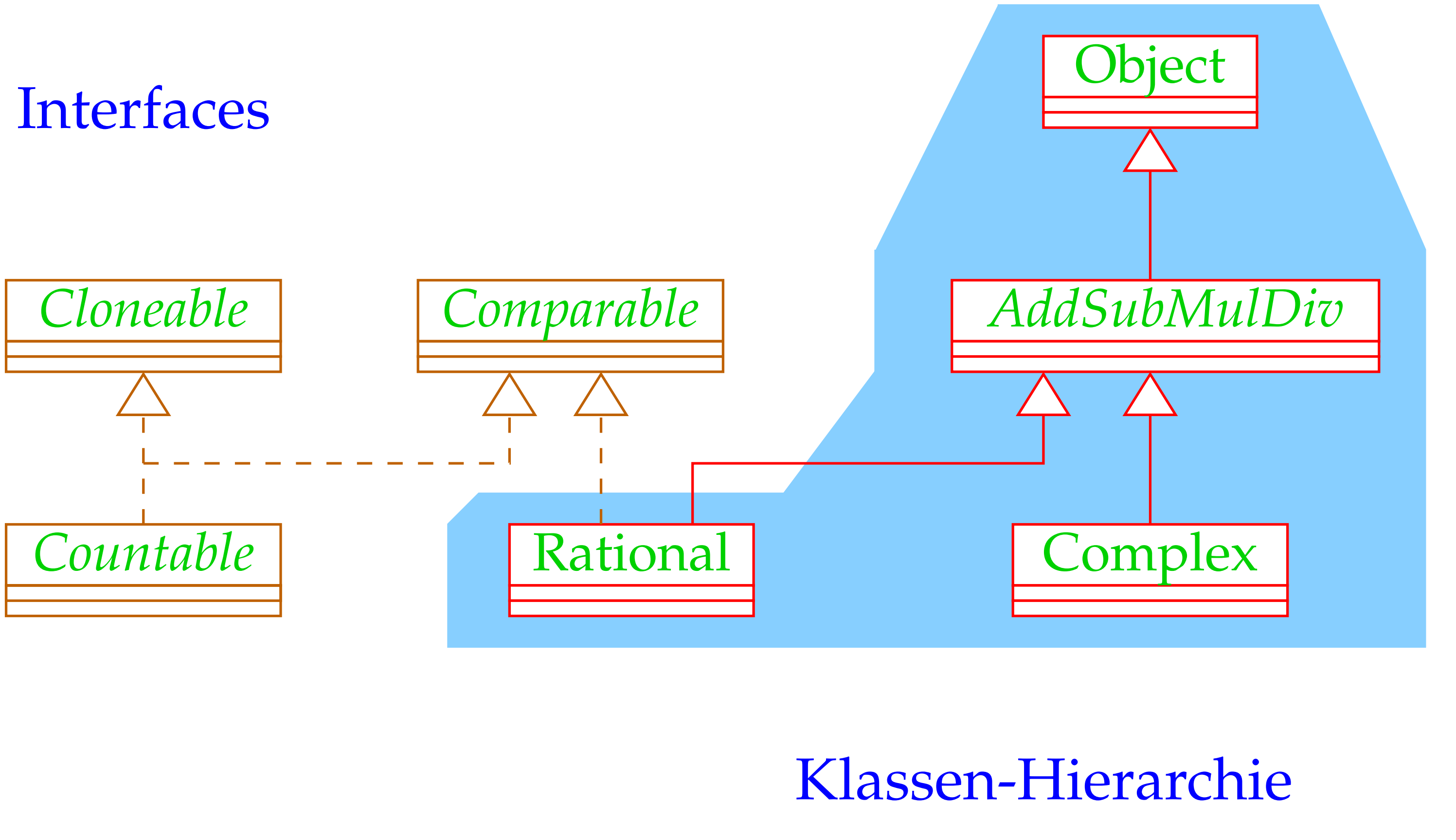
- Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- Erweiternde Interfaces können Konstanten auch umdefinieren...
- (kommen Konstanten gleichen Namens in verschiedenen implementierten Interfaces vor, gibt's einen Laufzeit-Fehler...)

Beispiel (Forts.):

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

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

Übersicht:


Interfaces
Object
Cloneable
Comparable
AddSubMulDiv
Countable
Rational
Complex
Klassen-Hierarchie

14 Polymorphie

Problem:

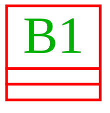
- Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur grade nochmal definieren :-(

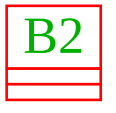
14.1 Unterklassen-Polymorphie

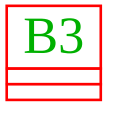
Idee:

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

Statt:

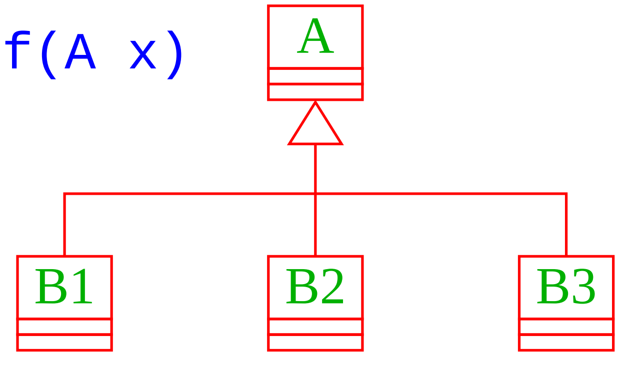
`f(B1 x)` 

`f(B2 x)` 

`f(B3 x)` 

... besser:

`f(A x)`



Fakt:

- Die Klasse `Object` ist eine gemeinsame Oberklasse für alle Klassen.
- Eine Klasse ohne angegebene Oberklasse ist eine direkte Unterklasse von `Object`.
- Einige nützliche Methoden der Klasse `Object` :
  - `String toString()` liefert (irgendeine) Darstellung als `String`;
  - `boolean equals(Object obj)` testet auf Objekt-Identität oder Referenz-Gleichheit:

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

    ...

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

Achtung:

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

Beispiel:

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

... liefert:

```
Hello World!
```

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

Bemerkung:

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

Achtung:

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

... liefert ...

... einen Compiler-Fehler:

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

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

Ausweg:

- Benutze einen expliziten cast in die entsprechende Unterklasse!

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

Fazit:

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

Beispiel: Unsere Listen

```
public class List {
   public Object info;
   public List next;
   public List(Object x, List l) {
      info=x; next=l;
   }
   public void insert(Object x) {
      next = new List(x,next);
   }
   public void delete() {
      if (next!=null) next=next.next;
   }
   ...
```

```
    public String toString() {
        String result = "["+info;
        for (List t=next; t!=null; t=t.next)
            result=result+", "+t.info;
        return result+"]";
    }
    ...
} // end of class List
```

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

... aber Achtung:

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

liefert ...

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

Stattdessen müssen wir schreiben:

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

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

14.2 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 !!!
- 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; }
}
```

14.3 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 :-)

14.3 Wrapper-Klassen

... bleibt ein Problem:

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

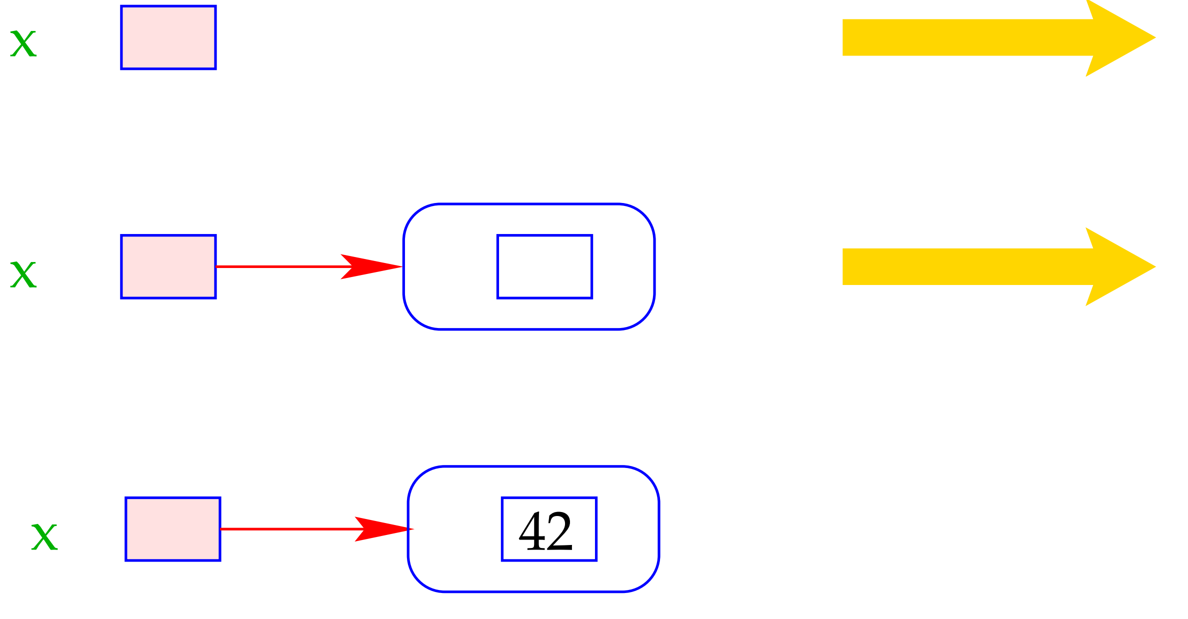
  (Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl :-)

Ausweg:

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

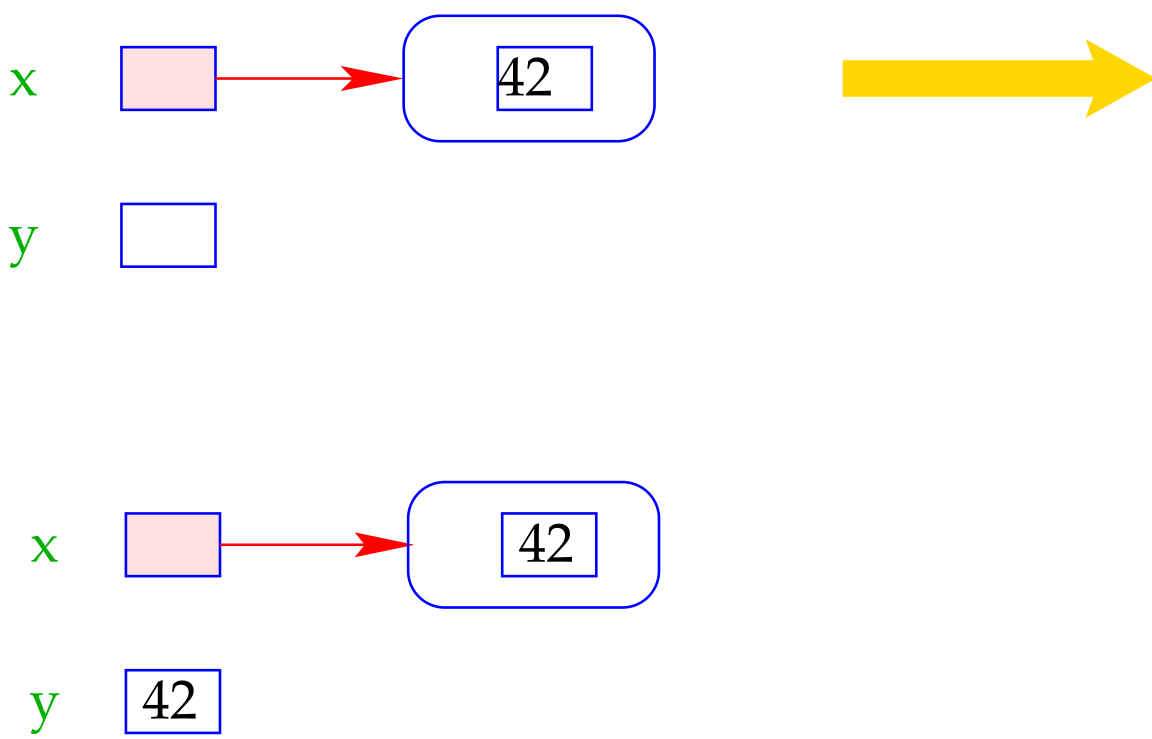
  $\Longrightarrow$ Wrapper-Objekte aus Wrapper-Klassen.

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

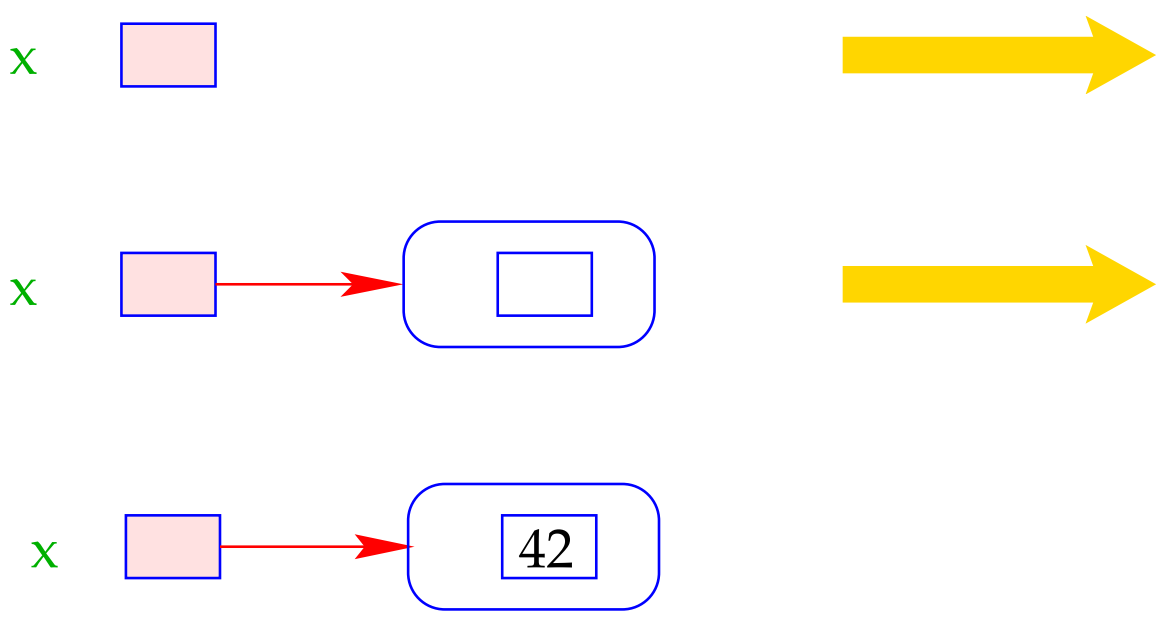


Eingewickelte Werte können auch wieder ausgewickelt werden.

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:

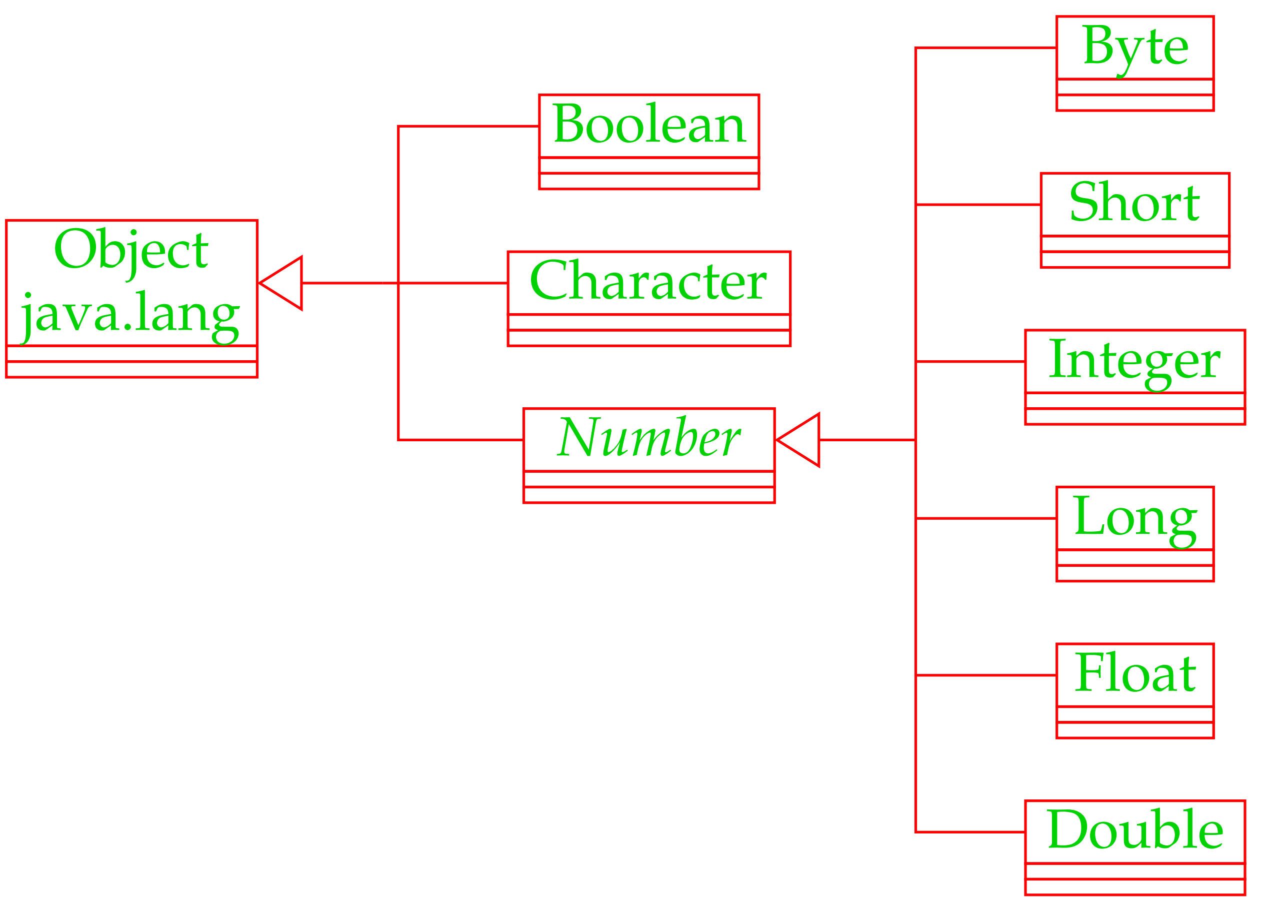
- Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- `public boolean equals(Object obj);` liefert `true` genau dann wenn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

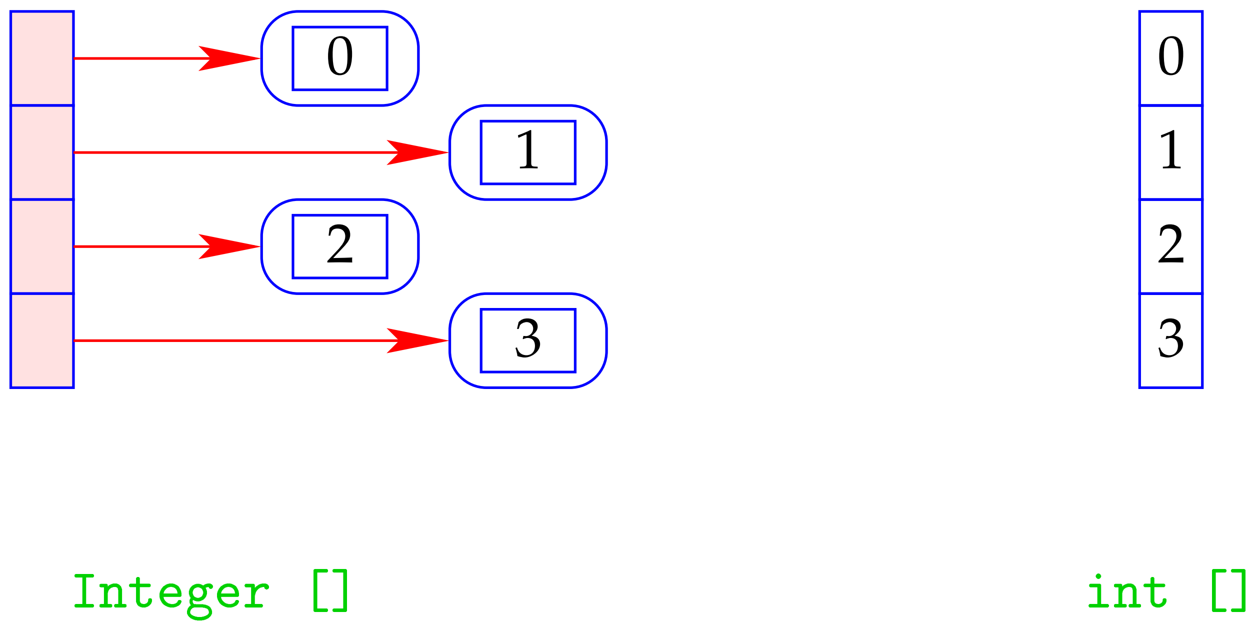
- `Double` und `Float` enthalten zusätzlich die Konstanten

`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`:



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

15 Ein- und Ausgabe

- Ein- und Ausgabe ist nicht Bestandteil von Java.
- Stattdessen werden (äußerst umfangreiche :-| Bibliotheken von nützlichen Funktionen zur Verfügung gestellt.

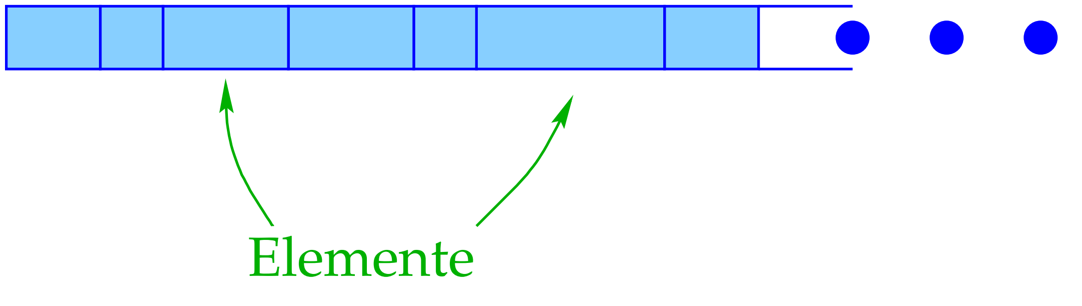
15 Ein- und Ausgabe

- Ein- und Ausgabe ist nicht Bestandteil von Java.
- Stattdessen werden (äußerst umfangreiche :-| Bibliotheken von nützlichen Funktionen zur Verfügung gestellt.

Vorteil:

- Weitere Funktionalität, neue IO-Medien können bereit gestellt werden, ohne gleich die Sprache ändern zu müssen.
- Programme, die nur einen winzigen Ausschnitt der Möglichkeiten nutzen, sollen nicht mit einem komplexen Laufzeit-System belastet werden.

Vorstellung:



- Sowohl Ein- wie Ausgabe vom Terminal oder aus einer Datei wird als Strom aufgefasst.
- Ein Strom (Stream) ist eine (potentiell unendliche) Folge von Elementen.
- Ein Strom wird gelesen, indem links Elemente entfernt werden.
- Ein Strom wird geschrieben, indem rechts Elemente angefügt werden.

Unterstützte Element-Typen:

- Bytes;
- Unicode-Zeichen.

Achtung:

- Alle Bytes enthalten 8 Bit :-)
- Intern stellt Java 16 Bit pro Unicode-Zeichen bereitgestellt ...
- standardmäßig benutzt Java (zum Lesen und Schreiben) den Zeichensatz Latin-1 bzw. ISO8859_1.
- Diese externen Zeichensätze benötigen (welch ein Zufall :-) ein Byte pro Zeichen.

Orientierung:

- Will man mehr oder andere Zeichen (z.B. chinesische), kann man den gesamten Unicode-Zeichensatz benutzen.
- Wieviele Bytes dann extern für einzelne Unicode-Zeichen benötigt werden, hängt von der benutzten Codierung ab ...
- Java unterstützt (in den Klassen `InputStreamReader`, `OutputStreamReader`) die UTF-8-Codierung.
- In dieser Codierung benötigen Unicode-Zeichen 1 bis 3 Bytes.

Problem 1: Wie repräsentiert man Daten, z.B. Zahlen?

- binär codiert, d.h. wie in der Intern-Darstellung
  $\Longrightarrow$ vier Byte pro `int`;
- textuell, d.h. wie in Java-Programmen als Ziffernfolge im Zehner-System (mithilfe von Latin-1-Zeichen für die Ziffern)
  $\Longrightarrow$ bis zu elf Bytes pro `int`.

	Vorteil	Nachteil
binär	platzsparend	nicht menschenlesbar
textuell	menschenlesbar	platz-aufwendig

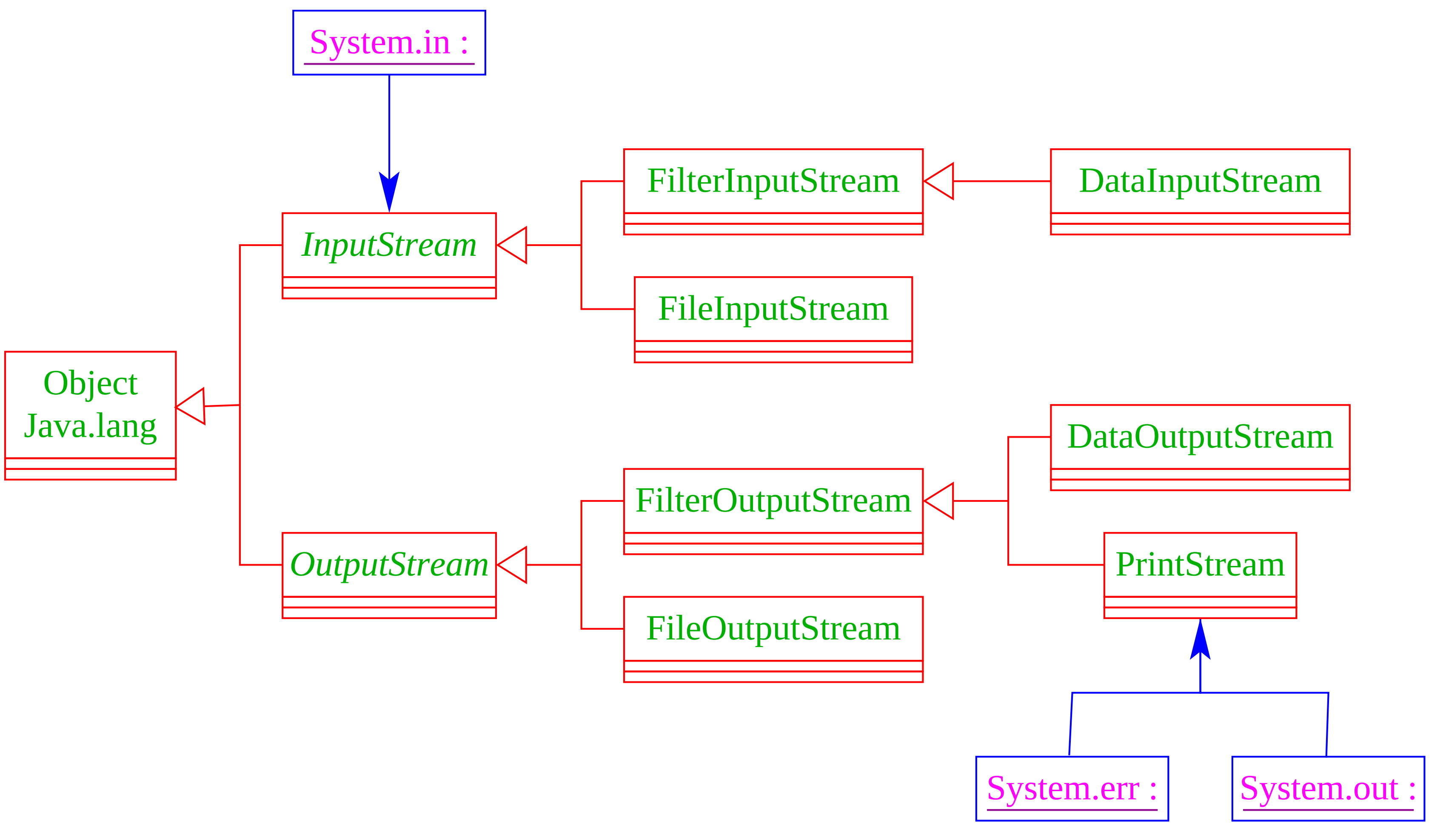
Problem 2:

Wie schreibt bzw. wie liest man die beiden unterschiedlichen Darstellungen?

Dazu stellt Java im Paket `java.io` eine Vielzahl von Klassen zur Verfügung ...

15.1 Byteweise Ein- und Ausgabe

Zuerst eine (unvollständige :-) Übersicht ...


System.in :
InputStream
FilterInputStream
DataInputStream
FileInputStream
Object
Java.lang
DataOutputStream
FilterOutputStream
OutputStream
PrintStream
FileOutputStream
System.err :
System.out :

- Die grundlegende Klasse für `byte`-Eingabe heißt `InputStream`.
- Diese Klasse ist abstrakt.
- Trotzdem ist `System.in` ein Objekt dieser Klasse :-)

Nützliche Operationen:

- `public int available()` :
  gibt die Anzahl der vorhandenen Bytes an;
- `public int read() throws IOException` :
  liest ein Byte vom Input als `int`-Wert; ist das Ende des Stroms erreicht, wird -1 geliefert;
- `void close() throws IOException` : schließt den Eingabe-Strom.

Achtung:

- `System.in.available()` liefert stets `0`.
- Außer `System.in` gibt es keine Objekte der Klasse `InputStream`
- ... außer natürlich Objekte von Unterklassen.

Konstruktoren von Unterklassen:

- `public FileInputStream(String path) throws IOException`
  öffnet die Datei `path`;
- `public DataInputStream(InputStream in)`
  liefert für einen InputStream `in` einen `DataInputStream`.

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   int t;
   while(-1 != (t = file.read()))
      System.out.print((char)t);
   System.out.print("\n");
   } // end of main
} // end of FileCopy
```

- Das Programm interpretiert das erste Argument in der Kommando-Zeile als Zugriffspfad auf eine Datei.
- Sukzessive werden Bytes gelesen und als `char`-Werte interpretiert wieder ausgegeben.
- Das Programm terminiert, sobald das Ende der Datei erreicht ist.

Achtung:

- `char`-Werte sind intern 16 Bit lang ...
- Ein Latin-1-Text wird aus dem Input-File auf die Ausgabe geschrieben, weil ein Byte/Latin-1-Zeichen `xxxx xxxx`
  - intern als `0000 0000 xxxx xxxx` abgespeichert und dann
  - extern als `xxxx xxxx` ausgegeben wird :-)

Erweiterung der Funktionalität:

- In der Klasse `DataInputStream` gibt es spezielle Lese-Methoden für jeden Basis-Typ.

  Unter anderem gibt es:

  - `public byte readByte() throws IOException;`
  - `public char readChar() throws IOException;`
  - `public int readInt() throws IOException;`
  - `public double readDouble() throws IOException.`

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   DataInputStream data = new DataInputStream(file);
   int n = file.available(); char x;
   for(int i=0; i<n/2; ++i) {
      x = data.readChar();
      System.out.print(x);
      }
   System.out.print("\n");
   } // end of main
} // end of erroneous FileCopy
```

... führt i.a. zur Ausgabe: ?????????

Der Grund:

- `readChar()` liest nicht ein Latin-1-Zeichen (i.e. 1 Byte), sondern die 16-Bit-Repräsentation eines Unicode-Zeichens ein.
- Das Unicode-Zeichen, das zwei Latin-1-Zeichen hintereinander entspricht, ist (i.a.) auf unseren Bildschirmen nicht darstellbar. Deshalb die Fragezeichen ...

- Analoge Klassen stehen für die Ausgabe zur Verfügung.
- Die grundlegende Klasse für `byte`-Ausgabe heißt `OutputStream`.
- Auch `OutputStream` ist abstrakt :-)

Nützliche Operationen:

- `public void write(int b) throws IOException` : schreibt das unterste Byte von `b` in die Ausgabe;
- `void flush() throws IOException` : falls die Ausgabe gepuffert wurde, soll sie nun ausgegeben werden;
- `void close() throws IOException` : schließt den Ausgabe-Strom.

- Weil `OutputStream` abstrakt ist, gibt es keine Objekte der Klasse `OutputStream`, nur Objekte von Unterklassen.

Konstruktoren von Unterklassen:

- `public FileOutputStream(String path) throws IOException;`
- `public FileOutputStream(String path, boolean append) throws IOException;`
- `public DataOutputStream(OutputStream out);`
- `public PrintStream(OutputStream out)` — der Rückwärts-Kompatibilität wegen, d.h. um Ausgabe auf `System.out` und `System.err` zu machen ...

Beispiel:

```
import java.io.*;
public class File2FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream fileIn = new FileInputStream(args[0]);
   FileOutputStream fileOut = new FileOutputStream(args[1]);
   int n = fileIn.available();
   for(int i=0; i<n; ++i)
      fileOut.write(fileIn.read());
   fileIn.close(); fileOut.close();
   System.out.print("\t\tDone!!!\n");
   } // end of main
} // end of File2FileCopy
```

- Das Programm interpretiert die 1. und 2. Kommando-Zeilen-Argumente als Zugriffspfade auf eine Ein- bzw. Ausgabe-Datei.
- Die Anzahl der in der Eingabe enthaltenen Bytes wird bestimmt.
- Dann werden sukzessive die Bytes gelesen und in die Ausgabe-Datei geschrieben.

Erweiterung der Funktionalität:

Die Klasse `DataOutputStream` bietet spezielle Schreib-Methoden für verschiedene Basis-Typen an.

Beispielsweise gibt es:

- `void writeByte(int x) throws IOException;`
- `void writeChar(int x) throws IOException;`
- `void writeInt(int x) throws IOException;`
- `void writeDouble(double x) throws IOException.`

Beachte:

- `writeChar()` schreibt genau die Repräsentation eines Zeichens, die von `readChar()` verstanden wird, d.h. 2 Byte.

Beispiel:

```
import java.io.*;
public class Numbers {
public static void main(String[] args) throws IOException {
   FileOutputStream file = new FileOutputStream(args[0]);
   DataOutputStream data = new DataOutputStream(file);
   int n = Integer.parseInt(args[1]);
   for(int i=0; i<n; ++i)
      data.writeInt(i);
   data.close();
   System.out.print("\t\tDone!\n");
   } // end of main
} // end of Numbers
```

- Das Programm entnimmt der Kommando-Zeile den Datei-Pfad sowie eine Zahl `n`.
- Es wird ein `DataOutputStream` für die Datei eröffnet.
- In die Datei werden die Zahlen 0,...,`n-1` binär geschrieben.
- Das sind also 4`n` Bytes.

Achtung:

- In der Klasse `System` sind die zwei vordefinierten Ausgabe-Ströme `out` und `err` enthalten.
- `out` repräsentiert die Ausgabe auf dem Terminal.
- `err` repräsentiert die Fehler-Ausgabe für ein Programm (i.a. ebenfalls auf dem Terminal).
- Diese sollen jedes Objekt als Text auszugeben können.
- Dazu dient die Klasse `PrintStream`.

- Die `public`-Objekt-Methoden `print()` und `println()` gibt es für jeden möglichen Argument-Typ.
- Für (Programmierer-definierte) Klassen wird dabei auf die `toString()`-Methode zurückgegriffen.
- `println()` unterscheidet sich von `print`, indem nach Ausgabe des Arguments eine neue Zeile begonnen wird.

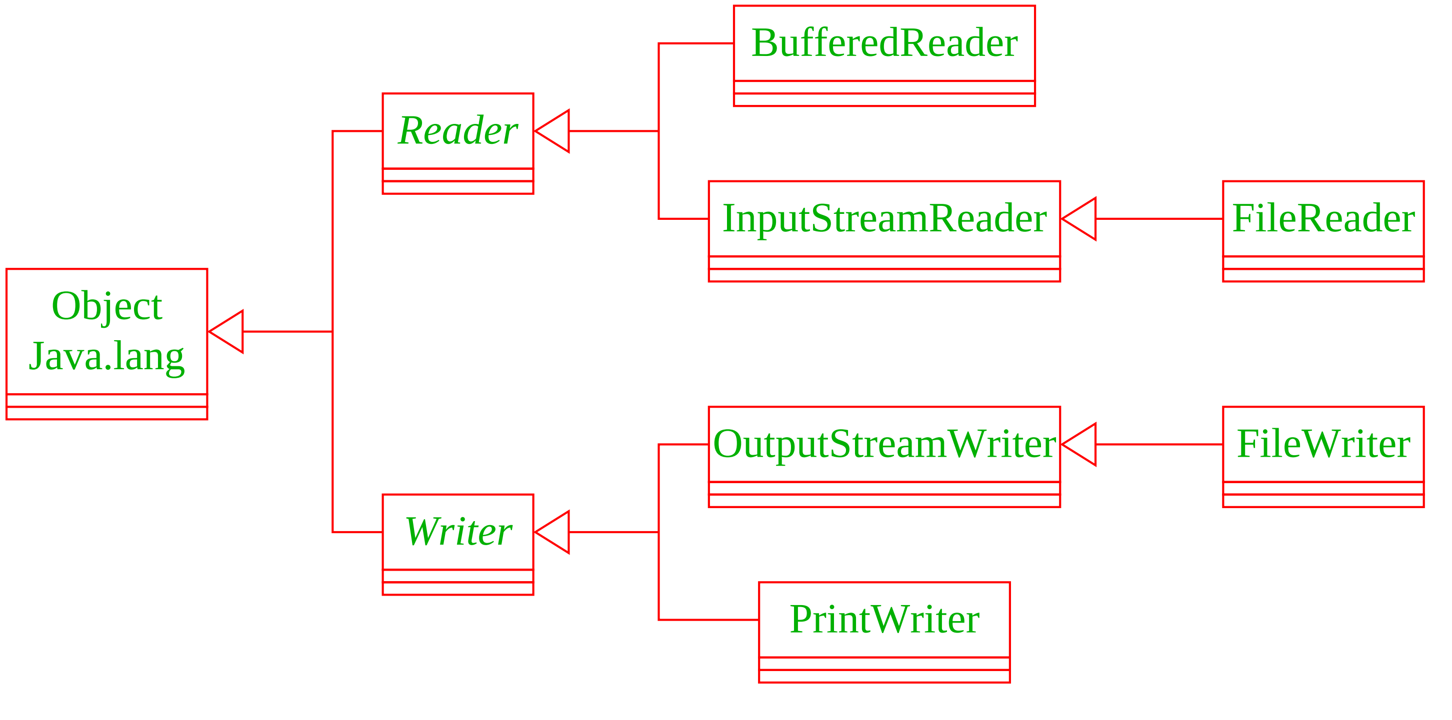
Achtung:

- `PrintStream` benutzt (neuerdings) die Standard-Codierung des Systems, d.h. bei uns Latin-1.
- Um aus einem (Unicode-)String eine Folge von sichtbaren (Latin-1-)Zeichen zu gewinnen, wird bei jedem Zeichen, das kein Latin-1-Zeichen darstellt, ein '?' gedruckt ...

$\Longrightarrow$ für echte (Unicode-) Text-Manipulation schlecht geeignet ...

15.2 Textuelle Ein- und Ausgabe

Wieder erstmal eine Übersicht über hier nützliche Klassen ...



- Die Klasse `Reader` ist abstrakt.

Wichtige Operationen:

- `public boolean ready() throws IOException :`
  liefert `true`, sofern ein Zeichen gelesen werden kann;
- `public int read() throws IOException :`
  liest ein (Unicode-)Zeichen vom Input als `int`-Wert; ist das Ende des Stroms erreicht, wird -1 geliefert;
- `void close() throws IOException :` schließt den Eingabe-Strom.

- Ein `InputStreamReader` ist ein spezieller Textleser für Eingabe-Ströme.
- Ein `FileReader` gestattet, aus einer Datei Text zu lesen.
- Ein `BufferedReader` ist ein `Reader`, der Text zeilenweise lesen kann, d.h. eine zusätzliche Methode `public String readLine() throws IOException;` zur Verfügung stellt.

Konstruktoren:

- `public InputStreamReader(InputStream in);`
- `public InputStreamReader(InputStream in, String encoding) throws IOException;`
- `public FileReader(String path) throws IOException;`
- `public BufferedReader(Reader in);`

Beispiel:

```
import java.io.*;
public class CountLines {
public static void main(String[] args) throws IOException {
   FileReader file = new FileReader(args[0]);
   BufferedReader buff = new BufferedReader(file);
   int n=0; while(null != buff.readLine()) n++;
   buff.close();
   System.out.print("Number of Lines:\t\t"+ n);
   } // end of main
} // end of CountLines
```

- Die Objekt-Methode `readLine()` liefert `null`, wenn beim Lesen das Ende der Datei erreicht wurde.
- Das Programm zählt die Anzahl der Zeilen einer Datei   :-)

- Wieder stehen analoge Klassen für die Ausgabe zur Verfügung.
- Die grundlegende Klasse für textuelle Ausgabe heißt `Writer`.

Nützliche Operationen:

- `public void write(` *type* `x) throws IOException` :
  gibt es für die Typen `int` (dann wird ein einzelnes Zeichen geschrieben), `char[]` sowie `String`.
- `void flush() throws IOException` :
  falls die Ausgabe gepuffert wurde, soll sie nun tatsächlich ausgegeben werden;
- `void close() throws IOException` : schließt den Ausgabe-Strom.

- Weil `Writer` abstrakt ist, gibt keine Objekte der Klasse `Writer`, nur Objekte von Unterklassen.

Konstruktoren:

- `public OutputStreamWriter(OutputStream str);`
- `public OutputStreamWriter(OutputStream str, String encoding);`
- `public FileWriter(String path) throws IOException;`
- `public FileWriter(String path, boolean append) throws IOException;`
- `public PrintWriter(OutputStream out);`
- `public PrintWriter(Writer out);`

Beispiel:

```
import java.io.*;
public class Text2TextCopy {
public static void main(String[] args) throws IOException {
   FileReader fileIn = new FileReader(args[0]);
   FileWriter fileOut = new FileWriter(args[1]);
   int c = fileIn.read();
   for(; c!=-1; c=fileIn.read())
      fileOut.write(c);
   fileIn.close(); fileOut.close();
   System.out.print("\t\tDone!!!\n");
   } // end of main
} // end of Text2TextCopy
```

Wichtig:

- Ohne einen Aufruf von `flush()` oder `close()` kann es passieren, dass das Programm beendet wird, bevor die Ausgabe in die Datei geschrieben wurde :-(
- Zum Vergleich: in der Klasse `OutputStream` wird `flush()` automatisch nach jedem Zeichen aufgerufen, das ein Zeilenende markiert.

Bleibt, die zusätzlichen Objekt-Methoden für einen `PrintWriter` aufzulisten...

- Analog zum `PrintStream` sind dies
  `public void print(` *type* `x);` und
  `public void println(` *type* `x);`

- ... die es gestatten, Werte jeglichen Typs (aber nun evt. in geeigneter Codierung) auszudrucken.

Text-Ein- und Ausgabe arbeitet mit (Folgen von) Zeichen.

Nicht schlecht, wenn man dafür die Klasse `String` etwas näher kennen würde ...

16 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(Object obj);`
- `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);`

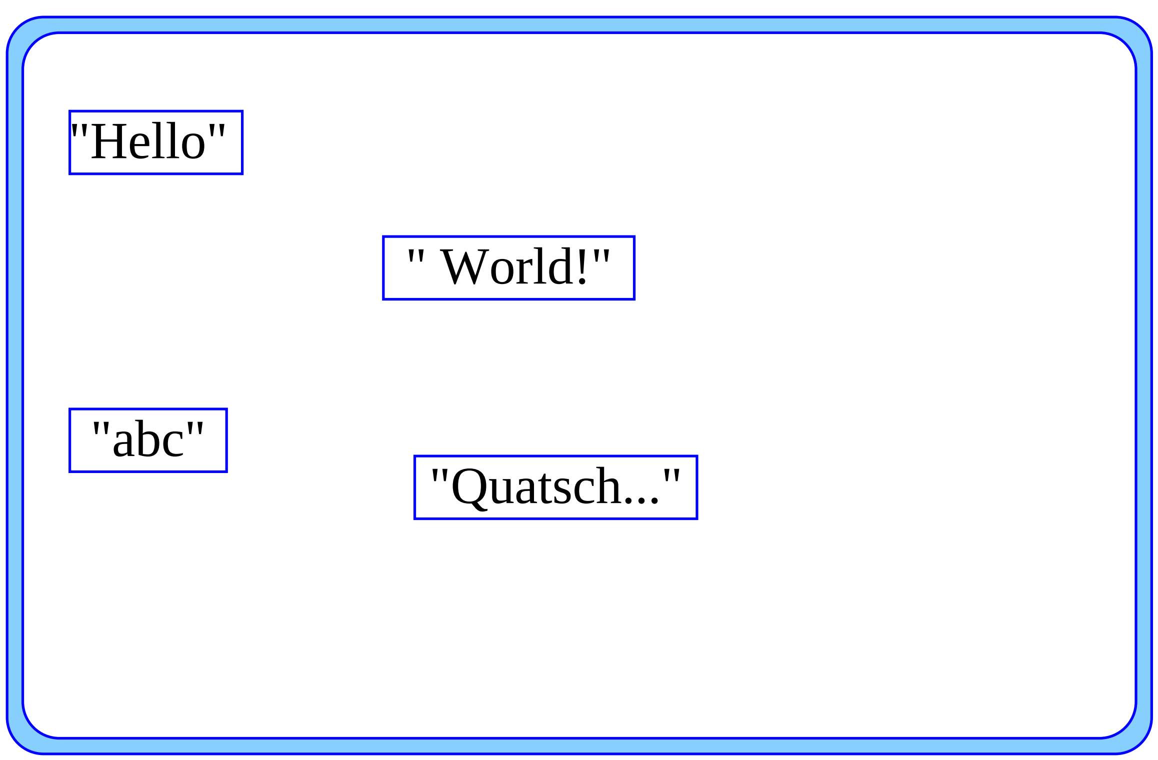
... weitere Objekt-Methoden:

- `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.

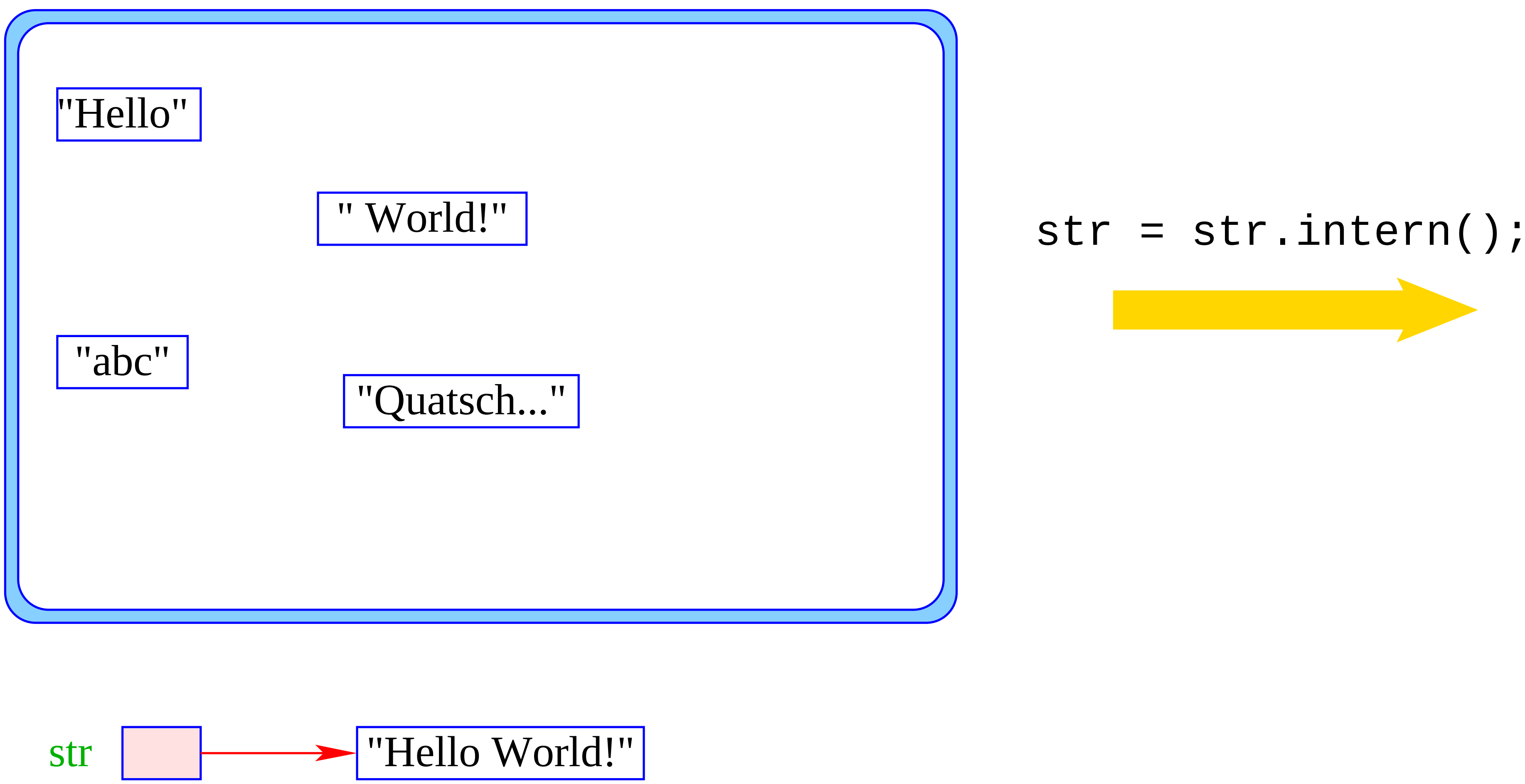
... sowie viele weitere :-)

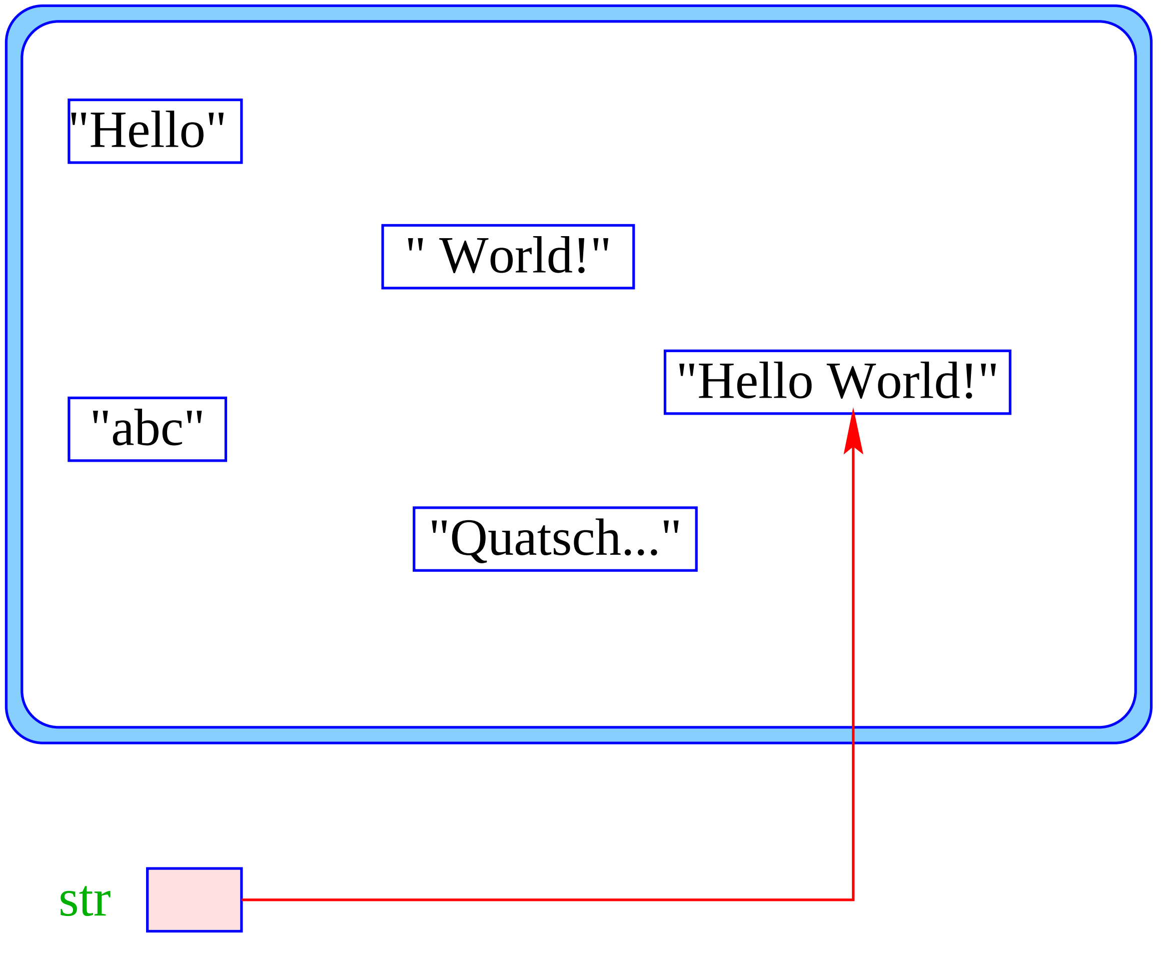
Zur Objekt-Methode `intern()`:

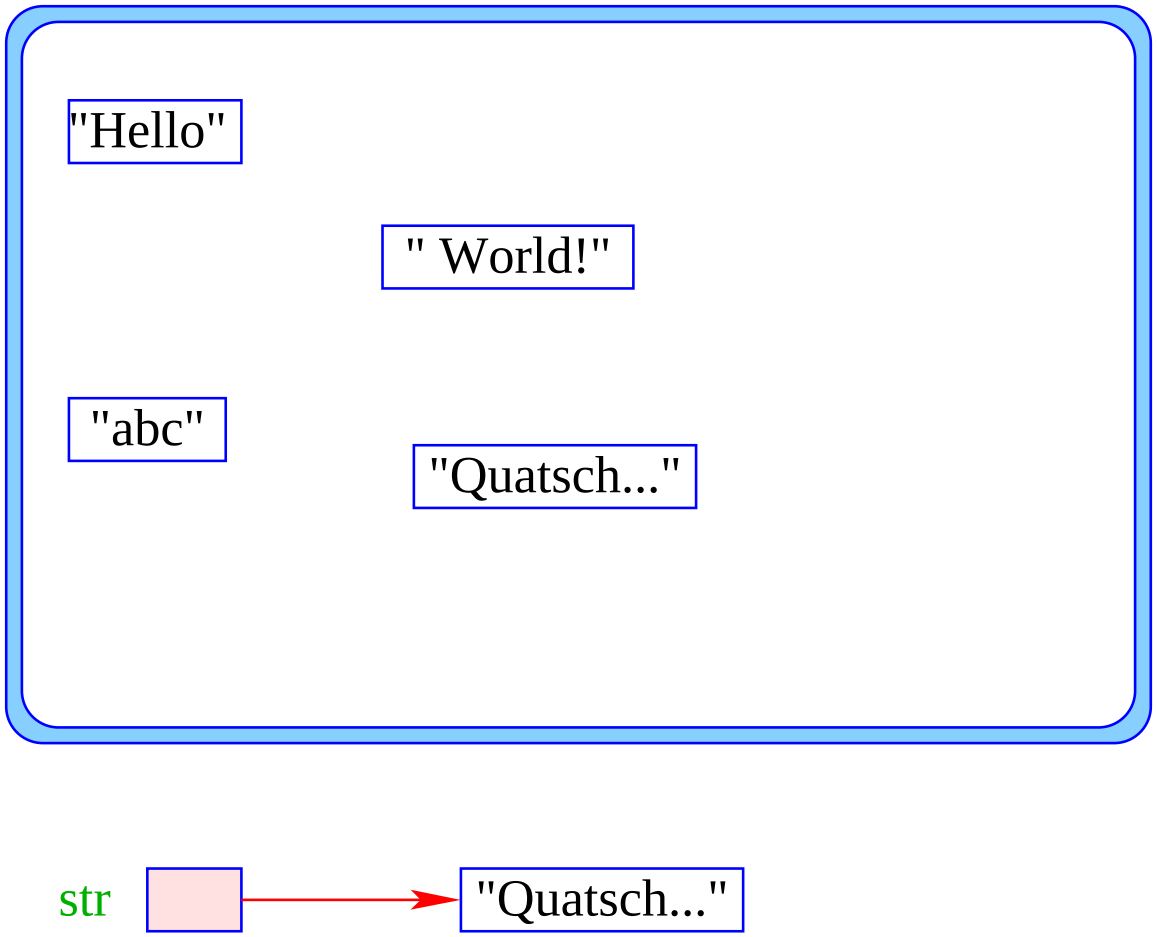
- Die Java-Klasse `String` verwaltet einen privaten String-Pool:



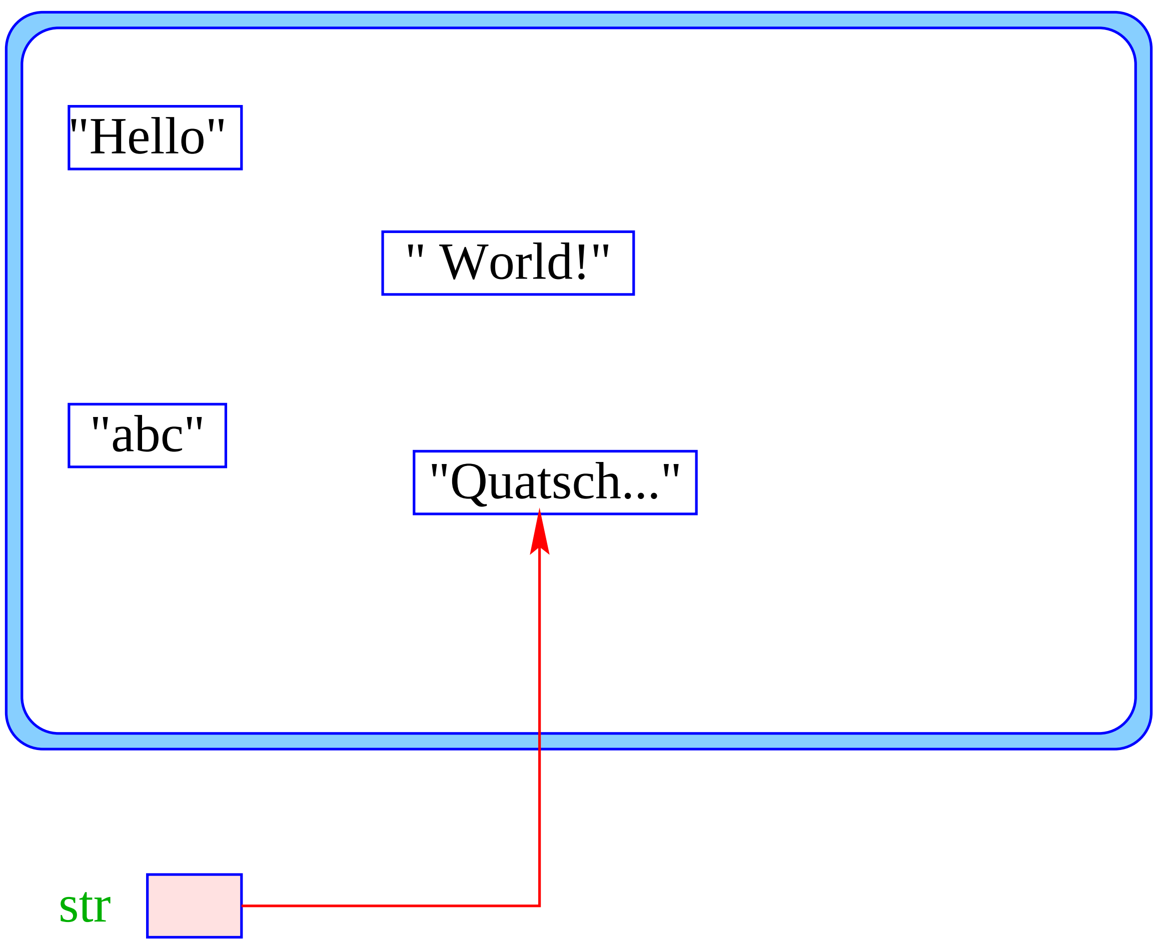
- Alle `String`-Konstanten des Programms werden automatisch in den Pool eingetragen.
- `s.intern();` überprüft, ob die gleiche Zeichenfolge wie `s` bereits im Pool ist.
- Ist dies der Fall, wird ein Verweis auf das Pool-Objekt zurück gegeben.
- Andernfalls wird `s` in den Pool eingetragen und `s` zurück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


"Hello"
" World!"
"abc"
"Quatsch..."
str = str.intern();
str
"Hello World!"


"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. “==”

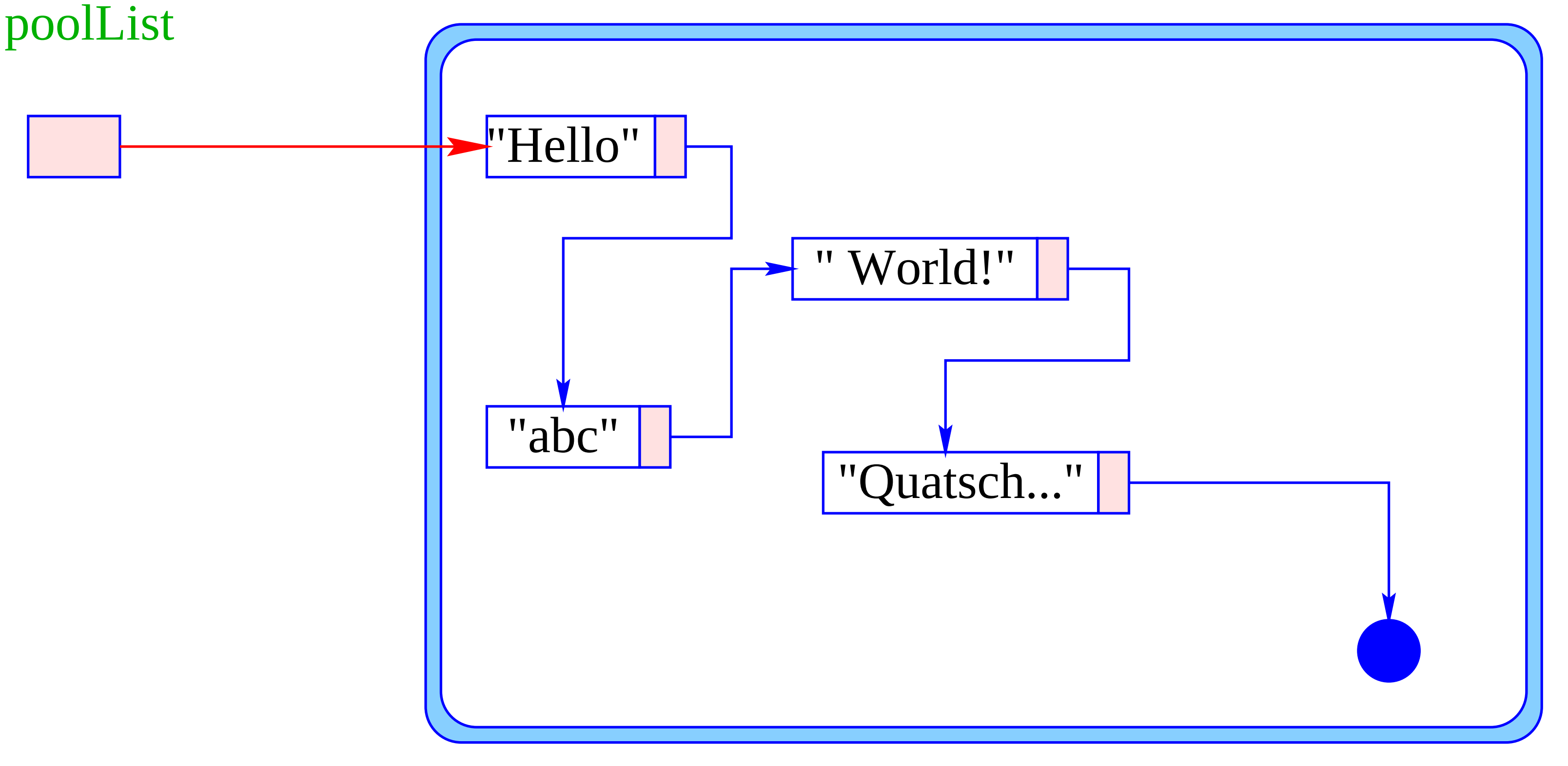
  $\Longrightarrow$ erheblich effizienter als zeichenweiser Vergleich !!!

... 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.

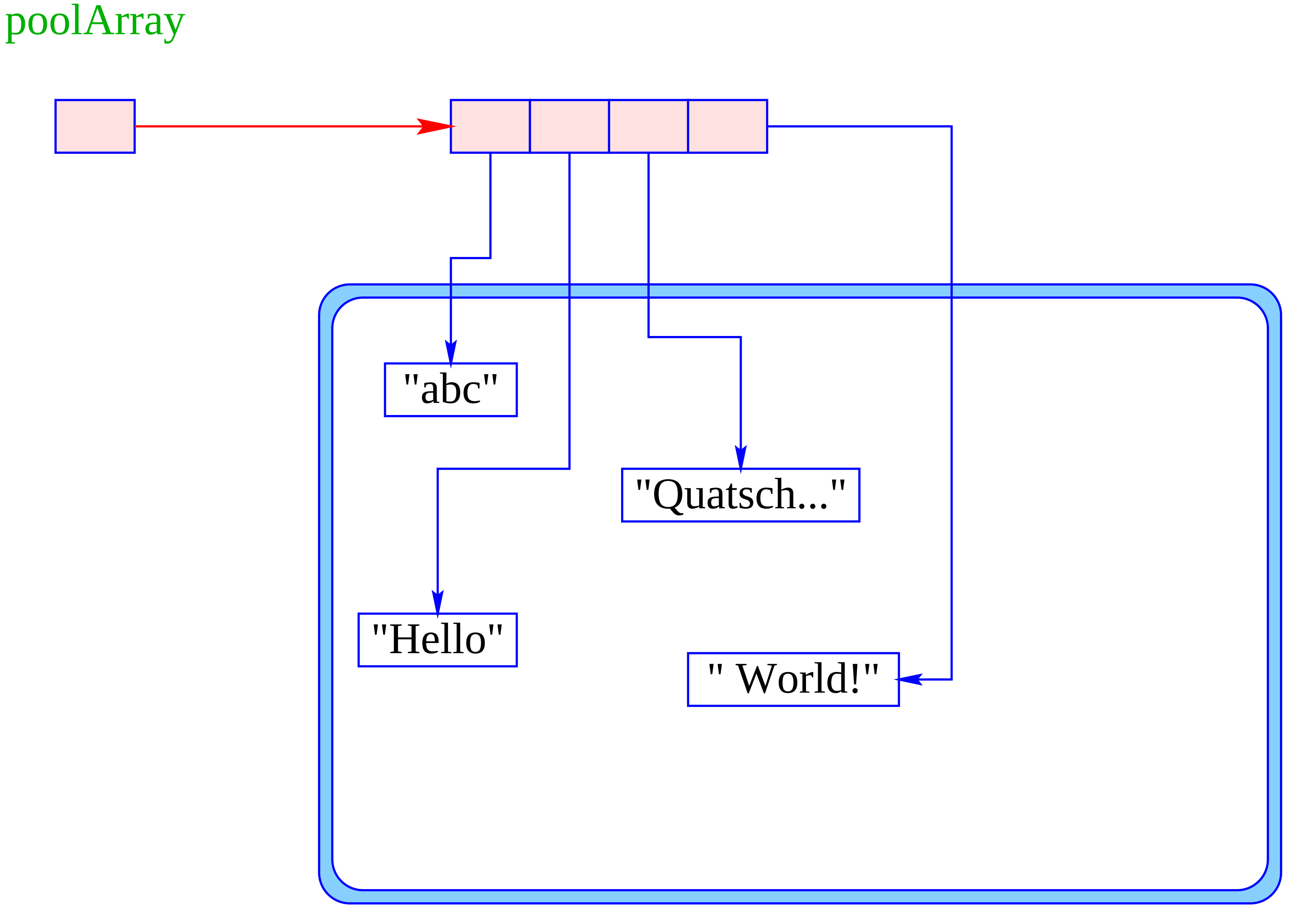

poolList
"Hello"
" World!"
"abc"
"Quatsch..."

+ 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
"abc"
"Quatsch..."
"Hello"
" World!"

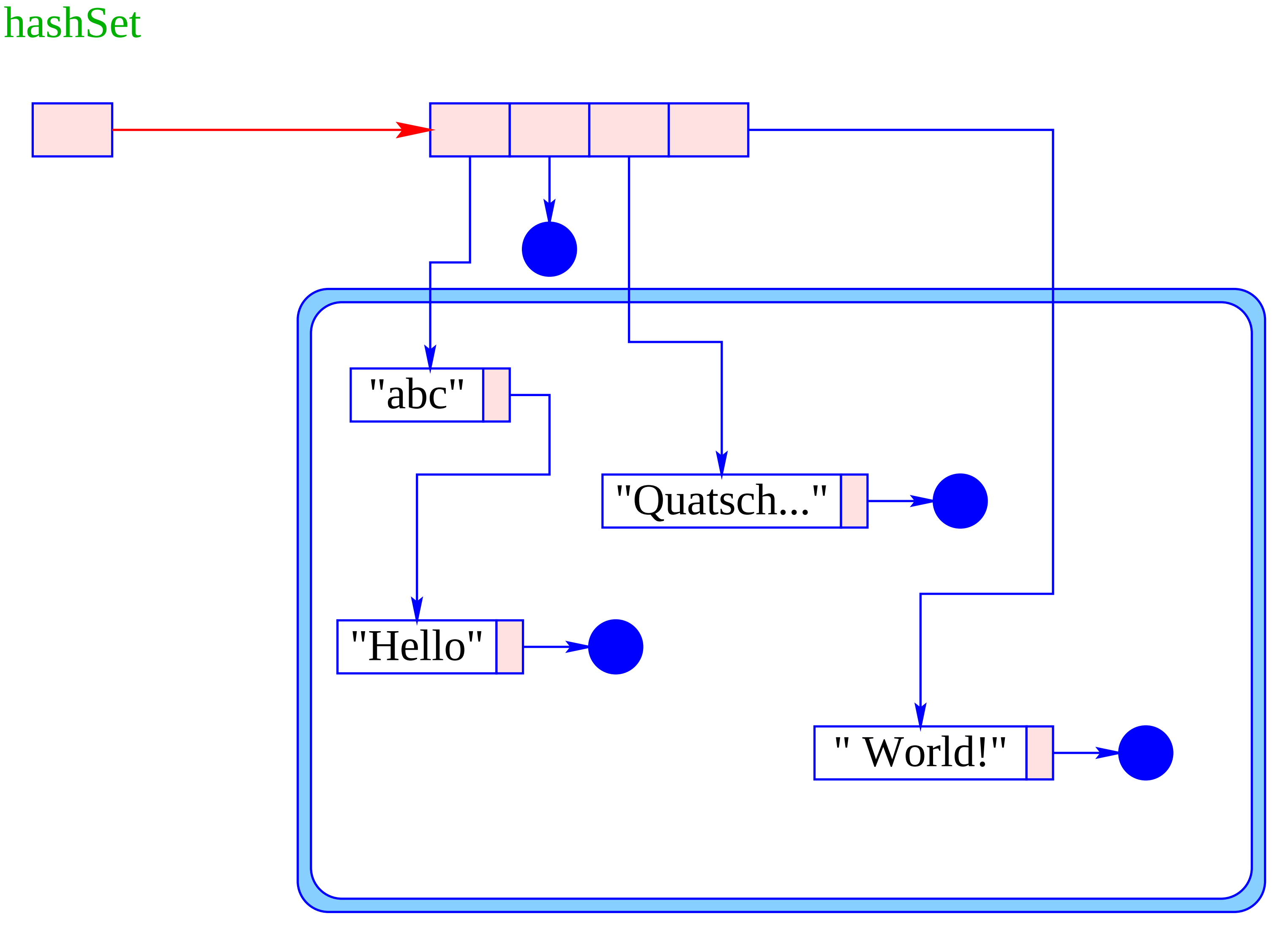
+ Auffinden eines Worts im Pool ist einfach.

- 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
"abc"
"Quatsch..."
"Hello"
" World!"

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 $p = 31$.)

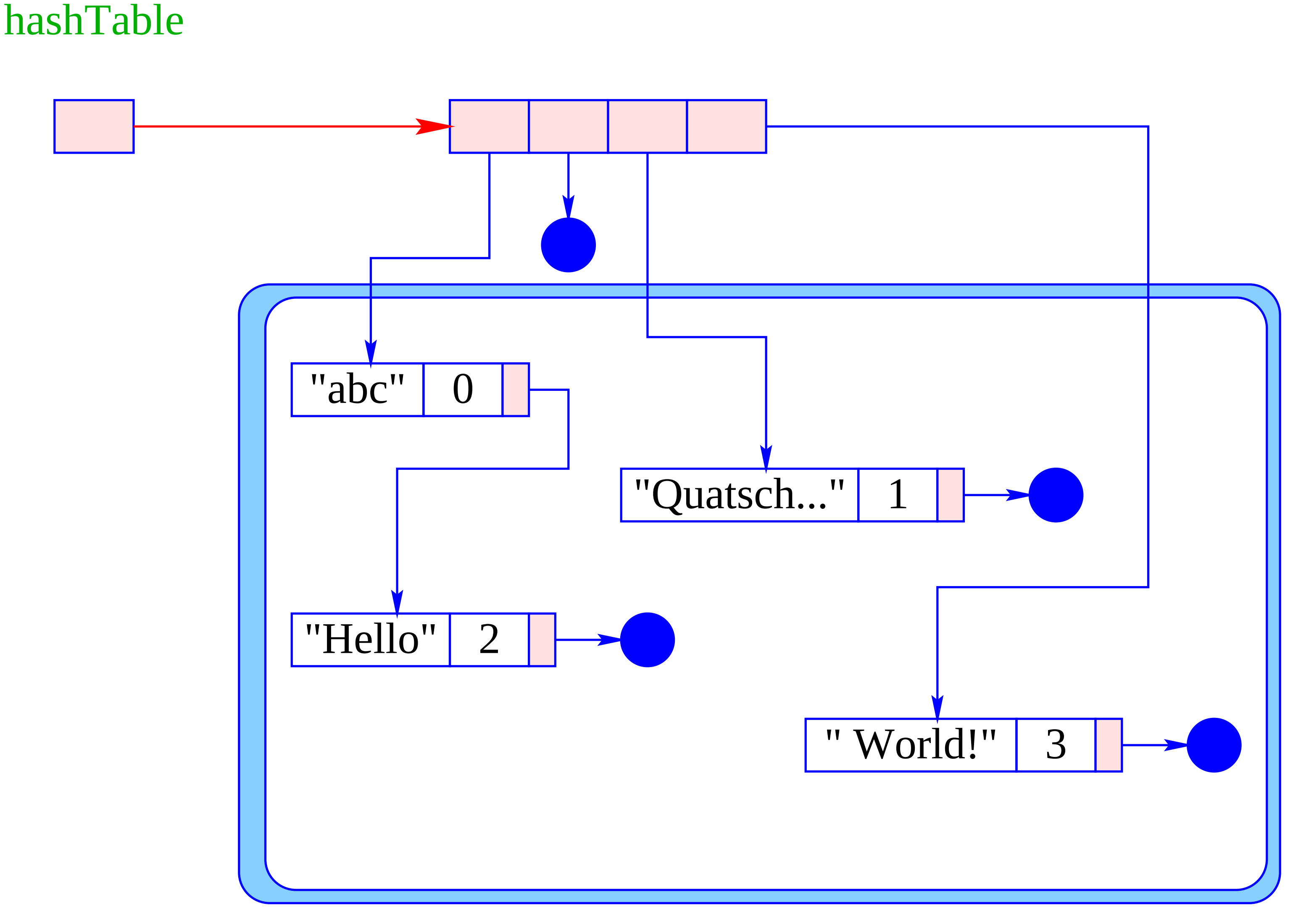
`String`	h_0	h_1	h_2 $(p = 7)$
alloc	196	523	276109
add	197	297	5553
and	197	307	5623
const	215	551	282083
div	218	323	5753
eq	214	214	820
fjump	214	546	287868
false	203	523	284371
halt	220	425	41297
jump	218	444	42966
less	223	439	42913
leq	221	322	6112
...		...	

`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 StringList[] hashSet = new StringList[n];
   public String intern() {
      int i = (Math.abs(hashCode())%n;
      for (StringList t=hashSet[i]; t!=null; t=t.next)
         if (equals(t.info)) return t.info;
      hashSet[i] = new StringList(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
"abc"
0
"Quatsch..."
1
"Hello"
2
" World!"
3

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.

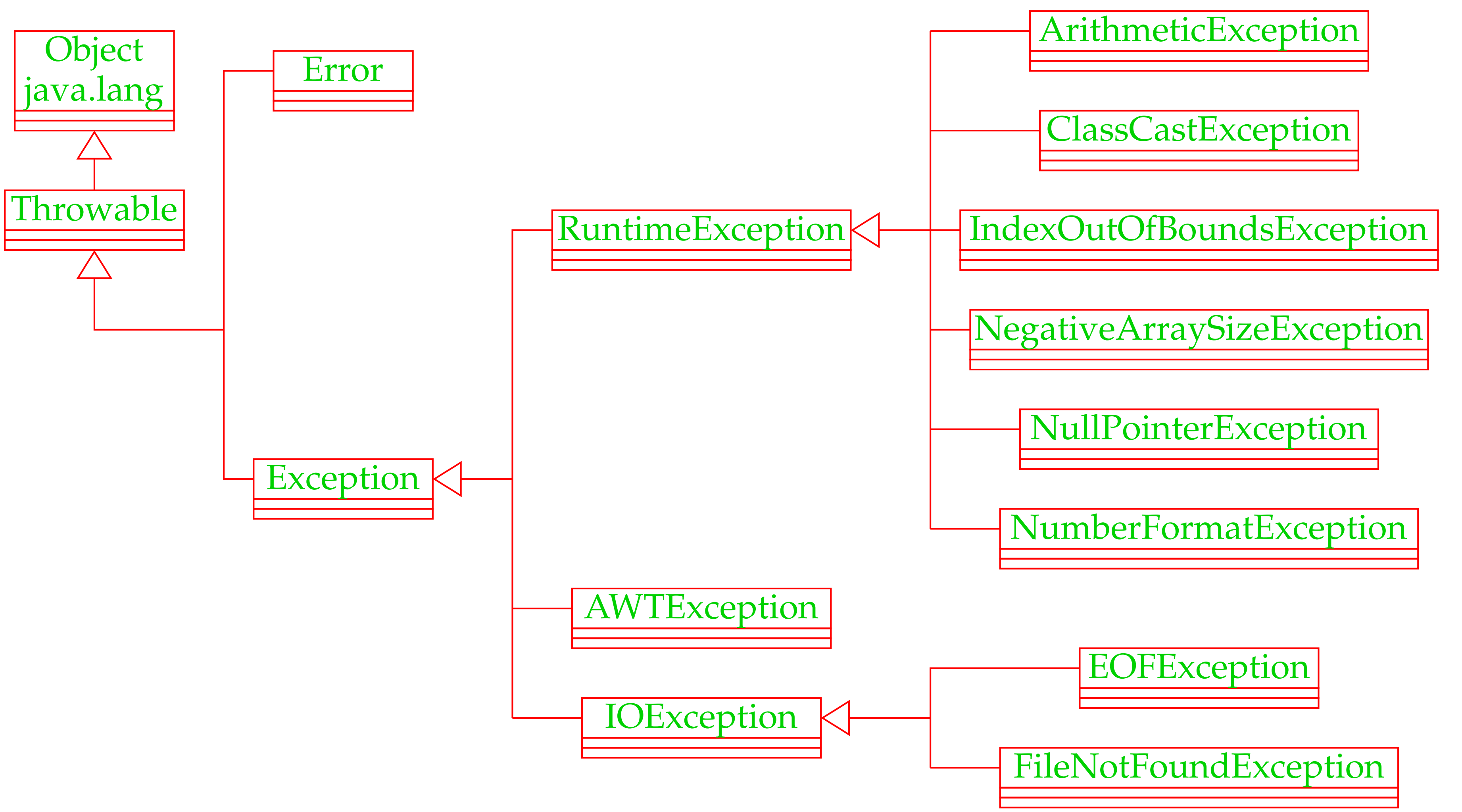
17 Fehler-Objekte: Werfen, Fangen, Behandeln

- 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`

```
import java.io.*;
public class Adding {
   private static BufferedReader stdin = new BufferedReader
          (new InputStreamReader(System.in));
   public static void main(String[] args) {
      int x = getInt("1. Zahl:\t");
      int y = getInt("2. Zahl:\t");
      System.out.println("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.
- Die `boolean`-Variable der Funktion `getInt()` soll den Wert `true` enthalten, wenn die Eingabe einer Zahl erfolgreich abgeschlossen ist ...

