

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 allokkieren 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:

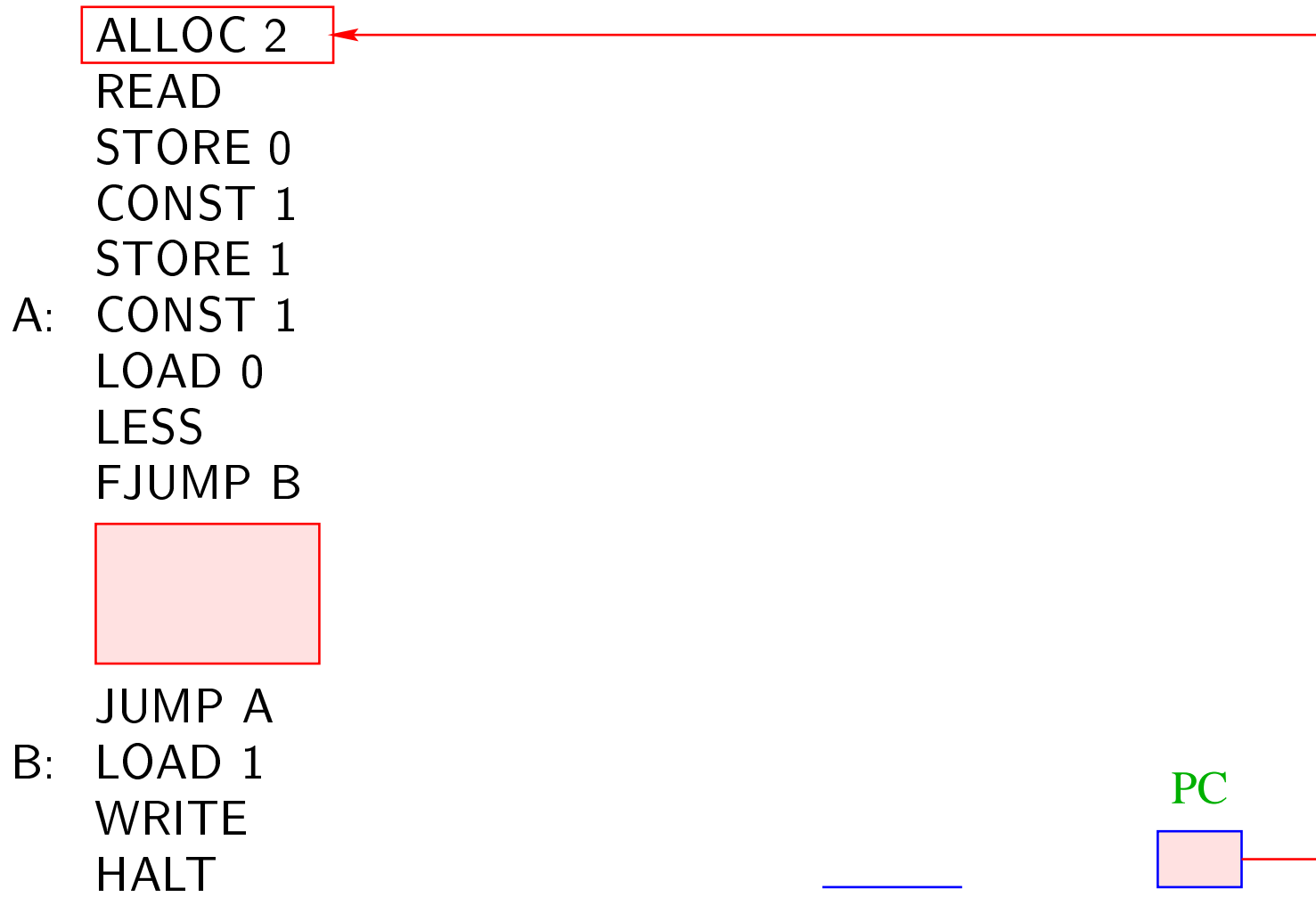
Für das Programm

```
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) :

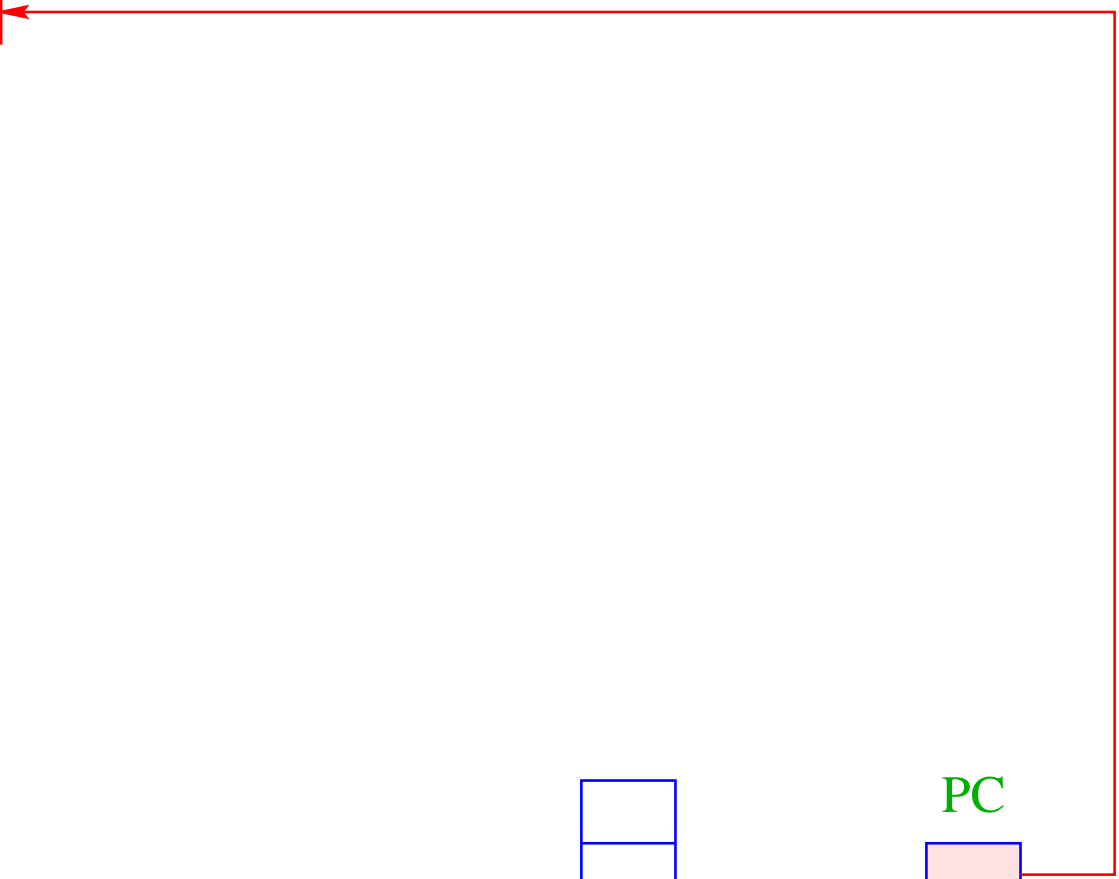
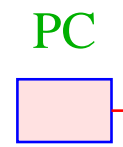
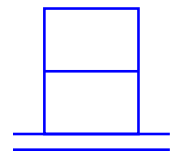
ALLOC 2	A: CONST 1
READ	LOAD 0
STORE 0	LESS
CONST 1	FJUMP B
STORE 1	

LOAD 1	LOAD 0	B: LOAD 1
LOAD 0	CONST 1	WRITE
MUL	SUB	HALT
STORE 1	STORE 0	
	JUMP A	



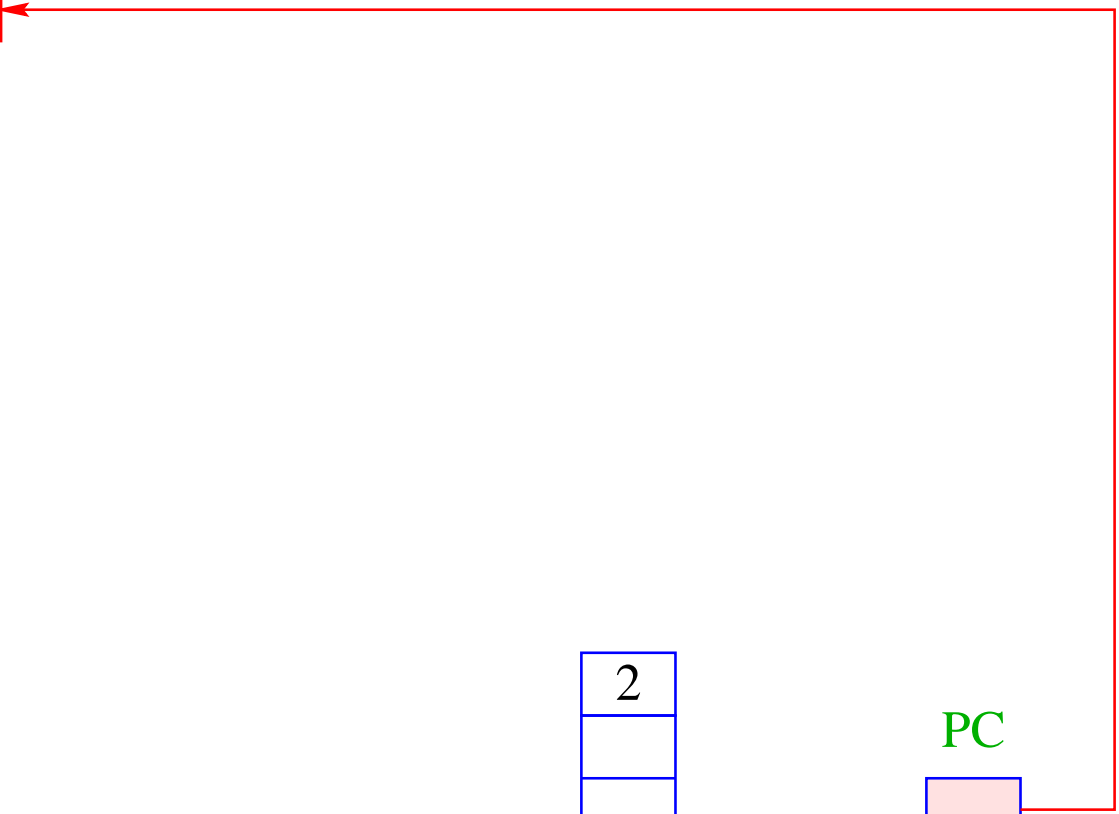
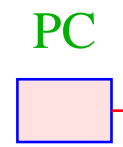
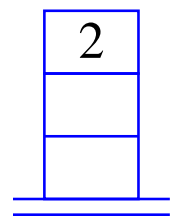
ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B

JUMP A
B: LOAD 1
WRITE
HALT



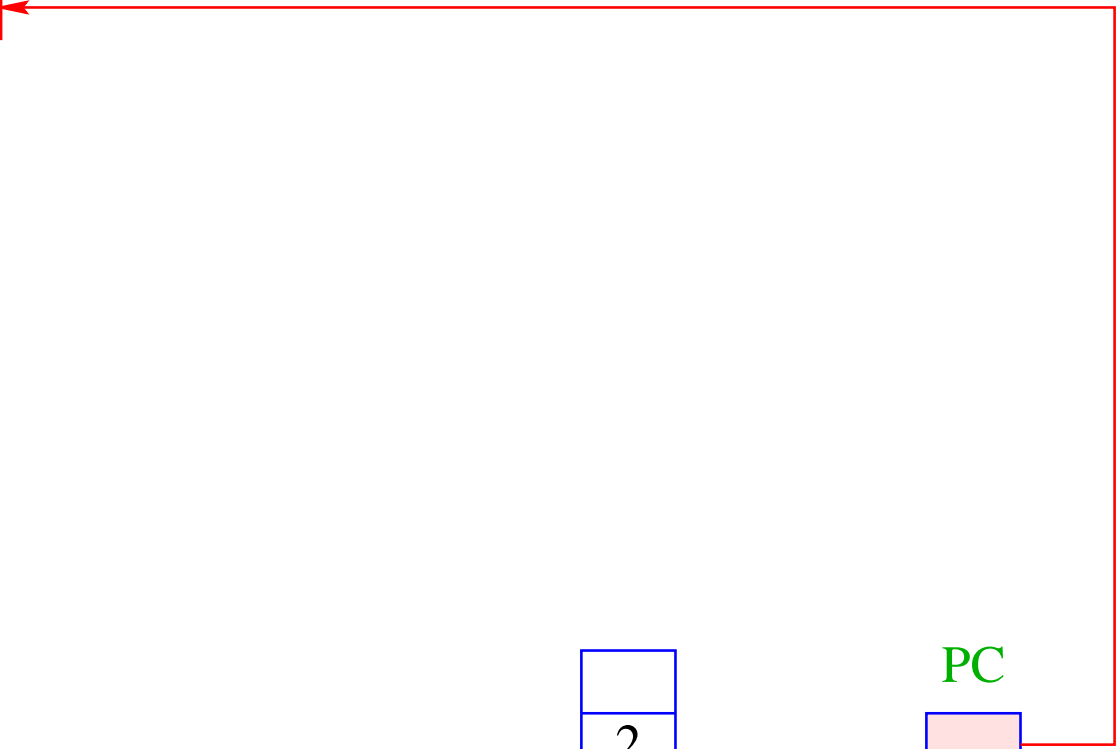
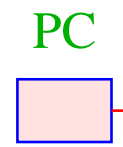
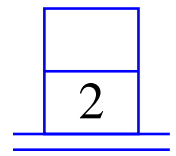
ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B

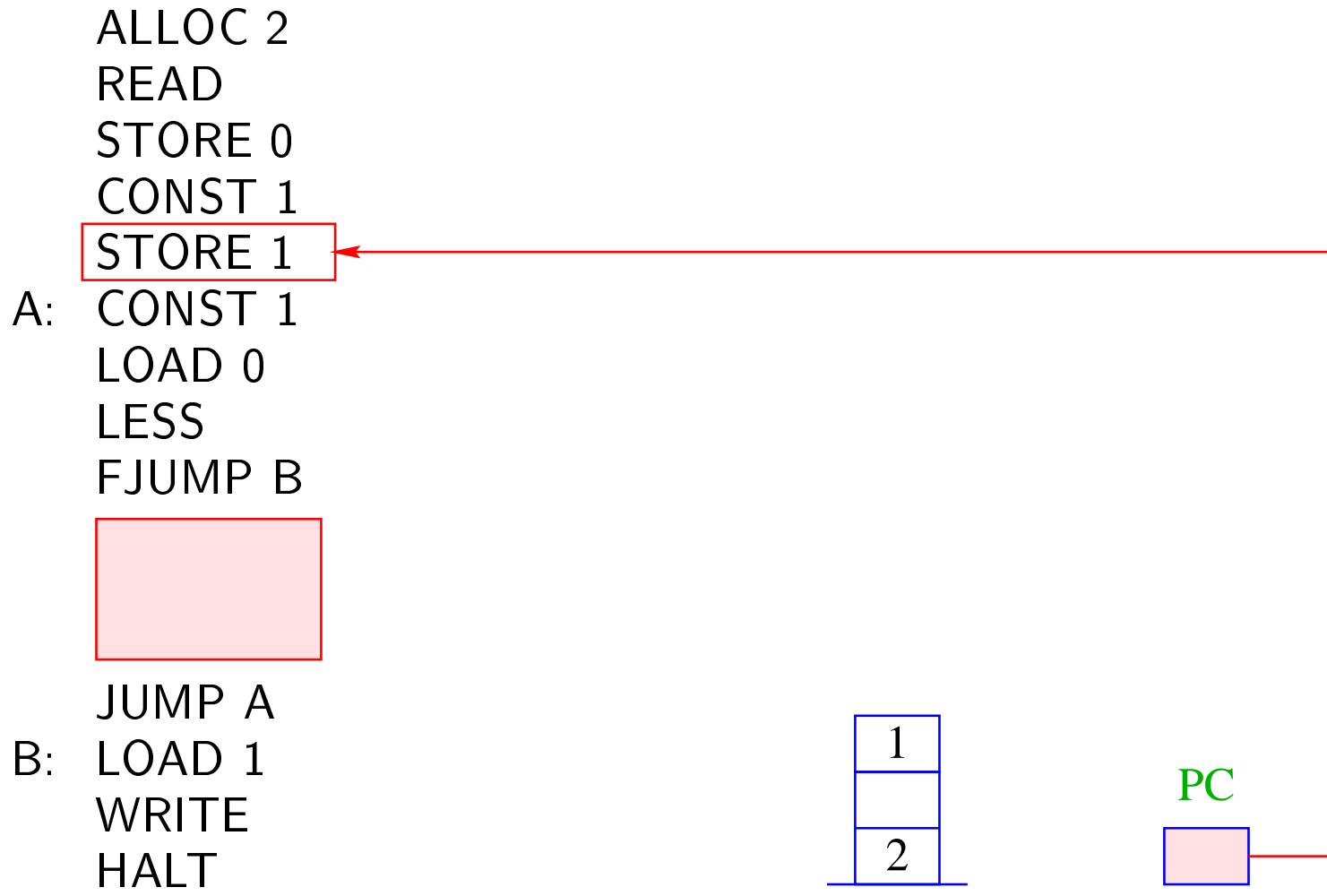
JUMP A
B: LOAD 1
WRITE
HALT

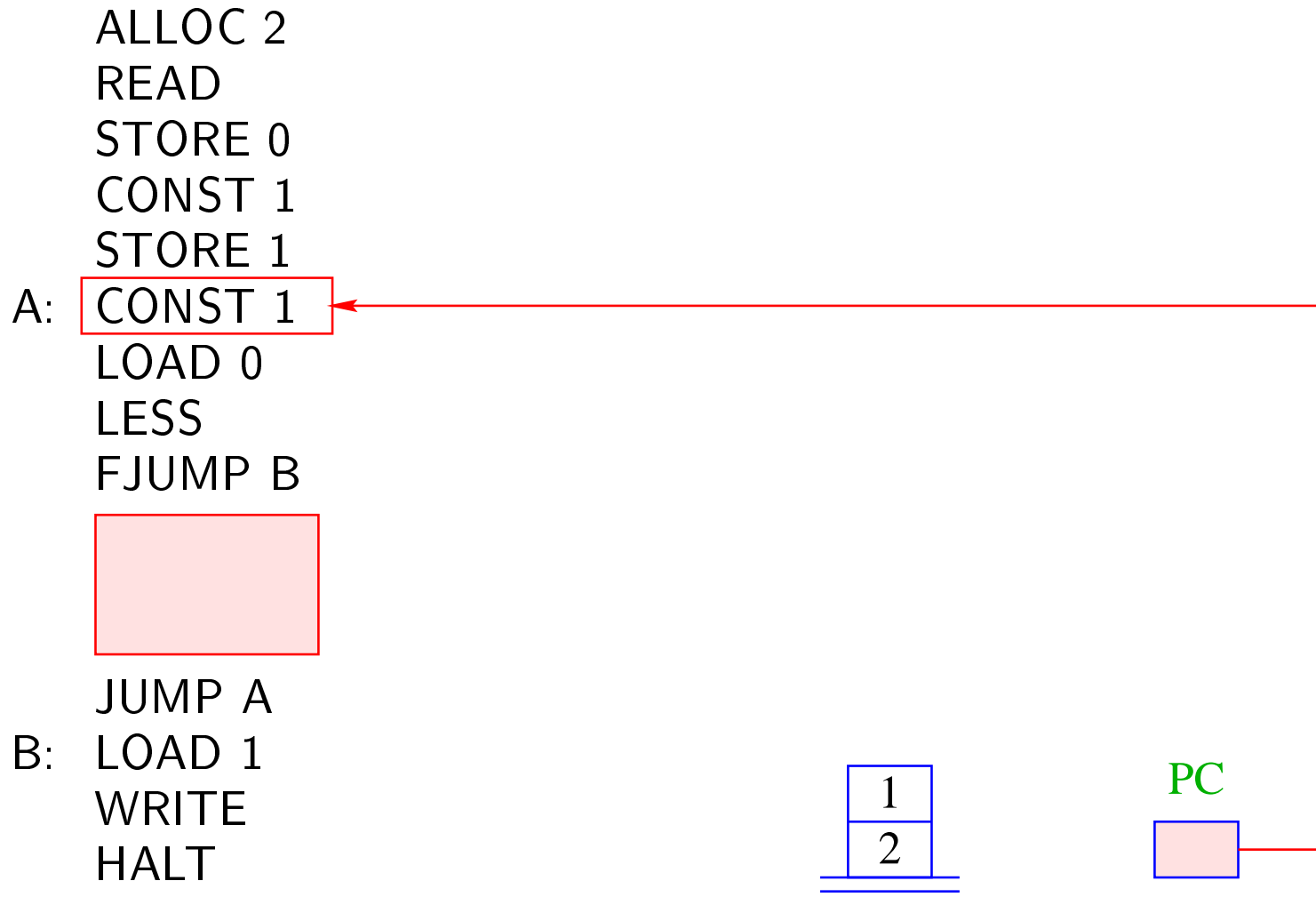


ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B

JUMP A
B: LOAD 1
WRITE
HALT

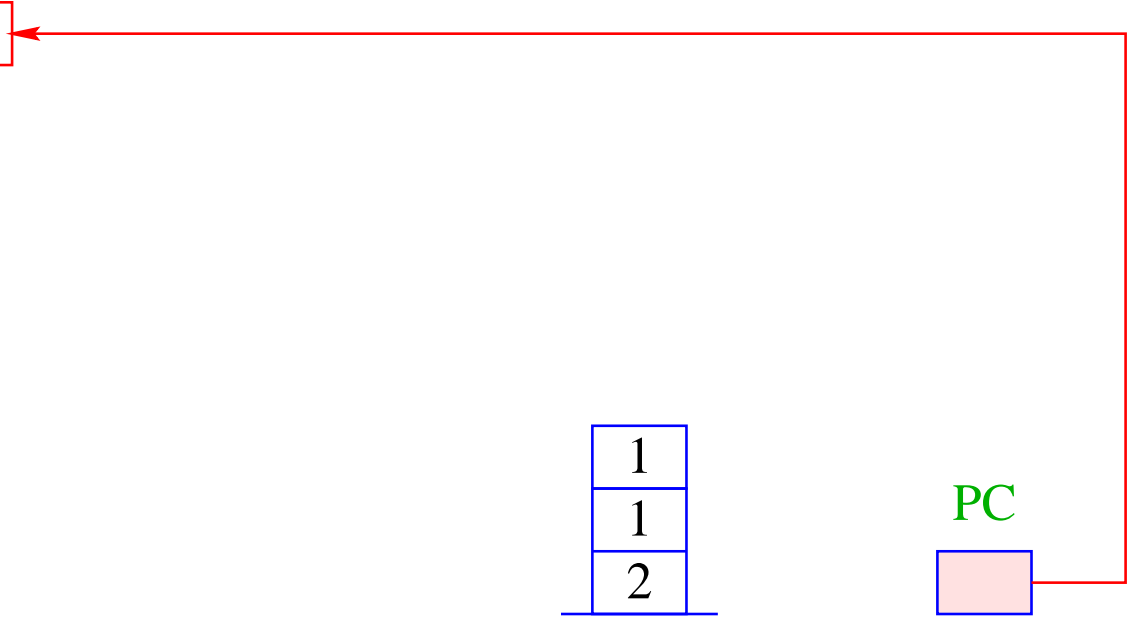
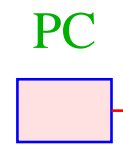
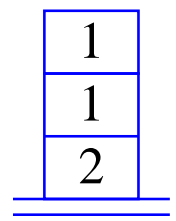






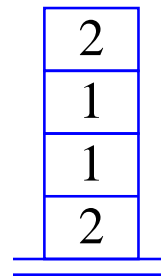
ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B

JUMP A
B: LOAD 1
WRITE
HALT

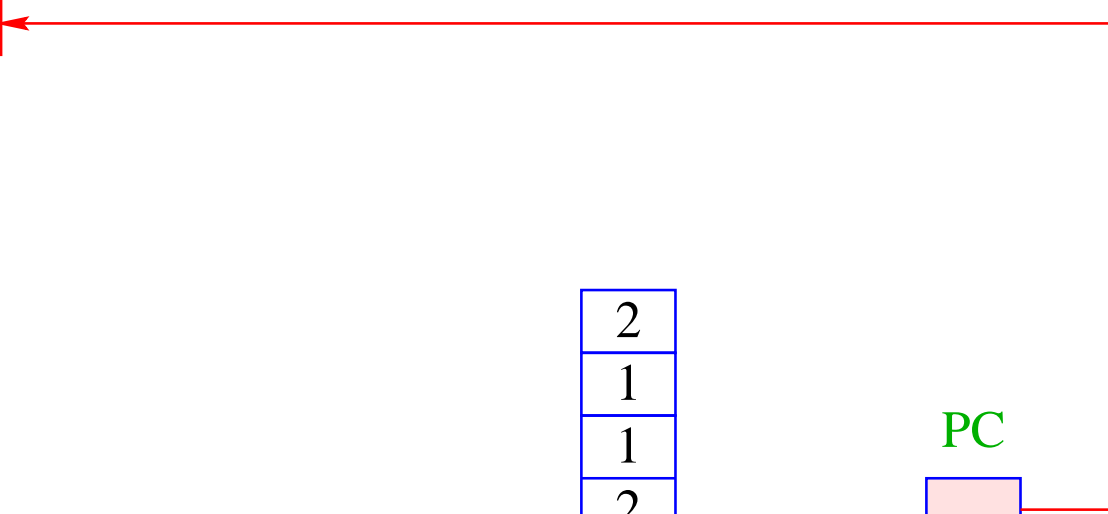


ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B

JUMP A
B: LOAD 1
WRITE
HALT

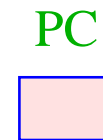
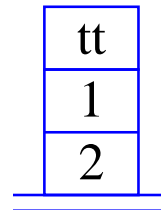
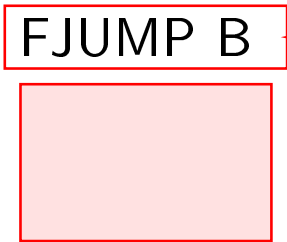


PC



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

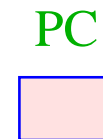
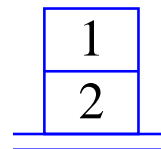
JUMP A
B: LOAD 1
WRITE
HALT



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



JUMP A
B: LOAD 1
WRITE
HALT

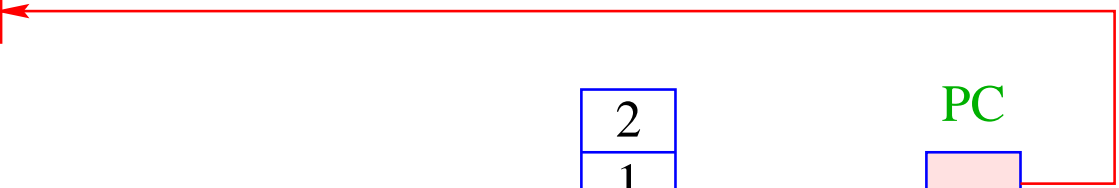
