Informatik I im WS 2013/2014

Prof. Dr. Alexander Pretschner

TUM

Software Engineering, i22

WS 2013/2014

Vorab

Diese Vorlesung baut auf Folien von vorherigen Einführungsvorlesungen an der TUM (Prof. Seidl) und dem KIT (Prof. Pretschner, Prof. Snelting) auf. Das Copyright liegt bei den jeweils Erstellenden.

Inhalt dieser Vorlesung

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

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

Probleme und Algorithmen

Ein Problem besteht darin, aus einer gegebenen Menge von Daten weitere (bisher unbekannte) Daten 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: $f : E \rightarrow A$,
wobei E = zulässige Eingabedaten, A = mögliche Ausgabedaten.

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

Programmieren

Implementieren von Datenbeschreibungen und Algorithmen!

Berechenbarkeit

Achtung:

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

Der Algorithmus (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.*

Programme

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

Beachte:

- Ein Programmsystem berechnet i.a. nicht nur eine Funktion, sondern immer wieder Funktionen in Interaktion mit Benutzerinnen und/oder der Umgebung.
- Es gibt viele Programmiersprachen: Java, C, Prolog, Fortran, Cobol, PostScript ....

Los geht's!

- Im ggT-Beispiel sind die Ein- und Ausgabedaten ganze Zahlen.
- Darauf lässt sich zwar alles reduzieren, aber häufig sind diese Daten komplexer: Profile in sozialen Netzwerken, Einkaufswagen, Buchungen, ...
- ... und Dokumente und Anfragen im Kontext von Suchmaschinen!
- Deswegen beginnen wir mit Datenmodellierung!

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

2. Objektbasierte Programmierung

Objektbasiertes Programmieren

- Die Welt besteht aus Objekten

- Grundidee der Objektbasierung:
  - Jedes Objekt der Realität hat ein virtuelles Gegenstück
  - Objekte kooperieren durch Datenaustausch und Aufrufe von Diensten

- Perspektive:
  - „Wer macht was?“
    $\Longrightarrow$ Eigenschaften und Fähigkeiten von Objekten
    (algorithmische Perspektive: „Wie wird's gemacht?“)

Beispiel: Bibliothek (I)

Objekt der realen Welt

Beispiel: Bibliothek (I)

Modellierung

mmm:Buch

Attribute

autor="Fred Brooks"

title="The mythical man month"

auflage=2

...

Methoden

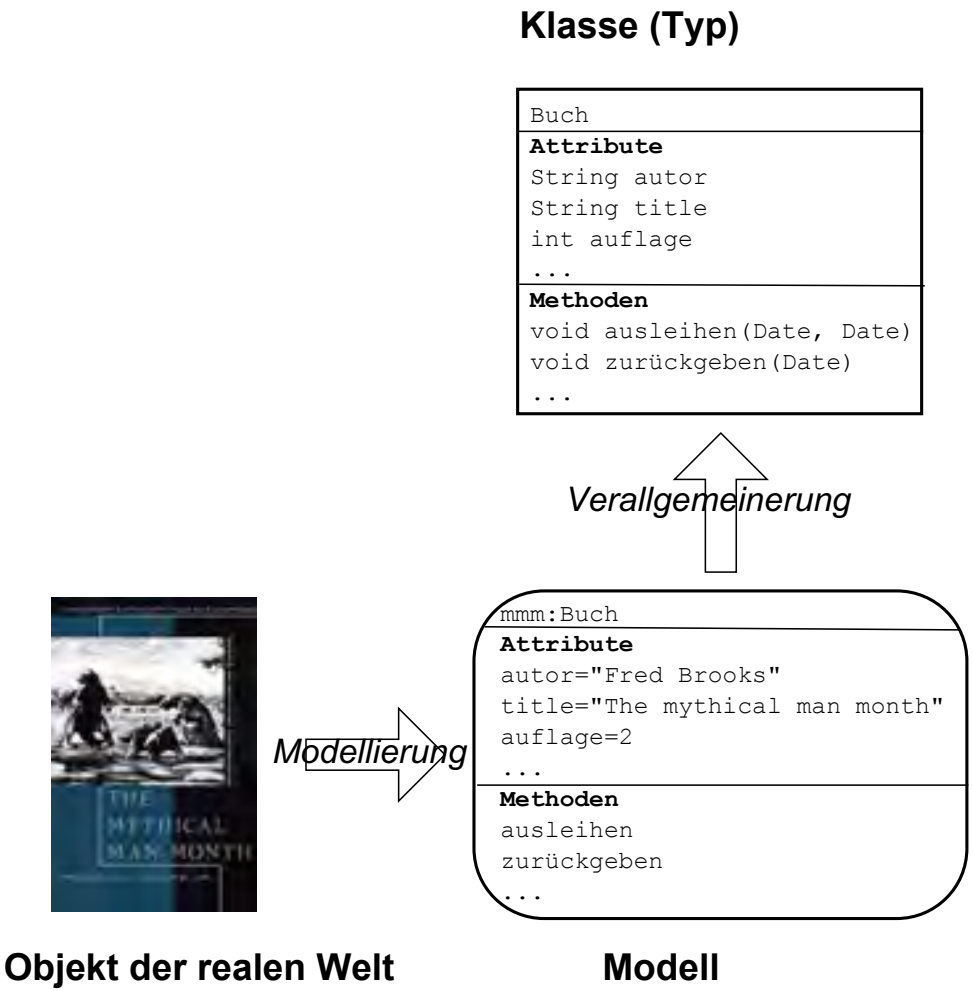
ausleihen

zurückgeben

...

Objekt der realen Welt **Modell**

Beispiel: Bibliothek (I)



Beispiel: Bibliothek (I)

Klasse (Typ)

Buch
Attribute
String autor String title int auflage ...
Methoden
void ausleihen(Date, Date) void zurückgeben(Date) ...

Codierung

Java-Klasse

```
class Buch {
  String autor;
  String title;
  int auflage;
  ...
  void ausleihen (Date a, Date e) {
    ...
  }
  void zurueckgeben(Date today) {
    ...
  }
  ...
}
```

Verallgemeinerung

Modellierung

mmm:Buch
Attribute
autor="Fred Brooks" title="The mythical man month" auflage=2 ...
Methoden
ausleihen zurückgeben ...

Objekt der realen Welt **Modell**

Beispiel: Bibliothek (I)

Klasse (Typ)

Buch
Attribute
String autor String title int auflage ...
Methoden
void ausleihen(Date, Date) void zurückgeben(Date) ...

Codierung

Java-Klasse

```
class Buch {
  String autor;
  String title;
  int auflage;
  ...
  void ausleihen (Date a, Date e) {
    ...
  }
  void zurueckgeben(Date today) {
    ...
  }
  ...
}
```

Verallgemeinerung

Verwendung

Modellierung

mmm:Buch
Attribute
autor="Fred Brooks" title="The mythical man month" auflage=2 ...
Methoden
ausleihen zurückgeben ...

```
Buch mmm = new Buch();
mmm.autor="Fred Brooks";
mmm.title="The mythical man month";
mmm.auflage=2;
```

Objekt der realen Welt **Modell**

Beispiel: Bibliothek (I)

Klasse (Typ)

Buch
Attribute
String autor String title int auflage ...
Methoden
void ausleihen(Date, Date) void zurückgeben(Date) ...

Codierung

Java-Klasse

```
class Buch {
  String autor;
  String title;
  int auflage;
  ...
  void ausleihen (Date a, Date e) {
    ...
  }
  void zurueckgeben(Date today) {
    ...
  }
  ...
}
```

Verallgemeinerung

Verwendung

Modellierung

mmm:Buch
Attribute
autor="Fred Brooks" title="The mythical man month" auflage=2 ...
Methoden
ausleihen zurückgeben ...

Erzeugung

```
Buch mmm = new Buch();
mmm.autor="Fred Brooks";
mmm.title="The mythical man month";
mmm.auflage=2;
```

Objekt der realen Welt

Modell/Instanz

Definition: Objekt und Klasse

Drei Aspeke charakterisieren **Objekte**:

- **Identität**
  Eigenschaften können sich ändern, die Identität ist konstant
  - Durch *Namen (Referenz)* sichergestellt
- **Zustand**
  Menge der Eigenschaften zu einem Zeitpunkt
  - Durch *Attribute* realisiert
- **Verhalten**
  Ausführen von Aktionen, die den Zustand ändern
  - Durch *Methoden* realisiert (später!)

Eine **Klasse** ist ein „Bauplan“ für gleichartige Objekte und legt fest,

- welche **Attribute** die Objekte haben
- welche **Methoden** die Objekte haben

Klassen in Java

```
class Buch {
   // Attribute
   String titel;
   String autor;
   int auflage;
      ...
   // Methoden
      ...
}
```

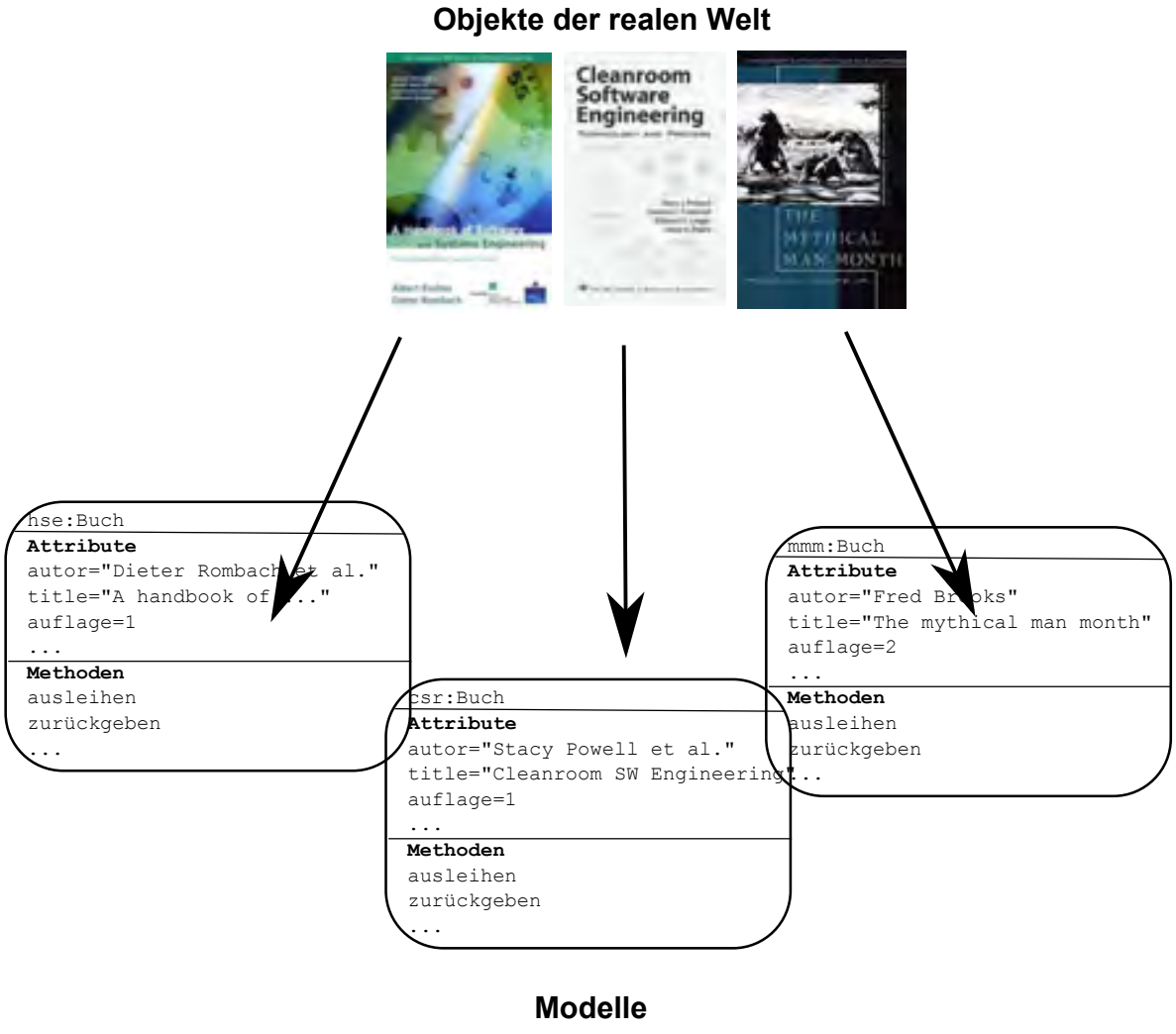
Schema:

- **class** *Name* { *Klassenrumpf* }

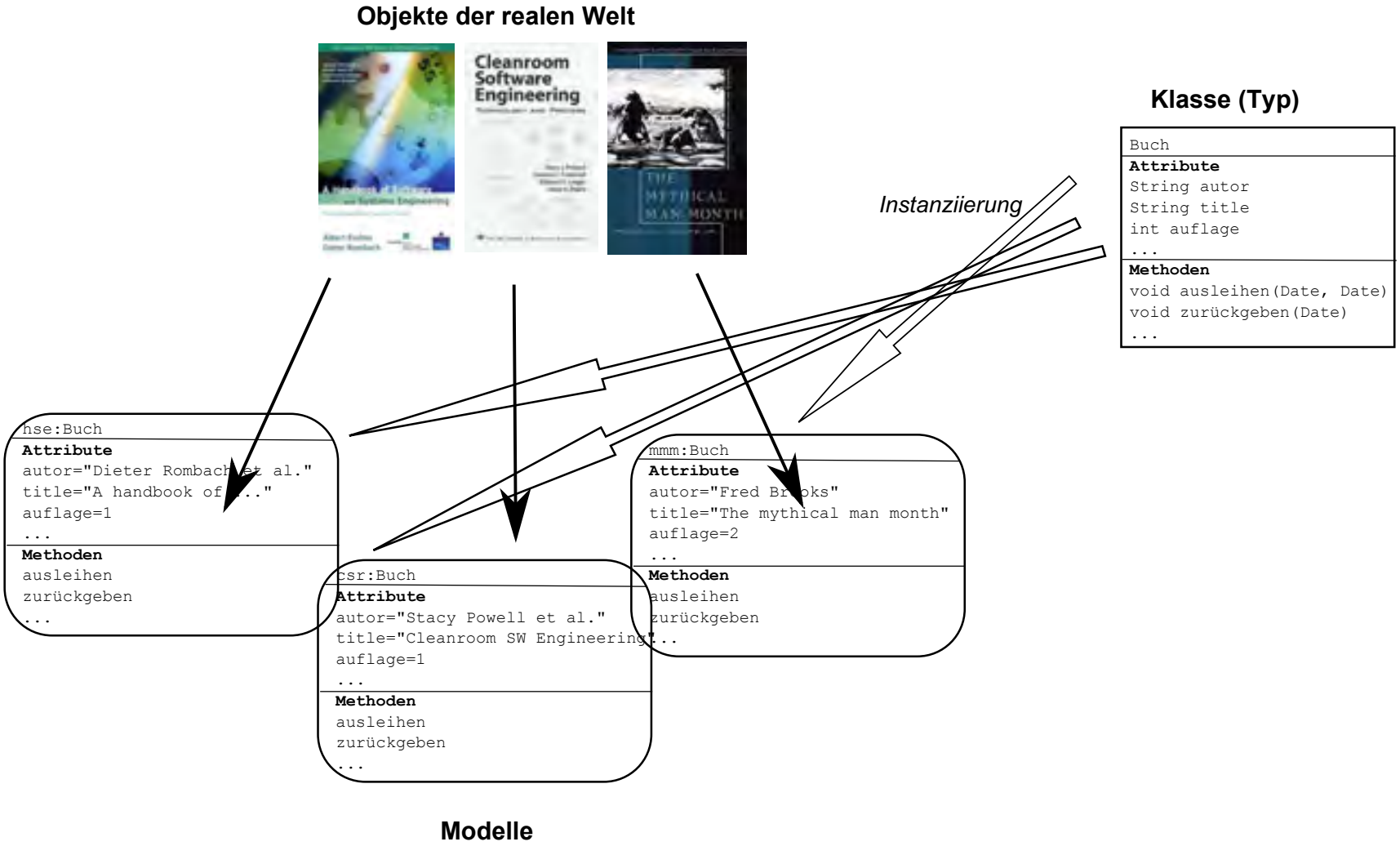
Beispiel: Bibliothek (II)

Objekte der realen Welt

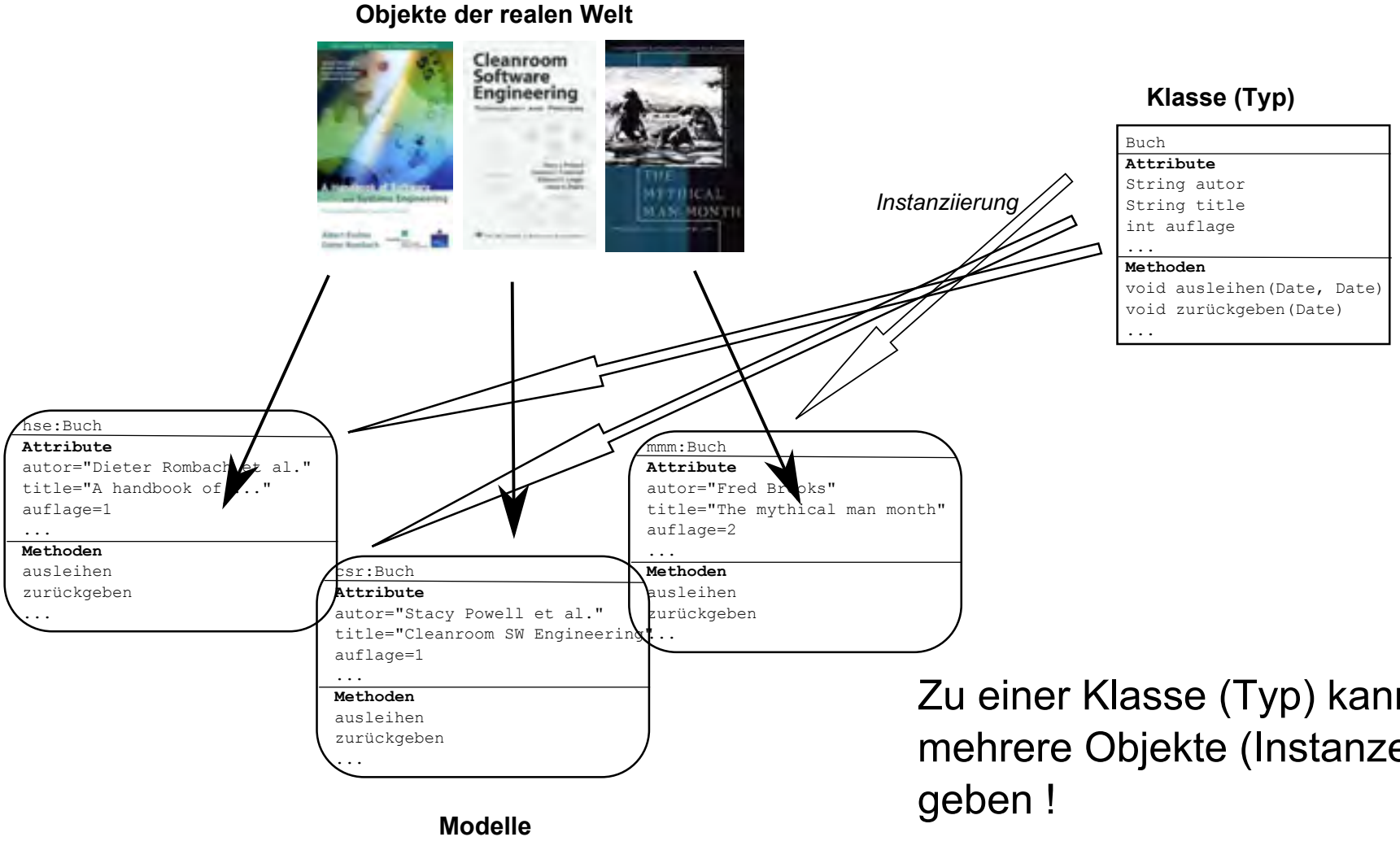
Beispiel: Bibliothek (II)



Beispiel: Bibliothek (II)



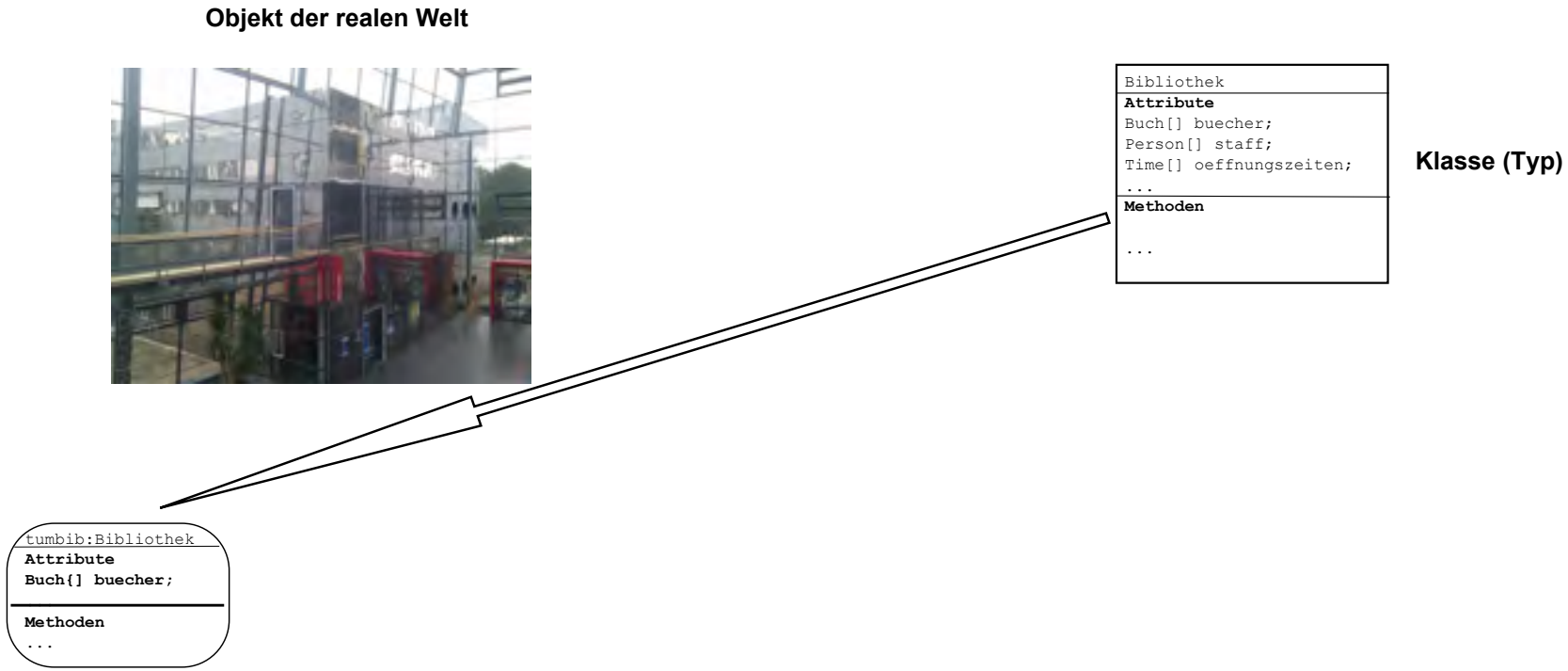
Beispiel: Bibliothek (II)



Zu einer Klasse (Typ) kann es mehrere Objekte (Instanzen) geben !