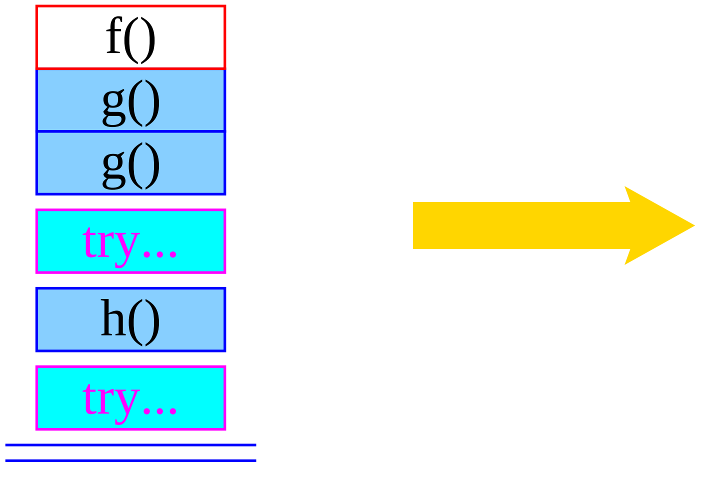
```
    while (true) {
      System.out.print(str);
      System.out.flush();
      try {
        return Integer.parseInt(stdin.readLine());
      } catch (NumberFormatException e) {
        System.out.println("Falsche Eingabe! ...");
      } catch (IOException e) {
        System.out.println("Eingabeproblem: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

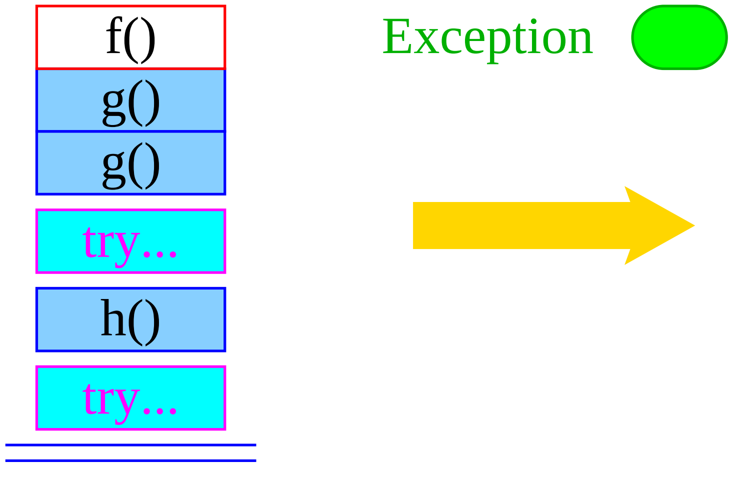
... ermöglicht folgenden Dialog:

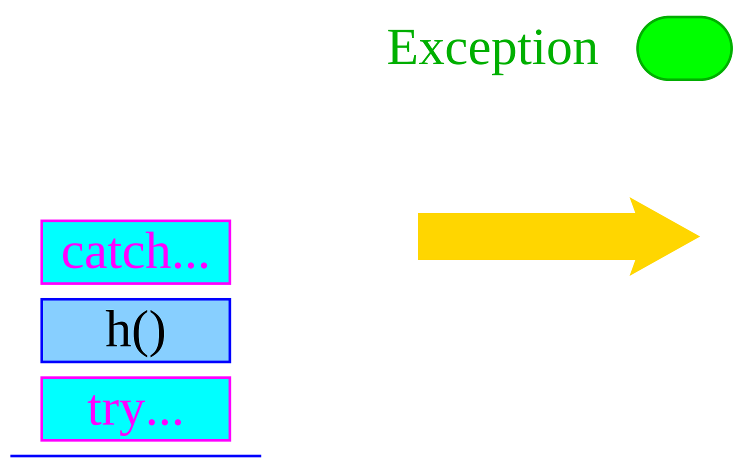
```
> java Adding
1. Zahl:        abc
Falsche Eingabe! ...
1. Zahl:        0.3
Falsche Eingabe! ...
1. Zahl:        17
2. Zahl:        25
Summe:          42
```

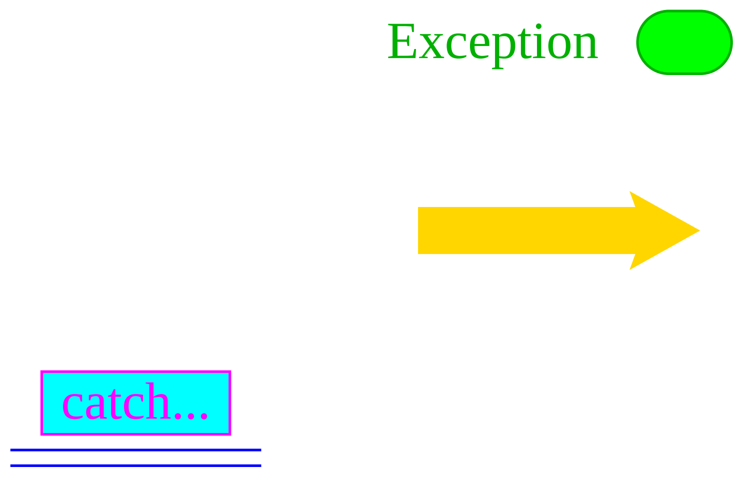
- Ein Exception-Handler besteht aus einem `try`-Block `try { ss }`, in dem der Fehler möglicherweise auftritt; gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge ss kein Fehler-Objekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.
- Wird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

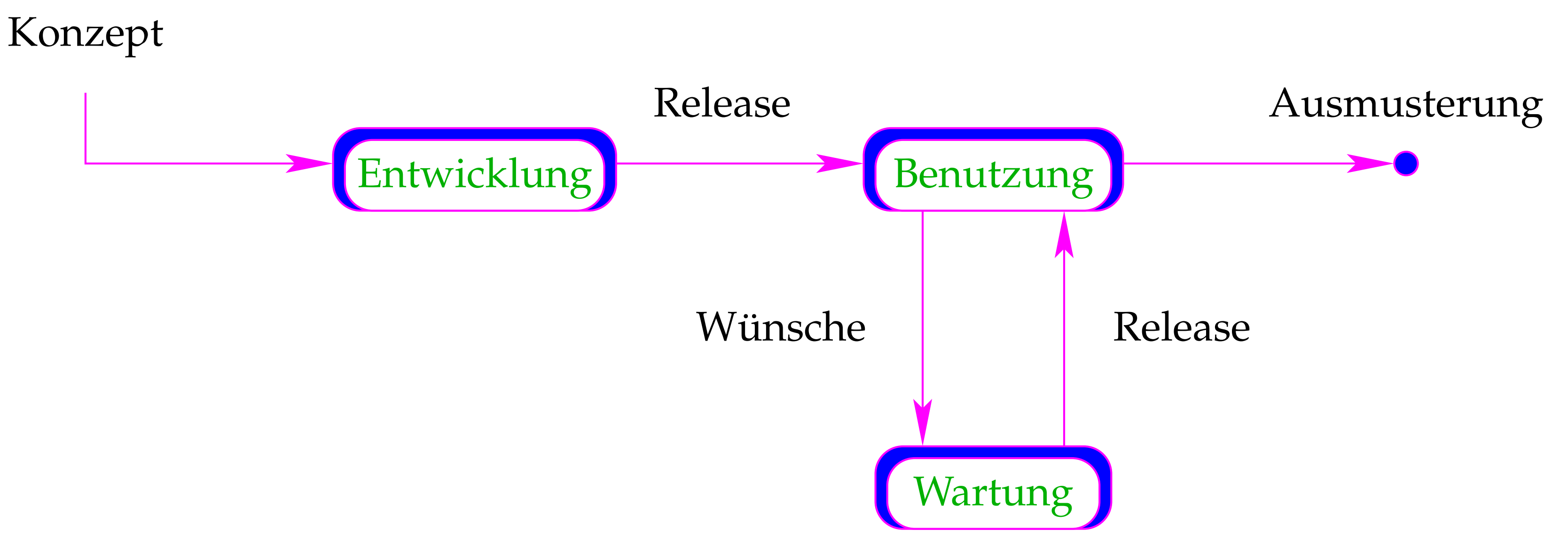
- Der Fehler-Mechanismus von Java sollte auch nur zur Fehler-Behandlung eingesetzt werden:
  - Installieren eines Handlers ist billig; fangen einer `Exception` dagegen teuer.
  - Ein normaler Programm-Ablauf kann durch eingesetzte `Exceptions` bis zur Undurchsichtigkeit verschleiert werden.
  - Was passiert, 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) { }`

18 Programmieren im Großen

Neu:

- Das Programm ist groß.
- Das Programm ist unübersichtlich.
- Das Programm ist teuer.
- Das Programm wird lange benutzt.

Software-Zyklus:

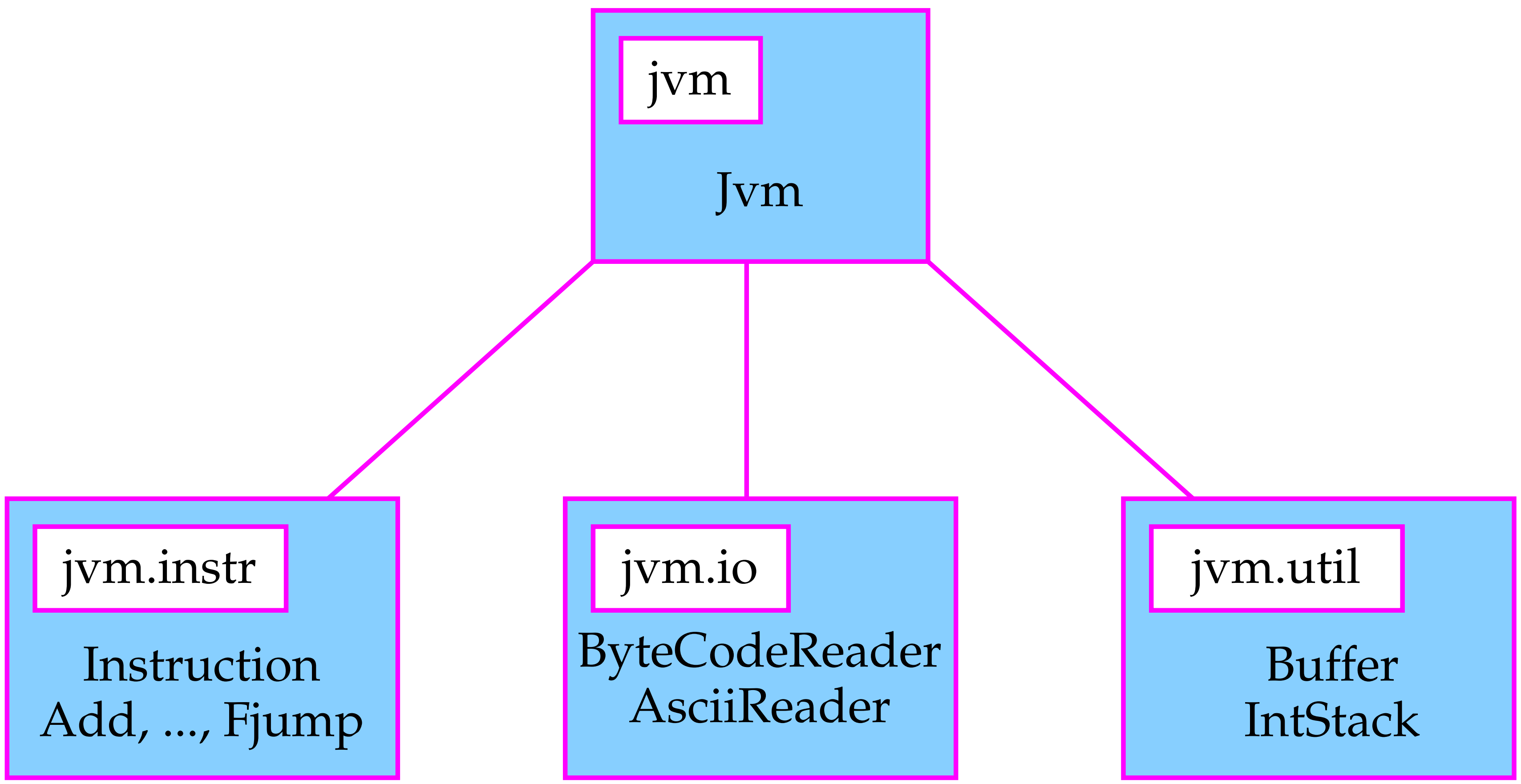


- Wünsche können sein:
  - Beseitigen von Fehlern;
  - Erweiterung der Funktionalität;
  - Portierung auf eine andere Plattform;
  - ...
- Die Leute, die die Wartung vornehmen, sind i.a. verschieden von denen, die das System implementierten.
- Gute Wartbarkeit ergibt sich aus
  - einem klaren Design;
  - einer übersichtlichen Strukturierung $\Longrightarrow$ packages;
  - einer sinnvollen, verständlichen Dokumentation.

Programm-Pakete in Java

- Ein großes System sollte hierarchisch in Teilsysteme zerlegt werden.
- Jedes Teilsystem bildet ein Paket oder `package` ...
- und liegt in einem eigenen Verzeichnis.

Beispiel: Unsere JVM


jvm
Jvm
jvm.instr
Instruction
Add, ..., Fjump
jvm.io
ByteCodeReader
AsciiReader
jvm.util
Buffer
IntStack

- Für jede Klasse muss man angeben:
  1. zu welchem Paket sie gehört;
  2. welche Pakete bzw. welche Klassen aus welchen Paketen sie verwendet.

Im Verzeichnis `a` liege die Datei `A.java` mit dem Inhalt:

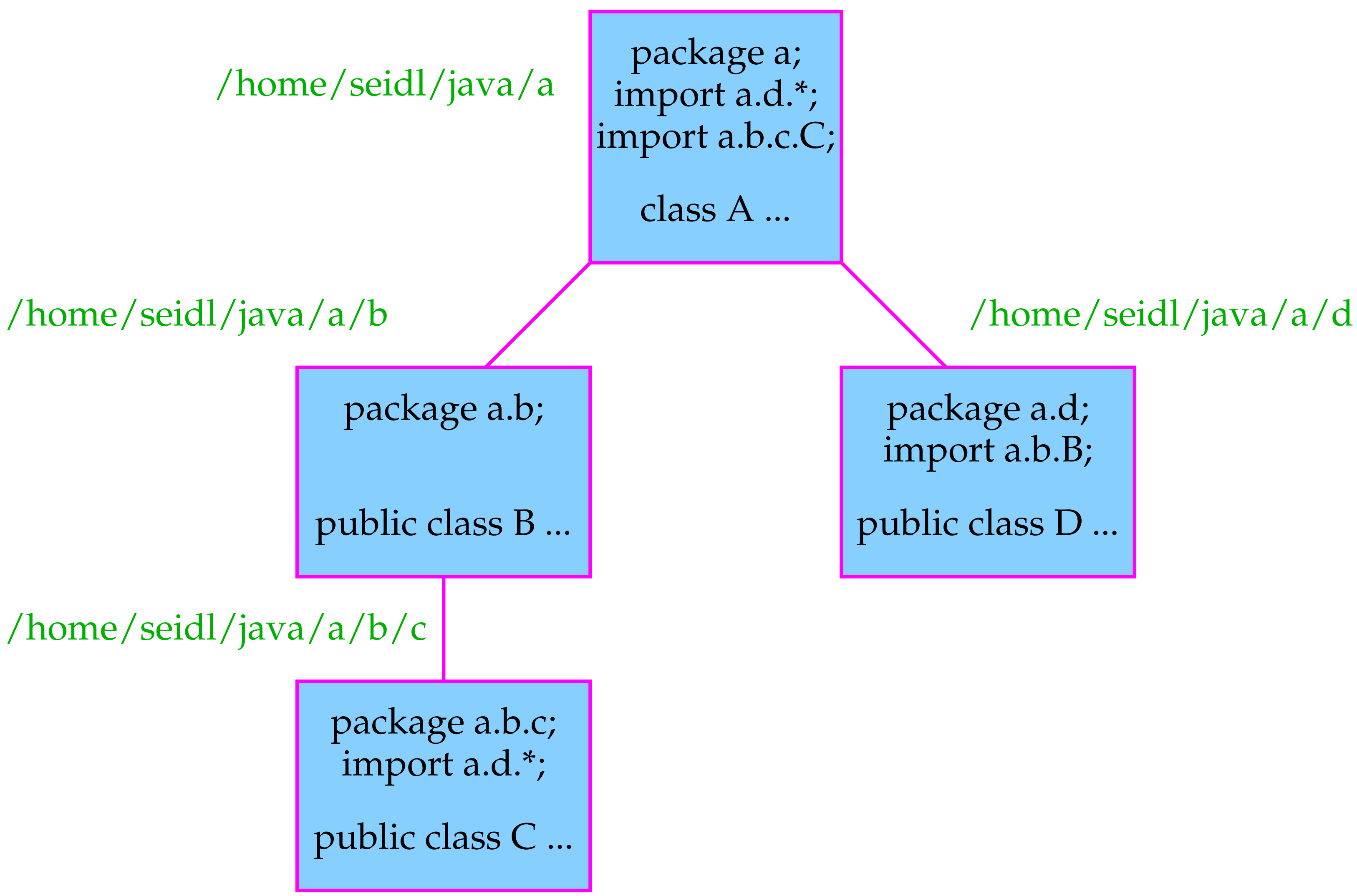
```
package a;
import a.d.*;
import a.b.c.C;
class A {
   public static void main(String[] args) {
      C c = new C();
      D d = new D();
      System.out.println(c+ "   "+d);
   }
}
```

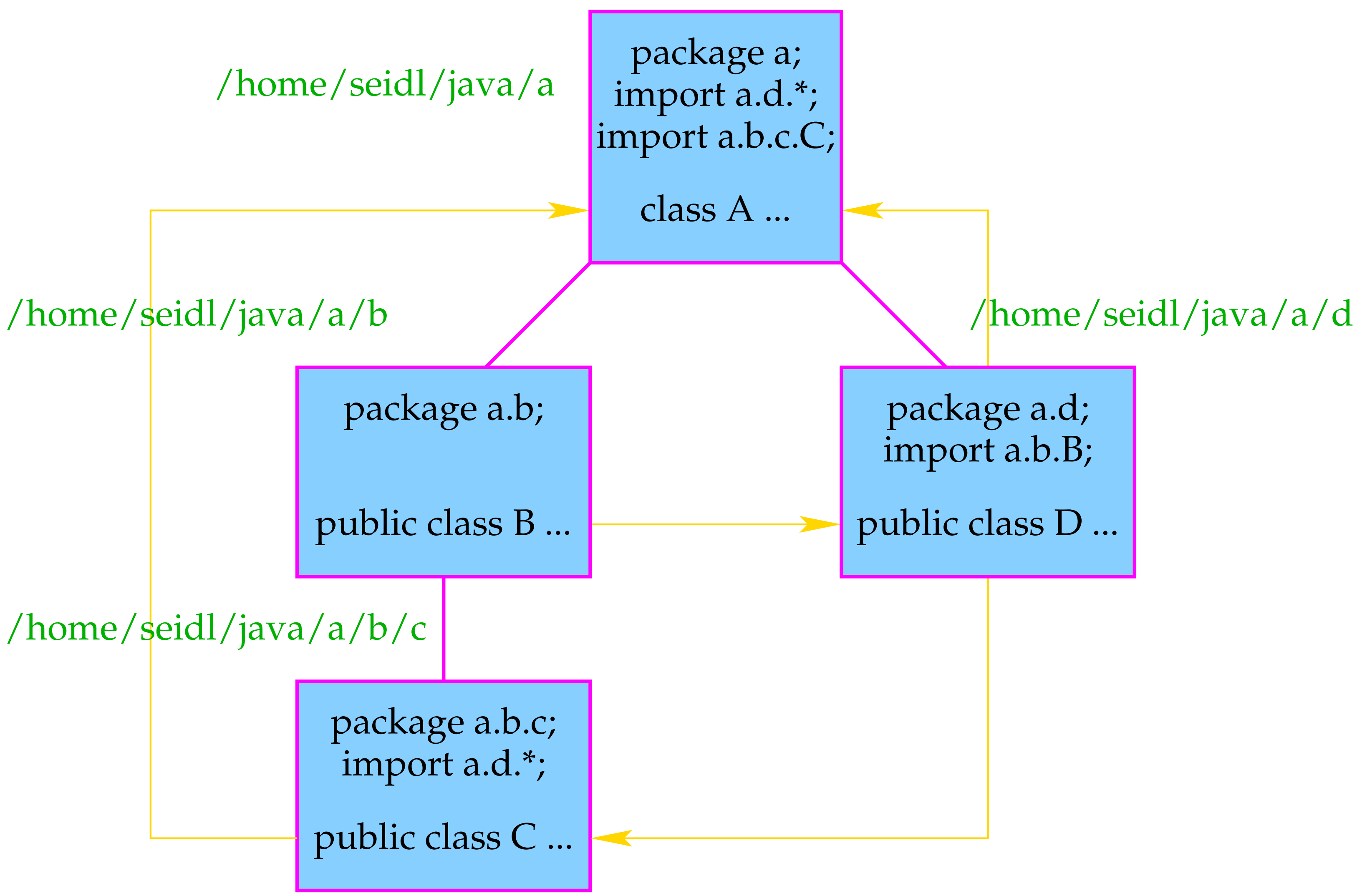
- Jede Datei mit Klassen des Pakets pckg muss am Anfang gekennzeichnet sein mit der Zeile `package` pckg;
- Die Direktive `import` pckg`.*;` stellt sämtliche öffentlichen Klassen des Pakets pckg den Klassen in der aktuellen Datei zur Verfügung – nicht dagegen die Unterverzeichnisse :-| .
- Die Direktive `import` pckg.Cls`;` stellt dagegen nur die Klasse Cls des Pakets pckg (d.h. genauer die Klasse pckg.Cls) zur Verfügung.

In den Unterverzeichnissen `b`, `b/c` und `d` von `a` liegen Dateien mit den Inhalten:

```java
package a.b;
public class B { }
class Ghost { }

package a.b.c;
import a.d.*;
public class C { }
```

```java
package a.d;
import a.b.B;
public class D {
   private B b = null;
}
```


/home/seidl/java/a
package a;
import a.d.*;
import a.b.c.C;
class A ...
/home/seidl/java/a/b
package a.b;
public class B ...
/home/seidl/java/a/d
package a.d;
import a.b.B;
public class D ...
/home/seidl/java/a/b/c
package a.b.c;
import a.d.*;
public class C ...


/home/seidl/java/a
package a;
import a.d.*;
import a.b.c.C;
class A ...
/home/seidl/java/a/b
/home/seidl/java/a/d
package a.b;
public class B ...
package a.d;
import a.b.B;
public class D ...
/home/seidl/java/a/b/c
package a.b.c;
import a.d.*;
public class C ...

Achtung:

- Jede Klasse eines Pakets, die in einer Klasse außerhalb des Pakets benutzt werden soll, muss als `public` gekennzeichnet werden.
- Jede Datei darf zwar mehrere Klassen-Definitionen enthalten, aber nur eine einzige, die `public` ist.
- Der Name der öffentlichen Klasse muss mit demjenigen der Datei übereinstimmen ... :-)
- Der Paket-Name enthält den gesamten absoluten Zugriffs-Pfad von dem Wurzel-Paket.
- Abhängigkeiten zwischen Paketen können zirkulär sein.

Im Verzeichnis a lässt sich das Programm compilieren. Allerdings liefert ...

```
> java A
Exception in thread "main" java.lang.NoClassDefFoundError: a/A (wrong name: A)
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(Compiled Code)
        at java.security.SecureClassLoader.defineClass(Compiled Code)
        at java.net.URLClassLoader.defineClass(Compiled Code)
        at java.net.URLClassLoader.access$1(Compiled Code)
        at java.net.URLClassLoader$1.run(Compiled Code)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Compiled Code)
        at java.lang.ClassLoader.loadClass(Compiled Code)
        at sun.misc.Launcher$AppClassLoader.loadClass(Compiled Code)
        at java.lang.ClassLoader.loadClass(Compiled Code)
```

Aufruf von `java a.A` ist schon besser:

```
> java a.A
Exception in thread "main" java.lang.NoClassDefFoundError: a/A
```

Aufruf von `java a.A` ein Verzeichnis oberhalb von `a` liefert dagegen:

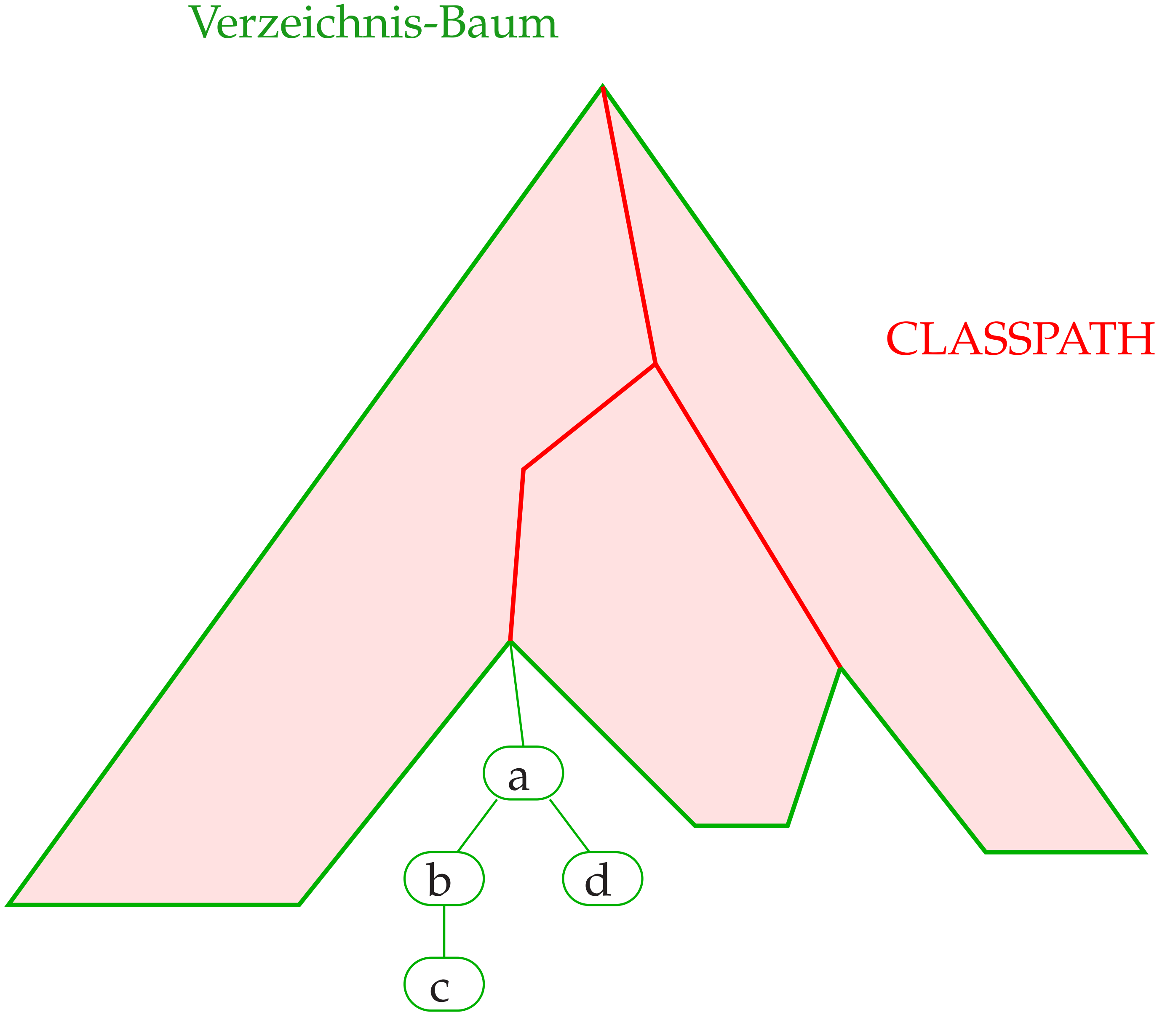
```
> java a.A
a.b.c.C@67bb4c68  a.d.D@69df4c68
```

Der Grund:

- Damit Java Verzeichnisse mit Paketen findet, sollte die Umgebungsvariable `CLASSPATH` gesetzt werden, z.B. hier mithilfe des Kommandos:

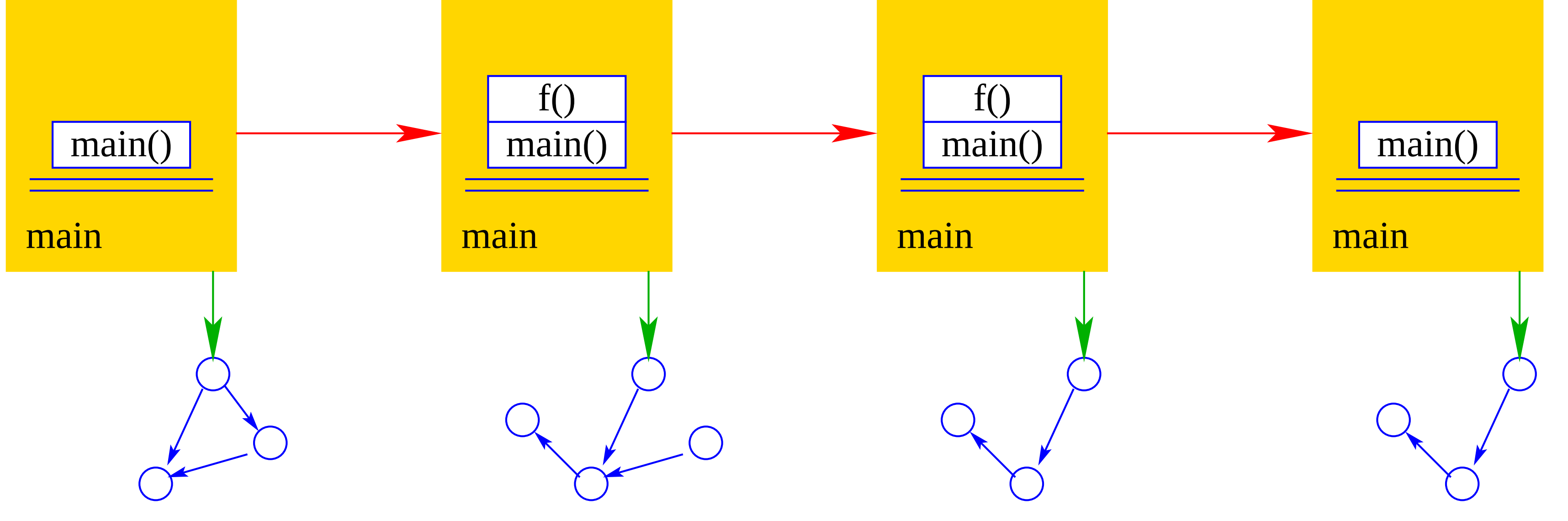
```
setenv CLASSPATH ~/java:.
```

- Diese Variable enthält die Start-Verzeichnisse, in denen bei einem Aufruf nach Klassen oder Paketen gesucht wird.
- Bei einem Aufruf `> java A` durchsucht das Laufzeit-System sämtliche in `CLASSPATH` angegebenen Verzeichnisse nach einer Datei `A.class` und führt diese aus (– sofern sie vorhanden ist).
- Bei einem Aufruf `> java a.b.c.A` sucht das Laufzeit-System eine Datei `A.class` in Unterverzeichnissen a/b/c von Verzeichnissen aus `CLASSPATH`.
- Voreingestellt ist das aktuelle Verzeichnis, d.h.: `CLASSPATH=.`

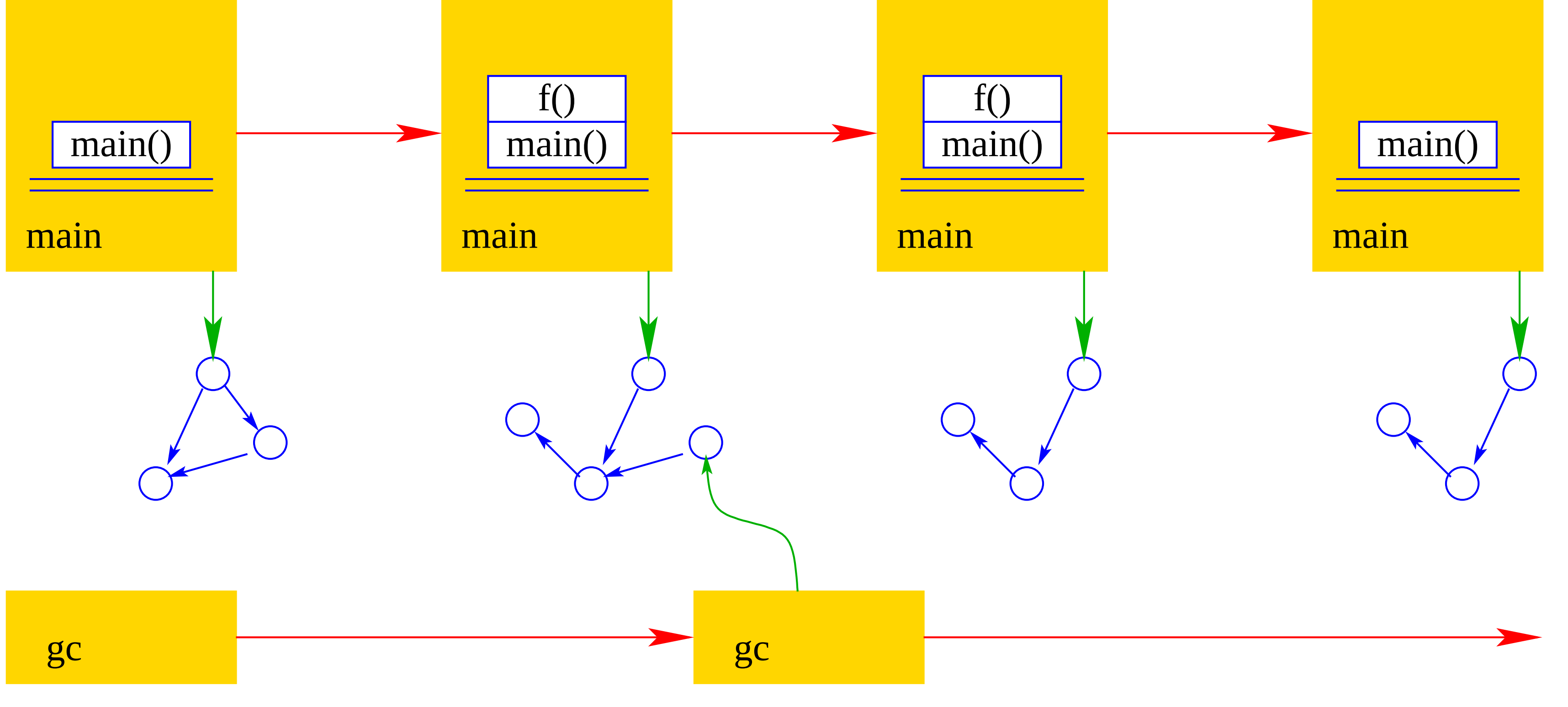

Verzeichnis-Baum
CLASSPATH
a
b
d
c

19 Threads

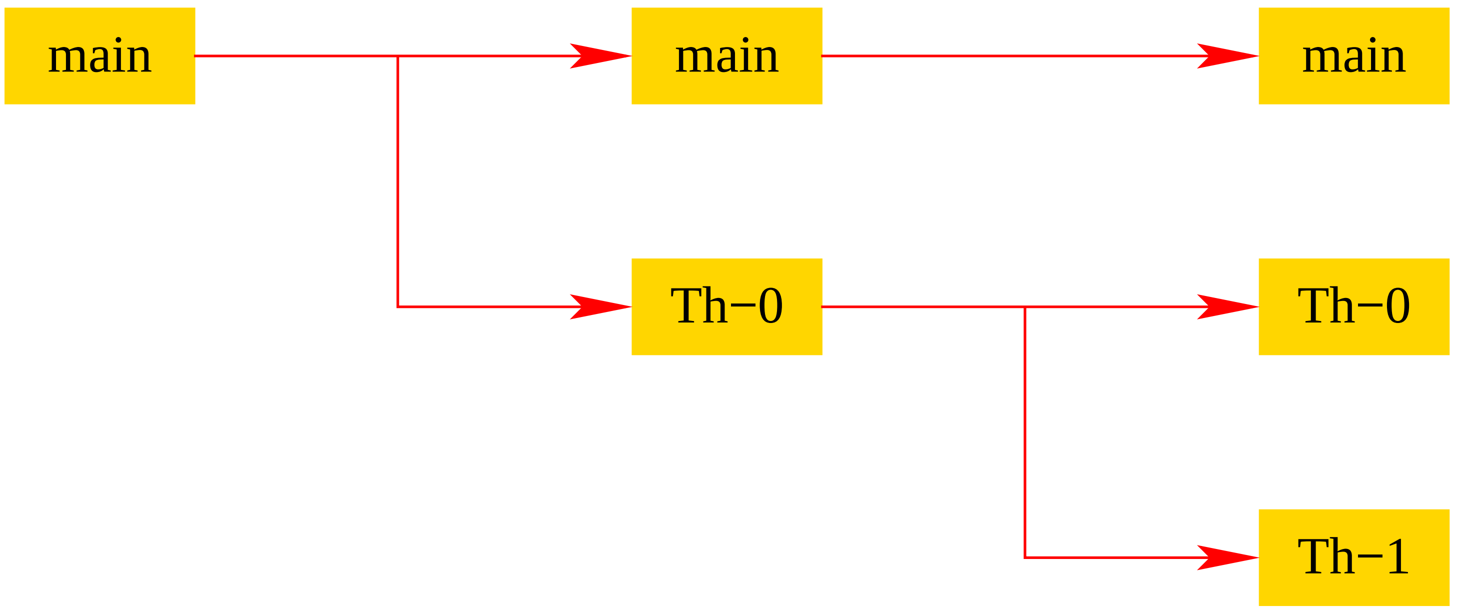
- Die Ausführung eines Java-Programms besteht in Wahrheit nicht aus einem, sondern mehreren parallel laufenden Threads.
- Ein Thread ist ein sequentieller Ausführungs-Strang.
- Der Aufruf eines Programms startet einen Thread `main`, der die Methode `main()` des Programms ausführt.
- Ein weiterer Thread, den das Laufzeitsystem parallel startet, ist die Garbage Collection.
- Die Garbage Collection soll mittlerweile nicht mehr erreichbare Objekte beseitigen und den von ihnen belegten Speicherplatz der weiteren Programm-Ausführung zur Verfügung stellen.



main()
main
f()
main()
main
f()
main()
main
main()
main


main()
main
f()
main()
main
f()
main()
main
main()
main
gc
gc

- Mehrere Threads sind auch nützlich, um
  - ... mehrere Eingabe-Quellen zu überwachen (z.B. Mouse-Klicks und Tastatur-Eingaben) ↑Graphik;
  - ... während der Blockierung einer Aufgabe etwas anderes Sinnvolles erledigen zu können;
  - ... die Rechenkraft mehrerer Prozessoren auszunutzen.
- Neue Threads können deshalb vom Programm selbst erzeugt und gestartet werden.
- Dazu stellt Java die Klasse `Thread` und das Interface `Runnable` bereit.


main
main
main
Th−0
Th−0
Th−1

Beispiel:

```
public class MyThread extends Thread {
   public void hello(String s) {
      System.out.println(s);
   }
   public void run() {
      hello("I’m running ...");
   }  // end of run()
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();
      System.out.println("Thread has been started ...");
   }  // end of main()
}     // end of class MyThread
```

- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede (konkrete) Unterklasse von `Thread` muss die abstrakte Objekt-Methode `public void run();` implementieren.
- Ist `t` ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für `t` wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

Beispiel:

```
public class MyRunnable implements Runnable {
   public void hello(String s) {
      System.out.println(s);
   }
   public void run() {
      hello("I'm running ...");
   }  // end of run()
   public static void main(String[] args) {
      Thread t = new Thread(new MyRunnable());
      t.start();
      System.out.println("Thread has been started ...");
   }  // end of main()
}     // end of class MyRunnable
```

- Auch das Interface `Runnable` verlangt die Implementierung einer Objekt-Methode `public void run();`
- `public Thread(Runnable obj);` legt für ein `Runnable`-Objekt obj ein `Thread`-Objekt an.
- Ist `t` das `Thread`-Objekt für das `Runnable` obj, dann bewirkt der Aufruf `t.start();` das folgende:

  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für obj wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

Mögliche Ausführungen:

```
Thread has been started ...
I'm running ...
```

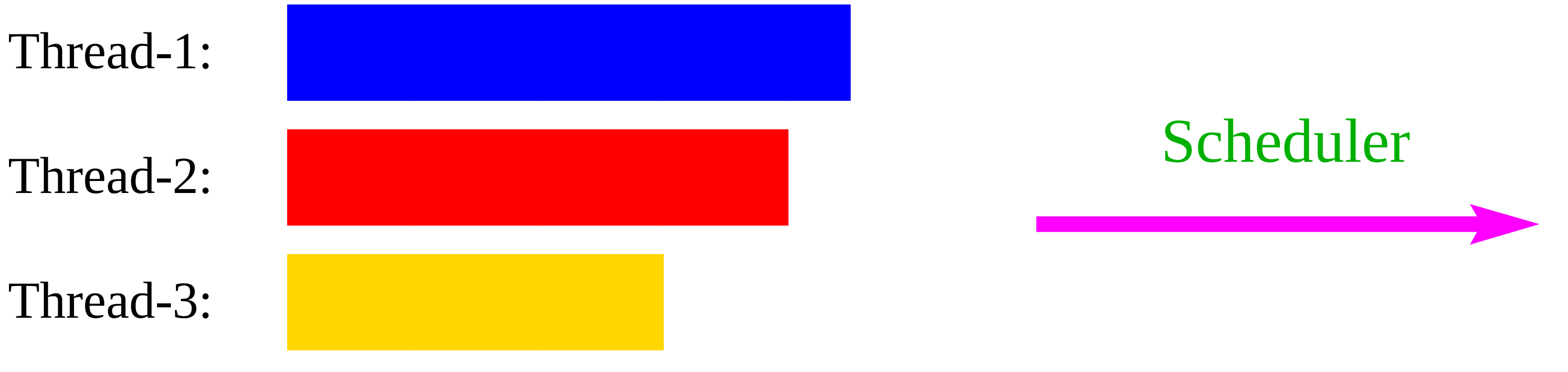
... oder:

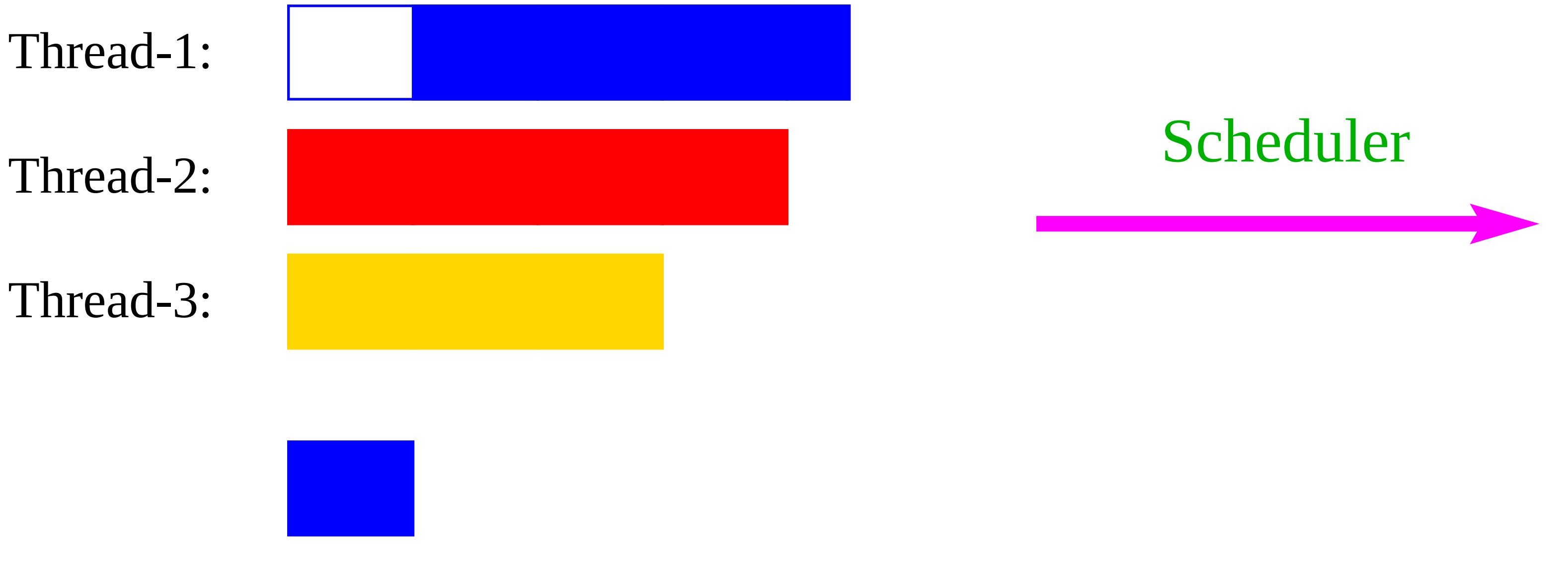
```
I'm running ...
Thread has been started ...
```

- Ein Thread kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) zur Ausführung zugeteilt worden ist.
- Im Allgemeinen gibt es mehr Threads als CPUs.
- Der Scheduler verwaltet die verfügbaren CPUs und teilt sie den Threads zu.
- Bei verschiedenen Programm-Läufen kann diese Zuteilung verschieden aussehen!!!
- Es gibt verschiedene Politiken, nach denen sich Scheduler richten können ↑Betriebssysteme.

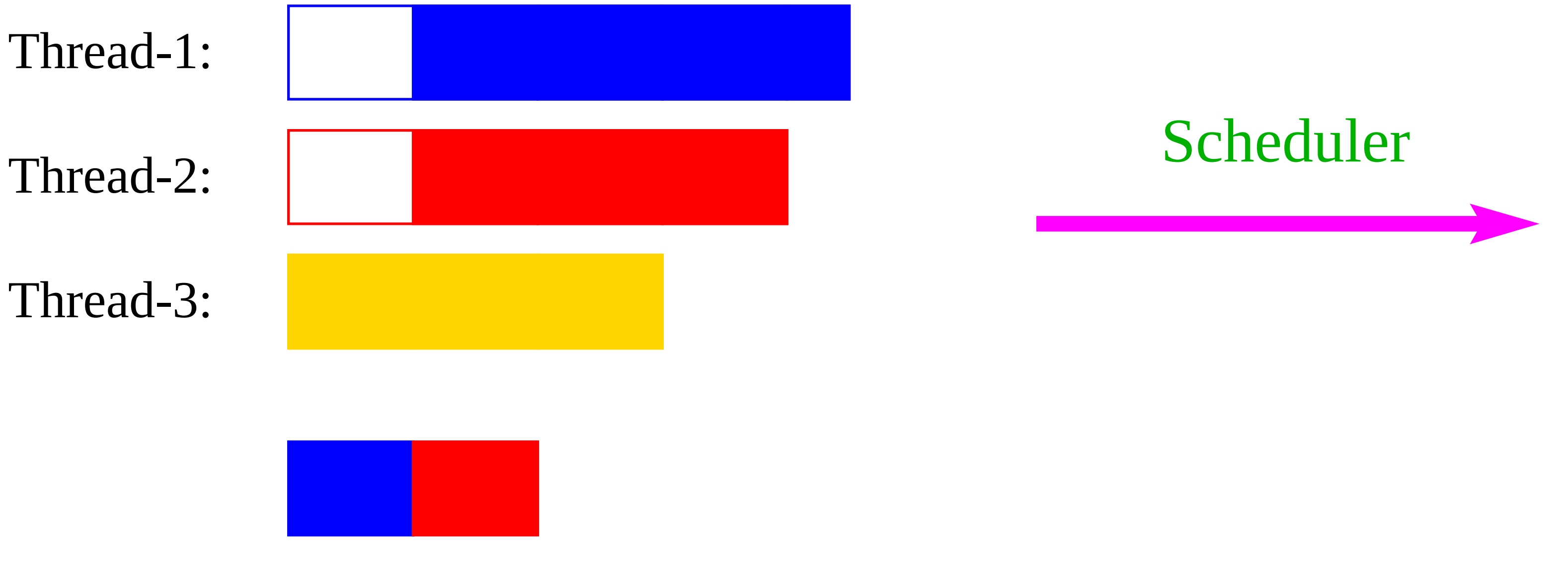
1. Zeitscheiben-Verfahren:

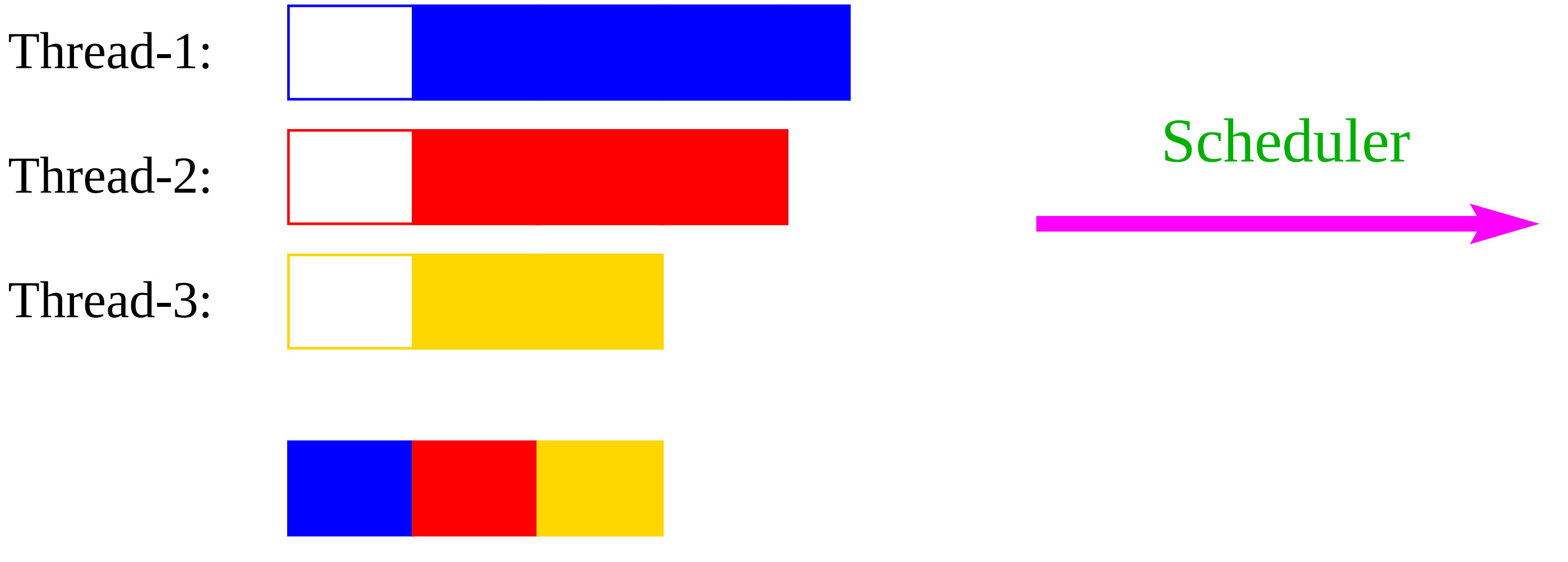
- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

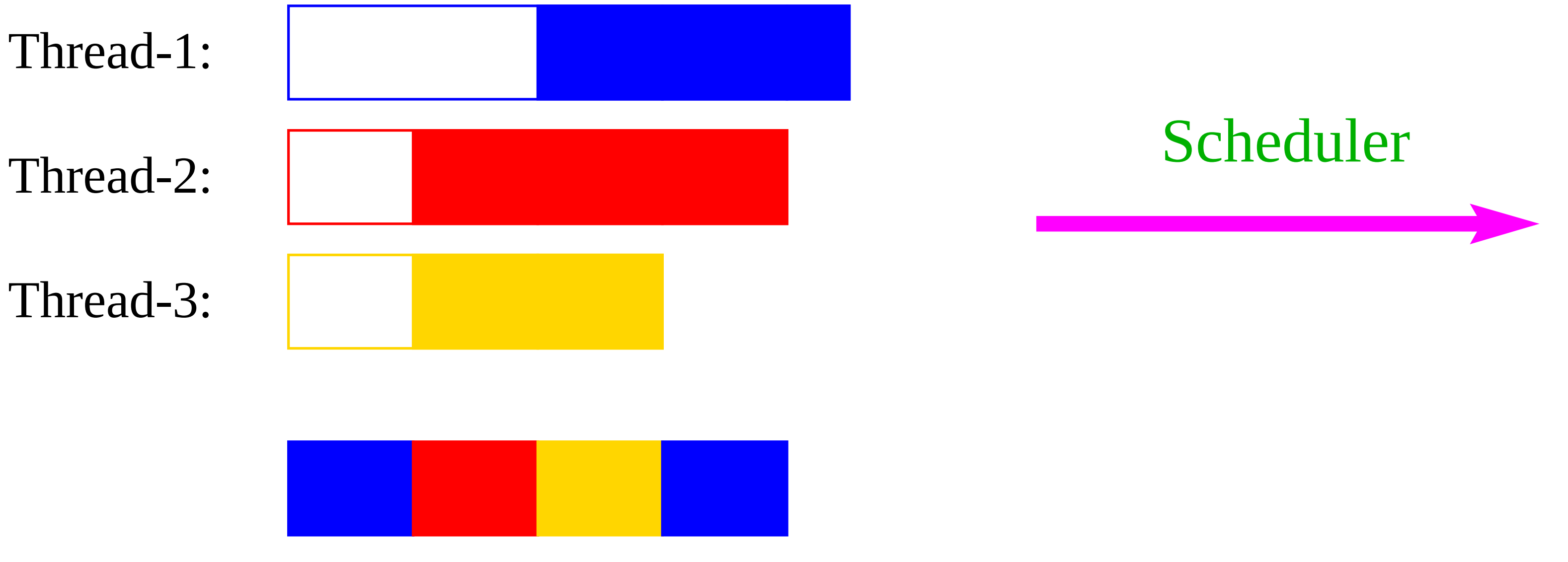

Thread-1:
Thread-2:
Thread-3:
Scheduler

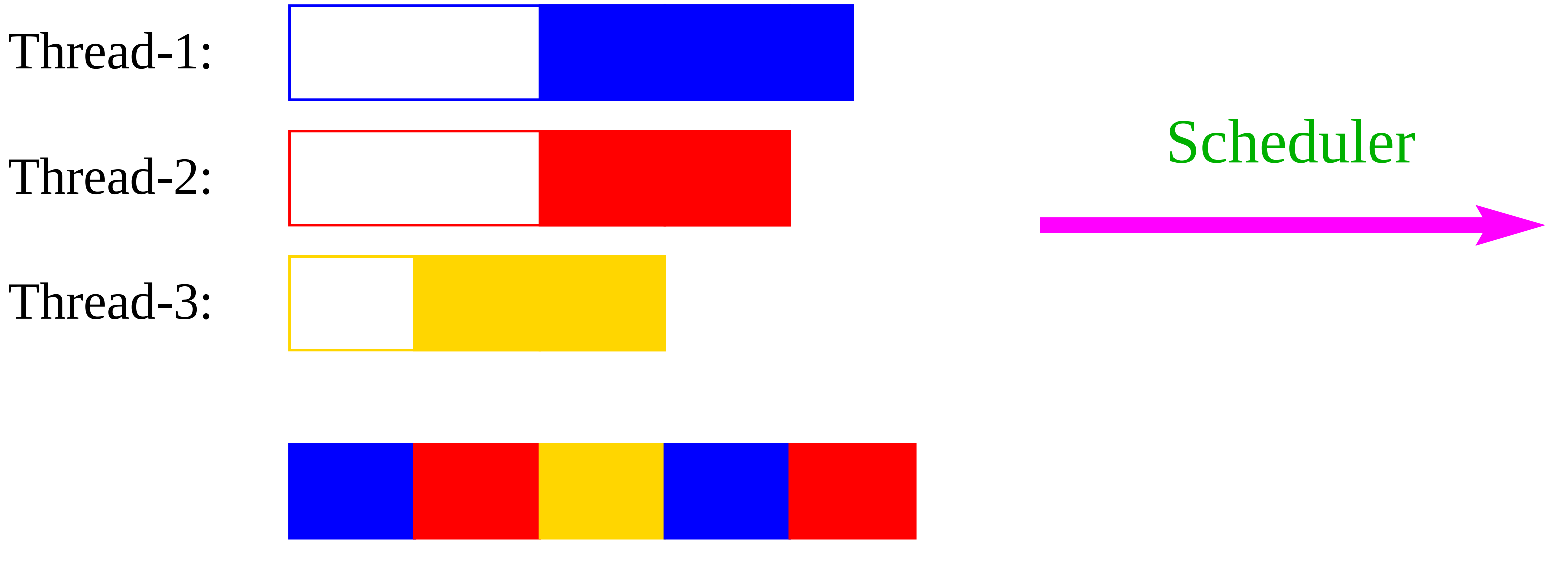


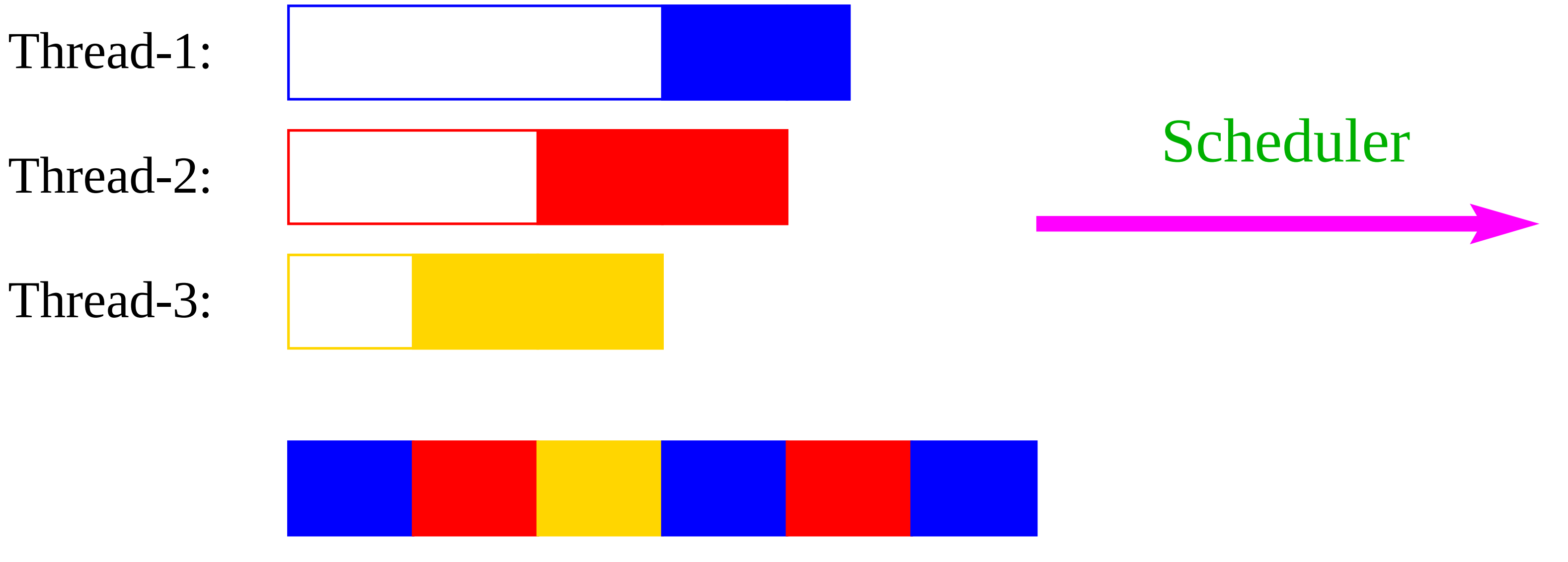
Thread-1:
Thread-2:
Thread-3:
Scheduler

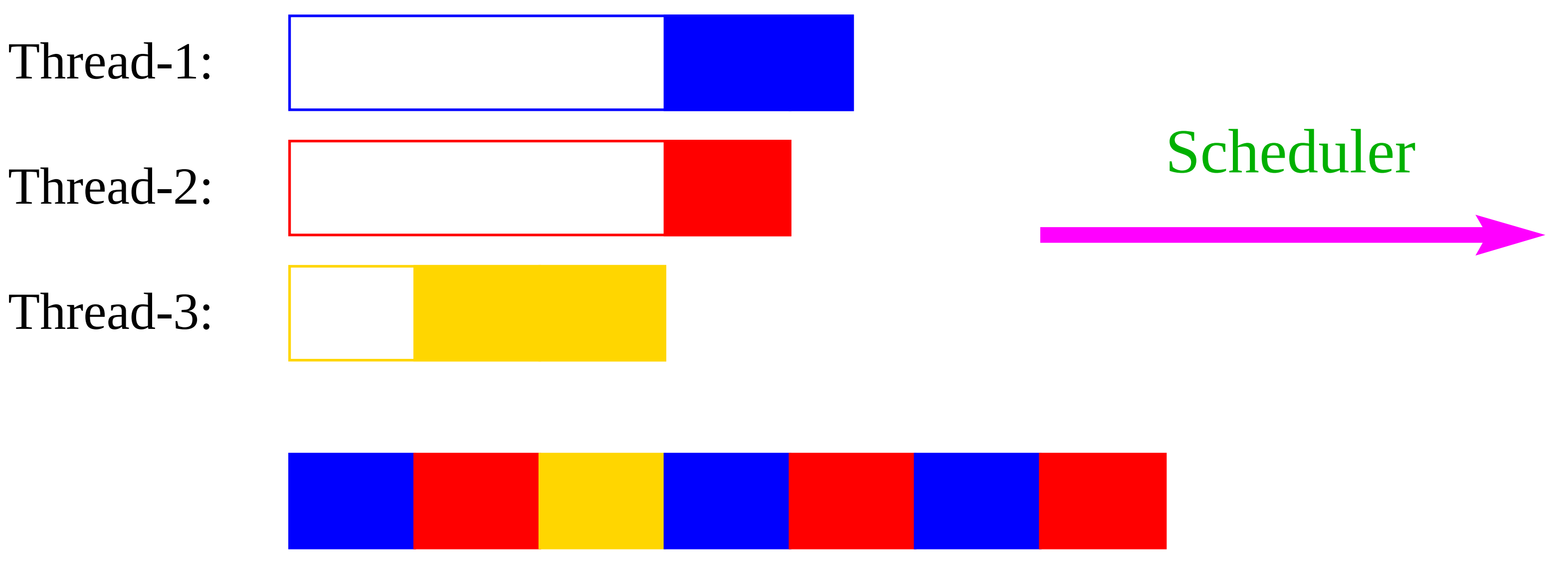

Thread-1:
Thread-2:
Thread-3:
Scheduler

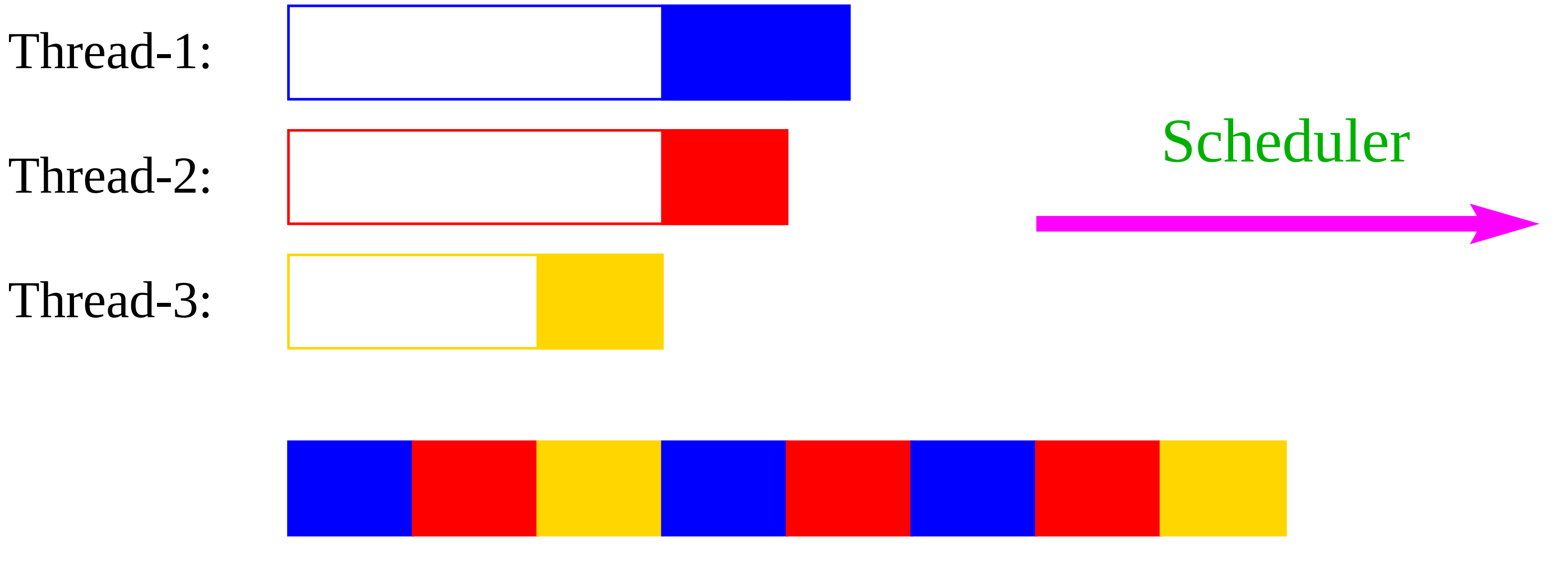

Thread-1:
Thread-2:
Thread-3:
Scheduler

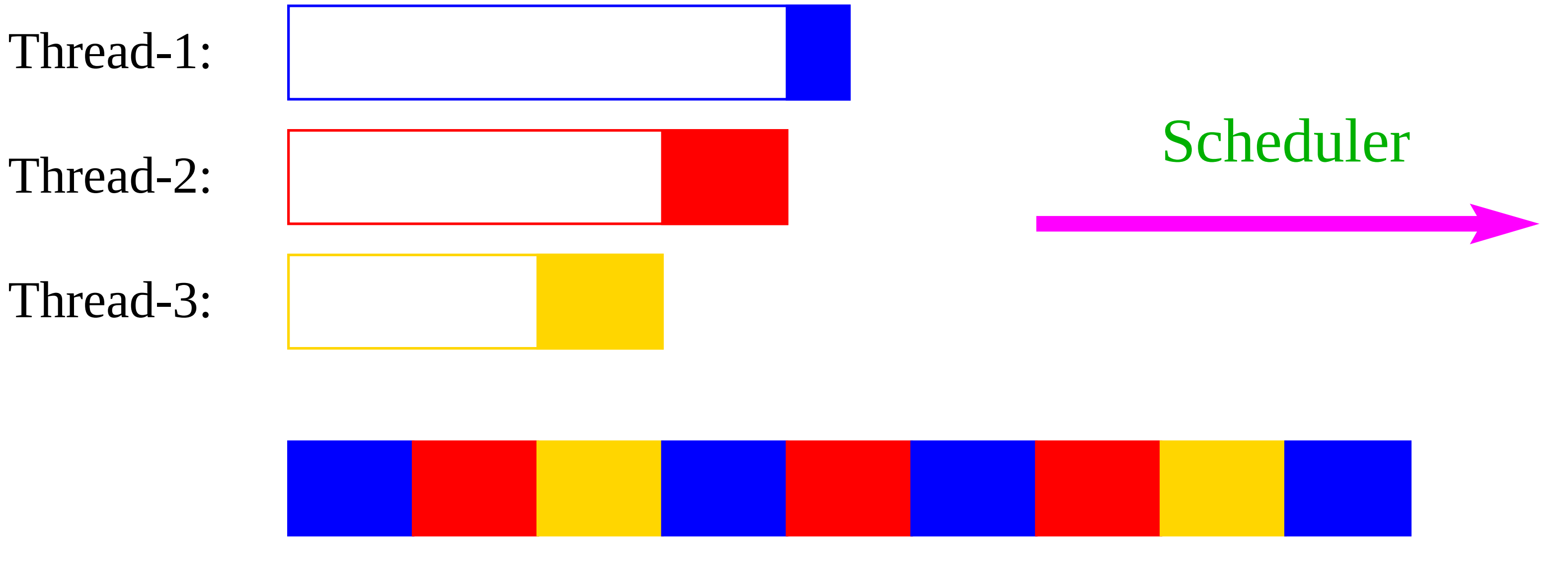

Thread-1:
Thread-2:
Thread-3:
Scheduler

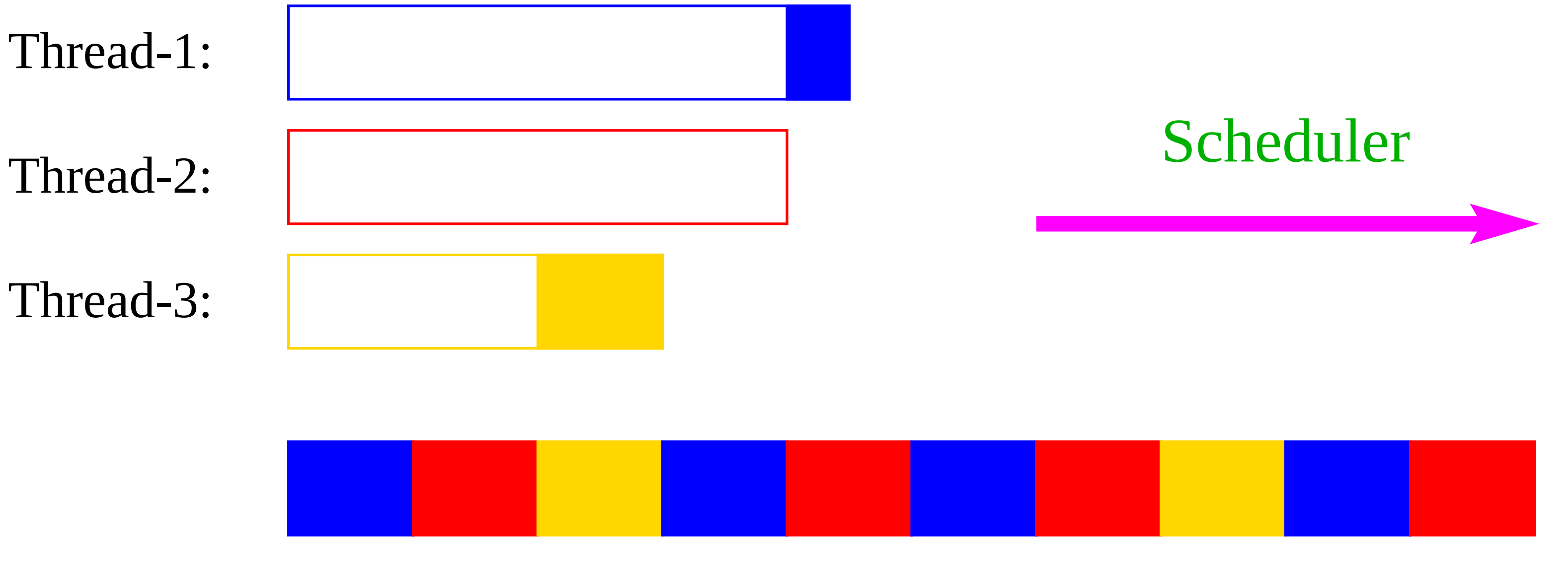

Thread-1:
Thread-2:
Thread-3:
Scheduler

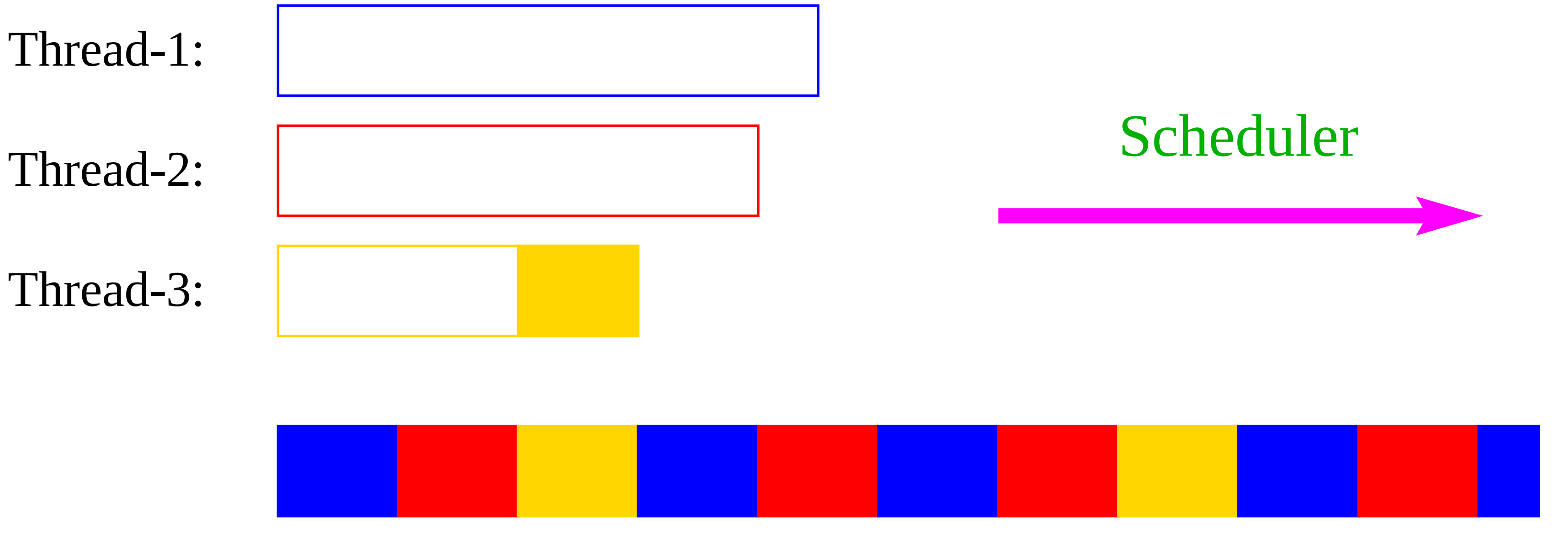

Thread-1:
Thread-2:
Thread-3:
Scheduler

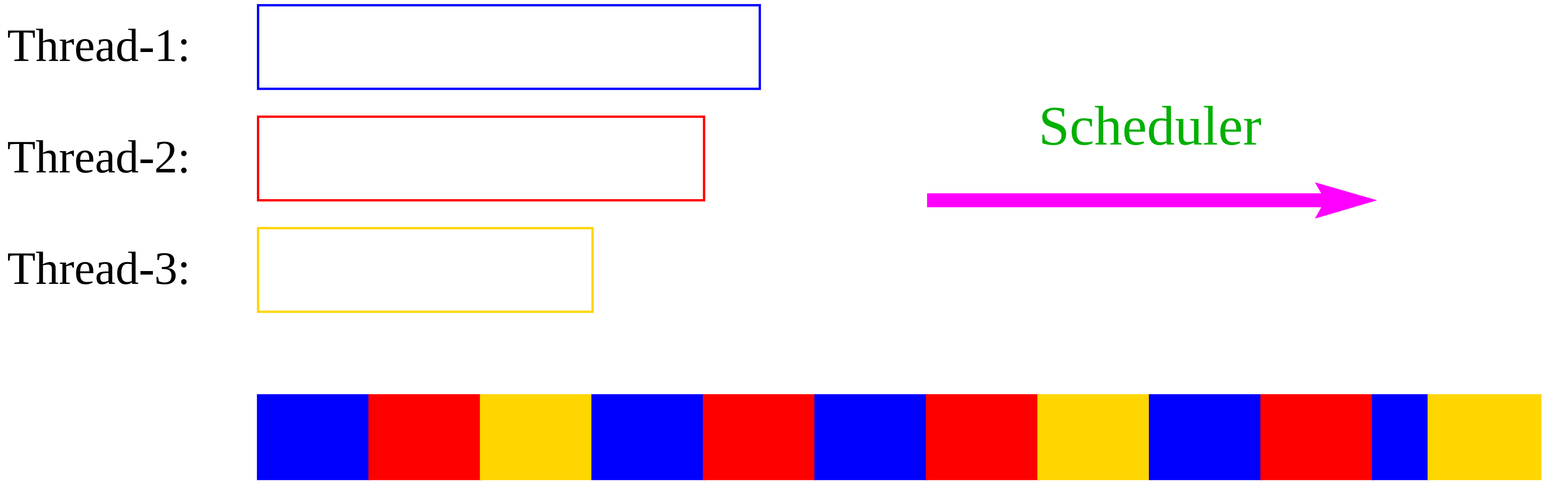

Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler

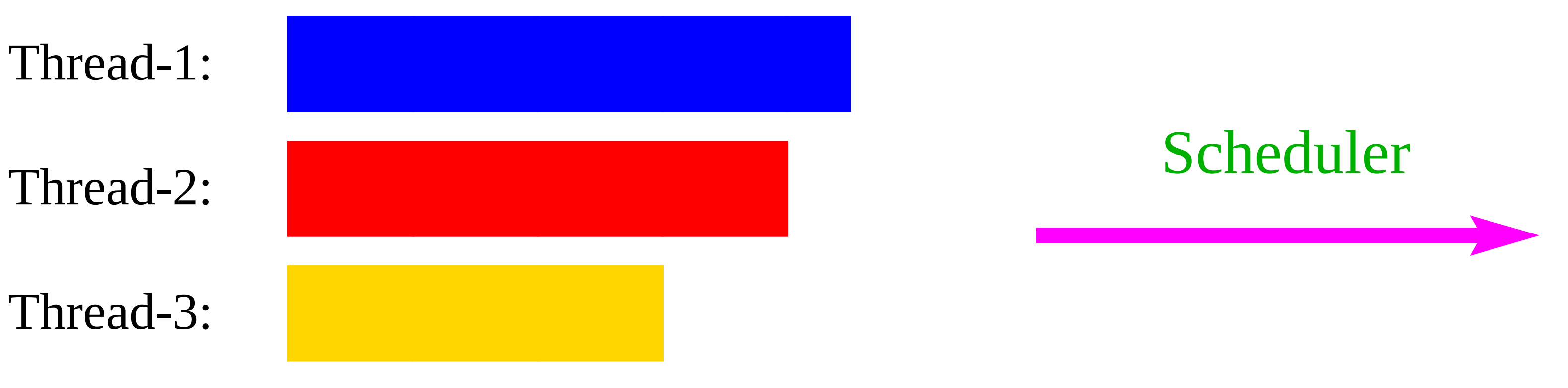

Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler

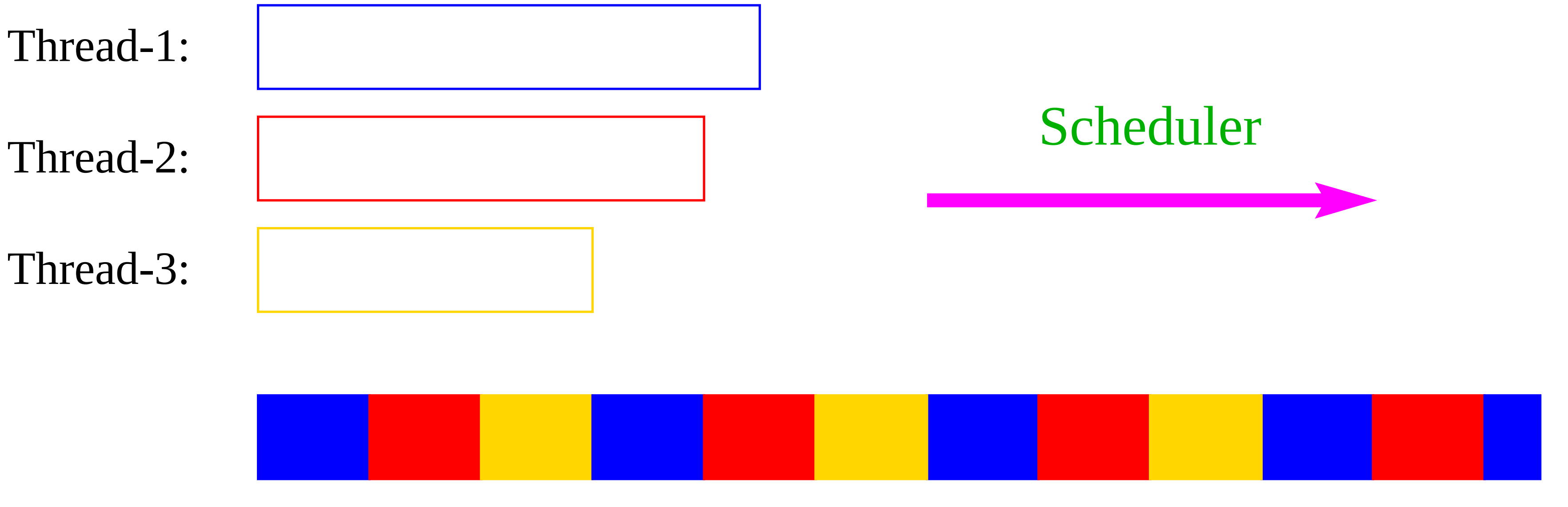
Achtung:

Eine andere Programm-Ausführung mag dagegen liefern:



Achtung:

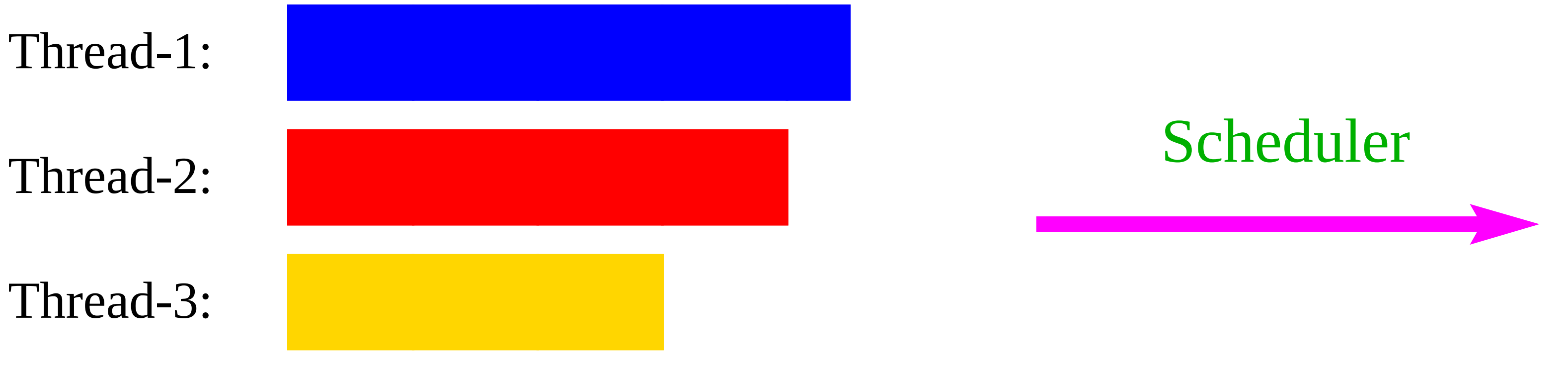
Eine andere Programm-Ausführung mag dagegen liefern:

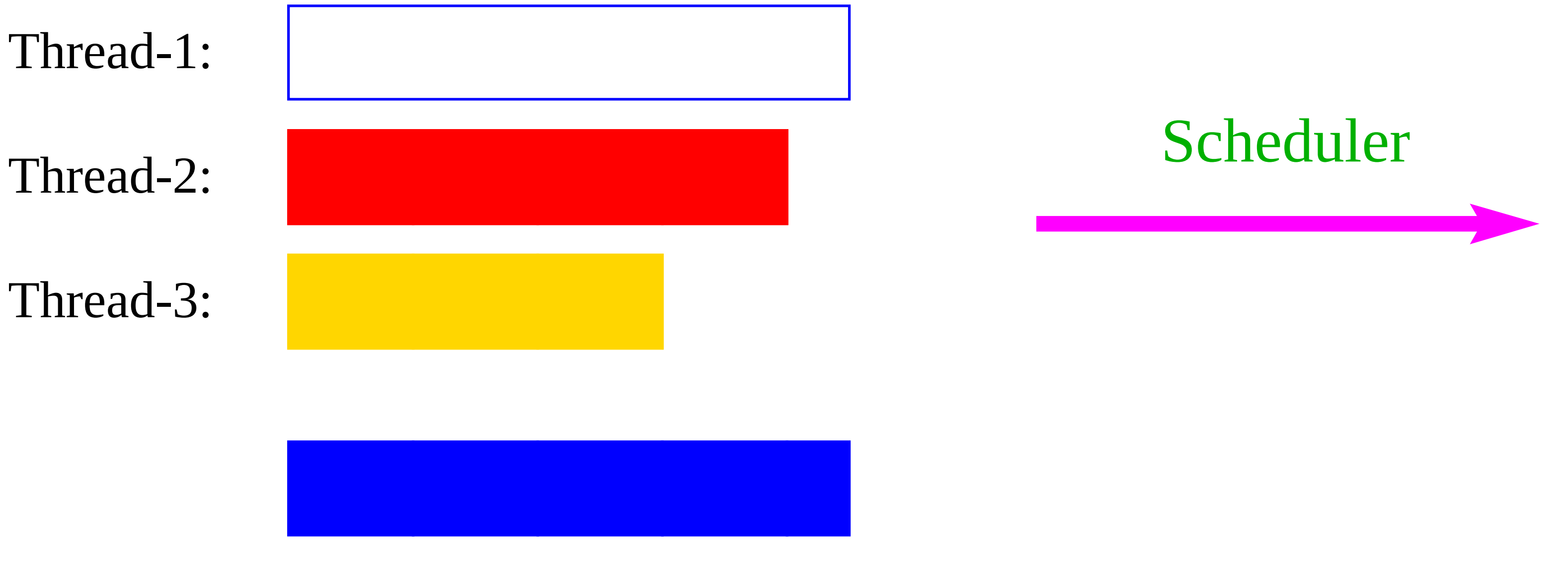


- Ein Zeitscheiben-Scheduler versucht, jeden Thread fair zu behandeln, d.h. ab und zu Rechenzeit zuzuordnen – egal, welche Threads sonst noch Rechenzeit beanspruchen.
- Kein Thread hat jedoch Anspruch auf einen bestimmten Time-Slice.
- Für den Programmierer sieht es so aus, als ob sämtliche Threads "echt" parallel ausgeführt werden, d.h. jeder über eine eigene CPU verfügt :-)

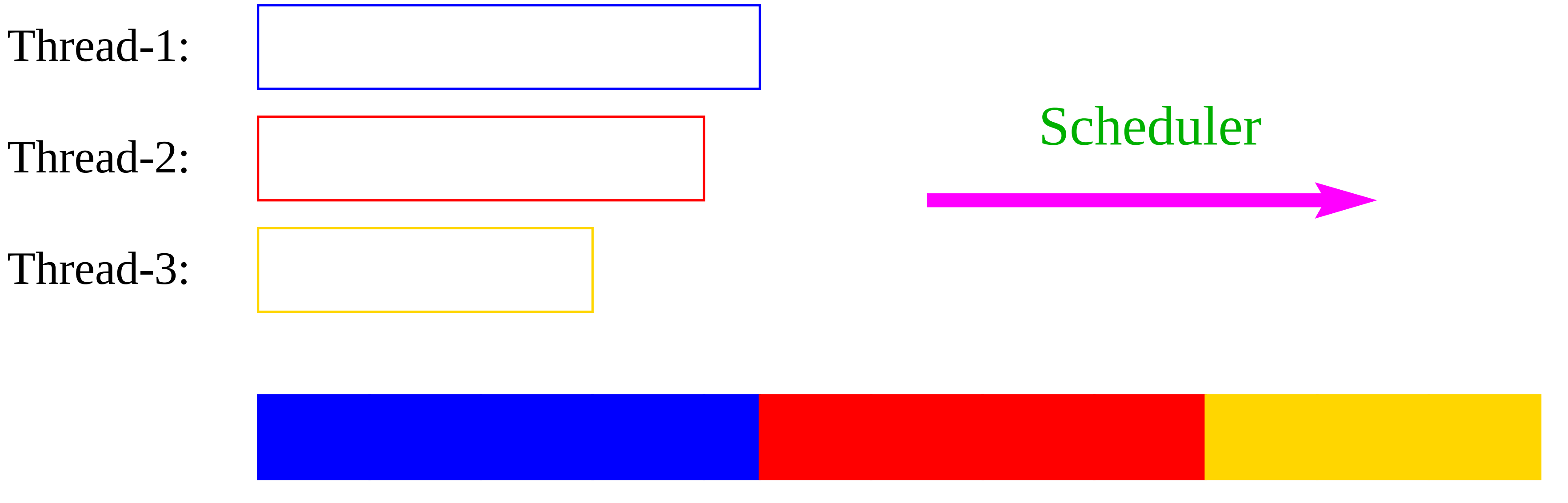
2. Naives Verfahren:

- Erhält ein Thread eine CPU, darf er laufen, so lange er will ...
- Gibt er die CPU wieder frei, darf ein anderer Thread arbeiten ...


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler


Thread-1:
Thread-2:
Thread-3:
Scheduler



Thread-1:
Thread-2:
Thread-3:
Scheduler

Beispiel:

```
public class Start extends Thread {
public void run() {
   System.out.println("I'm running ...");
   while(true) ;
}
public static void main(String[] args) {
   (new Start()).start();
   (new Start()).start();
   (new Start()).start();
   System.out.println("main is running ...");
   while(true) ;
}
}  // end of class Start
```

... liefert als Ausgabe (bei naivem Scheduling und einer CPU) :

```
main is running ...
```

... liefert als Ausgabe (bei naivem Scheduling und einer CPU) :

```
main is running ...
```

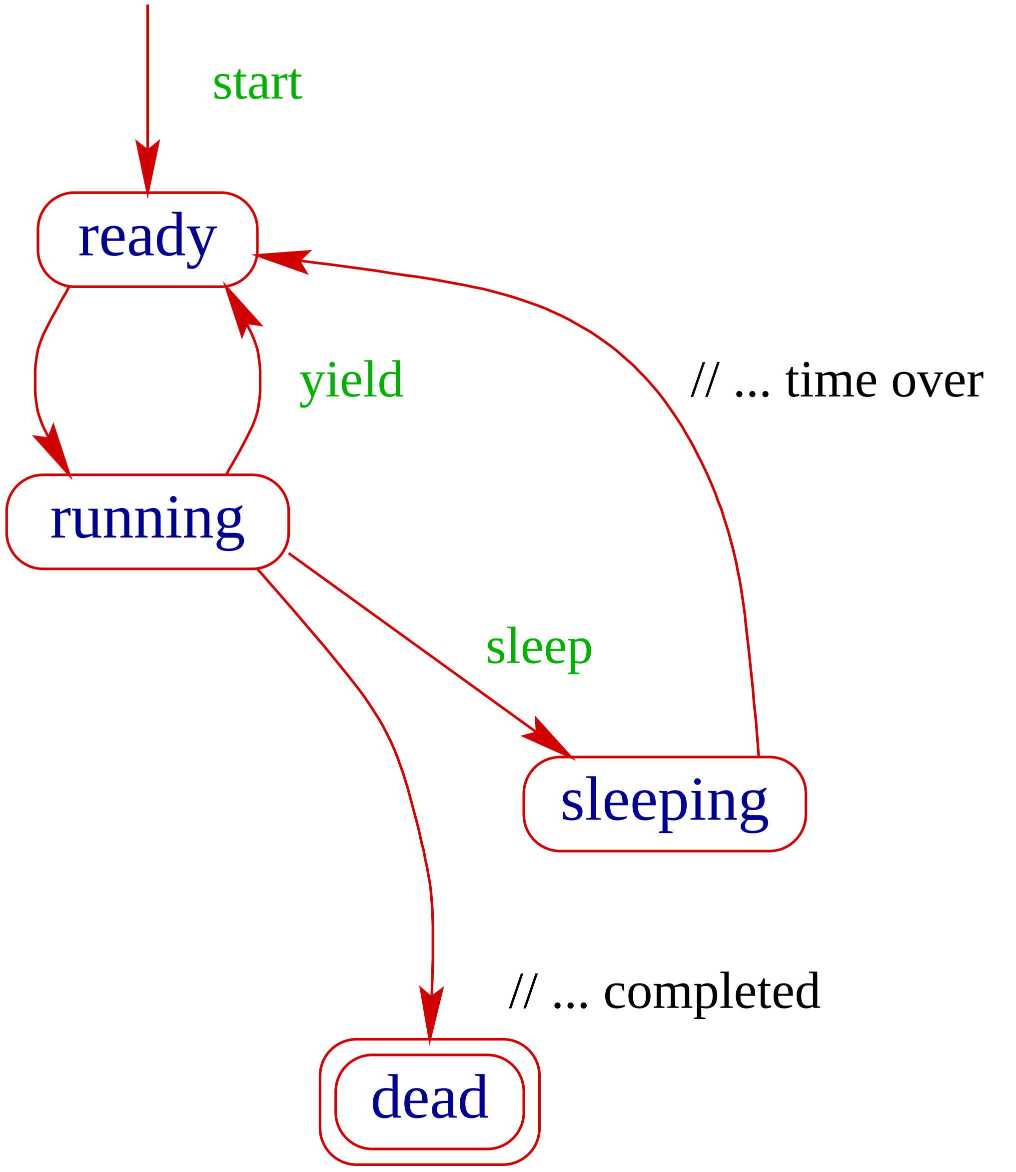
- Weil `main` nie fertig wird, erhalten die anderen Threads keine Chance, sie verhungern.
- Faires Scheduling mit einem Zeitscheiben-Verfahren würde z.B. liefern:

```
I'm running ...
main is running ...
I'm running ...
I'm running ...
```

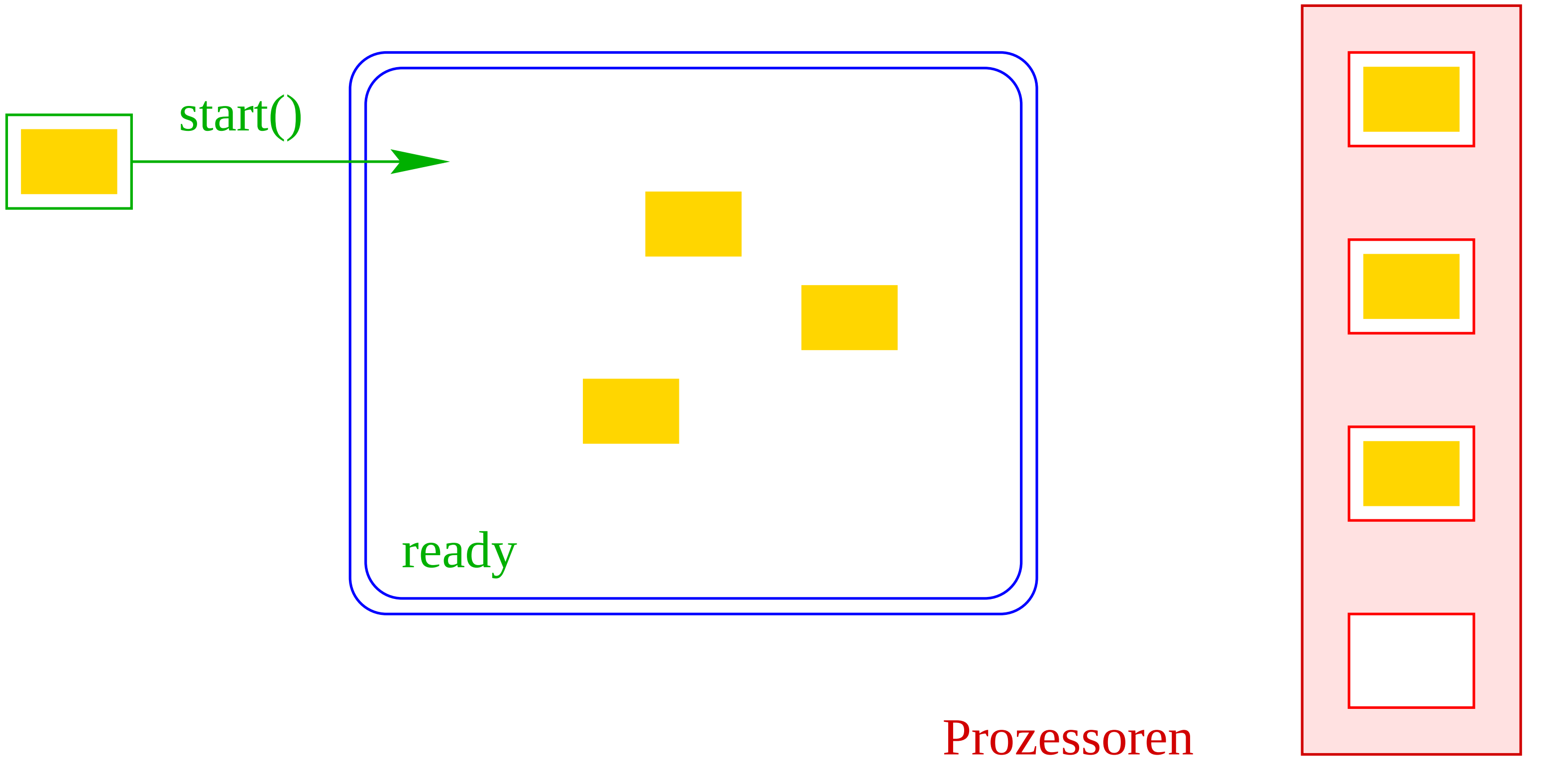
- Java legt nicht fest, wie intelligent der Scheduler ist.
- Die aktuelle Implementierung unterstützt faires Scheduling :-)
- Programme sollten aber für jeden Scheduler das gleiche Verhalten zeigen. Das heißt:
- ... Threads, die aktuell nichts sinnvolles zu tun haben, z.B. weil sie auf Verstreichen der Zeit oder besseres Wetter warten, sollten stets ihre CPU anderen Threads zur Verfügung stellen.
- ... Selbst wenn Threads etwas Vernünftiges tun, sollten sie ab und zu andere Threads laufen lassen.

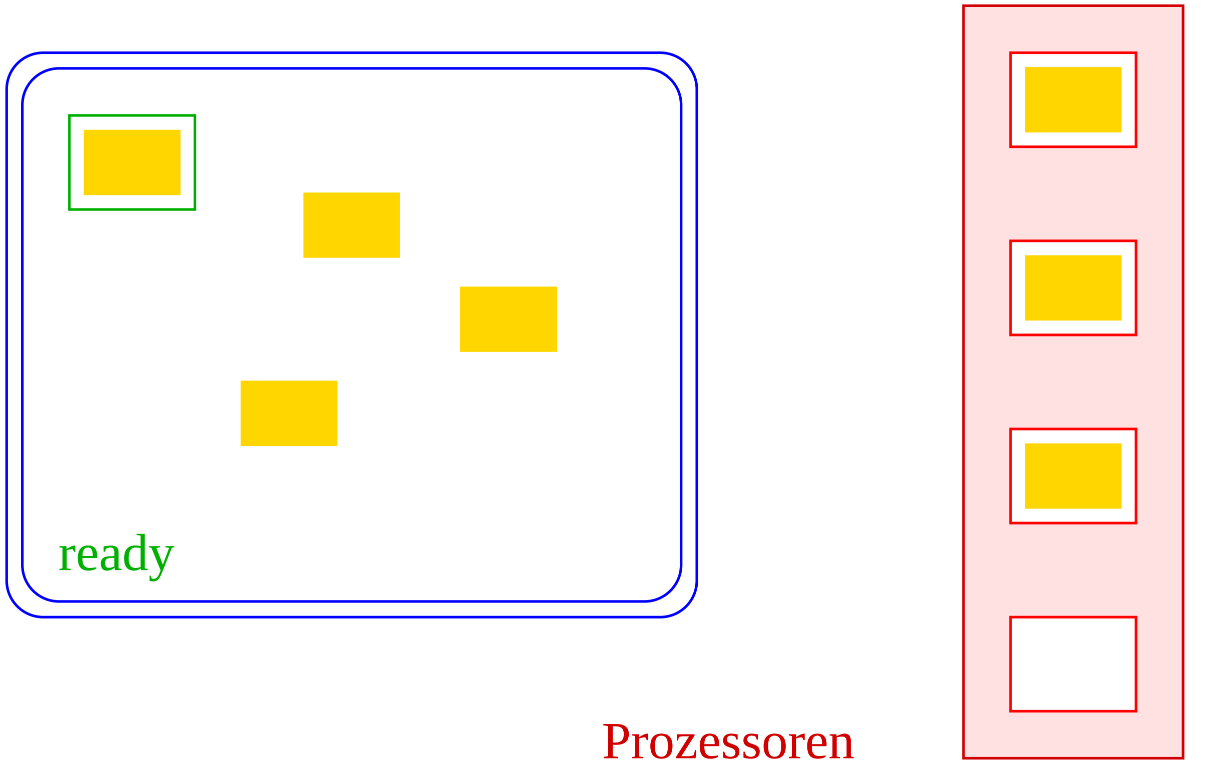
  (Achtung: Wechsel des Threads ist teuer!!!)
- Dazu verfügt jeder Thread über einen Zustand, der bei der Vergabe von Rechenzeit beücksichtigt wird.

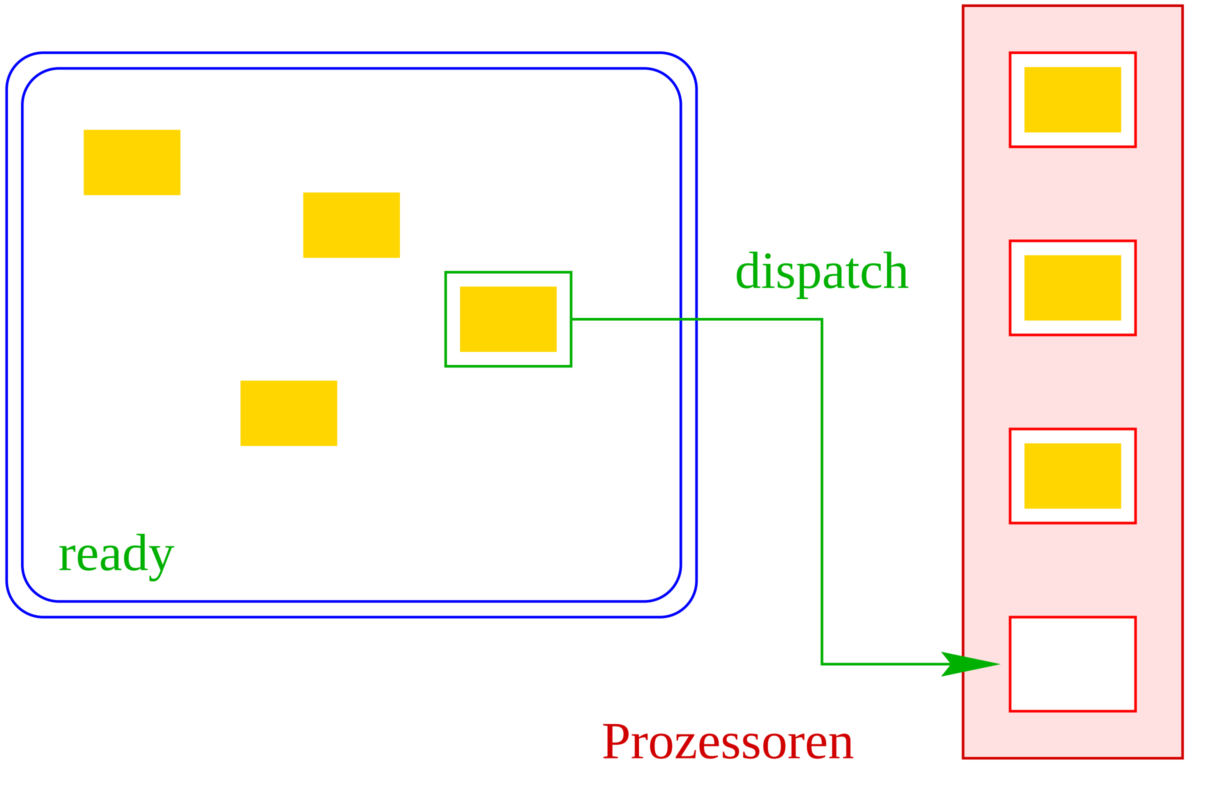
Einige Thread-Zustände:

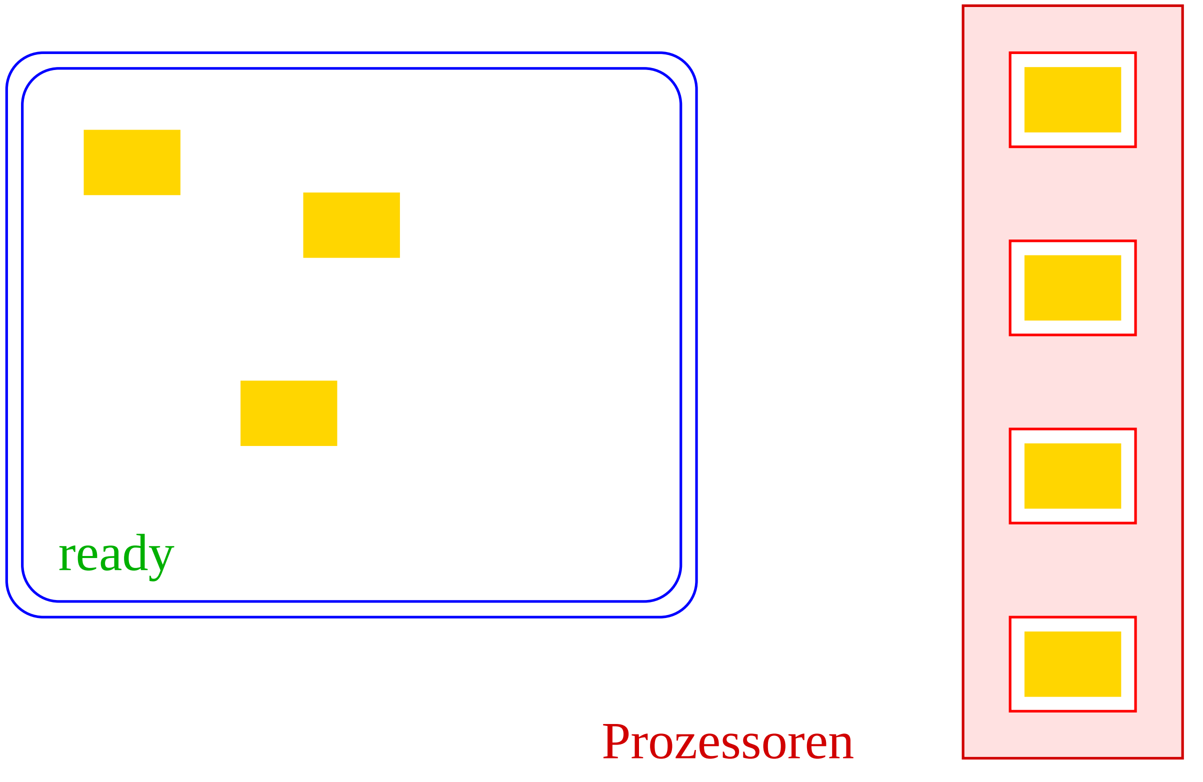


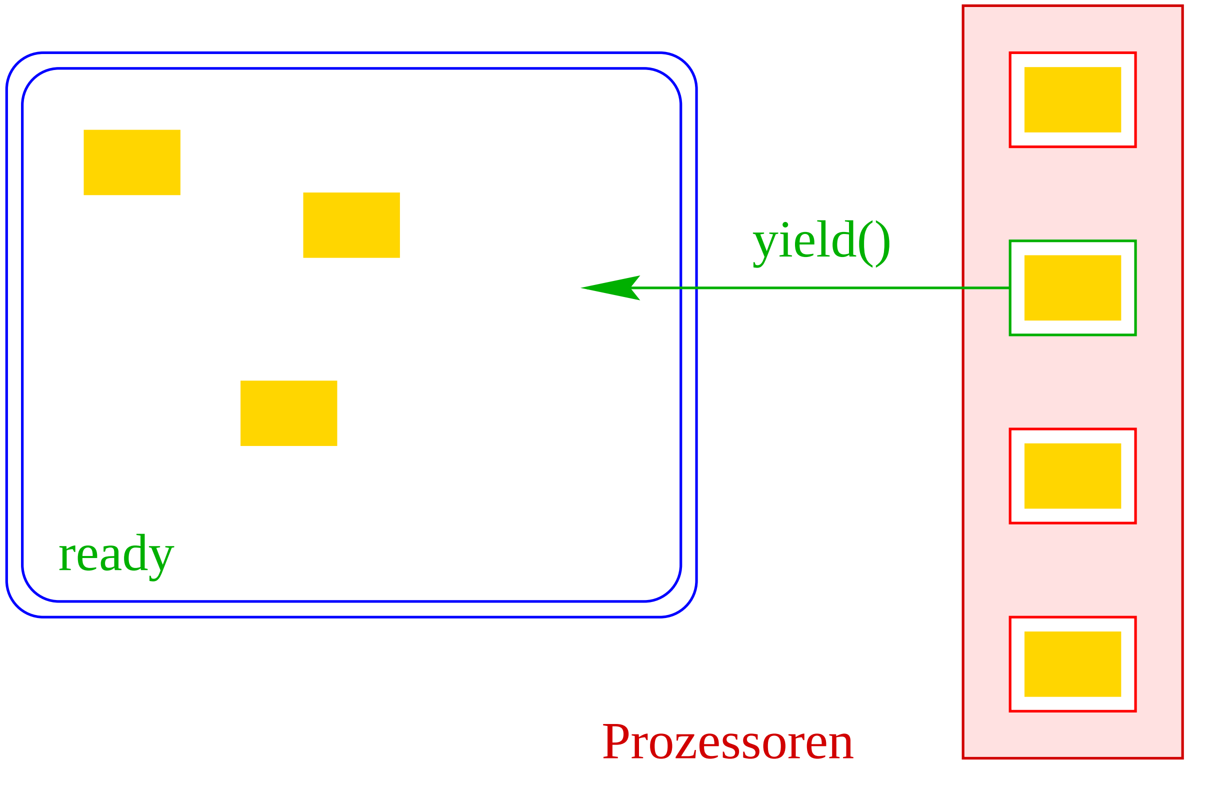
- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu (“dispatching”). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programm-Ausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

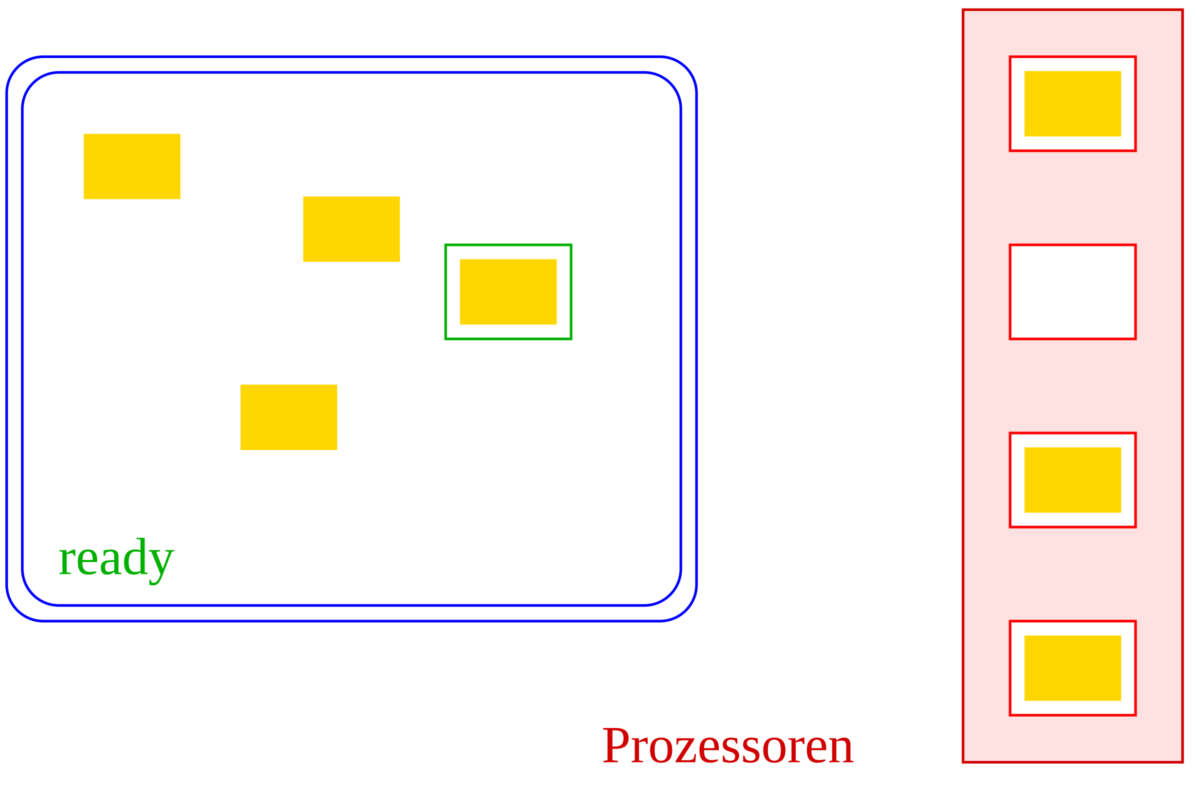

start()
ready
Prozessoren

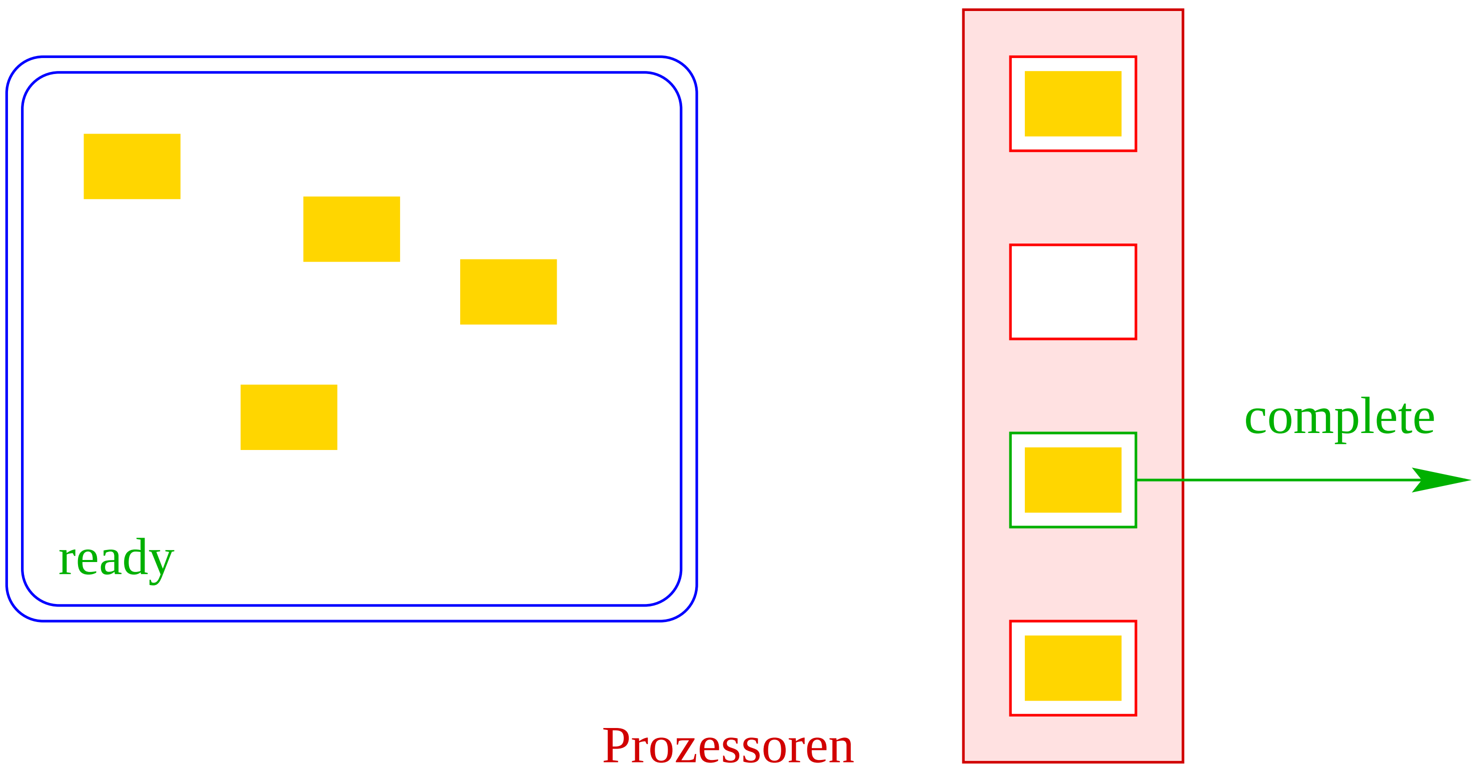

ready
Prozessoren

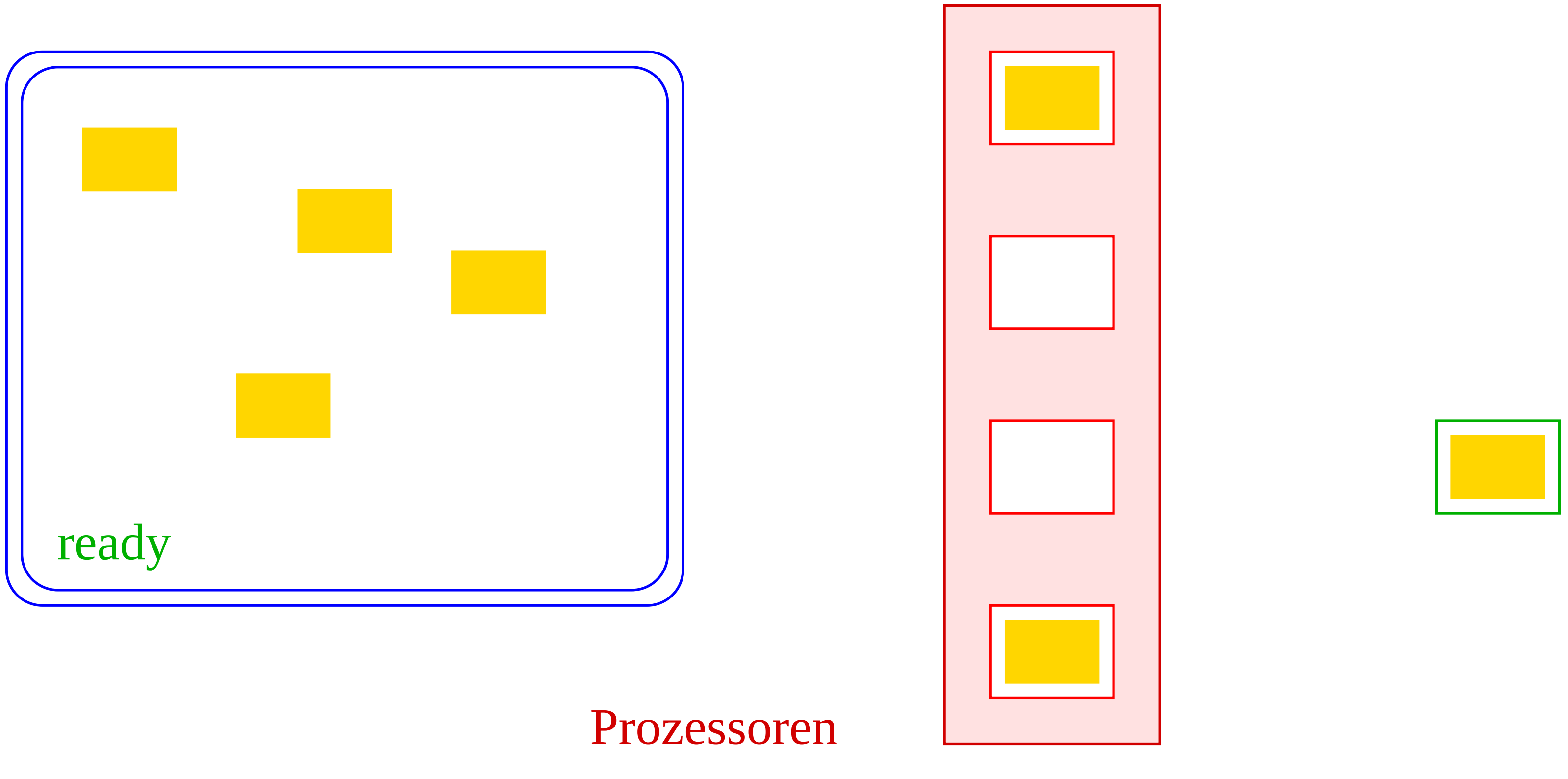

dispatch
ready
Prozessoren


ready
Prozessoren


yield()
ready
Prozessoren


ready
Prozessoren


complete
ready
Prozessoren


ready
Prozessoren

19.1 Monitore

- Damit Threads sinnvoll miteinander kooperieren können, müssen sie miteinander Daten austauschen.
- Zugriff mehrerer Threads auf eine gemeinsame Variable ist problematisch, weil nicht feststeht, in welcher Reihenfolge die Threads auf die Variable zugreifen.
- Ein Hilfsmittel, um geordnete Zugriffe zu garantieren, sind Monitore.

... ein Beispiel:

```
public class Inc implements Runnable {
   private static int x = 0;
   private static void pause(int t) {
      try {
         Thread.sleep((int) (Math.random()*t*1000));
      } catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   }
   public void run() {
      String s = Thread.currentThread().getName();
      pause(3); int y = x;
      System.out.println(s+ " read "+y);
      pause(4); x = y+1;
      System.out.println(s+ " wrote "+(y+1));
   }
```

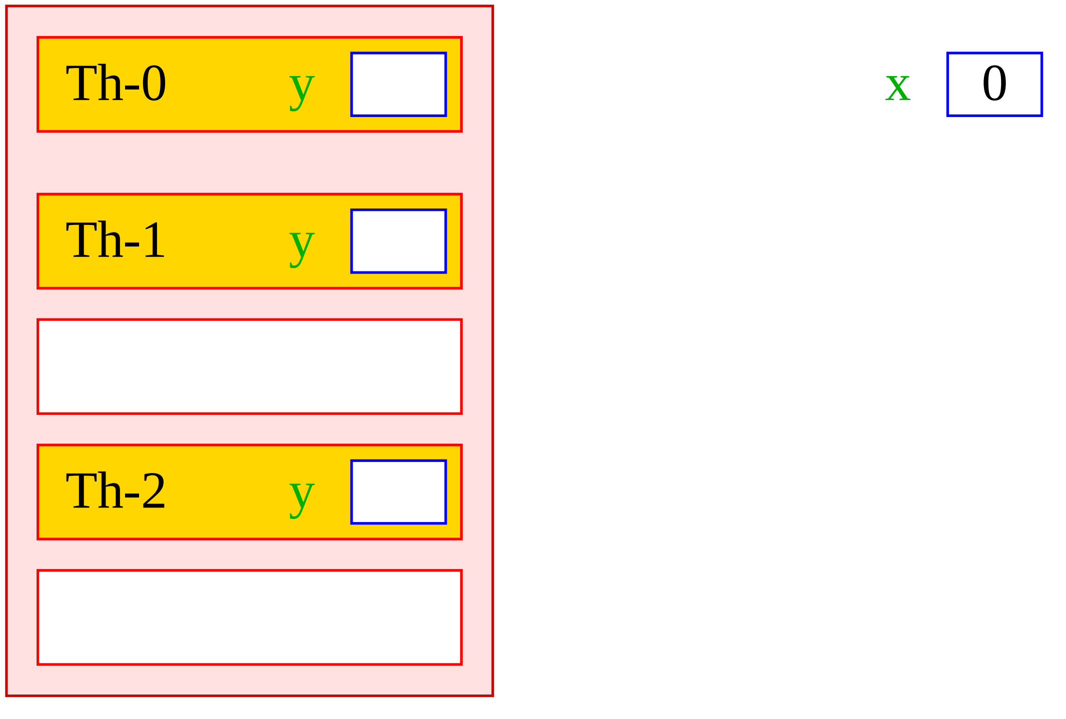
```
        ...
        public static void main(String[] args) {
            (new Thread(new Inc())).start();
            pause(2);
            (new Thread(new Inc())).start();
            pause(2);
            (new Thread(new Inc())).start();
        }
    } // end of class Inc
```

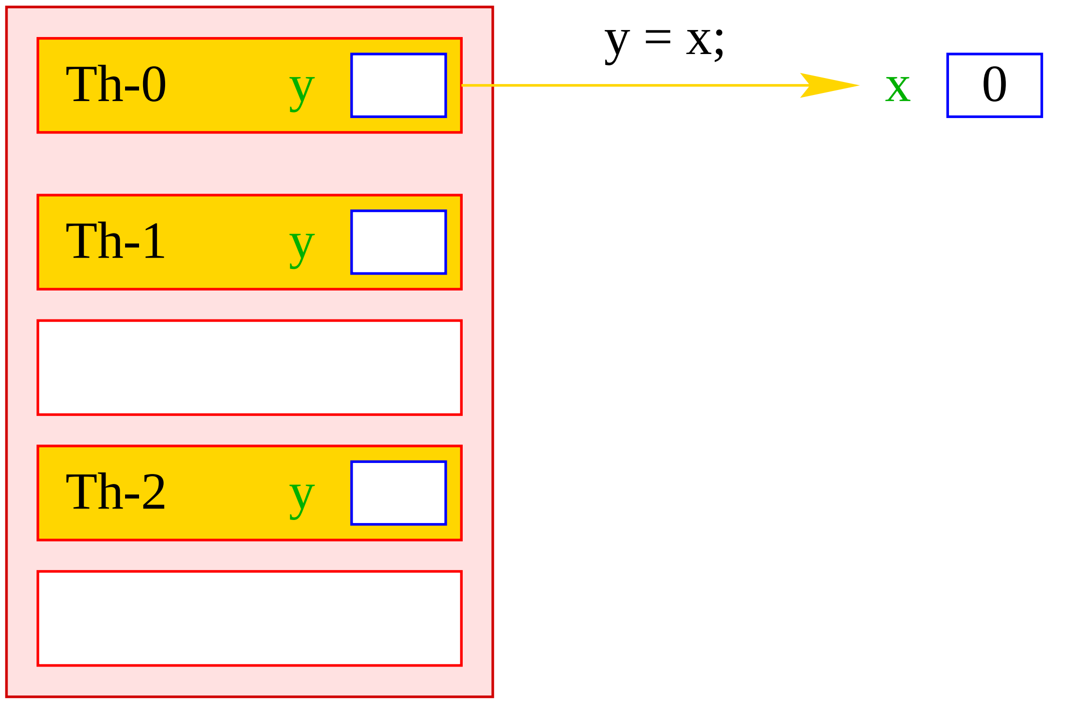
- `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- `public final String getName();` liefert den Namen des Thread-Objekts.
- Das Programm legt für drei Objekte der Klasse `Inc` Threads an.
- Die Methode `run()` inkrementiert die Klassen-Variable `x`.

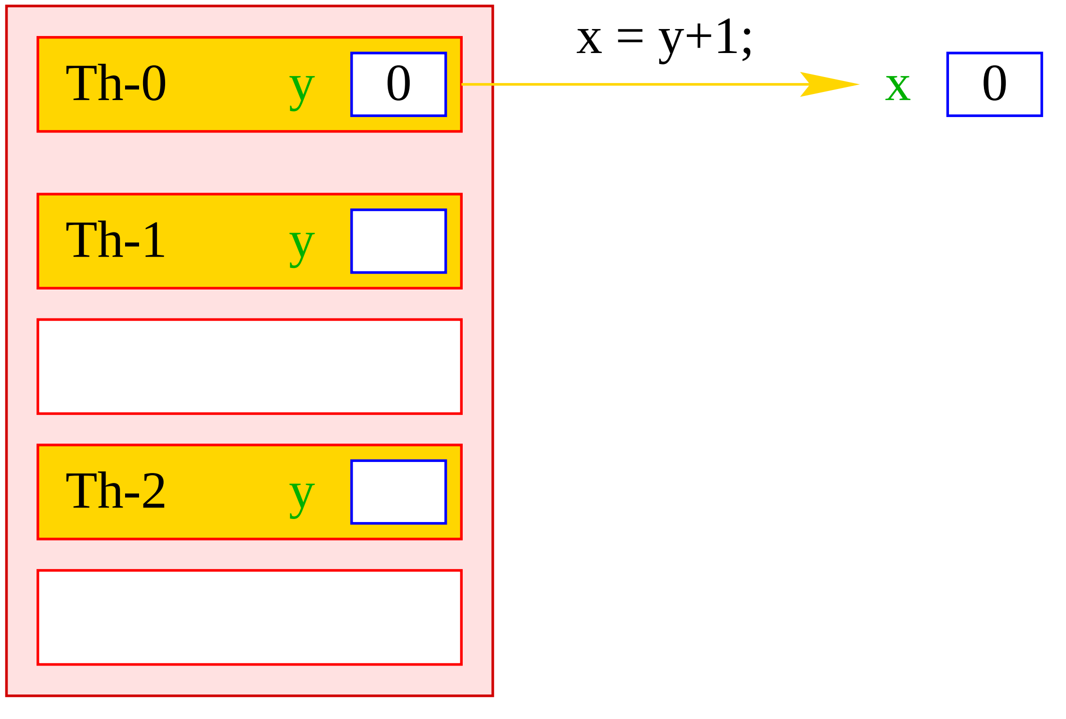
Die Ausführung liefert z.B.:

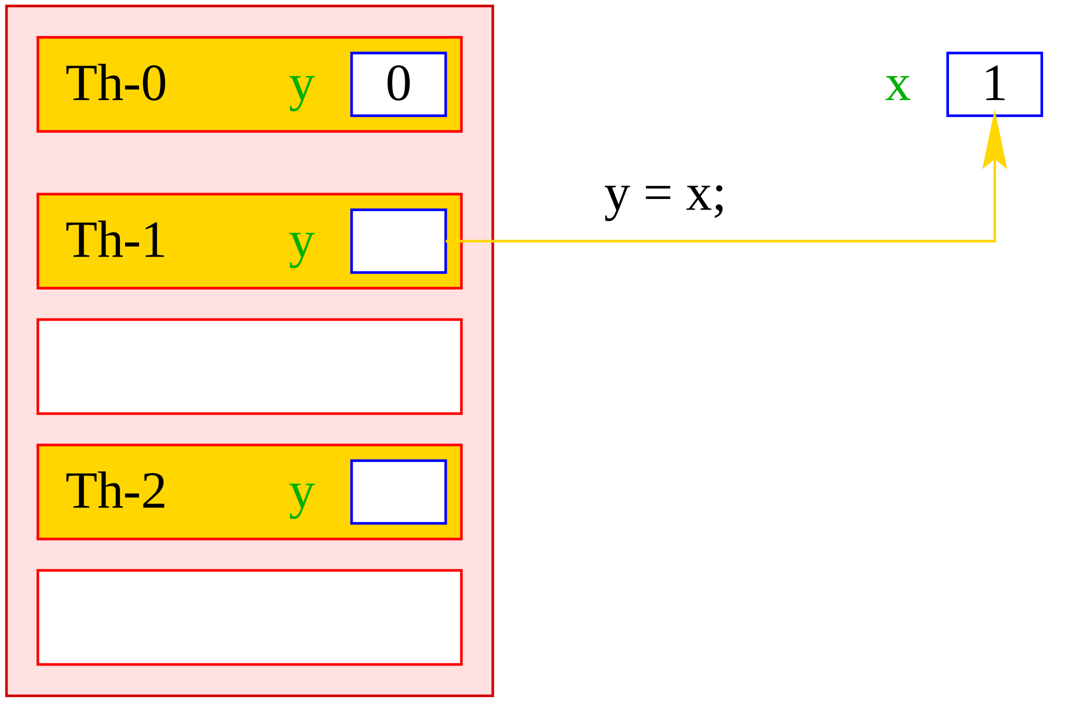
```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

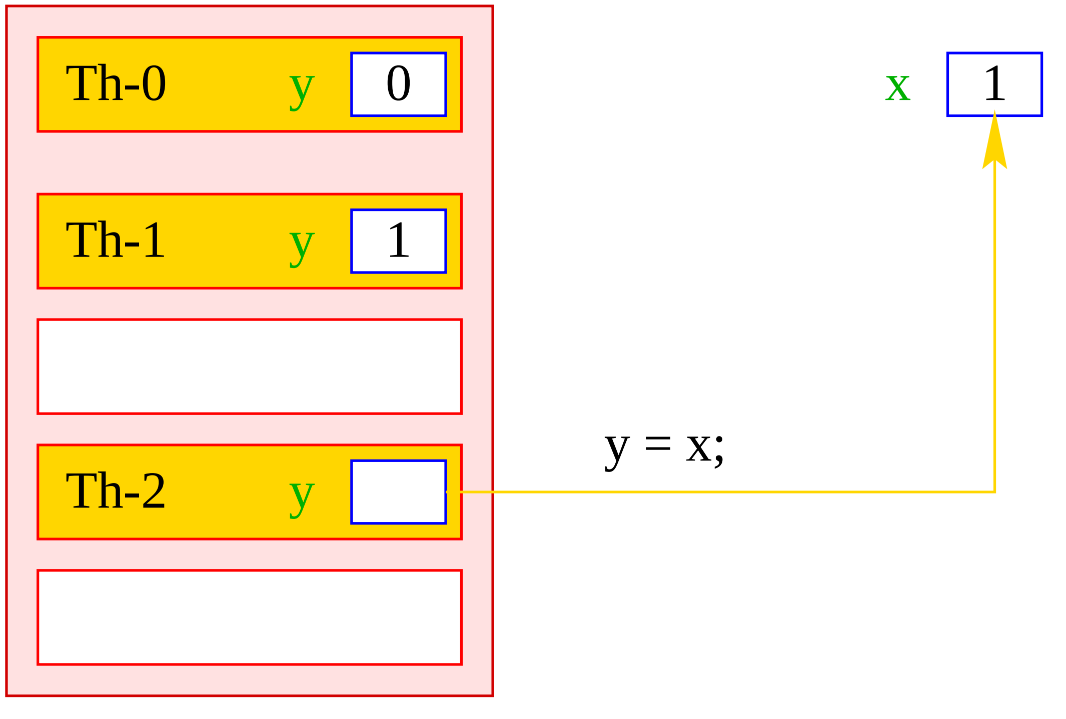
Der Grund:


Th-0
y
Th-1
y
Th-2
y
x
0

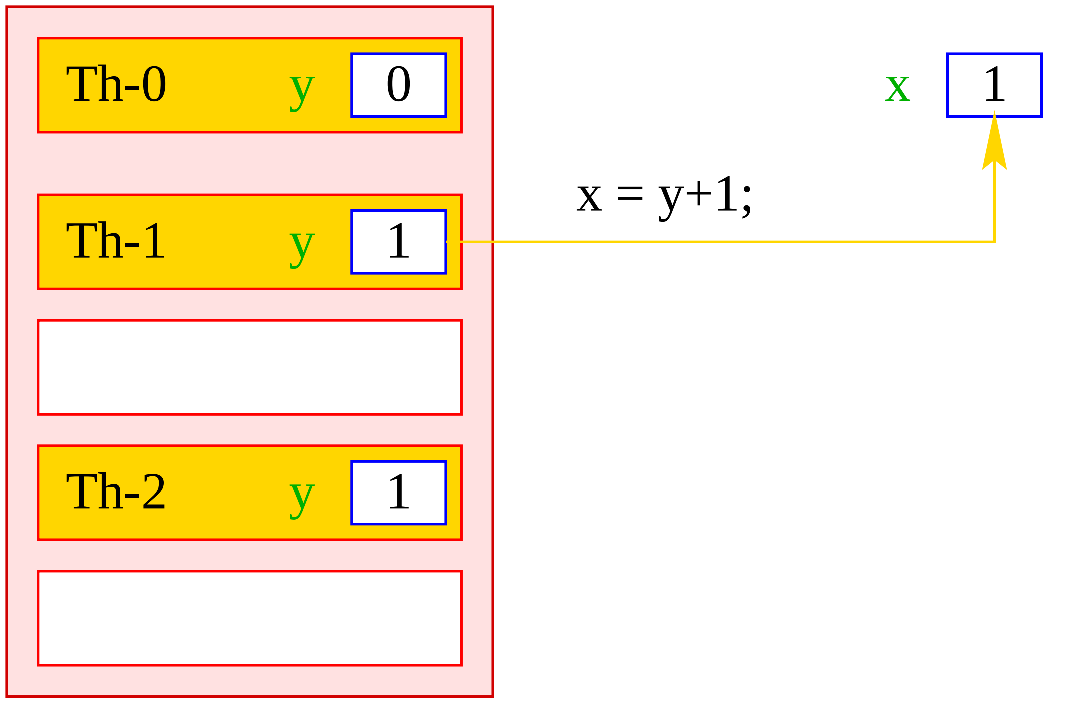

y = x;
Th-0
y
x
0
Th-1
y
Th-2
y

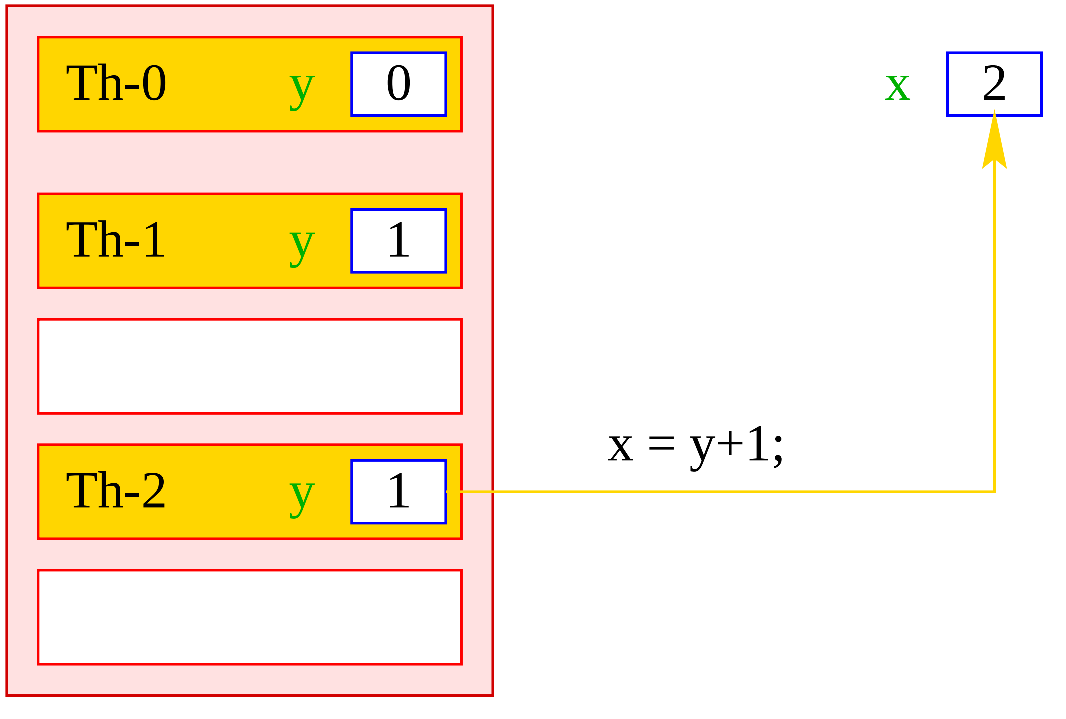

x = y+1;
Th-0
y
0
x
0
Th-1
y
Th-2
y

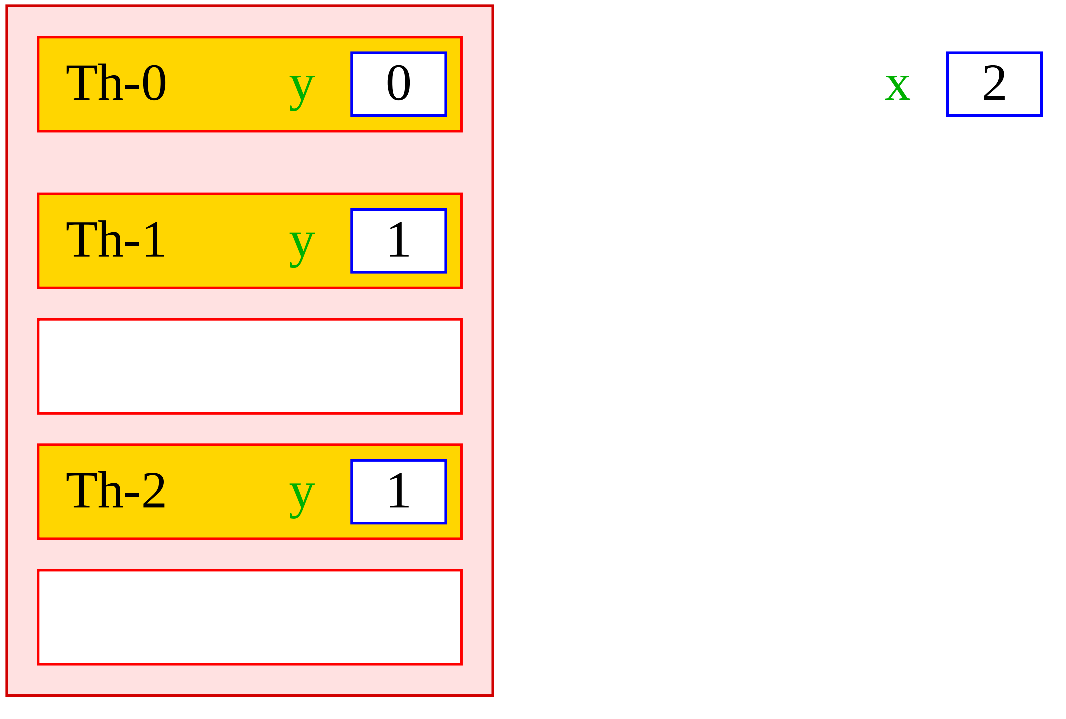

Th-0
y
0
x
1
y = x;
Th-1
y
Th-2
y



Th-0
y
0
x
1
Th-1
y
1
y = x;
Th-2
y

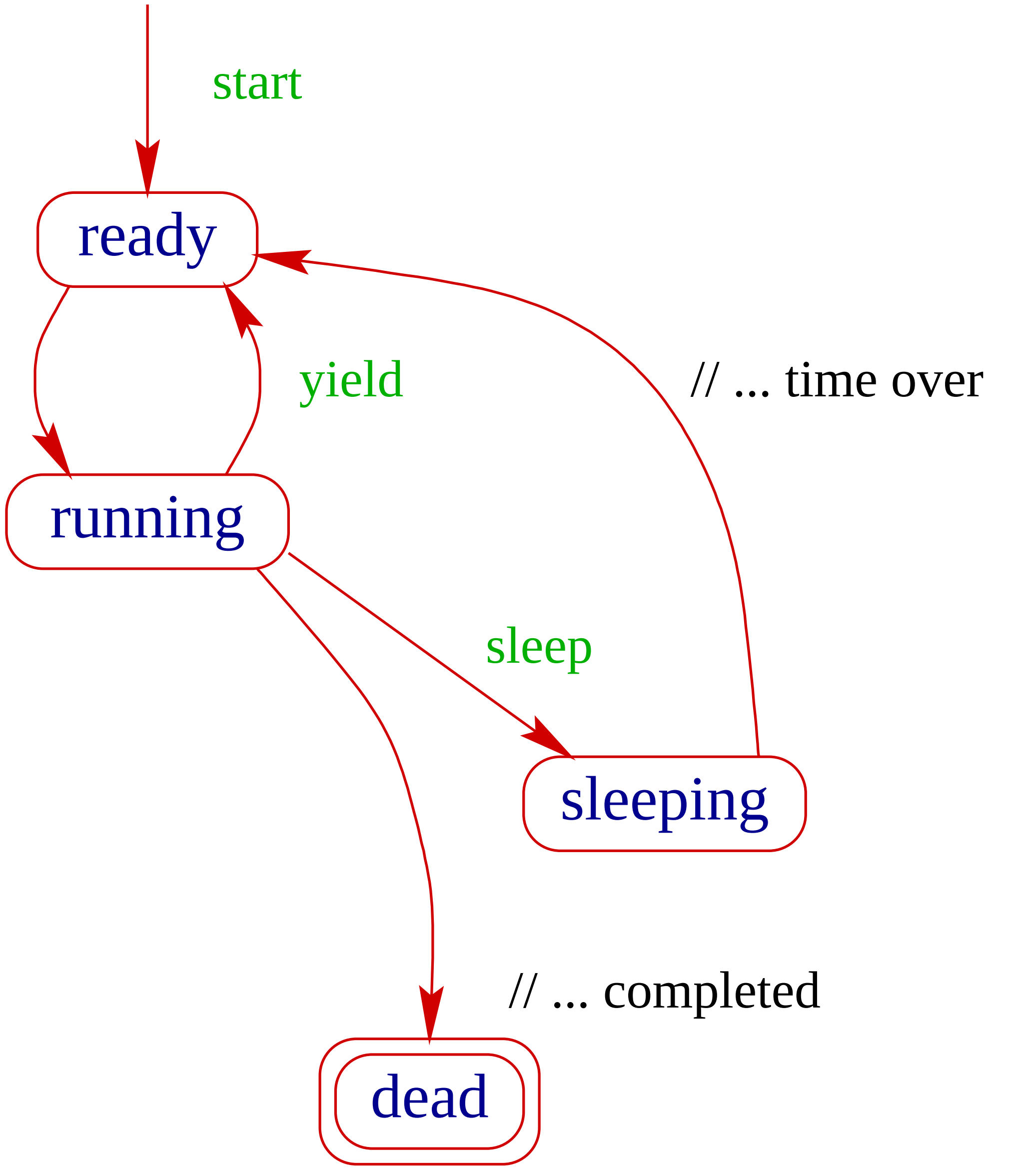

Th-0
y
0
x
1
x = y+1;
Th-1
y
1
Th-2
y
1

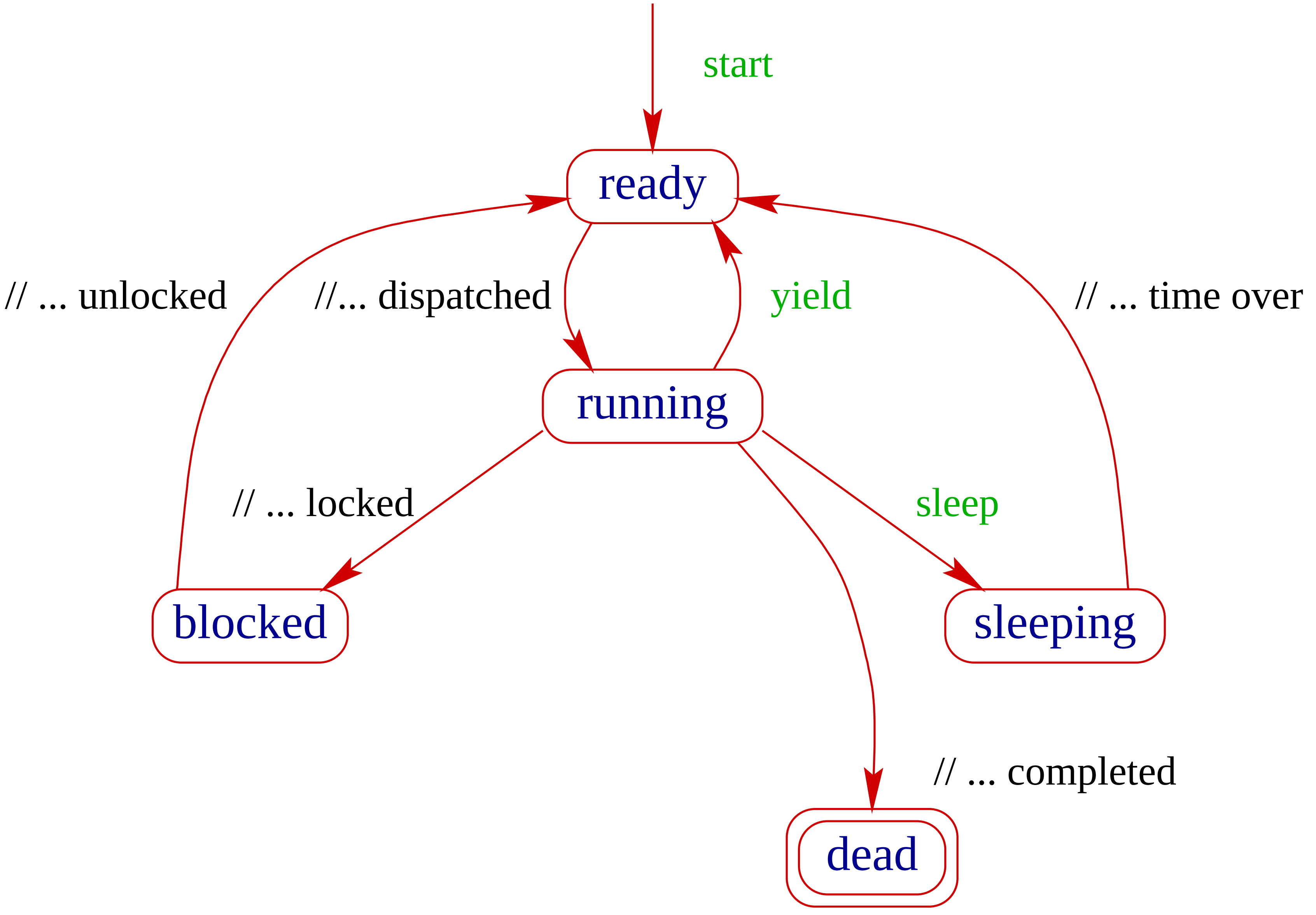

Th-0
y
0
Th-1
y
1
Th-2
y
1
x
2
x = y+1;


Th-0
y
0
Th-1
y
1
Th-2
y
1
x
2

Idee:

- Inkrementieren der Variable `x` sollte ein atomarer Schritt sein, d.h. nicht von parallel laufenden Threads unterbrochen werden können.
- Mithilfe des Schlüsselworts `synchronized` kennzeichnen wir Objekt-Methoden einer Klasse L als ununterbrechbar.
- Für jedes Objekt obj der Klasse L kann zu jedem Zeitpunkt nur ein Aufruf obj`.synchMeth(...)` einer `synchronized`-Methode `synchMeth()` ausgeführt werden. Die Ausführung einer solchen Methode nennt man kritischen Abschnitt ("critical section") für die gemeinsame Resource obj.
- Wollen mehrere Threads gleichzeitig in ihren kritischen Abschnitt für das Objekt obj eintreten, werden alle bis auf einen blockiert.