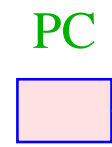
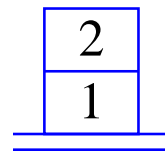


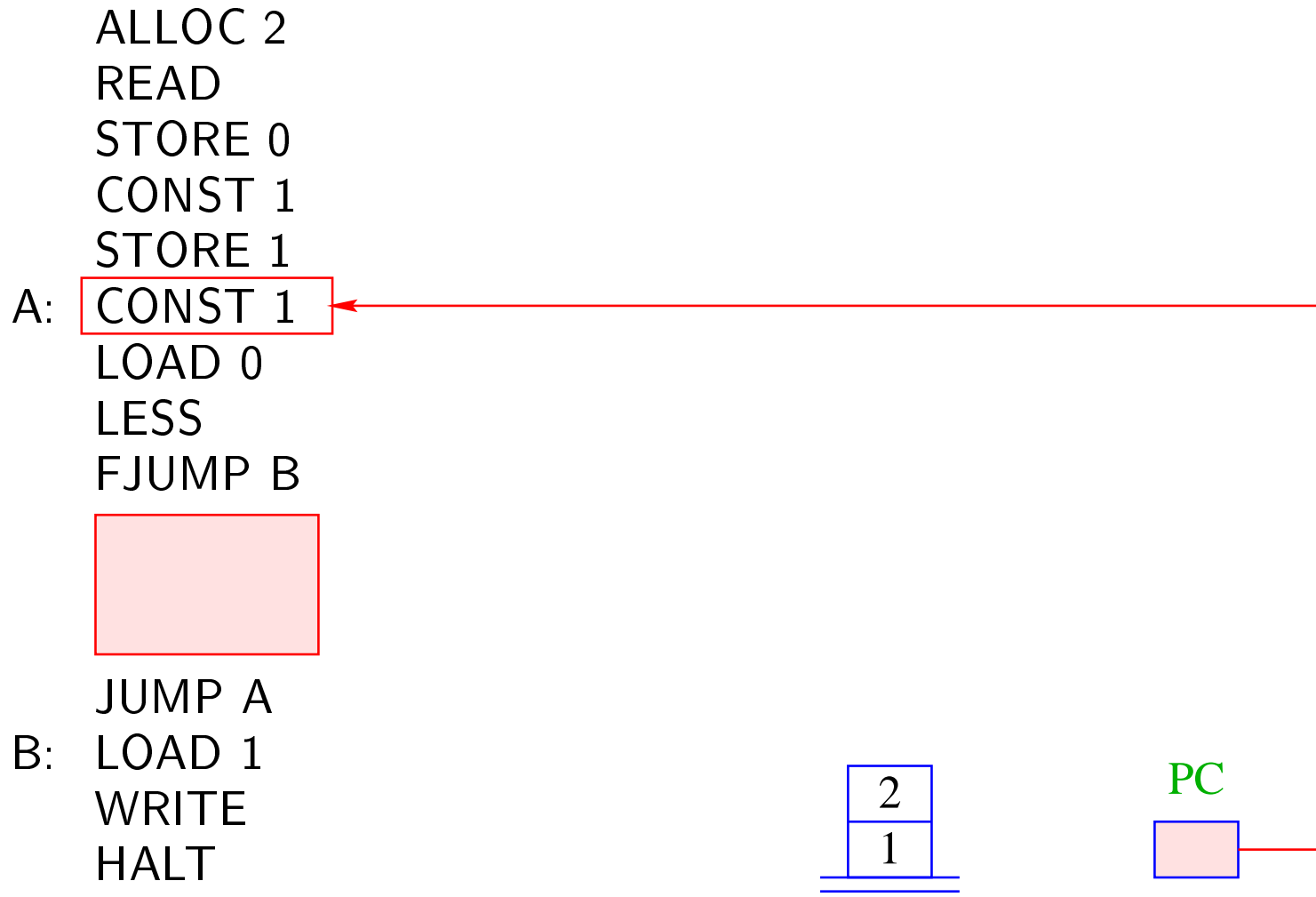
ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B



JUMP A

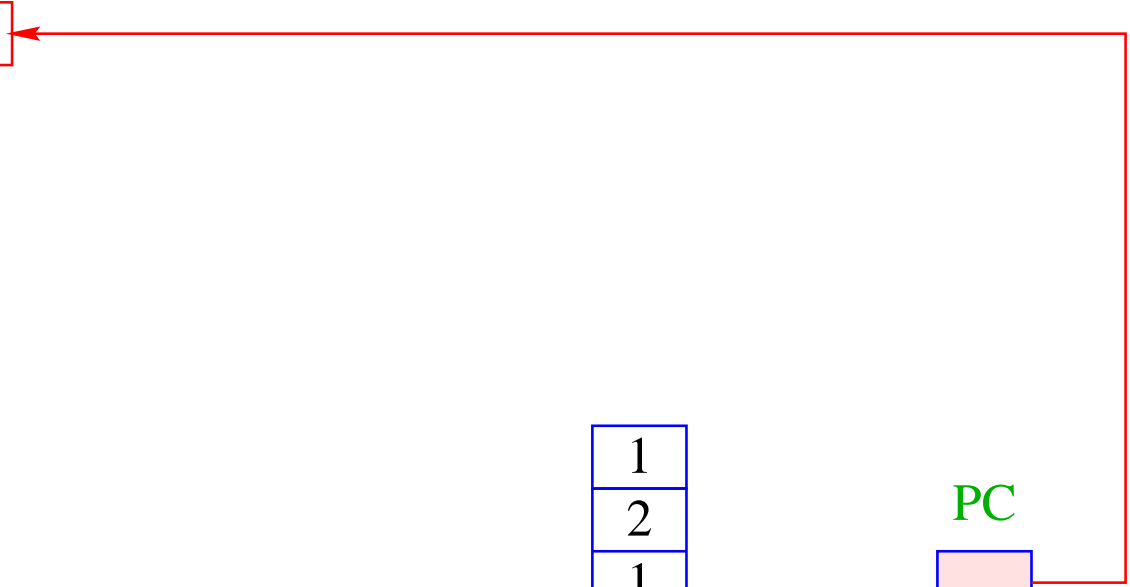
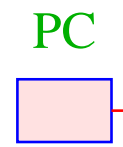
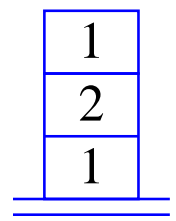
B: LOAD 1
WRITE
HALT





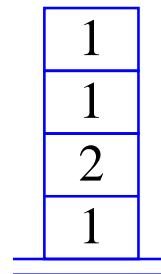
ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B

JUMP A
B: LOAD 1
WRITE
HALT

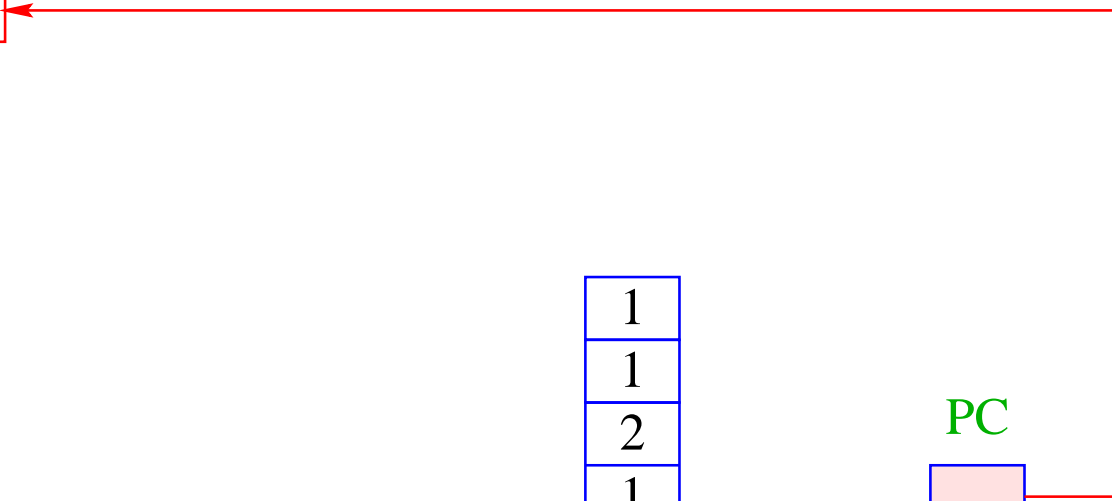


ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B

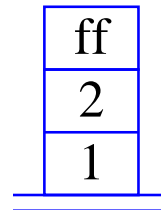
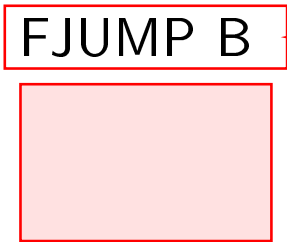
JUMP A