Beispiel: Bibliothek (III)

Beispiel: Bibliothek (III)

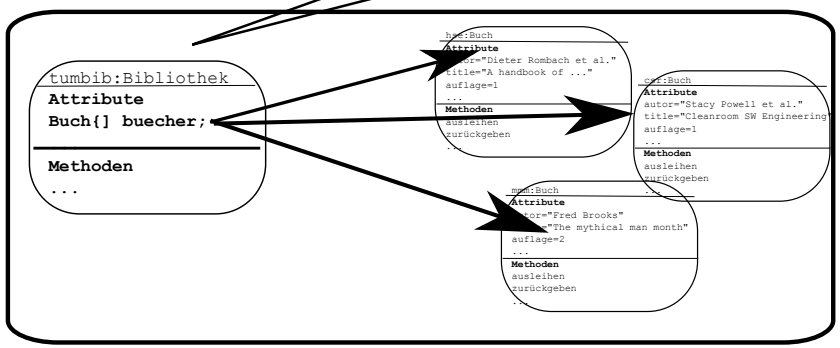


Beispiel: Bibliothek (III)

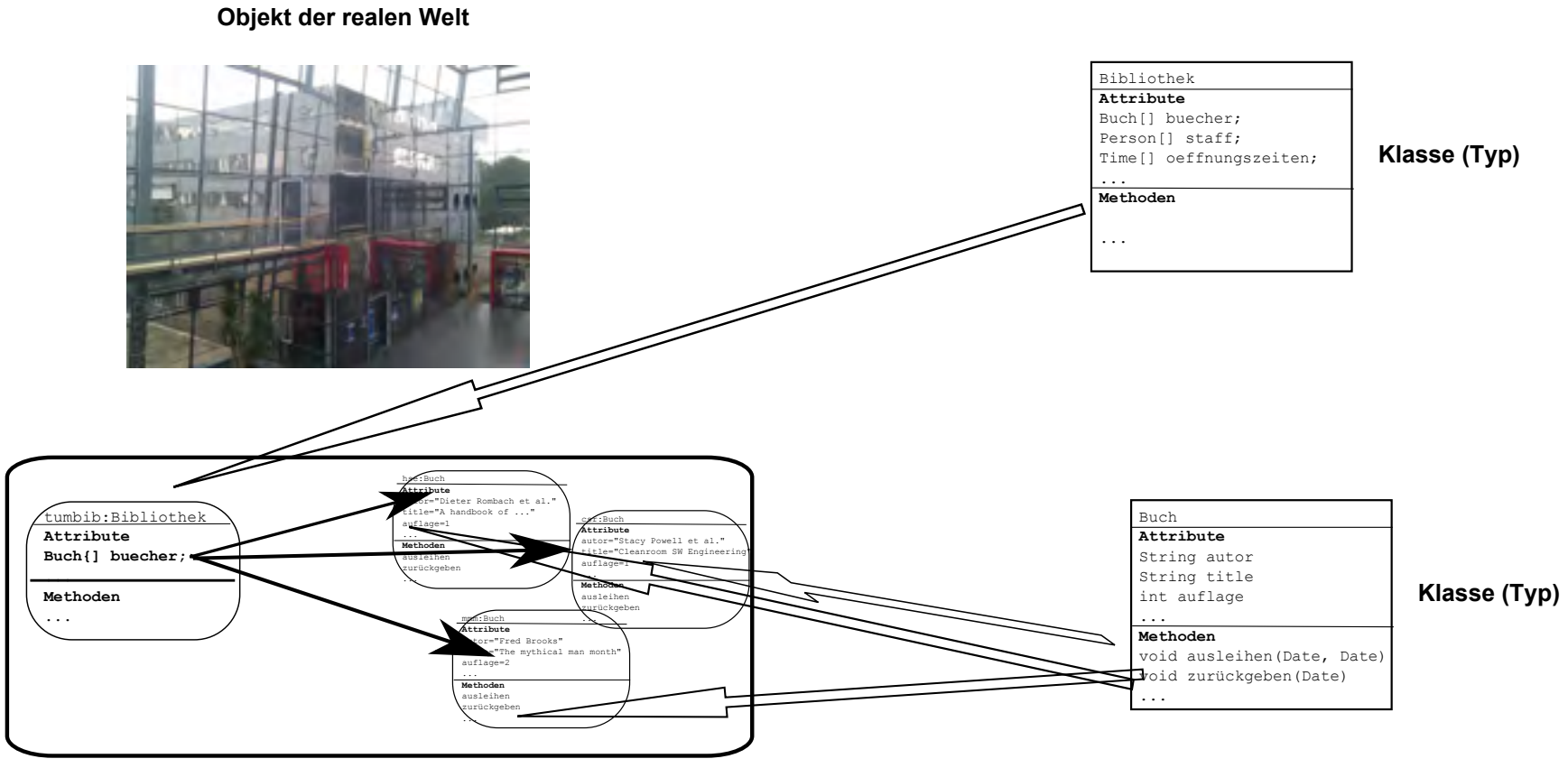
Objekt der realen Welt

```
Bibliothek
Attribute
Buch[] buecher;
Person[] staff;
Time[] oeffnungszeiten;
...
Methoden
...
```

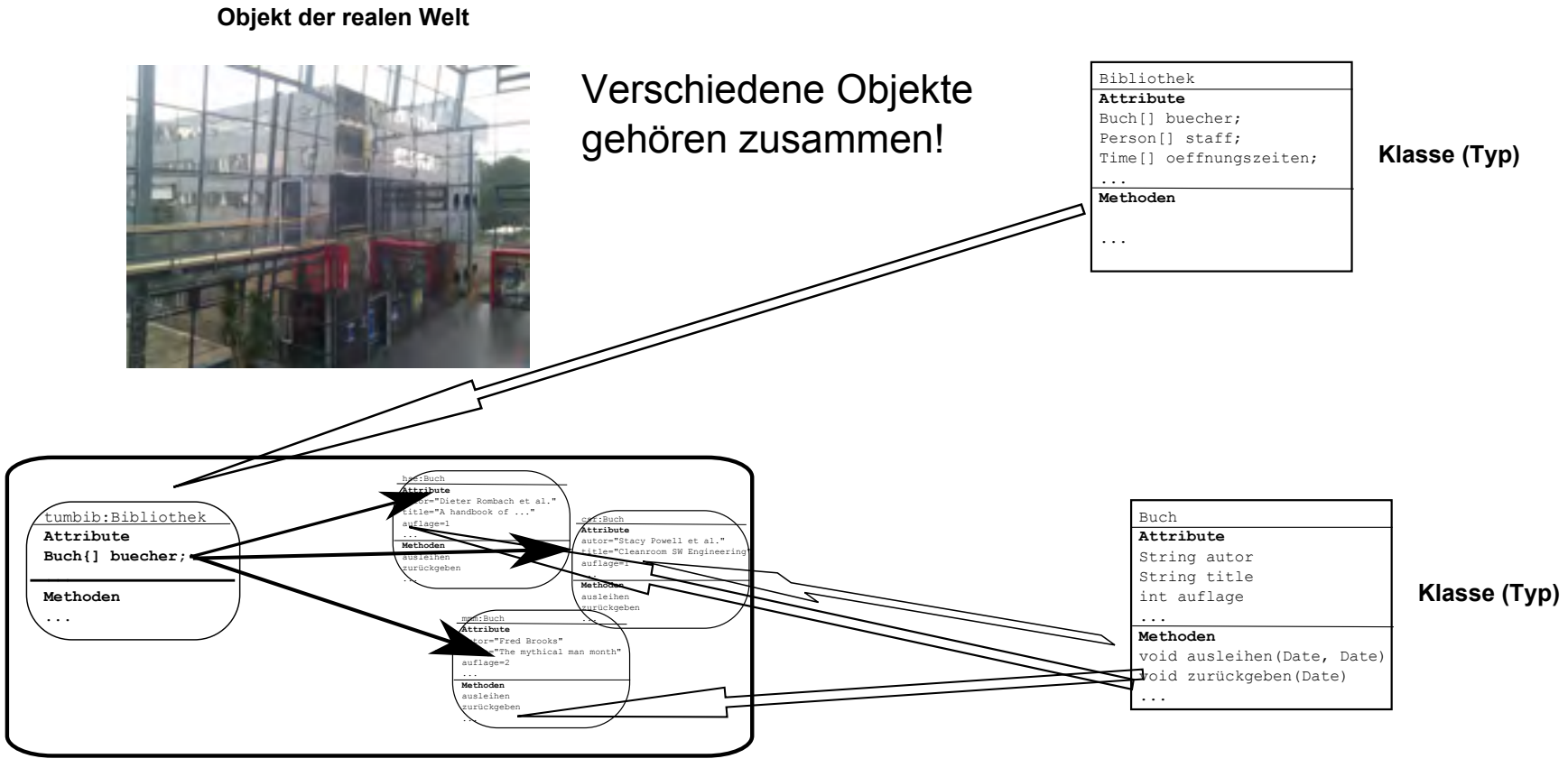
Klasse (Typ)



Beispiel: Bibliothek (III)



Beispiel: Bibliothek (III)



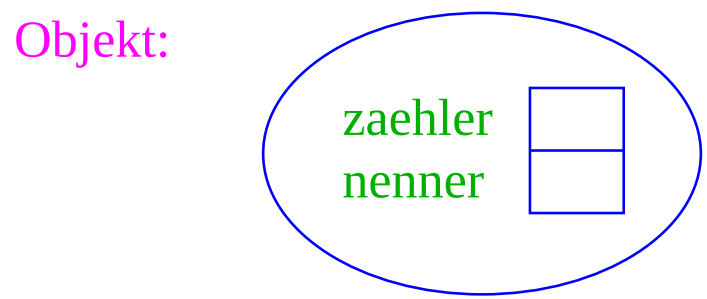
Variablen

- Die Attribute (Daten-Komponenten) eines Objekts heißen auch Instanz-Variablen.
- Später lernen wir weitere Variablen kennen, die
  - für eine Klasse oder
  - lokal für eine Methode

  definiert werden.
- Variablen müssen deklariert werden
- Außer für sog. Basistypen muss noch ein Konstruktor aufgerufen werden.

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 (Attribute) zwei `int`-Variablen `zaehler` und `nenner` enthalten:



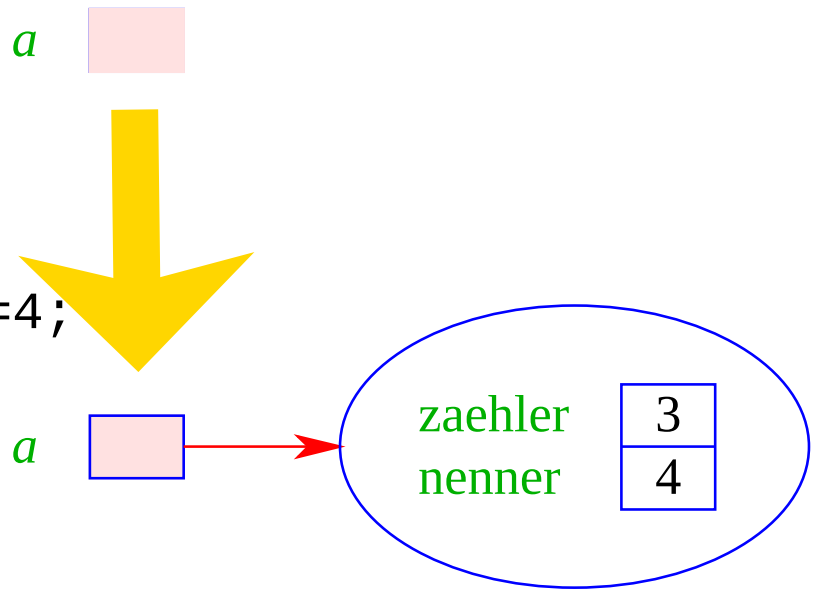
Code

```
class Rational {
  int zaehler;   //Deklaration eines int-Attributs
  int nenner;    //Deklaration eines int-Attributs
  // Konstruktor Rational() existiert automatisch

  // Methoden...
}
```

Deklaration und Konstruktor

- `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:

```
Rational a;
a = new Rational ();
a.zaehler=3; a.nenner=4;
```



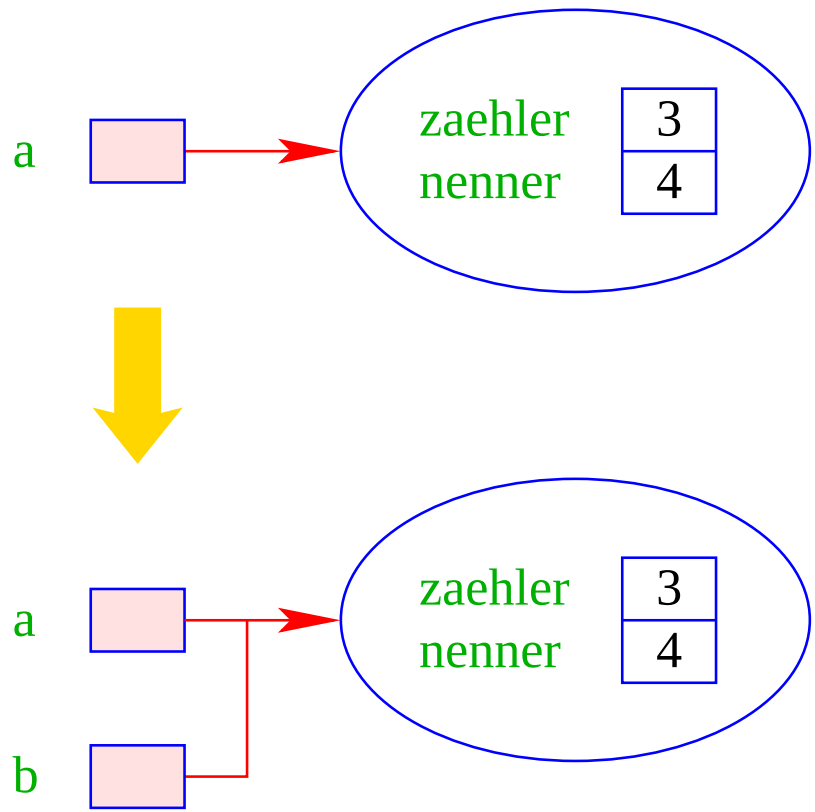
Konstruktoren

- Der Konstruktor ist eine spezielle Methode, die im Zusammenhang mit `new` Speicher reserviert (und Attribute des neuen Objekts initialisieren kann, später!)
- Konstruktoren tragen stets den Namen ihrer Klasse
- Ein Default-Konstruktor `KlassenName()` in der Klasse `KlassenName` wird automatisch zur Verfügung gestelt
- Konstruktoren können auch Argumente haben, betrachten wir später

Referenzen

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

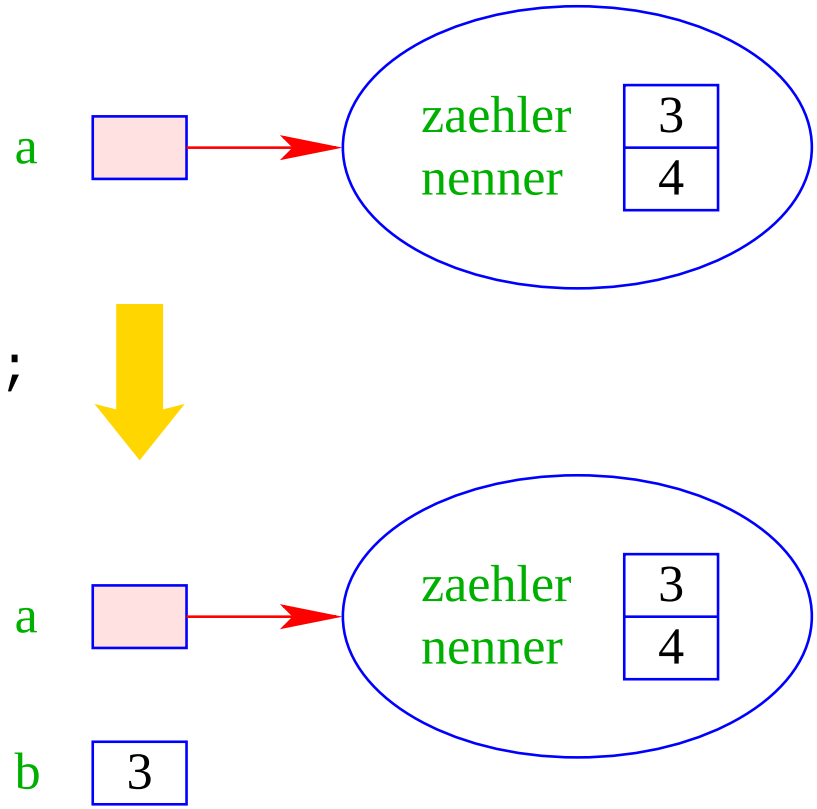
```
Rational b = a;
```



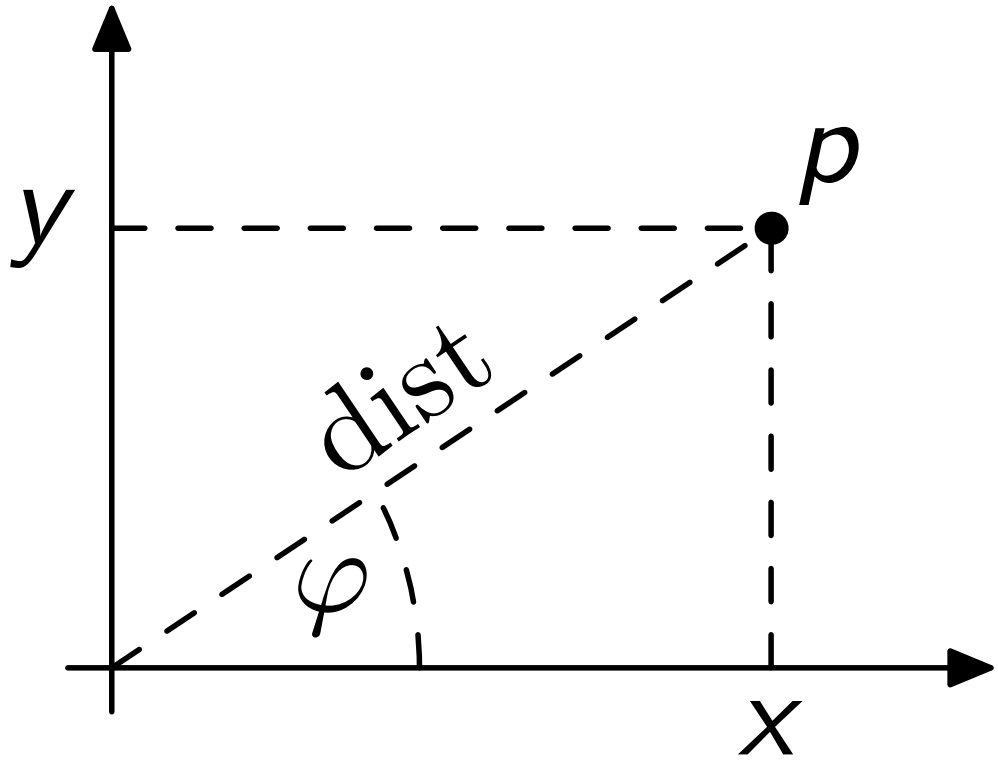
Zugriff auf Attribute

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

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



Beispiel: Punkte im $\mathbb{R}^2$



Code

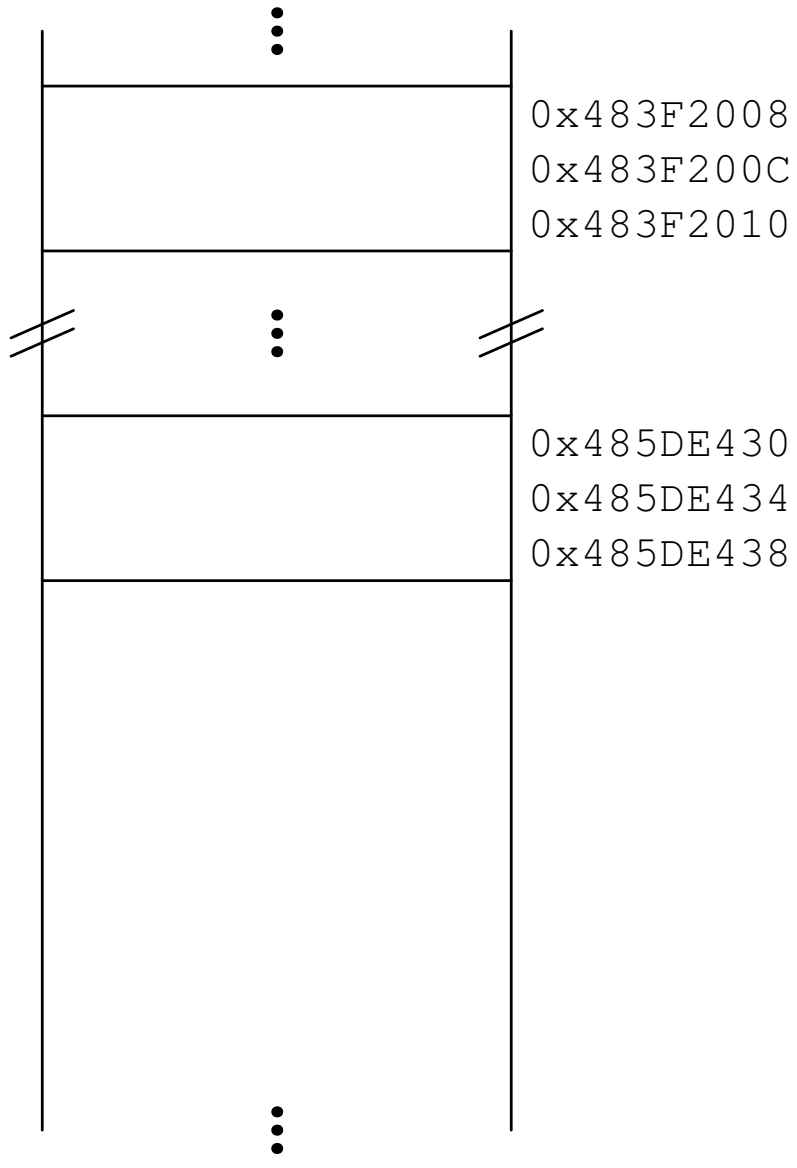
```
class Point {
  double x;
  double y;
  // Konstruktor Point() existiert automatisch
  // Methoden ...
}
```

Punkte erzeugen / Referenzen

- Auch Punkte werden mit `new` erzeugt
  - `Point p = new Point(); // erzeuge Punkt p`
  - `Point q = new Point(); // erzeuge Punkt q`
  - `p`, `q` sind Namen für Referenzen auf neue Objekte der Klasse `Point`
- Auch für einen Punkt kann es mehrere Bezeichner geben
  - `Point r = p // r, p referenzieren dasselbe Objekt`