start
ready
yield
// ... time over
running
sleep
sleeping
// ... completed
dead


start
ready
// ... unlocked
//... dispatched
yield
// ... time over
running
// ... locked
sleep
blocked
sleeping
// ... completed
dead

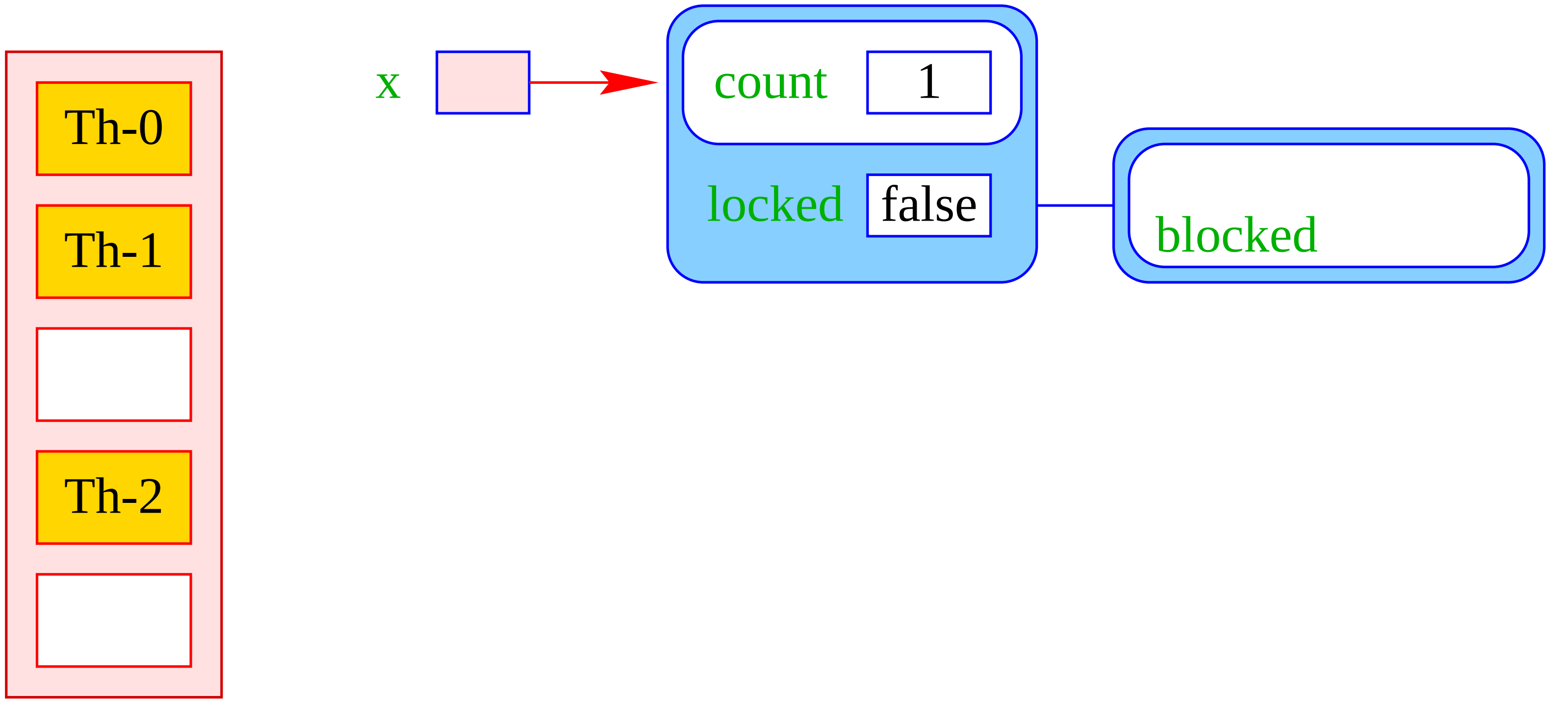
- Jedes Objekt obj mit `synchronized`-Methoden verfügt über:
  1. über ein boolesches Flag `boolean locked;` sowie
  2. über eine Warteschlange `ThreadQueue blockedThreads`.
- Vor Betreten seines kritischen Abschnitts führt ein Thread (implizit) die atomare Operation obj`.lock()` aus:

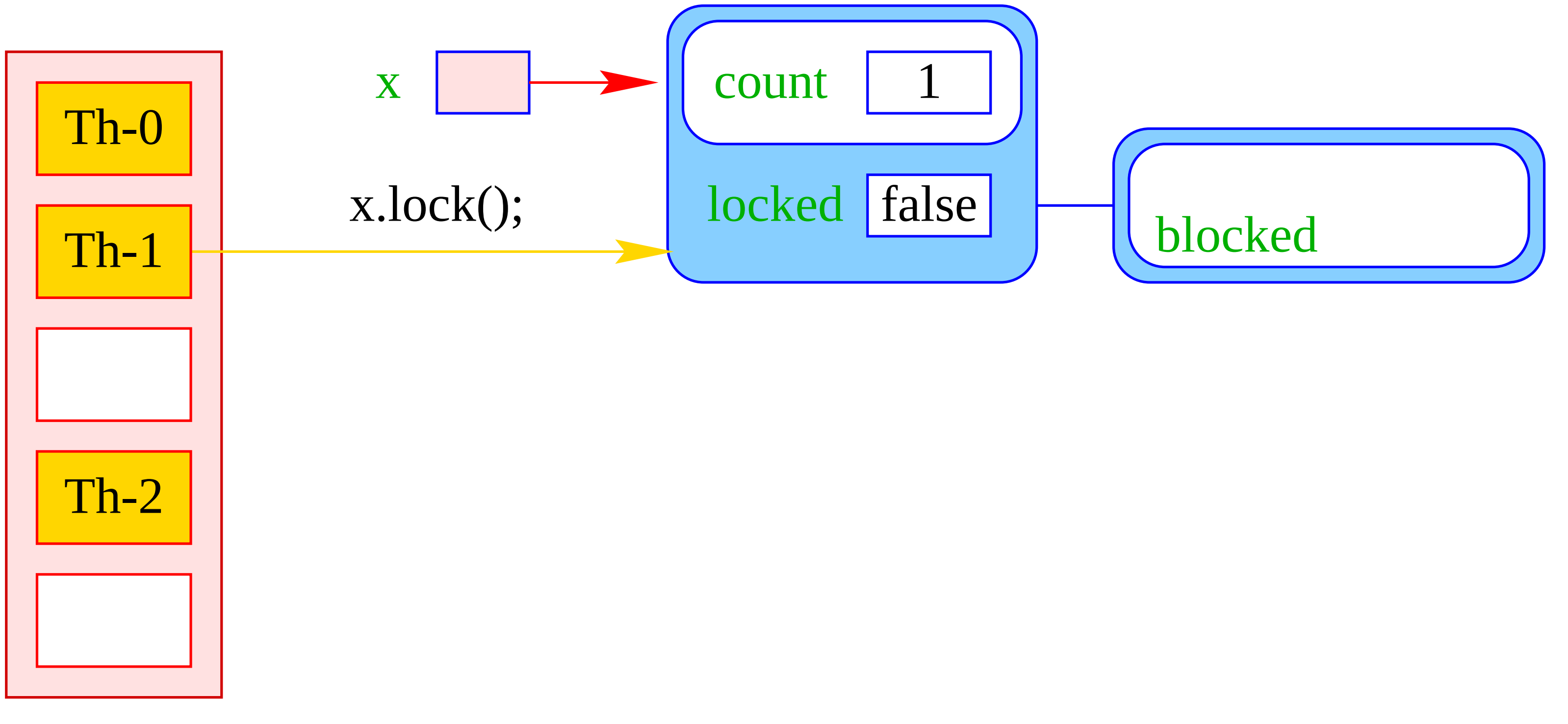
```
private void lock() {
   if (!locked) locked = true; // betrete krit. Abschnitt
   else {                      // Lock bereits vergeben
      Thread t = Thread.currentThread();
      blockedThreads.enqueue(t);
      t.state = blocked;       // blockiere
   }
}  // end of lock()
```

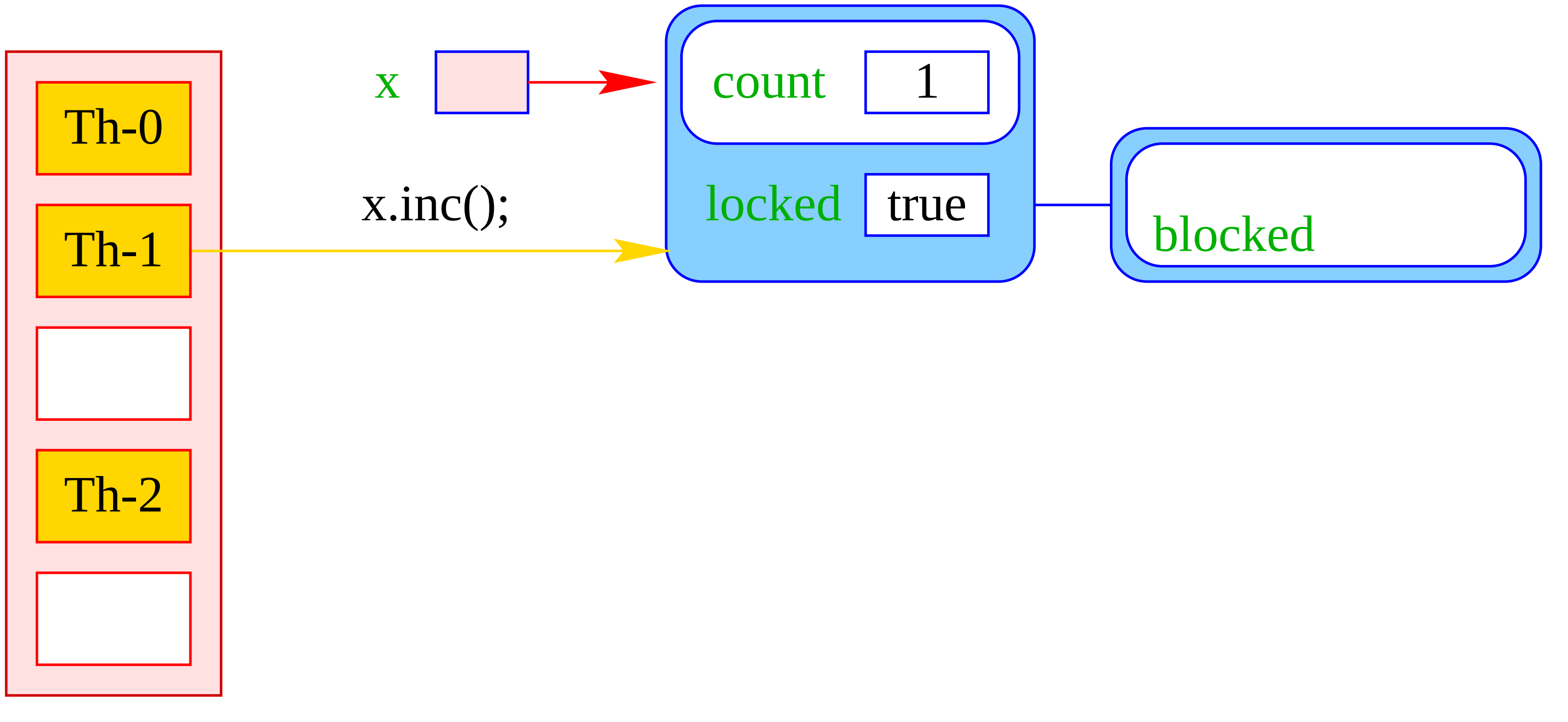
- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception :-), führt er (implizit) die atomare Operation obj`.unlock()` aus:

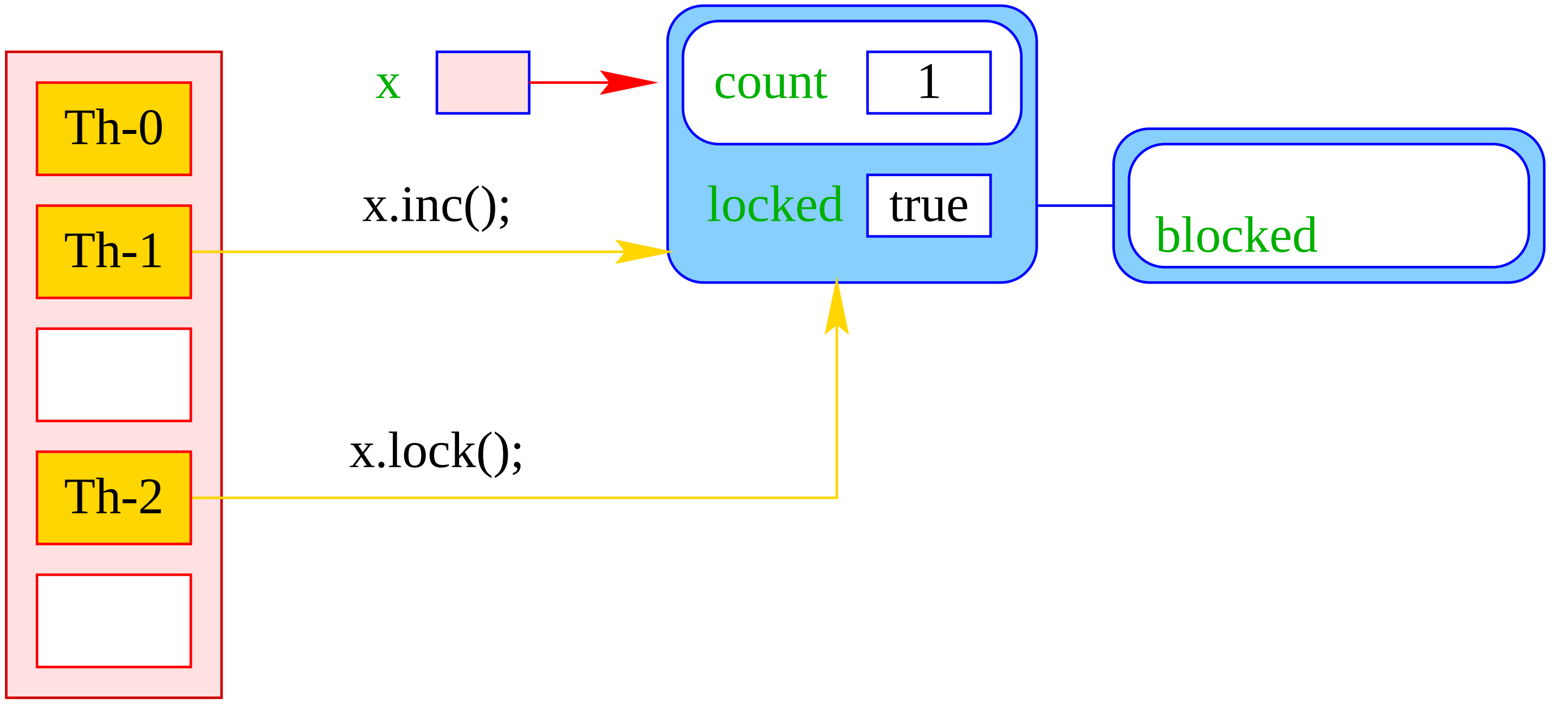
```
private void unlock() {
   if (blockedThreads.empty())
      locked = false; // Lock frei geben
   else {             // Lock weiterreichen
      Thread t = blockedThreads.dequeue();
      t.state = ready;
   }
}  // end of unlock()
```

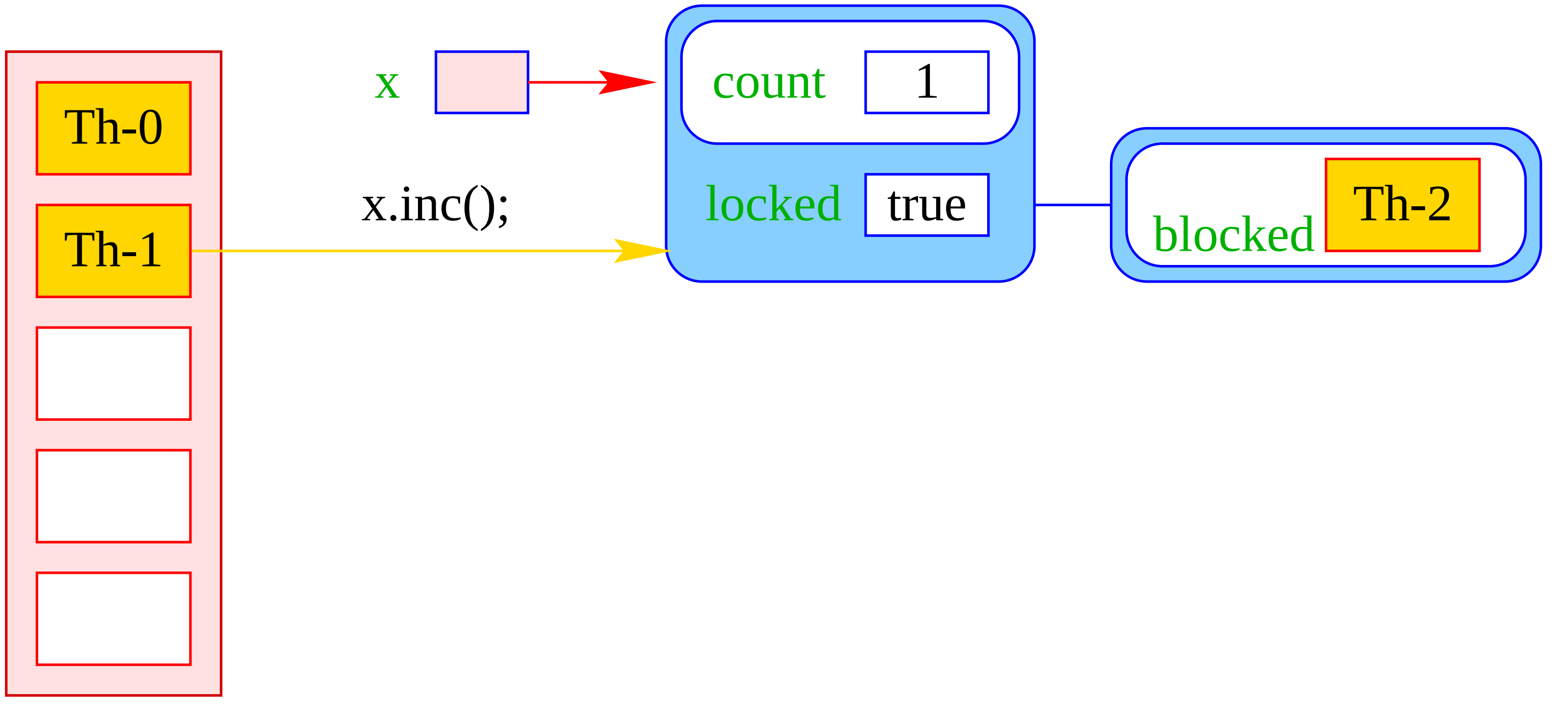
- Dieses Konzept nennt man Monitor.

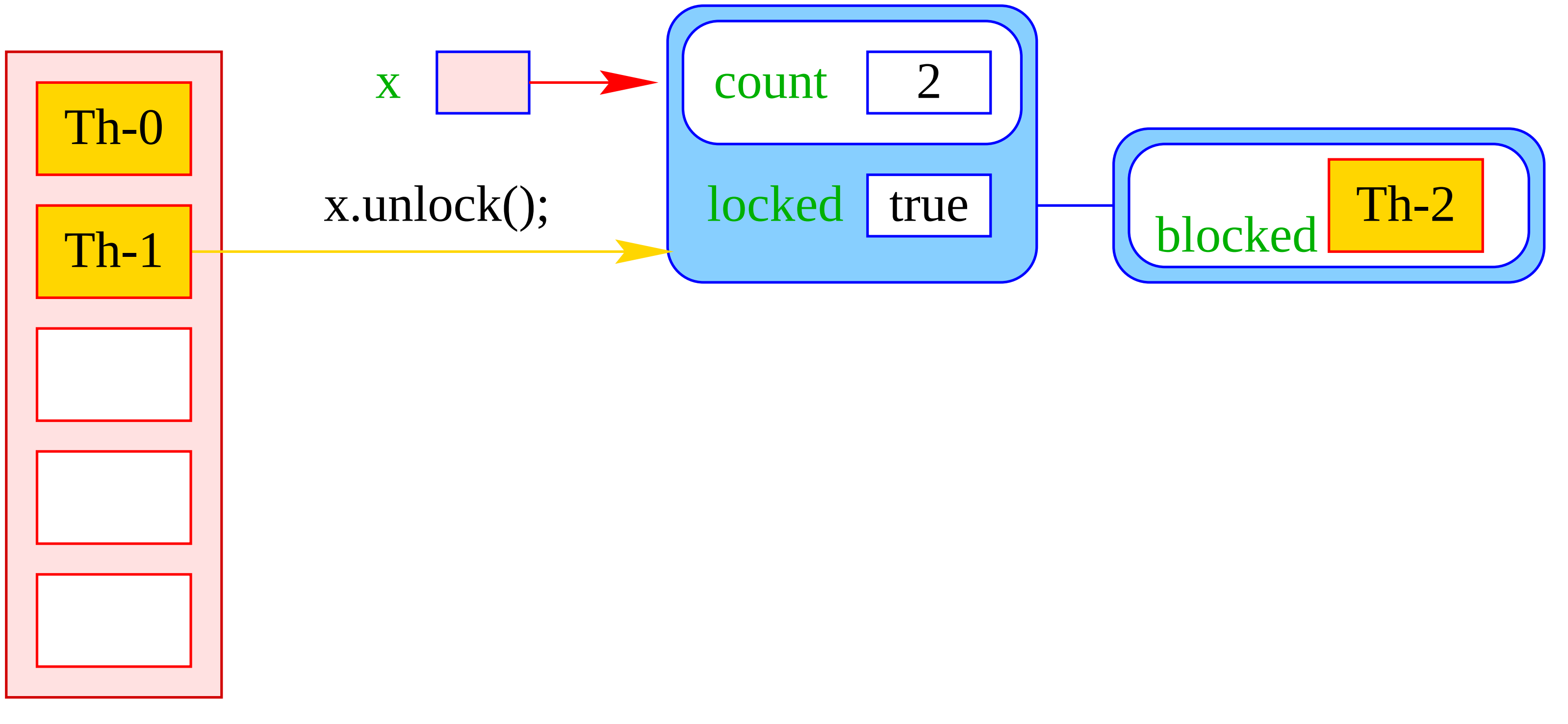

Th-0
Th-1
Th-2
x
count
1
locked
false
blocked

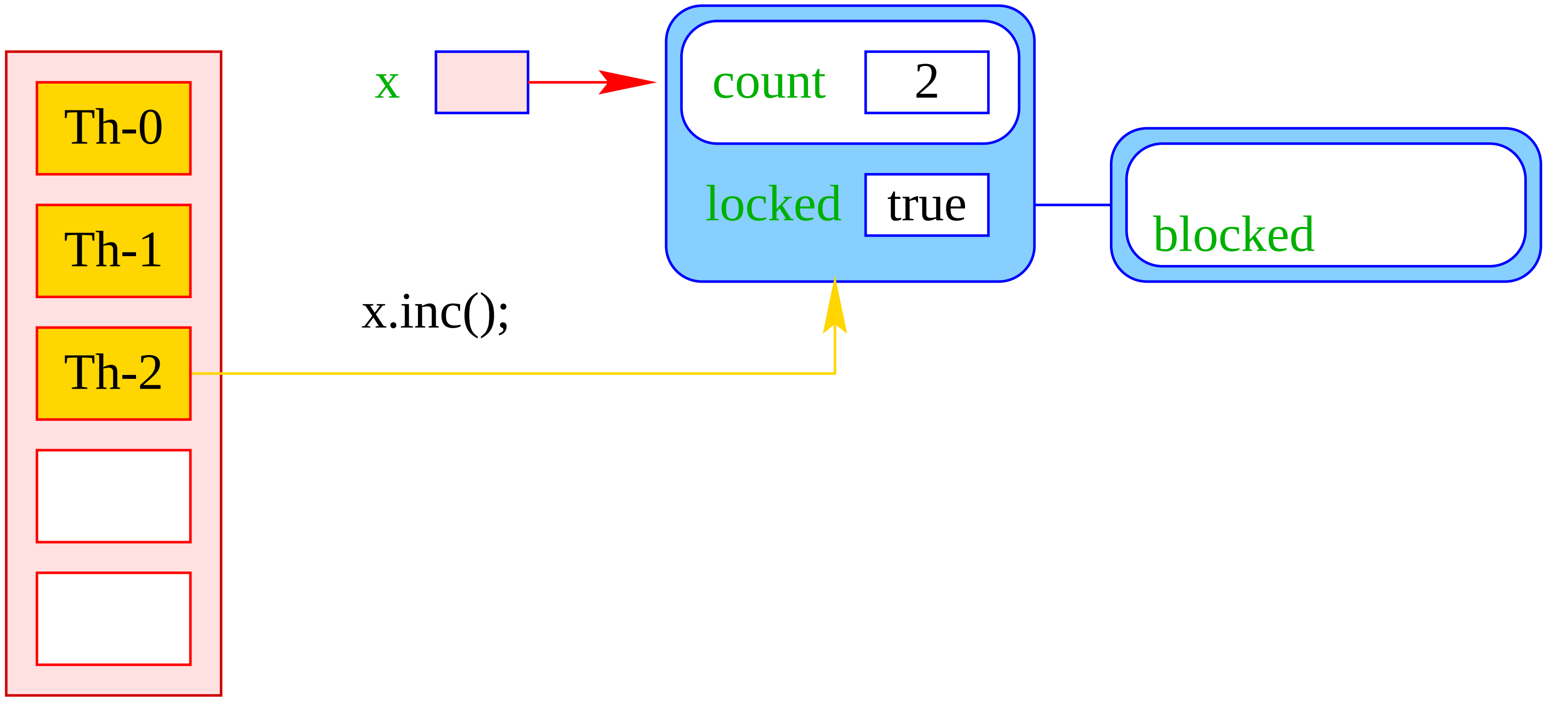

x
count
1
Th-0
x.lock();
locked
false
Th-1
blocked
Th-2


Th-0
Th-1
Th-2
x
x.inc();
count
1
locked
true
blocked

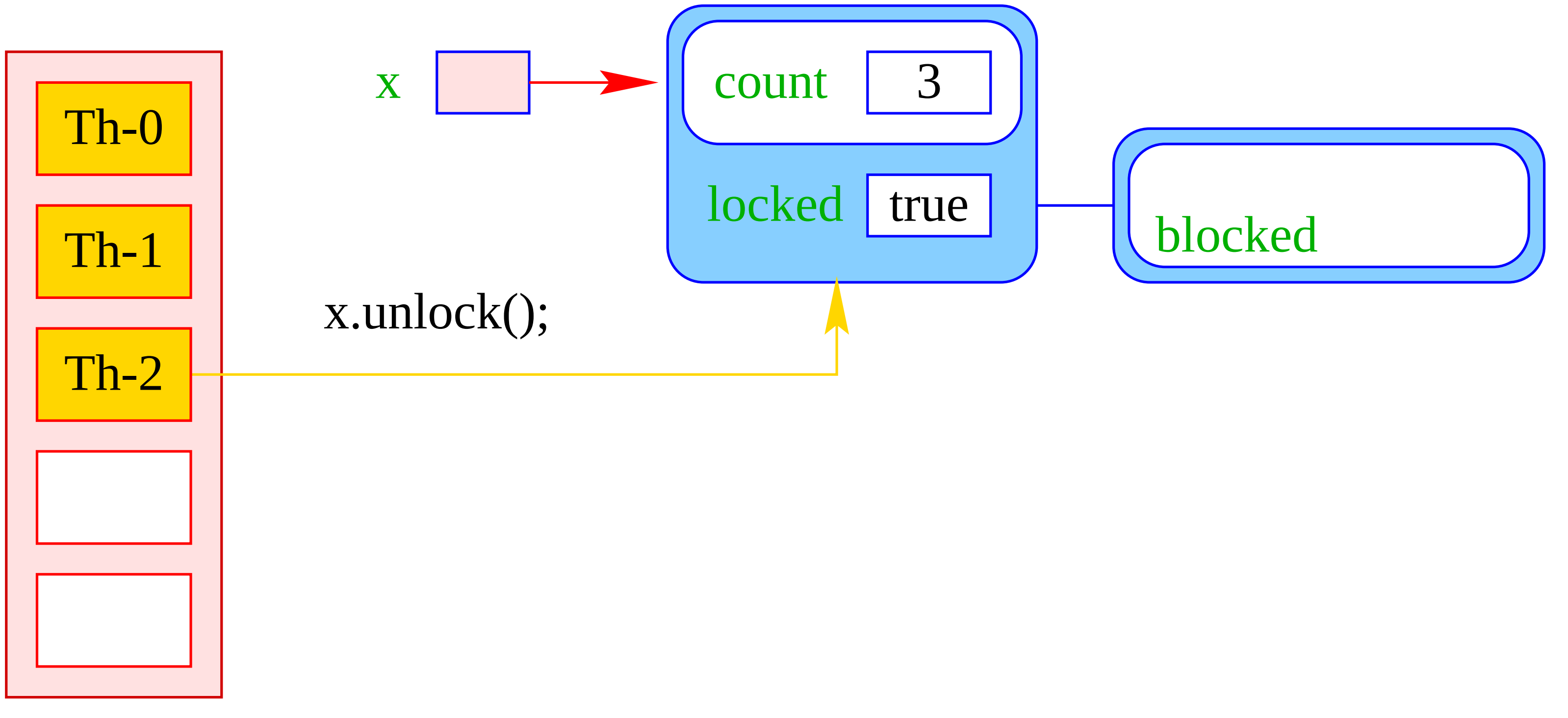

x
count
1
locked
true
blocked
Th-0
x.inc();
Th-1
x.lock();
Th-2

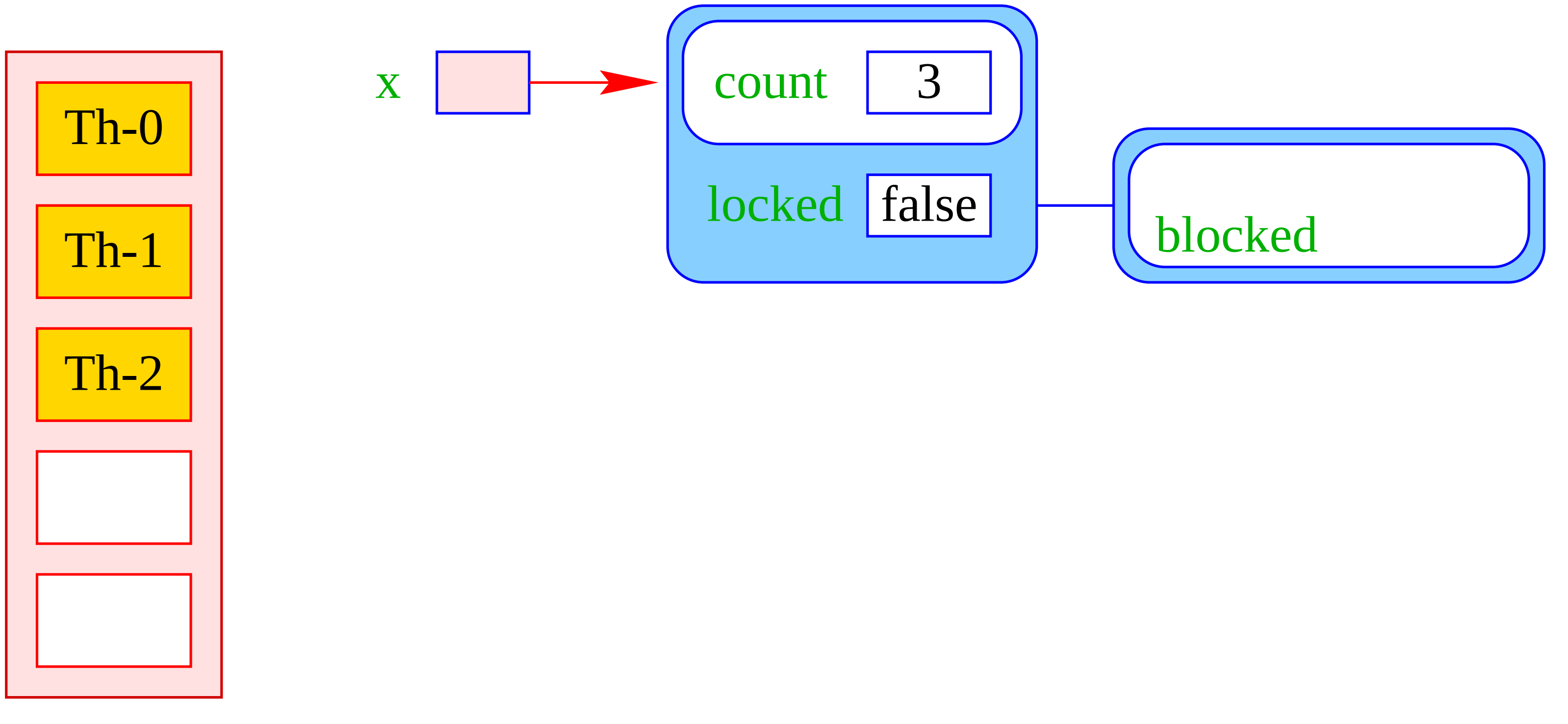

Th-0
Th-1
x
x.inc();
count
1
locked
true
blocked
Th-2


Th-0
Th-1
x
x.unlock();
count
2
locked
true
blocked
Th-2



Th-0
Th-1
Th-2
x
count
2
locked
true
blocked
x.inc();


x
count
3
locked
true
blocked
Th-0
Th-1
Th-2
x.unlock();


Th-0
Th-1
Th-2
x
count
3
locked
false
blocked

```
public class Count {
   private int x = 0;
   public synchronized void inc() {
      String s = Thread.currentThread().getName();
      int y = x;    System.out.println(s+ " read "+y);
      x = y+1;      System.out.println(s+ " wrote "+(y+1));
   }
}  // end of class Count
public class IncSync implements Runnable {
   private static Count x = new Count();
   public void run() { x.inc(); }
   public static void main(String[] args) {
      (new Thread(new IncSynch())).start();
      (new Thread(new IncSynch())).start();
      (new Thread(new IncSynch())).start();
   }
}  // end of class IncSync
```

... liefert:

```
> java IncSync
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-1 wrote 2
Thread-2 read 2
Thread-2 wrote 3
```

Achtung:

- Die Operationen `lock()` und `unlock()` erfolgen nur, wenn der Thread nicht bereits vorher das Lock des Objekts erworben hat.
- Ein Thread, der das Lock eines Objekts obj besitzt, kann weitere Methoden für obj aufrufen, ohne sich selbst zu blockieren :-)

- Um das zu garantieren, legt ein Thread für jedes Objekt obj, dessen Lock er nicht besitzt, aber erwerben will, einen neuen Zähler an:

```
int countLock[obj] = 0;
```

- Bei jedem Aufruf einer `synchronized`-Methode `m(...)` für obj wird der Zähler inkrementiert, für jedes Verlassen (auch mittels Exceptions :-) dekrementiert:

```
if (0 == countLock[obj]++) lock();
Ausführung von obj.m(...)
if (--countLock[obj] == 0) unlock();
```

- `lock()` und `unlock()` werden nur ausgeführt, wenn

```
(countLock[obj] == 0)
```

Andere Gründe für Blockierung:

- Warten auf Beendigung einer IO-Operation;
- `public final void join() throws InterruptedException` (eine Objekt-Methode der Klasse `Thread`) wartet auf die Beendigung eines anderen Threads...

... ein Beispiel:

```
public class Join implements Runnable {
   private static int count = 0;
   private int n = count++;
   private static Thread[] task = new Thread[3];
   public void run() {
      try {
         if (n>0) {
            task[n-1].join();
            System.out.println("Thread-"+n+" joined Thread-"+(n-1));
         }
      } catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   }
   ...
```

```
  ...
  public static void main(String[] args) {
     for(int i=0; i<3; i++)
        task[i] = new Thread(new Join());
     for(int i=0; i<3; i++)
        task[i].start();
  }
} // end of class Join
```

... liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```

Beachte:

- Threads, die auf Beendigung eines anderen Threads warten, gehen in einen Zustand joining über.
- Threads, die auf Beendigung einer IO-Operation warten, gehen in einen Zustand joiningIO über.
- Diese Zustände ähneln dem Zustand blocked für wechselseitigen Ausschluss von kritischen Abschnitten. Insbesondere gibt es
- ... für jeden Thread t eine Schlange `ThreadQueue joiningThreads;`
- ... analoge Warteschlangen für verschiedene IO-Operationen.

Beachte:

- Threads, die auf Beendigung eines anderen Threads warten, gehen in einen Zustand joining über.
- Threads, die auf Beendigung einer IO-Operation warten, gehen in einen Zustand joiningIO über.
- Diese Zustände ähneln dem Zustand blocked für wechselseitigen Ausschluss von kritischen Abschnitten. Insbesondere gibt es
- ... für jeden Thread t eine Schlange `ThreadQueue joiningThreads;`
- ... analoge Warteschlangen für verschiedene IO-Operationen.

Spaßeshalber betrachten wir noch eine kleine Variation des letzten Programms:

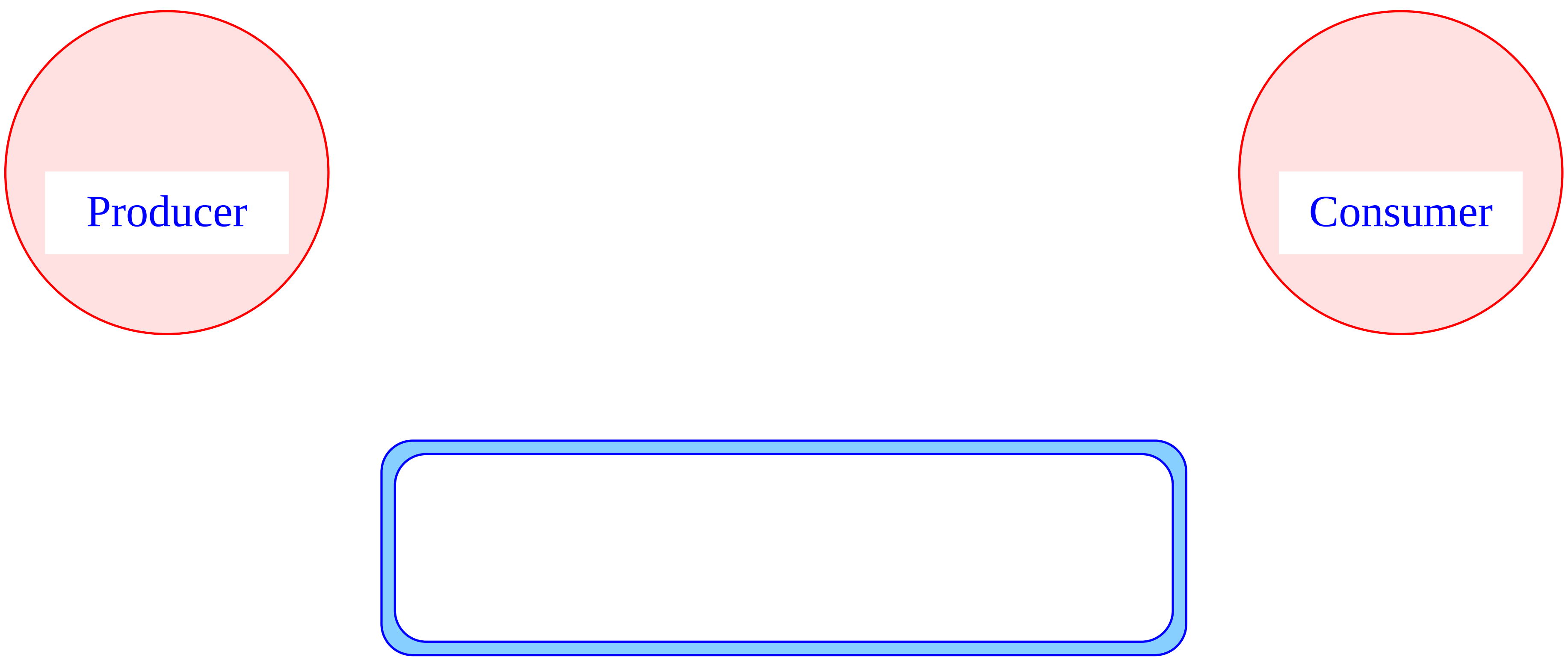
```
public class CW implements Runnable {
   private static int count = 0;
   private int n = count++;
   private static Thread[] task = new Thread[3];
   public void run() {
      try { task[(n+1)%3].join(); }
      catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   }
   public static void main(String[] args) {
      for(int i=0; i<3; i++)
         task[i] = new Thread(new CW());
      for(int i=0; i<3; i++) task[i].start();
   }
}  // end of class CW
```

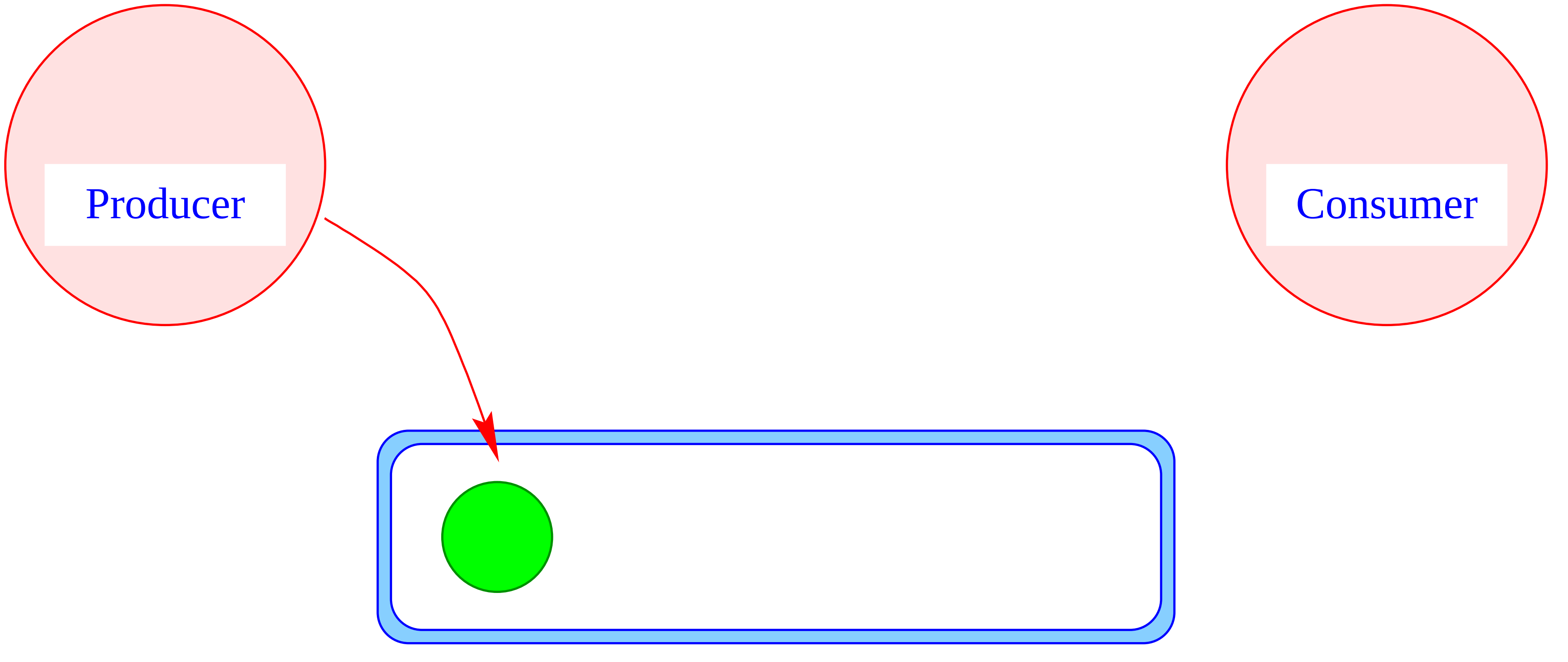
- Jeder Thread geht in einen Wartezustand (hier: joining) über und wartet auf einen anderen Thread.
- Dieses Phänomen heißt auch Circular Wait oder Deadlock – eine unangenehme Situation, die man in seinen Programmen tunlichst vermeiden sollte :-)

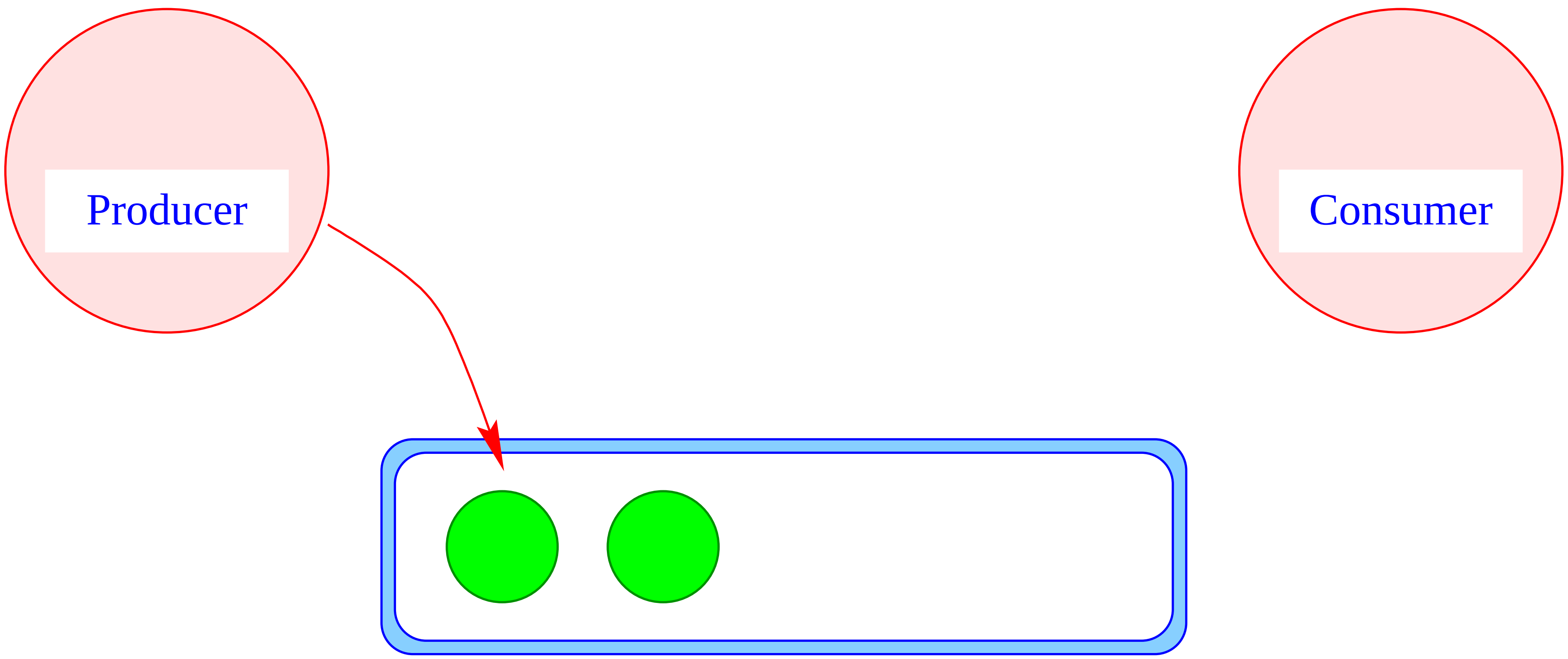
19.2 Semaphore und das Producer-Consumer-Problem

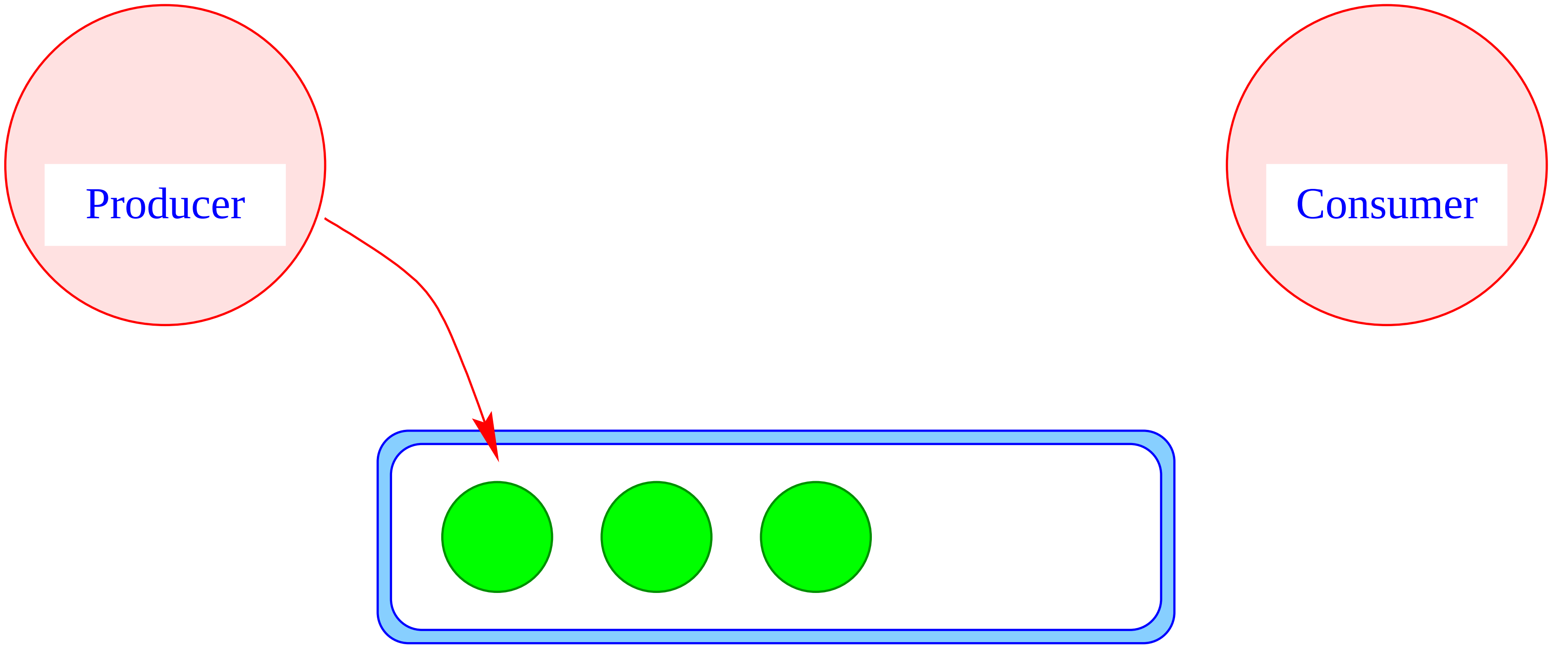
Aufgabe:

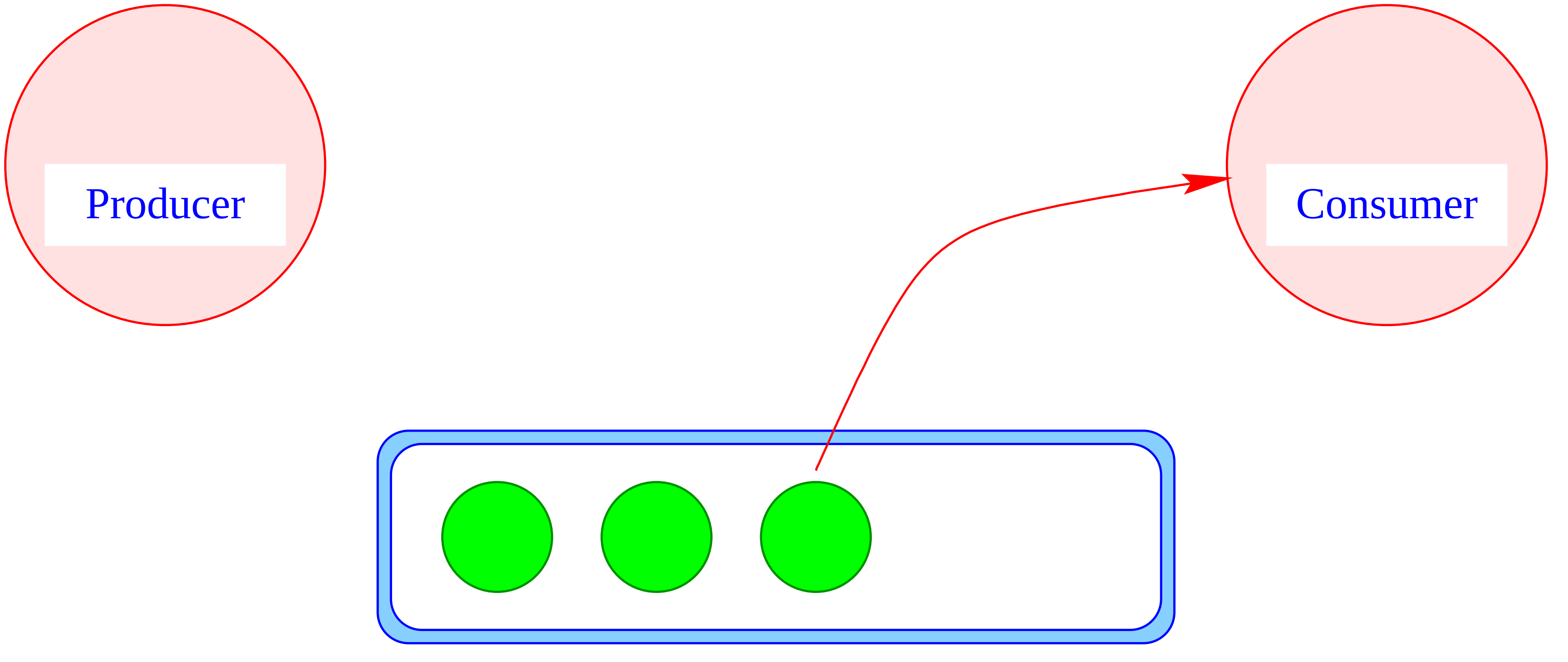
- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur Übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

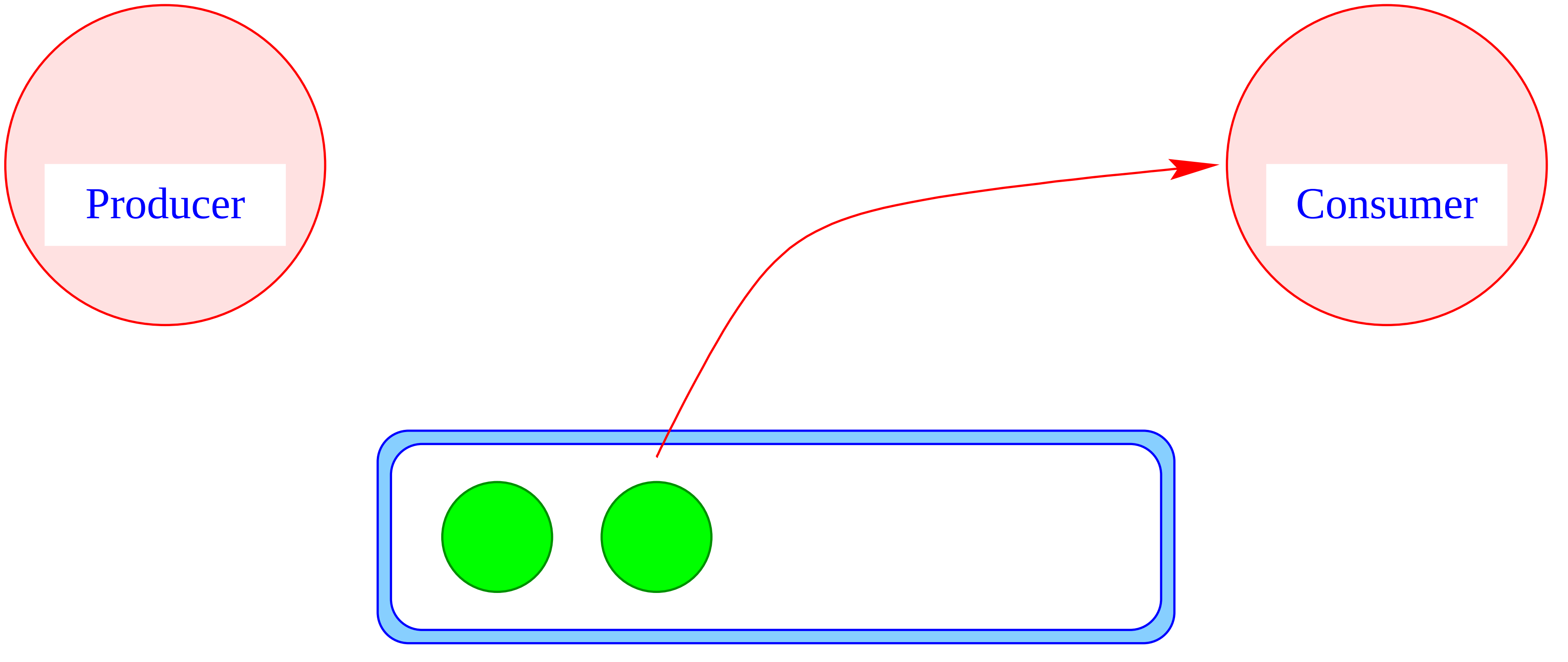

Producer
Consumer


Producer
Consumer


Producer
Consumer


Producer
Consumer


Producer
Consumer


Producer
Consumer

1. Idee:

- Wir definieren eine Klasse `Buffer2`, die (im wesentlichen) aus einem Feld der richtigen Größe, sowie zwei Verweisen `int first, last` zum Einfügen und Entfernen verfügt:

```
public class Buffer2 {
   private int cap, free, first, last;
   private Data[] a;
   public Buffer2(int n) {
      free = cap = n; first = last = 0;
      a = new Data[n];
   }
   ...
```

- Einfügen und Entnehmen sollen synchrone Operationen sein ...

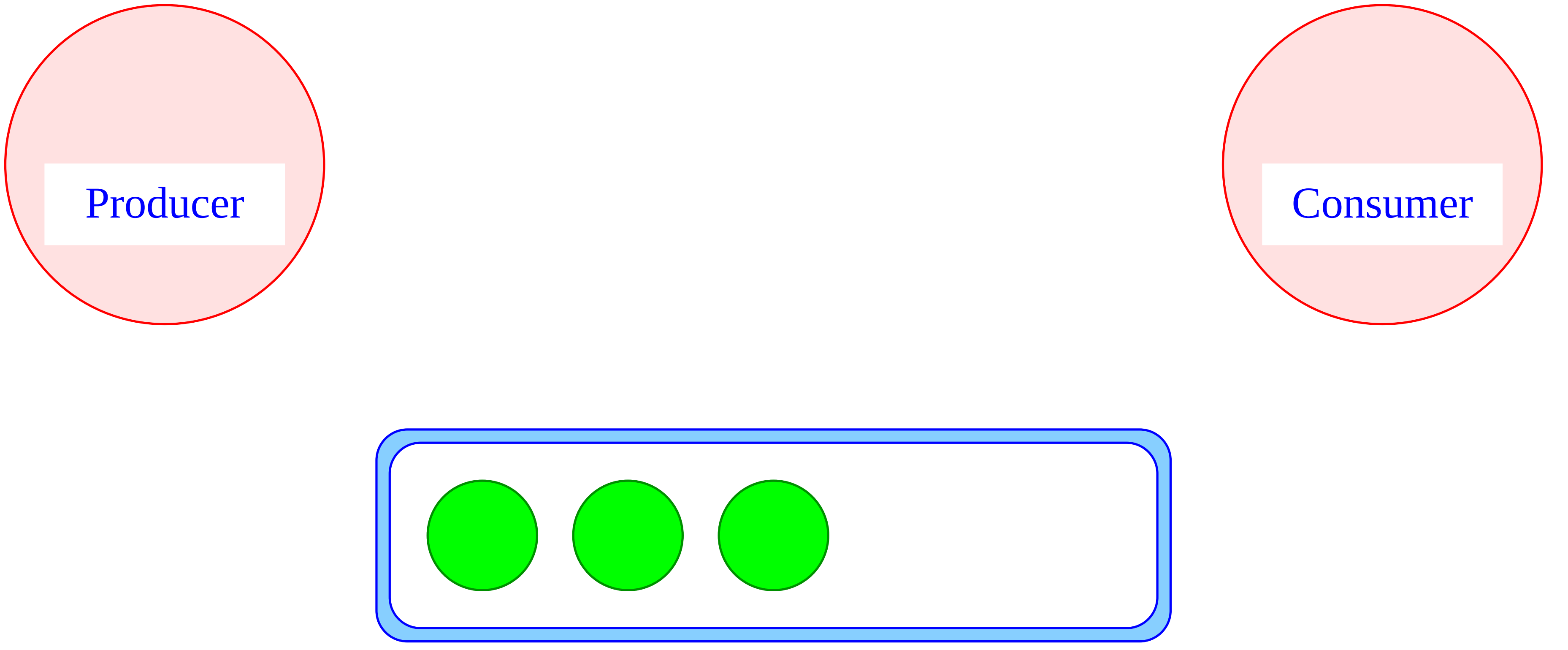
Probleme:

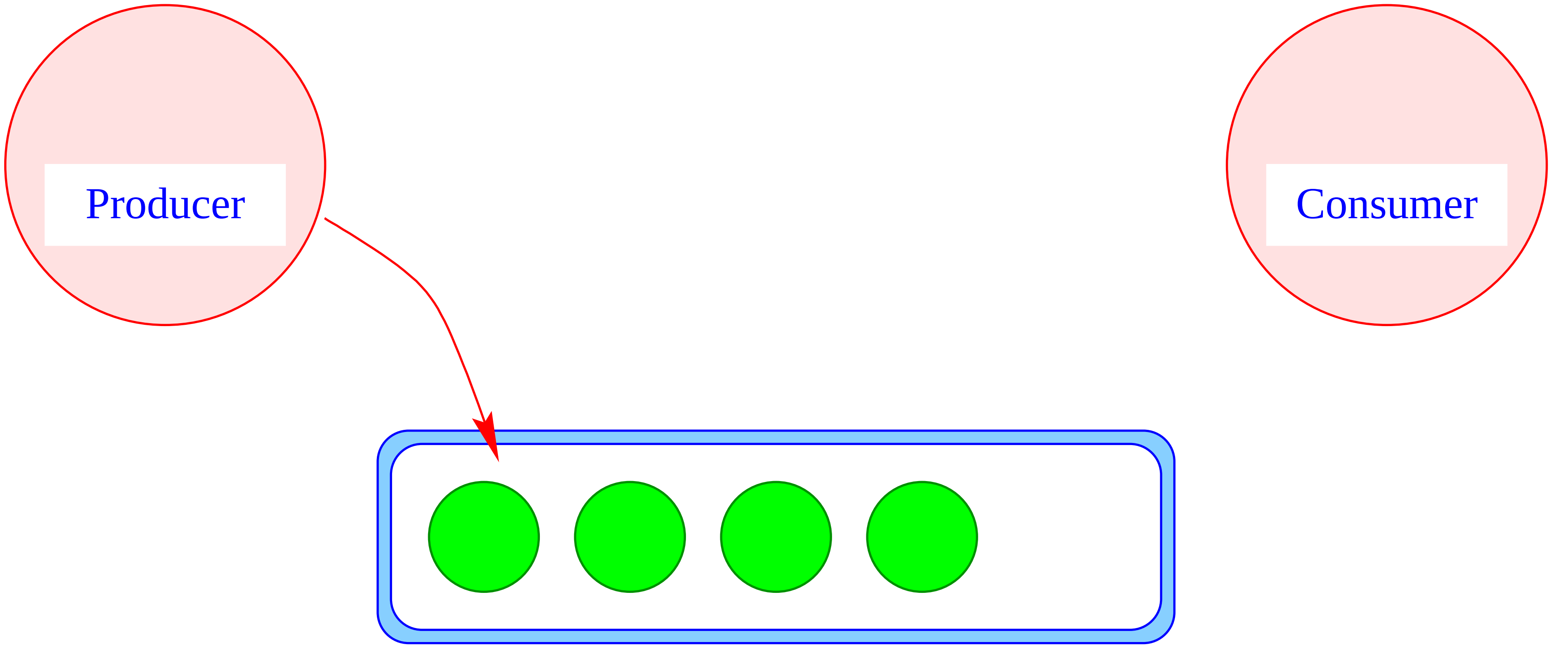
- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

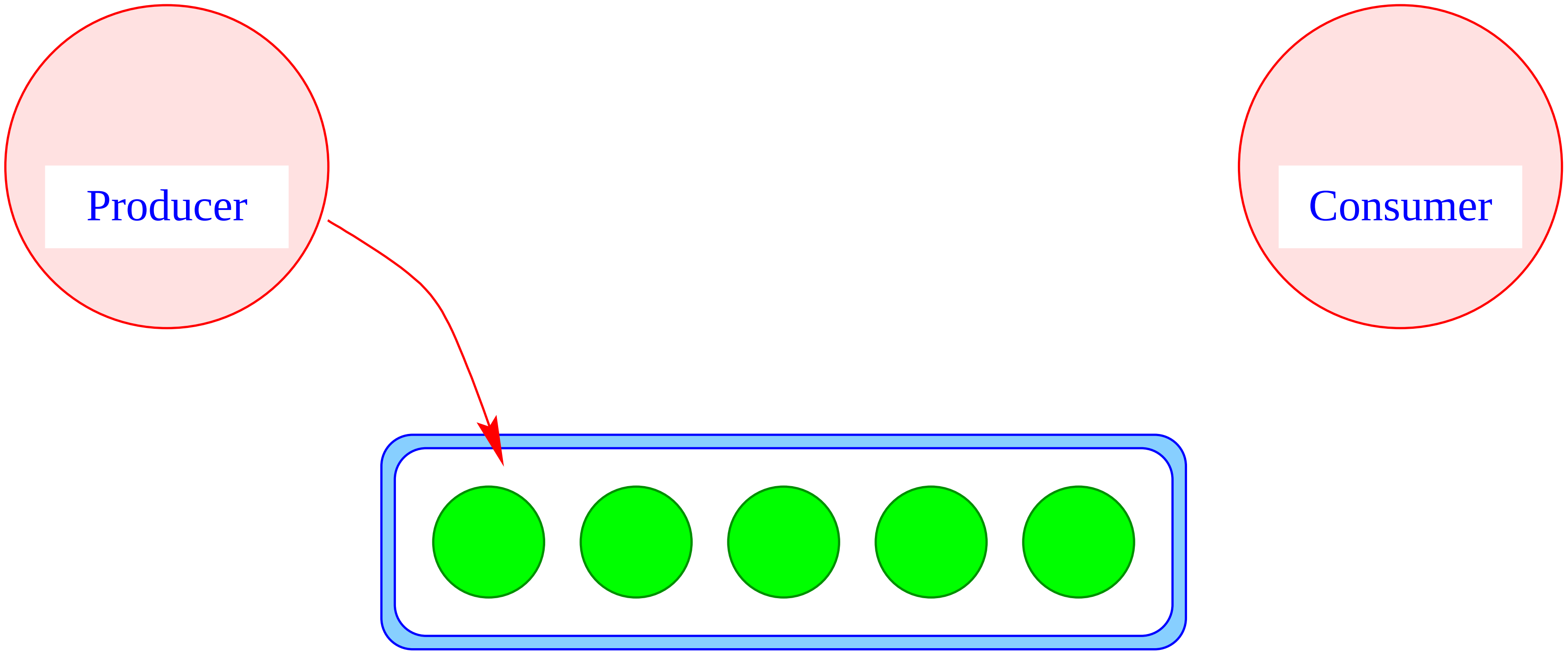
Probleme:

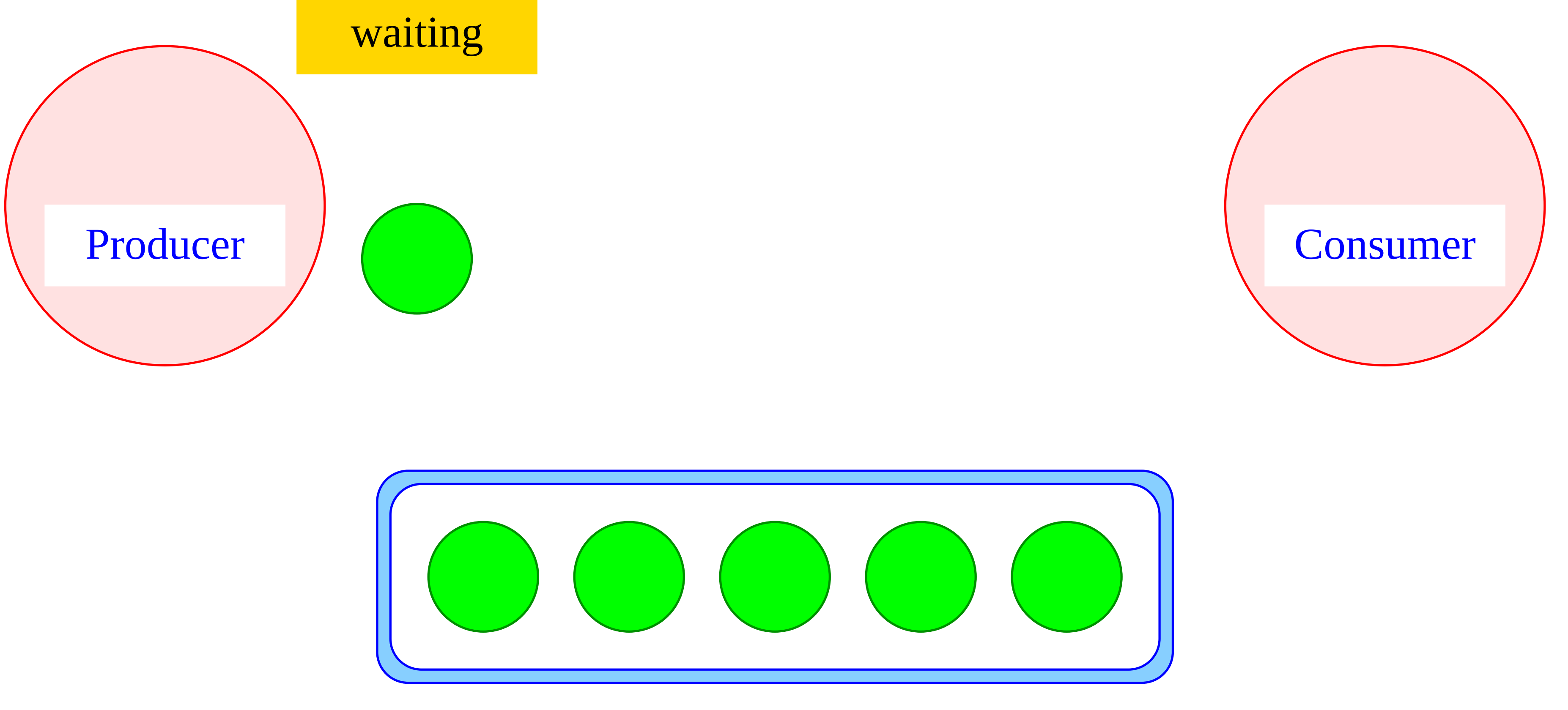
- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

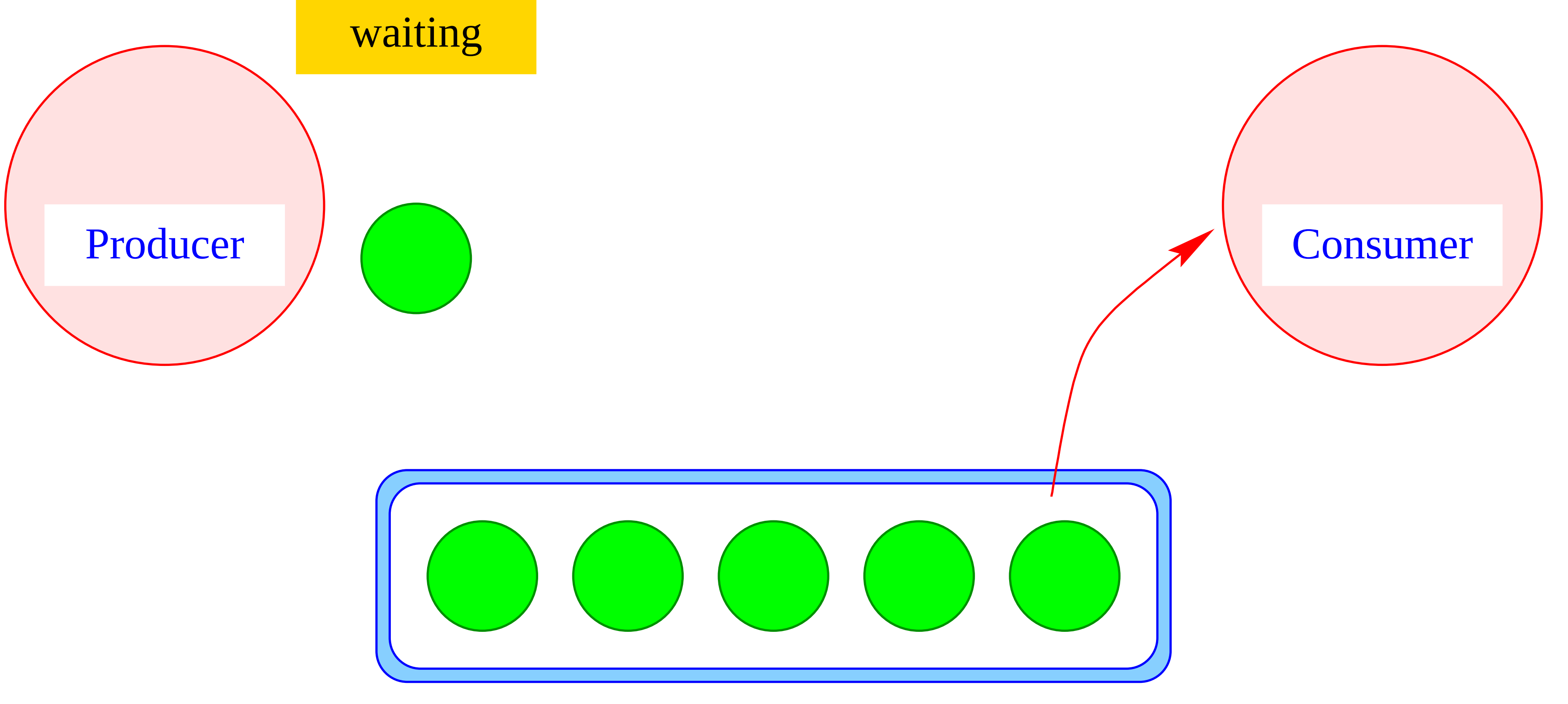
Java's Lösungsvorschlag: Warten ...