B: LOAD 1
WRITE
HALT



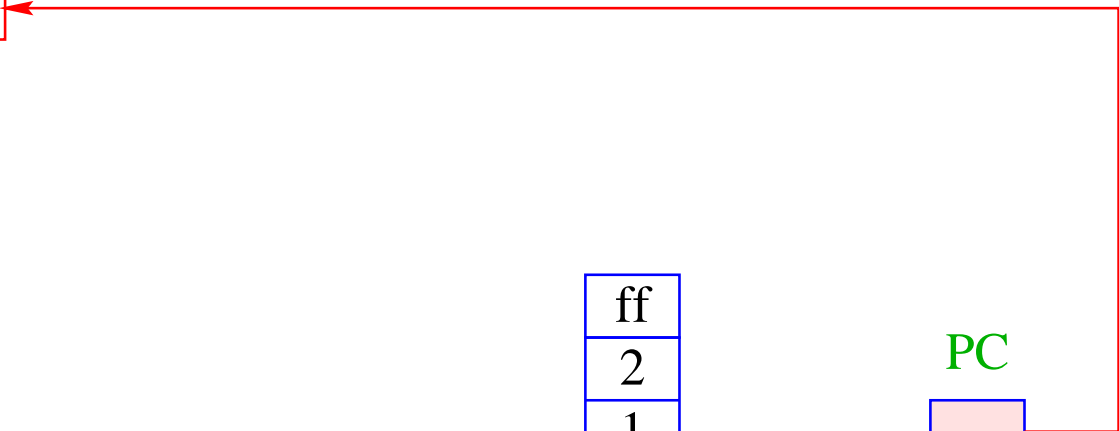
PC



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



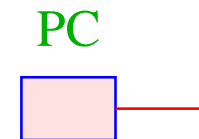
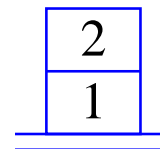
PC



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



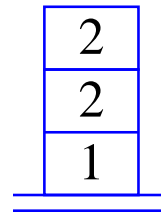
JUMP A
B: LOAD 1
WRITE
HALT



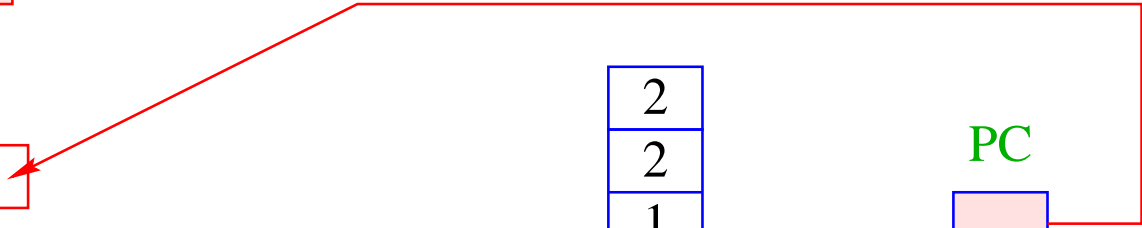
ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B



B: JUMP A
LOAD 1
WRITE
HALT



PC

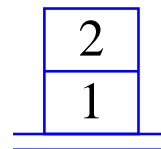


ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B

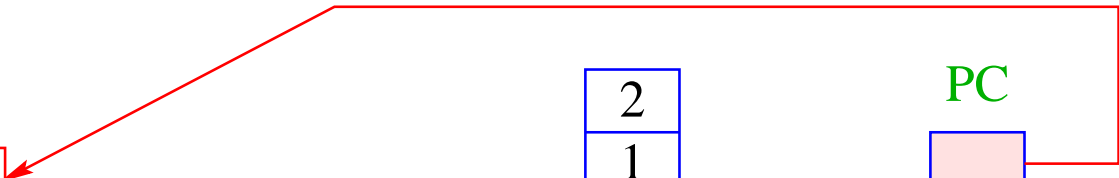


JUMP A
B: LOAD 1
WRITE