Objekte im Speicher

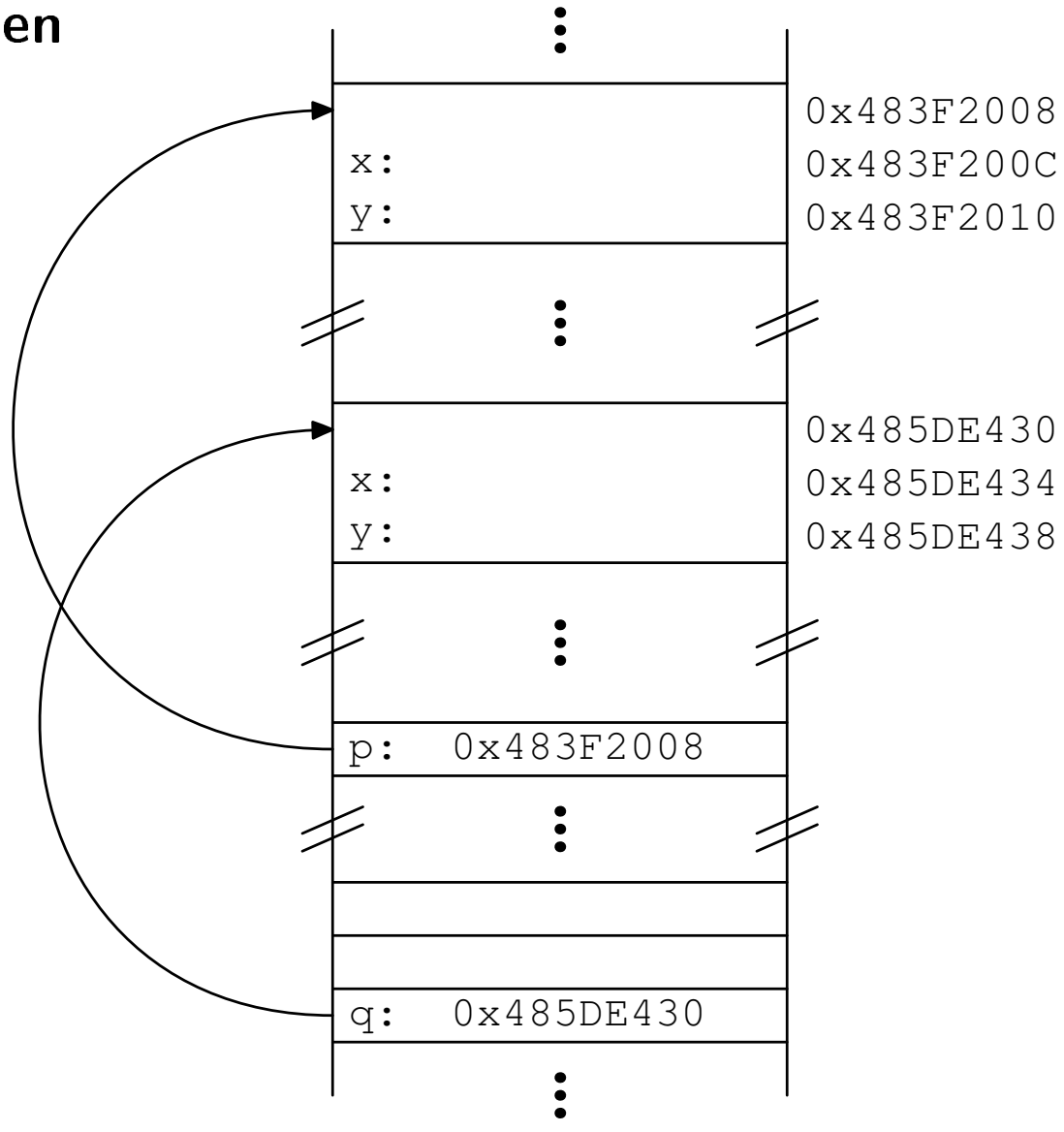
Erzeugen von Objekten



Objekte im Speicher

Erzeugen von Objekten

```
Point p = new Point();
Point q = new Point();
```

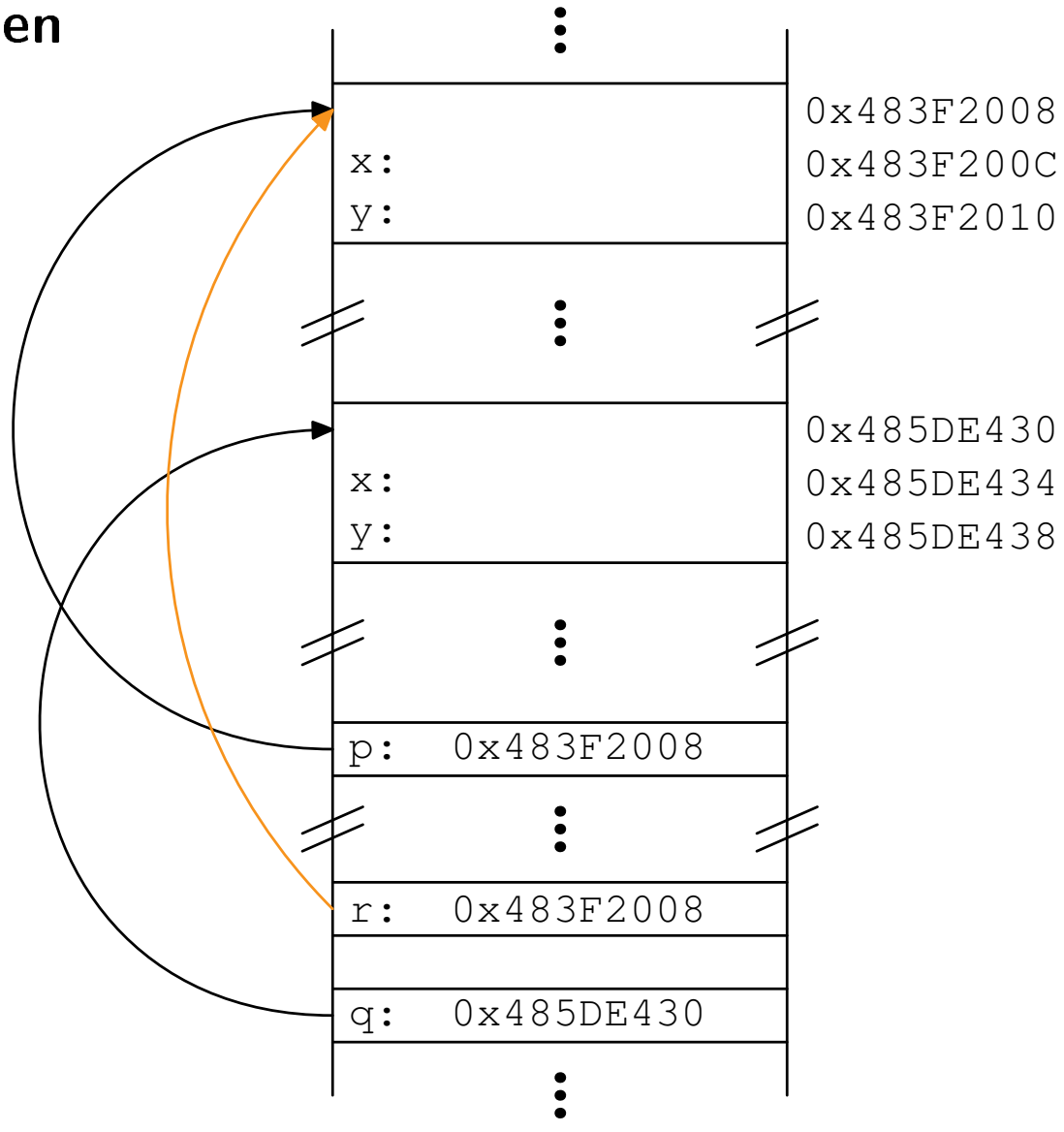


Objekte im Speicher

Erzeugen von Objekten

(Beispiel für „Aliasing“)

```
Point p = new Point();
Point q = new Point();
Point r = p;
```

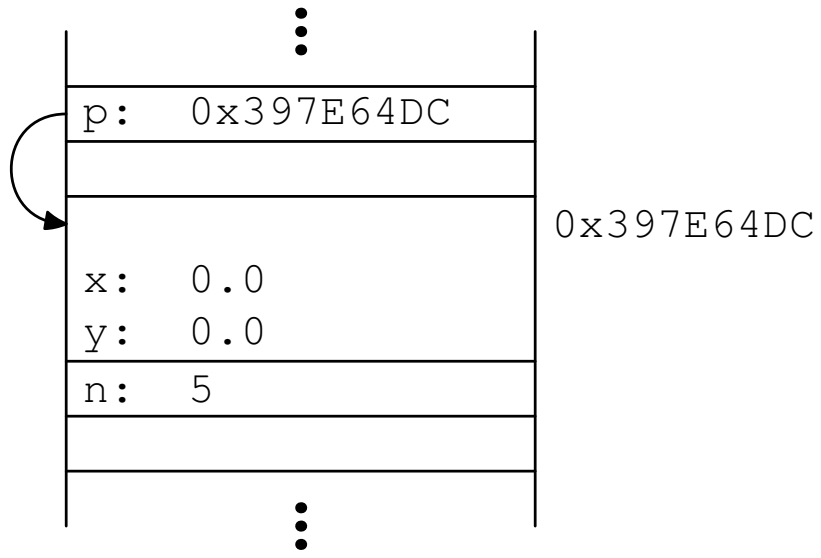


Funktionsweise von Referenzen (I)

- Betrachte zwei Deklarationen lokaler Variablen
  - `Point p;`
  - `int n;`
- Beide Variablen sind (noch) nicht initialisiert
- Initialisiere:
  - `p = new Point();`
  - `n = 5;`
- „`new Point()`“ erzeugt (auf dem Heap) ein Objekt des Typs `Point` und liefert einen Verweis auf das Objekt. Dieser Verweis wird in der Referenz-Variablen `p` gespeichert.
- „`n = 5`“ speichert den Wert `5` direkt in der Variablen `n`

Funktionsweise von Referenzen (II)

- Betrachte zwei weitere Variablen-Deklarationen
  - ```
    Point p;
    p=new Point();
    Point q = p;
    ```
  - ```
    int n;
    n = 5;
    int k = n;
    ```

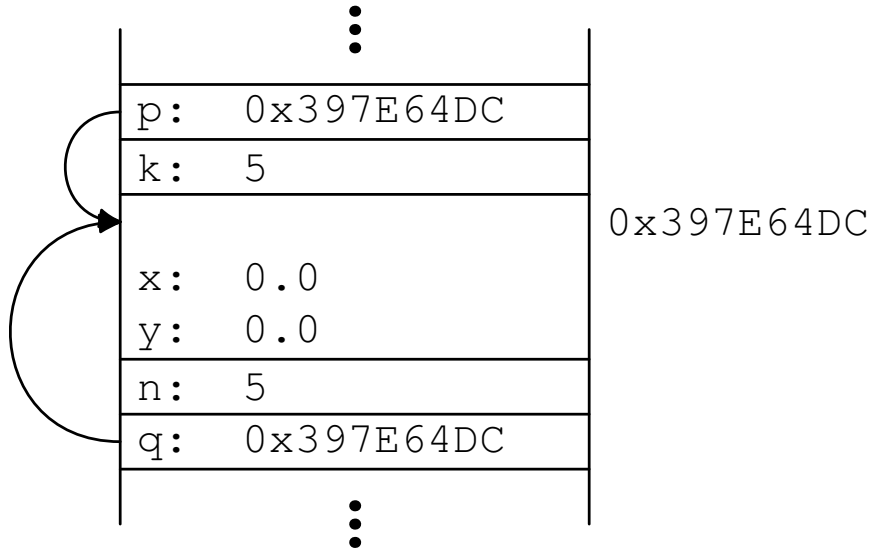


- Durch die Zuweisung „`q = p`“ bzw. „`k = n`“ wird
  - in der Variablen `q` der Wert der Variablen `p` gespeichert
  - in der Variablen `k` der Wert der Variablen `n` gespeichert
- Im Falle einer Referenz ist dies der Verweis auf das jeweilige Objekt
- Im Falle eines primitiven Typs ist es der Wert selbst.

$\Longrightarrow$ `p` und `q` verweisen auf ein und dasselbe Objekt

Funktionsweise von Referenzen (II)

- Betrachte zwei weitere Variablen-Deklarationen
  - ```
    Point p;
    p=new Point();
    Point q = p;
    ```
  - ```
    int n;
    n = 5;
    int k = n;
    ```



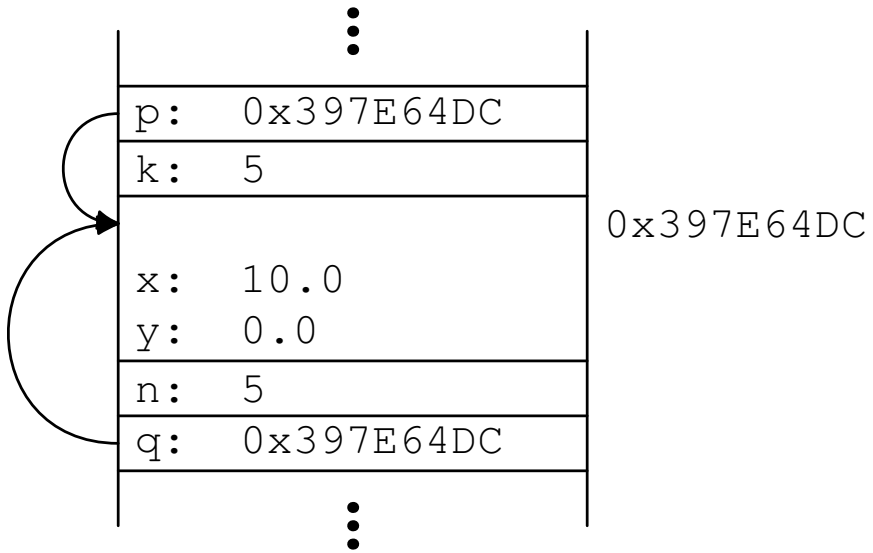
- Durch die Zuweisung „`q = p`“ bzw. „`k = n`“ wird
  - in der Variablen `q` der Wert der Variablen `p` gespeichert
  - in der Variablen `k` der Wert der Variablen `n` gespeichert
- Im Falle einer Referenz ist dies der Verweis auf das jeweilige Objekt
- Im Falle eines primitiven Typs ist es der Wert selbst.

$\Longrightarrow$ `p` und `q` verweisen auf ein und dasselbe Objekt

Funktionsweise von Referenzen (III)

`p` und `q` verweisen auf ein und dasselbe Objekt

- Betrachte:
  - `q.x = 10;`

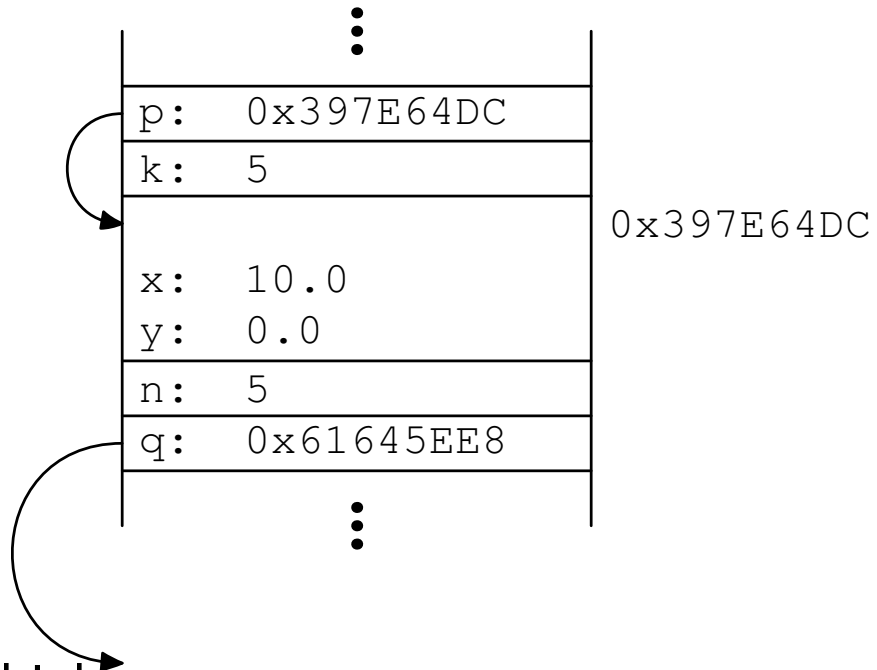


- Welchen Wert hat dann `p.x`?

Funktionsweise von Referenzen (III)

`p` und `q` verweisen auf ein und dasselbe Objekt

- Betrachte:
  - `q.x = 10;`
  - `q = new Point(3,5);`



- Welchen Wert hat jetzt `p.x`?
- `q` verweist jetzt auf ein neues Objekt, `p` aber immernoch auf das zuerst erzeugte.

Gleichheit und Identität

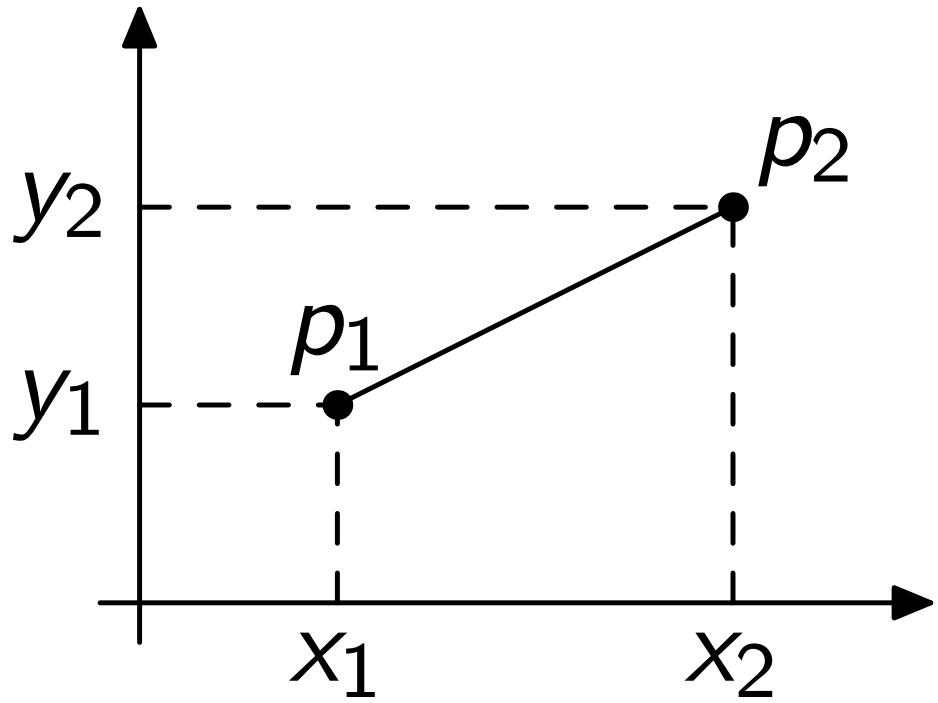
Wann sind zwei Objekte “gleich”?

- Wenn sie identisch sind, also dasselbe Objekt: Gleichheit der Referenzen.
  Überprüfung mit ==.
- Wenn die Attributwerte gleich sind, es sich aber ggf. um zwei unterschiedliche Objekte handelt.
  Überprüfung mit eigener Methode, kommt später.

`null`

- Der Standardwert für alle Referenz-Typen ist **`null`**
- Wurde eine Referenz-Variable oder ein Referenz-Attribut nicht initialisiert, so hat diese(s) also den Wert **`null`**
- Hat eine Variable den Wert **`null`**, so verweist sie auf *kein* Objekt
- z.B.: `Point p;  // jetzt gilt: p == null`
- Zugriff auf `p.x` und `p.y` mit `p==null` führt zu Laufzeitfehler!

Beispiel: Objekte als Attribute von Objekten—Linien im $\mathbb{R}^2$



Code

```
class Line {
  // Attribute: Endpunkte
  Point p1;
  Point p2;

  // Konstruktor Line() existiert automatisch
  // Ein weiterer Konstruktor
  // explizit mit zwei Argumenten!
  Line(Point p, Point q) {
    p1=p;
    p2=q;
  }

  // andere Methoden
  ...
}
```

Code: Objekterzeugung

```
Point p1;
p1 = new Point();
p1.x = 2.0; p1.y = 2.7;
Point p2 = new Point();
p2.x = 5.8; p2.y = 16.0;

Line l = new Line(p1, p2);
```

Nocheinmal: Gleichheit und Identität

Wann sind zwei Objekte “gleich”?

- Wenn sie identisch sind, also dasselbe Objekt: Gleichheit der Referenzen.
  Überprüfung mit ==.
- Wenn die Attributwerte gleich sind, es sich aber ggf. um zwei unterschiedliche Objekte handelt.
  Überprüfung mit eigener Methode, kommt später.

Wenn im zweiten Fall Attribute selbst Verweise auf Objekte sind, muss diesen Verweisen für die Überprüfung auf Gleichheit rekursiv gefolgt werden!

Klassen-Attribute

- Objekt-Attribute werden für jedes Objekt neu angelegt,
- Klassen-Attribute einmal für die gesamte Klasse.

```
class Count {
    private static int count = 0;
    private int info;
    // Konstruktor
    Count() {
        info = count; count = count+1;
        } ...
    } // end of class Count
```

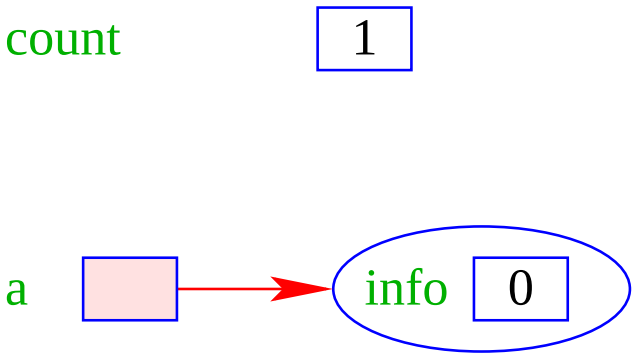
- `private` Attribute sind nur innerhalb der Klasse sichtbar!
- Klassen-Attribute einmal für die gesamte Klasse.
- Klassen-Attribute erhalten die Qualifizierung `static`.