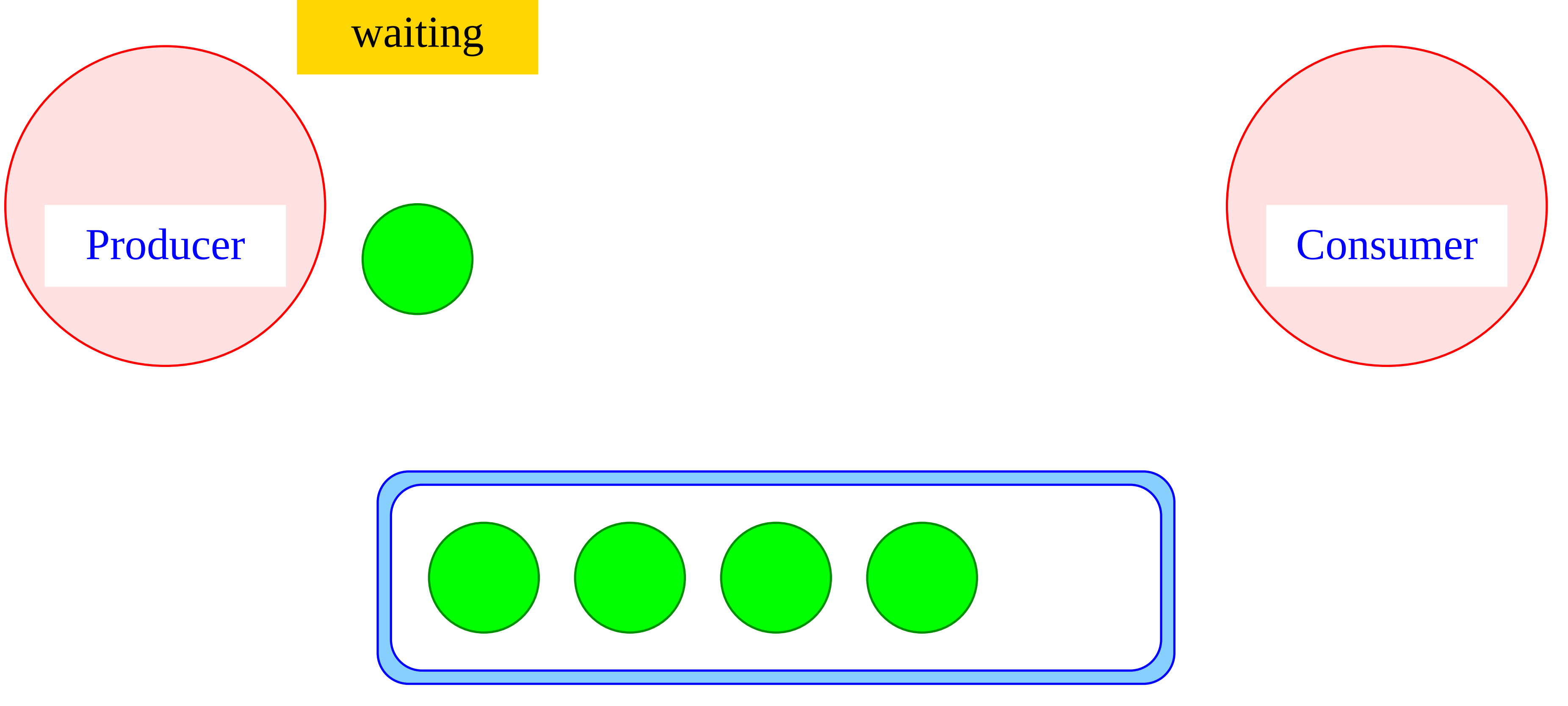

Producer
Consumer

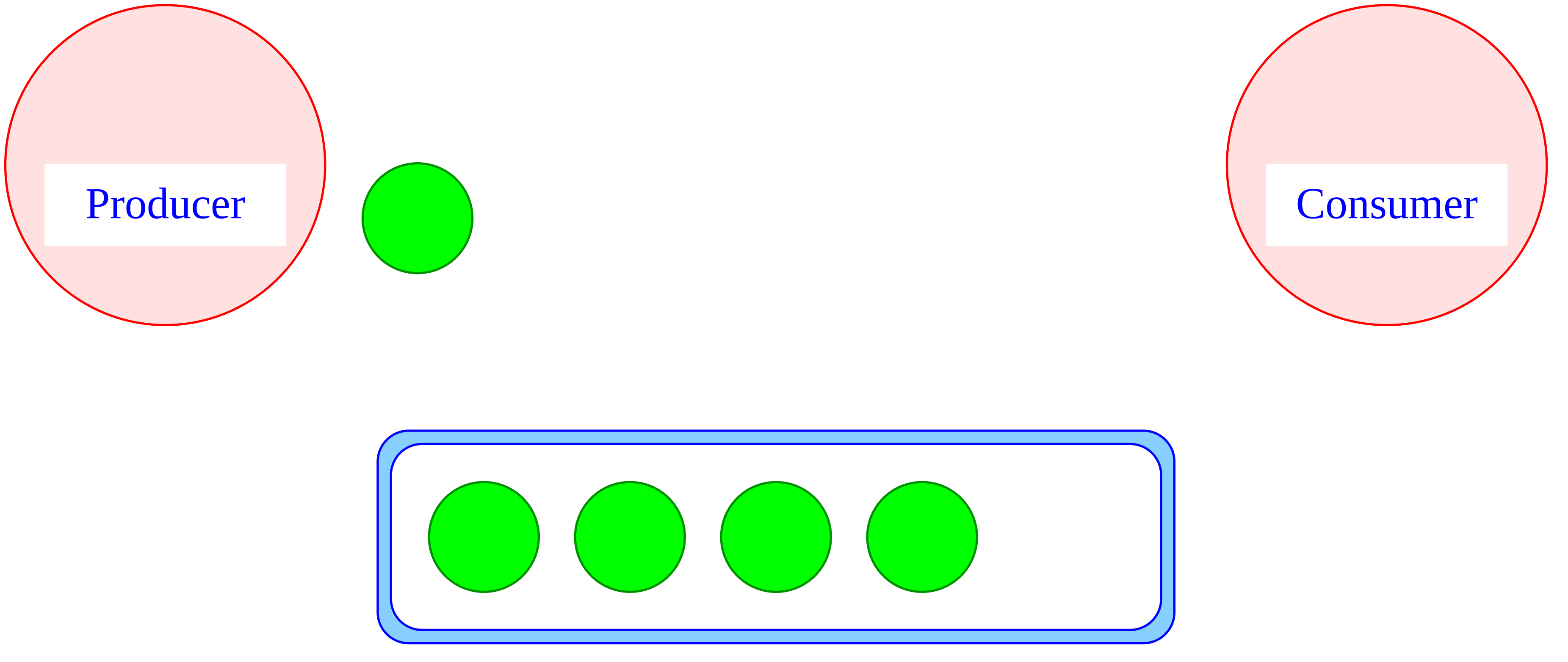

Producer
Consumer

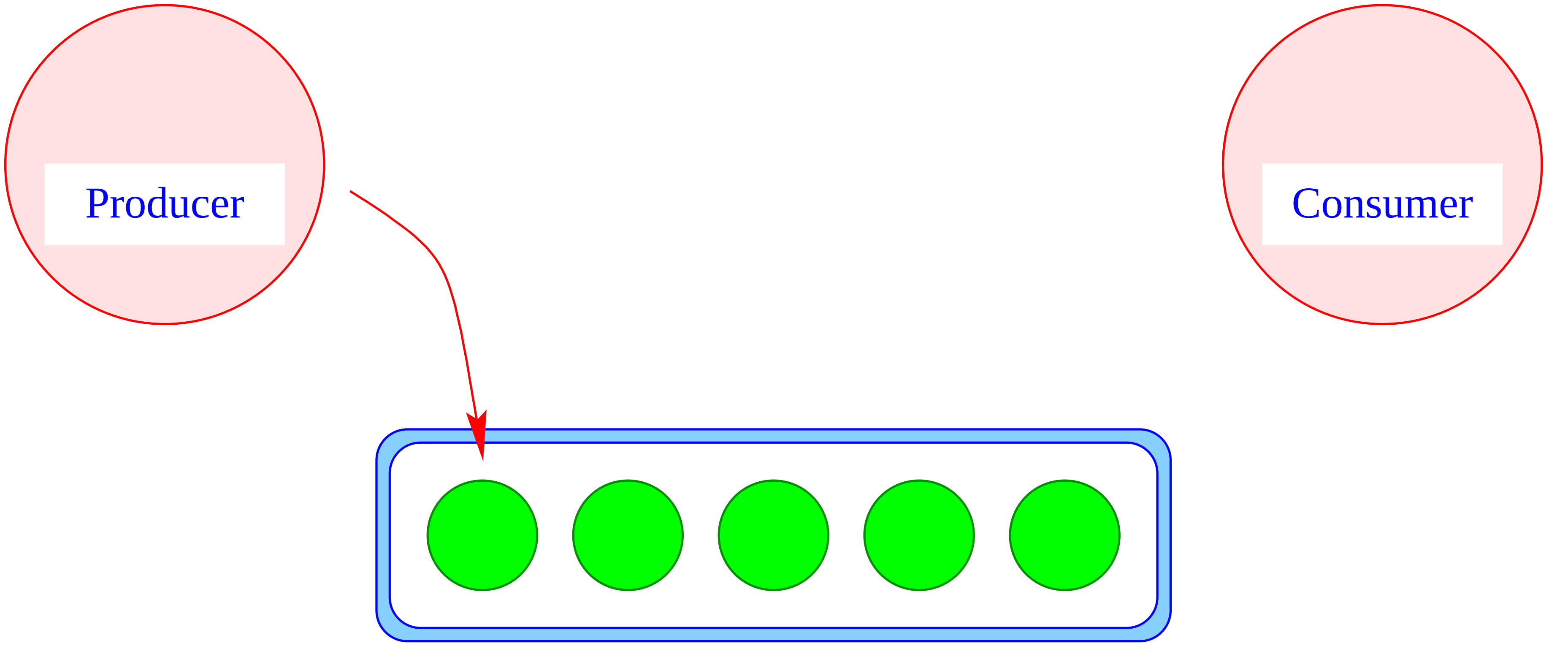

Producer
Consumer


waiting
Producer
Consumer


waiting
Producer
Consumer


waiting
Producer
Consumer


Producer
Consumer

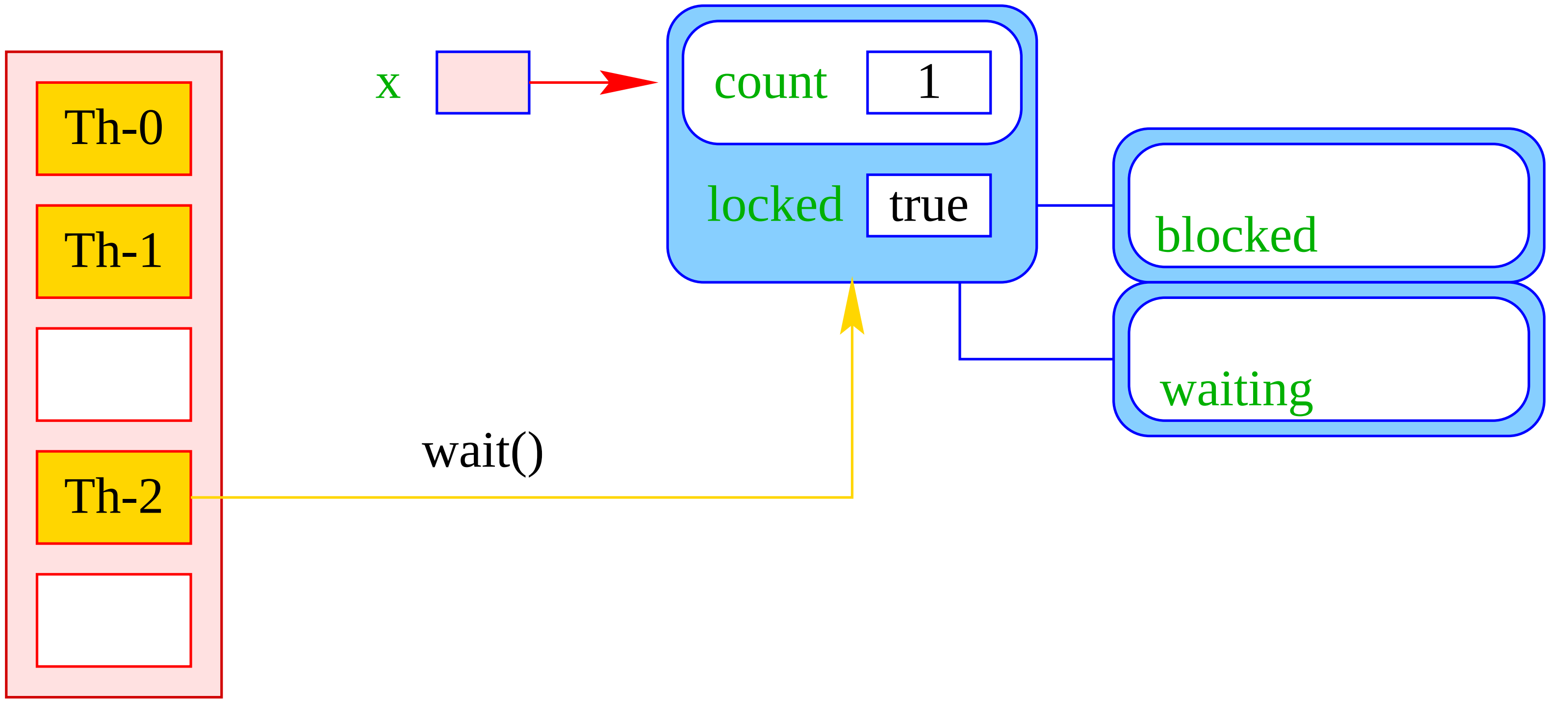

Producer
Consumer

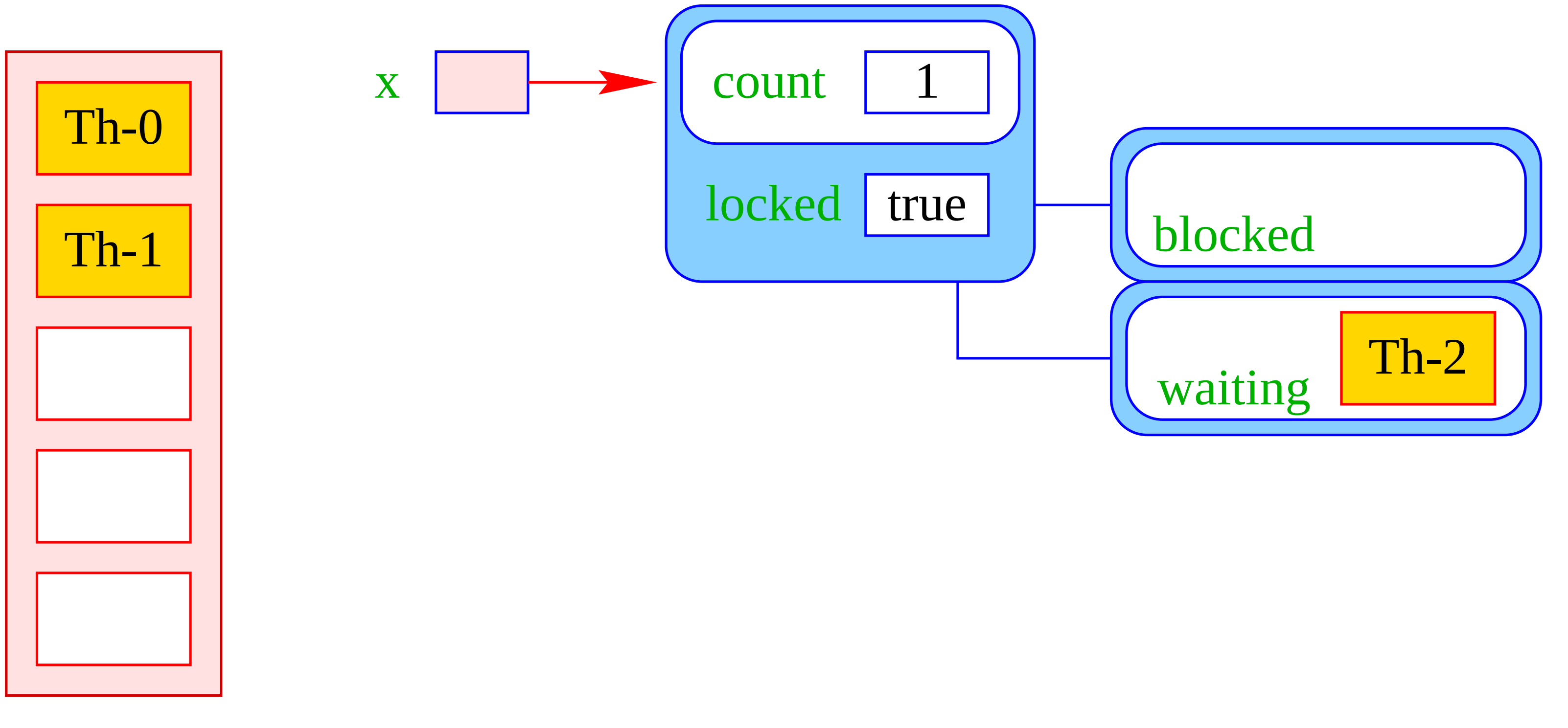
- Jedes Objekt (mit `synchronized`-Methoden) verfügt über eine weitere Schlange `ThreadQueue waitingThreads` am Objekt wartender Threads sowie die Objekt-Methoden:

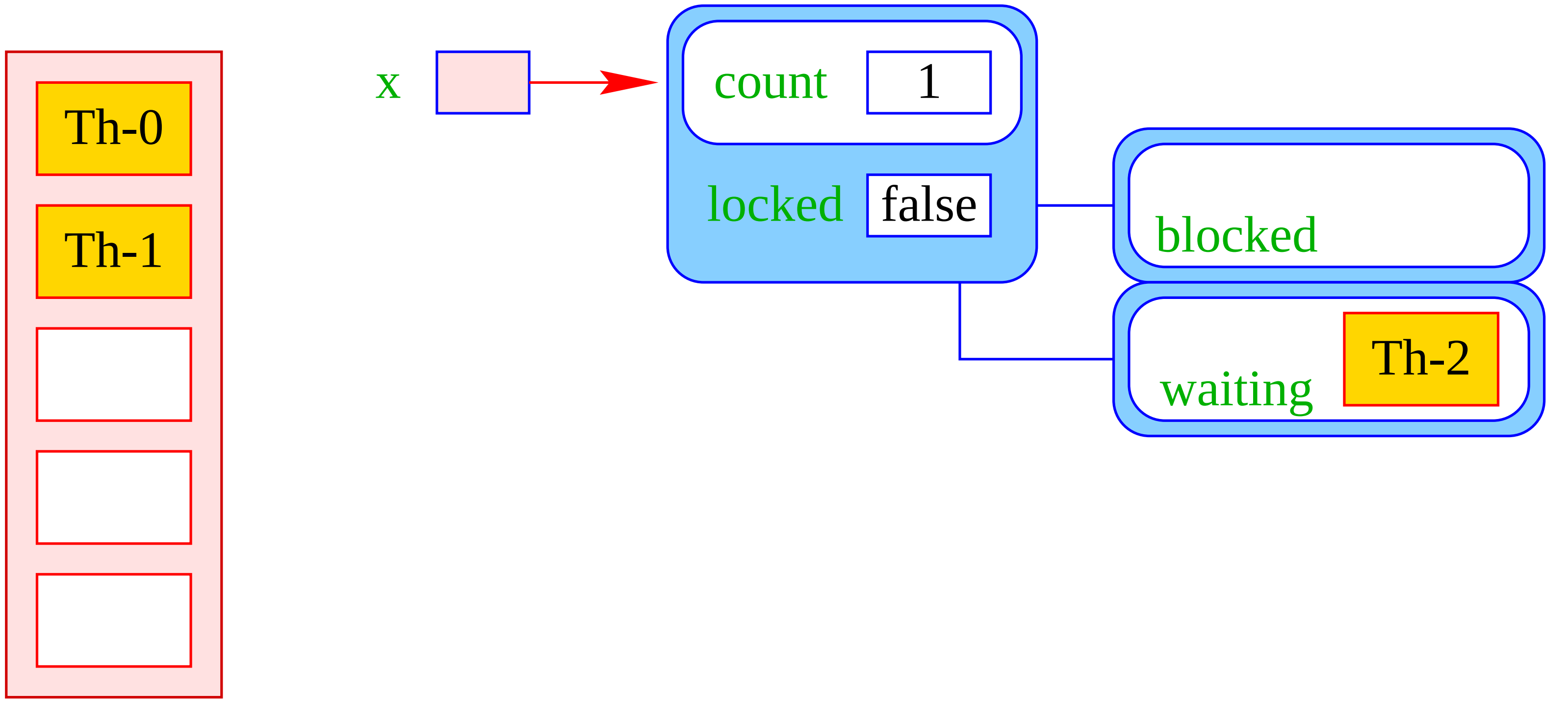
```
public final void wait() throws InterruptedException;

public final void notify();

public final void notifyAll();
```

- Diese Methoden dürfen nur für Objekte aufgerufen werden, über deren Lock der Thread verfügt !!!
- Ausführen von `wait();` setzt den Zustand des Threads auf waiting, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {
   Thread t = Thread.currentThread();
   t.state = waiting;
   waitingThreads.enqueue(t);
   unlock();
}
```


x
count
1
locked
true
blocked
waiting
Th-0
Th-1
Th-2
wait()


Th-0
Th-1
x
count
1
locked
true
blocked
waiting
Th-2


Th-0
Th-1
x
count
1
locked
false
blocked
waiting
Th-2

- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand ready ...

```
public void notify() {
   if (!waitingThreads.isEmpty()) {
      Thread t = waitingThreads.dequeue();
      t.state = ready;
   }
}
```

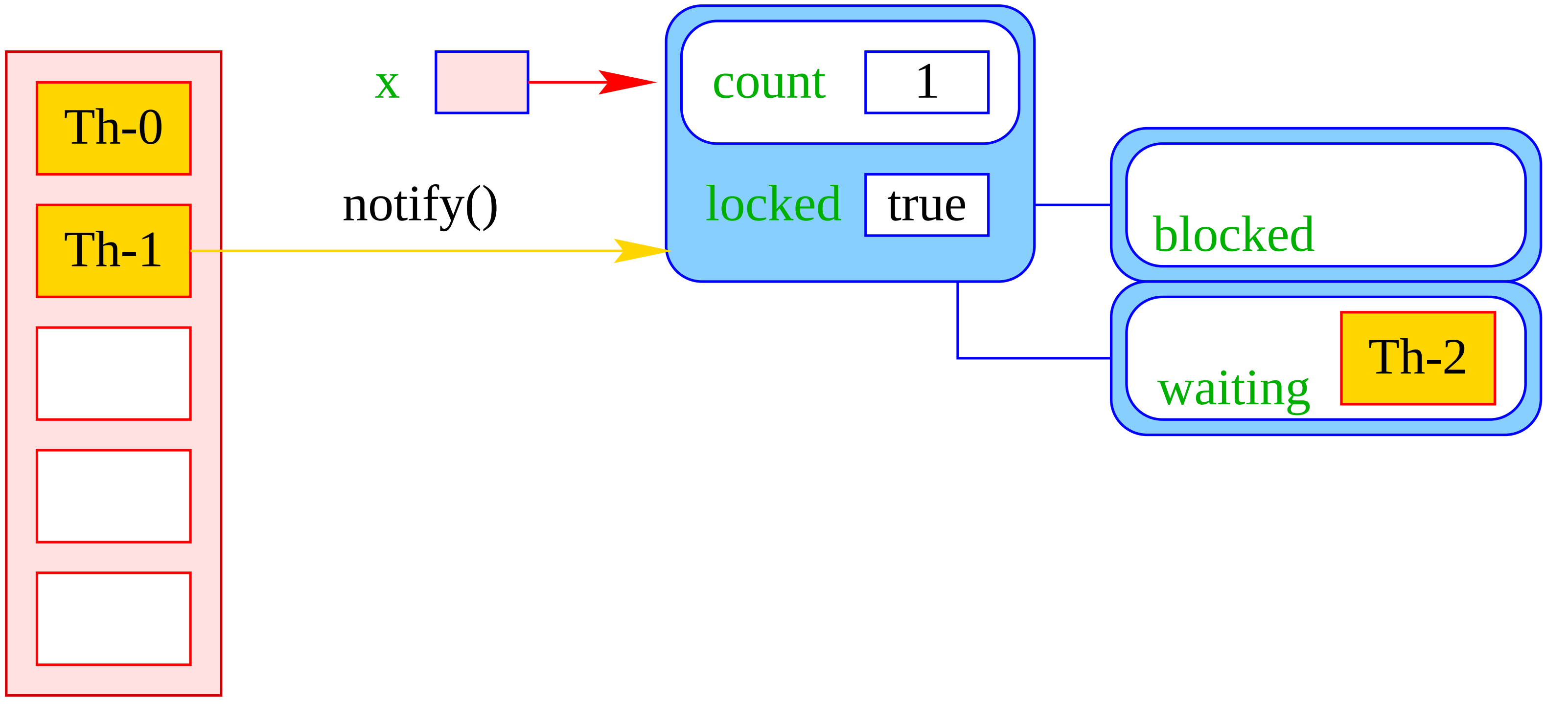
- ... mit der Auflage, erneut das Lock zu erwerben, d.h. als erste Operation hinter dem `wait();` ein `lock()` auszuführen.

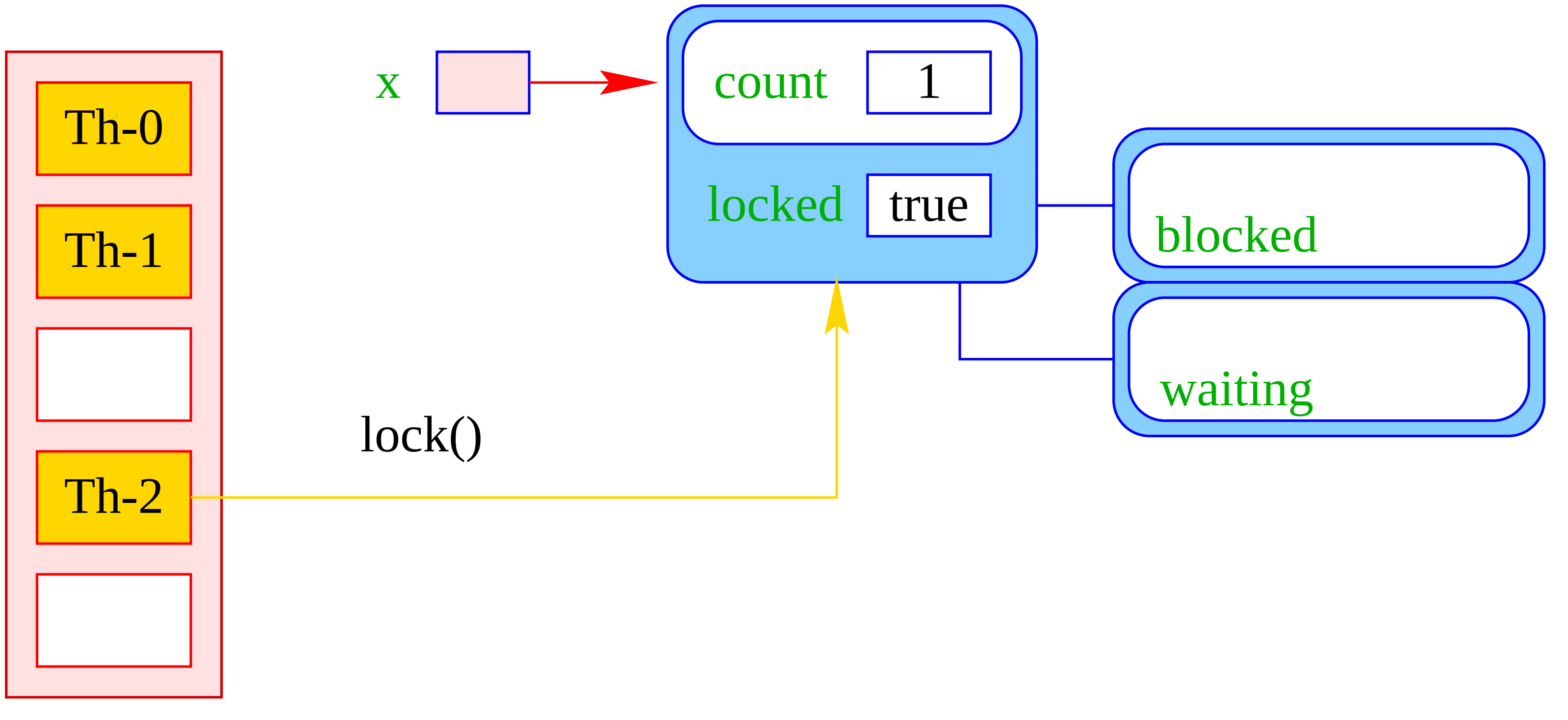
- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand ready ...

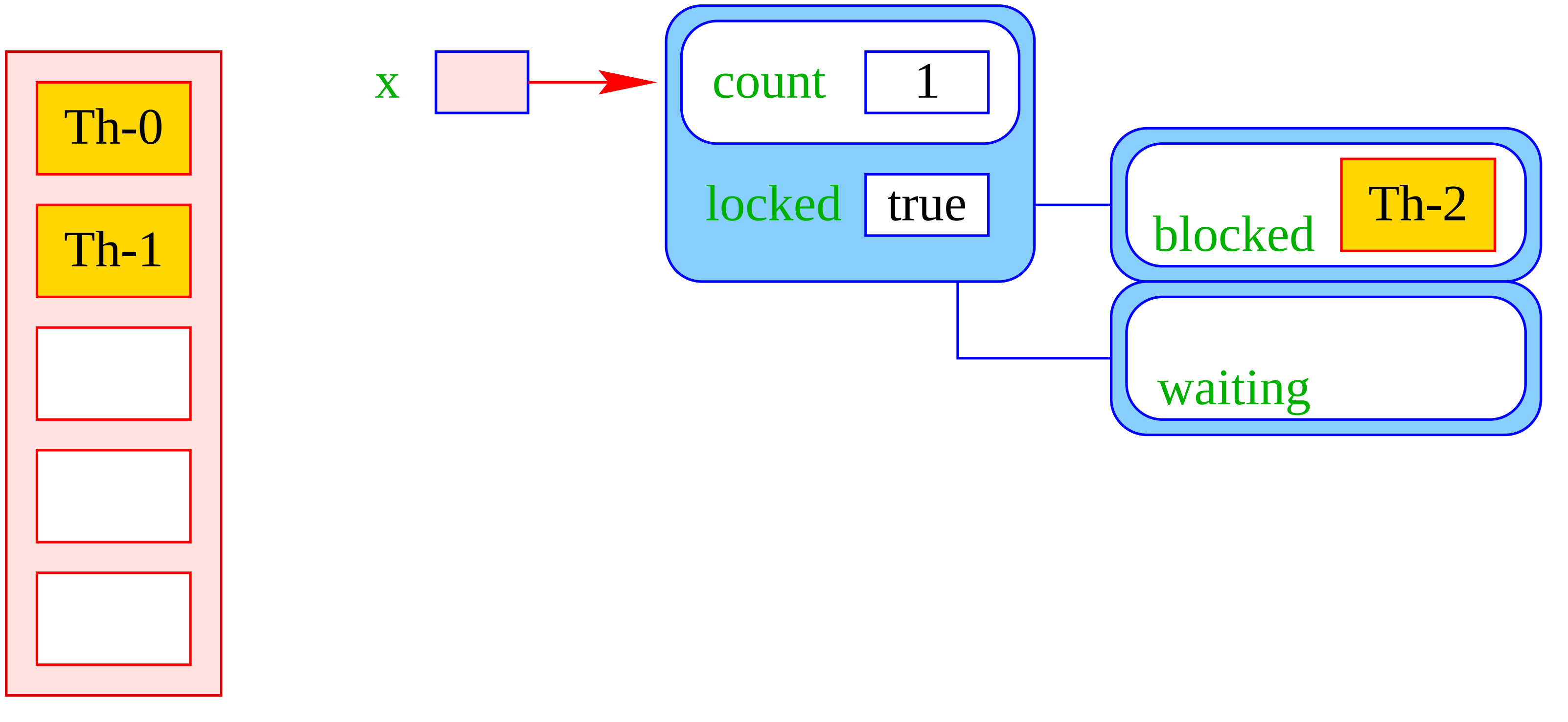
```
public void notify() {
   if (!waitingThreads.isEmpty()) {
      Thread t = waitingThreads.dequeue();
      t.state = ready;
   }
}
```

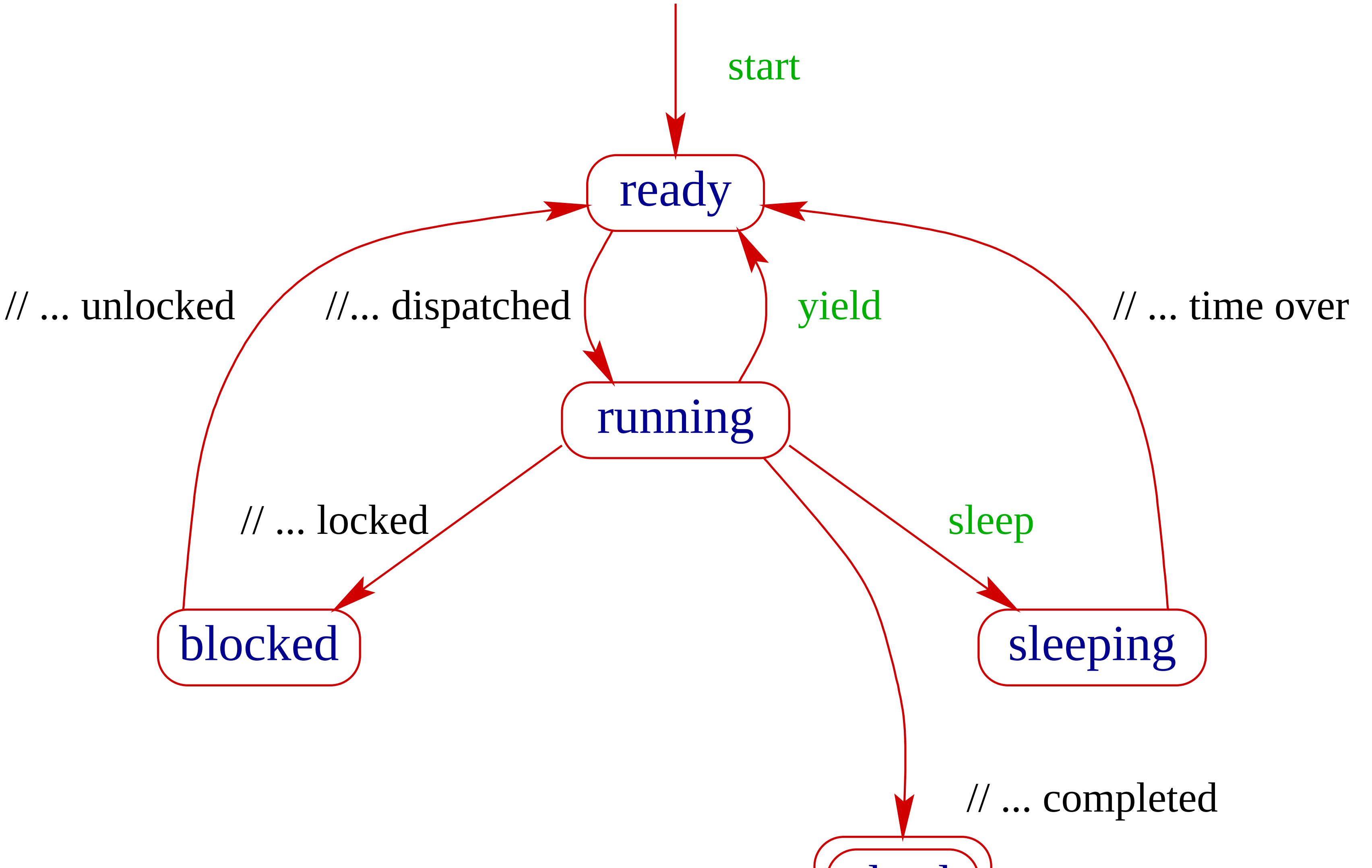
- ... mit der Auflage, erneut das Lock zu erwerben, d.h. als erste Operation hinter dem `wait();` ein `lock()` auszuführen.
- `notifyAll();` weckt alle wartenden Threads auf:

```
public void notifyAll() {
   while (!waitingThreads.isEmpty()) notify();
}
```

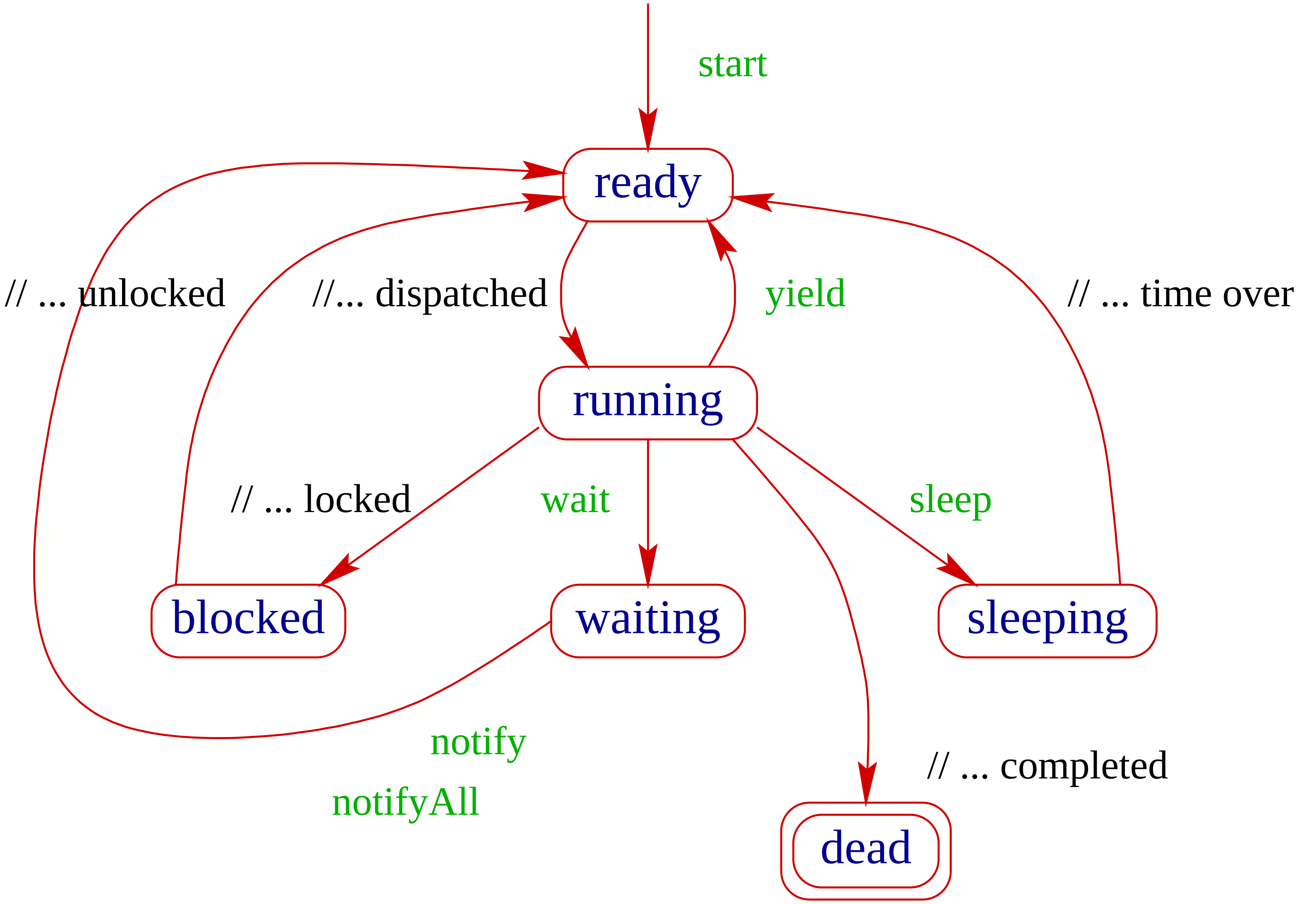

Th-0
Th-1
x
notify()
count
1
locked
true
blocked
waiting
Th-2


x
count
1
locked
true
blocked
waiting
Th-0
Th-1
Th-2
lock()


Th-0
Th-1
x
count
1
locked
true
blocked
Th-2
waiting



start
ready
//... dispatched
yield
// ... unlocked
// ... time over
running
// ... locked
sleep
blocked
sleeping
// ... completed
dead



start
ready
// ... unlocked
//... dispatched
yield
// ... time over
running
// ... locked
wait
sleep
blocked
waiting
sleeping
notify
notifyAll
// ... completed
dead

Anwendung:

```
  ...
  public synchronized void produce(Data d) throws InterruptedException {
     if (free==0) wait(); free--;
     a[last] = d;
     last = (last+1)%cap;
     notify();
  }
  public synchronized Data consume() throws InterruptedException {
     if (free==cap) wait(); free++;
     Data result = a[first];
     first = (first+1)%cap;
     notify(); return result;
  }
} // end of class Buffer2
```

- Ist der Puffer voll, d.h. keine Zelle frei, legt sich der Producer schlafen.
- Ist der Puffer leer, d.h. alle Zellen frei, legt sich der Consumer schlafen.
- Gibt es für einen Puffer genau einen Producer und einen Consumer, weckt das `notify()` des Consumers (wenn überhaupt, dann) stets den Producer ...

  ... und umgekehrt.
- Was aber, wenn es mehrere Producers gibt? Oder mehrere Consumers ???

2. Idee: Wiederholung der Tests

- Teste nach dem Aufwecken erneut, ob Zellen frei sind.
- Wecke nicht einen, sondern alle wartenden Threads auf ...

```
...
public synchronized void produce(Data d)
                              throws InterruptedException {
   while (free==0) wait(); free--;
   a[last] = d;
   last = (last+1)%cap;
   notifyAll();
}
...
```

```
  ...
  public synchronized Data consume() throws InterruptedException {
     while (free==cap) wait();
     free++;
     Data result = a[first];
     first = (first+1)%cap;
     notifyAll();
     return result;
  }
} // end of class Buffer2
```

- Wenn ein Platz im Puffer frei wird, werden sämtliche Threads aufgeweckt – obwohl evt. nur einer der Producer bzw. nur einer der Consumer aktiv werden kann :-(

3. Idee: Semaphore

- Producers und Consumers warten in verschiedenen Schlangen.
- Die Producers warten darauf, dass `free > 0` ist.
- Die Consumers warten darauf, dass `cap-free > 0` ist.

3. Idee: Semaphore

- Producers und Consumers warten in verschiedenen Schlangen.
- Die Producers warten darauf, dass `free > 0` ist.
- Die Consumers warten darauf, dass `cap-free > 0` ist.

```
public class Sema { private int x;
   public Sema(int n) { x = n; }
   public synchronized void up() {
      x++; if (x<=0) notify();
   }
   public synchronized void down() throws InterruptedException {
      x--; if (x<0) wait();
   }
}  // end of class Sema
```

- Ein Semaphor enthält eine private `int`-Objekt-Variable und bietet die `synchronized`-Methoden `up()` und `down()` an.
- `up()` erhöht die Variable, `down()` erniedrigt sie.
- Ist die Variable positiv, gibt sie die Anzahl der verfügbaren Resourcen an.

  Ist sie negativ, zählt sie die Anzahl der wartenden Threads.
- Eine `up()`-Operation weckt genau einen wartenden Thread auf.

Anwendung (1. Versuch :-)

```
public class Buffer {
   private int cap, first, last;
   private Sema free, occupied;
   private Data[] a;
   public Buffer(int n) {
      cap = n; first = last = 0;
      a = new Data[n];
      free = new Sema(n);
      occupied = new Sema(0);
   }
   ...
```

```
   ...
   public synchronized void produce(Data d) throws InterruptedException {
      free.down();
      a[last] = d;
      last = (last+1)%cap;
      occupied.up();
   }
   public synchronized Data consume() throws InterruptedException {
      occupied.down();
      Data result = a[first];
      first = (first+1)%cap;
      free.up();
      return result;
   }
}  // end of faulty class Buffer
```

- Gut gemeint – aber leider fehlerhaft ...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück ... nicht aber dasjenige für den Puffer !!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

- Gut gemeint – aber leider fehlerhaft ...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück ... nicht aber dasjenige für den Puffer !!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

Anwendung (2. Versuch :-) Entkopplung der Locks

```
    ...
    public void produce(Data d) throws InterruptedException {
       free.down();
       synchronized (this) {
          a[last] = d; last = (last+1)%cap;
       }
       occupied.up();
    }
    public Data consume() throws InterruptedException {
       occupied.down();
       synchronized (this) {
          Data result = a[first]; first = (first+1)%cap;
       }
       free.up(); return result;
    }
 }  // end of corrected class Buffer
```

- Das Statement `synchronized ( obj ) { stmts }` definiert einen kritischen Bereich für das Objekt obj, in dem die Statement-Folge stmts ausgeführt werden soll.
- Threads, die die neuen Objekt-Methoden `void produce(Data d)` bzw. `Data consume()` ausführen, benötigen zu jedem Zeitpunkt nur genau ein Lock :-)

Warnung:

Threads sind nützlich, sollten aber nur mit Vorsicht eingesetzt werden. Es ist besser,

- ... wenige Threads zu erzeugen als mehr.
- ... unabhängige Threads zu erzeugen als sich wechselseitig beeinflussende.
- ... kritische Abschnitte zu schützen, als nicht synchronisierte Operationen zu erlauben.
- ... kritsche Abschnitte zu entkoppeln, als sie zu schachteln.

Warnung (Forts.):

Finden der Fehler bzw. Überprüfung der Korrektheit ist ungleich schwieriger als für sequentielle Programme:

- Fehlerhaftes Verhalten tritt eventuell nur gelegentlich auf...
- bzw. nur für bestimmte Scheduler :-(
- Die Anzahl möglicher Programm-Ausführungsfolgen mit potentiell unterschiedlichem Verhalten ist gigantisch.

20 Applets

Applet $\quad == \quad$ Java-Programm in einer HTML-Seite

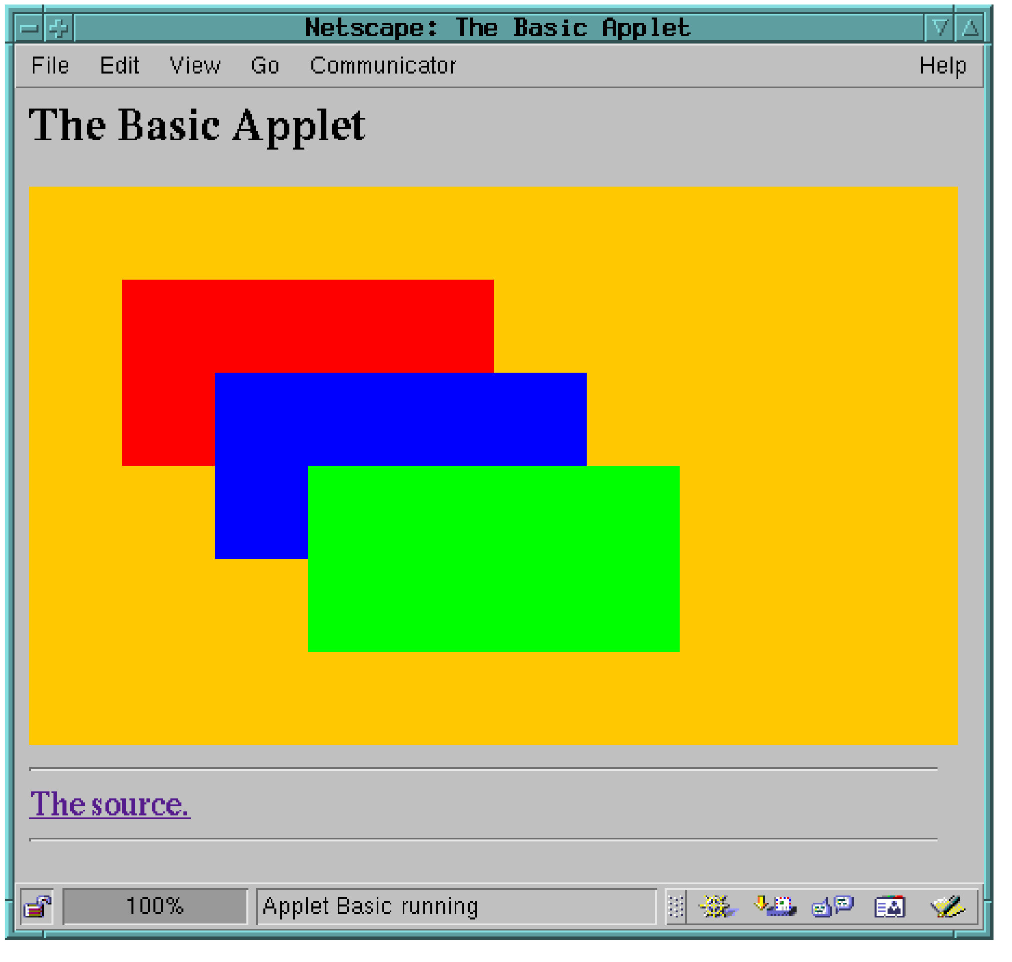
Ziele:

- audio-visuelle Gestaltung der Seite;
- Animation, d.h. Darstellen von Abfolgen von Bildern (evt. parallel zur Wiedergabe von Ton)
- Interaktion :-)

Achtung:

- Verfügbarkeit toller technischer Möglichkeiten bedeutet noch lange nicht, dass das Ergebnis beeindruckend ist ...
  ↑Kreativität
- Applets werden aus dem Internet gezogen:
  - Es gibt keine Garantie, dass sie nicht bösartig sind, d.h. versuchen, unerwünschte Effekte hervorzurufen ...
  - Applets werden darum innerhalb des Browsers ausgeführt und haben nur begrenzten Zugang zu System-Resourcen. ↑Sicherheit
- Statt im Browser können Applets auch mithilfe des Programms `appletviewer` betrachtet werden.

Ein Beispiel:



```
<html><head><title>The Basic Applet </title>
            <!-- Created by: Helmut Seidl -->
    </head>
    <body><h1>The Basic Applet</h1>
    <applet   codebase="."
              documentbase="pictures"
              code=Basic.class
              width=500
              height=300>
    Your browser is completely ignoring the &lt;APPLET&gt; tag!
    </applet>
    <hr>
    <a href="Basic.java">The source.</a>
    <hr>
    </body>
</html>
```

- HTML-Seiten bestehen aus (möglicherweise geschachtelten) Elementen, Kommentaren und Text.
- Ein Kommentar beginnt mit `<!--` und endet mit `-->`.
- Nicht-leere Elemente b beginnen mit einem Start-Tag `<b>` und enden mit einem End-Tag `</b>`.

Beispiele:

`html`	—	die gesamte Seite;
`head`	—	der Kopf der Seite;
`title`	—	der Titel der Seite;
`body`	—	der Rumpf der Seite;
`a`	—	ein Link;
`h1`	—	eine Überschrift;
`applet`	—	ein Applet.

- Bei leeren Elementen kann das End-Tag auch fehlen.

Beispiele:

`br` — Zeilen-Umbruch;

`hr` — horizontale Linie.

- Sonderzeichen beginnen mit `&` und enden mit `;` (Character references).

Beispiele:

`<` — “<”

`>` — “>”

`ä` — “ä”

- Start-Tags von Elementen können Attribute enthalten, die Zusatz-Informationen für die “Darstellung” des Elements enthalten.
- Der Wert eines Attributs kann (u.a.) ein String sein.
- Das Attribut `href` des Elements `a` gibt eine Internet-Adresse an:

```
<a href="Basic.java">The source.</a>
```

Einige Attribute des Elements `applet`:

- `codebase` — gibt das Verzeichnis an, in dem das Applet liegt;
- `documentbase` — gibt ein weiteres Verzeichnis an;
- `code` — gibt die Datei an, in der ausführbare Java-Code abgespeichert ist;
- `width` — gibt die Breite der verfügbaren Fläche an;
- `height` — gibt die Höhe der verfügbaren Fläche an.

... und jetzt das Applet selber:

```java
import java.applet.Applet;
import java.awt.*;
public class Basic extends Applet {
   public void init() {
      showStatus("... no variables to initialize!");
   }
   public void start() {
      setBackground(Color.orange);
   }
   public void stop() {
      showStatus("... stopping the Basic Applet!");
   }
   public void destroy() {
      showStatus("... destroying the Basic Applet!");
   }
   ...
```

- Ein neues Applet erweitert die Klasse `java.applet.Applet`.
- Ein Applet braucht nicht über eine Klassen-Methode `main()` zu verfügen.
- Aufrufen des Applets durch das Element `applet` einer HTML-Seite führt die folgenden Aktionen aus:
  1. Auf der Seite wird dem Applet eine Fläche zur Verfügung gestellt, die entsprechend den Attribut-Werten `width` und `height` dimensioniert ist.
  2. Ein Objekt der Applet-Klasse (hier der Klasse: `Basic`) wird angelegt.
  3. Zur Initialisierung des Objekts wird die Objekt-Methode `init()` aufgerufen.
  4. Dann wird die Objekt-Methode `start()` aufgerufen.
  5. ...

```
    ...
    public void paint(Graphics g) {
        g.setColor(Color.red);
        g.fillRect(50,50,200,100);
        g.setColor(Color.blue);
        g.fillRect(100,100,200,100);
        g.setColor(Color.green);
        g.fillRect(150,150,200,100);
        showStatus("... painting the Basic Applet!");
    }
} // end of Applet Basic
```

- Die Klasse `java.awt.Graphics` ist eine abstrakte Klasse, um Bilder zu erzeugen.
- Jede Implementierung auf einem konkreten System muss für diese Klasse eine konkrete Unterklasse bereitstellen ↑Portierbarkeit.
- Mit dem Applet verknüpft ist ein Objekt (der konkreten Unterklasse) der Klasse `Graphics`.
- Nach Ausführen der Methode `start()` wird das `Graphics`-Objekt `page` des Applets auf dem Fenster sichtbar gemacht.
- Auf dem sichtbaren `Graphics`-Objekt kann nun gemalt werden ...

Achtung:

- Wenn die Applet-Fläche verdeckt wurde und erneut sichtbar wird, muss die Applet-Fläche neu gemalt werden. Dazu verwaltet Java eine Ereignis-Schlange (event queue).
- Verändern der Sichtbarkeit der Fläche erzeugt ein `AWTEvent`-Objekt, das in die Ereignis-Schlange eingefügt wird.
- Das erste Ereignis ist (natürlich :-) die Beendigung der `start()`-Methode ...
- Das Applet fungiert als Consumer der Ereignisse.
- Es consumiert ein Ereignis "Fenster wurde sichtbar(er)", indem es den Aufruf `paint(page)` (für das aktuelle Applet) ausführt.

  Dadurch wird das Bild (hoffentlich) wiederhergestellt ...

Weitere Ereignisse:

- Modifizieren des Browser-Fenster $\Longrightarrow$ die `start()`-Methode wird erneut aufgerufen.
- Verlassen der HTML-Seite $\Longrightarrow$ die `stop()`-Methode wird aufgerufen.
- Verlassen des Browsers $\Longrightarrow$ die `destroy()`-Methode wird aufgerufen.

Nützliche Objekt-Methoden der Klasse `applet`:

- `public void showStatus(String status)` schreibt den String `status` in die Status-Zeile des Browsers.
- `public void setBackground(Color color)` setzt die Hintergrundsfarbe.
- `public Color getBackground()` liefert die aktuelle Hintergrundsfarbe.
- `public void setVisible(boolean b)` macht das `Graphics`-Objekt sichtbar bzw. unsichtbar.
- `public boolean isVisible()` teilt mit, ob Sichtbarkeit vorliegt.

20.1 Malen mit der Klasse `Graphics`

- Die Klasse `Graphics` stellt eine Reihe Methoden zur Verfügung, um einfache graphische Elemente zu zeichnen, z.B.:

  - `public void drawLine(int xsrc, int ysrc, int xdst, int ydest);` — zeichnet eine Linie von `(xsrc, ysrc)` nach `(xdst, ydst)`;

  - `public void drawRect(int x, int y, int width, int height);` — zeichnet ein Rechteck mit linker oberer Ecke `(x,y)` und gegebener Breite und Höhe;

  - `public void drawPolygon(int[] x, int[] y, int n);` — zeichnet ein Polygon, wobei die Eckpunkte gegeben sind durch `(x[0],y[0])`,..., `(x[n-1],y[n-1])`.

- Diese Operationen zeichnen nur die Umrisse.
- Soll auch das Innere gezeichnet werden, muss man statt `drawXX(...);` die entsprechende Methode `fillXX(...);` benutzen.

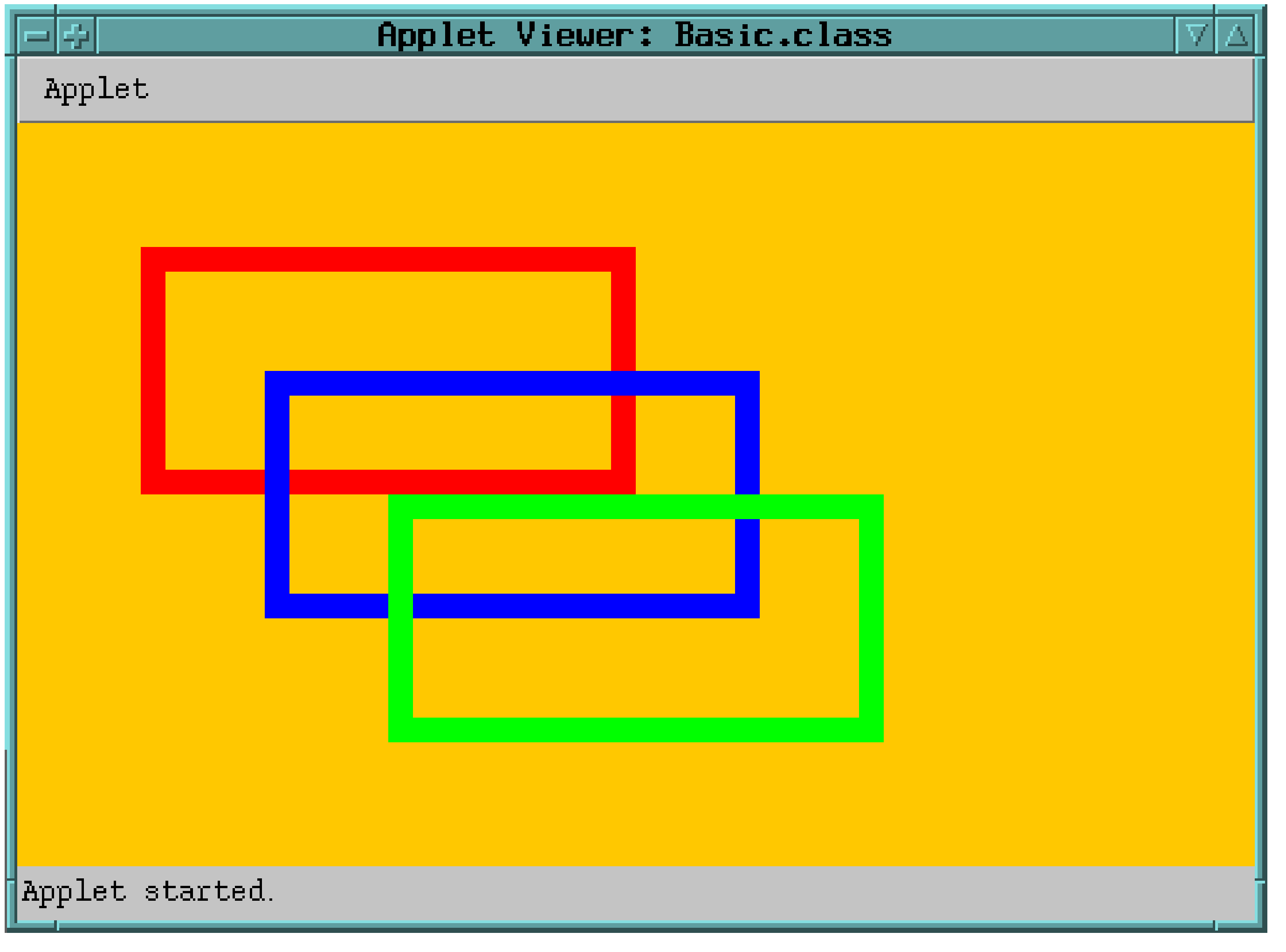
Achtung:

- Die gemalten Elemente werden sequentiell vor dem Hintergrund des Applets abgelegt.
- Die Farbe, in der aktuell gemalt wird, muss mit der `Graphics`-Objekt-Methode `public void setColor(Color color):` gesetzt werden.

Beispiel: Rechteck mit Rand

```
import java.awt.*;
public class AuxGraphics {
   private Graphics page;
   public AuxGraphics (Graphics g) { page = g;}
   public void drawRect(int x, int y,
                        int w, int h, int border) {
      page.fillRect(x,y,border,h);
      page.fillRect(x+border,y,w-2*border,border);
      page.fillRect(x+border,y+h-border,w-2*border,border);
      page.fillRect(x+w-border,y,border,h);
   }
}
```

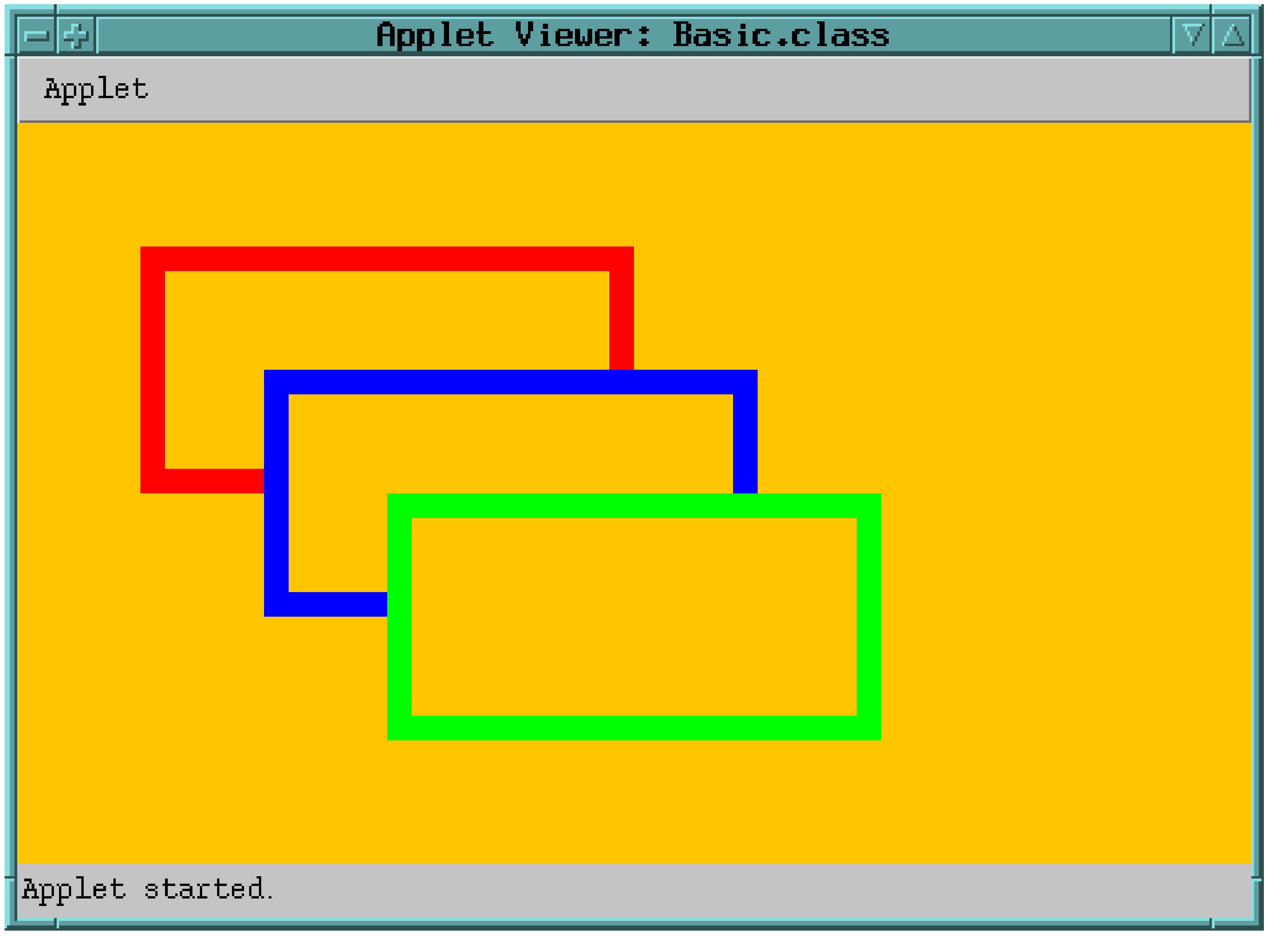
Der Rand wird aus vier kleineren Rechtecken zusammen gesetzt ...


Applet Viewer: Basic.class
Applet
Applet started.

- Man könnte auch auf die Idee kommen, das Innere des Rechtecks durch ein Rechteck in Hintergrunds-Farbe zu übermalen:

```
public void drawRect(int x, int y, int w, int h, int border) {
   Color col = page.getColor();
   page.fillRect(x, y, w, h);
   page.setColor(app.getBackground());
   page.fillRect(x+border,y+border,w-2*border,h-2*border);
   page.setColor(col);
}
```

... mit dem Effekt:


Applet Viewer: Basic.class
Applet
Applet started.

- Seit der Java-Version 1.2 ist das `Graphics`-Objekt, auf dem das Applet malt, aus der Unterklasse `Graphics2D`.
- Diese Klasse bietet tolle weitere Mal-Funktionen :-).
- Insbesondere macht sie den beim Malen verwendeten Strich (Stroke) der Programmiererin zugänglich.
- Statt dem Konzept "Farbe" gibt es nun das Interface `Paint`, das es z.B. gestattet, auch farbliche Abstufungen und Texturen zu malen :-))

20.2 Schreiben mit `Graphics`

Um (z.B. auf dem Bildschirm) schreiben zu können, benötigt man eine Schrift (Font).

Eine Schrift ...

- gehört zu einer Schrift-Familie;
- besitzt eine Ausprägung
- ... und eine Größe.

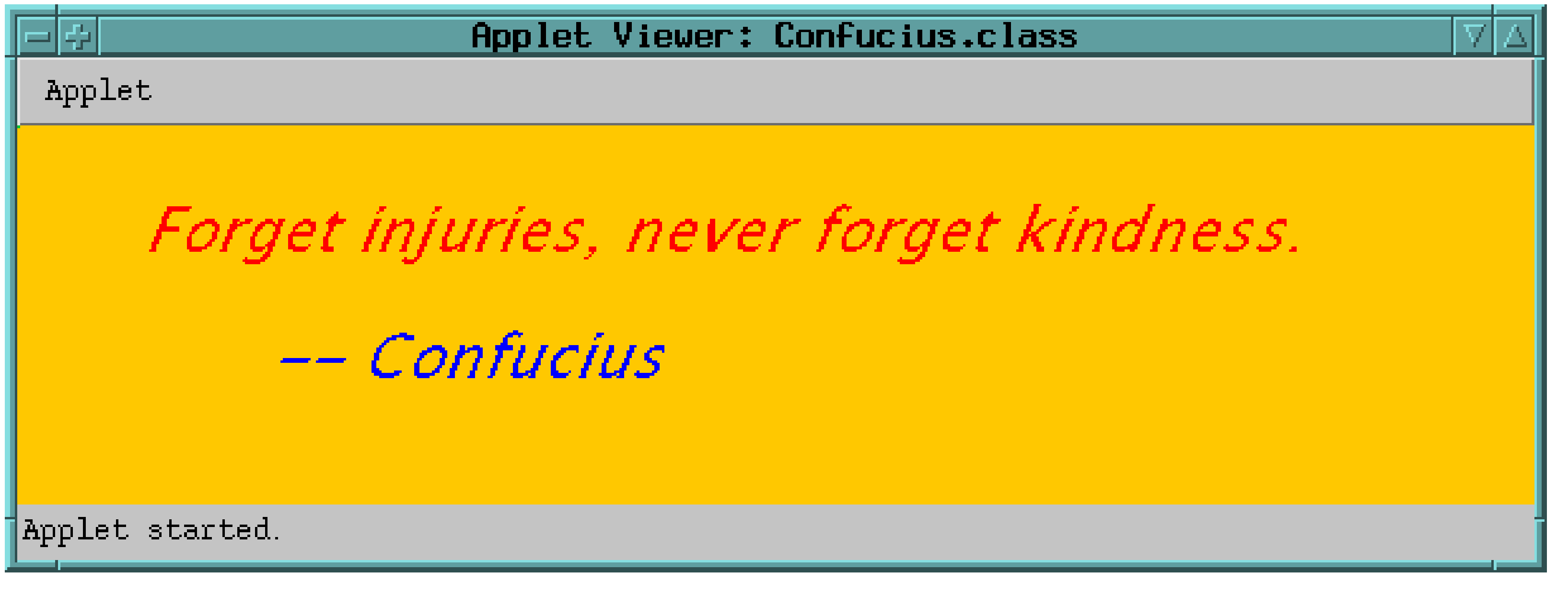
Betrachten wir erstmal ein Applet, das Zeichen mit dem voreingestellten Font darstellt ...

Beispiel: Das Confucius-Applet

```
import java.applet.Applet;
import java.awt.*;
public class Confucius extends Applet {
   public void paint (Graphics page) {
      setBackground(Color.orange);
      page.setColor(Color.red);
      page.drawString ("Forget injuries, never forget kindness.",
                       50, 50);
      page.setColor(Color.blue);
      page.drawString ("-- Confucius", 70, 70);
   } //      method paint()
}    //      end of class Confucius
```

- `public void drawString(String str, int x, int y);` ist eine Objekt-Methode der Klasse `Graphics`, die den String `str` ab der Position `(x, y)` in der aktuellen Farbe auf den Bildschirm malt.
- Der Effekt:

- Die Qualität der Wiedergabe ist so schlecht, weil
  - die Zeichen klein sind im Verhältnis zur Größe der Pixel;
  - der Screenshot für die Folie skaliert wurde :-)
- Wollen wir ein anderes Erscheinungsbild für die Zeichen des Texts, müssen wir einen anderen Font wählen ...



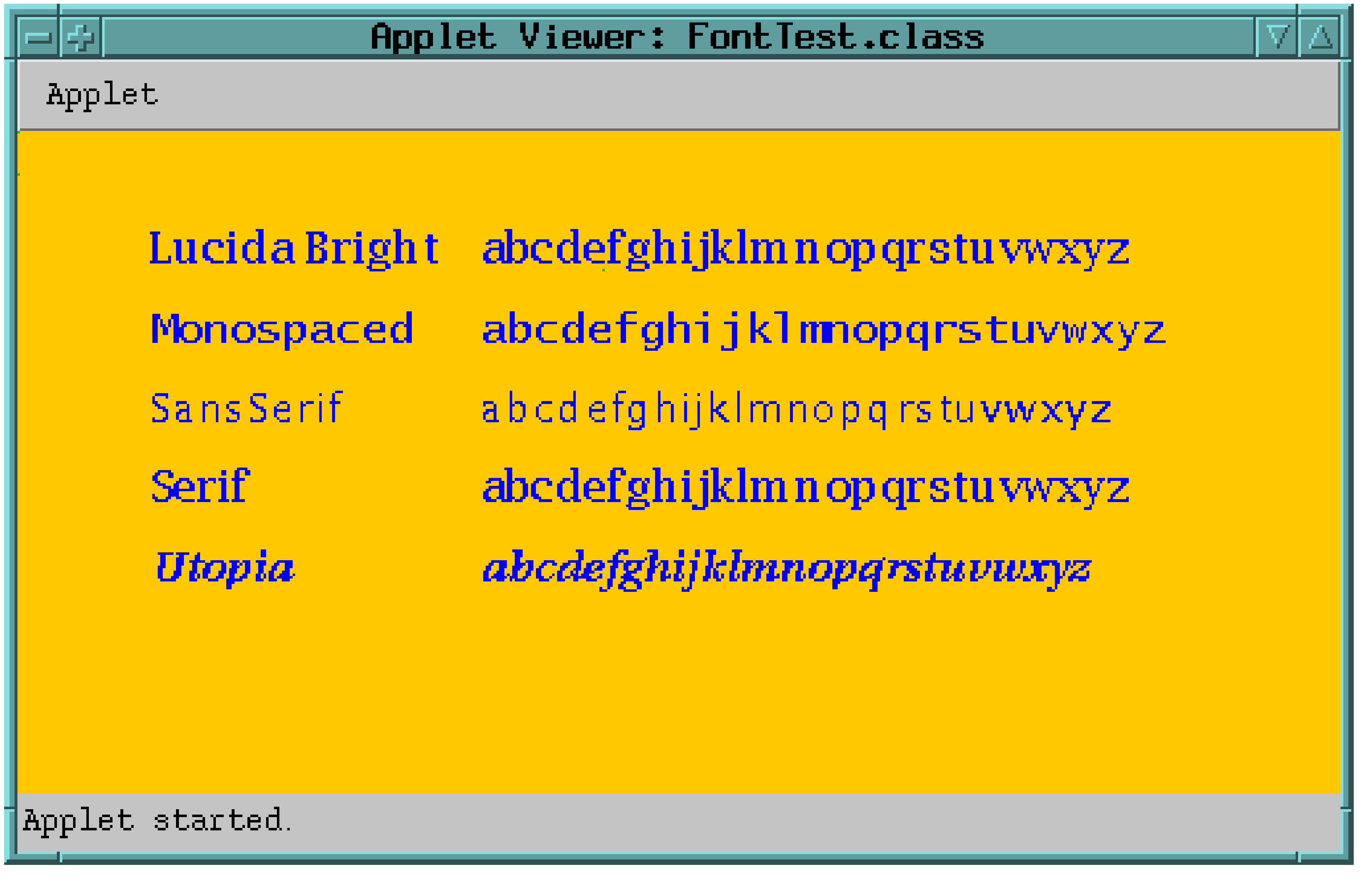
```
import java.applet.Applet;
import java.awt.*;
public class Confucius extends Applet {
   private Font font = new Font("SansSerif",Font.ITALIC,24);
   public void init() {
      setBackground(Color.orange);
      }
   public void paint (Graphics page) {
      page.setColor(Color.red);
      page.setFont(font);
      page.drawString ("Forget injuries, never forget kindness.",
                        50, 50);
      page.setColor(Color.blue);
      page.drawString ("-- Confucius", 100, 100);
   } //      method paint()
}    //      end of class Confucius
```

- Ein Java-Font wird repräsentiert durch ein Objekt der Klasse `Font` (wer hätte das gedacht? ;-)
- Eine Schrift-Familie fasst eine Menge von Schriften zusammen, die gewisse graphische und ästhetische Eigenschaften gemeinsam haben.
- SansSerif zum Beispiel verzichtet auf sämtliche Füßchen und Schwänzchen ...
- Einige Schrift-Familien, die mein Java-System kennt:

  Lucida Bright, Utopia,

  Monospaced, SansSerif, Serif

- Die untere Reihe enthält logische Familien, die verschiedenen konkreten (vom System zur Verfügung gestellten) Familien entsprechen können.


Applet Viewer: FontTest.class
Applet
Lucida Bright abcdefghijklmnopqrstuvwxyz
Monospaced abcdefghijklmnopqrstuvwxyz
SansSerif abcdefghijklmnopqrstuvwxyz
Serif abcdefghijklmnopqrstuvwxyz
Utopia abcdefghijklmnopqrstuvwxyz
Applet started.

- Als Ausprägungen unterstützt Java Normalschrift, **Fettdruck**, *Schrägstellung* und ***fette Schrägstellung***.
- Diesen entsprechen die (`int`-) Konstanten `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` und `(Font.BOLD + Font.ITALIC)`.
- Als drittes benötigen wir die Größe der Schrift (in Punkten).
- Der Konstruktor `public Font(String family, int style, int size);` erzeugt ein neues `Font`-Objekt.
- Die Objekt-Methoden:
  - `public Font getFont();`
  - `public void setFont(Font f);`
  - `public void drawString(String str, int x, int y);`

  ... der Klasse `Graphics` erlauben es, die aktuelle Schrift abzufragen bzw. zu setzen und in der aktuellen Schrift zu schreiben (`(x, y)` gibt das linke Ende der Grundlinie an)

Achtung:

- Für die Positionierung und den Zeilen-Umbruch ist der Programmierer selbst verantwortlich :-)
- Dazu sollte er die Dimensionierung seiner Schrift-Zeichen bzw. der damit gesetzten Worte bestimmen können ...
- Dafür ist die abstrakte Klasse `FontMetrics` zuständig ...

Ein Beispiel:

```
import java.applet.Applet;
import java.awt.*;
public class FontTest extends Applet {
   private String[] fontNames = {
      "Lucida Bright", "Monospaced",
      "SansSerif", "Serif", "Utopia"
      };
   private Font[] fonts = new Font[5];
   public void init() {
      for(int i=0; i<5; i++)
         fonts[i] = new Font(fontNames[i],Font.PLAIN,16);
   }
   public void start() {
      setBackground(Color.orange);
      }
   ...
```

```
   ...
   public void paint (Graphics page) {
      page.setColor(Color.blue);
      String length; FontMetrics metrics;
      for(int i=0; i<5; i++) {
         page.setFont(fonts[i]);
         page.drawString(fontNames[i],50,50+i*30);
         page.setFont(fonts[3]);
         page.drawString(":",175,50+i*30);
         metrics = page.getFontMetrics(fonts[i]);
         length = Integer.toString(metrics.stringWidth
                  ("abcdefghijklmnopqrstuvwxyz"));
         page.drawString(length,230,50+i*30);
      }
   }   //     method paint
}      //     class FontTest
```

- Die Objekt-Methoden

```
public FontMetrics getFontMetrics();
public FontMetrics getFontMetrics(Font f);
```

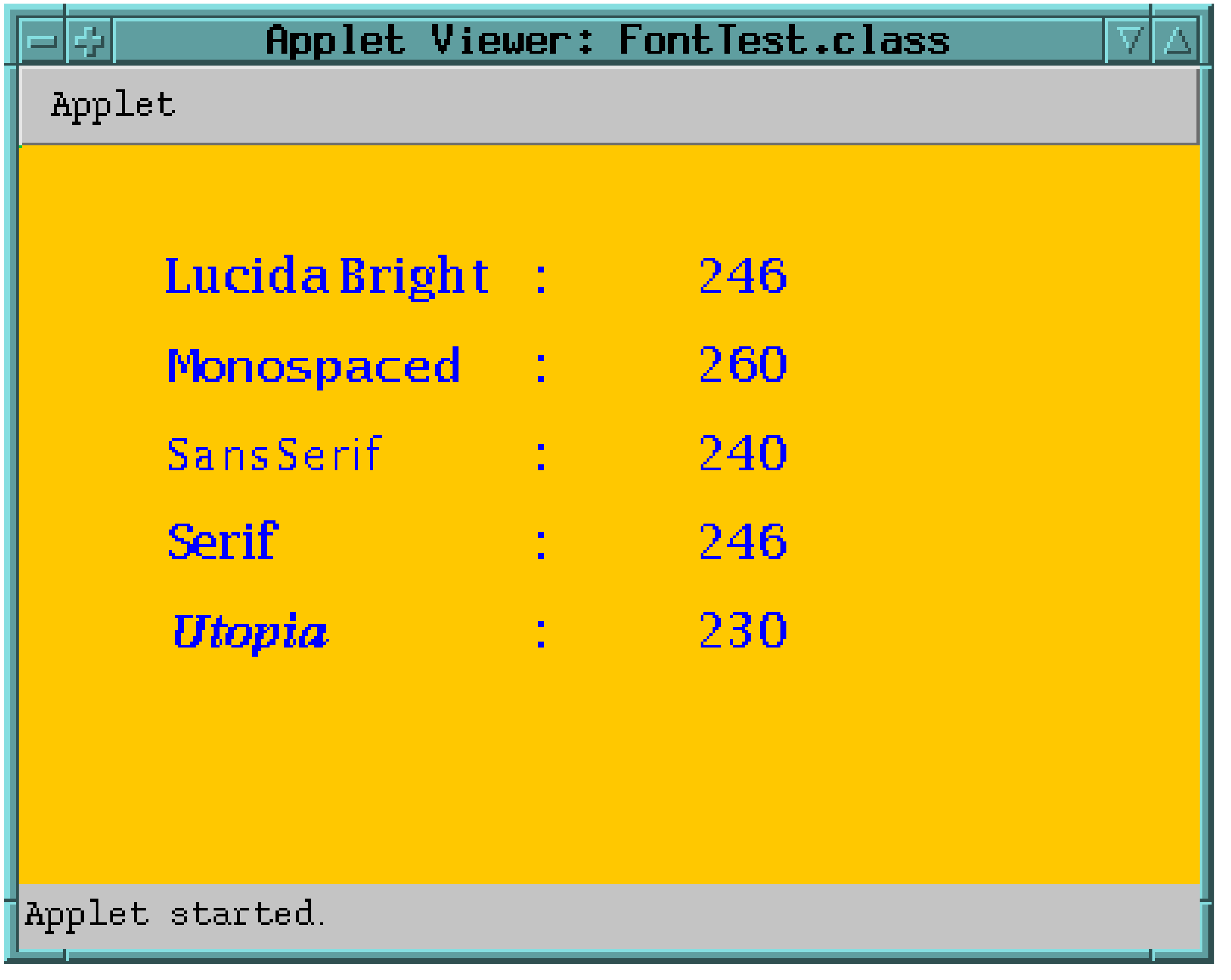
  der Klasse `Graphics` liefern das `FontMetrics`-Objekt für die aktuelle Schrift bzw. die Schrift `f`.

- Die Objekt-Methode `public int stringWidth(String string);` liefert die Länge von `string` in der aktuellen Font-Metrik ...

- Die Klassen `Font` und `FontMetrics` bieten noch viele weitere Einzelheiten bzgl. der genauen Größen-Verhältnisse :-)

- Die Objekt-Methode `public int getHeight();` liefert z.B. die maximale Höhe eines Worts in der aktuellen Schrift.

- Tja, und so sieht dann das Applet im `appletviewer` aus ...


Applet Viewer: FontTest.class
Applet
Lucida Bright : 246
Monospaced : 260
SansSerif : 240
Serif : 246
Utopia : 230
Applet started.

20.3 Animation

- Animation ist eine Bewegung vortäuschende Abfolge von Bildern (evt. mit Ton unterlegt :-)
- Für das menschliche Auge genügen 24 Bilder pro Sekunde.
- In der Zeit dazwischen legen wir das Applet schlafen ...