HALT



PC



Bemerkungen:

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

↑ **Compilerbau**

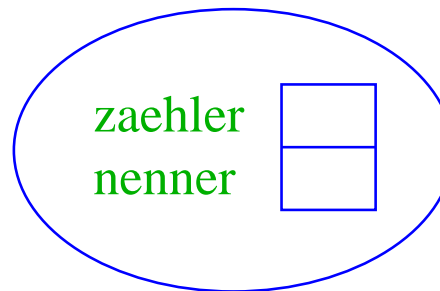
10 Klassen und Objekte

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

Beispiel: Rationale Zahlen

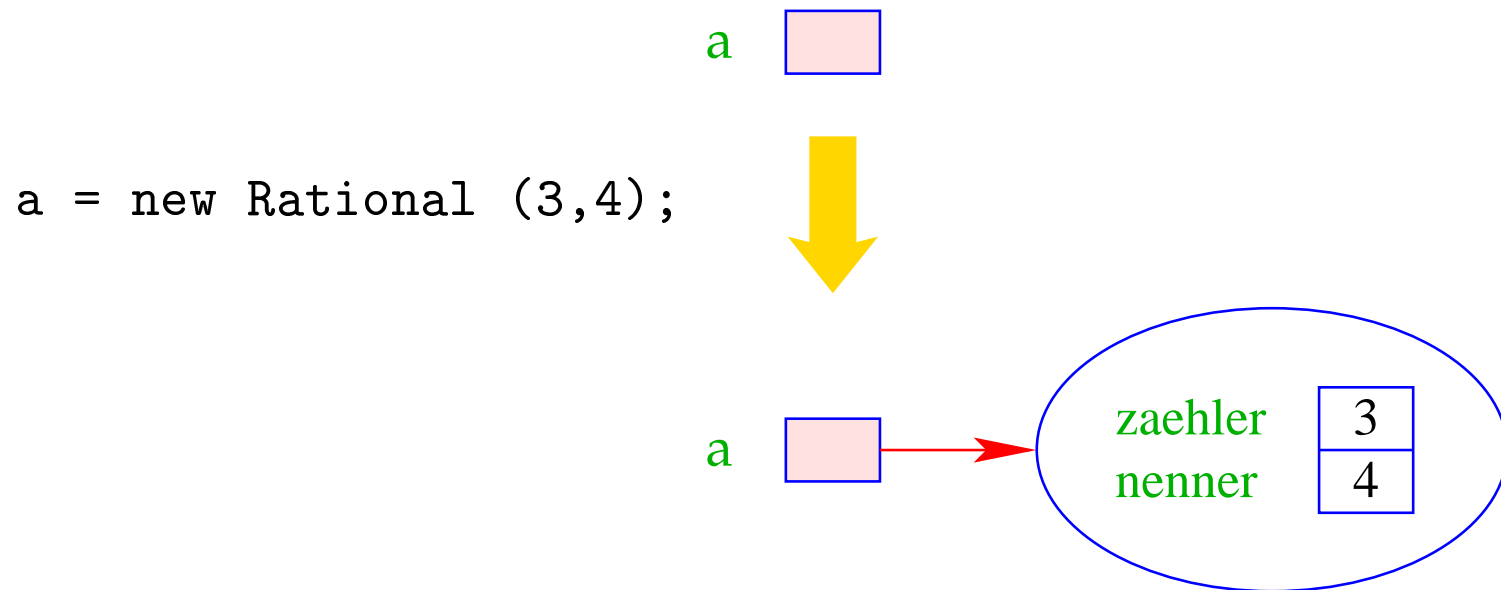
- Eine rationale Zahl $q \in \mathbb{Q}$ hat die Form $q = \frac{x}{y}$, wobei $x, y \in \mathbb{Z}$.
- x und y heißen Zähler und Nenner von q .
- Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` enthalten:

Objekt:



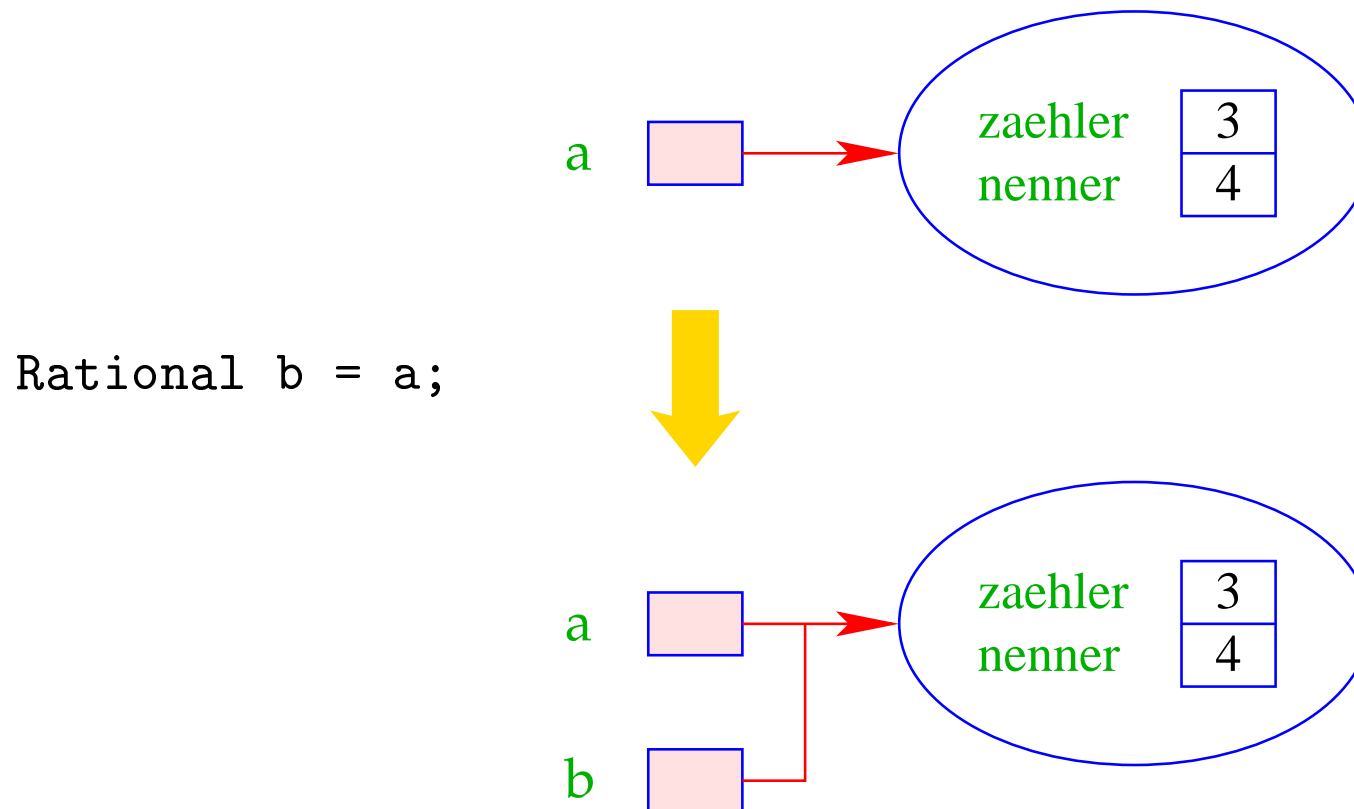
- Die Daten-Komponenten eines Objekts heißen **Instanz-Variablen** oder **Attribute**.