Beispiel Klassen-Attribute

count 0

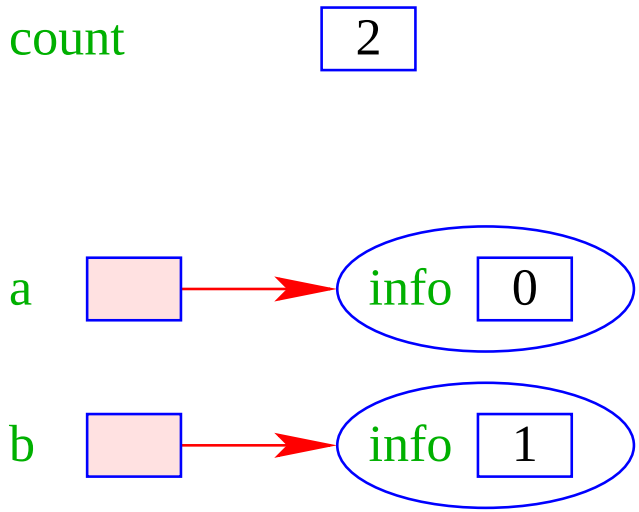
```
Count a = new Count();
```

Beispiel Klassen-Attribute



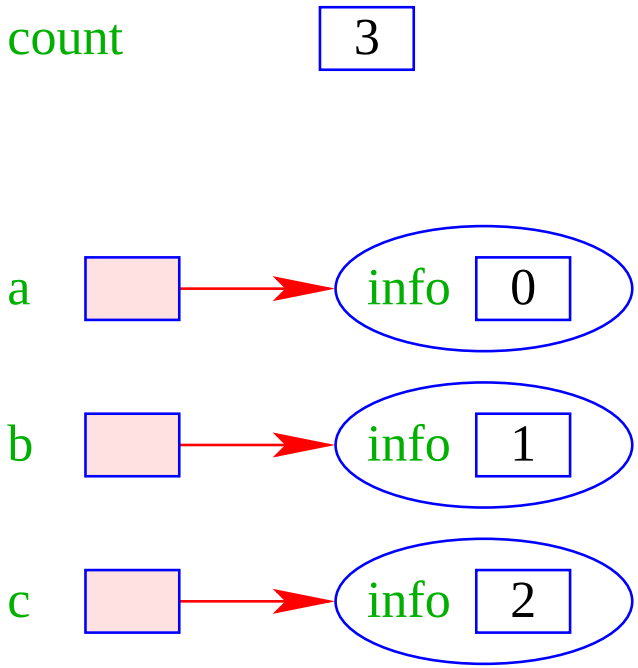

```
Count b = new Count();
```

Beispiel Klassen-Attribute




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

Beispiel Klassen-Attribute



Beispiel Klassen-Attribute

- 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.

Geheimnisprinzip

Das **Geheimnisprinzip** ist ein wichtiges Konzept der Informatik

Warum werden in der Klasse `Line` *die Endpunkte nicht durch zwei Koordinatenpaare* `x1, y1` *und* `x2, y2` *repräsentiert?*

- Wechsel der Implementierung von `Point` würde Wechsel der Implementierung von `Line` erfordern (z.B. bei Wechsel auf Polarkoordinaten)

$\Longrightarrow$ Die Implementierung von Punkten ist **allein das Geheimnis** der Klasse `Point`

Nur so können die Auswirkungen von Änderungen lokal begrenzt werden (**Lokalitätsprinzip**)

Zusammenfassung

- Die (virtuelle) Welt besteht aus Objekten.
- Objekte kapseln Daten (Attribute) und Verhalten (Methoden).
- Der Zugriff auf Objekte erfolgt über Referenzen. Referenzen verweisen auf Speicherbereiche.
- Referenzen werden durch Attributdeklarationen definiert. Attribute können selbst Objekte sein.
- Speicher wird durch `new` bereitgestellt. `new` ruft immer einen Konstruktor auf, der Initialisierungen vornimmt.
- Zwei Referenzen können auf denselben Speicherbereich zeigen (Aliasing).

Übersicht

Basistypen

- Ein Datentyp (kurz: Typ) legt eine Menge möglicher Werte und die möglichen Operationen (Methoden) auf diesen Typen fest.
- Alle Variablen und Attribute haben einen Typen (und Methoden haben einen Rückgabetypen und Typen für die Argumente).
- In Java definieren Klassen Datentypen.
- Java stellt außerdem acht Basistypen zur Verfügung. (`int` und `double` haben wir schon kennengelernt, ohne sie einzuführen.)
- 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?)

Ganze Zahlen

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.

Ganze Zahlen

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;
System.out.println(x);
```

... liefert `-2147483648` ...

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

Gleitkommazahlen

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.

Verschiedene Wertdarstellungen

- Das Präfix `0` signalisiert oktale Werte, `0x` hexadezimale Werte
- `18` dezimal ist `022` oktal und `0x12` hexadezimal
- `65535` dezimal ist `0177777` oktal und `0xFFFF` hexadezimal.

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'`.

Basistypen und Konstruktoren

Weil sie so oft benutzt werden und ihre Implementierung vergleichsweise effizient ist, muss für Basistypen nicht explizit mit `new` Speicher reserviert und kein Konstruktor aufgerufen werden!

Es reicht also beispielsweise, `int x=2;` zu schreiben. Damit wird insgesamt nur ein Speicherbereich reserviert (für den Wert), nicht zwei wie im Fall von Referenztypen (Referenz+Objekt).

Haben wir schon im Zusammenhang mit der Speicherverwaltung gesehen!

Und ein drittes Mal: Gleichheit und Identität

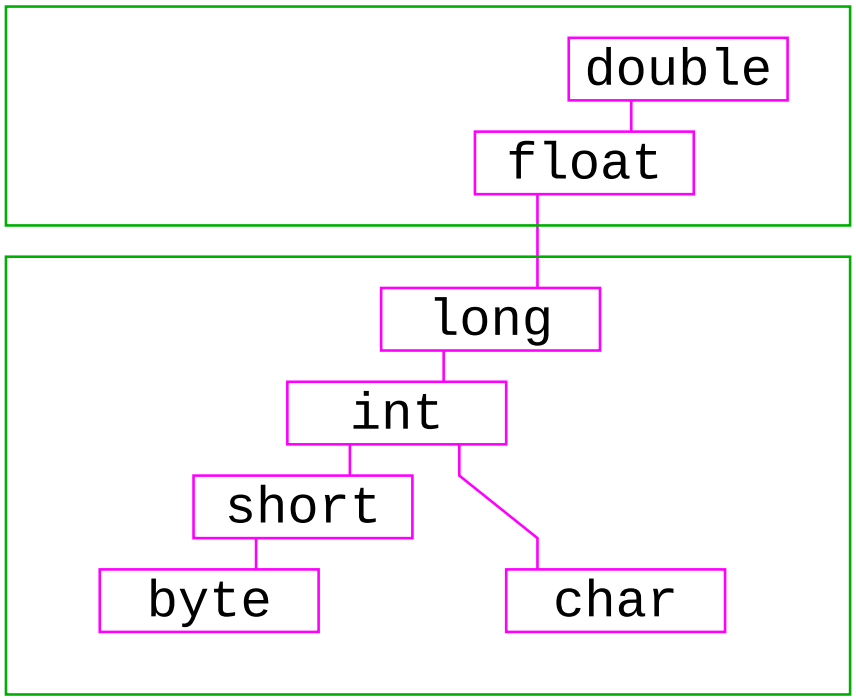
Wann sind zwei Objekte eines Basistypen “gleich”?

Für Basistypen wird der Wert direkt gespeichert, ohne Referenz. Überprüfung der Gleichheit des Werts für Basistypen deshalb mit ==. Bei Referenztypen wird mit == die Referenz verglichen.

Basisoperationen

- 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) ...

Basistypen



Gleitkomma–Zahlen

ganze Zahlen

Beispiel

```
short xs = 1;
int x = 999999999;
System.out.println(x + xs);
```

... liefert den `int`-Wert `1000000000` ...

```
float xs = 1.0f;
int x = 999999999;
System.out.println(x + xs);
```

... liefert den `float`-Wert `1.0E9` ...

Beispiel

```
short xs = 1;
int x = 999999999;
System.out.println(x + xs);
```

... liefert den `int`-Wert `1000000000` ...

```
float xs = 1.0f;
int x = 999999999;
System.out.println(x + xs);
```

... liefert den `float`-Wert `1.0E9` ...

`System.out.println()` kann Gleitkomma-Zahlen ausgeben.

Achtung

- Das Ergebnis einer Operation auf `float` kann aus dem Bereich von `float` herausführen. Dann ergibt sich der Wert `Infinity` oder `-Infinity`.

  Das gleiche gilt für `double`.
- Das Ergebnis einer Operation auf Basistypen, die in `int` enthalten sind (außer `char`), liefern ein `int`.
- Wird das Ergebnis einer Variablen zugewiesen, sollte deren Typ dies zulassen.

Elementare Operationen

Präzedenz	Operator	Beschreibung
	Arithmetische und Vergleichs-Operatoren	
1	`+x, -x`	unäres Plus/Minus
2	`x*y, x/y, x%y`	Multiplikation, Division, Modulo
3	`x+y, x-y`	Addition, Subtraktion
5	`x<y, x<=y, x>y, x>=y`	Größenvergleiche
6	`x==y, x!=y`	Gleichheit, Ungleichheit
	Operatoren auf ganzen Zahlen	
1	`~x`	Bitweises Komplement (NOT)
	Operatoren auf ganzen Zahlen und Wahrheitswerten	
7	`x & y`	Bitweises Und (AND)
8	`x ^ y`	Bitweises Entweder-Oder (XOR)
9	`x | y`	Bitweises Oder (OR)
	Operatoren auf Wahrheitswerten	
1	`!x`	NOT
10	`x && y`	Sequenzielles AND
11	`x || y`	Sequenzielles OR
	Operatoren auf ganzen Zahlen	
4	`x << y`	Linksshift
4	`x >> y`	Rechtsshift (vorzeichenkonform)
4	`x >>> y`	Rechtsshift (ohne Vorzeichen)

Strings

Der Datentyp `String` für Wörter ist kein Basistyp, sondern eine Klasse

Hier behandeln wir zunächst 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!";
System.out.println(s0 + s1 + s2 + s3);
```

... schreibt `Hello World!` auf die Ausgabe.

Implizite Konvertierung in Strings

- 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 !!!

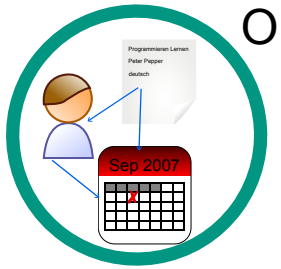
Beispiel:

```
double x = -0.55e13;
System.out.println("Eine␣Gleitkomma-Zahl:␣"+x);
```

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

Übersicht

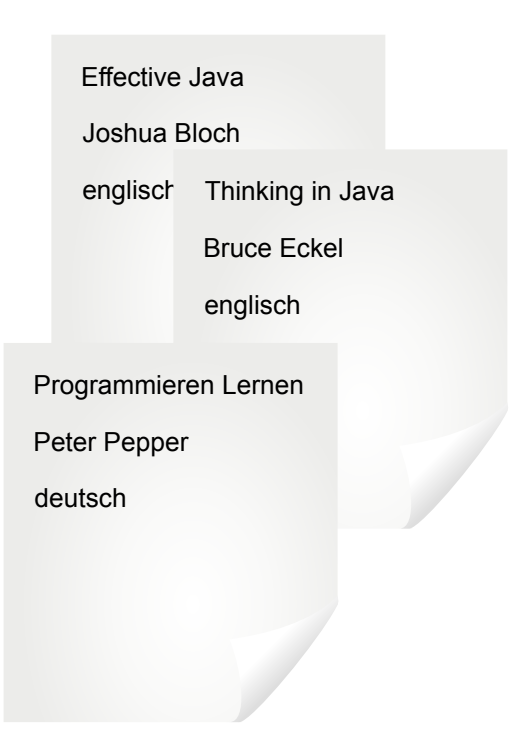
Suchmaschine

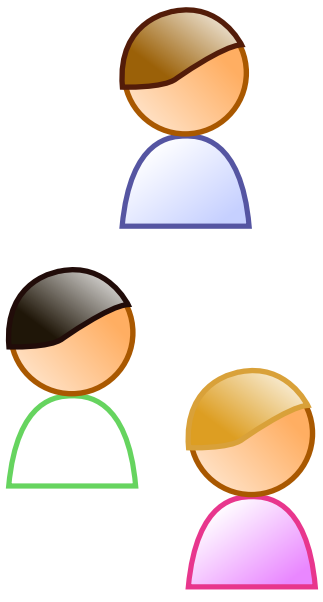


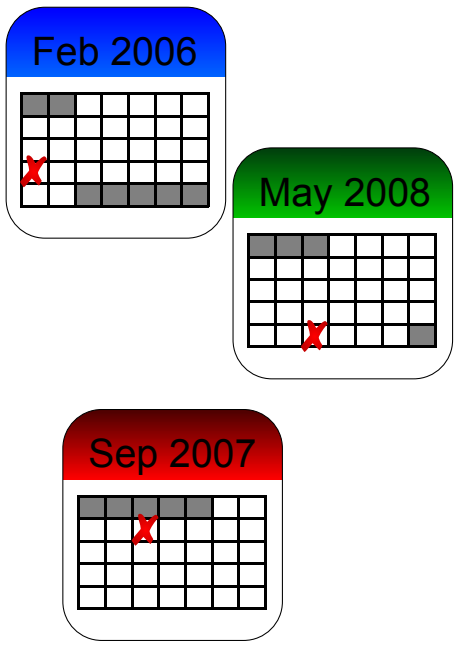
Objekte identifizieren

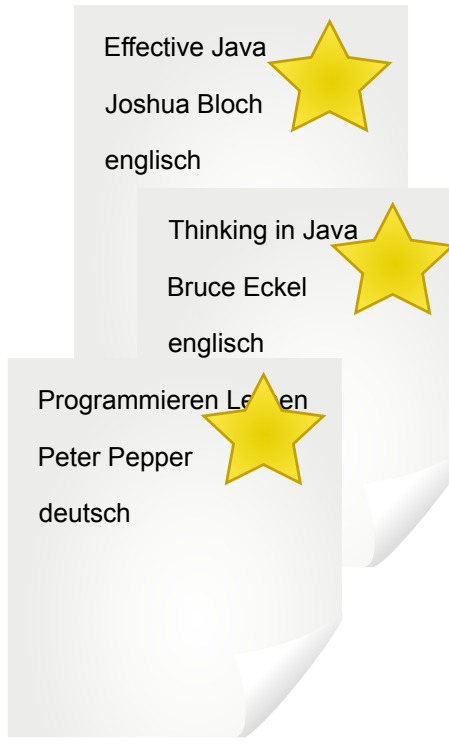
Suchmaschine: Objekte identifizieren

- Welche Objekte liegen im Problemfeld?





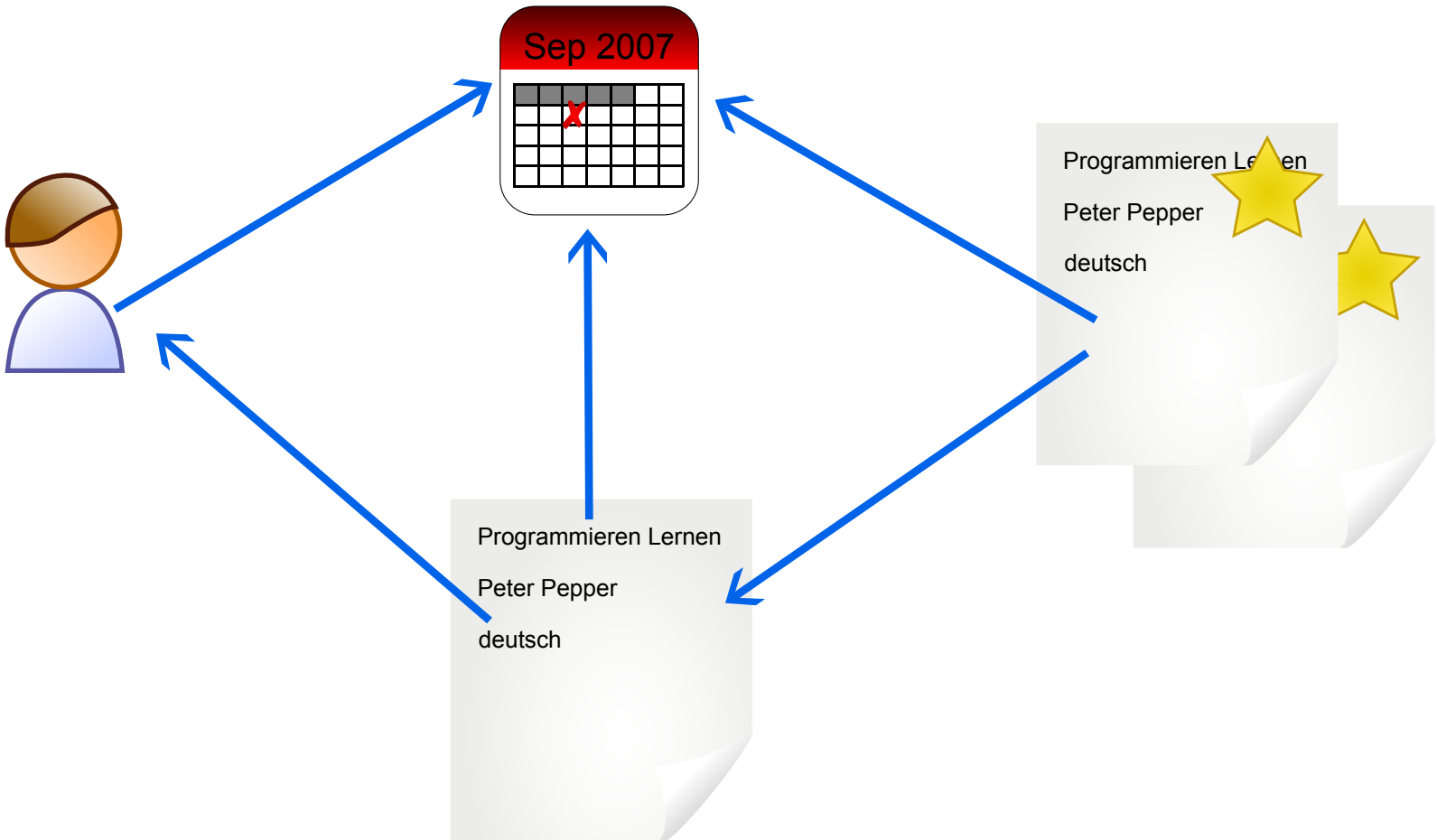




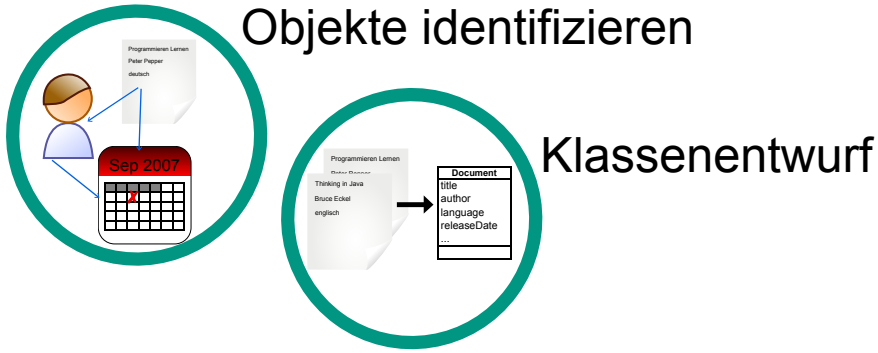
Dokumente Autoren Daten Rezensionen

Suchmaschine: Assoziationen identifizieren

- Welche Verhältnisse gibt es zwischen den Objekten?

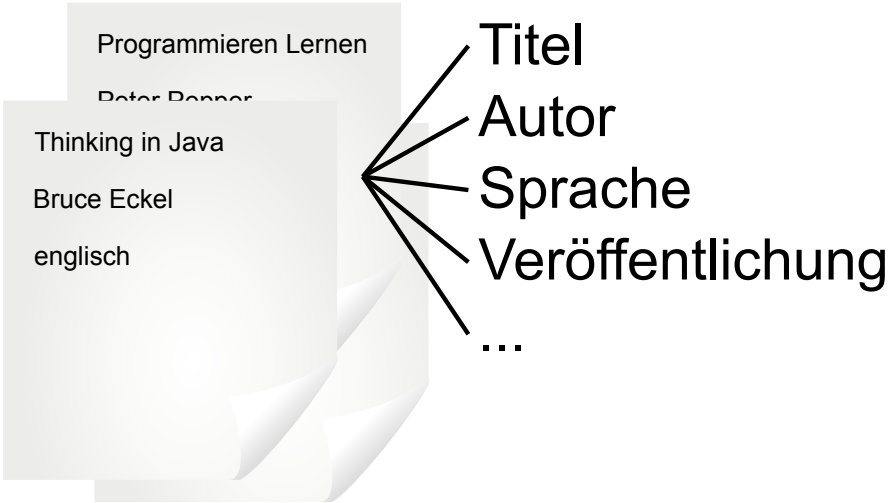


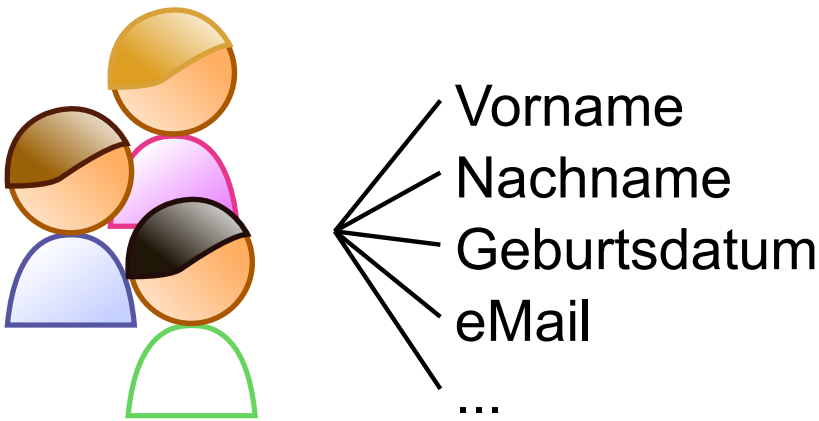
Suchmaschine



Suchmaschine: Klassenentwurf

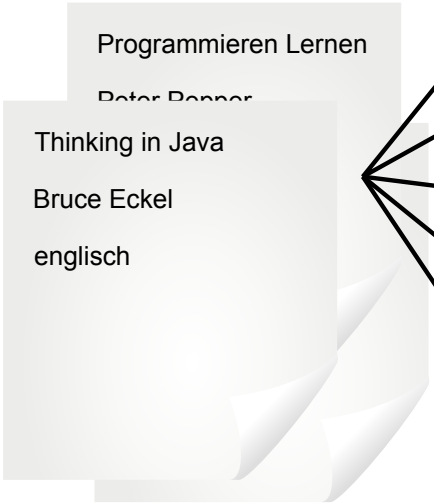
- Welche Attribute haben die Objekte?





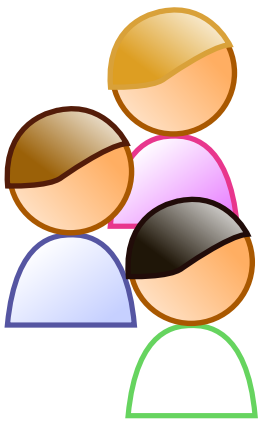
Suchmaschine: Klassenentwurf

- Entwerfen der identifizierten Klassen



Titel
Autor
Sprache
Veröffentlichung
...

Document
title author language releaseDate ...