```
import java.applet.Applet;
import java.awt.*;
public class Grow extends Applet {
   public void start() { setBackground(Color.orange); }
   public void grow(int x, int y, Color color, Graphics g) {
      g.setColor(color);
      for(int i=0; i<100; i++) {
         g.fillRect(x,y,2*i,i);
         try {Thread.sleep(40);}
         catch (InterruptedException e) {
            System.err.println("Growing interrupted!");
         }
      }
   }
   ...
```

- Die Objekt-Methode `grow()` erhält als Argument eine Position, eine Farbe und ein `Graphics`-Objekt `g`.
- An die gegebene Position malt es in der gegebenen Farbe sukzessive ein größer werdendes gefülltes Rechteck.
- Zwischen zwei Bildern schläft es 40 Millisekunden lang ...

```
    ...
    public void paint(Graphics g) {
       grow(50,50,Color.red,g);
       grow(100,100,Color.blue,g);
       grow(150,150,Color.green,g);
    }
} // end of Applet Grow
```

- Das Ergebnis sieht miserabel aus :-(
- Das Applelt ruckelt (“Flicker”).
- Offenbar ist das Malen eines Rechtecks ein längerer Vorgang, der vom Auge durchaus wahrgenommen wird ...

Lösung: Buffering

- Statt direkt auf den Bildschirm zu malen, stellen wir die Pixel-Matrix erst in einem (unsichtbaren) Puffer her.
- Den Puffer zeigen wir dann auf einen Schlag an!

```
import java.applet.Applet;
import java.awt.*;
public class BufferedGrow extends Applet {
   private Image buffer;
   private Graphics g;
   public void init() {
      buffer = createImage(500,300);
      g = buffer.getGraphics();
   }
   public void start() {
      g.setColor(Color.orange);
      g.fillRect(0,0,500,300);
   }
   public void destroy() { g.dispose(); }
   ...
```

- Objekte der Klasse `Image` enthalten eine (i.a. implementierungsabhängige) Darstellung der Pixel-Representation eines Bilds.
- `public Image createImage(int width, int height);` (Objekt-Methode einer Oberklasse von `Applet`) liefert ein neues `Image`-Objekt der gegebenen Breite und Höhe.
- `public Graphics getGraphics();` (Objekt-Methode der Klasse `Image`) liefert ein `Graphics`-Objekt für das `Image`-Objekt.

  Malen auf diesem `Graphics`-Objekt modifiziert die Pixel des `Image`-Objekts.
- `public void dispose();` (Objekt-Methode der Klasse `Graphics`) gibt das `Graphics`-Objekt wieder frei (sollte man immer tun :-)

```
...
public void grow(int x, int y, Color color, Graphics page) {
   g.setColor(color);
   for(int i=0; i<100; i++) {
      page.drawImage(buffer,0,0,this);
      g.fillRect(x,y,2*i,i);
      try {Thread.sleep(40);}
      catch (InterruptedException e) {
         System.err.println("Growing interrupted!");
      }
   }
}
...
```

```
   ...
   public void paint(Graphics page) {
      page.setClip (0,0,500,300);
      grow(50,50,Color.red,page);
      grow(100,100,Color.blue,page);
      grow(150,150,Color.green,page);
   }
} // end of Applet BufferedGrow
```

- `public void setClip(int x, int y, int width, int height)` setzt den Bereich, der neu gemalt werden soll :-)
- Ein `Image`-Objekt enthält die komplette Pixel-Information.
- `public boolean drawImage(Image buf, int x, int y, ImageObserver obs);`
  `public boolean drawImage(Image buf, int x, int y, int width, int height, ImageObserver obs);`
  (Objekt-Methoden der Klasse `Graphics`) malen das Bild `buf` an die Stelle `(x, y)` (evt. skaliert auf die gegebene Größe).
- `ImageObserver` ist dabei ein Interface, das von `Applet` implementiert wird.

Hintergrund:

- Manchmal werden fertige Bilder aus dem Internet gezogen :-)
- Dabei helfen folgende Objekt-Methoden der Klasse `Applet`:

```
public Image getImage(URL base, String file);

public URL getCodeBase();

public URL getDocumentBase();
```

  ...

- Bis ein Bild ganz geladen ist, kann es evt. bereits partiell angezeigt werden.
- Damit das klappt, muss eine Interaktion zwischen dem Empfänger-Thread und der Hardware-Komponente erfolgen, die die Pixel einsaugt ...

```
import java.applet.Applet;
import java.awt.*;

public class DrFun extends Applet {
   public void paint(Graphics g) {
      Image image = getImage(getDocumentBase(),"df20050201.jpg");
      g.drawImage(image,0,0,this);
   }
} // end of Applet DrFun
```

... zeigt die `jpg`-Datei `df20050201.jpg` auf dem Bildschirm an:


Applet-Ansicht: DrFun.class
Applet
DOCTOR FUN
1 Feb 2005
WE CALL HIM THE TALL COOL ONE.
Copyright © 2005 David Farley, d-farley@ibiblio.org
http://ibiblio.org/Dave/drfun.html
This cartoon is made available on the Internet for personal viewing only. Opinions expressed herein are solely those of the author.
Penguin researchers often use nicknames to help keep track of their subjects.
Applet gestartet

Das Applet ist jetzt gepuffert, hat aber schreckliche Nachteile:

- Bei jedem `Window`-Ereignis wird die Animation neu gestartet.
- eine laufende Animation lässt sich nicht mehr unterbrechen.

Das Applet ist jetzt gepuffert, hat aber schreckliche Nachteile:

- Bei jedem `Window`-Ereignis wird die Animation neu gestartet.
- eine laufende Animation lässt sich nicht mehr unterbrechen.

Plan 1:

1. Die Animation wird von der `start()`-Methode gestartet ...
2. ... und läuft in einem separaten Thread.
3. `paint()` wiederholt nur den aktuellen Puffer-Inhalt.
4. Um die Animation zu unterbrechen, verwalten wir eine separate Variable `boolean stopped`, die von `stop()` gesetzt wird.
5. Ist `stopped == true`, beendet sich die Animation.

```
import java.applet.Applet;
import java.awt.*;
class StopAux extends Thread {
   private Image buffer;
   private Graphics gBuff;
   private StopThread app;
   public StopAux(Image b, StopThread a) {
      buffer = b; app = a;
      gBuff = buffer.getGraphics();
      gBuff.setColor(Color.orange);
      gBuff.fillRect(0,0,500,300);
      app.repaint();
   }
   ...
```

```
...
public void run() {
   try {
      grow(50,50,Color.red);
      grow(100,100,Color.blue);
      grow(150,150,Color.green);
   } catch (InterruptedException e) { }
   gBuff.dispose();
}
...
```

- Die Animation wird von der Klasse `StopAux` realisiert.
- Einem neuen `StopAux`-Threads wird der Puffer und das Applet selbst übergeben.
- Die `run()`-Methode führt die Animation aus.

- Damit die Animation an jeder beliebigen Stelle unterbrochen werden kann, verlassen wir sie mithilfe des Werfens einer `InterruptedException`, die von der `run()`-Methode aufgefangen wird.

```
public void grow(int x, int y, Color color) throws
                                     InterruptedException {
   gBuff.setColor(color);
   for(int i=0; i<100; i++) {
      synchronized (app) {
         if (app.stopped)
            throw (new InterruptedException());
         gBuff.fillRect(x,y,2*i,i);
      }
      ...
```

```
        ... // continuing the for loop
        try {Thread.sleep(40);}
        catch (InterruptedException e) {
             System.err.println("Growing interrupted!"); }
        app.repaint();
     }      // end of for loop
  }         // end of grow ()
}           // end of class StopAux()
...
```

- Das Malen des Bilds erfolgt durch Aufruf der Methode `public void repaint();` für das Applet.
- Diese Methode sorgt dafür, dass die Applet-Darstellung neu gemalt wird. Dazu wird (für das gegebene `Applet`-Objekt) `update(getGraphics());` aufgerufen.
- Die Methode `public void update(Graphics page);` füllt die Fläche des Applets mit der Hintergrund-Farbe. Dann wird mithilfe von `paint(page);` das Applet neu gemalt.

```
   ...
public class StopThread extends Applet {
   public boolean stopped;
   private Image buffer;
   public void init() { buffer = createImage(500,300); }
   ...
```

```
  ...
  public void start() {
     synchronized (this) { stopped = false;}
     (new StopAux(buffer, this)).start();
  }
  public void stop() {
     synchronized (this) { stopped = true;}
  }
  public void update(Graphics page) {
     paint(page);
  }
  public synchronized void paint(Graphics page) {
     page.setClip (0,0,500,300);
     page.drawImage(buffer,0,0,this);
  }
} // end of Applet StopThread
```

- Auch der Zugriff auf die Variable `stopped` ist (sicherheitshalber :-) synchronisiert.
- `stop()` setzt die Variable `stopped` auf `true`, `start()` setzt sie wieder zurück.
- Außerdem legt `start()` ein neues `StopAux`-Objekt für die Animation an und startet die Animation.
- Damit wir nicht vor einem grauen Bildschirm sitzen müssen, setzen sowohl die `start()`- wie `stop()`-Methode die Sichtbarkeit auf `true`.
- die `paint()`-Methode wiederholt offensichtlich (wie beabsichtigt) das letzte Bild im Puffer.
- Die Methode `update()` wurde überschrieben, da es offenbar überflüssig ist, zuerst den Hintergrund zu malen, um dann `paint()` aufzurufen ...

Frage:

- Was, wenn beim `stop()` die Animation nicht unterbrochen, sondern nur angehalten werden soll?
- Auch soll das `start()` nicht immer eine neue Animation starten, sondern eine eventuell bereits angelaufene, aber angehaltene fortsetzen!!!

Frage:

- Was, wenn beim `stop()` die Animation nicht unterbrochen, sondern nur angehalten werden soll?
- Auch soll das `start()` nicht immer eine neue Animation starten, sondern eine eventuell bereits angelaufene, aber angehaltene fortsetzen!!!

Plan 2:

1. Ist `stopped == true`, wird die Animation nicht beendet, sondern führt ein `wait()` aus !!!
2. Damit `start()` feststellen kann, ob bereits eine Animation läuft, führen wir eine zusätzliche Variable `boolean running` ein.
3. Ist `running == true`, schickt `start()` der Animation `notify()`.

```
import java.applet.Applet;
import java.awt.*;
class WaitAux extends Thread {
   private Graphics gBuff;
   private Image buffer;
   private WaitingThread app;
   public WaitAux(Image b, WaitingThread a) {...}
   public void run() {
      grow(50,50,Color.red);
      grow(100,100,Color.blue);
      grow(150,150,Color.green);
      synchronized (app) { app.running = false; }
      gBuff.dispose();
   }
   ...
```

- Der Konstruktor für die Animation hat sich nicht geändert.
- Die `run()`-Methode braucht dagegen keine Exception mehr zu fangen.
- Dafür setzt sie am Ende die Variable `running` des Applets auf `false`.

```
   ...
   public void grow(int x, int y, Color color) {
      gBuff.setColor(color);
      for(int i=0; i<100; i++) { try {
            synchronized (app) {
               if (app.stopped) app.wait();
               gBuff.fillRect(x,y,2*i,i);
            }
            Thread.sleep(40);
         } catch (InterruptedException e) {
            System.err.println("Growing interrupted ...");
         }
         app.repaint();
      }
   }
} // end of class WaitAux()
```

- Die Methode `grow()` testet wieder jeweils vor jedem neuen Bild, ob die Animation unterbrochen wurde.
- Wurde sie unterbrochen, führt sie ein `wait()` auf das Applet selbst aus.
- Aufgeweckt werden soll sie dabei von einem erneuten Aufruf der Methode `start()`.
- Diese schickt das `notify()` natürlich nur, sofern `running == true`. Andernfalls muss eine neue Animation gestartet werden:

```
public class WaitingThread extends Applet {
   public boolean running = false;
   public boolean stopped;
   private Image buffer;
   public void init() { buffer = createImage(500,300); }
   public synchronized void start() {
      stopped = false;
      if (!running) {
         running = true;
         (new WaitAux(buffer, this)).start();
      } else notify();
   }
... // wie gehabt
}   // end of class WaitingThread
```

Fazit:

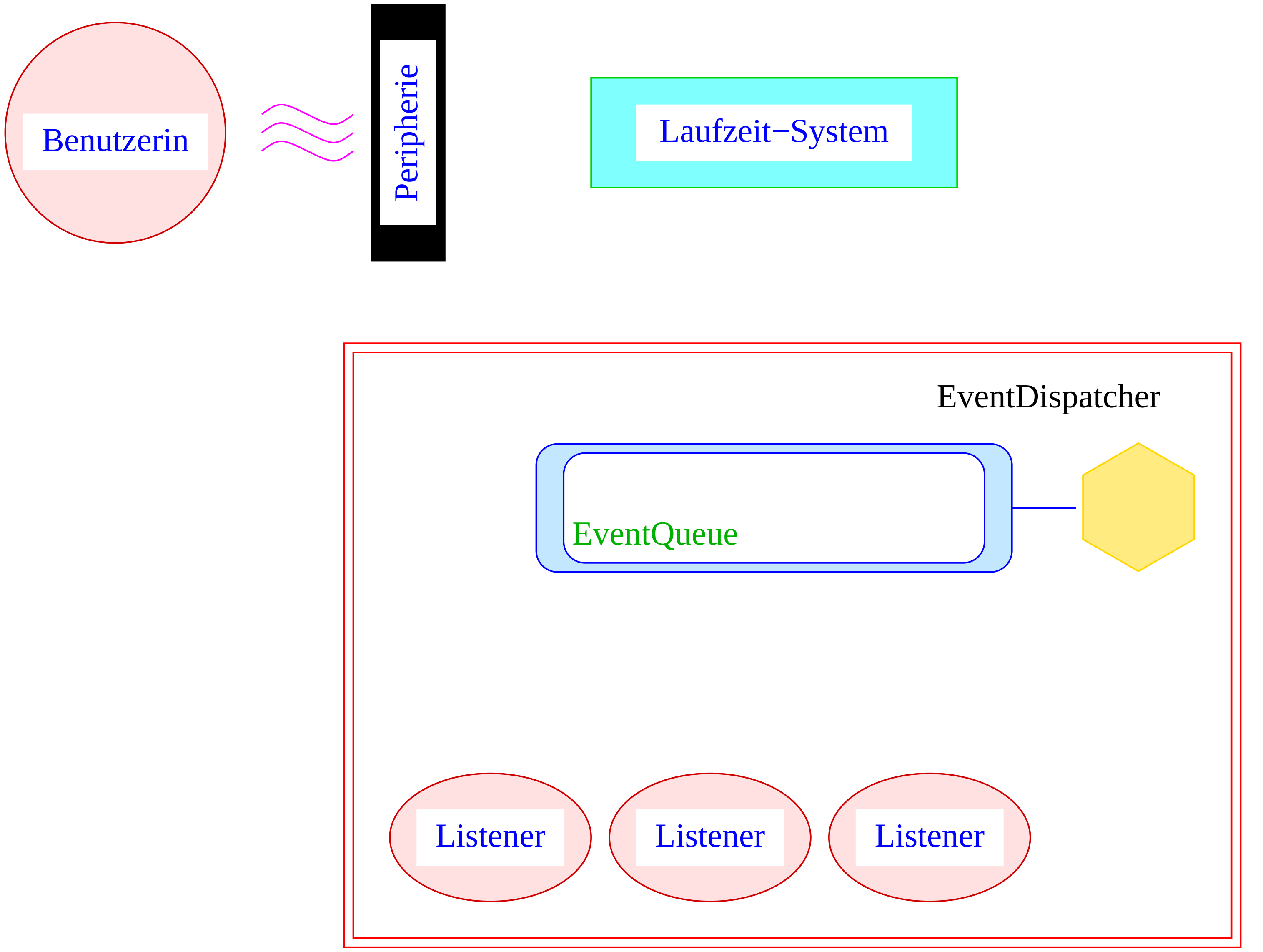
- Es ist nicht völlig trivial, eine überzeugende und robuste Animation zu programmieren :-)
- Eine Animation sollte in einem separaten Thread laufen.
- Mit den `Applet`-Methoden `start()`, `stop()`, `paint()` und `destroy()` sollte der Animations-Thread kontrolliert und auf `Window`-Ereignisse reagiert werden.
- Die Verfügbarkeit von Programmier-Hilfsmitteln wie z.B. der Klasse `Stroke` hängt hängt stark von der verwendeten Java-Version ab ...

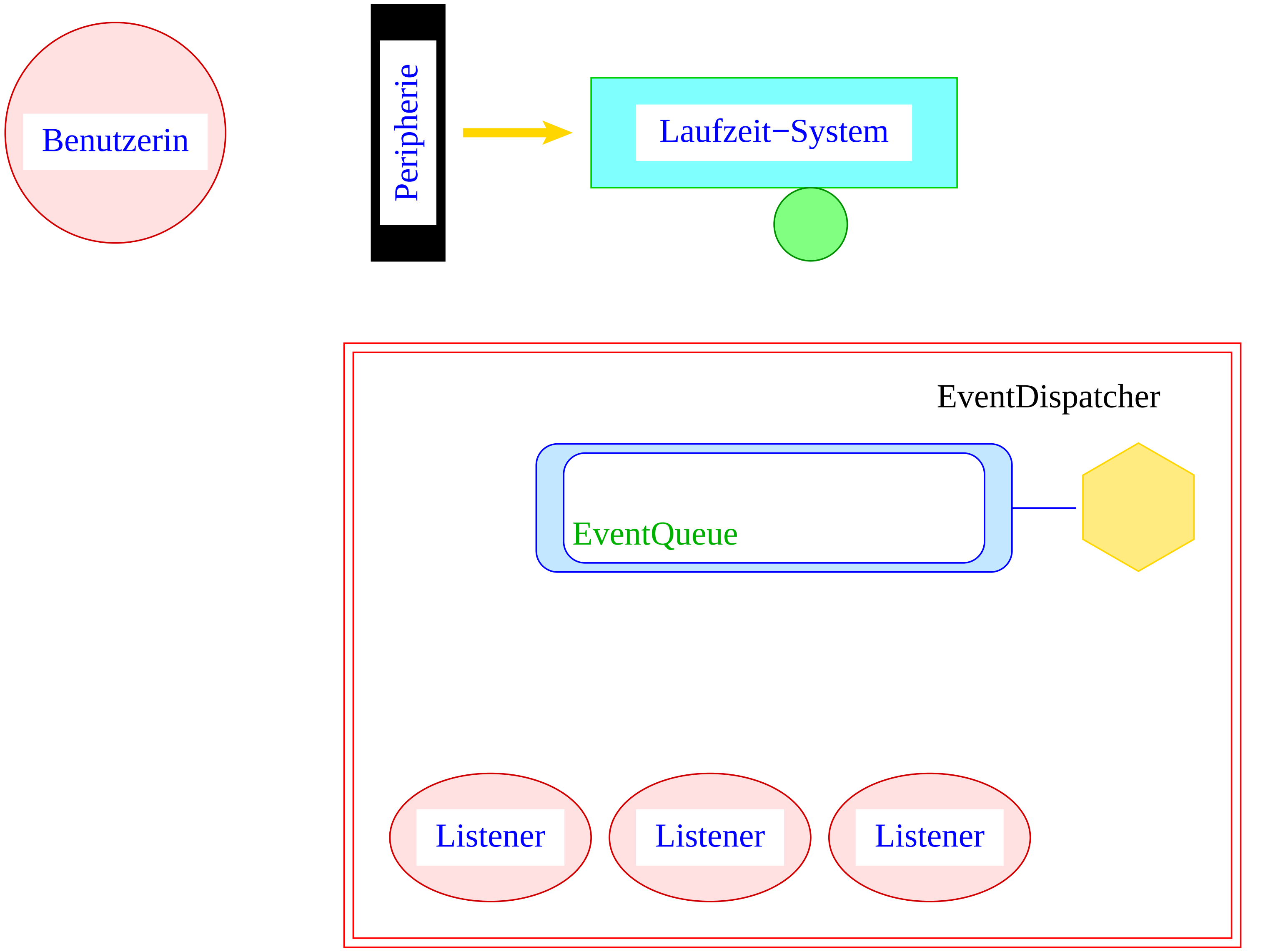
21 Graphische Benutzer-Oberflächen

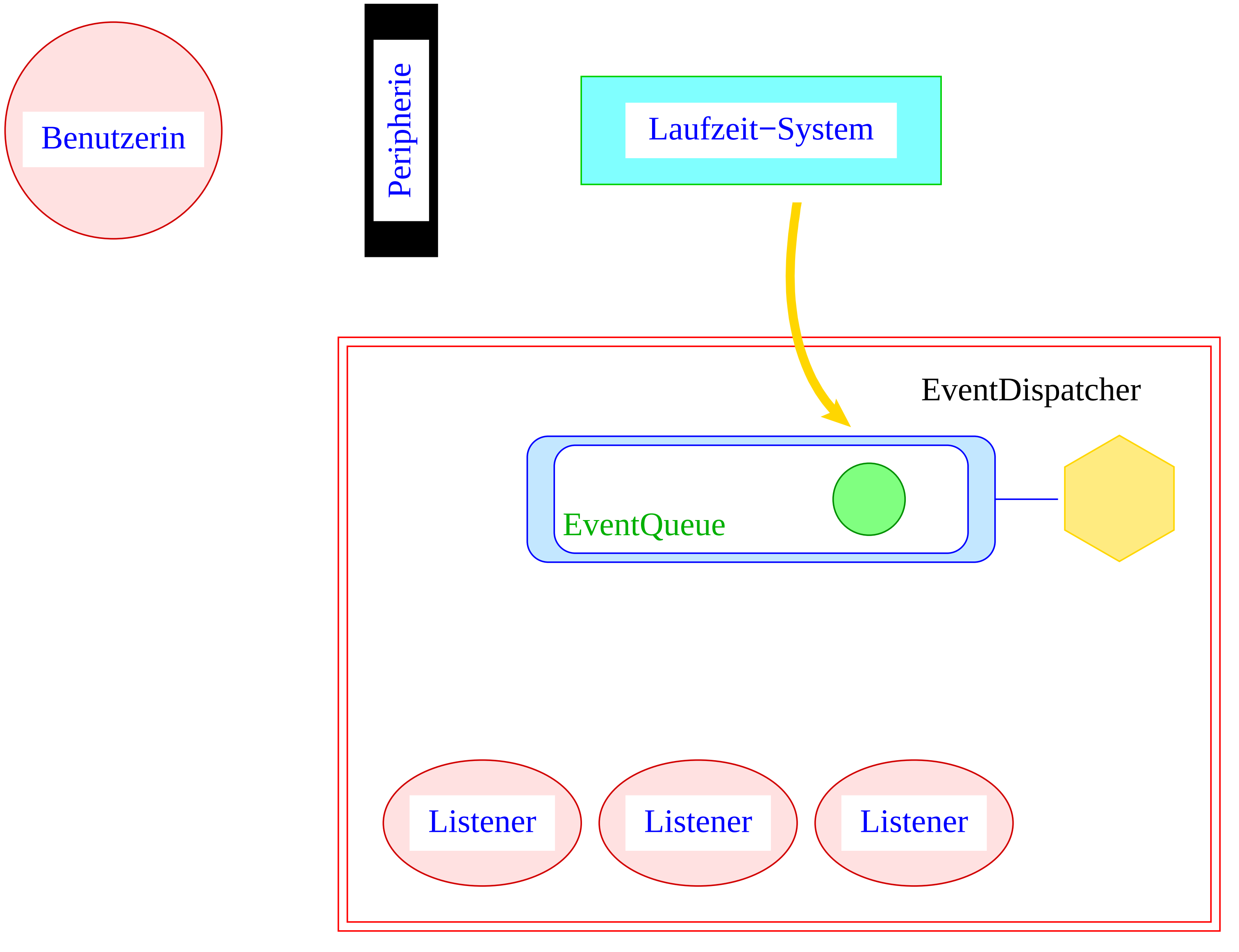
Eine graphische Benutzer-Oberfläche (GUI) ist i.a. aus mehreren Komponenten zusammen gesetzt, die einen (hoffentlich :-} intuitiven Dialog mit der Benutzerin ermöglichen sollen.

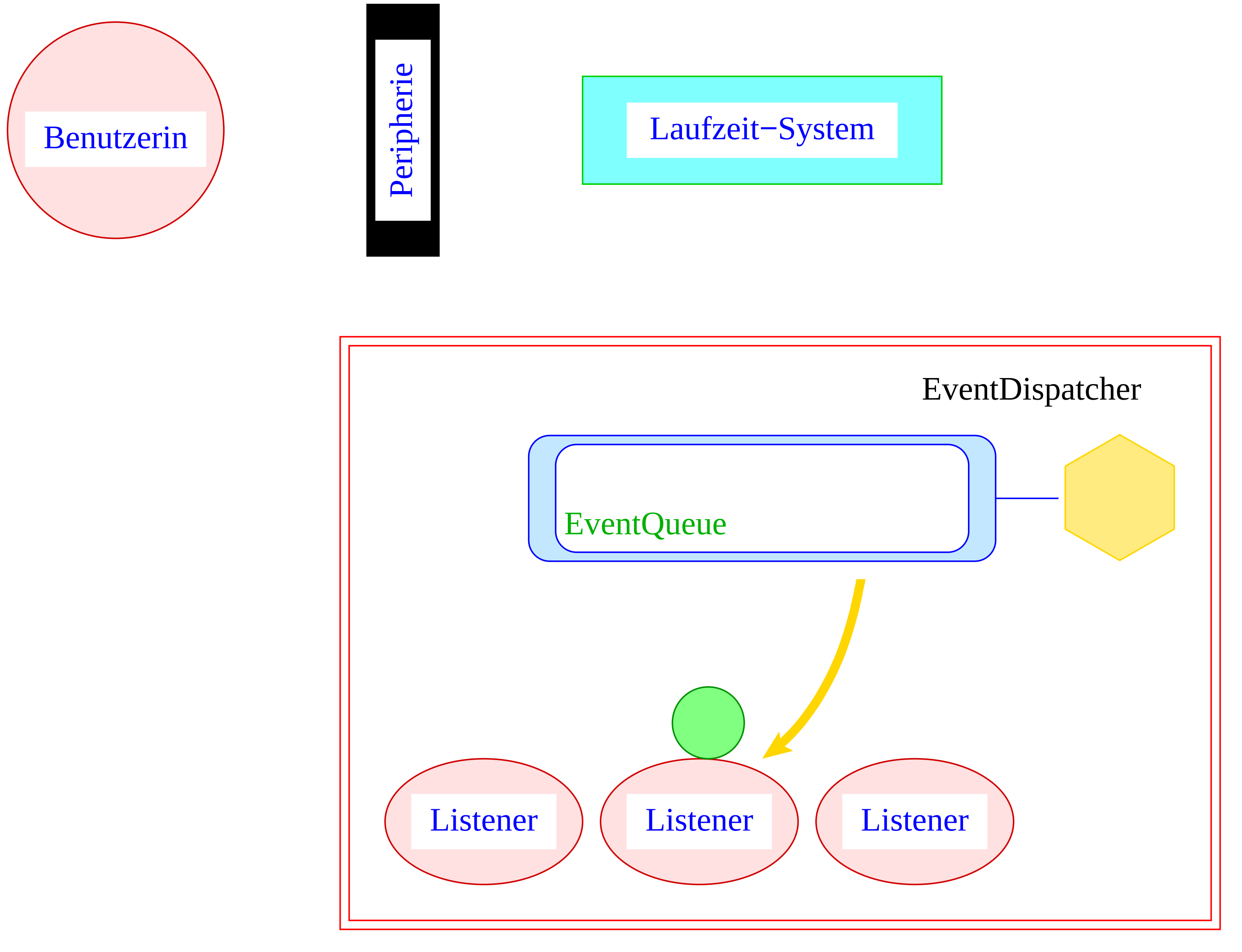
Idee:

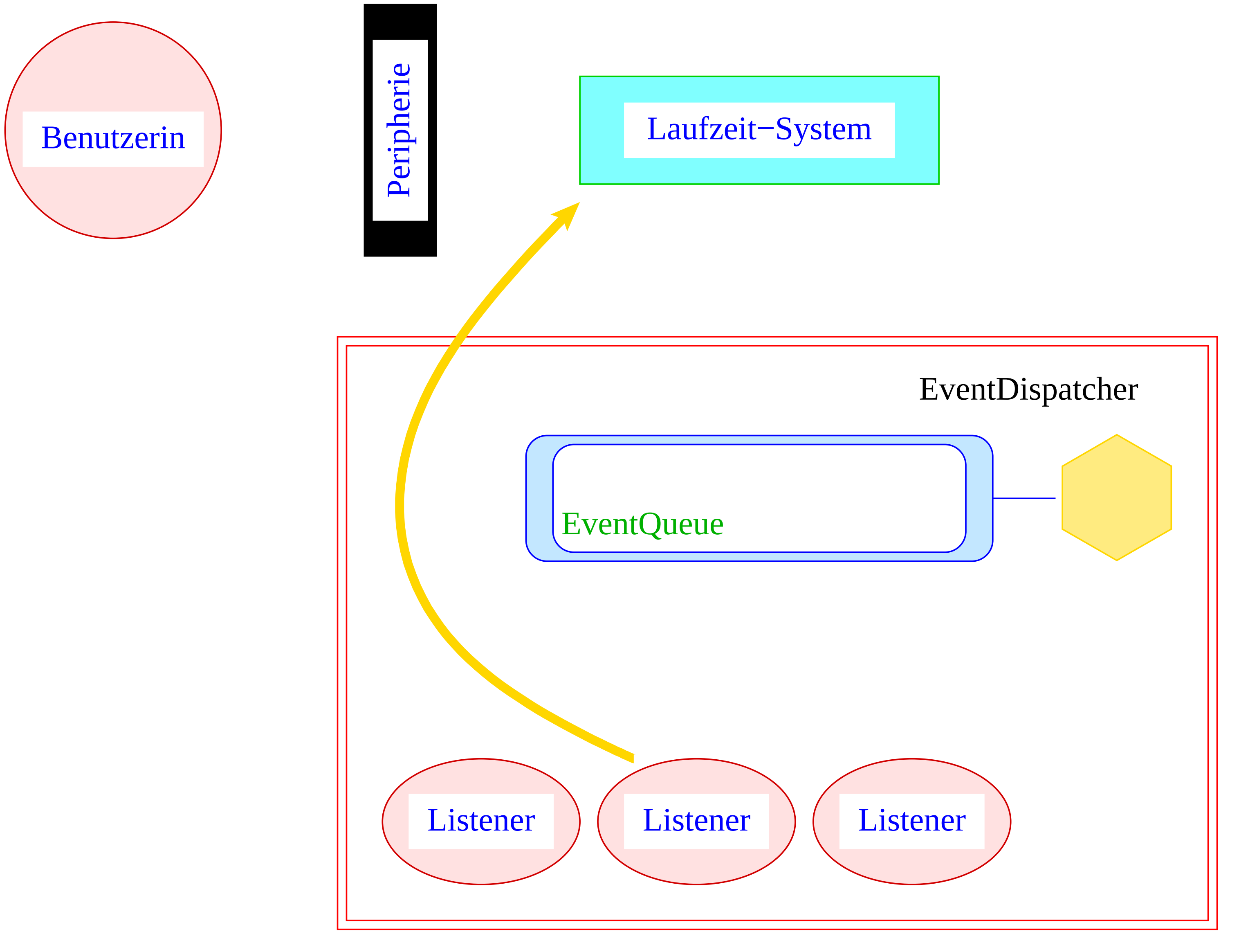
- Einzelne Komponenten bieten der Benutzerin Aktionen an.
- Ausführen der Aktionen erzeugt Ereignisse.
- Ereignisse werden an die dafür zuständigen Listener-Objekte weiter gereicht ↑Ereignis-basiertes Programmieren.

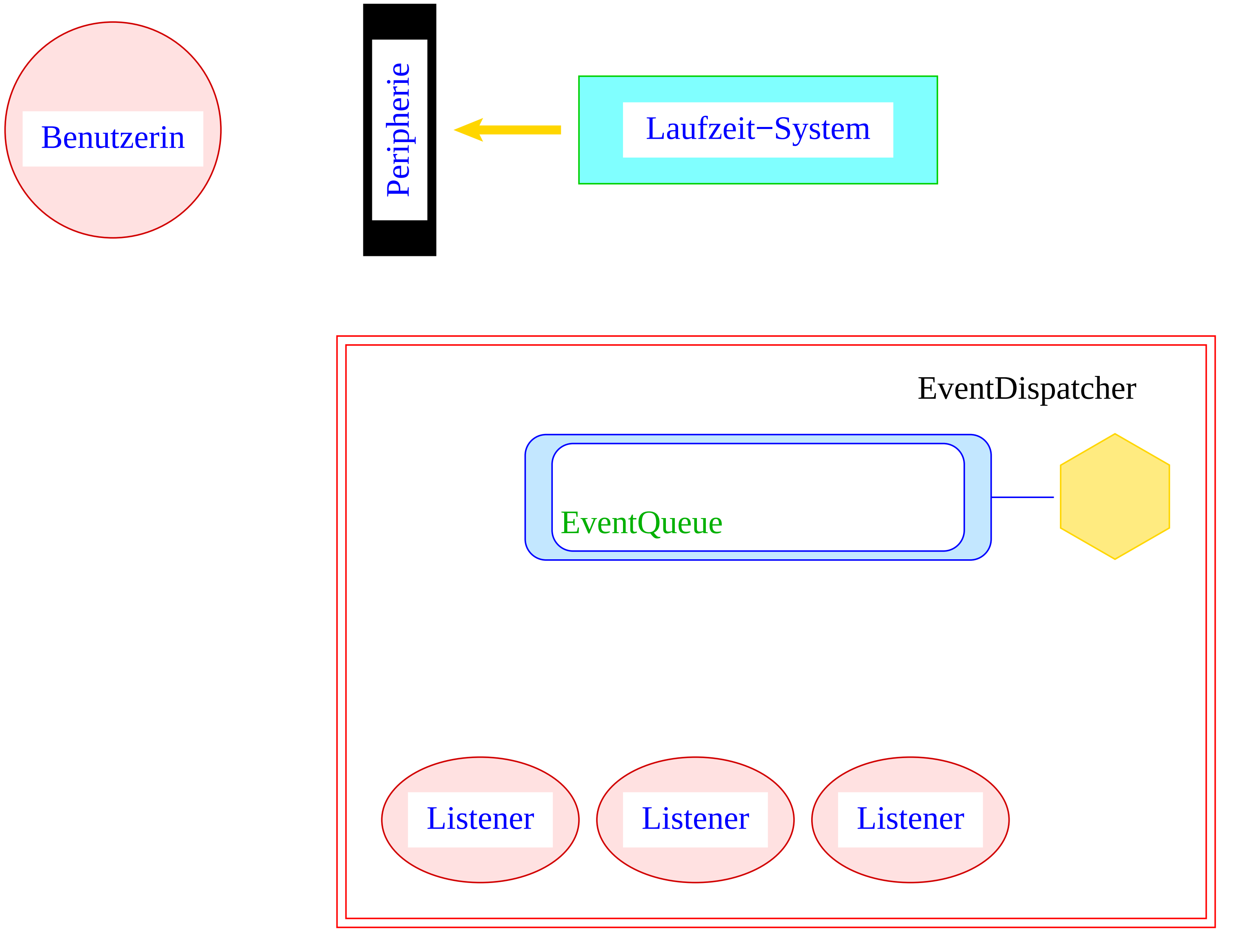

Benutzerin
Peripherie
Laufzeit−System
EventDispatcher
EventQueue
Listener
Listener
Listener

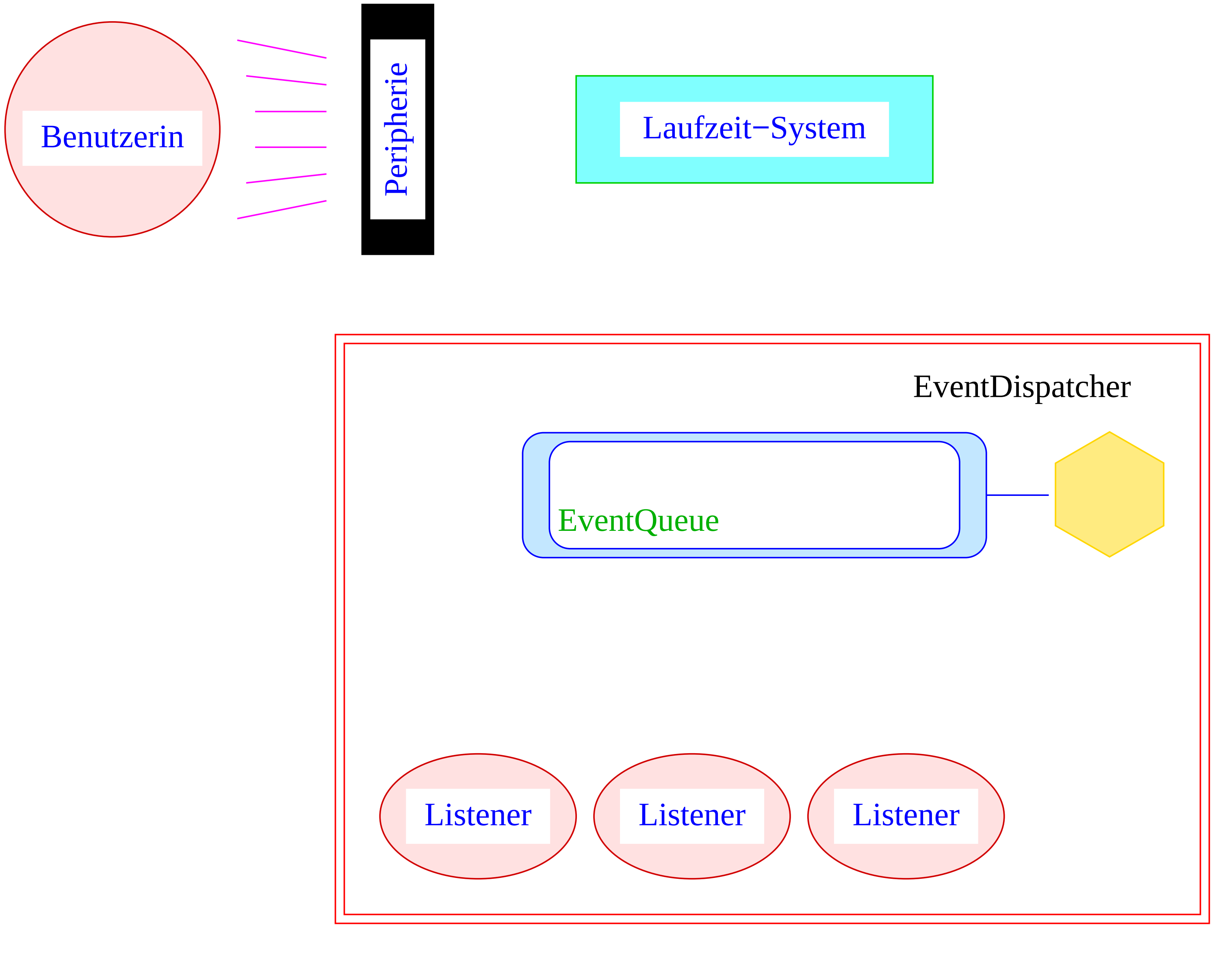

Benutzerin
Peripherie
Laufzeit−System
EventDispatcher
EventQueue
Listener
Listener
Listener


Benutzerin
Peripherie
Laufzeit−System
EventDispatcher
EventQueue
Listener
Listener
Listener


Benutzerin
Peripherie
Laufzeit−System
EventDispatcher
EventQueue
Listener
Listener
Listener


Benutzerin
Peripherie
Laufzeit−System
EventDispatcher
EventQueue
Listener
Listener
Listener


Benutzerin
Peripherie
Laufzeit−System
EventDispatcher
EventQueue
Listener
Listener
Listener


Benutzerin
Peripherie
Laufzeit−System
EventDispatcher
EventQueue
Listener
Listener
Listener

- Maus-Bewegungen und -Klicks, Tastatur-Eingaben etc. werden von der Peripherie registriert und an das ↑Betriebssystem weitergeleitet.
- Das Java-Laufzeit-System nimmt die Signale vom Betriebssystem entgegen und erzeugt dafür `AWTEvent`-Objekte.
- Diese Objekte werden in eine `AWTEventQueue` eingetragen $\Longrightarrow$ Producer!
- Die Ereignis-Schlange verwaltet die Ereignisse in der Reihenfolge, in der sie entstanden sind, kann aber auch mehrere ähnliche Ereignisse zusammenfassen ...
- Der `AWTEvent`-Dispatcher ist ein weiterer Thread, der die Ereignis-Schlange abarbeitet $\Longrightarrow$ Consumer!

- Abarbeiten eines Ereignisses bedeutet:
  1. Weiterleiten des `AWTEvent`-Objekts an das Listener-Objekt, das vorher zur Bearbeitung solcher Ereignisse angemeldet wurde;
  2. Aufrufen einer speziellen Methode des Listener-Objekts.
- Die Objekt-Methode des Listener-Objekts hat für die Reaktion des Applets zu sorgen.

$\Longrightarrow$

- Ereignisse können jederzeit eintreten.
- Ihre Abarbeitung erfolgt sequentiell.

21.1 Schon mal ein Anfang ...

Beispiel: Ein Knopf :-)

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class FirstButton extends Applet
                         implements ActionListener {
   Label label;
   Button button;
   ...
```

```
public void init() {
   setBackground(Color.orange);
   setFont(new Font("SansSerif", Font.BOLD, 18));
   label = new Label("This is my first button :-)");
   add(label);
   button = new Button("Knopf");
   button.setBackground(Color.white);
   button.addActionListener(this);
   add(button);
}
...
```

- Das Applet enthält zwei weitere Komponenten:
  (1) ein `Label`; (2) einen `Button`.
- Objekte dieser Klassen besitzen eine Aufschrift :-)

- Wie bei `Applet`-Objekten kann der Hintergrund gesetzt werden.
- `public void addActionListener(ActionListener listener)` registriert ein Objekt `listener` als dasjenige, das die von der Komponente ausgelösten `ActionEvent`-Objekte behandelt, hier: das Applet selber :-).
- `ActionListener` ist ein Interface. Zu seiner Implementierung muss die Methode `void actionPerformed(ActionEvent e)` bereitgestellt werden.
- Die Objekt-Methoden:

  ```
  void add(Component c)
  void add(Component c, int i)
  ```

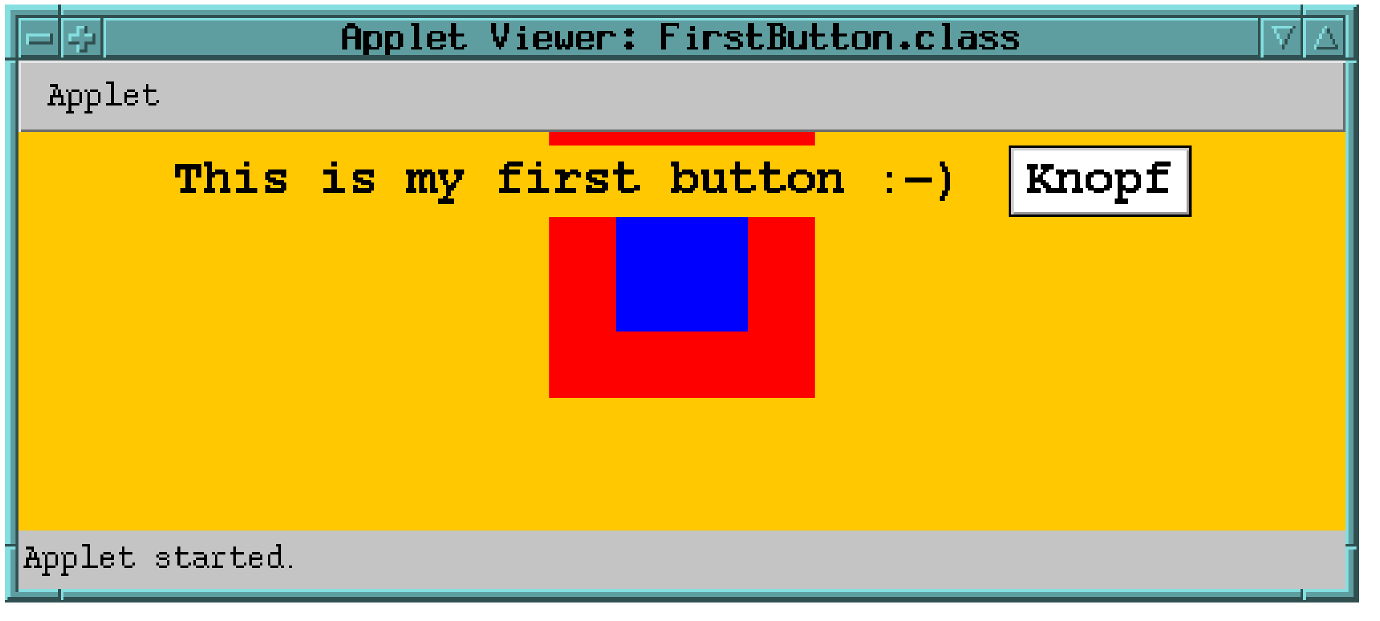
  ... fügen die Komponente `c` zum Applet hinten (bzw. an der Stelle `i`) hinzu.

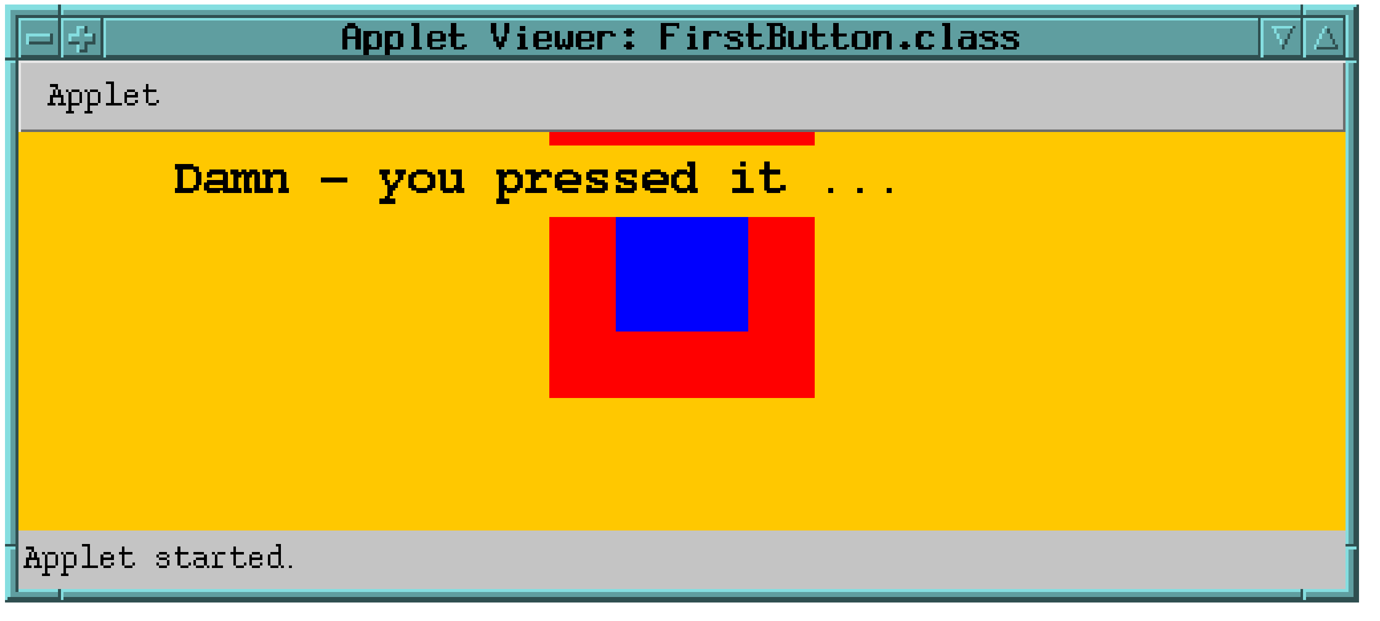
```
   ...
   public void paint(Graphics page) {
      page.setColor(Color.red);
      page.fillRect(200,0,100,100);
      page.setColor(Color.blue);
      page.fillRect(225,25,50,50);
   }
   public void actionPerformed(ActionEvent e) {
      label.setText("Damn - you pressed it ...");
      System.out.println(e);
      remove(button);
   }
} // end of Applet FirstButton
```

- Die Methode `public void actionPerformed(ActionEvent e)` setzt den Text des Labels auf einen neuen Wert und beseitigt anschließend den Knopf mithilfe der Objekt-Methode `public void remove(Component c);`
- Beim Drücken des Knopfs passiert das Folgende:

  1. ein `ActionEvent`-Objekt action wird erzeugt und in die Ereignis-Schlange eingefügt.
  2. Der `AWTEvent`-Dispatcher holt action wieder aus der Schlange. Er identifiziert das Applet app selbst als das für `action` zuständige Listener-Objekt. Darum ruft er app`.actionPerformed(` action `);` auf.

- Wären mehrere Objekte als `listener` registriert worden, würden sukzessive auch für diese entsprechende Aufrufe abgearbeitet werden.


Applet Viewer: FirstButton.class
Applet
This is my first button :-)
Knopf
Applet started.


Applet Viewer: FirstButton.class
Applet
Damn - you pressed it ...
Applet started.

Beachte:

- Die in den Labels verwendete Schriftart richtet sich nach der des umgebenden Applets (zumindest in der Größe :-).

- Die Objekt-Methoden:

```
public String getText();
public void setText(String text);
```

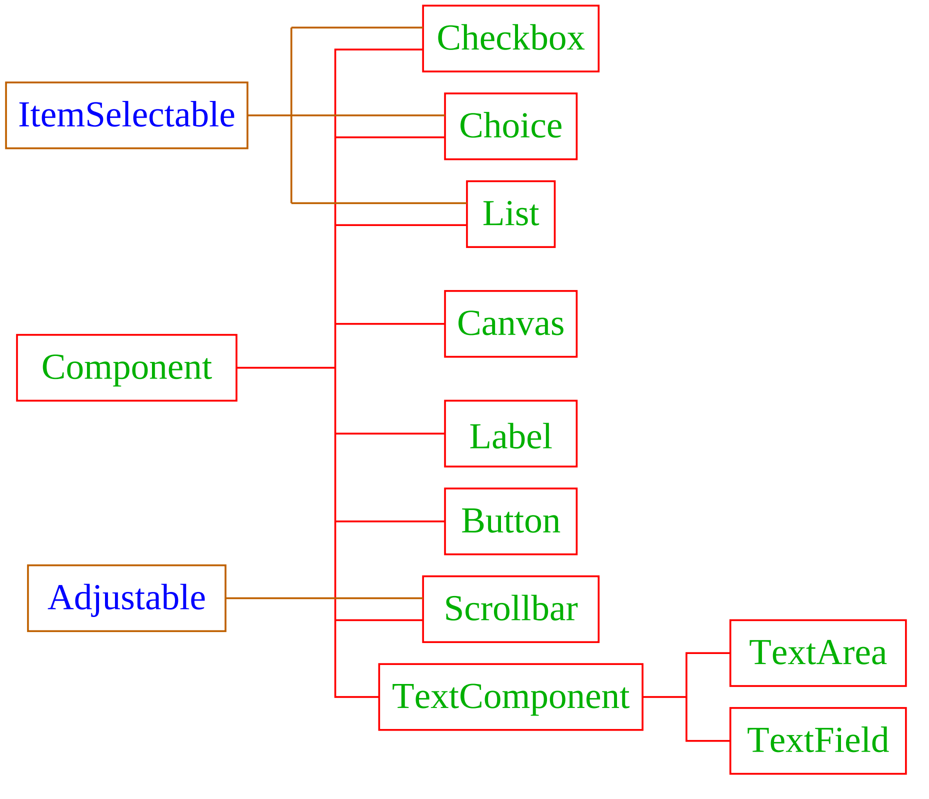
  ... gestatten den Zugriff auf den Text eines `Label-` (`Button-` oder `TextComponent`) Objekts.

- Die hinzugefügten Komponenten liegen im Applet-Feld vor der graphischen Darstellung, die die `paint()`-Methode erzeugt.

- Jede Komponente weiss, wie sie neu gemalt werden muss ...

21.2 Einfache AWT-Komponenten

Eine Übersicht ...



Canvas eine Fläche zum Malen ...

Label zeigt eine Text-Zeile.

Button einzelner Knopf, um eine Aktion auszulösen.

Scrollbar Schieber zur Eingabe von (kleinen) `int`-Zahlen.

Canvas eine Fläche zum Malen ...

Label zeigt eine Text-Zeile.

Button einzelner Knopf, um eine Aktion auszulösen.

Scrollbar Schieber zur Eingabe von (kleinen) `int`-Zahlen.

Beispiel:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class ScalingFun extends Applet
                        implements AdjustmentListener {
   private Scrollbar sHeight, sWidth;
   private Image image;
   ...
```

```
private int width = 600; private int height = 500;
public void init() {
   setBackground(Color.white);
   image = getImage(getDocumentBase(),"df991230.jpg");
   setLayout(new BorderLayout());
   sHeight = new Scrollbar(Scrollbar.VERTICAL,
                                  500,50,0,550);
   sHeight.addAdjustmentListener(this);
   add(sHeight,"West");
   sWidth = new Scrollbar(Scrollbar.HORIZONTAL,
                                  600,50,0,650);
   sWidth.addAdjustmentListener(this);
   add(sWidth,"South");
}
...
```

- Ein `Scrollbar`-Objekt erzeugt `AdjustmentEvent`-Ereignisse.
- Entsprechende Listener-Objekte müssen das Interface `AdjustmentListener` implementieren.
- Dieses Interface verlangt die Implementierung einer Methode `public void adjustmentValueChanged(AdjustmentEvent e);`
- Die `init()`-Methode des Applets legt zwei `Scrollbar`-Objekte an, eines horizontal, eines vertikal.

  Dafür gibt es in der Klasse `Scrollbar` die `int`-Konstanten `HORIZONTAL` und `VERTICAL`.
- Der Konstruktor `public Scrollbar(int dir,int init, int slide, int min, int max);` erzeugt ein `Scrollbar` der Ausrichtung `dir` mit Anfangsstellung `init`, Dicke des Schiebers `slide`, minimalem Wert `min` und maximalem Wert `max`.

  Aufgrund der Dicke des Schiebers ist der wirkliche Maximal-Wert `max - slide`.

- `void addAdjustmentListener(AdjustmentListener adj);` registriert das `AdjustmentListener`-Objekt als Listener für die `AdjustmentEvent`-Objekte des Scrollbars.
- Das selbe Objekt kann mehrmals als Listener auftreten ;-)

Bleibt, das Geheimnis um `Layout` und `West` bzw. `South` zu lüften ...

- Jeder Container, in den man weitere Komponenten schachteln möchte, muss über eine Vorschrift verfügen, wie die Komponenten anzuordnen sind.
- Diese Vorschrift heißt Layout.

  Zur Festlegung des Layouts stellt Java das Interface `LayoutManager` zur Verfügung sowie nützliche implementierende Klassen ...

- Eine davon ist das `BorderLayout`.
- Mithilfe der `String`-Argumente:

  `BorderLayout.NORTH = "North"`,

  `BorderLayout.SOUTH = "South"`,

  `BorderLayout.WEST = "West"`,

  `BorderLayout.EAST = "East"` und

  `BorderLayout.CENTER = "Center"`

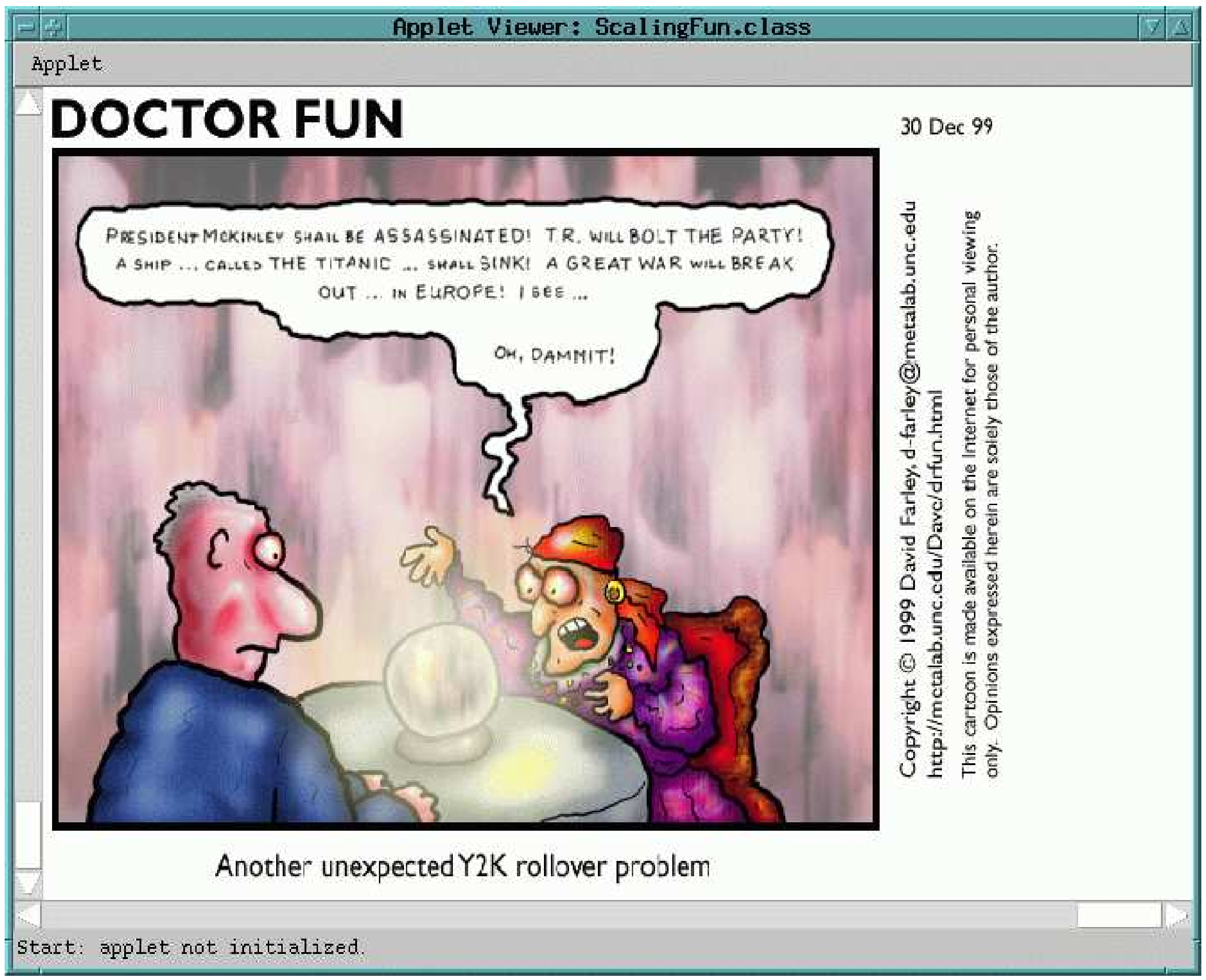
  kann man genau eine Komponente am bezeichneten Rand bzw. der Mitte positionieren.

```
   public void paint(Graphics page) {
      page.drawImage(image,0,0,width,height,this);
   }
   public void adjustmentValueChanged(AdjustmentEvent e) {
      Adjustable s = e.getAdjustable();
      int value = e.getValue();
      if (s == sHeight) height = value;
      else width = value;
      repaint();
   }
} // end of Applet ScalingFun
```

- Um `AdjustmentEvent`-Objekte behandeln zu können, implementieren wir die Methode `AdjustmentValueChanged(AdjustmentEvent e);`
- Jedes `AdjustmentEvent`-Objekt verfügt über die Objekt-Methoden:

```
public AdjustmentListener getAdjustable();
public int getValue();
```

  ... mit denen das auslösende Objekt sowie der eingestellte `int`-Wert abgefragt werden kann.
- Dann sieht das Applet so aus ...


Applet Viewer: ScalingFun.class
Applet
DOCTOR FUN
30 Dec 99
PRESIDENT McKINLEY SHALL BE ASSASSINATED! T.R. WILL BOLT THE PARTY! A SHIP ... CALLED THE TITANIC ... SHALL SINK! A GREAT WAR WILL BREAK OUT ... IN EUROPE! I SEE ...
OH, DAMMIT!
Copyright © 1999 David Farley, d-farley@metalab.unc.edu
http://metalab.unc.edu/Dave/d-fun.html
This cartoon is made available on the Internet for personal viewing only. Opinions expressed herein are solely those of the author.
Another unexpected Y2K rollover problem
Start: applet not initialized.


Applet Viewer: ScalingFun.class
Applet
DOCTOR FUN
30 Dec 99
PRESIDENT McKINLEY SHALL BE ASSASSINATED! T.R. WILL BOLT THE PARTY! A SHIP ... CALLED THE TITANIC ... SHALL SINK! A GREAT WAR WILL BREAK OUT ... IN EUROPE! I SEE ...
OH, DAMMIT!
Copyright © 1999 David Farley, d-farley@metalab.unc.edu
http://metalab.unc.edu/Dave/drfun.html
This cartoon is made available on the Internet for personal viewing only. Opinions expressed herein are solely those of the author.
Another unexpected Y2K rollover problem
Start: applet not initialized.

Texteingabe:

TextField zeigt eine Text-Zeile, die vom Benutzer modifiziert werden kann.

TextArea zeigt mehrere modifizierbare Text-Zeilen.

Auswahl aus mehreren Alternativen:

List scrollbare Liste wählbarer Items;

Choice analog `List` – nur mit Anzeige des ausgewählten Items.

Checkbox kann nur die Werte `true` oder `false` annehmen. Mehrere davon können jedoch in einer `CheckBoxGroup` zusammengefasst werden.

Beispiel: Eine Choice-Liste ...

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class FirstChoice extends Applet
                            implements ItemListener {
   private Color color = Color.white;
   private Choice colorChooser;
   ...
```

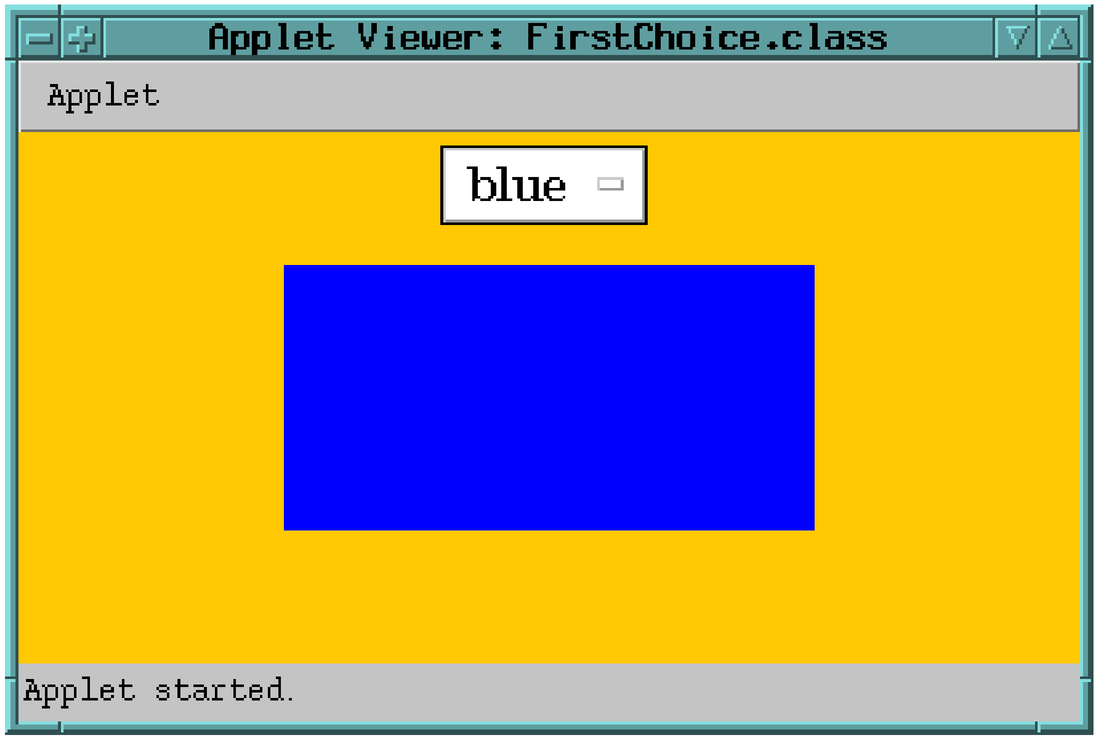
```
public void init() {
   setFont(new Font("Serif", Font.PLAIN, 18));
   colorChooser = new Choice();
   colorChooser.setBackground(Color.white);
   colorChooser.add("white");
   colorChooser.add("red");
   colorChooser.add("green");
   colorChooser.add("blue");
   colorChooser.addItemListener(this);
   add(colorChooser);
   setBackground(Color.orange);
}
...
```

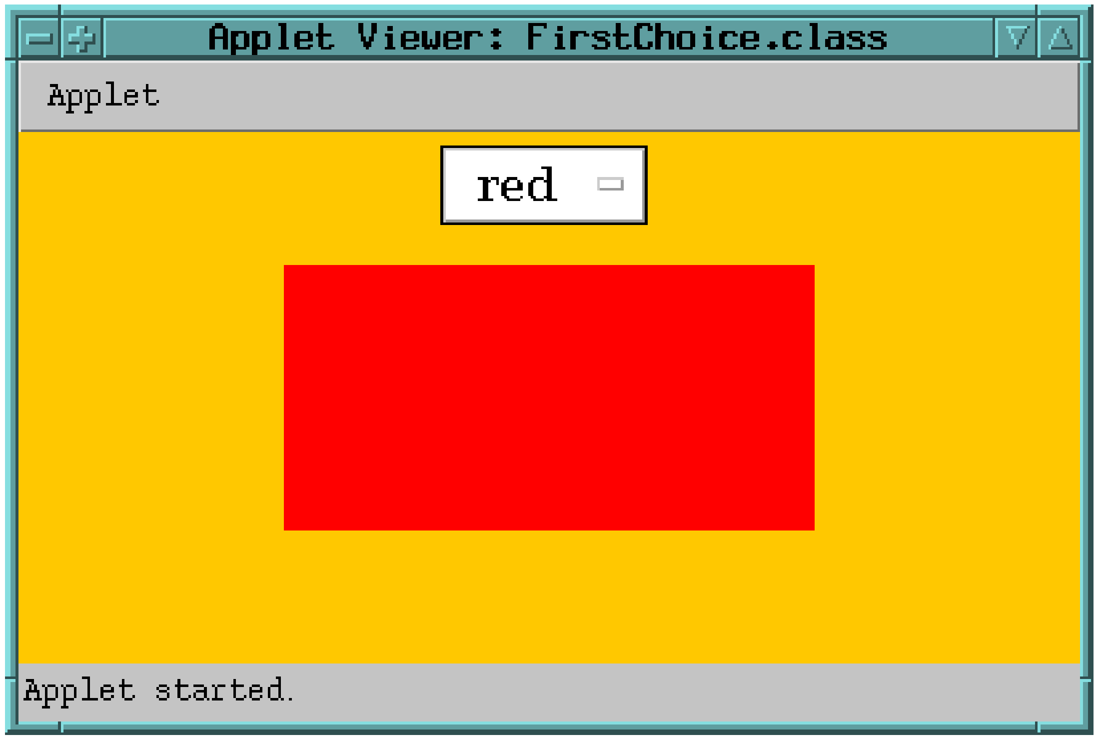
- `public Choice();` legt ein neues `Choice`-Objekt an;
- Zu diesem Objekt können beliebig viele Items hinzugefügt werden. Dazu dient die ObjektMethode:

  `public void add(String str);`
- `public void addItemListener(ItemListener listener);` registriert das Objekt `listener` für die erzeugten `ItemEvent`-Objekte.
- `ItemListener` ist ein Interface ähnlich wie `ActionListener`.
- Wieder fügen wir die neue Komponente mithilfe von `void add(Component comp)` dem Applet hinzu.

```
   ...
   public void paint(Graphics g) {
      g.setColor(color);
      g.fillRect(100,50,200,100);
   }
   public void itemStateChanged(ItemEvent e) {
      String s = (String) e.getItem();
      switch(s.charAt(0)) {
         case 'w':   color = Color.white; break;
         case 'r':   color = Color.red; break;
         case 'g':   color = Color.green; break;
         case 'b':   color = Color.blue;
      }
      repaint();
   }
} // end of Applet FirstChoice
```

- Das Interface `ItemListener` verlangt die Implementierung einer Methode `public void itemStateChanged(ItemEvent e);`
- Diese Methode ist für die Behandlung von `ItemEvent`-Objekten zuständig.
- `ItemEvent`-Objekte bieten (u.a. :-) die folgenden Methoden and:
  - `public ItemSelectable getItemSelectable();` — liefert den Selektions-Knopf;
  - `public Object getItem();` — liefert den Text des Items.
- Dann sieht das Ganze so aus:

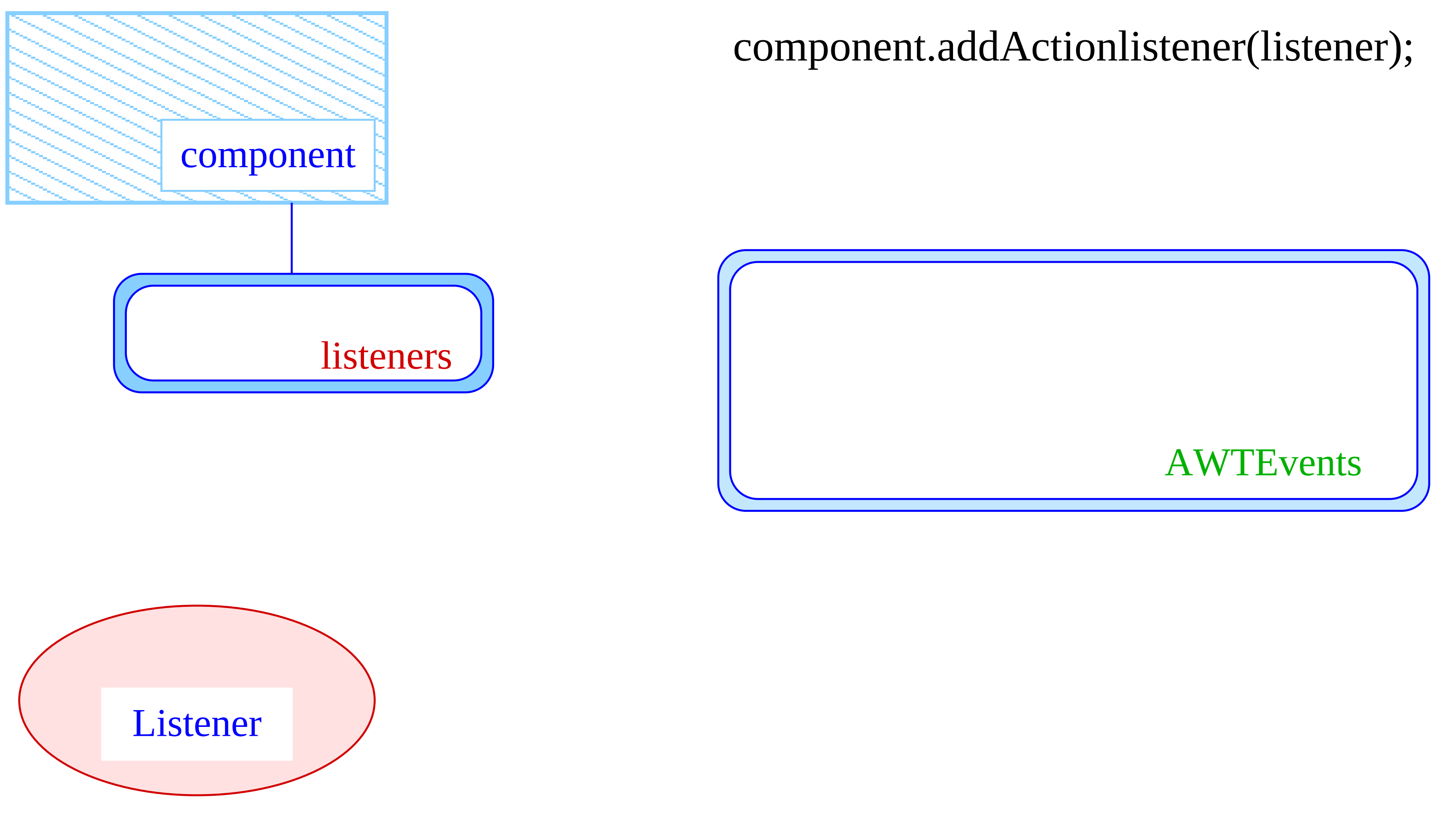

Applet Viewer: FirstChoice.class
Applet
blue
Applet started.