- 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:

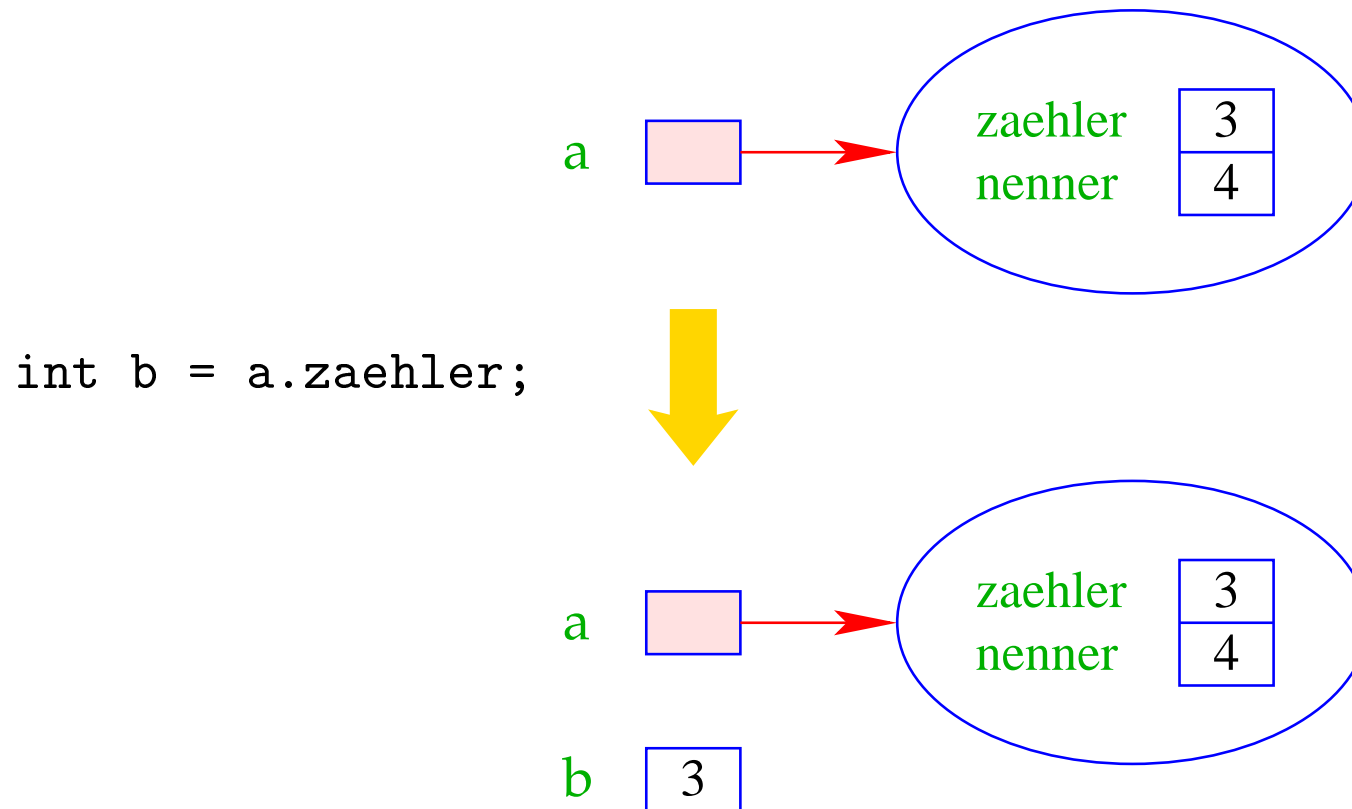


- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objekts initialisieren kann.

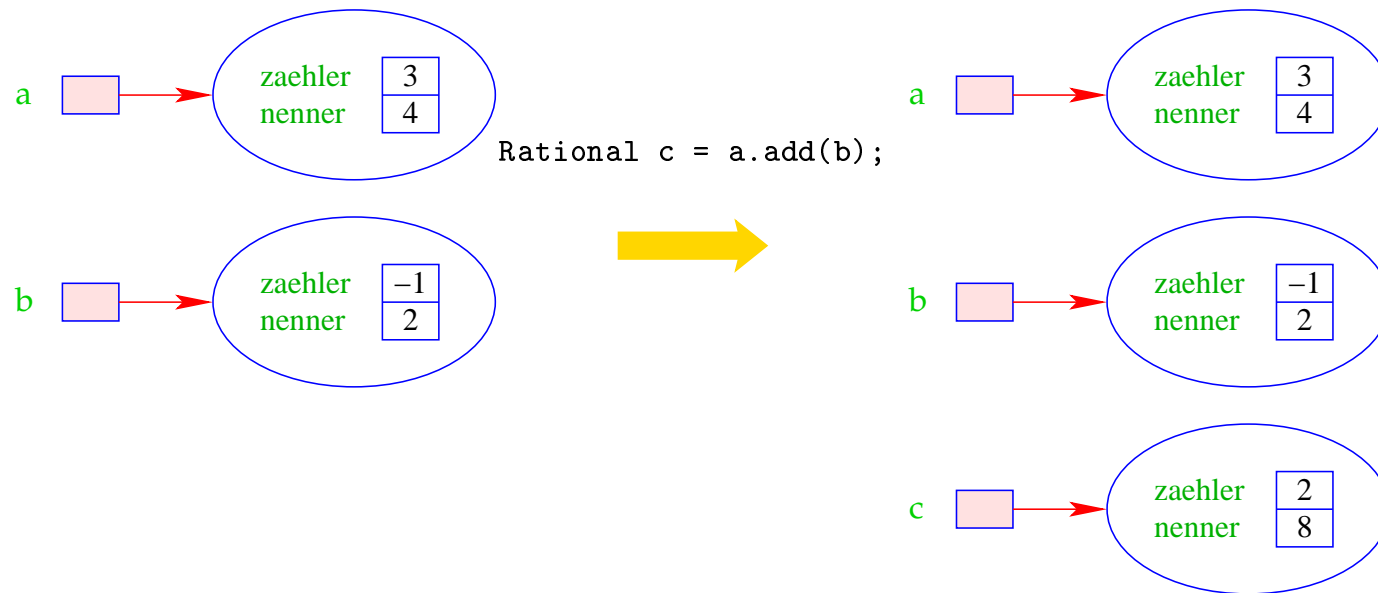
- Der Wert einer Rational-Variable ist ein **Verweis** auf einen Speicherbereich.