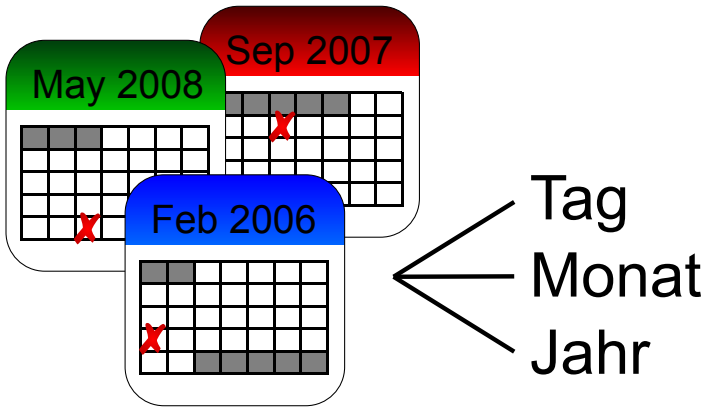


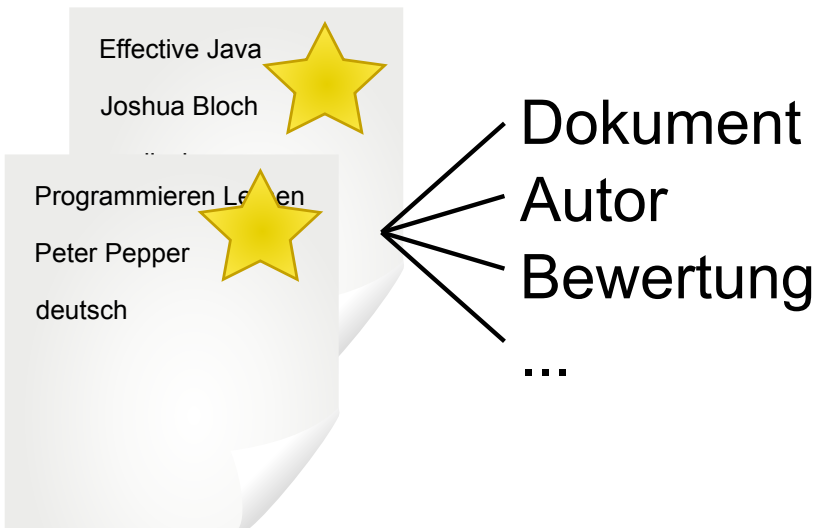
Vorname
Nachname
Geburtsdatum
eMail
...

Author
firstName lastName birthday eMail ...

Suchmaschine: Klassenentwurf

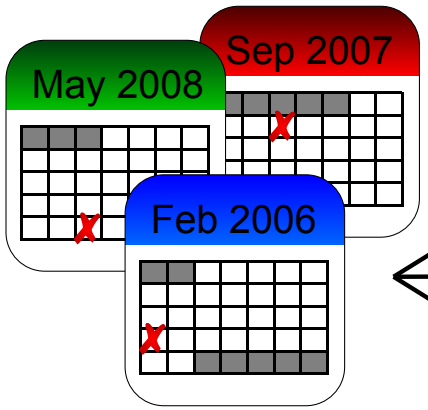
- Welche Attribute haben die Objekte?



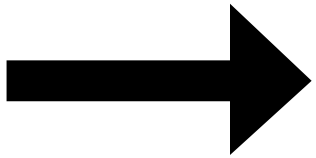


Suchmaschine: Klassenentwurf

- Entwerfen der identifizierten Klassen

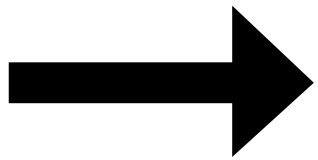


Tag
Monat
Jahr



Date
day month year

Dokument
Autor
Bewertung
...



Review
document author rating ...

Übersicht

Operationen und Funktionen

Diverse Operationen haben wir bereits verwendet:

- Zuweisung (=) wie in
  `int r=3;` oder in `Point p,r; ...; p=r;`
- Operationen auf Basistypen wie in
  `r=3*7;` oder in `r=p|2;`

Aus der Mathematik kennen Sie außerdem Funktionen, z.B.

- durchschnitt(x,y)=(x+y)/2
- fahrenheit(c)=c*9/5+32
- fac(0)=1 und n>0 $\Rightarrow$ fac(n)=n*fac(n-1)

Die Operationen auf Basistypen sind ebenfalls—infix notierte—Funktionen:

- +(a,b)=a *inkrementiert um* b
- bitwiseor(a,b)=a *bitweises oder* b

Formal- und Aktualparameter

Offenbar gibt es einen Unterschied zwischen der Definition und der Verwendung einer Funktion:

- In der Definition durchschnitt(x,y)=(x+y)/2 sind x und y Parameter.
  Parameter in Funktionsdefinitionen heißen Formalparameter.
  (Verwendung im Definitionsrumpf als Variablen!)
- In der Verwendung durchschnitt(7,3) mit Rückgabewert 5 sind die konkreten Eingabewerte 7 und 3 Aktualparameter.

Methoden

Methoden realisieren das *dynamische Verhalten* von Objekten—und definieren so Funktionen:

```
int durchschnitt(int v1, int v2) {
 return (v1+v2) / 2;
}

int fahrenheit(int celsius) {
  return (celsius * 9) / 5 + 32;
}

double spannung(double widerstand, double strom) {
  return widerstand*strom;
}
```

Methodensignatur

Eine **Methodensignatur** ist die Schnittstelle einer Methode

Sie besteht aus

- dem Namen der Methode
- der Anzahl der (Formal-)Parameter
- der Reihenfolge der (Formal-)Parameter
- dem Rückgabetypen der Methode

Beispiele:

```
int fahrenheit(int celsius) {
    //nicht Teil der Signatur:
    return (celsius * 9) / 5 + 32;
}

double fahrenheit(double celsius) {
    //nicht Teil der Signatur:
    return (celsius * 9) / 5 + 32;
}
```

Methodenaufruf

Beispiel: Ohmsches Gesetz

```
double u(double r, double i) {
  return r*i;
}
```

Aufruf:

```
double r1 = 3.0;
double u1 = u(r1,2.0);
```

Syntax: *Methoden-Name*(*Aktualparameter*)

Ergebnis: Ein Element mit dem in der Methodensignatur angegebenen Typ.

$\Longrightarrow$ Welchen Wert hat `u1`?

Beispiel

Mathematische Notation:

- $volumen(r, h) = r^2 \cdot \pi \cdot h$

Java-Deklaration:

- ```
  double kreisFlaeche(double radius) {
    return radius * radius * 3.1416;
  }
  ```

- ```
  double zylinderVolumen(double hoehe, double radius) {
    return hoehe * kreisFlaeche(radius);
  }
  ```

- „`kreisFlaeche(2.0)`“ liefert den Wert `12.5664`
- „`zylinderVolumen(1.5, 2.0)`“
  - wertet den Ausdruck „`1.5 * kreisFlaeche(2.0)`“ aus und liefert daher den Wert `18.8496`

Nochmals Konstruktoren

- Konstruktoren sind spezielle Methoden, die nach Speicherreservierung mit `new` Attribute initialisieren.
- Konstruktoren haben keinen Rückgabetypen (aber sie geben nichts zurück, *konzeptionell* ist der Typ also `void`, kommt gleich).
- Die Formalparameter können für die Initialisierung verwendet werden.

Beispiel:

```
class Rational{
  double zaehler, nenner;
  Rational(double z,double n) {
   zaehler = z;
   nenner = n;
  }
}
```

Der Rückgabetyp void

Der Rückgabetyp **void** wird verwendet, wenn eine Methode keinen Wert zurückliefern soll. Die Methode heißt dann Prozedur.

Beispiel:

```
void alarm(String message) {
  write("GEFAHR: " + message);
}
```

- kein **return** nötig
- zustandsändernde Methode
- Deklaration:
  void *Name* **(** *Formalparameterliste* **) { ***Rumpf* **}**
- Verwendung:
  Name **(** *Aktualparameterliste* **) ;**

Bezugsobjekt

Normalerweise ist die Methode abhängig vom Zustand des aktuellen Objekts
$\Longrightarrow$ Beim Aufruf muss das *Bezugsobjekt* angegeben sein.
Das bedeutet, dass man innerhalb einer Methode dieses Bezugsobjekt („sich selbst") adressieren können muss!

Dazu gibt es das Schlüsselwort `this`.

Beispiel this – im Konstruktor

```
class Point {
  double x, y;
  Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

Nutzung des Objektzustands in Methoden

Beispiel: Definiere Methode `shift` in der Klasse `Point`

```
class Point {
  double x, y;
  void shift(double dx, double dy) {
    this.x = this.x + dx;
    this.y = this.y + dy;
  }
}
```

Aufruf:

```
Point p = new Point(3.0, 4.0);
p.shift(2.0, 4.0);
```

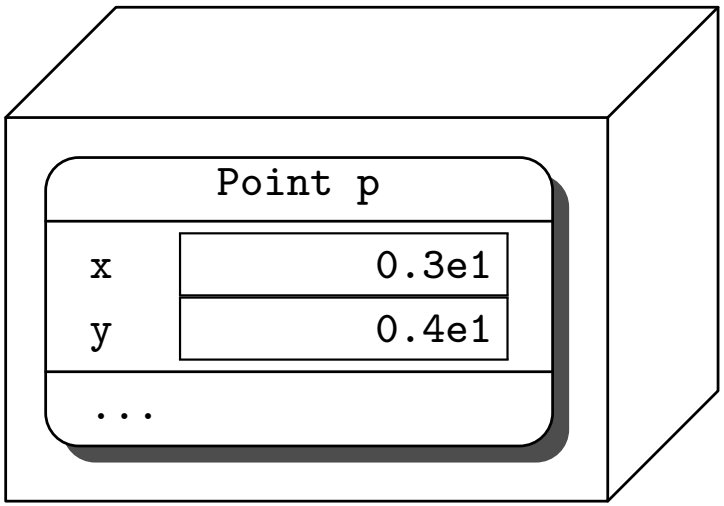
Aufruf-Syntax:

Bezugsobjekt.*Methoden-Name*(*Parameter*);

Ist kein Bezugsobjekt angegeben $\Longrightarrow$ immer implizit **this**.

Effekt von shift

```
Point p = new Point(3.0, 4.0);
```

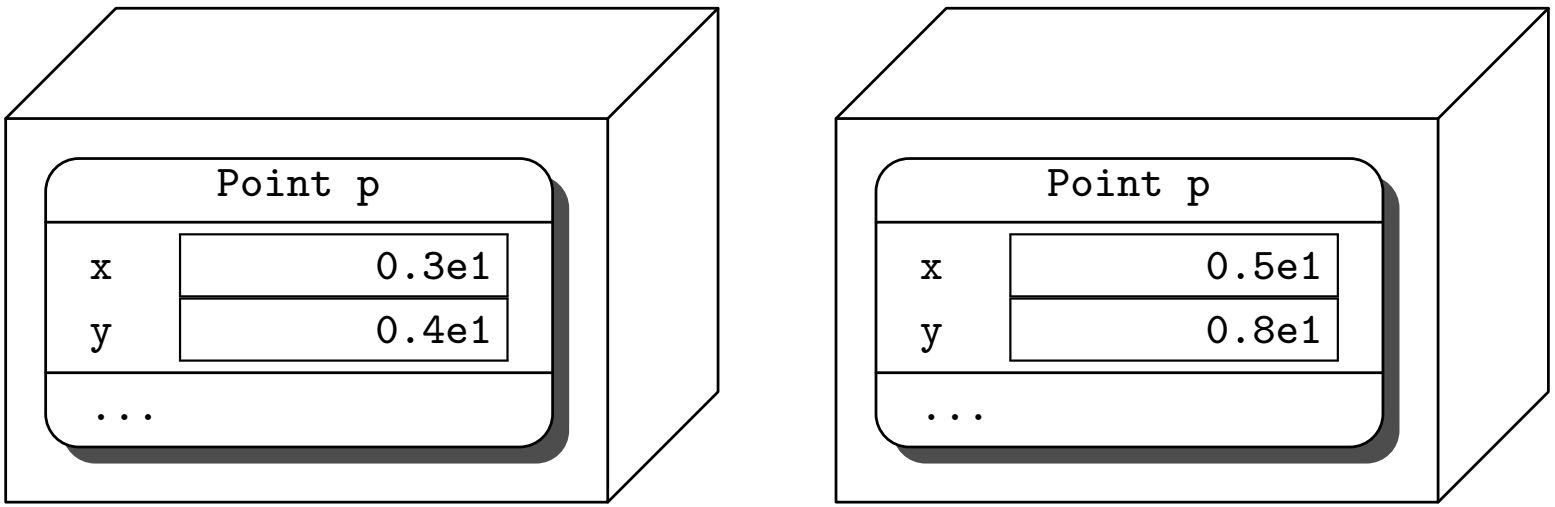


(a) Vor `p.shift(2.0, 4.0)`

- `p` wird beim Aufruf von `shift` an **`this`** zugewiesen
  $\Longrightarrow$ **`this`** ermöglicht Aufruf derselben Methode auf verschiedenen Objekten (Abstraktionsprinzip)

Effekt von shift

```
Point p = new Point(3.0, 4.0);
p.shift(2.0, 4.0);
```



(a) Vor `p.shift(2.0, 4.0)` (b) Nach `p.shift(2.0, 4.0)`

- `p` wird beim Aufruf von `shift` an **`this`** zugewiesen
  $\Longrightarrow$ **`this`** ermöglicht Aufruf derselben Methode auf verschiedenen Objekten (Abstraktionsprinzip)

Selbst-Referenzen in Konstruktoren: Zyklizität

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

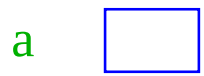
Selbst-Referenzen in Konstruktoren: Zyklizität

```
class Cyclic {
    int info;
    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.

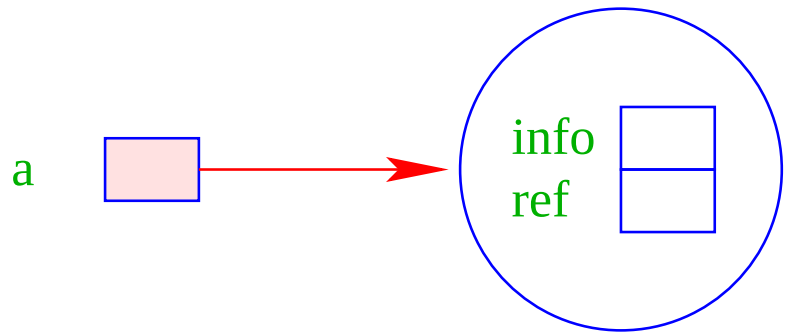
Selbst-Referenzen: Zyklizität

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



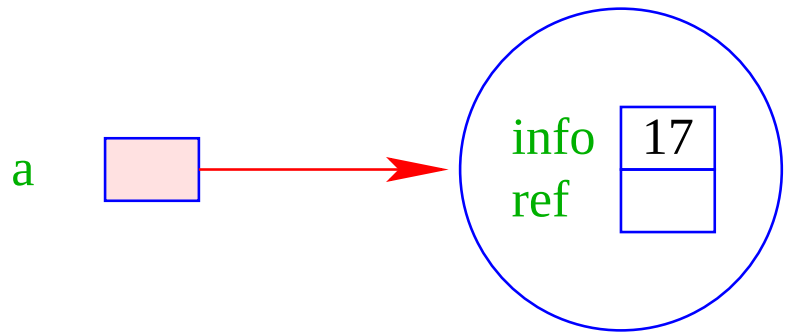
Selbst-Referenzen: Zyklizität

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



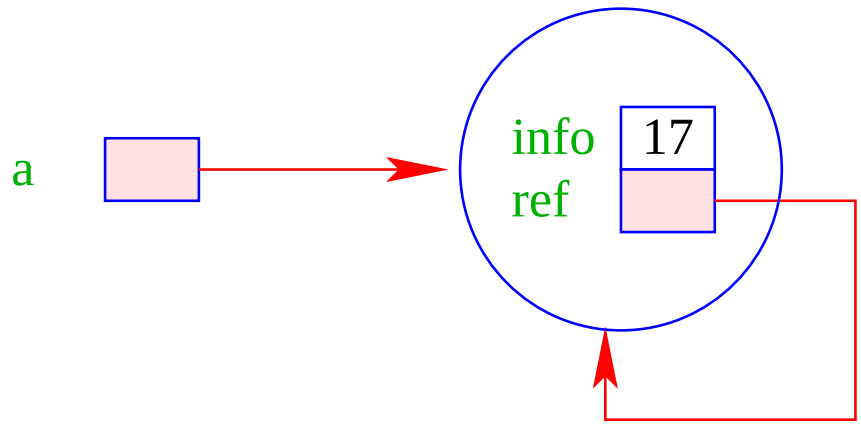
Selbst-Referenzen: Zyklizität

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



Selbst-Referenzen: Zyklizität

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



Bezugsklasse – statische Methoden

Eine *statische* Methode ist **unabhängig** vom Zustand konkreter Objekte.

Beispiel:

```
class Weather {
  static int fahrenheit(int celsius) {
    return (celsius * 9) / 5 + 32;
  }
}

... Weather.fahrenheit(15); ...
```

Syntax:

```
static Rückgabetyp
Methoden-Name(Parameter) {
    // Methodenrumpf
}
```

Aufruf-Syntax:

```
Klassen-Name.Methoden-Name(Parameter);
```

Die main-Methode

Eine statische Methode ist besonders: Die `main()`-Methode.

```
class K {
  public static void main(String[] argv) {
    ...
  }
}
```

`main(p1,...,pn)` in Klasse `K` wird aufgerufen, wenn Sie nach Compilieren von `K` `java K` ausführen. Die Argumente `p1,...,pn` werden in einem sogenannten Feld namens `argv` abgelegt. Felder behandeln wir später!

Übersicht

Aufrufen von Methoden

mit Bezugs**klasse**:

Beispiel: `Weather.fahrenheit(15);`

- Nur statische Methoden (Schlüsselwort **`static`**)
- Diese sind unabhängig vom Zustand konkreter Objekte

mit Bezugs**objekt**:

Beispiel: `p.shift(2.0, 4.0);`

- Ohne expliziten Bezug wird immer **`this`** als Bezugsobjekt angenommen

Zur Erinnerung: Klasse Point

Verschieben von Punkten:

```
void shift(double dx, double dy) {
  this.x = this.x + dx;
  this.y = this.y + dy;
}
```

Aktuelle Koordinaten:

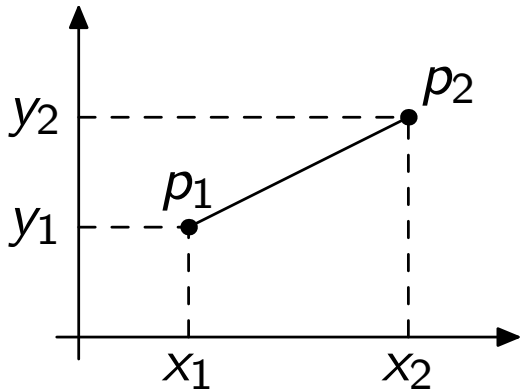
```
double getX() {
  return this.x;
}
```

```
double getY() {
  return this.y;
}
```

Linien verschieben

Beispiel: Linien im $\mathbb{R}^2$

Konzept:

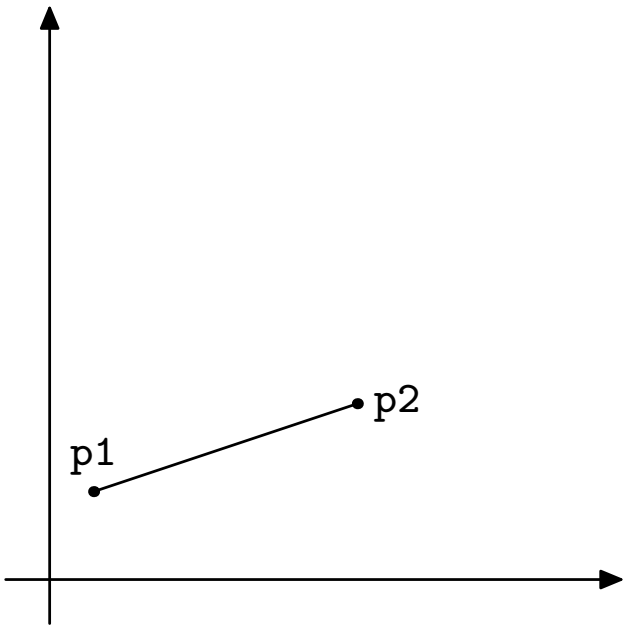


JAVA-Code:

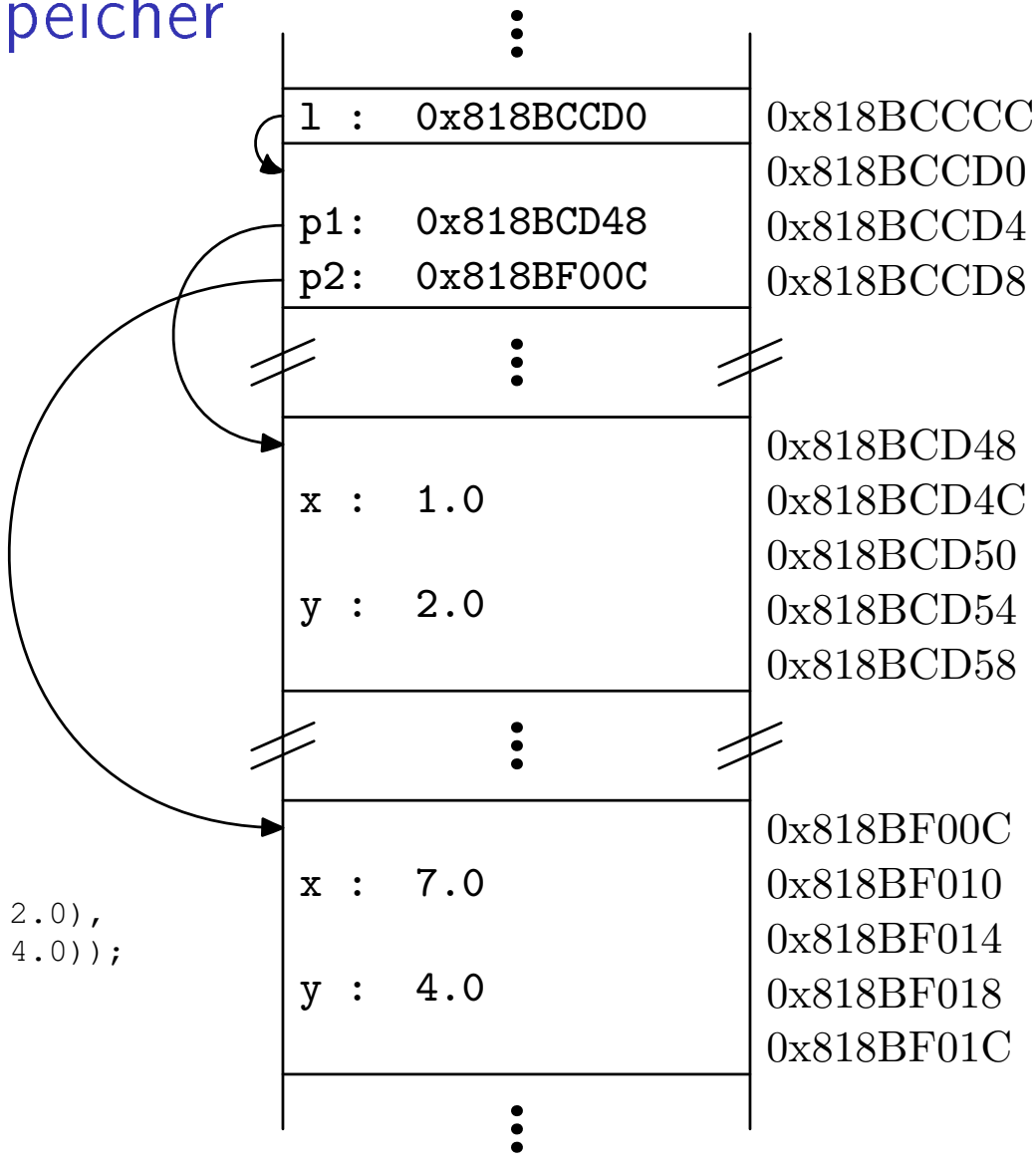
```
class Line {

  // Attribute: Endpunkte
  Point p1;
  Point p2;

  // Methoden
  Line(Point p1, Point p2) {
    this.p1 = p1;
    this.p2 = p2;
  }

  void shift(double dx, double dy) {
    this.p1.shift(dx, dy);
    this.p2.shift(dx, dy);
  }
}
```