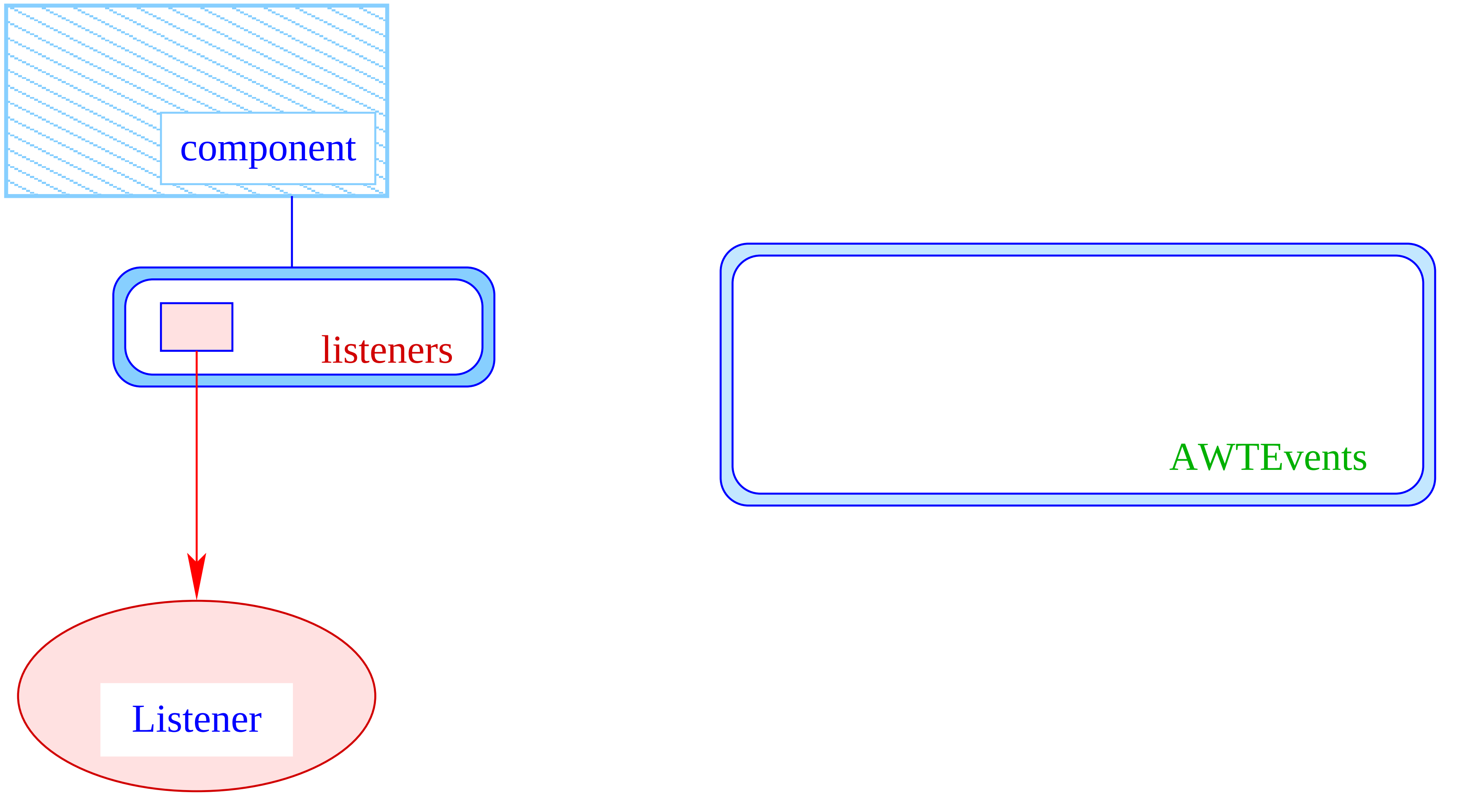

Applet Viewer: FirstChoice.class
Applet
red
Applet started.

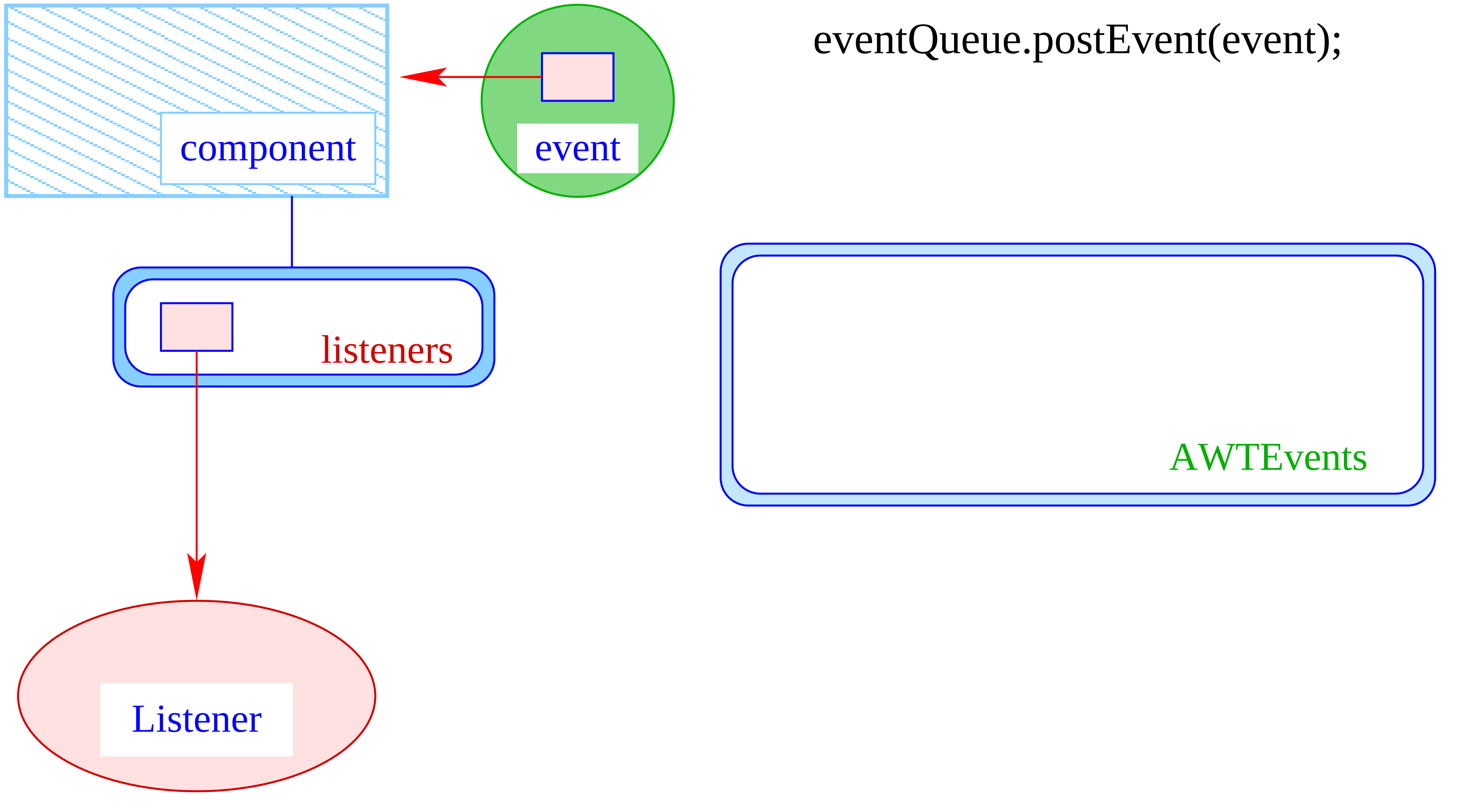
21.3 Ereignisse

- Komponenten erzeugen Ereignisse;
- Listener-Objekte werden an Komponenten für Ereignis-Klassen registriert;
- Ereignisse werden entsprechend ihrer Herkunft an Listener-Objekte weitergereicht.

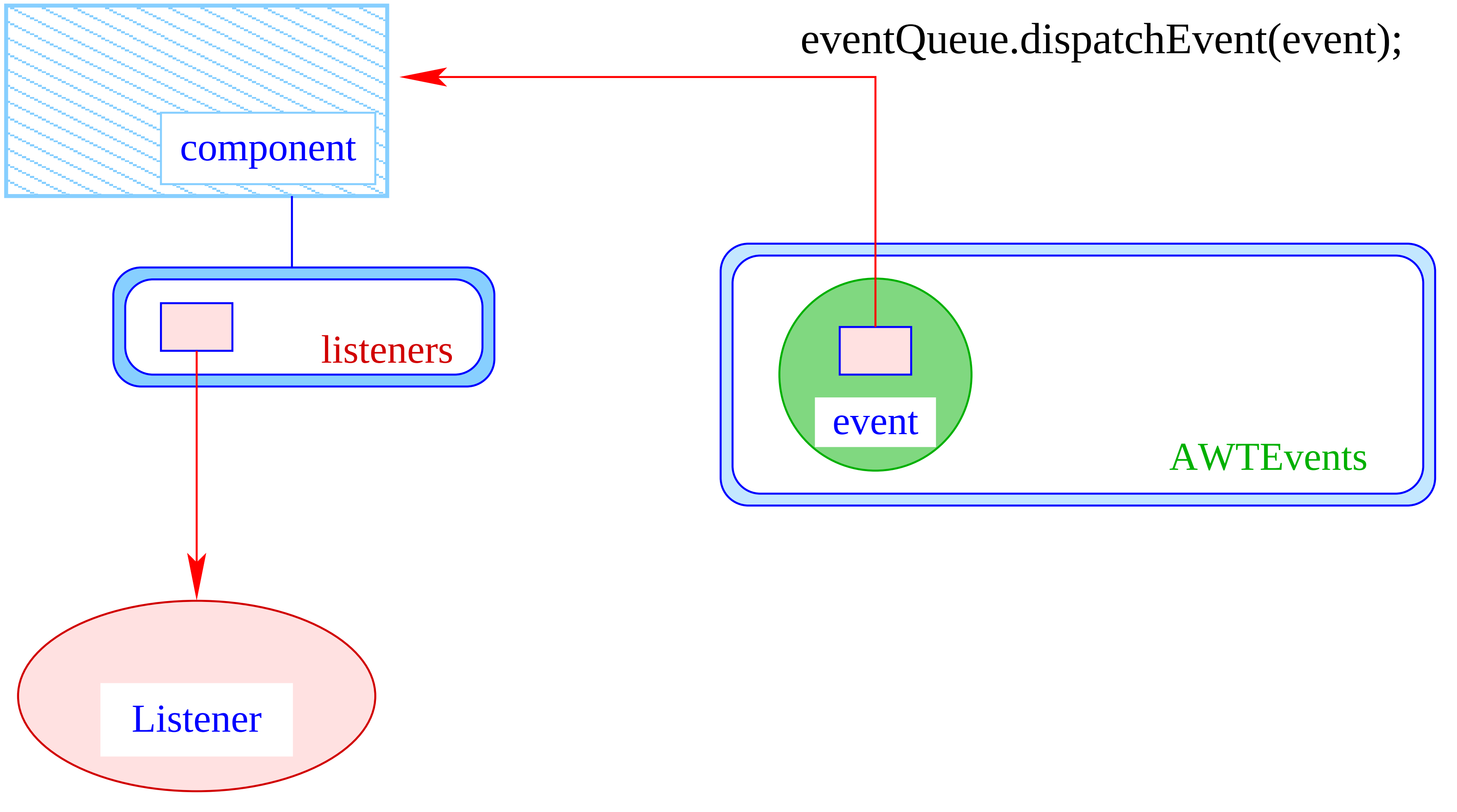
Im Detail:


component.addActionlistener(listener);
component
listeners
AWTEvents
Listener

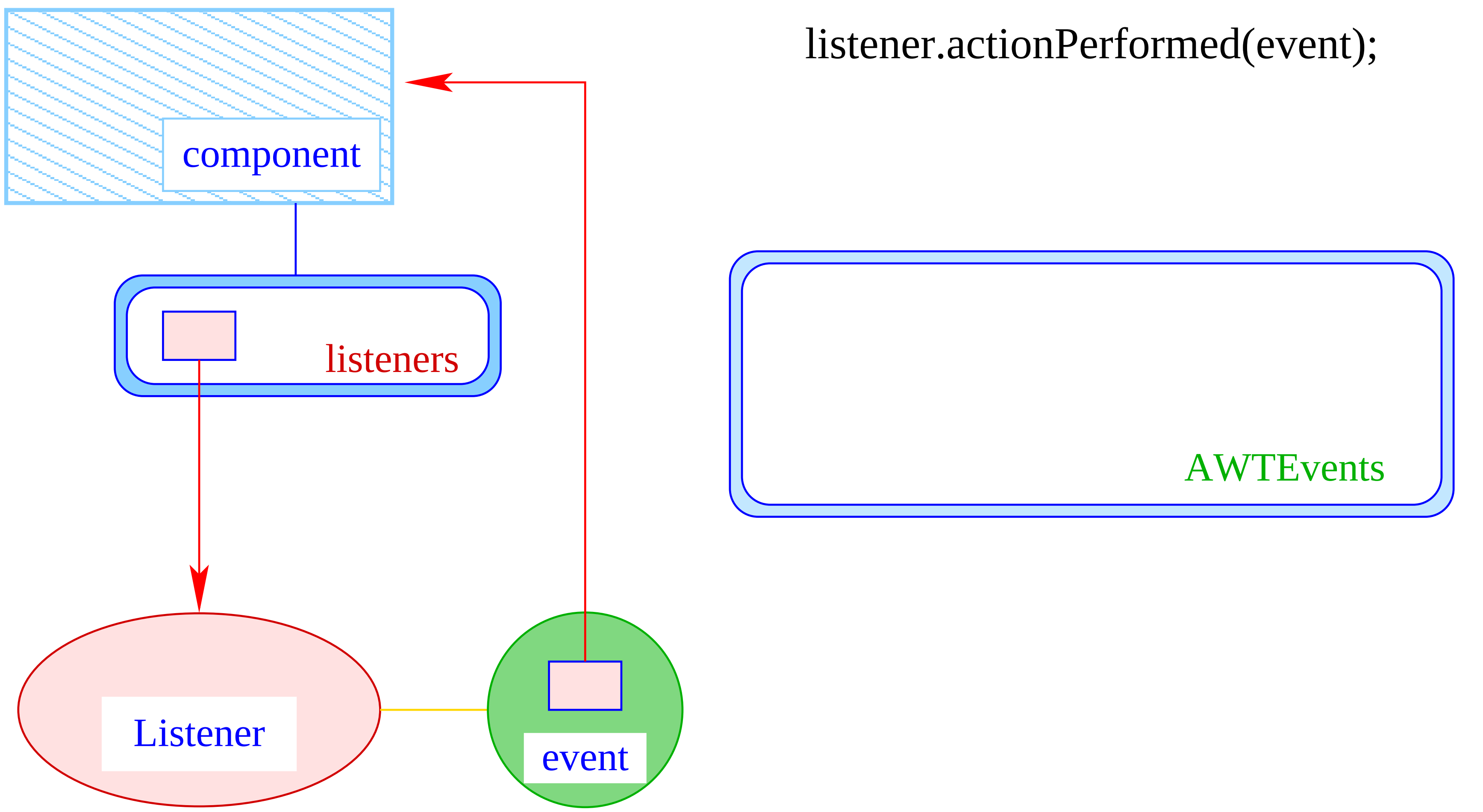

component
listeners
AWTEvents
Listener



eventQueue.postEvent(event);


eventQueue.dispatchEvent(event);
component
listeners
event
AWTEvents
Listener

listener.actionPerformed(event);



- Jedes `AWTEvent`-Objekt verfügt über eine Quelle, d.h. eine Komponente, die dieses Ereignis erzeugte.

  `public` Object `getSource();` (der Klasse `java.util.EventObject`) liefert dieses Objekt.

- Gibt es verschiedene Klassen von Komponenten, die Ereignisse der gleichen Klasse erzeugen können, werden diese mit einem geeigneten Interface zusammengefasst.

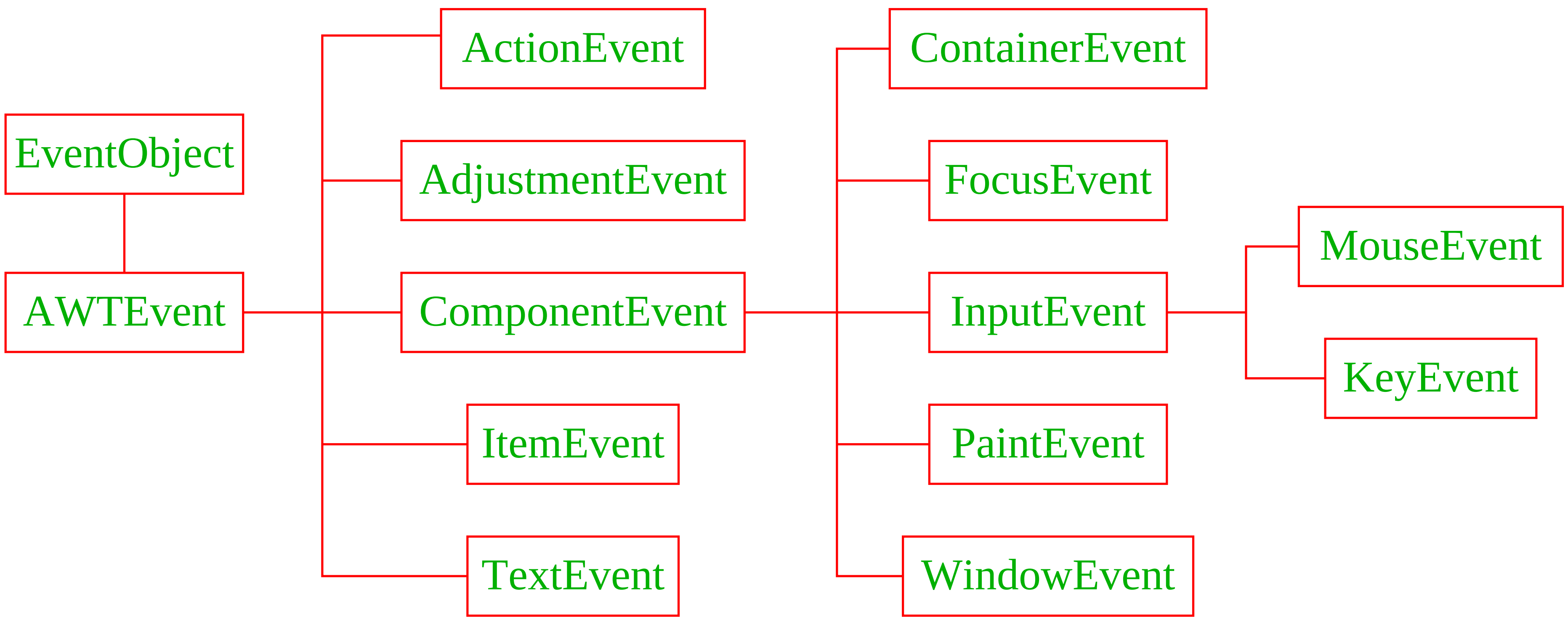
Beispiele:

Ereignis-Klasse	Interface	Objekt-Methode
`ItemEvent`	`ItemSelectable`	`ItemSelectable getItemSelectable();`
`AdjustmentEvent`	`Adjustable`	`Adjustable getAdjustable()`

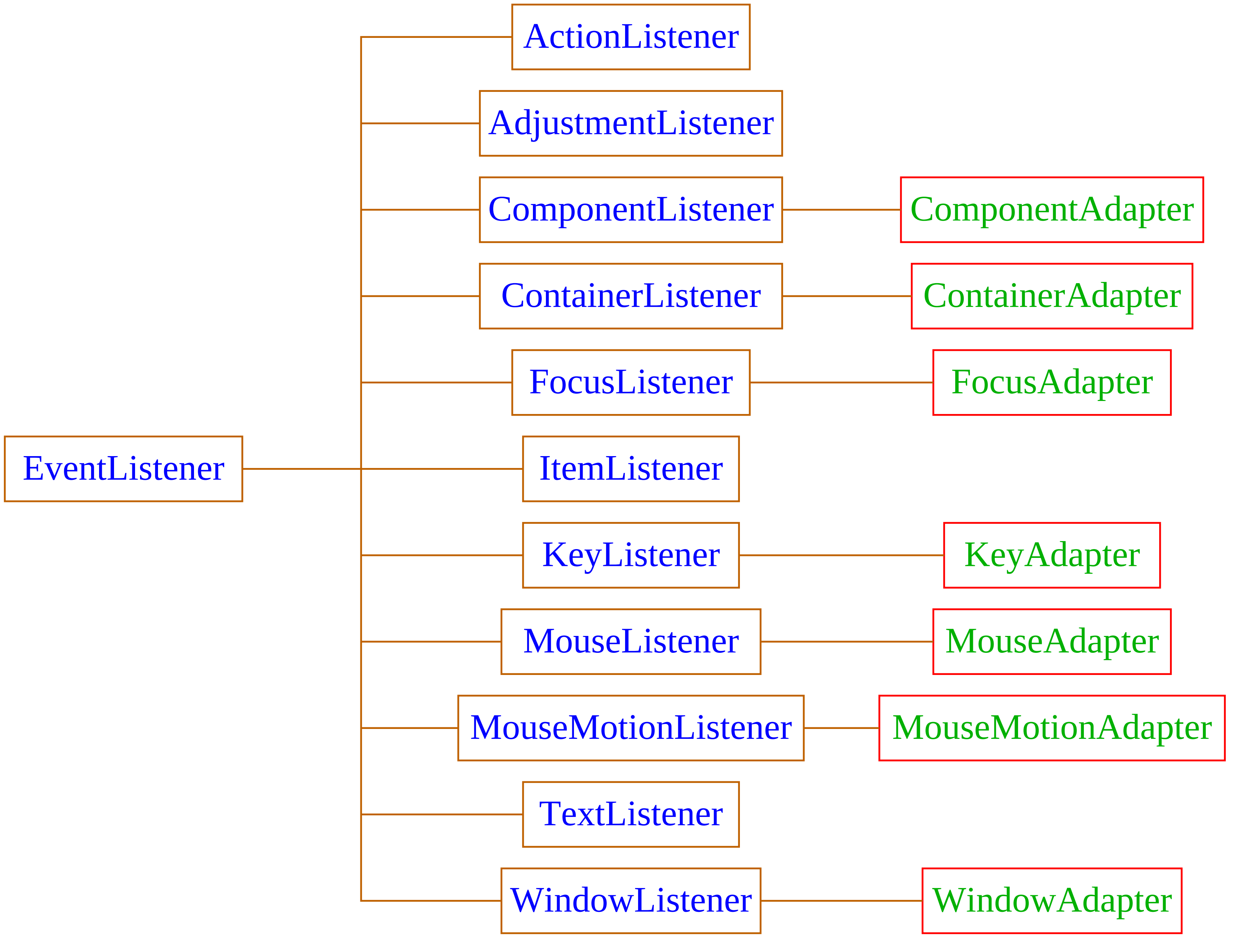
- Eine Komponente kann Ereignisse verschiedener `AWTEvent`-Klassen erzeugen.
- Für jede dieser Klassen können getrennt Listener-Objekte registriert werden ...
- Man unterscheidet zwei Sorten von Ereignissen:

  1. semantische Ereignis-Klassen — wie `ActionEvent` oder `AdjustmentEvent`;
  2. low-level Ereignis-Klassen — wie `WindowEvent` oder `MouseEvent`.

Ein Ausschnitt der Ereignis-Hierarchie ...


EventObject
AWTEvent
ActionEvent
AdjustmentEvent
ComponentEvent
ItemEvent
TextEvent
ContainerEvent
FocusEvent
InputEvent
PaintEvent
WindowEvent
MouseEvent
KeyEvent

- Zu jeder Klasse von Ereignissen gehört ein Interface, das die zuständigen Listener-Objekte implementieren müssen.
- Manche Interfaces verlangen die Implementierung mehrerer Methoden.
- In diesem Fall stellt Java Adapter-Klassen zur Verfügung.
- Die Adapterklasse zu einem Interface implementiert sämtliche geforderten Methoden auf triviale Weise ;-)
- In einer Unterklasse der Adapter-Klasse kann man sich darum darauf beschränken, nur diejenigen Methoden zu implementieren, auf die man Wert legt.


EventListener
ActionListener
AdjustmentListener
ComponentListener
ComponentAdapter
ContainerListener
ContainerAdapter
FocusListener
FocusAdapter
ItemListener
KeyListener
KeyAdapter
MouseListener
MouseAdapter
MouseMotionListener
MouseMotionAdapter
TextListener
WindowListener
WindowAdapter

Beispiel: ein `MouseListener`

- Das Interface `MouseListener` verlangt die Implementierung der Methoden:
  - `void mousePressed(MouseEvent e);`
  - `void mouseReleased(MouseEvent e);`
  - `void mouseEntered(MouseEvent e);`
  - `void mouseExited(MouseEvent e);`
  - `void mouseClicked(MouseEvent e);`
- Diese Methoden werden bei den entsprechenen Maus-Ereignissen der Komponente aufgerufen.
- Unser Beispiel-Applet soll bei jedem Maus-Klick eine kleine grüne Kreisfläche malen ...

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
class MyMouseListener extends MouseAdapter {
   private Graphics gBuff;
   private Applet app;
   public MyMouseListener(Graphics g, Applet a) {
      gBuff = g; app = a;
   }
   public void mouseClicked(MouseEvent e) {
      int x = e.getX(); int y = e.getY();
      gBuff.setColor(Color.green);
      gBuff.fillOval(x-5,y-5,10,10);
      app.repaint();
   }
}  // end of class MyMouseListener
```

- Wir wollen nur die Methode `mouseClicked()` implementieren. Darum definieren wir unsere `MouseListener`-Klasse `MyMouseListener` als Unterklasse der Klasse `MouseAdapter`.
- Die `MouseEvent`-Methoden:

  ```
  public int getX();    public int getY();
  ```

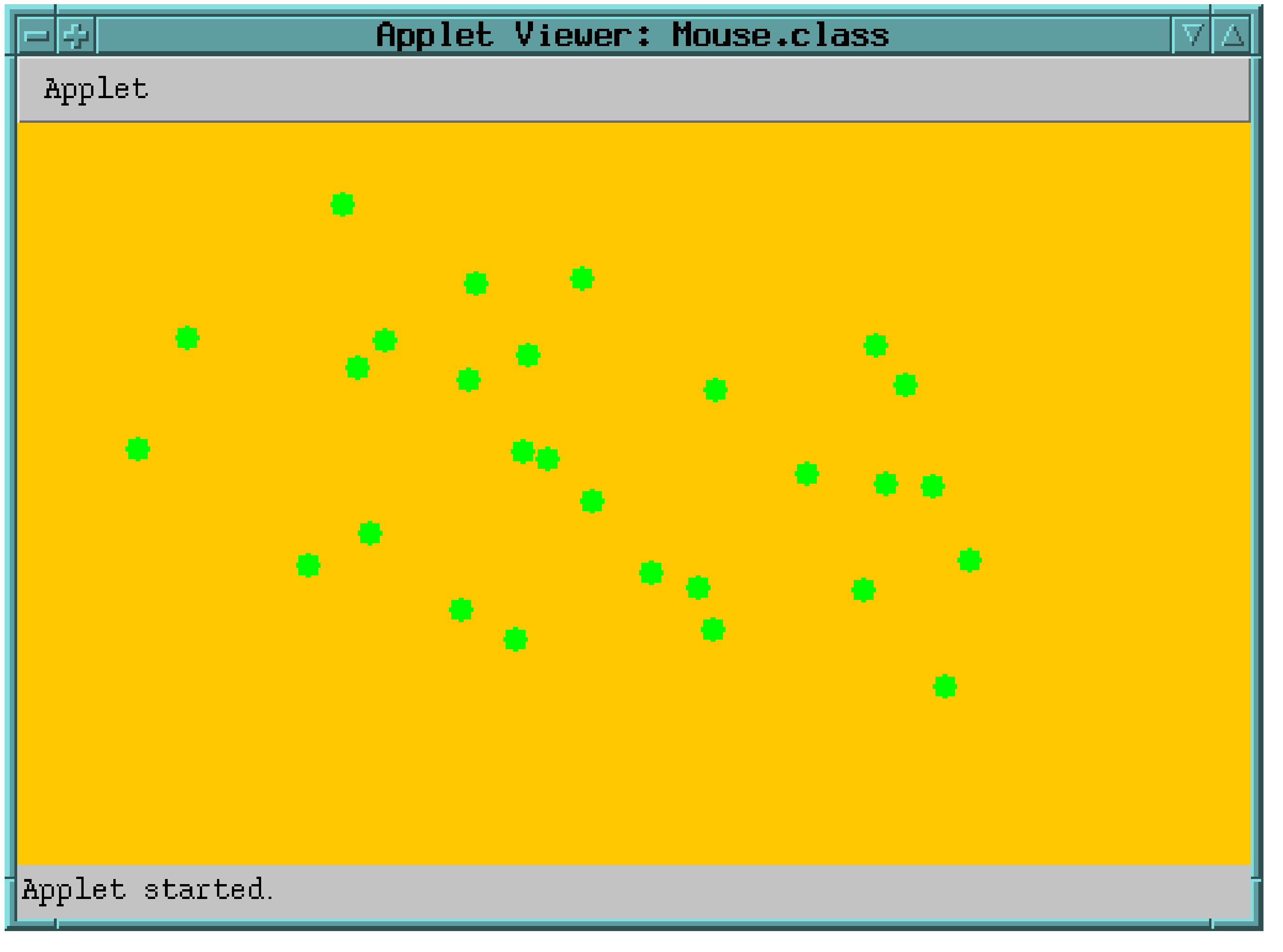
  liefern die Koordinaten, an denen der Mouse-Klick erfolgte ...
- an dieser Stelle malen wir einen gefüllten Kreis in den Puffer.
- Dann rufen wir für das Applet die Methode `repaint()` auf, um die Änderung sichtbar zu machen ...

```
public class Mouse extends Applet  {
   private Image buffer; private Graphics gBuff;
   public void init() {
      buffer = createImage(500,300);
      gBuff = buffer.getGraphics();
      addMouseListener(new MyMouseListener(gBuff,this));
   }
   public void start() {
      gBuff.setColor(Color.orange);
      gBuff.fillRect(0,0,500,300);
   }
   public void paint(Graphics page) {
      page.drawImage(buffer,0,0,500,300,this);
   }
} // end of class Mouse
```

- Die Methode `init()` legt den Puffer an, in dem die kleinen grünen Scheiben gemalt werden. Dann erzeugt sie ein `MyMouseListener`-Objekt und registriert es als `MouseListener` des Applets.
- Die Methode `start()` malt den Puffer orange.
- Die Methode `paint()` überträgt den Puffer auf die Applet-fläche.

... der Effekt:

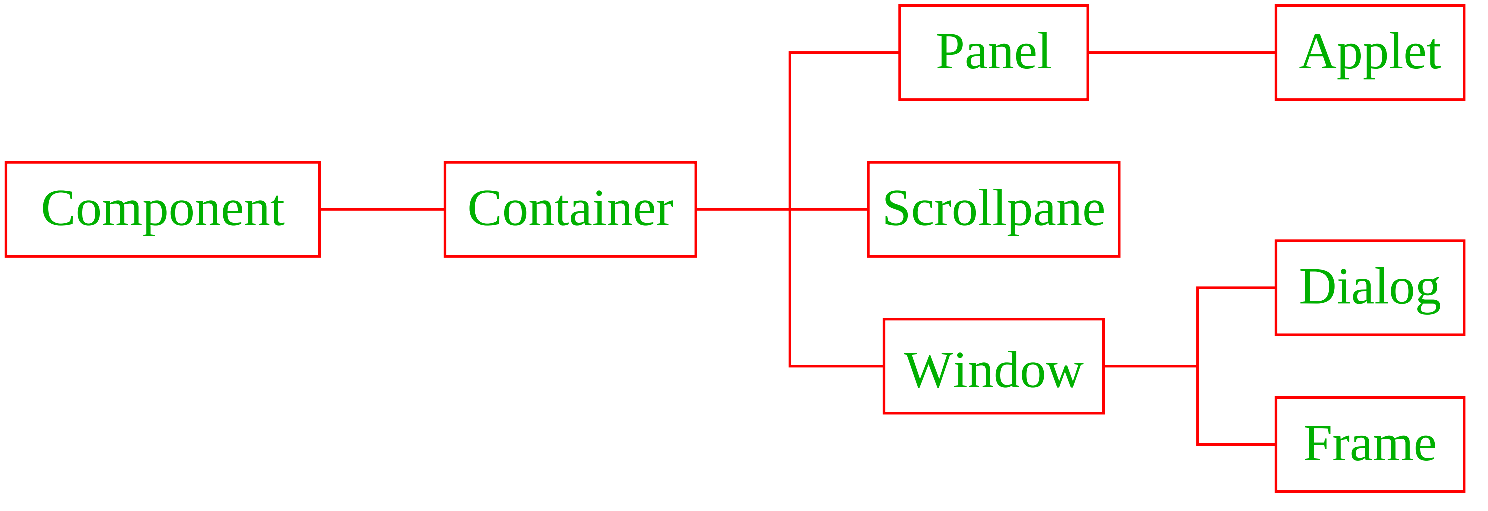

Applet Viewer: Mouse.class
Applet
Applet started.

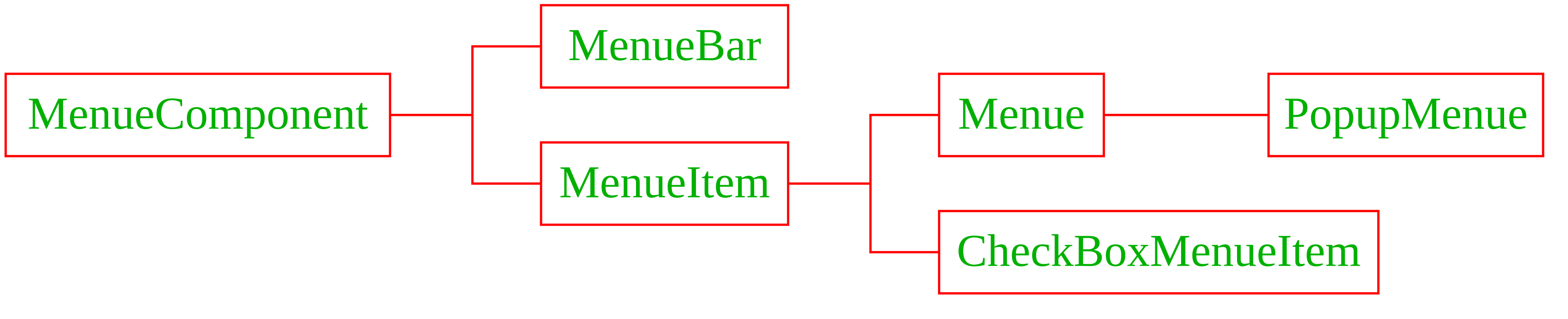

Applet Viewer: Mouse.class
Applet
Applet started.

21.4 Schachtelung von Komponenten

- Komponenten, die andere Komponenten aufnehmen können, heißen Container.
- Der LayoutManager des Containers bestimmt, wie Komponenten innerhalb eines Containers angeordnet werden.

Ein Ausschnitt der `Container`-Hierarchie:


Component
Container
Panel
Applet
Scrollpane
Window
Dialog
Frame


MenueComponent
MenueBar
MenueItem
Menue
PopupMenue
CheckBoxMenueItem

Container: Abstrakte Oberklasse aller Komponenten, die andere als Bestandteil enthalten können.

Panel: Konkrete `Container`-Klasse zum Gruppieren von Komponenten.

Applet: Unterklasse von `Panel` für das Internet.

Window: Ein nackter `Container` zur Benutzung in normalen Programmen. Kein Rand, kein Titel, keine Menue-Leiste.

Frame: Ein `Window` mit Rand und Titel-Zeile. Unterstützt die Benutzung von Menues.

Dialog: Spezielles `Window`, das sämtlichen sonstigen Benutzer-Input blockieren kann, bis das `Dialog`-Fenster geschlossen wurde.

Beispiel: Das `Basic Applet` als `Frame`

- Statt der Klasse `Applet` benutzen wir die (Ober-)Klasse `Panel`.

  Der Grund: `Applet` ist eine Unterklasse – allerdings mit zusätzlichen Multimedia-Features, über die `Panel` nicht verfügt – wie z.B. Bilder aus dem internet zu laden.

  Indem wir nur `Panel`-Methoden zulassen, garantieren wir, dass die Extra-Features nicht benutzt werden.

  Da wir nur auf eine Fläche malen wollen, würde (hier) auch ein `Canvas`-Objekt reichen.

- Das `Panel`-Objekt passen wir in einen `Frame` ein.
- Ein `Frame`-Objekt ist normalerweise unsichtbar. Um es sichtbar zu machen, rufen wir die Methode `public void setVisible(boolean b)` auf.

```
import java.awt.*;
class BasicPanel extends Panel {
   public BasicPanel() {
      setBackground(Color.orange);
   }
   public void paint(Graphics g) {
      g.setColor(Color.red);
      g.fillRect(50,50,200,100);
      g.setColor(Color.blue);
      g.fillRect(100,100,200,100);
      g.setColor(Color.green);
      g.fillRect(150,150,200,100);
   }
} // end of class BasicPanel
...
```

- Was in den Methoden `init()` bzw. `start()` passierte, erfolgt nun in den Konstruktoren des `Panel`-Objekts ...
- Der Methode `destroy()` entspricht die Methode `public void finalize();` die aufgerufen wird, wenn das Objekt freigegeben wird (deren Existenz wir bisher verschwiegen haben :-).
- Die `paint()`-Methode entspricht derjenigen des Applets und wird entsprechend automatisch aufgerufen, wenn die Fläche neu bemalt werden soll.

```
...
public class Basic extends Frame {
   public Basic(int x, int y) {
      setLocation(x,y);
      setSize(500,300);
      add(new BasicPanel());
   }
   public static void main(String[] args) {
      (new Basic(50,50)).setVisible(true);
      (new Basic(600,600)).setVisible(true);
   }
}  // end of class Basic
```

- Mithilfe der Objekt-Methoden

```
void setLocation(int x, int y);
void setSize(int width, int height);
```

  ... kann ein Fenster positioniert bzw. dimensioniert werden.

- Der Standard-`LayoutManager` der Klasse `Frame` ist `BorderLayout`. Für diesen fügt `void add(Component c)` die Komponente `c` in der Mitte ein (sind die Ränder unbesetzt, breitet sich die Mitte über die ganze Fläche aus :-)

- Die Klassen-Methode `main()` legt zwei `Basic`-Objekte an verschiedenen Stellen des Bildschirms an ...

- Der Aufruf `setVisible(true);` macht das Frame sichtbar.

- Achtung: Um auf Schließen des Fensters adequat reagieren zu können, empfiehlt es sich, einen `WindowListener` für das Frame zu implementieren !!!

Mögliche Anordnungen von Komponenten:

Frage: was passiert, wenn das Fenster (evt.) redimensioniert wird?

Frage: Wie kann man erreichen, dass das Erscheinungbild exakt unserer Vorstellung entspricht???

Flexible Proportionierung mithilfe eines `LayoutManagers`:

- Das Layout der Komponenten passt sich der Größe der zur Verfügung stehenden Fläche an :-)
- Knöpfe vergrößern sich unförmig oder wandern von einer Zeile in die nächste :-(
- Die eigene Vorstellung muss (evt. relativ mühsam) mithilfe vorgefertigter Manager realisiert werden :-{

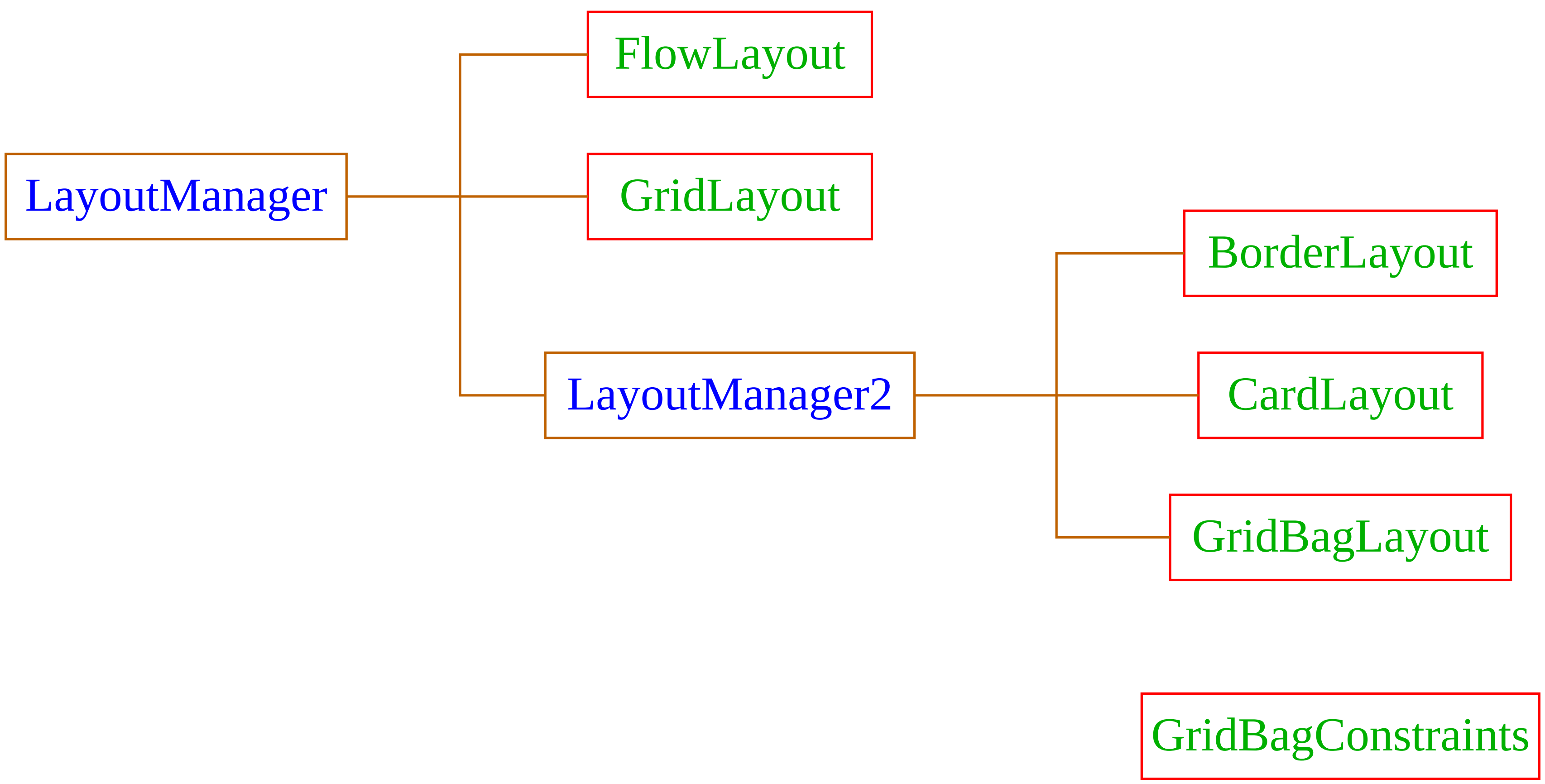
Absolute Positionierung und Dimensionierung:

- ... kennen wir bereits für Bildschirm-Fenster und graphische Elemente einer Fläche.
- Die belegte Fläche erhält bei Fenster-Deformierung einen unbenutzten Rand oder wird (teilweise) unsichtbar :-(
- Um sie für Komponenten in `Container`-Objekten zu ermöglichen, kann man mittels `setLayout(null);` den aktuellen Layout-Manager ausschalten :-)
- Sukzessive Aufrufe von `void add(Component c);` erzeugen einen Stack von Komponenten, deren Größe und Position mittels

  ```
  public void setSize(int width, int height);
  public void setLocation(int x, int y);
  ```

  ... modifiziert werden kann :-}

Einige vorgefertigte Layout-Manager:



FlowLayout: Das Default-Layout der Klasse Panel. Komponenten werden von links nach rechts zeilenweise abgelegt; passt eine Komponente nicht mehr in eine Zeile, rückt sie in die nächste.

BorderLayout: Das Default-Layout der Klasse Window. Die Fläche wird in die fünf Regionen North, South, West, East und Center aufgeteilt, die jeweils von einer Komponente eingenommen werden können.

CardLayout: Die Komponenten werden wie in einem Karten-Stapel abgelegt. Der Stapel ermöglicht sowohl den Durchgang in einer festen Reihenfolge wie den Zugriff auf spezielle Elemente.

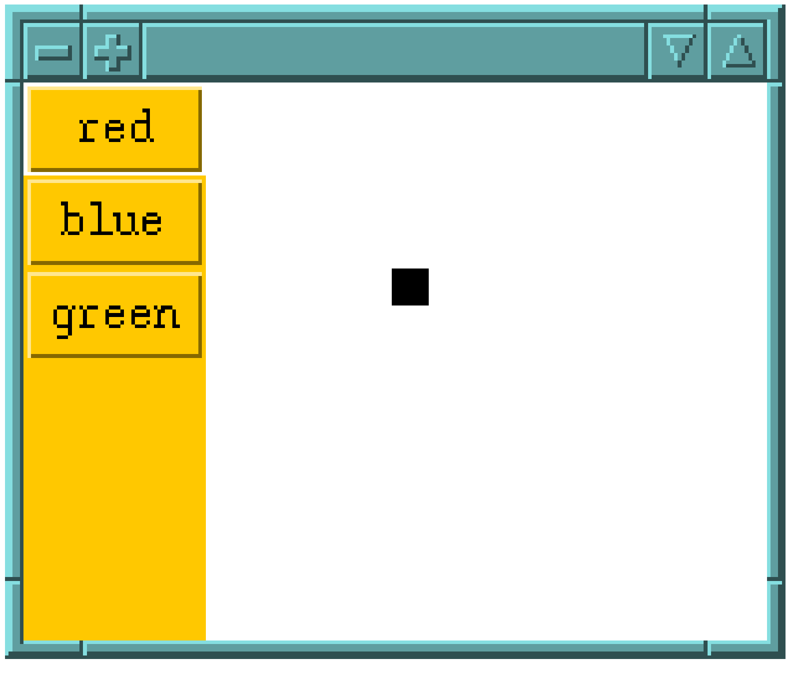
GridLayout: Die Komponenten werden in einem Gitter mit gegebener Zeilen- und Spalten-Anzahl abgelegt.

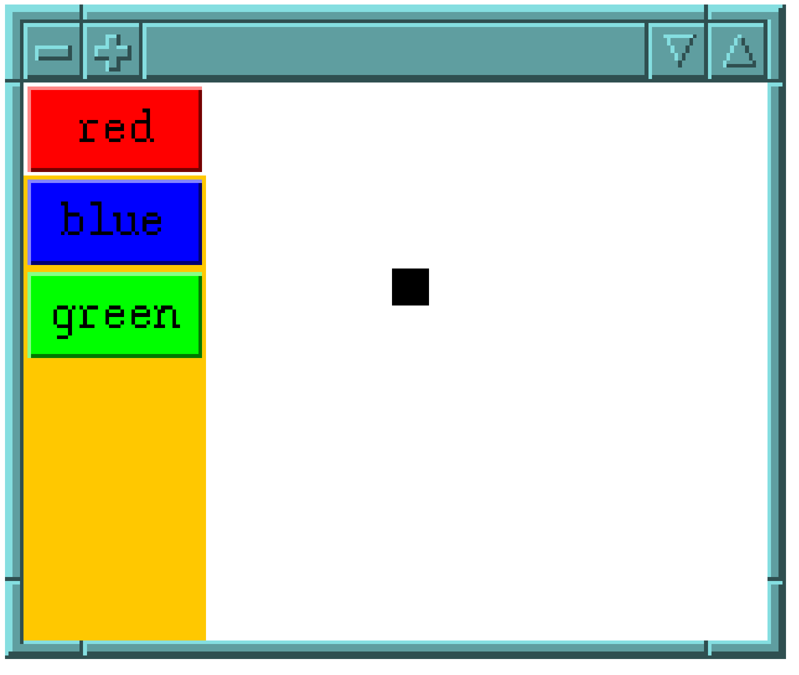
GridBagLayout: Wie `GridLayout`, nur flexibler, indem einzelne Komponenten auch mehrere Felder des Gitters belegen können.

Beispiel:

Ziel:

- Eine orange Knopfleiste am linken Rand;
- drei Knöpfe in der oberen Hälfte der Leiste;
- auf Knopf-Druck soll die Farbe des Knopfs wechseln;
- links daneben eine weiße Fläche mit einem schwarzen Quadrat
  ...


red
blue
green


red
blue
green

Implementierung:

- Die Knöpfe legen wir mittels `GridLayout` in ein `Panel`-Objekt.
- Die weiße Fläche mit schwarzem Quadrat malen wir auf ein `Canvas`-Objekt.
- Ein `Frame`-Objekt besitzt bereits das `BorderLayout`.
- Das `Panel`-Objekt legen wir im Westen des Frames ab, das `Canvas`-Objekt in der Mitte.

```
import java.awt.*;
import java.awt.event.*;
class MyPanel extends Panel implements ActionListener {
   Button b0, b1, b2;
   public MyPanel() {
      setBackground(Color.orange);
      b0 = new Button("red");
      b1 = new Button("blue");
      b2 = new Button("green");
      b0.addActionListener(this);
      b1.addActionListener(this);
      b2.addActionListener(this);
      setLayout(new GridLayout(6,1));
      add(b0); add(b1); add(b2);
   }
   ...
```

- Der Konstruktor `public GridLayout(int row, int col);` teilt die zur Verfügung stehende Fläche in ein Raster von gleich großen Feldern ein, die in `row` vielen Zeilen und `col` vielen Spalten angeordnet sind.
- die Felder werden sukzessive von links oben nach rechts unten aufgefüllt.
- Nicht alle Felder müssen tatsächlich belegt werden.

  Im Beispiel bleibt die Hälfte frei ...
- Gemeinsamer `ActionListener` für alle drei Knöpfe ist (hier) das Panel selbst ...

```
    public void actionPerformed(ActionEvent e) {
        Button b = (Button) e.getSource();
        if (b.getBackground() == Color.orange) {
            if (b == b0) b0.setBackground(Color.red);
            else if (b == b1) b1.setBackground(Color.blue);
            else b2.setBackground(Color.green);
        } else
            b.setBackground(Color.orange);
    }
}  // end class MyPanel
...
```

- Der Aufruf `e.getSource();` liefert das Objekt, das das `ActionEvent`-Objekt erzeugte, hier ein `Button`-Objekt.
- Falls die Hintergrunds-Farbe orange ist, modifizieren wir die Farbe. Ansonsten setzen wir sie auf orange zurück.