- `Rational b = a;` kopiert den Verweis aus `a` in die Variable `b`:

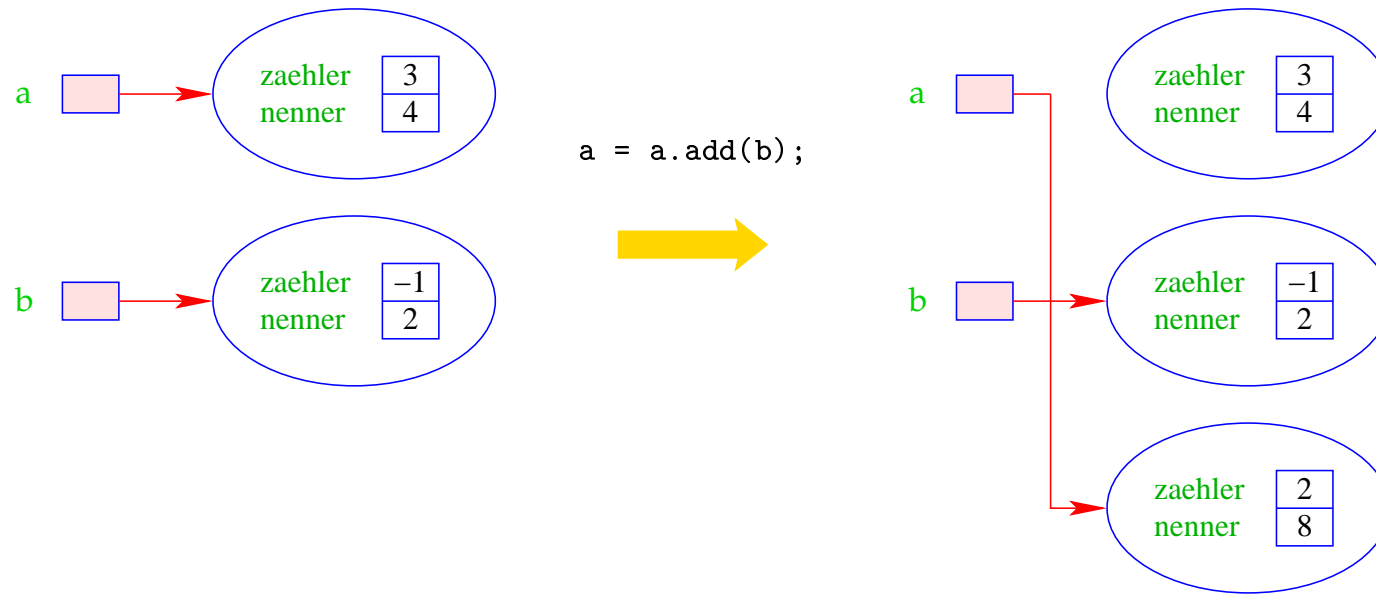


- `a.zaehler` liefert den Wert des Attributs `zaehler` des Objekts `a`:



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





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

Zusammenfassung:

Eine Klassen-Deklaration besteht folglich aus Deklarationen von:

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

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

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

```

// Objekt-Methoden:
public Rational add (Rational r) {
    int x = zaehler * r.nenner + r.zaehler * nenner;
    int y = nenner * r.nenner;
    return new Rational (x,y);
}

public boolean equals (Rational r) {
    return (zaehler * r.nenner == r.zaehler * nenner);
}

public String toString() {
    if (nenner == 1) return "" + zaehler;
    if (nenner > 0) return zaehler + "/" + nenner;
    return (-zaehler) + "/" + (-nenner);
}
} // end of class Rational

```

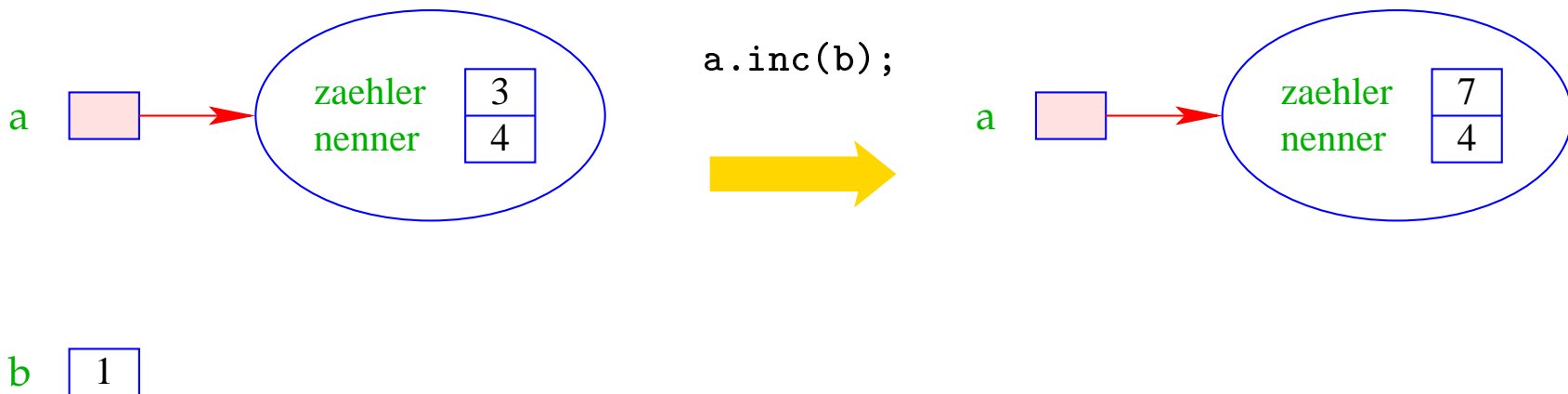
Bemerkungen:

- Jede Klasse **solte** 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 Rückgabotyp.
- Methoden haben dagegen **stets** einen Rückgabe-Typ, evt. void.

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

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



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

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

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

Diskussion und Ausblick:

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

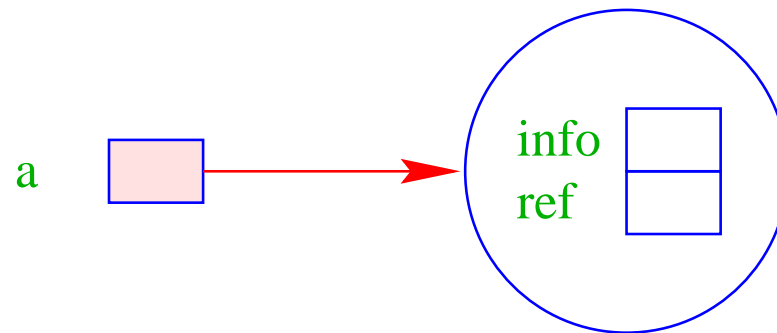
```
public class Cyclic {
    private int info;
    private Cyclic ref;
    // Konstruktor
    public Cyclic() {
        info = 17;
        ref = this;
    }
    ...
} // end of class Cyclic
```

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

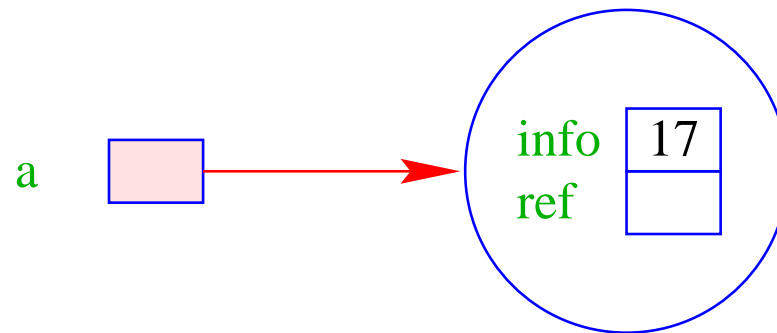
Für `Cyclic a = new Cyclic();` ergibt das:

a 

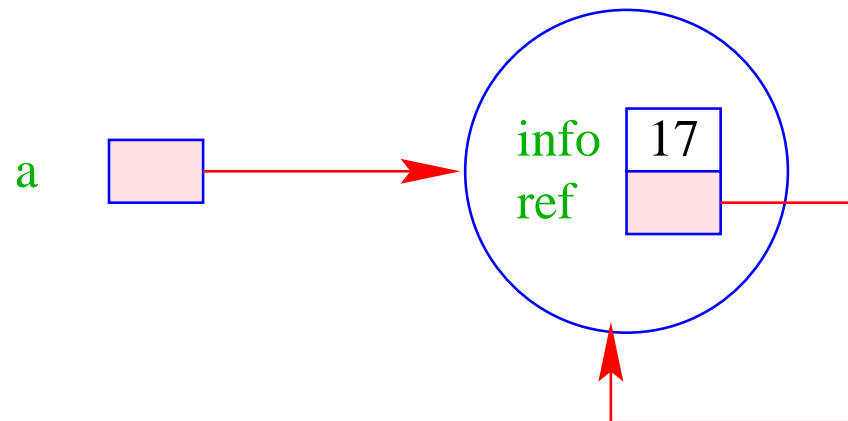
Für `Cyclic a = new Cyclic();` ergibt das:



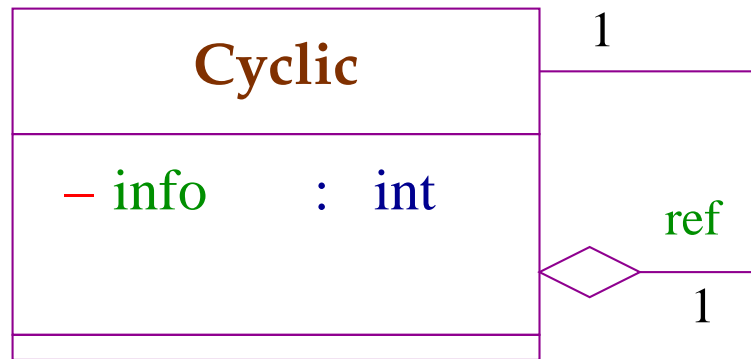
Für `Cyclic a = new Cyclic();` ergibt das:



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 :-)