```
public class Nest extends Frame {
   public Nest() {
      setSize(200,150); setLocation(500,500);
      add(new MyPanel(), "West");
      add(new MyCanvas(), "Center");
   }
   public static void main(String[] args) {
      (new Nest()).setVisible(true);}
} // end of class Nest
class MyCanvas extends Canvas {
   public MyCanvas() { setBackground(Color.white);}
   public void paint(Graphics page) {
      page.setColor(Color.black);
      page.fillRect(50,50,10,10);
   }
} // end of class MyCanvas
```

- Ein neues `Canvas`-Objekt besitzt eigentlich Breite und Höhe 0 — und ist damit unsichtbar!
- Im `BorderLayout` wird eine Komponente jedoch auf die gesamte zur Verfügung stehende Fläche ausgedehnt.
- Wie in der Klasse `Applet` wird zum (Neu-)Malen der `Canvas`-Fläche implizit die Objekt-Methode `public void paint(Graphics page);` aufgerufen.

... Schluss mit Java!!!

Was es sonst noch so in Java gibt:

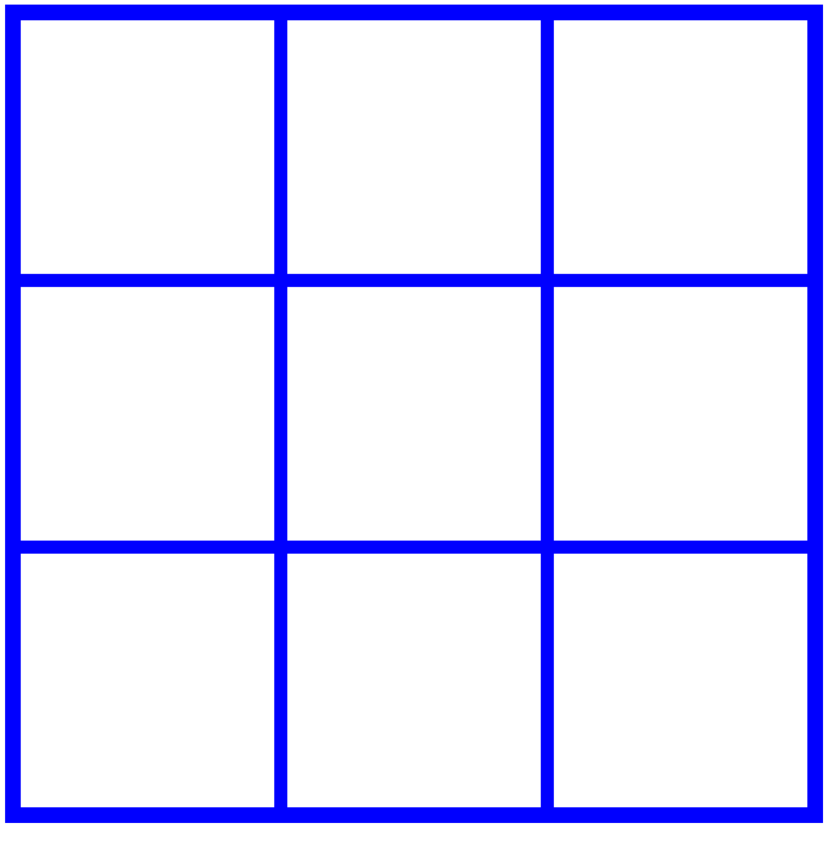
... an nützlichem:	innere Klassen;
... an mysteriösem:	Klassen zur Selbst-Reflektion;
... an comfortablem:	Malen mit Swing;
... an technischem:	Networking mit Sockets, RMI, Jini, Corba, ...
... an sonstigem:	... :-)

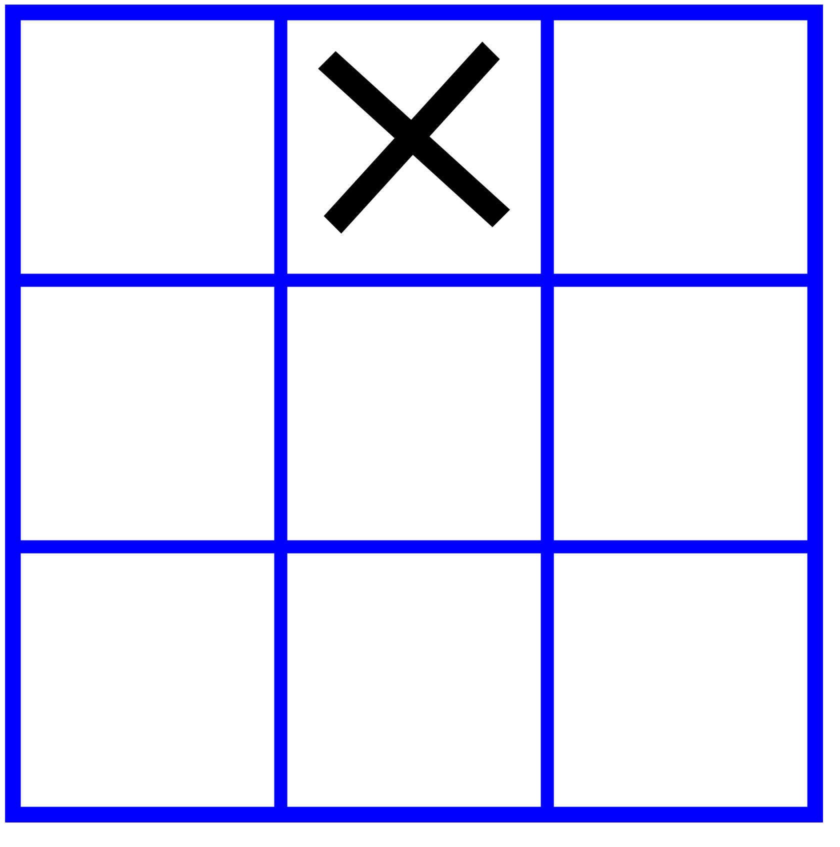
22 Tic-Tac-Toe

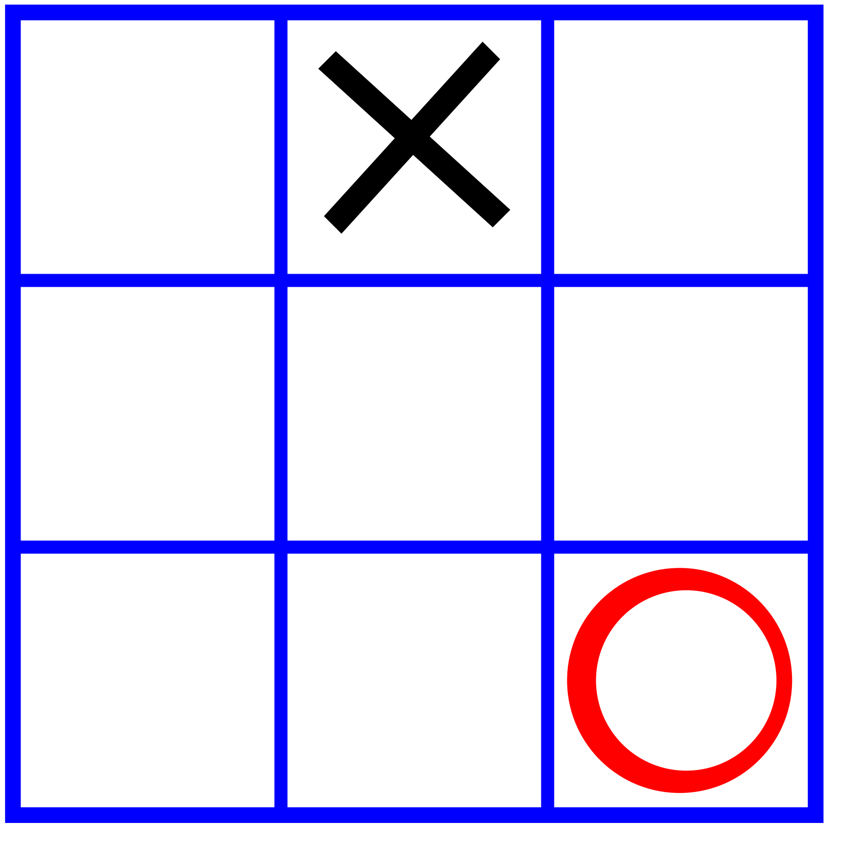
Regeln:

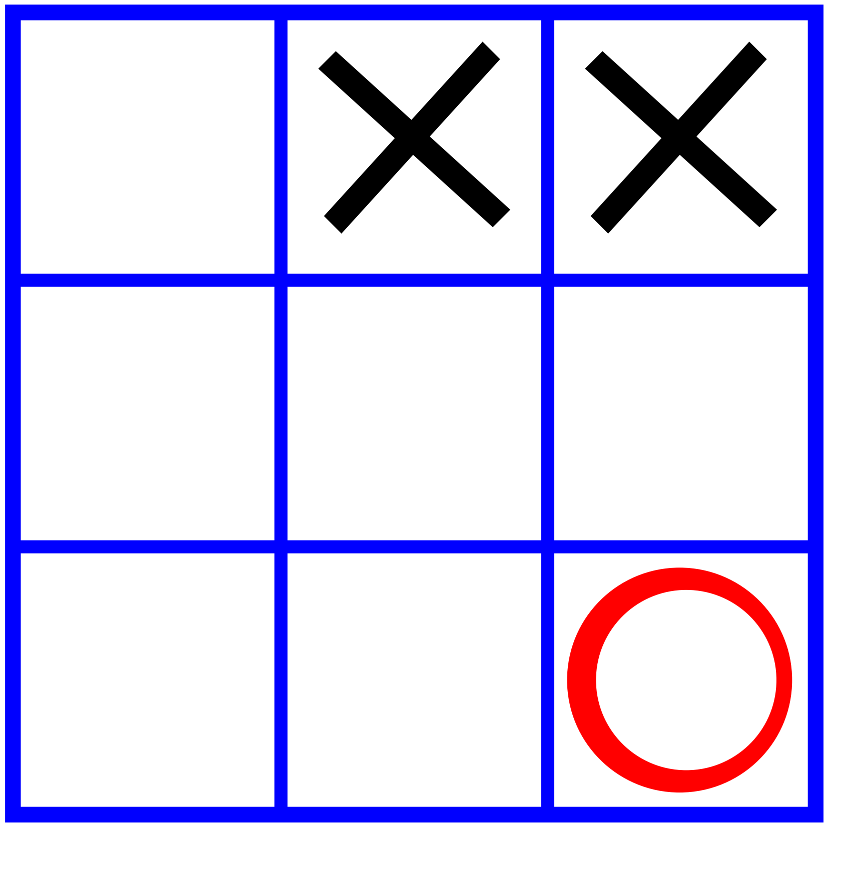
- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

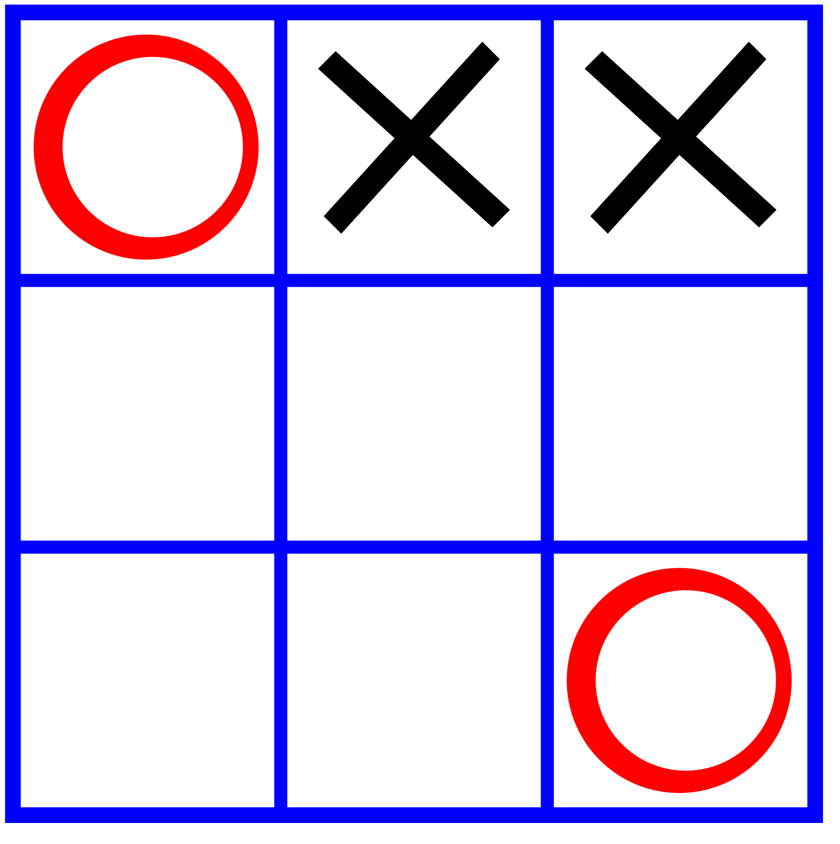
Beispiel:

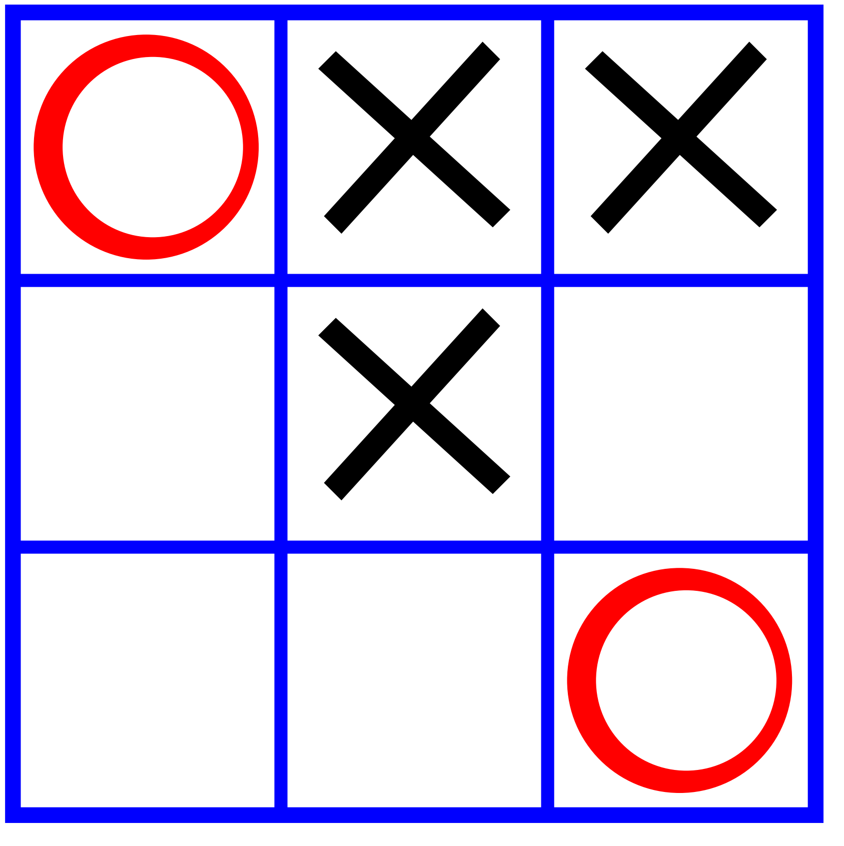


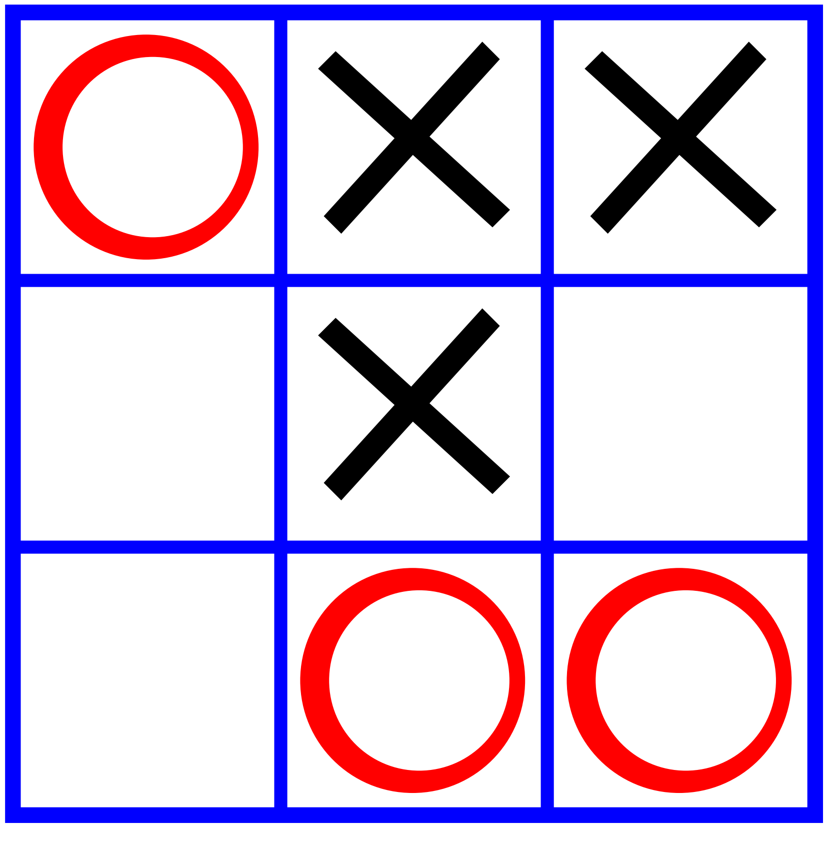


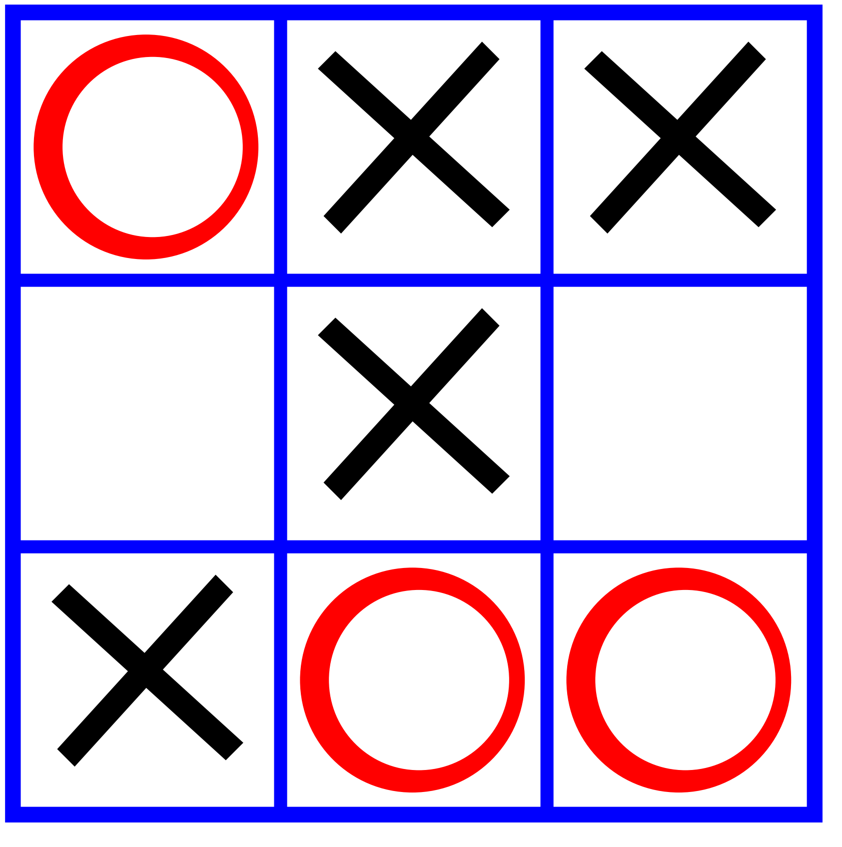












... offenbar hat die anziehende Partei gewonnen.

Fragen:

- Ist das immer so? D.h. kann die anziehende Partei immer gewinnen?
- Wie implementiert man ein Tic-Tac-Toe-Programm, das
  - ... möglichst oft gewinnt?
  - ... eine ansprechende Oberfläche bietet?

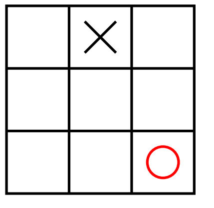
Hintergrund:

Tic-Tac-Toe ist ein endliches Zwei-Personen-Nullsummen-Spiel.
Das heißt:

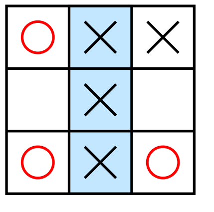
- Zwei Personen spielen gegeneinander.
- Was der eine gewinnt, verliert der andere.
- Es gibt eine endliche Menge von Spiel-Konfigurationen.
- Die Spieler ziehen abwechselnd. Ein Zug wechselt die Konfiguration, bis eine End-Konfiguration erreicht ist.
- Jede End-Konfiguration ist mit einem Gewinn aus $\mathbb{R}$ bewertet.
- Person 0 hat gewonnen, wenn eine End-Konfiguration erreicht wird, deren Gewinn negativ ist, Person 1, wenn er positiv ist.

... im Beispiel:

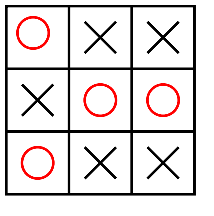
Konfiguration:



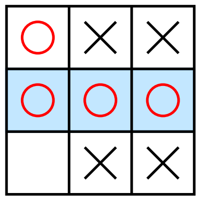
End-Konfigurationen:



Gewinn -1

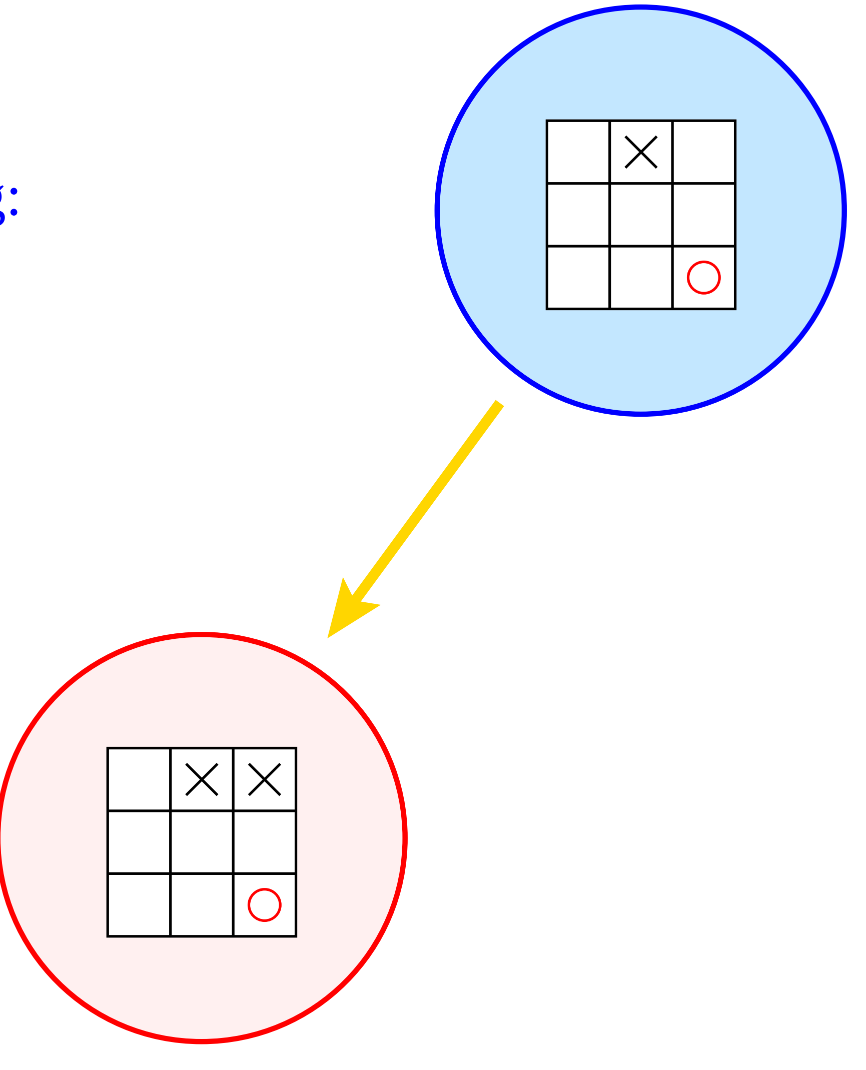


Gewinn 0

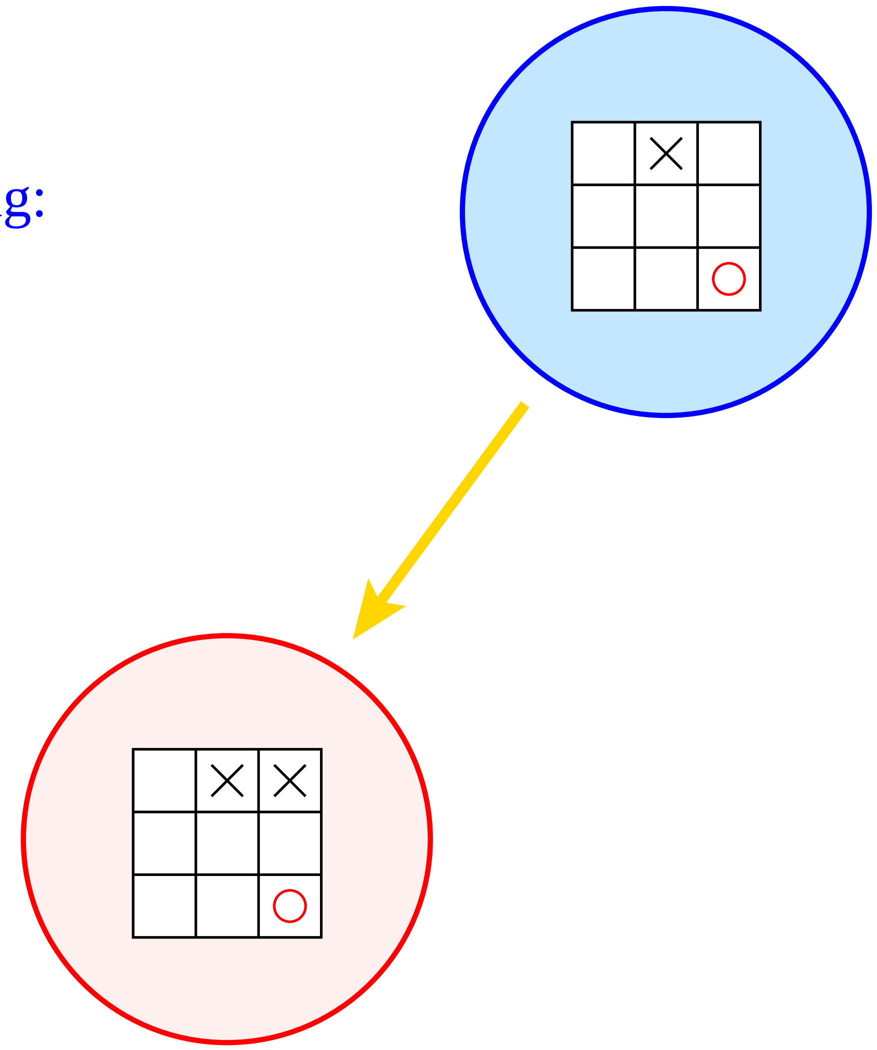


Gewinn +2

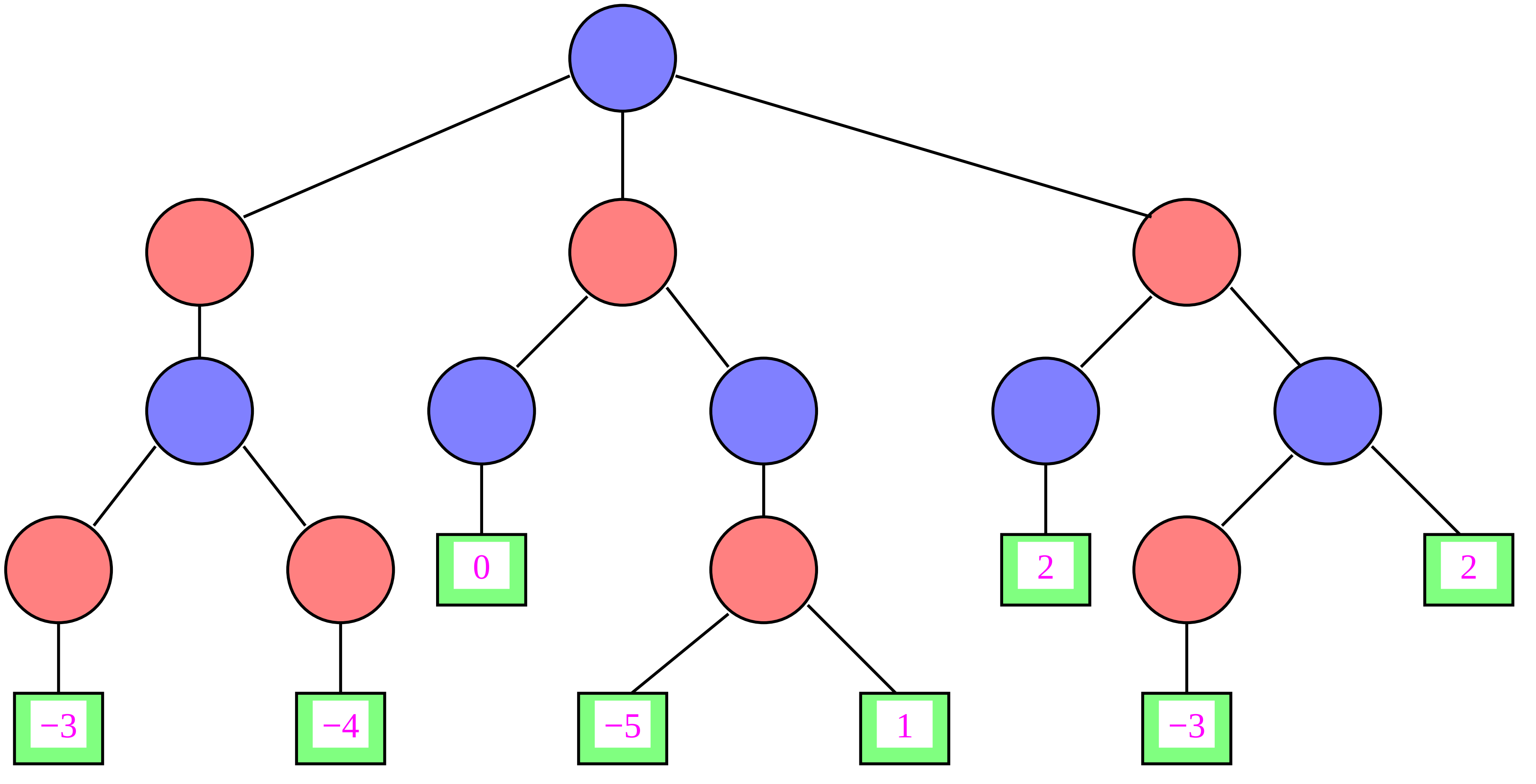
Spiel−Zug:



Spiel−Zug:



Alle möglichen Spiel-Verläufe lassen sich als ein Baum darstellen:


0
2
2
−3
−4
−5
1
−3

Knoten des Spielbaums	$==$	Konfigurationen
Kanten	$==$	Spiel-Züge
Blätter	$==$	End-Konfigurationen

Frage:

Wie finden wir (z.B. als blaue Person) uns im Spiel-Baum zurecht?

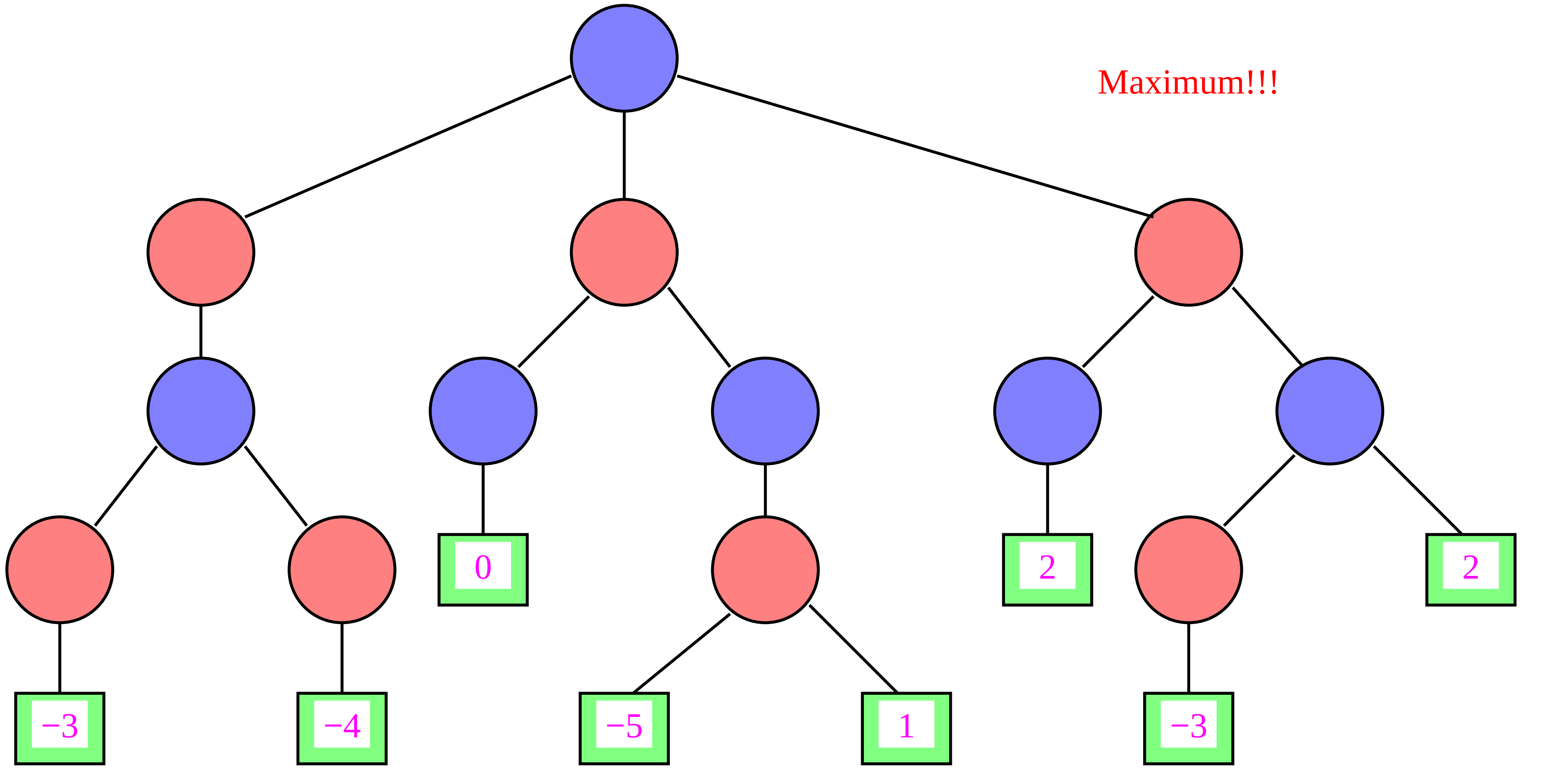
Was müssen wir tun, um sicher ein negatives Blatt zu erreichen?

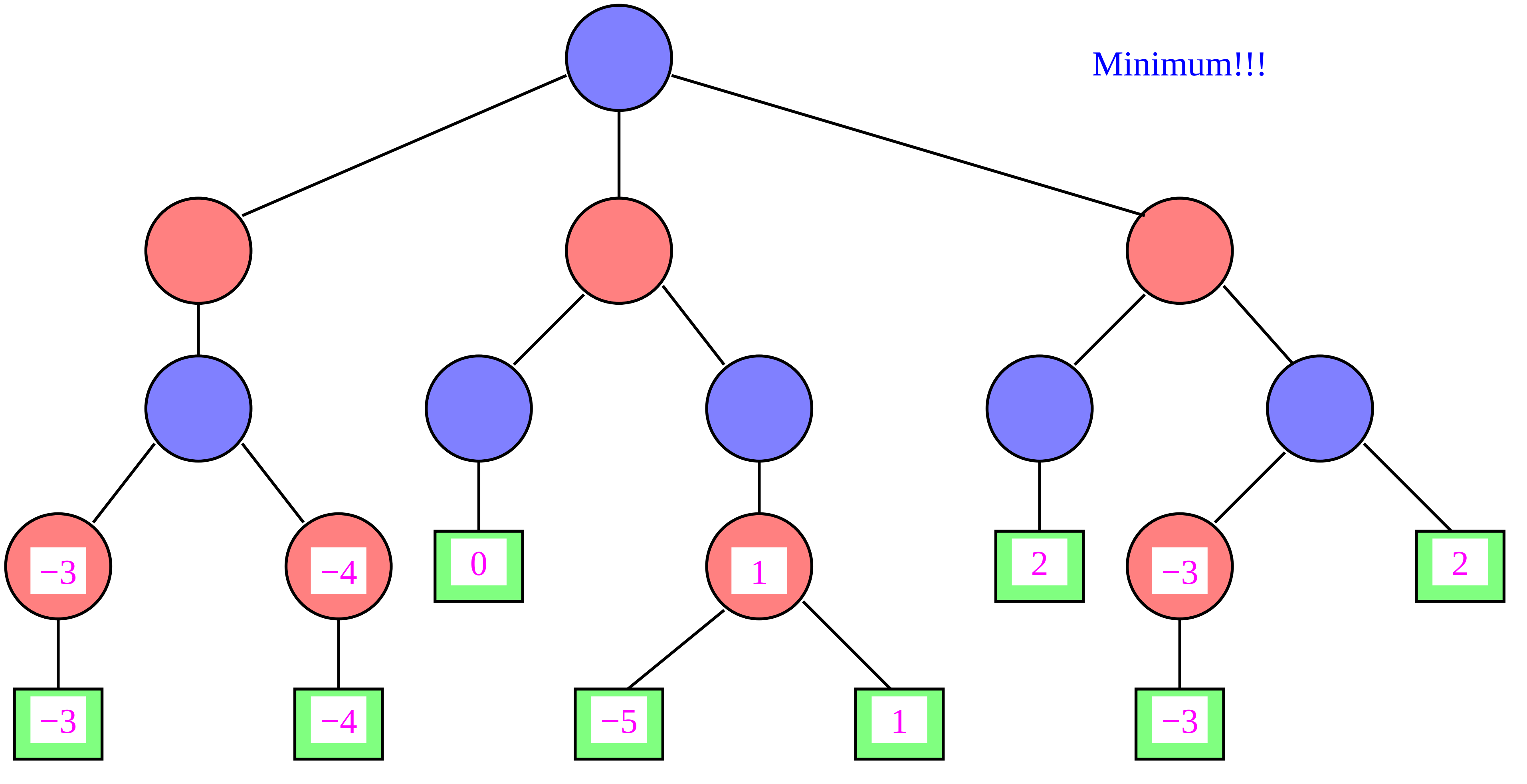
Idee:

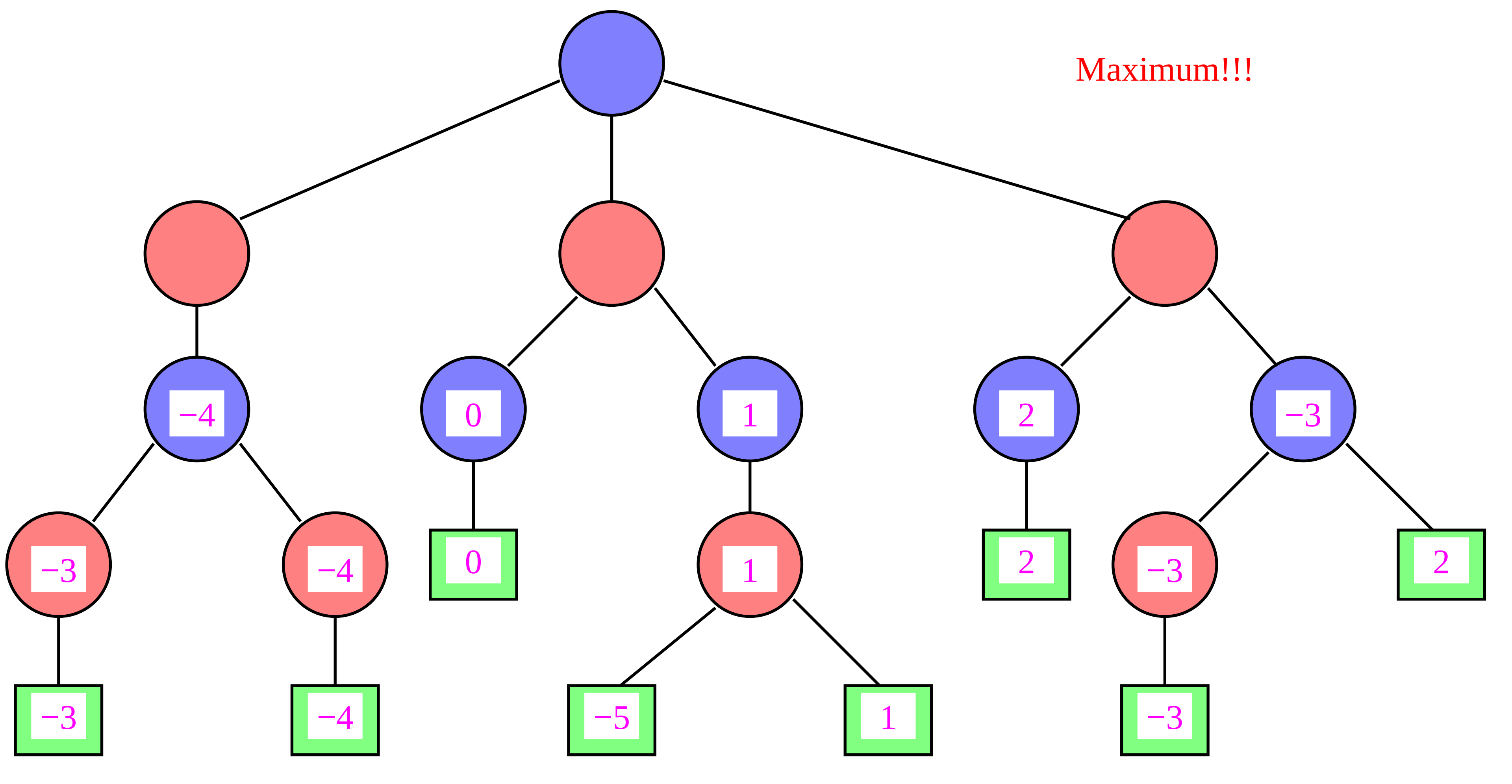
- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

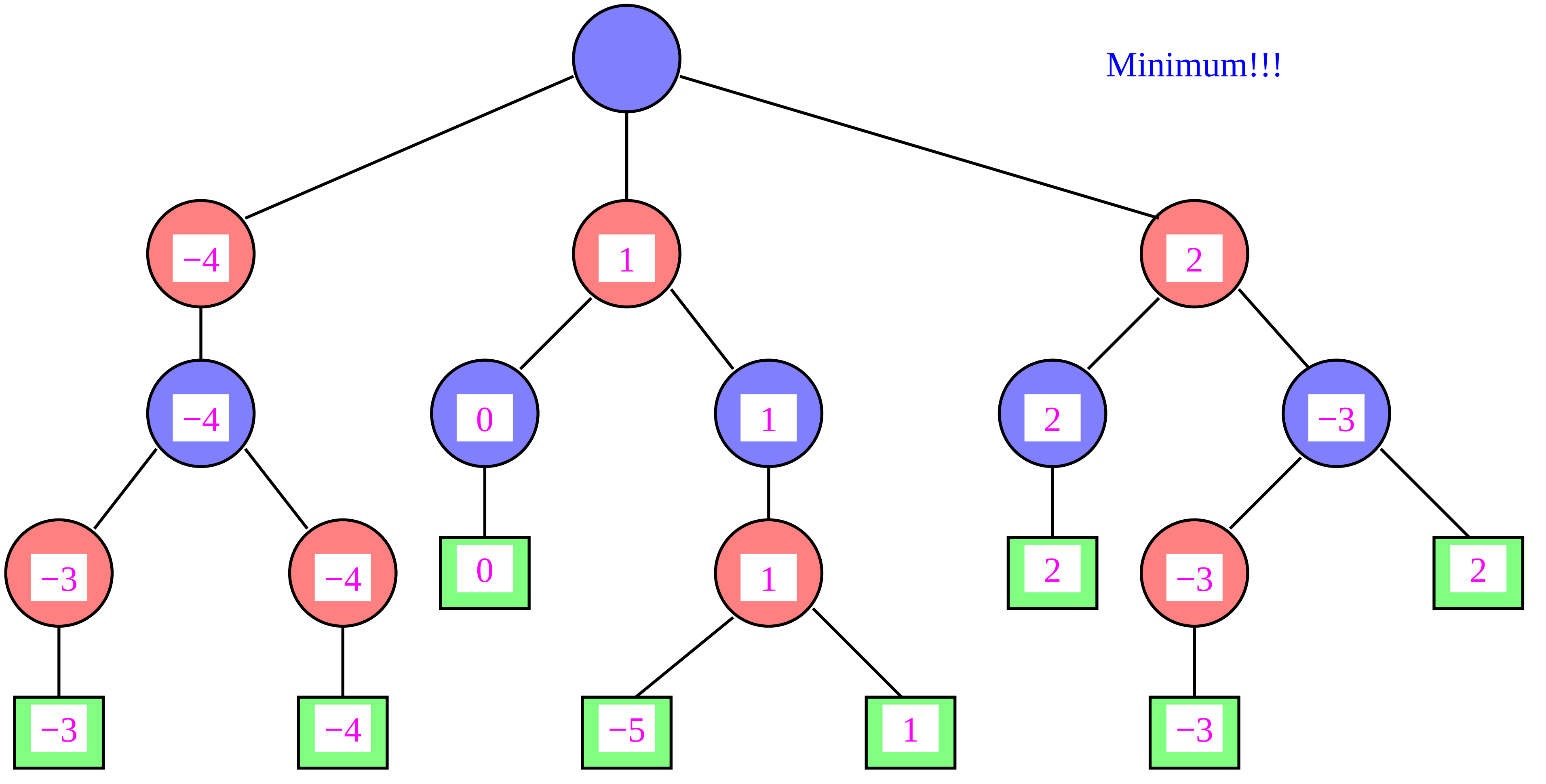
  Fall 1: Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

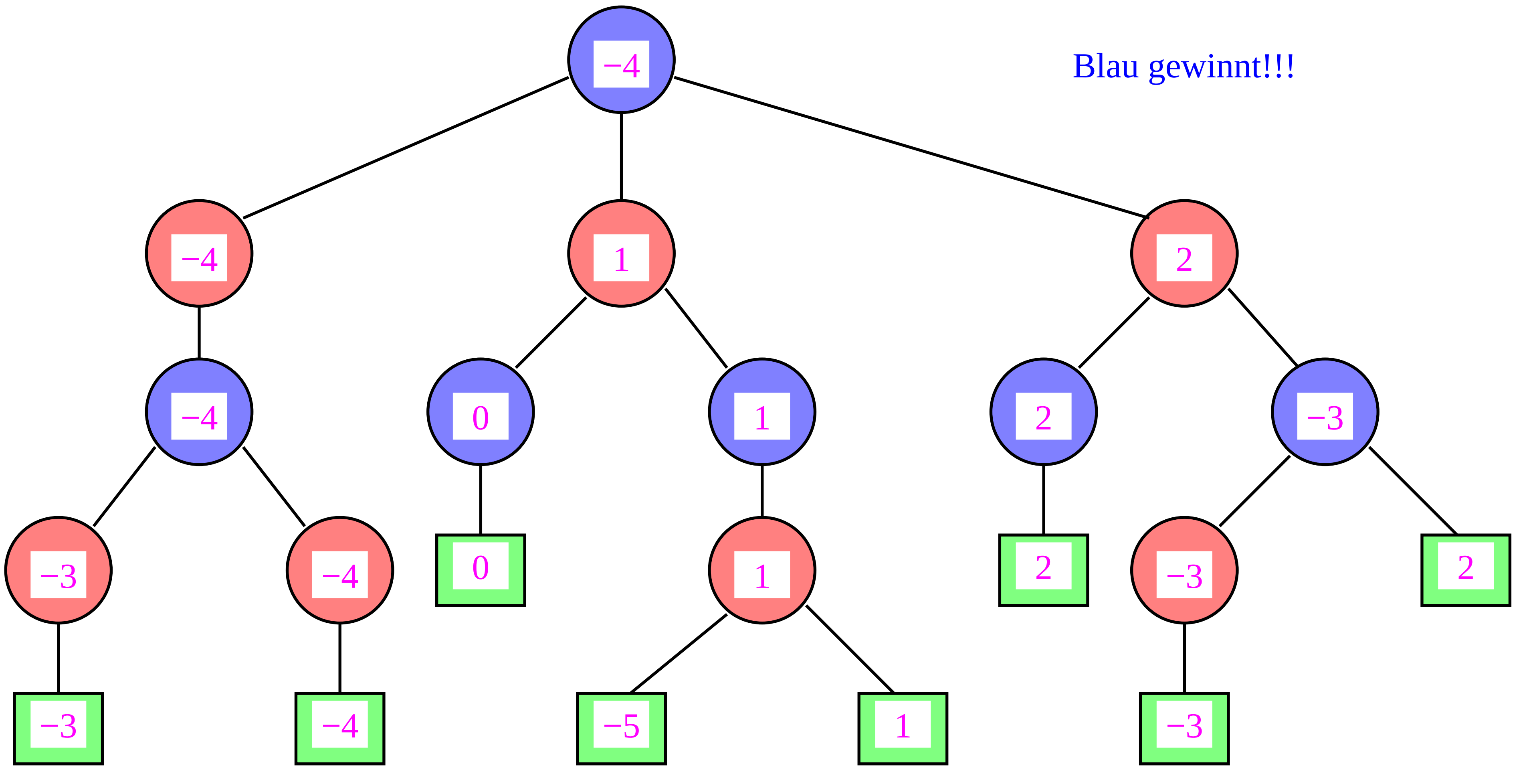
  Fall 2: Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.


Maximum!!!
0
2
2
−3
−4
−5
1
−3


Minimum!!!
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3


Maximum!!!
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3

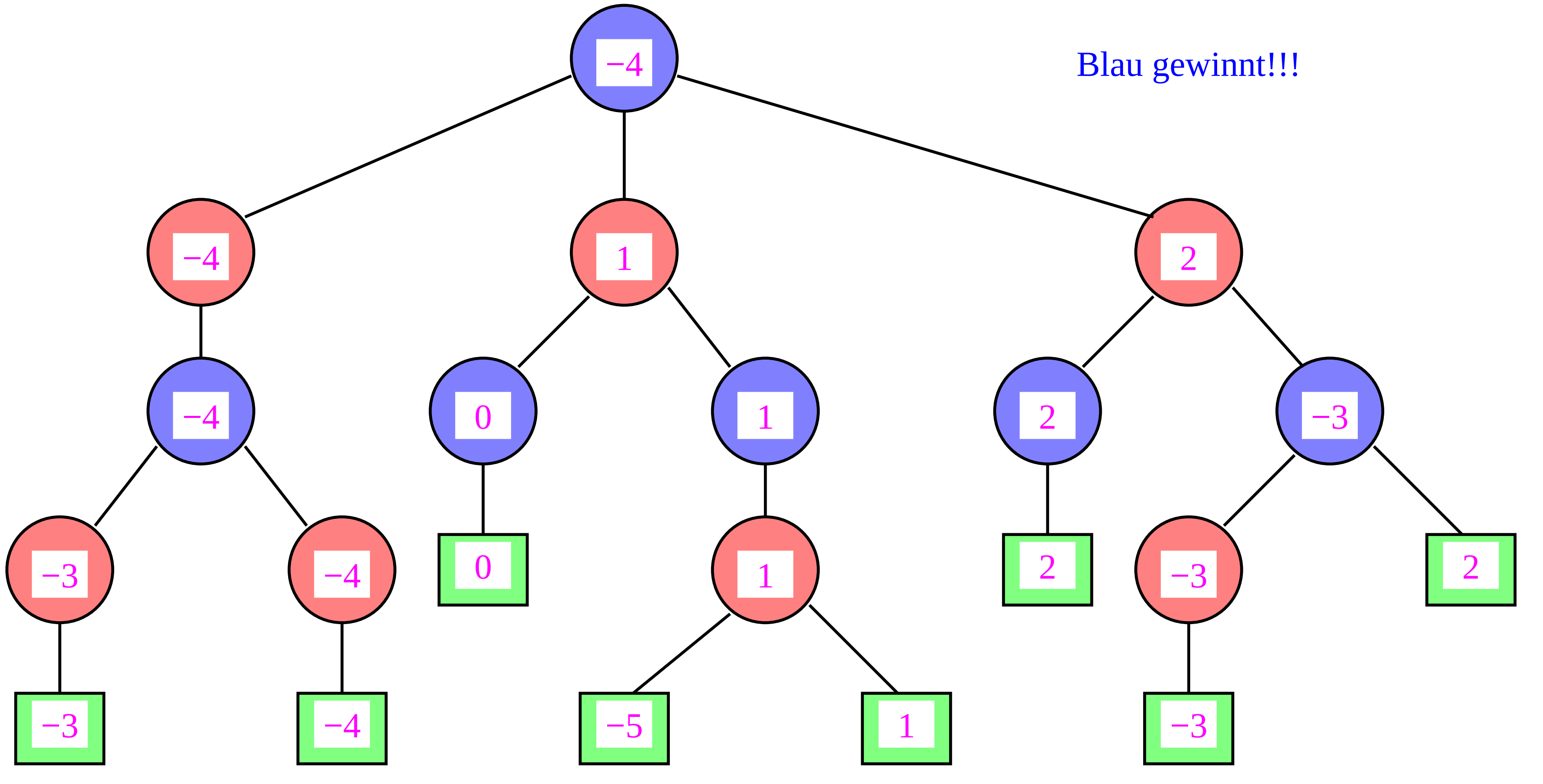

Minimum!!!
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3

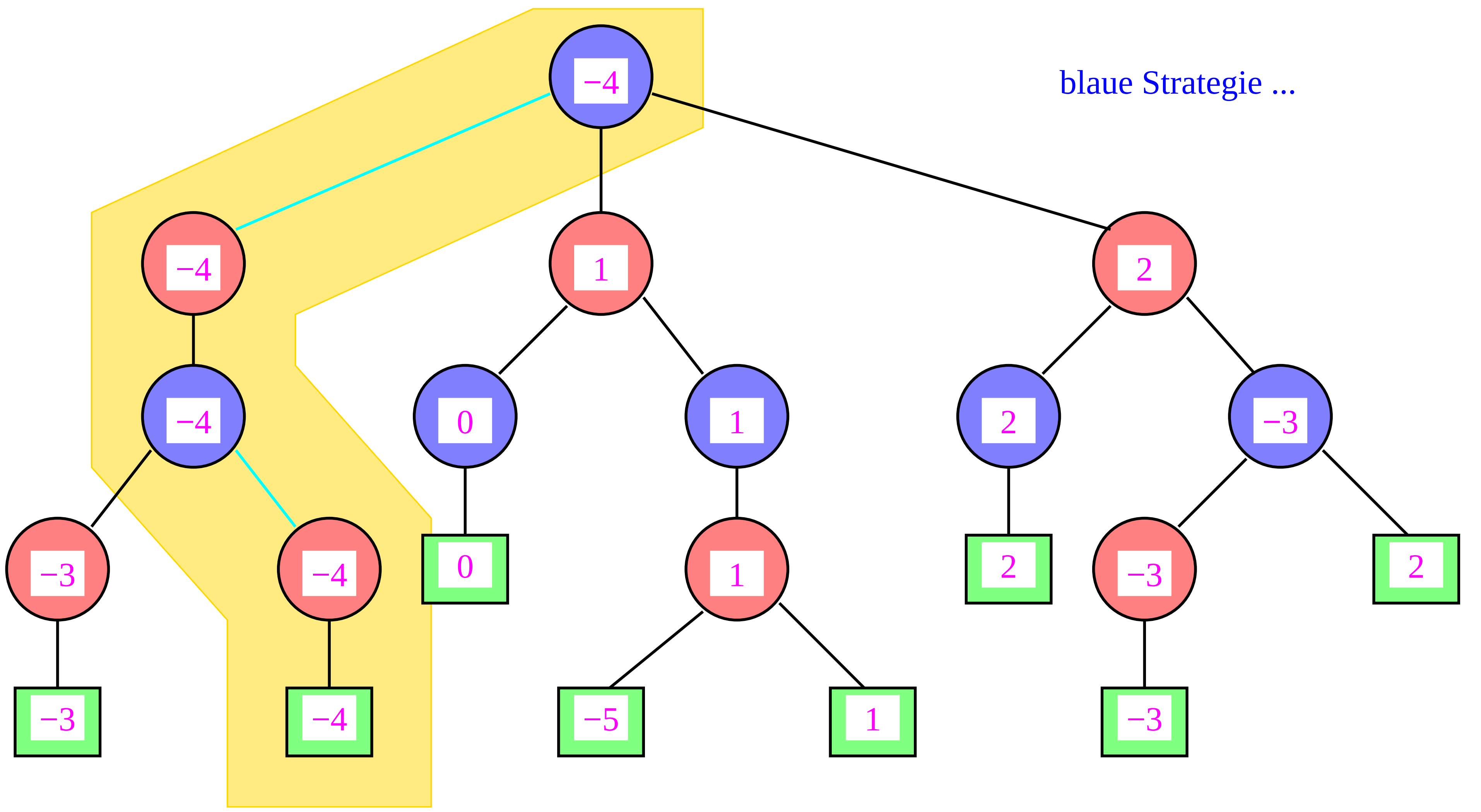

−4
Blau gewinnt!!!
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3

Eine Strategie ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

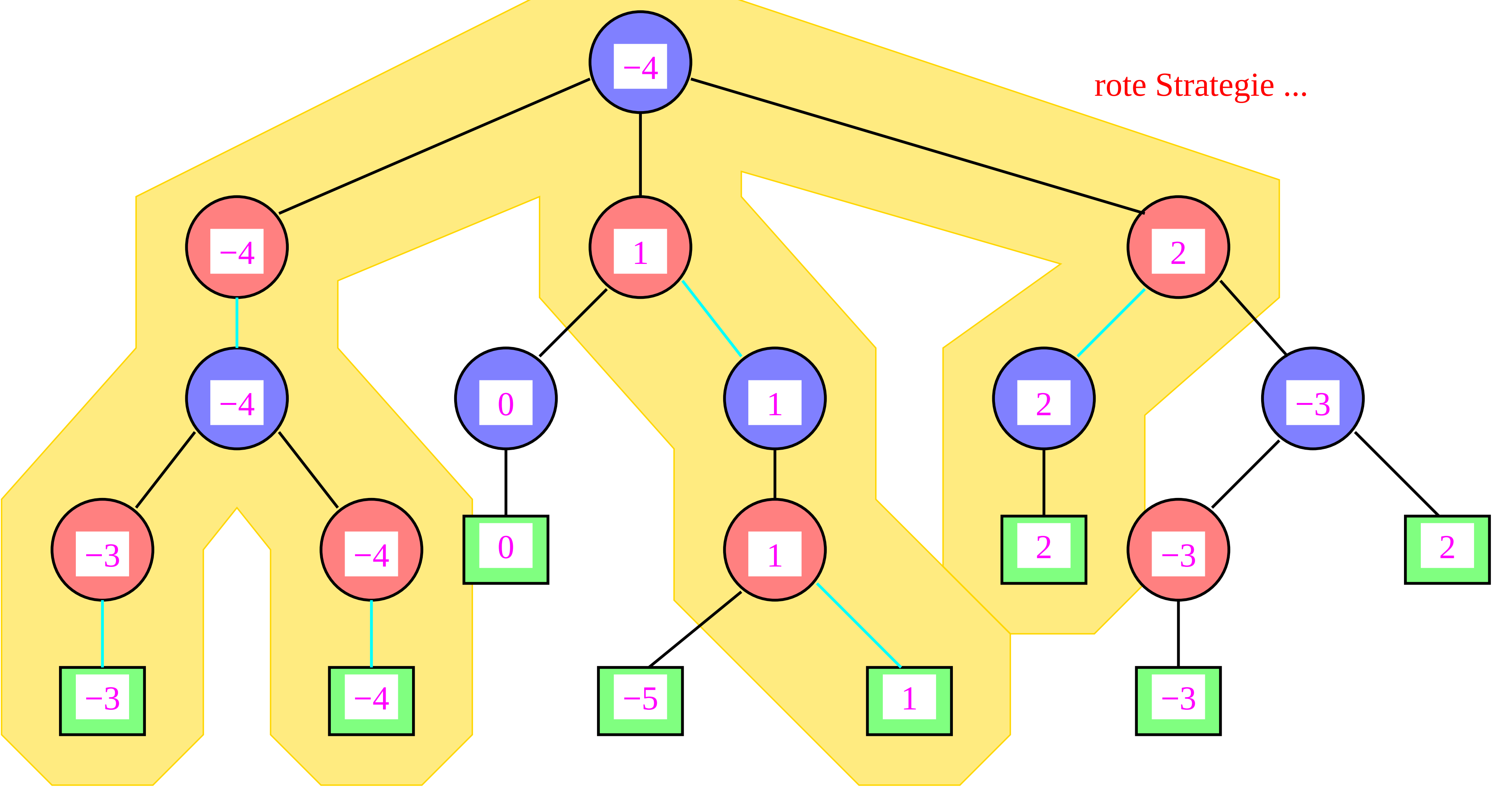
Eine optimale Strategie ist eine, deren Anwendung garantiert zu einer End-Konfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

Eine akzeptable Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wannimmer das möglich ist ...


Blau gewinnt!!!
−4
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3

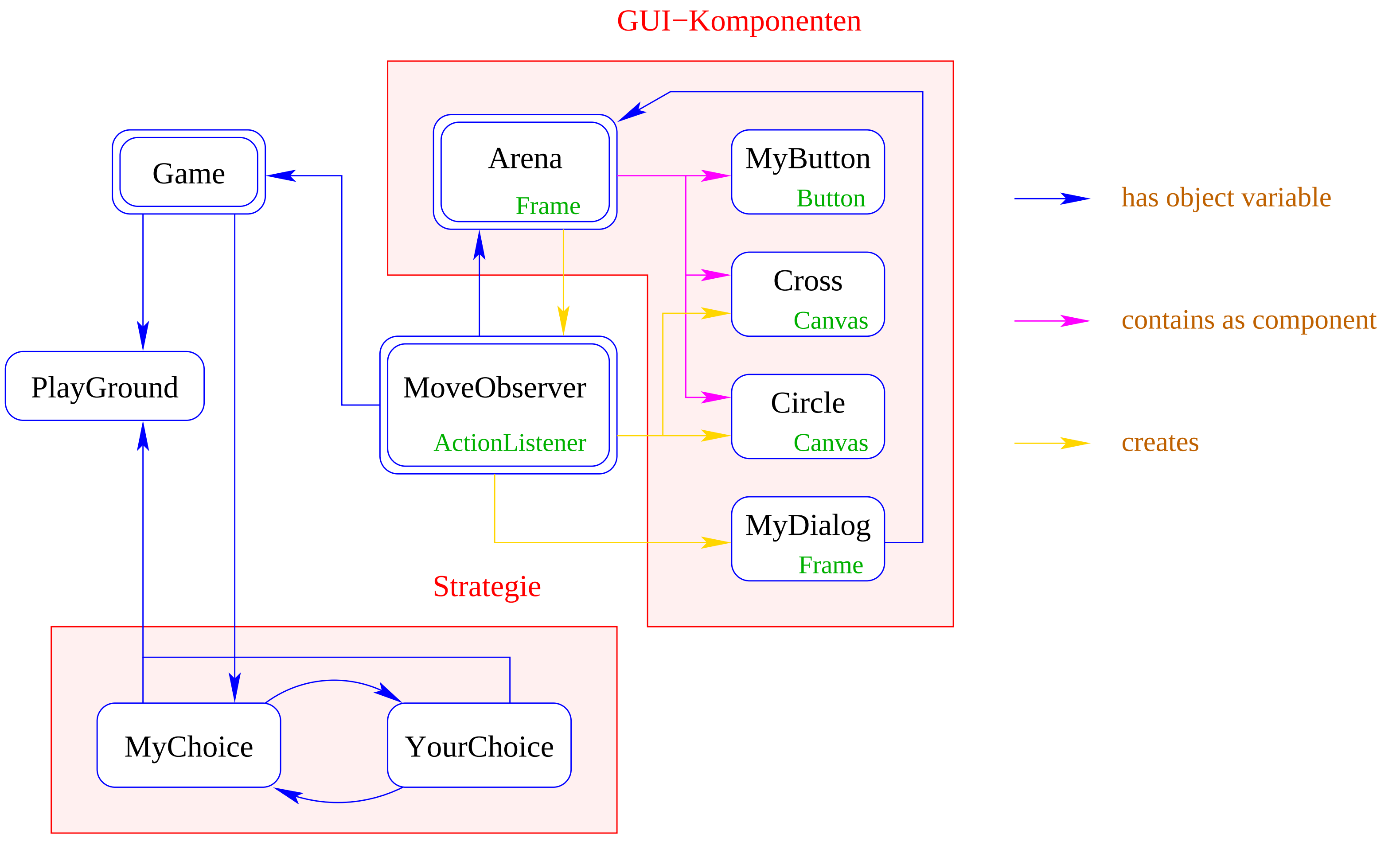


−4
blaue Strategie ...
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3



rote Strategie ...

22.1 Überblick über das Tic-Tac-Toe-System



Bemerkung:

Der Aufbau besteht konzeptuell aus drei Komponenten:

$\Rightarrow$ der internen Repräsentation des Spiels, des aktuellen Zustands und der Spiellogik

$\Longrightarrow$ das Modell

$\Rightarrow$ der externen graphische Benutzeroberfläche, mit der die Benutzerin interagiert

$\Longrightarrow$ die Ansicht

$\Rightarrow$ einer Kontrollschicht, die die Aktionen der Benutzerin an die Implementierung der Spiellogik weiterleitet und die Reaktion darauf auf der Oberfläche sichtbar macht

$\Longrightarrow$ die Kontrolle

Zusammen: Model-View-Control

- Ein solcher Aufbau ist typisch für viele interaktive Anwendungssysteme.
- Neben MVC gibt es für viele weitere häufige Anwendungssituationen nützliche Standard-Vorgehensweisen.

  Diese nennt man auch Design Patterns.

  ↑ Software Engineering

22.2 Berechnung einer Strategie

- Die Knoten des Spiel-Baums sind aus den Klassen `YourChoice` und `MyChoice`.
- `MyChoice` implementiert Knoten, in denen das Programm zieht.
- `YourChoice` implementiert Knoten der Spielerin.
- Jeder Knoten enthält:

  das aktuelle Spielbrett `ground`,
  einen garantierten Gewinn `value`
  sowie (Verweise auf) die Nachfolger-Knoten.
- `MyChoice`-Knoten enthalten zusätzlich den empfohlenen Zug `acceptableChoice`.

```
public class YourChoice implements PlayConstants {
   private PlayGround ground; private int value;
   private MyChoice[] answer;
   public YourChoice(PlayGround g, int place) {
      ground = new PlayGround(g,place,ME);
      answer = new MyChoice[9];
      value = ME;
      PossibleMoves moves = new PossibleMoves(ground);
      for(int choice = moves.next(); choice!=-1; choice = moves.next()) {
         if (ground.won(choice,YOU)) { value = YOU; continue;}
         answer[choice] = new MyChoice(ground,choice);
         int win = answer[choice].value();
         if (win < value) value = win;
      }
   }
   ...
```

- Das Interface `PlayConstants` fasst durchgängig benutzte Konstanten zusammen.
- `YOU` < `NONE` < `ME` repräsentieren die drei möglichen Ausgänge eines Spiels bzw. Belegungen eines Felds des Spielbretts (-1, 0 und 2).
- Die Klasse `PlayGround` repräsentiert Spielbretter.
- Das aktuelle Spielbrett wird aus demjenigen des Vater-Knotens und dem von `ME` gewählten Zug gestimmt.
- `MyChoice answers[choice]` enthält den Teilbaum für Zug `choice`.
- Das Objekt `PossibleMoves moves` enumeriert alle aktuell freien Positionen für Steine (mindestens eine :-)

  Gewinnt der Zug `choice`, wird `value = YOU` gesetzt. Andernfalls wird `value` auf das Minimum des Werts des Unterbaums `answers[choice]` und des alten Werts gesetzt.

```
    ...
    public int value() { return value;}
    public MyChoice answer(int place) {
        return answer[place];
    }
}  // end of class YourChoice
```

- `public int value();` liefert den Spielwert des aktuellen Spielbaums.
- `public MyChoice answer(int place);` liefert den Spiel-Teilbaum für den Zug `place`.

```
public class MyChoice implements PlayConstants {
   private PlayGround ground;
   private YourChoice yours;
   private int acceptableChoice, value;
   public MyChoice(PlayGround g, int place) {
      ground = new PlayGround(g,place,YOU);
      if (ground.won(place,YOU)) { value = YOU; return; }
      acceptableChoice = ground.force(ME);
      if (acceptableChoice != -1) { value = ME; return; }
      acceptableChoice = ground.force(YOU);
      if (acceptableChoice != -1) {
         yours = new YourChoice(ground,acceptableChoice);
         value = yours.value(); return;
      }
      ...
```

- Die Objekt-Methode `int force(int who);` liefert einen Zug, mit dem `who` unmittelbar gewinnt – sofern ein solcher existiert :-), andernfalls `-1`.
- Der Aufruf `ground.force(ME)` ermittelt einen solchen Gewinnzug für das Programm.
- Falls kein Gewinnzug existiert, liefert der Aufruf `ground.force(YOU)` eine Position, mit der der Gegner in einem Zug gewinnen könnte (was verhindert werden muss :-)
- Nur wenn keiner dieser Fälle auftritt, überprüfen wir sämtliche Zug-Möglichkeiten ...

```
    ...
    PossibleMoves moves = new PossibleMoves(ground);
    int tmp = moves.next();
    if (tmp == -1) { value = NONE; return;}
    value = YOU; do { acceptableChoice = tmp;
       yours = new YourChoice(ground,acceptableChoice);
       if (yours.value() > YOU) { value = yours.value(); return; }
       tmp=moves.next();
    } while (tmp != -1);
  }
  public int value() { return value;}
  public int acceptableChoice() { return acceptableChoice; }
  public MyChoice select(int place) {
    return yours.answer(place);
  }
} // end of class MyChoice
```

- Bei der Iteration über alle Zug-Möglichkeiten geben wir uns bereits mit einem akzeptablen Zug zufrieden, genauer gesagt: dem ersten, der mindestens unentschieden liefert.

Der Grund:

als nachziehender können wir (bei Tic-Tac-Toe und gegen einen optimalen Gegner) nichts besseres erreichen :-(

- `public int value();` liefert wieder den Spiel-Wert des Spielbaums.
- `public int acceptableChoice()` liefert einen akzeptablen Zug und
- `public MyChoice select(int place)` liefert bei gegebenem Spielzug für den akzeptablen Zug und die Antwort `place` des Gegners den nächsten Teilbaum.

22.3 Die Klasse `Game`

Die Klasse `Game` sammelt notwendige Datenstrukturen und Methoden zur Durchführung eines Spiels.

```
public class Game implements PlayConstants {
   public PlayGround ground = new PlayGround();
   private int count = 0;
   private MyChoice gameTree;
   public int nextMove(int place) {
      if (count == 1)
           gameTree = new MyChoice(ground,place);
      else gameTree = gameTree.select(place);
      return gameTree.acceptableChoice();
   }
   ...
```

- `PlayGround ground` enthält die jeweils aktuelle Spiel-Konfiguration.
- `int count` zählt die Anzahl der bereits ausgeführten Züge.
- `MyChoice gameTree` enthält eine Repräsentation der `ME`-Strategie.
- `public int nextMove(int place)` liefert den Antwort-Zug auf den `YOU`-Zug `place`:
    - War `place` der erste `YOU`-Zug, wird ein `MyChoice`-Objekt `gameTree` für das verbleibende Spiel angelegt.
    - Bei jedem weiteren Zug wird `gameTree` auf die gemäß `place` ausgewählte Teil-Strategie gesetzt.
    - Den nächsten `ME`-Zug liefert jeweils `acceptableChoice` in `gameTree`.

```
    ...
    public boolean possibleMove(int place) {
       if (count%2 == 1) return false;
       return (ground.free(place));
    }
    public boolean finished() { return (count == 9);}
    public void move(int place,int who) {
       count++;
       ground.move(place,who);
    }
 }  // end of class Game
```

Weitere nützliche Methoden:

- `boolean possibleMove(int place)` überprüft, ob der Zug `place` überhaupt möglich ist;
- `public boolean finished()` überprüft, ob bereits 9 Züge gespielt wurden;
- `public void move(int place,int who)` erhöht den Zug-Zähler und führt den Zug auf dem Spielfeld `ground` aus.

22.4 Die graphische Benutzer-Oberfläche

Idee:

- Unbesetzte Felder repräsentieren wir durch `Button`-Komponenten.
- Die Knöpfe werden in einem (3×3)-Gitter angeordnet.
- Wird ein Zug ausgeführt, wird der entsprechende Knopf durch eine (bemalte) `Canvas`-Fläche ersetzt.

```
import java.awt.*;
import java.awt.event.*;
class MyButton extends Button {
   private int number;
   public MyButton(int i) {
      number = i;
      setBackground(Color.white);
   }
   public int getNumber() { return number;}
}  // end of class MyButton
```

- `MyButton` erweitert die Klasse `Button` um die Objekt-Variable `int number`.
- `number` wird vom Konstruktor gesetzt und mit `int getNumber();` abgefragt.

```
public class Arena extends Frame {
   private MoveObserver moveObserver;
   public Arena() {
      setBackground(Color.blue);
      setSize(300,300); setLocation(400,400);
      setLayout(new GridLayout(3,3,3,3)); start();
   }
   public void start() {
      removeAll();
      moveObserver = new MoveObserver(this);
      for (int i=0; i<9;i++) {
         MyButton b = new MyButton(i);
         b.addActionListener(moveObserver); add(b);
      }
   }
```

- `Arena` ist die Unterklasse von `Frame`, in der wir das Spielfeld anlegen.
- Dazu wählen wir das `GridLayout` als Layout-Manager.
- `public GridLayout(int row, int col, int cs, int rs);` konstruiert ein `GridLayout`-Objekt mit `row` Zeilen, `col` Spalten sowie den Abständen `cs` und `rs` zwischen den Spalten bzw. Zeilen.
- Zu der Arena fügen wir neun (durch-numerierte) `MyButton`-Komponenten hinzu.
- Gemeinsamer `ActionListener` für sämtliche Knöpfe ist das `MoveObserver`-Objekt `moveObserver`.
- `public void removeAll();` beseitigt sämtliche Komponenten eines Containers.

```
    ...
    public void move(int place, Canvas canvas) {
        remove(place); add(canvas, place);
    }
    public static void main(String[] args) {
        Arena arena = new Arena(); arena.setVisible(true);
    }
} //end of class Arena
```

- `public void move(int place, Canvas canvas);` ersetzt die Komponente an der Stelle `place` durch die `Canvas`-Komponente `canvas`.
- `public void remove(int i);` beseitigt die `i`-te Komponente eines `Container`-Objekts.
- `public void add(Component c, int i);` fügt die Komponente `c` als `i`-te Komponente ein.

22.5 Beantwortung eines Zugs

Beachte:

- Ziehen der Tic-Tac-Toe-Spielerin heißt Drücken eines Knopfs.
- Drücken eines Knopfs löst einen `ActionEvent` aus.
- Die Klasse `MoveObserver` verarbeitet die Ereignisse.
- Aufgabe eines `MoveObserver`-Objekts ist es, das Spielfeld zu modifizieren und einen Antwort-Zug zu liefern, d.h. zu finden und dann anzuzeigen.

```
import java.awt.*;
import java.awt.event.*;
public class MoveObserver implements
                          ActionListener, PlayConstants {
   private Game game; private Arena frame;
   public MoveObserver(Arena a) {
      game = new Game(); frame = a;
   }
   public void actionPerformed(ActionEvent e) {
      MyButton button = (MyButton) e.getSource();
      int place = button.getNumber();
      if (!game.possibleMove(place)) return;
      frame.move(place,new Cross());
      game.move(place,YOU);
      frame.setVisible(true);
      ...
```

- Wurde das Feld `place` ermittelt, das die Gegnerin setzte, wird der Knopf an dieser Stelle durch ein neues `Cross`-Element ersetzt.
- Dann wird der Zug auch auf der internen Repräsentation des Spielbretts ausgeführt.
- `frame.setVisible(true);` macht diese Änderung auf dem Bildschirm sichtbar.

```
...
if (game.ground.won(place,YOU)) {
   game.ground.clear();
   (new MyDialog(frame,"You won ...")).setVisible(true);
   return;
}
if (game.finished()) {
   (new MyDialog(frame,"No winner ...")).setVisible(true);
   return;
}
place = game.nextMove(place);
frame.move(place,new Circle());
game.move(place,ME);
frame.setVisible(true);
...
```

- Die Objekt-Methode `public void clear();` der Klasse `PlayGround` verhindert weitere Züge, indem sämtliche Felder auf `3` gesetzt werden (Hack – sorry!).
- Falls die Spielerin `YOU` gewonnen hat, beenden wir das Spiel und erzeugen ein Dialog-Fenster der Klasse `MyDialog`.
- Ähnlich verfahren wir, wenn das Spiel nicht verloren, aber zuende ist ...
- Ist das Spiel weder verloren noch zuende, bestimmen wir unseren nächsten Zug.
- Diesen führen wir auf unserer Intern-Darstellung des Spielbretts wie in dem `Arena`-Frame aus.

```
      ...
      if (game.ground.won(place,ME)) {
         game.ground.clear();
         (new MyDialog(frame,"I won ...")).setVisible(true);
         return;
      }
   }
}  // end of class MoveObserver
```

- Falls unser Zug zum Gewinn führte, beenden wir das Spiel und erzeugen ein `MyDialog`-Fenster.

22.6 Ein Dialog-Fenster

Aufgaben:

1. Anzeige des Spiel-Ausgangs;
2. Ermöglichen der Beendigung bzw. Wiederholung des Spiels.

```
import java.awt.*;
import java.awt.event.*;
public class MyDialog extends Frame implements ActionListener {
   private Button repeat, kill; private Arena arena;
   public MyDialog(Arena frame, String string) {
      arena = frame;
      setLocation(200,200); setSize(250,50);
      setBackground(Color.yellow);
      setForeground(Color.blue);
      setLayout(new FlowLayout());
      add(new Label(string));
      repeat = new Button("new");
      repeat.addActionListener(this); add(repeat);
      kill = new Button("kill");
      kill.addActionListener(this);   add(kill);
   }
```

- `public void setForeground(Color c);` setzt die aktuelle Vordergrund-Farbe auf `c`. Schrift erscheint dann z.B. in dieser Farbe.
- Wir legen ein neues Fenster mit gelbem Hintergrund und blauem Vordergrund an.
- Wir benutzen das `FlowLayout`, um Elemente im Fenster zu positionieren.
- Den Kommentar zum Spiel-Ergebnis übergeben wir dem Konstruktor, der es in einem `Label`-Element anzeigt.
- Weiterhin fügen wir zwei Knöpfe für Wiederholung bzw. Beendigung hinzu.
- Gemeinsamer `ActionListener` beider Knöpfe ist das Fenster-Objekt selbst ...

```
    ...
    public void actionPerformed(ActionEvent e) {
       Button b = (Button) e.getSource();
       if (b == kill) { arena.setVisible(false); System.exit(0);}
       else {
          arena.start(); arena.setVisible(true);
          setVisible(false);
       }
    }
 }  // end of class MyDialog
```

- `public void exit(int x);` ist eine Klassen-Methode der Klasse `System`, die die Applikation (mit Rückgabe-Wert `x`) beendet.

- Falls der `"kill"`-Knopf gedrückt wurde, wird die Applikation beendet.
- Falls der `"new"`-Knopf gedrückt wurde, wird für `arena` die `start()`-Methode aufgerufen, d.h. ein neues `Game`-Objekt angelegt und die Komponenten in `arena` neu zu Knöpfen initialisiert.
- `setVisible(false);` lässt das Fenster verschwinden.

22.7 Effizienz

Problem:

Spielbäume können **RIESIG** werden!!

Unsere Lösung:

- Wir erzeugen die `ME`-Strategie nicht für alle möglichen Spiel-Verläufe, sondern erst nach dem ersten Zug der Gegnerin. Spart ... Faktor 9
- Wir berücksichtigen Zug-Zwang. Spart ... ??!!...:-)
- Wir sind mit akzeptablen `ME`-Zügen zufrieden. Spart ungefähr ... Faktor 2

Achtung:

- Für Tic-Tac-Toe reicht das vollkommen aus: pro Spielverlauf werden zwischen 126 und 1142 `MyChoice`-Knoten angelegt ...
- Für komplexere Spiele wie Dame, Schach oder gar Go benötigen wir weitere Ideen ...

1. Idee: Eröffnungen

- Tabelliere Anfangs-Stücke optimaler Spiel-Verläufe.
- Konstruiere die Strategie erst ab der ersten Konfiguration, die von den tabellierten Eröffnungen abweicht ...

Beispiel: Tic-Tac-Toe

Wir könnten z.B. beste Antworten auf jeden möglichen Eröffnungs-Zug tabellieren:

```
public interface Opening {
   int[] OPENING = {
      4, 4, 4, 4, 2, 4, 4, 4, 4
   };
}
```

- Die Funktion `int nextMove(int place);` schlägt dann den ersten Antwort-Zug in `OPENING` nach.

- Erst bei der zweiten Antwort (d.h. für den vierten Stein auf dem Brett) wird die `ME`-Strategie konstruiert.
- Dann bleiben grade mal höchstens $6! = 720$ Spiel-Fortsetzungen übrig ... die Anzahl der tatsächlich benötigten `MyChoice`-Knoten scheint aber nur noch zwischen 9 und 53 zu schwanken (!!!)

2. Idee: Bewertungen

Finde eine geeignete Funktion `advice`, die die Erfolgsaussichten einer Konfiguration direkt abschätzt, d.h. ohne Aufbau eines Spielbaums.

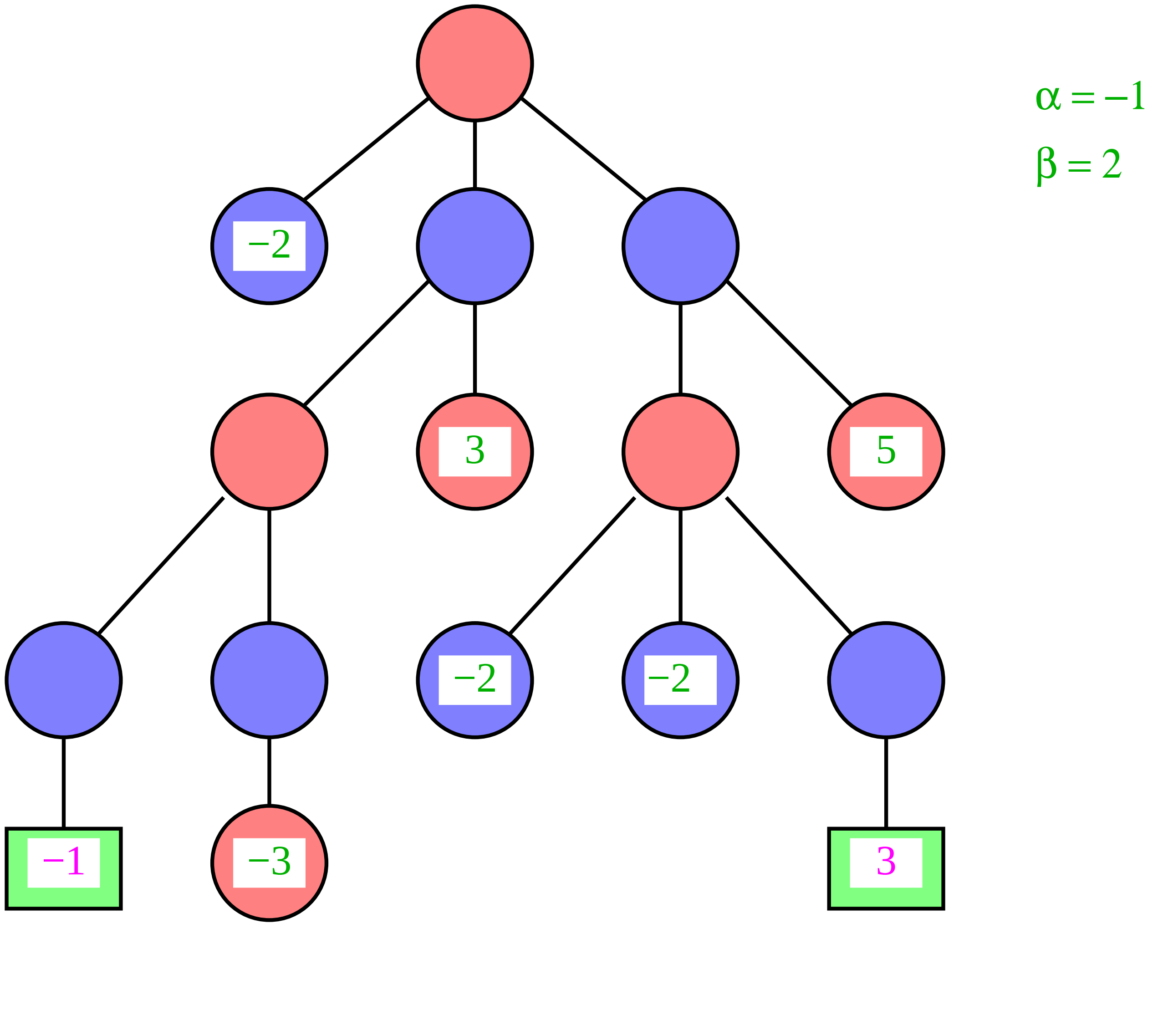
Achtung:

- I.a. ist eine solche Funktion nicht bekannt :-(
- Man muss mit unpräzisen bis fehlerhaften Bewertungs-Funktionen zurecht kommen ...

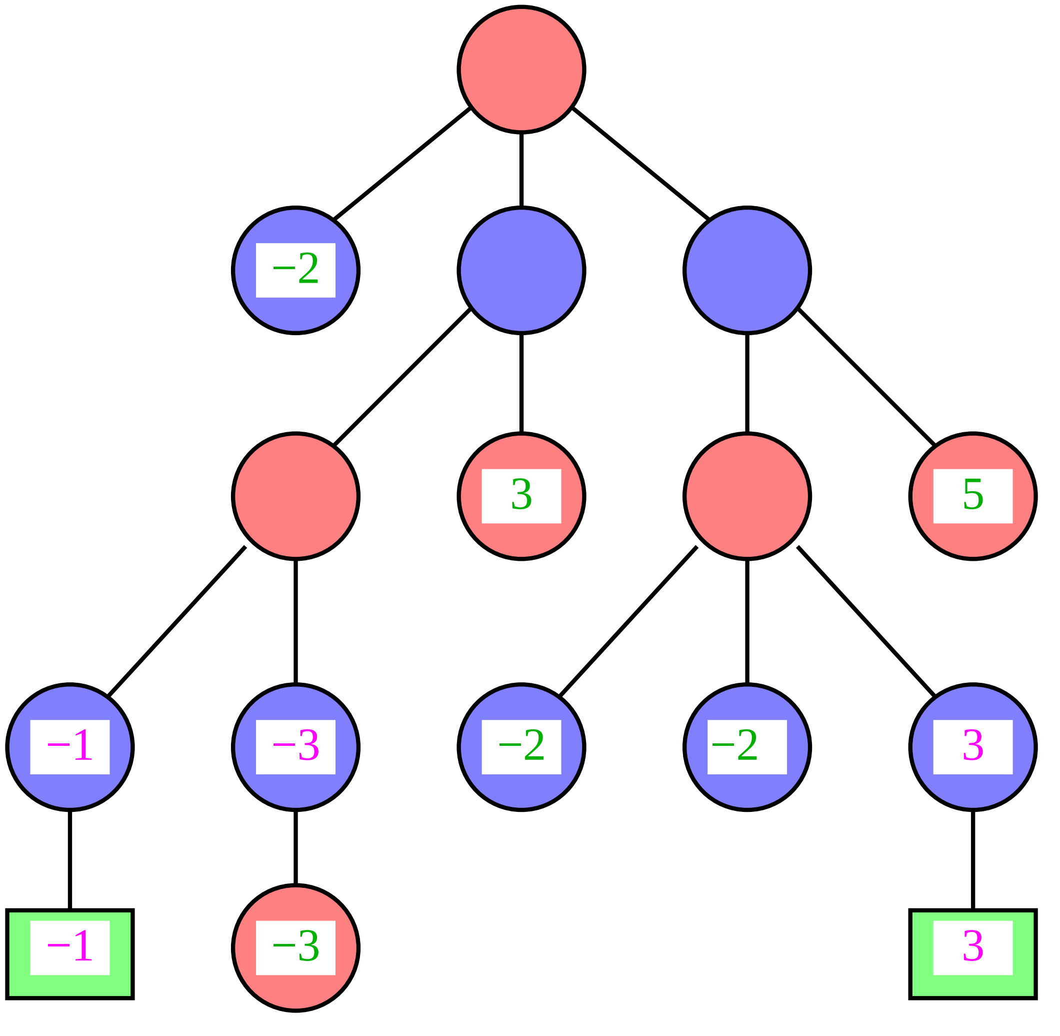
3. Idee: (α, β)-Pruning

- Wir nehmen an, wir hätten eine halbwegs zuverlässige Bewertungsfunktion `advice`, d.h. es gibt Zahlen $\alpha < 0 < \beta$ so dass für Konfigurationen conf gilt:
  - Ist `advice(conf) <` α, gewinnt voraussichtlich die Gegnerin;
  - Ist β `< advice(conf)`, gewinnt voraussichtlich das Programm.
- Zur Bestimmung unseres nächsten Zugs, betrachten wir sukzessive alle Nachfolger-Konfigurationen conf.
  - Ist β `< advice(conf)`, ist der Zug akzeptabel.

- Gibt es keinen akzeptablen Zug, betrachten wir rekursiv die Nachfolger aller Konfigurationen conf, für die $\alpha <$ `advice(conf)` .
- Für gegnerische Konfigurationen gehen wir dual vor ...

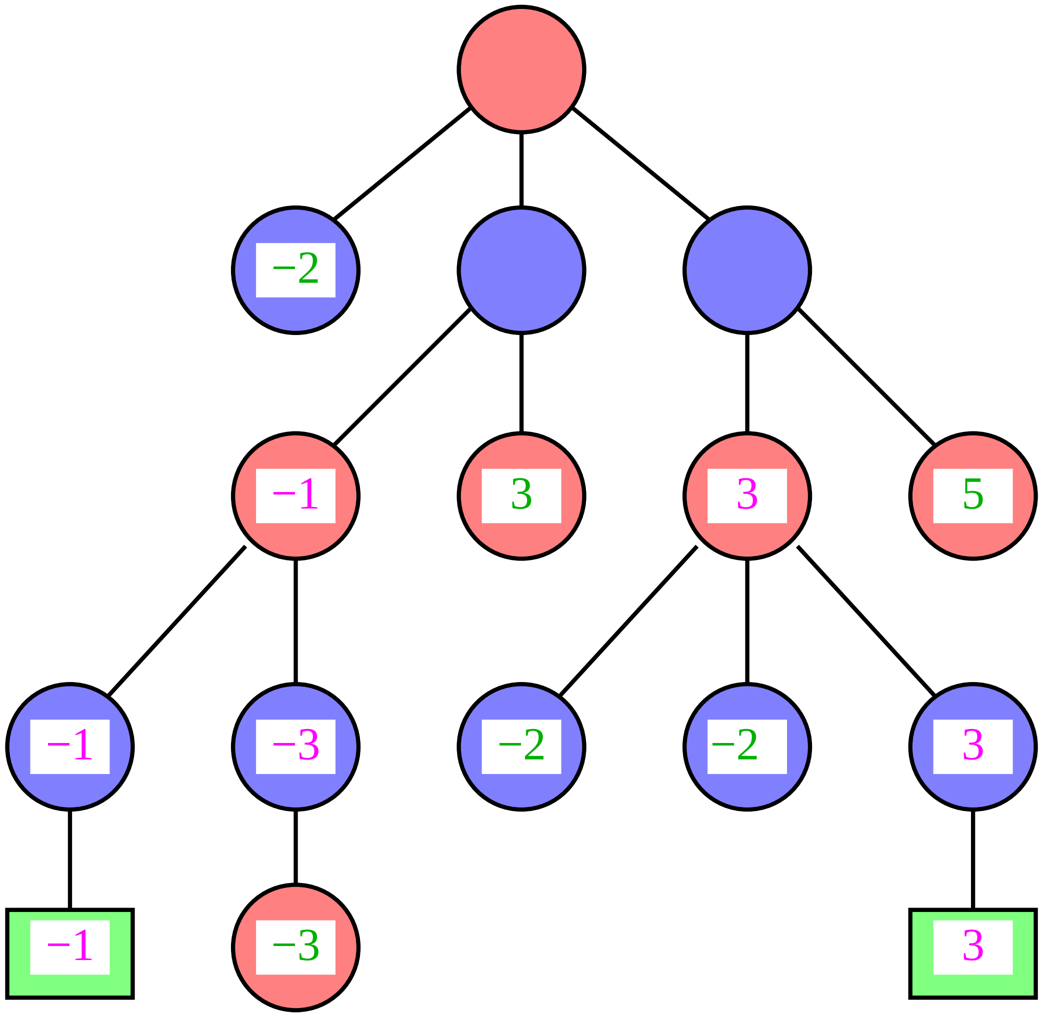


−2
3
5
−1
−3
−2
−2
3
α = −1
β = 2



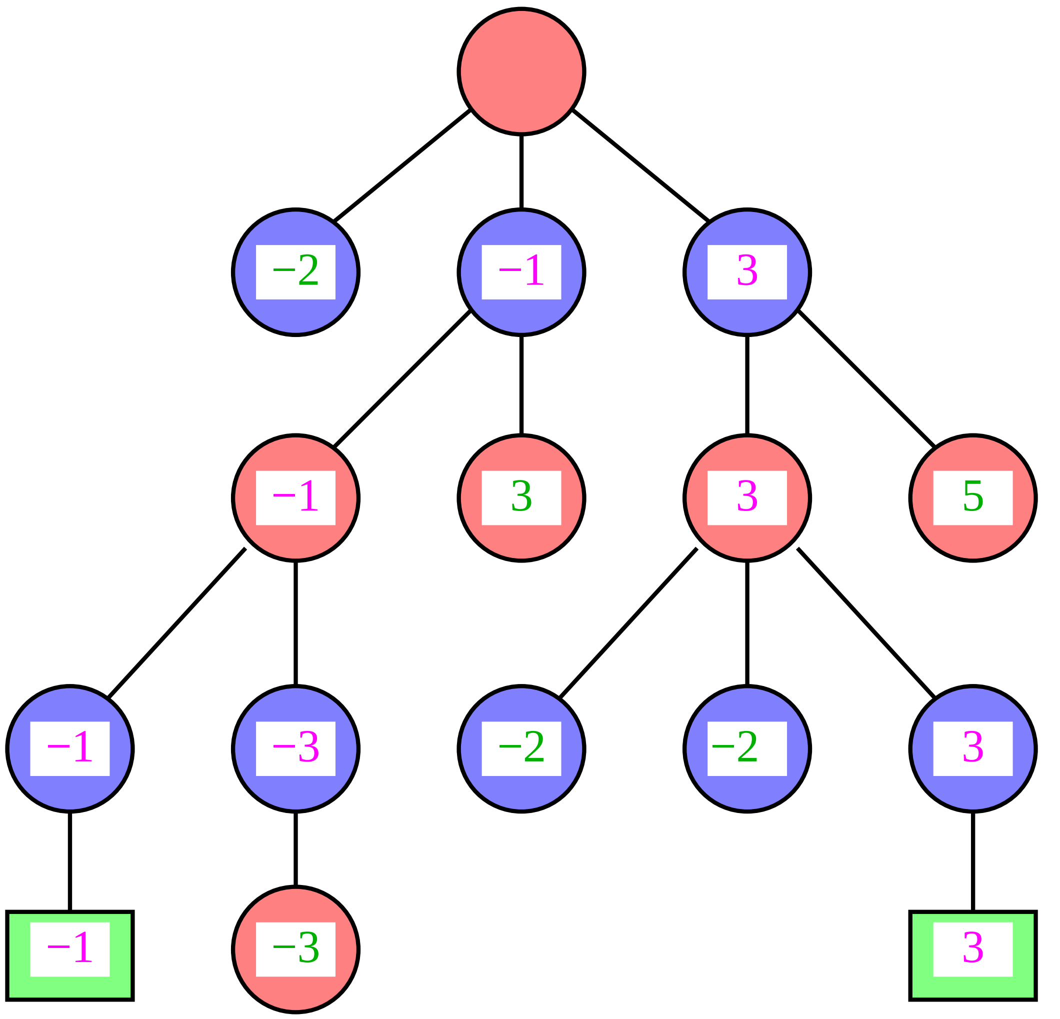
$\alpha = -1$

$\beta = 2$



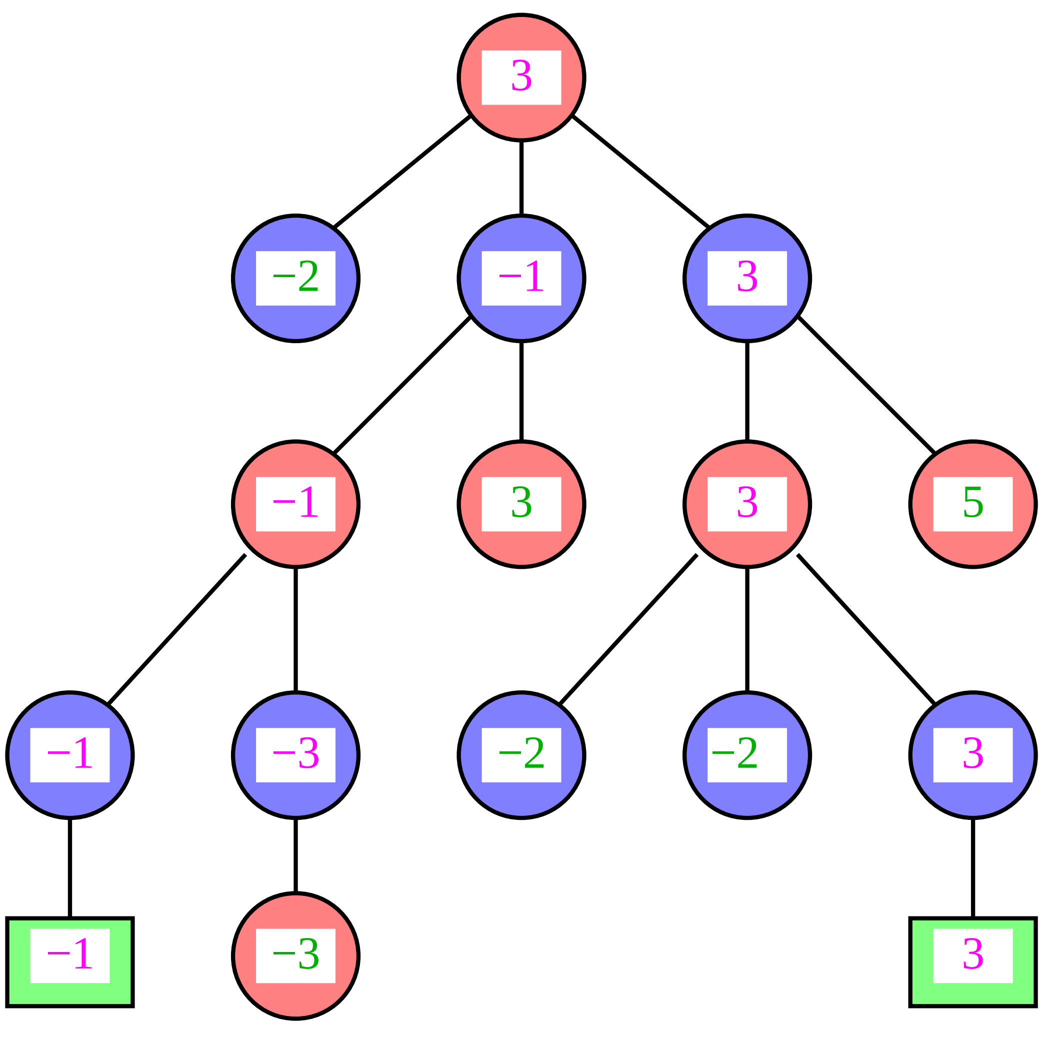
$\alpha = -1$

$\beta = 2$



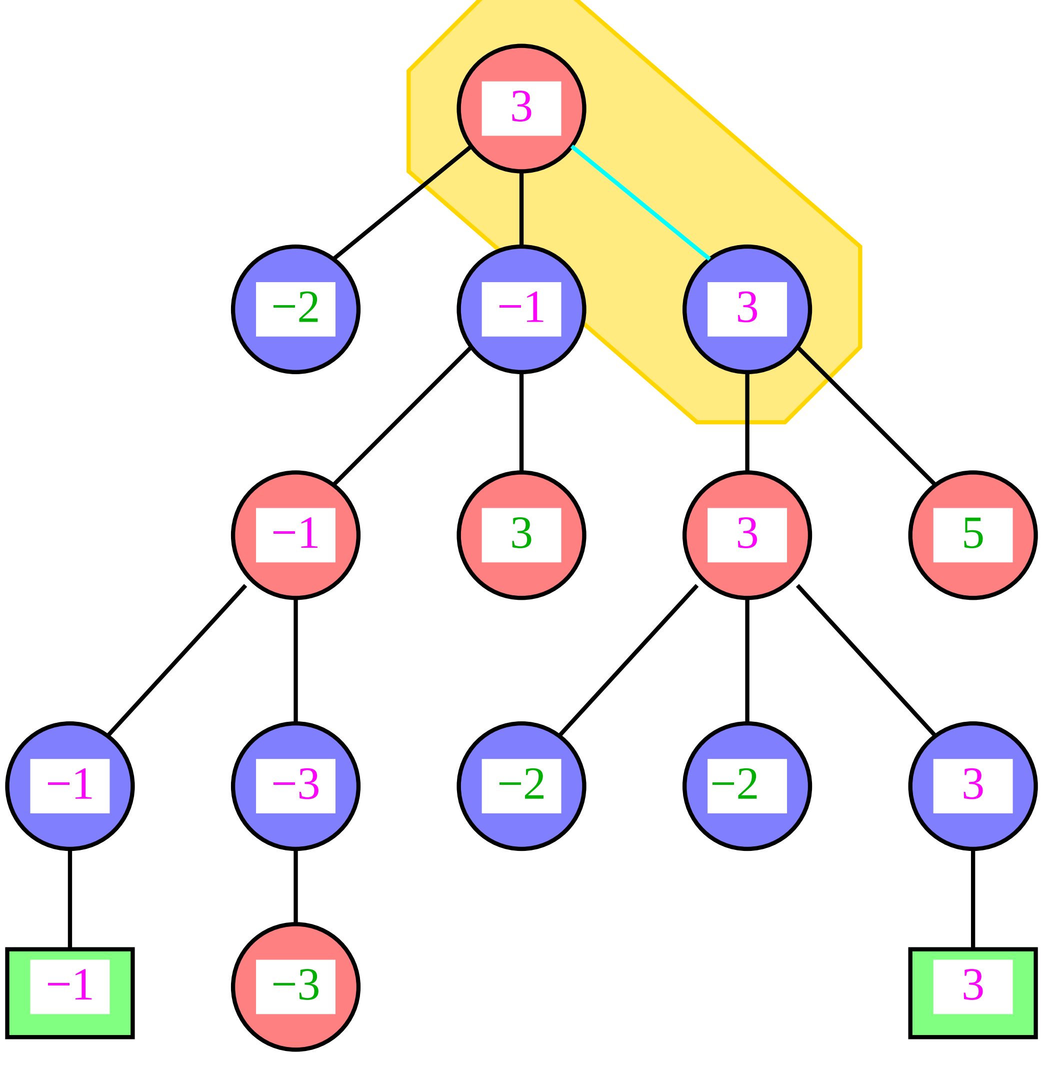
$\alpha = -1$

$\beta = 2$



$\alpha = -1$

$\beta = 2$



$\alpha = -1$

$\beta = 2$

Vorteil:

Die Anzahl der zu untersuchenden Konfigurationen wird (hoffentlich ;-) beträchtlich eingeschränkt!

Nachteil:

Ist die Bewertungsfunktion offensichtlich fehlerhaft, lässt sich das Programm austricksen ...

Frage:

Wie findet man eine Bewertungsfunktion deren Fehlerhaftigkeit nicht so offensichtlich ist???

Ausblick:

- Nicht alle 2-Personen-Spiele sind endlich.
- Gelegentlich hängt der Effekt eines Zugs zusätzlich vom Zufall ab.
- Eventuell ist die aktuelle Konfiguration nur partiell bekannt.

$\Longrightarrow$ Spieltheorie