Effekt von `shift` im Speicher

Verschieben einer Linie

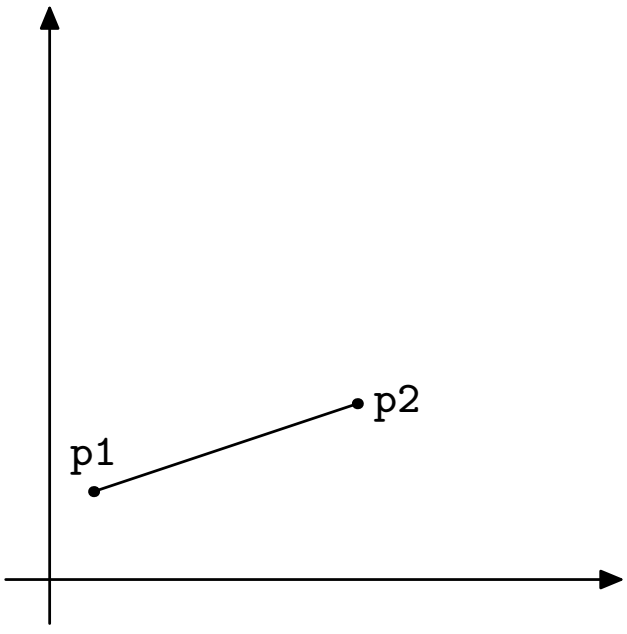



```
Line l = new Line(new Point(1.0, 2.0),
                  new Point(7.0, 4.0));
```

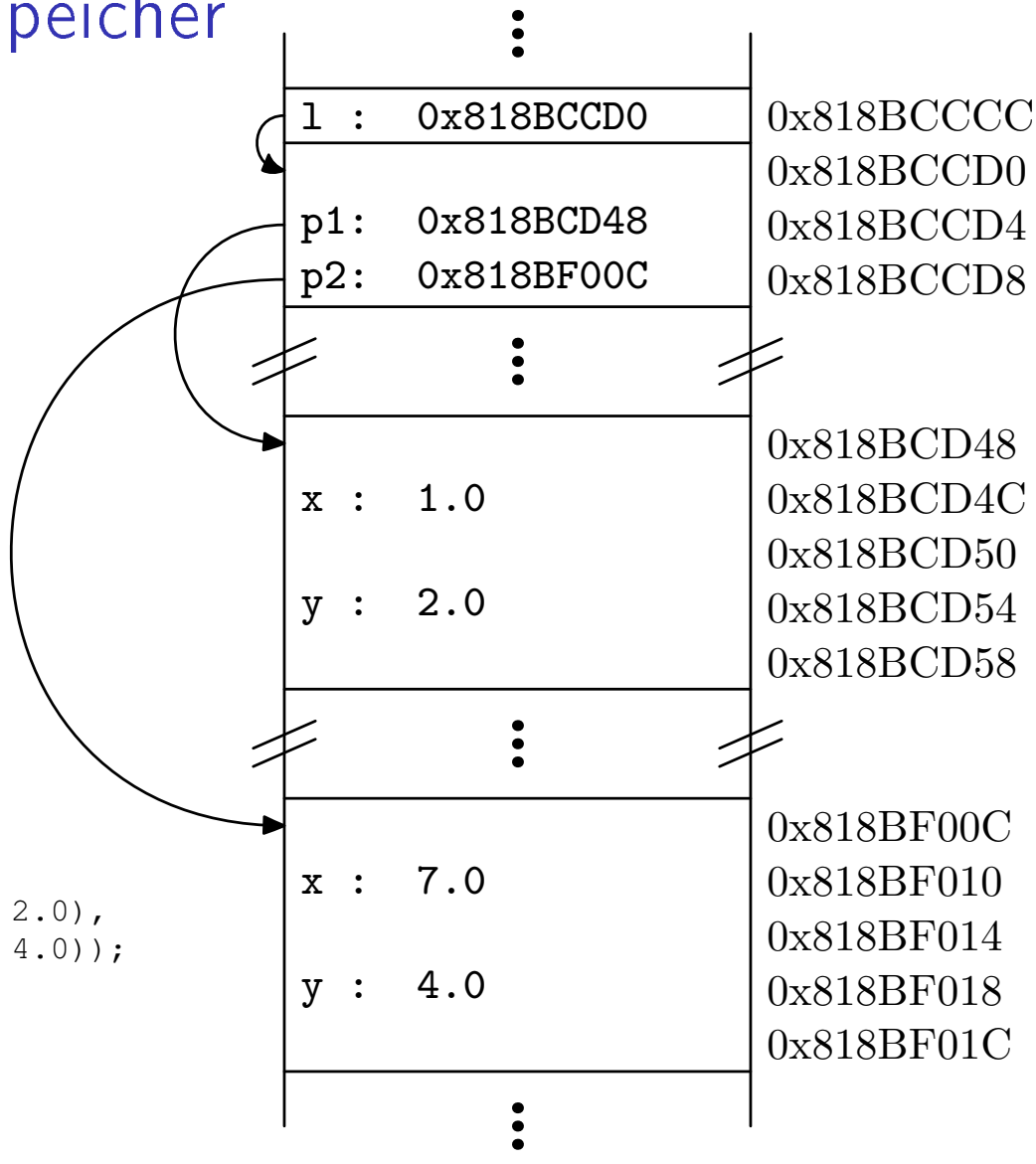


Effekt von `shift` im Speicher

Verschieben einer Linie



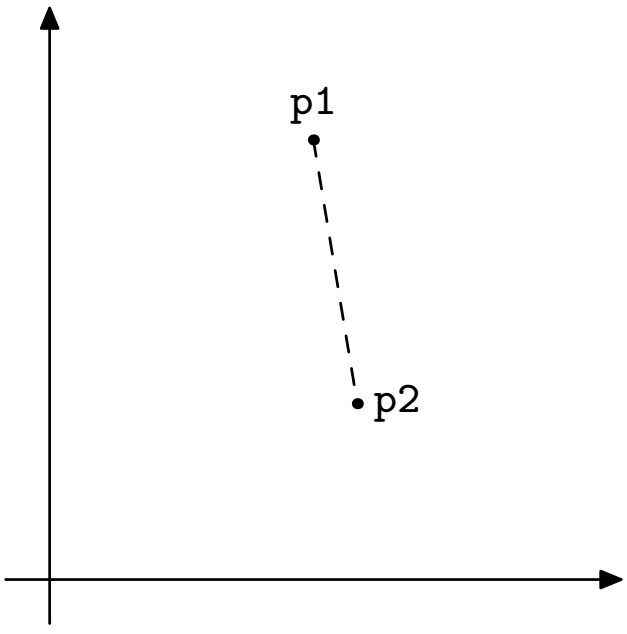

```
Line l = new Line(new Point(1.0, 2.0),
                  new Point(7.0, 4.0));
l.shift(5.0, 8.0);
```



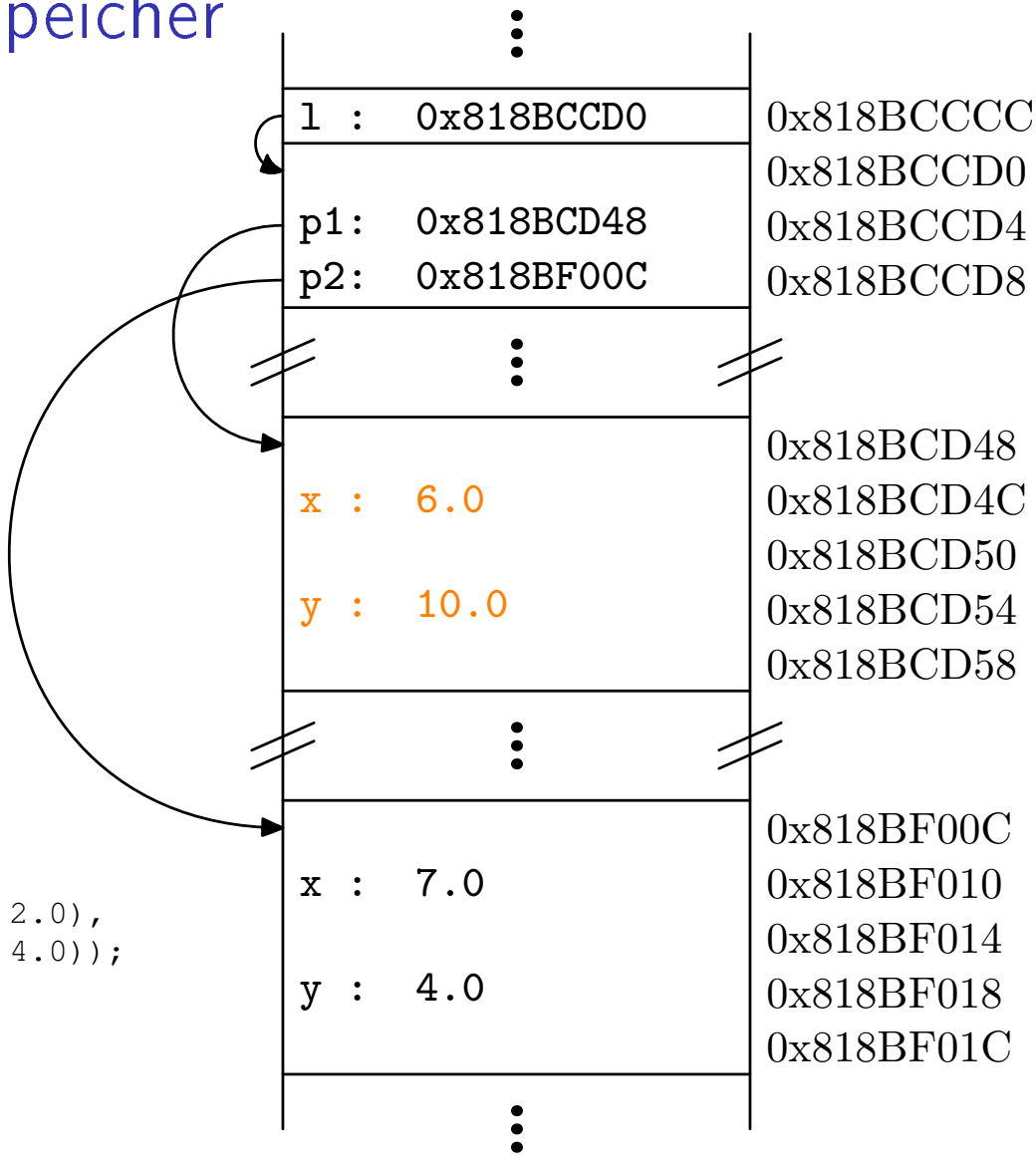
this = l, dx = 5.0, dy = 8.0
this.p1.shift(dx, dy); **this**.p2.shift(dx, dy);

Effekt von `shift` im Speicher

Verschieben einer Linie



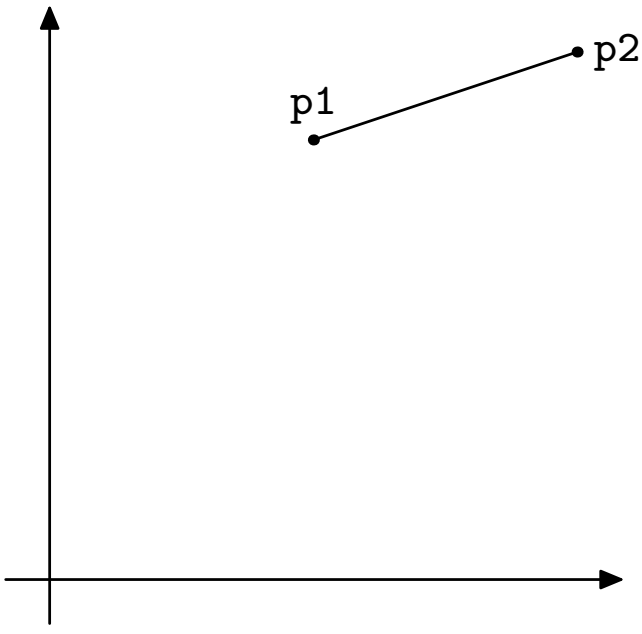

```
Line l = new Line(new Point(1.0, 2.0),
                  new Point(7.0, 4.0));
l.shift(5.0, 8.0);
```




```
this = l, dx = 5.0, dy = 8.0
this.p1.shift(dx, dy);
this.p2.shift(dx, dy);
```

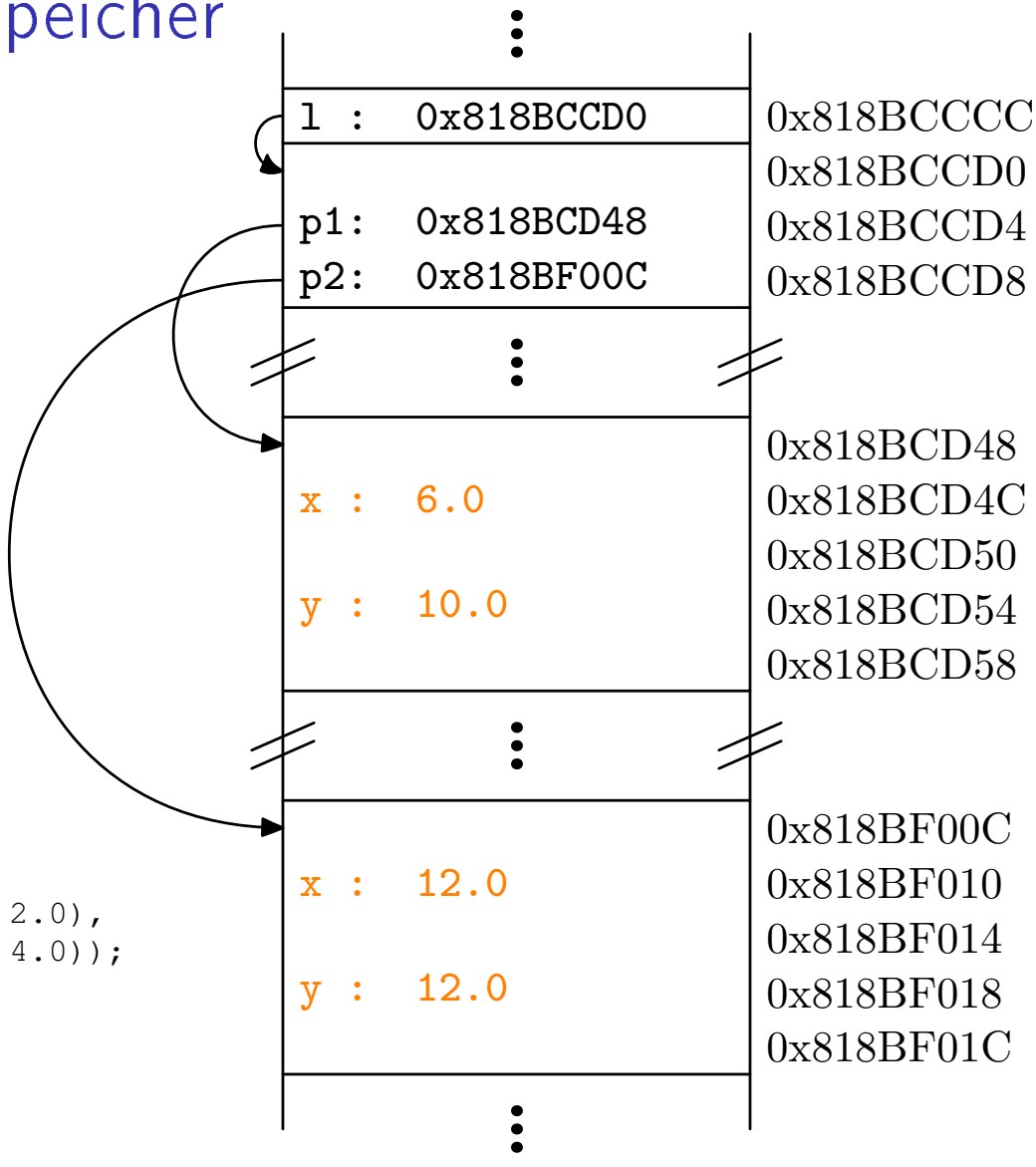
Effekt von `shift` im Speicher

Verschieben einer Linie




```
Line l = new Line(new Point(1.0, 2.0),
                  new Point(7.0, 4.0));
l.shift(5.0, 8.0);
```

this = l, dx = 5.0, dy = 8.0
this.p1.shift(dx, dy); **this**.p2.shift(dx, dy);



Was passiert beim Aufruf einer Methode?

```
Point p
  = new Point(3.0, 4.0);
p.shift(-1.0, 26.0);
```

```
// in der Klasse Point:
void shift(double dx, double dy) {
  this.x = this.x + dx;
  this.y = this.y + dy;
}
```

1. Das Bezugsobjekt und dessen Typ wird ermittelt:
   Bezugsobjekt ist `p` und vom Typ `Point`
2. Die aufzurufende Methode aus der in 1 ermittelten Klasse wird bestimmt:
   Verwende Methode `shift` aus der Klasse `Point`
3. Die Referenz des ermittelten Objekts wird an den **`this`**-Pointer der aufgerufenen Methode zugewiesen:
   `this` `= p`
4. Die aktuellen Parameter werden den formalen Parametern der Methode zugewiesen:
   `dx = -1.0` und `dy = 26.0`
5. Der Rumpf der Methode wird abgearbeitet

Referenzen und Methodenaufrufe

In JAVA wird bei der Parameterübergabe bei Methodenaufrufen ausschließlich Call-by-value verwendet

Definition Call-by-value

Beim Aufruf von Methoden werden alle Parameter zuerst komplett ausgewertet und dann die **Werte** als Kopie an die aufgerufene Methode übergeben.

Beachte: Werte von nicht-primitiven Typen sind Referenzen!

$\Longrightarrow$ Es wird eine Kopie der Referenz übergeben. Damit lässt sich noch immer *der Zustand* des referenzierten Objekts verändern

Referenzen und Methodenaufrufe – Beispiel

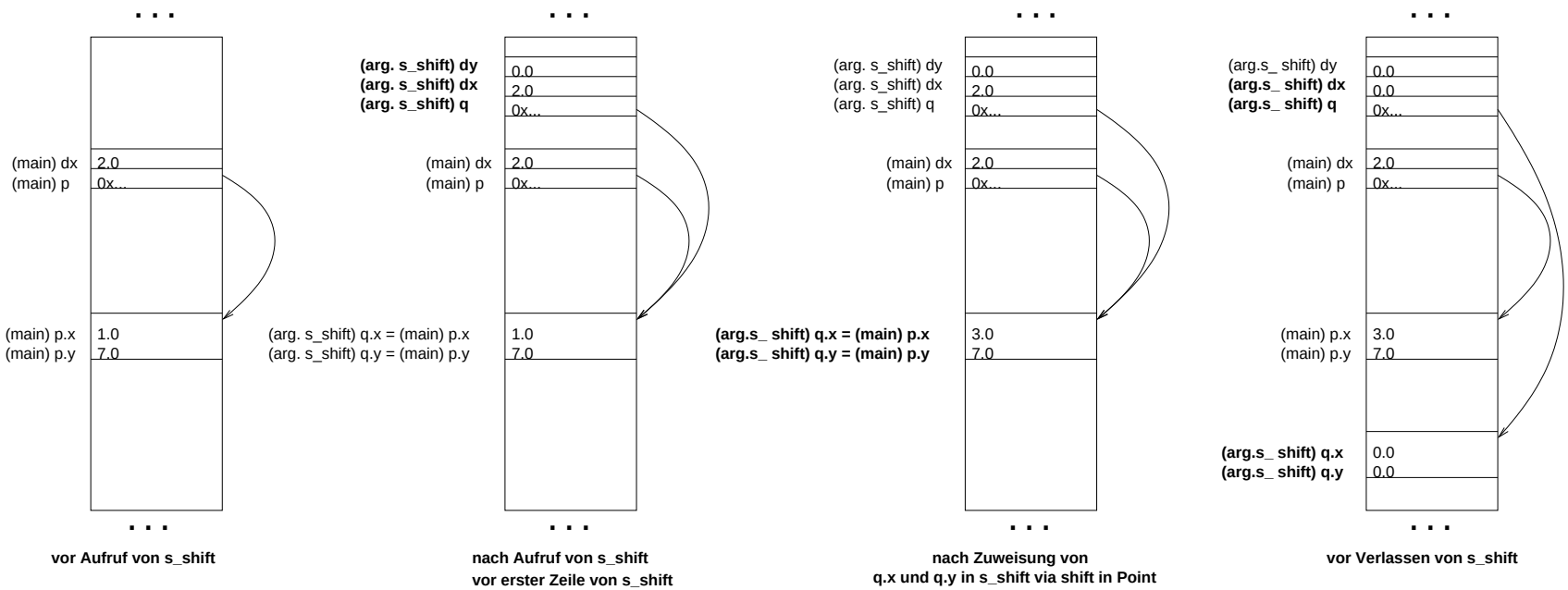
- Betrachte das Programm:

```
public static void s_shift(Point q, double dx, double dy) {
   q.shift(dx,dy);
   q = new Point();  // schlecht und unnoetig
   dx = 0;           // schlecht und unnoetig
}

static void main(String[] args) {
   Point p = new Point(1.0, 7.0);
   double dx = 2.0;
   s_shift(p, dx, 0.0);
   System.out.println("p.x = " + p.x);
   System.out.println("p.y = " + p.y);
   System.out.println("dx = " + dx);
}
```

- Ausgabe:

```
p.x = 3.0
p.y = 7.0
dx = 2.0
```

Referenzen und Methodenaufrufe – Beispiel 2




```
public static void s_shift(Point q, double dx, double dy) {
   q.shift(dx,dy);
   q = new Point();                    // schlecht und unnoetig
   dx = 0;                             // schlecht und unnoetig
}

public static void main(String[] args) {
 // Parameter args wird nicht in der Grafik gezeigt!
   Point p = new Point(1.0, 7.0);
   double dx = 2.0;                    // lokale Variable, kommt gleich
   s_shift(p, dx, 0.0);
}
```

Und ein viertes Mal: Gleichheit von Objekten

In JAVA gibt es zwei Möglichkeiten auf Gleichheit zu prüfen:

1. Der „==“-Operator prüft auf Wert-Gleichheit
   - Bei Objekten heißt das, Gleichheit der Referenz (also ob auf dasselbe Objekt verwiesen wird)
2. Die Methode `equals` prüft auf (inhaltliche) Objekt-Gleichheit
   - `equals` sollte für eigene Klassen überschrieben werden

Beispiel:

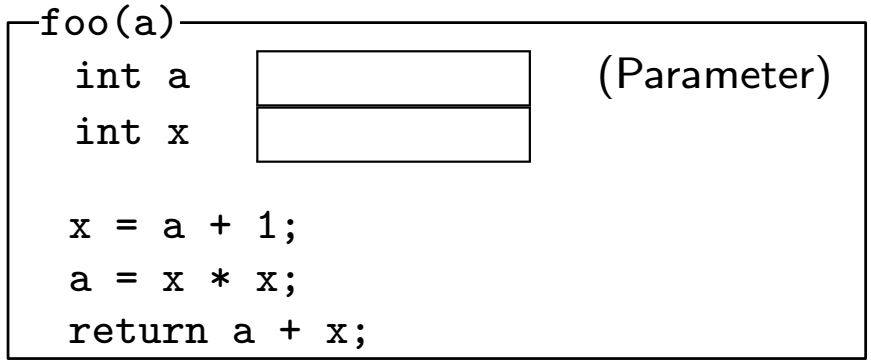
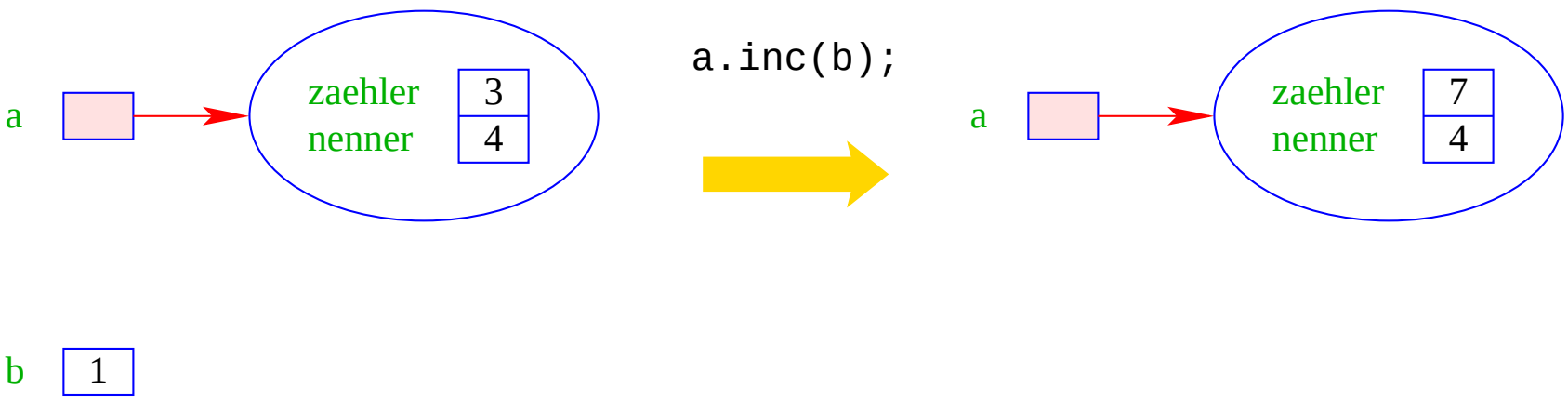
```
public class Rational {
   int zaehler, nenner;
   ...
   public boolean equals(Rational r) {
      return (zaehler * r.nenner == r.zaehler * nenner);
   }
}
```

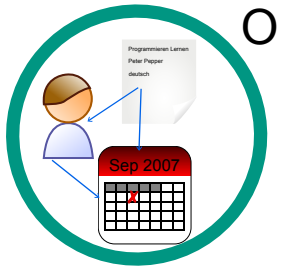
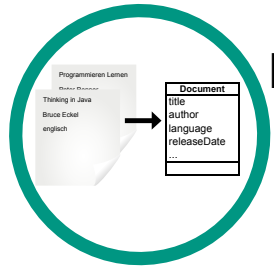
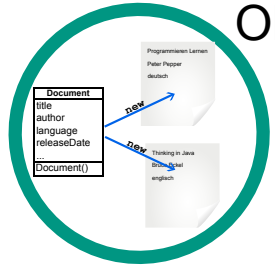
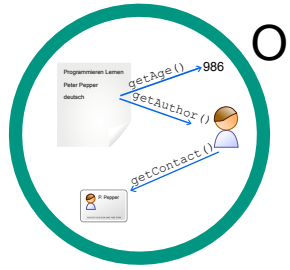
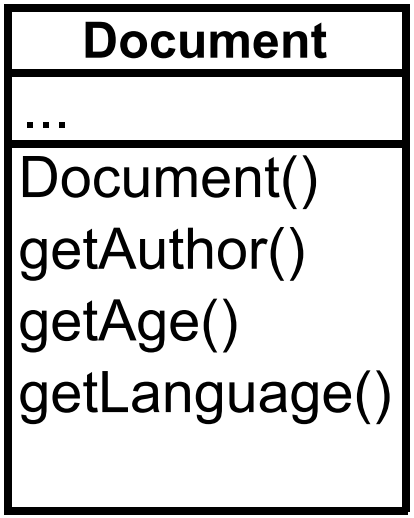
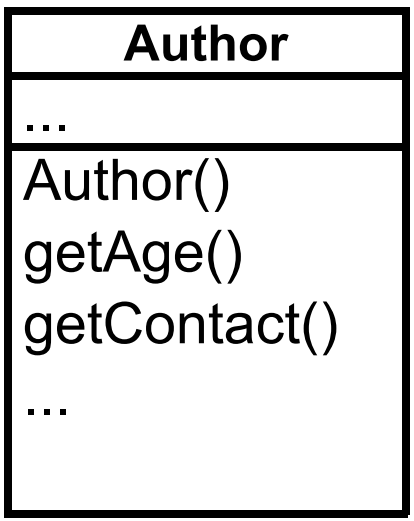
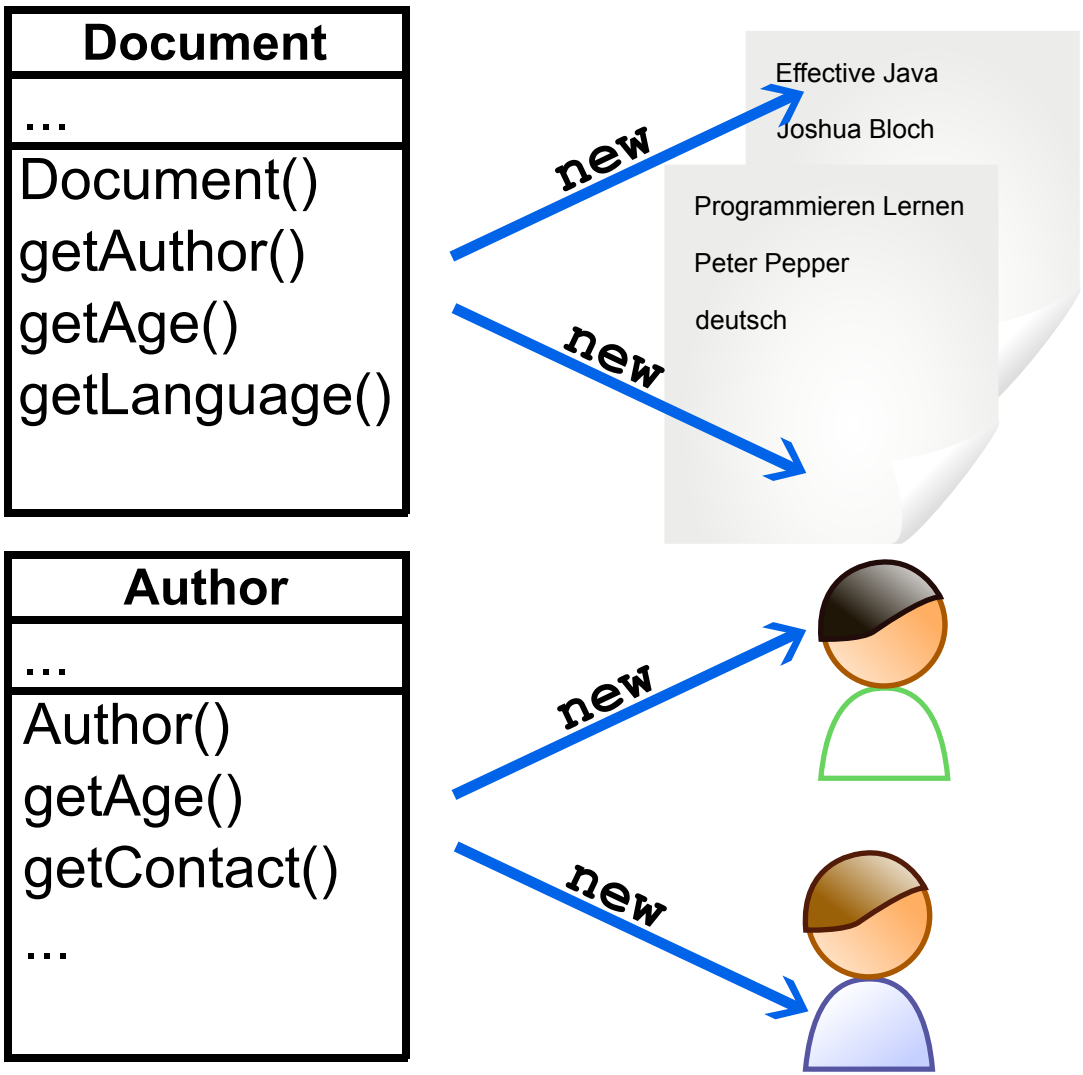
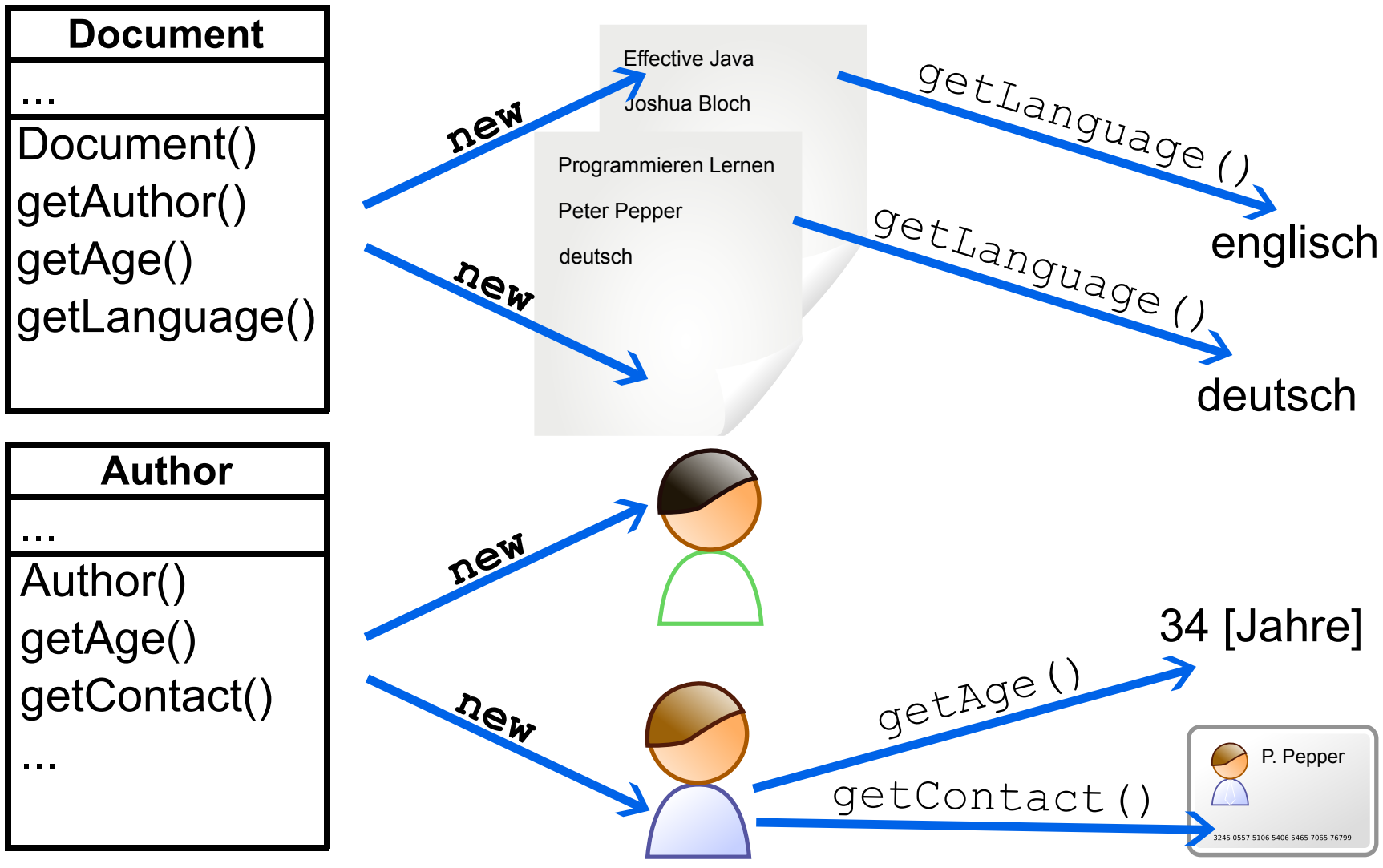
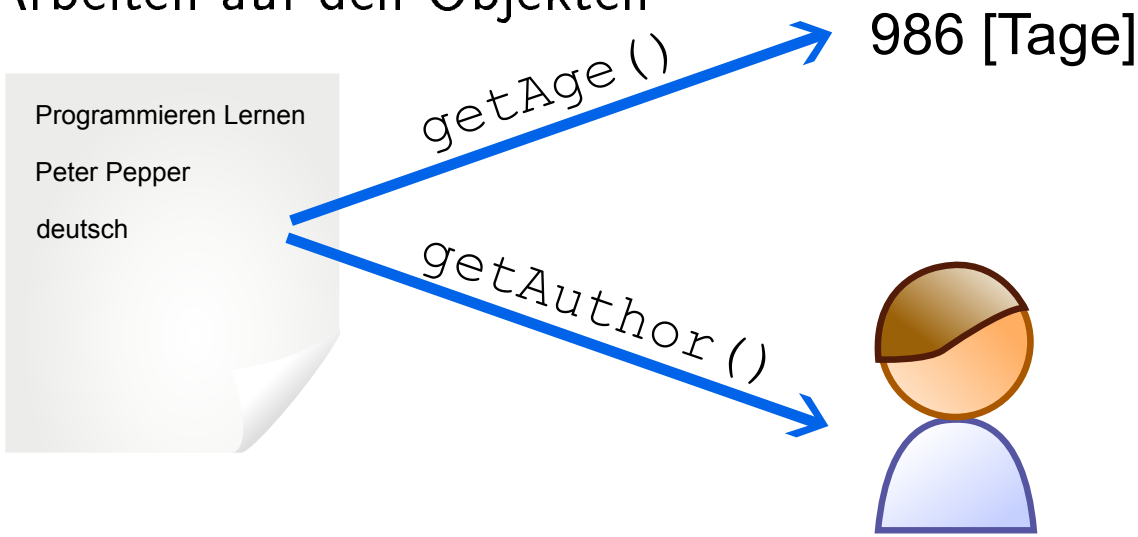
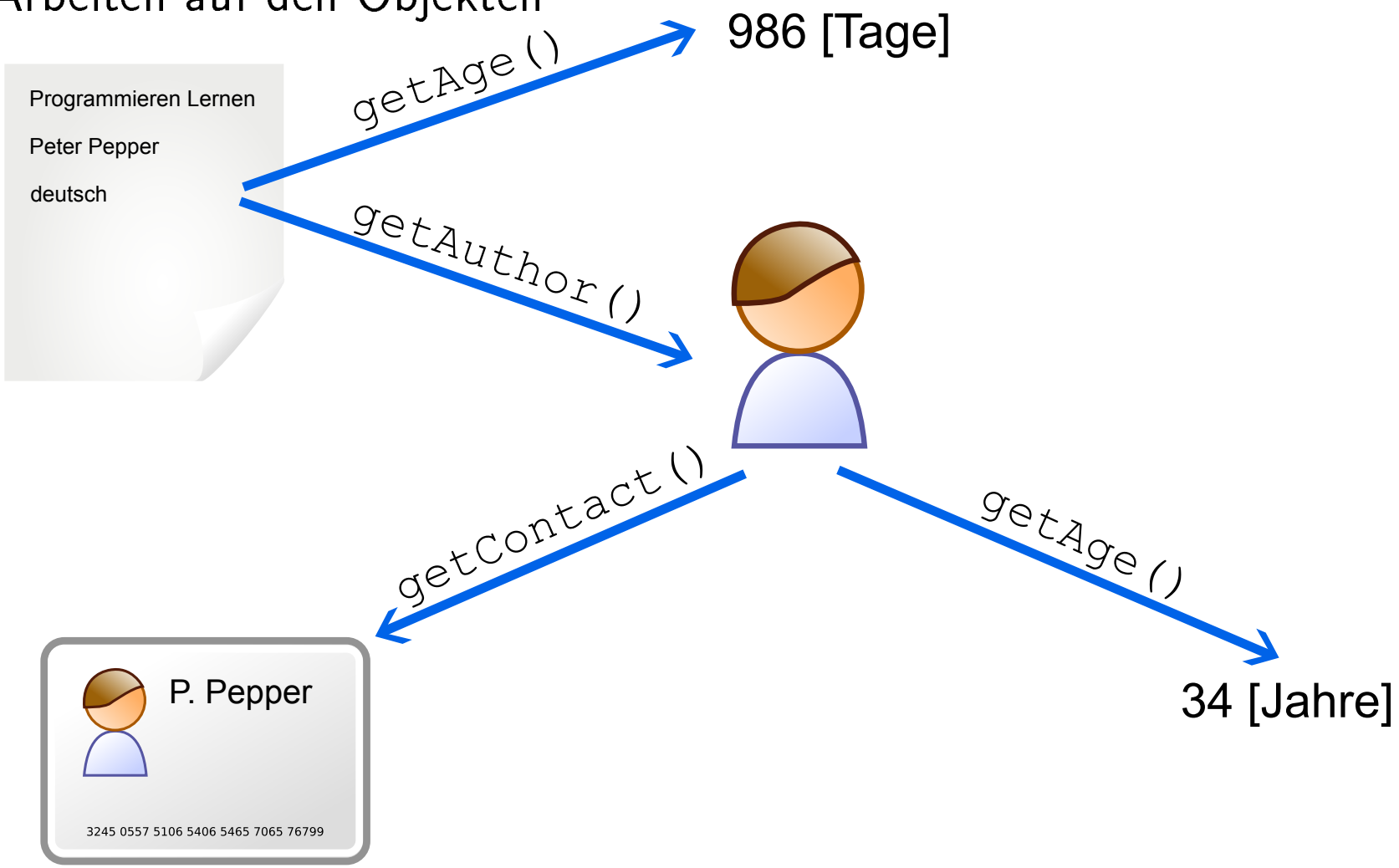
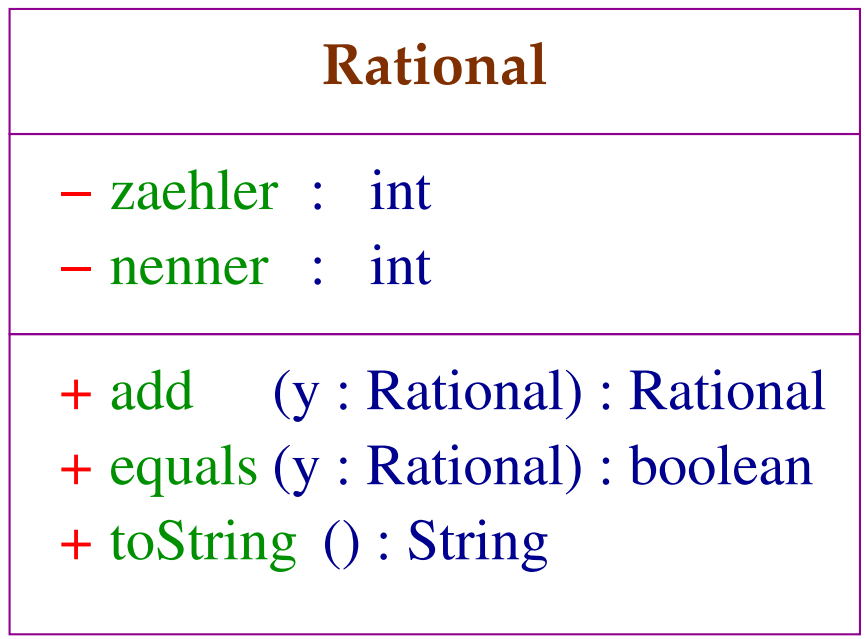
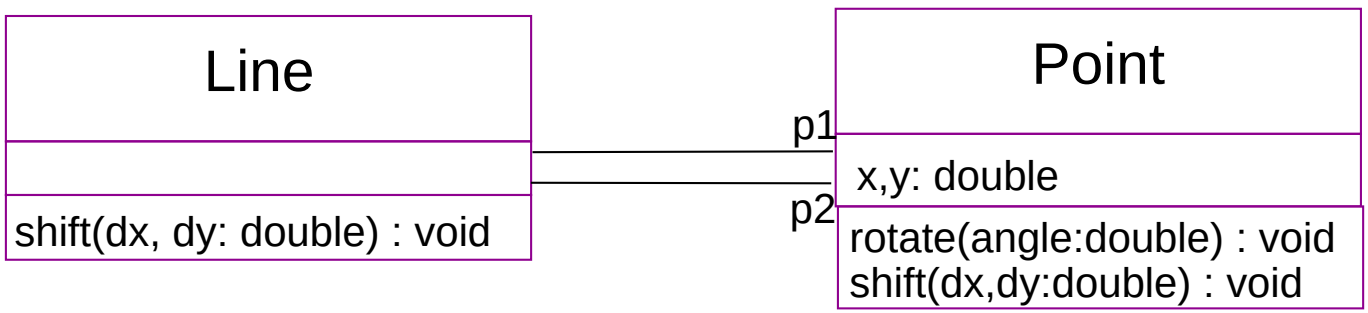
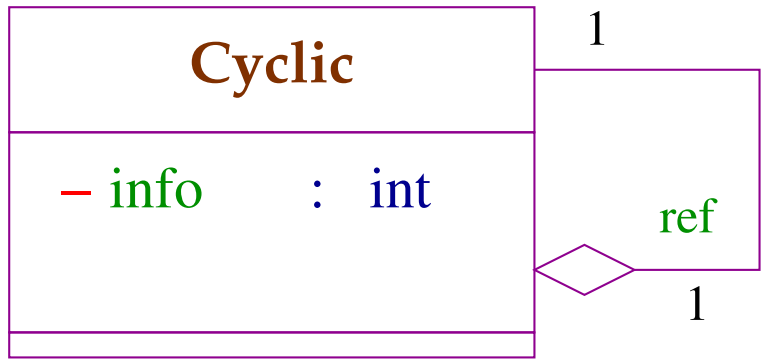
Lokale Variablen (I)

Lokale Variablen

- Dienen als Hilfsspeicher für Zwischenergebnisse in Methoden
- Werden auf dem sogenannten „Laufzeit-Stack“ gespeichert
- Existieren nur solange, wie die Methode abgearbeitet wird

Beispiel: (Berechnung der Kreisfläche)

- ```
  double kreisFlaeche(double radius) {
    double radiusQuadrat = radius * radius;
    return radiusQuadrat * 3.1416;
  }
  ```
  ```

Lokale Variablen (II)

Parameter sind in JAVA lokale Variablen, die beim Methoden-Aufruf initialisiert werden (call-by-value).

Beispiel:

```
int foo(int a) {
  int x = a + 1;
  a = x * x;          // Schlechter Programmierstil
  return a + x;
}
```



Achtung: Einfache Argumenttypen (`int`, `float`, ...) werden anders behandelt als komplexe Typen (`Buch`, `Author`, ...). Behandeln wir später.

Lokale Variablen (III)

Unterschied zu Attributen

	Lokale Variable	**Attribut**
Deklaration	innerhalb von Methoden	außerhalb von Methoden
Lebensdauer	Methoden-Aufruf	Lebensdauer des zugehörigen Objektes
Zugänglich	nur innerhalb einer Methode	für alle Methoden der Klasse
Zweck	Zwischenspeicher für Werte	Zustand des Objektes

Überladen von Methoden (I)

Es kann verschiedene Methoden mit *demselben Namen* geben, wenn sich diese in den *Typen* und/oder der *Anzahl* ihrer *Parameter unterscheiden*.

Beispiel für Klasse `Point`:

- ```
  void rotate(double angle) {
    double phi = Math.toRadians(angle);
    double xOld = this.x;
    double yOld = this.y;
    this.x = xOld * Math.cos(phi) - yOld * Math.sin(phi);
    this.y = xOld * Math.sin(phi) + yOld * Math.cos(phi);
  }
  ```

- ```
  void rotate(Point center, double angle) {
    double xOfCenter = center.x;
    double yOfCenter = center.y;
    this.shift(-xOfCenter, -yOfCenter);
    this.rotate(angle);
    this.shift(xOfCenter, yOfCenter);
  }
  ```

Überladen von Methoden (II)

Beispiel für Klasse `Line`:

- ```
  void rotate(double angle) {
    this.p1.rotate(angle);
    this.p1.rotate(angle);
  }
  ```

- ```
  void rotate(Point center, double angle) {
    this.p1.rotate(center, angle);
    this.p2.rotate(center, angle);
  }
  ```

- ... und wir haben schon gesehen, wie Konstruktoren überladen werden
- (Achtung: Default-Konstruktoren existieren nur, wenn kein Konstruktor explizit definiert wird!)

Private Hilfsmethoden

Beispiel: Berechne die Länge einer Linie (Klasse `Line`)

- Mathematische Lösung:
  $\sqrt{(\texttt{p2.x} - \texttt{p1.x})^2 + (\texttt{p2.y} - \texttt{p1.y})^2}$
- Methode in JAVA:

```
double length() {
  return Math.sqrt((p2.x-p1.x) * (p2.x-p1.x)
                   + (p2.y-p1.y) * (p2.y-p1.y));
}
```

Private Hilfsmethoden

Beispiel: Berechne die Länge einer Linie (Klasse `Line`)

- Mathematische Lösung:
  $\sqrt{(\texttt{p2.x} - \texttt{p1.x})^2 + (\texttt{p2.y} - \texttt{p1.y})^2}$
- Methode in JAVA (Bessere Lösung):

```
double length() {
  return Math.sqrt(square(p2.x - p1.x)
                   + square(p2.y - p1.y));
}
```

mit der *privaten* Hilfsmethode

```
private double square(double x) {
  return x * x;
}
```

Übersicht

Zusammenfassung Klassen

Eine Klassen-Deklaration besteht aus Deklarationen von:

- Attributen für die verschiedenen Wert-Komponenten der Objekte;
- Konstruktoren zur Initialisierung der Objekte:
  Konstruktoren haben den gleichen Namen wie die Klasse.
  Es kann mehrere geben, sofern sie sich in Typ oder Anzahl ihrer Argumente unterscheiden.
  Sofern keine Konstruktoren mit ≥ 0 Argumenten implementiert werden, wird automatisch ein nullstelliger Default-Konstruktor zur Verfügung gestellt.
- Methoden und Prozeduren, d.h. Operationen auf Objekten.

Diese Elemente heißen auch Members der Klasse.

Klasse Rational

```
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;
      }
       ...
}
```

Klasse Rational

```
    // 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 void addToThis(Rational r) {
        zaehler = zaehler * r.nenner + r.zaehler * nenner;
        nenner = nenner * r.nenner;
    }
    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
```

Zusammenfassung

- 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.

Zusammenfassung

- Konstruktoren haben den gleichen Namen wie die Klasse.
- Es kann mehrere geben, sofern sie sich in Typ oder Anzahl ihrer Argumente unterscheiden.
- Konstruktoren haben keine Rückgabewerte und darum auch keinen Rückgabetyp.
- Methoden haben dagegen stets einen Rückgabe-Typ, evt. `void`.

Beispiel:

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

Zusammenfassung

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



Zusammenfassung

- Aktualparameter werden immer als Kopie per call-by-value übergeben. Bei Basistypen ist das eine Kopie des Werts, bei Referenztypen eine Kopie der Referenz.
- Wenn ein Parameter mit Referenztyp an eine Methode übergeben wird, kann innerhalb der Methode der durch den Parameter referenzierte Wert verändert werden. Diese Änderung gilt dann auch nach Verlassen der Methode!

```
class C {
  private Rational r1;
  public C() {
    r1=new Rational(1,2);
    incArgByOne(r1);
    // Wert von r1 jetzt 3/2 !!!!
  }
  private void incArgByOne(Rational r) {
    r.addToThis(new Rational(1));
  }
}
```

Aber Achtung: Übergabe von Referenzen

Weil bei Referenztypen als Argumente eine Kopie der Referenz übergeben wird, kann die dem Aktualparameter ursprünglich entsprechende Referenz nicht geändert werden, sondern nur der Wert, auf den diese Referenz verweist (so ein Beispiel haben wir eben gesehen!).

Hier Beispiel für problematischen Einsatz:

```
class T {
 int wert;
 public static void main(String[] argv) {
   T y=null;
   test(y);
   System.out.println(y.wert);  //Laufzeitfehler!!
 }

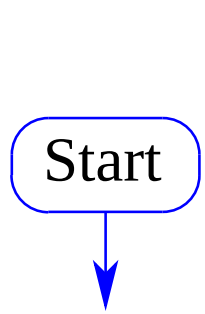
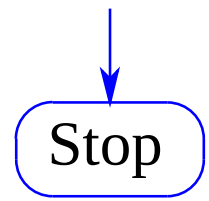
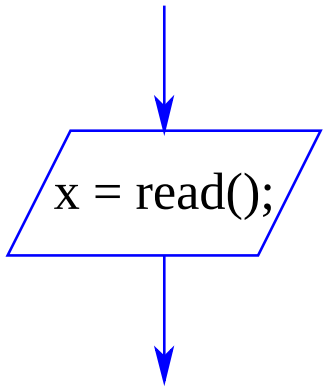
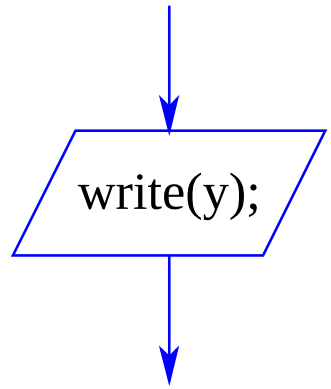
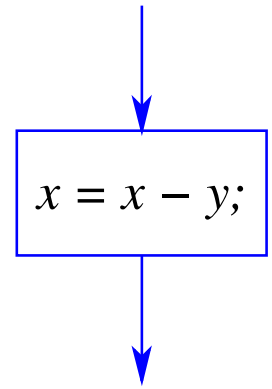
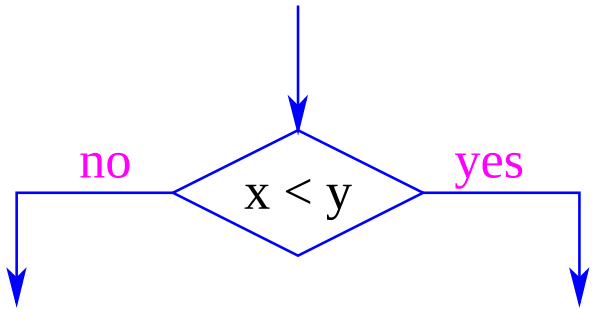
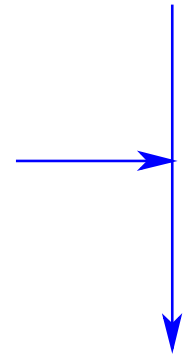
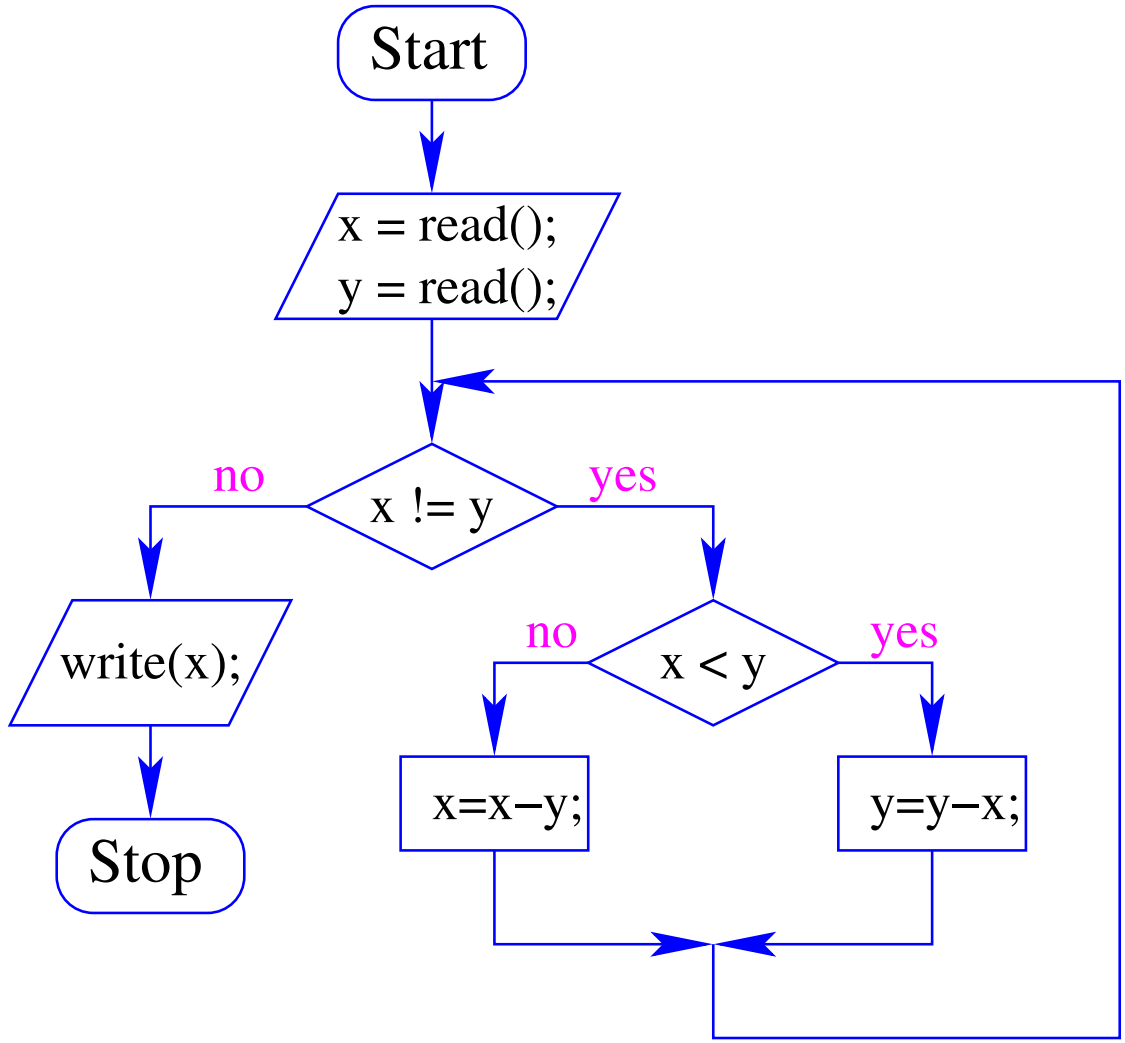
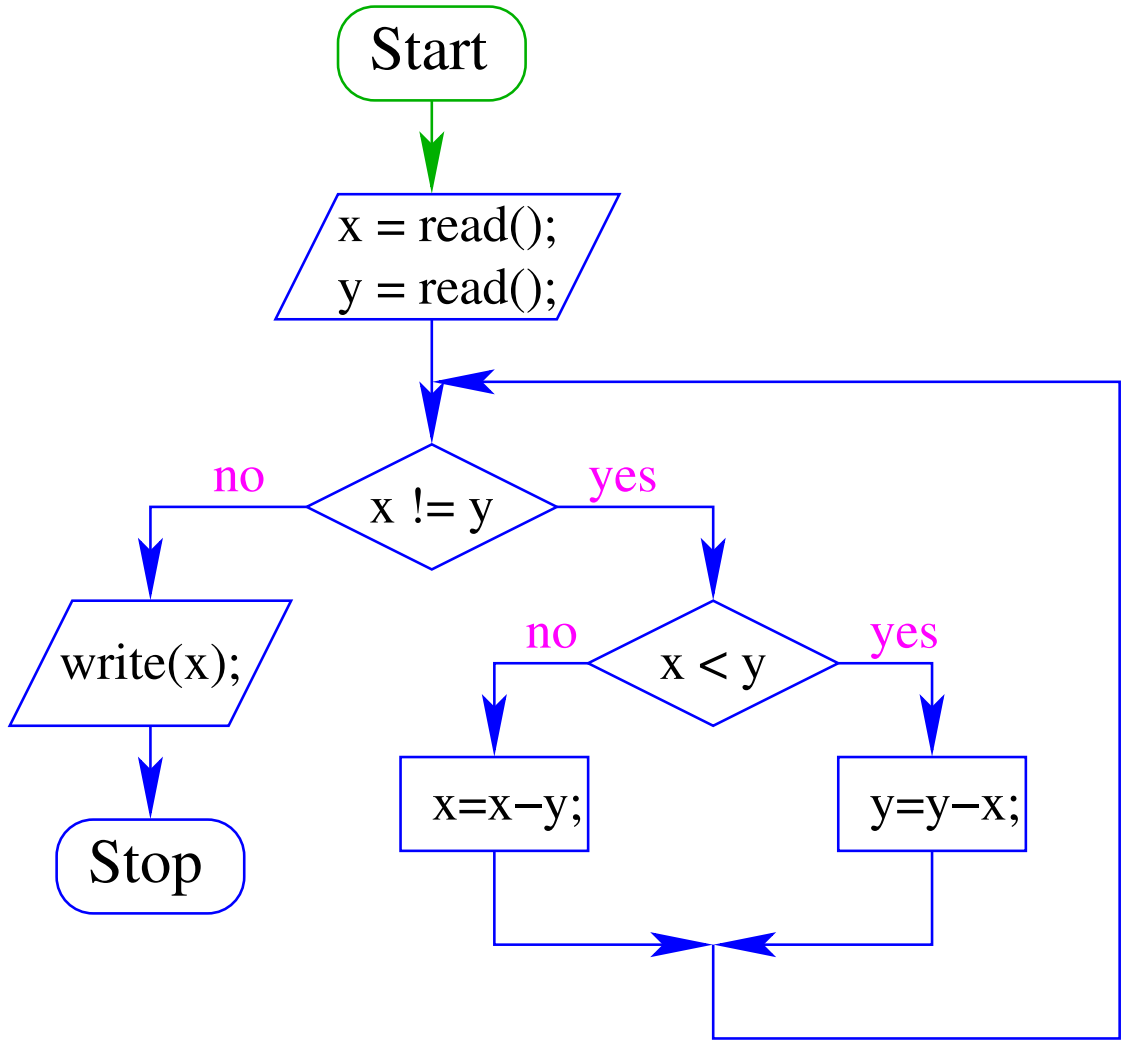
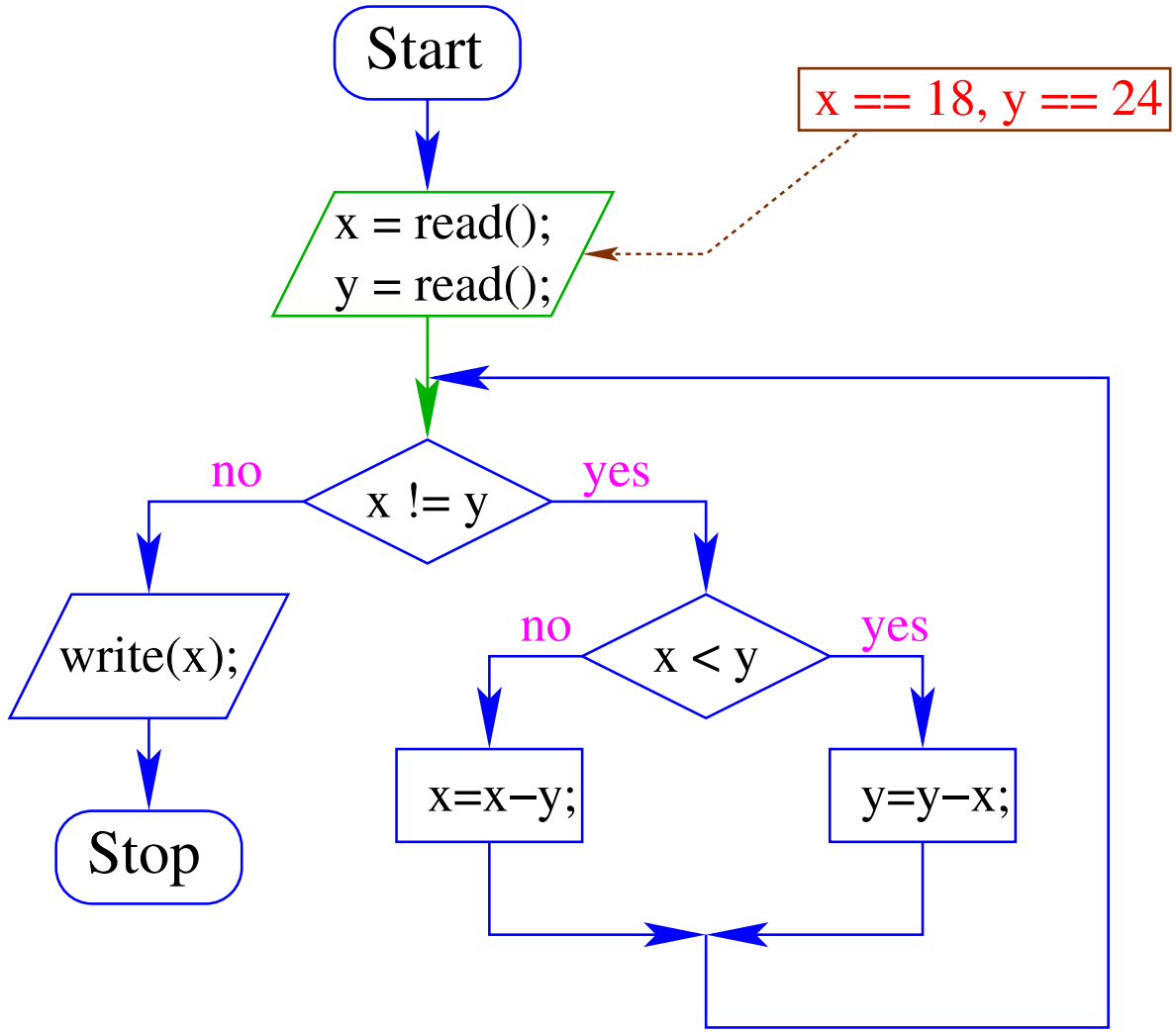
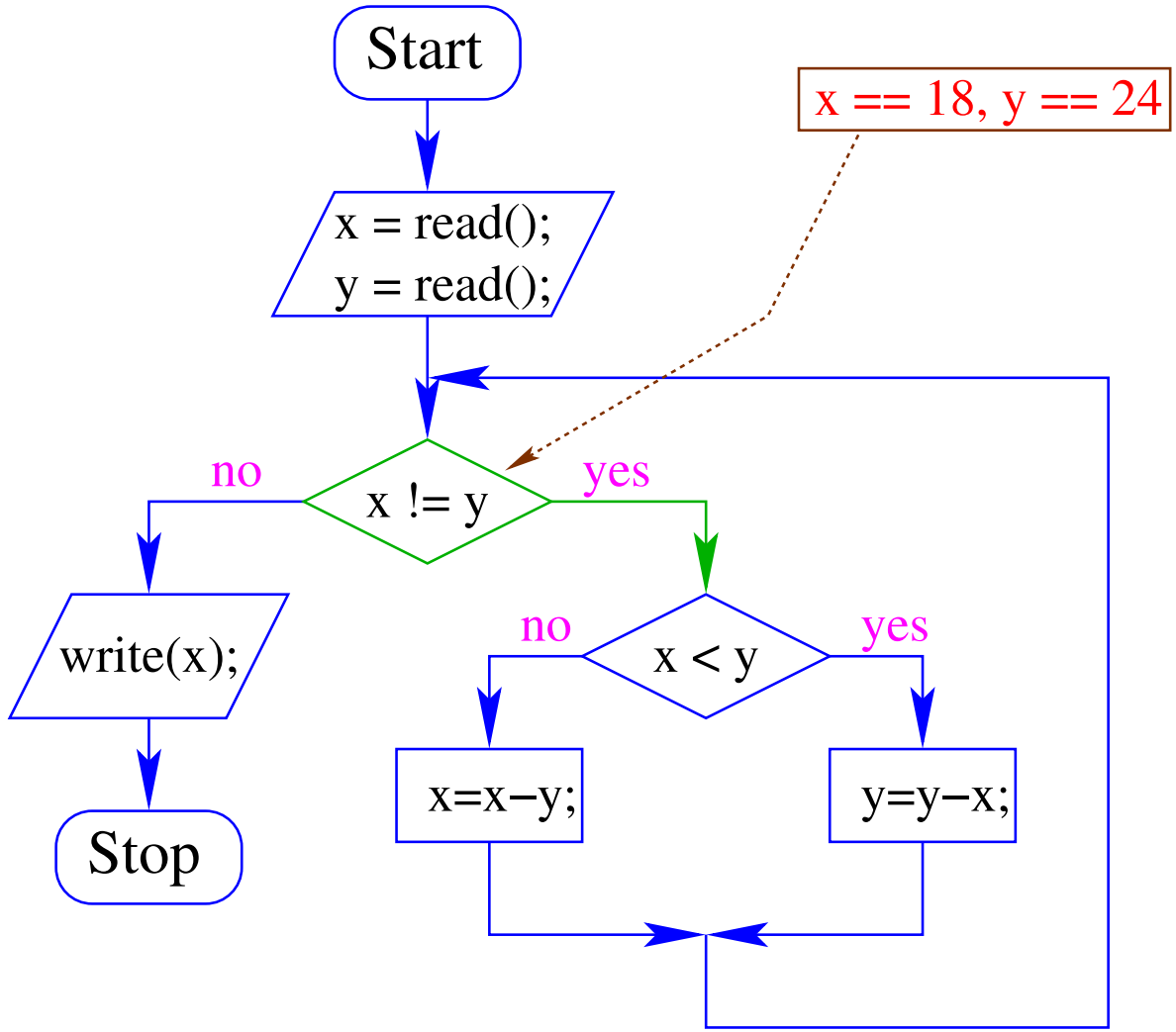
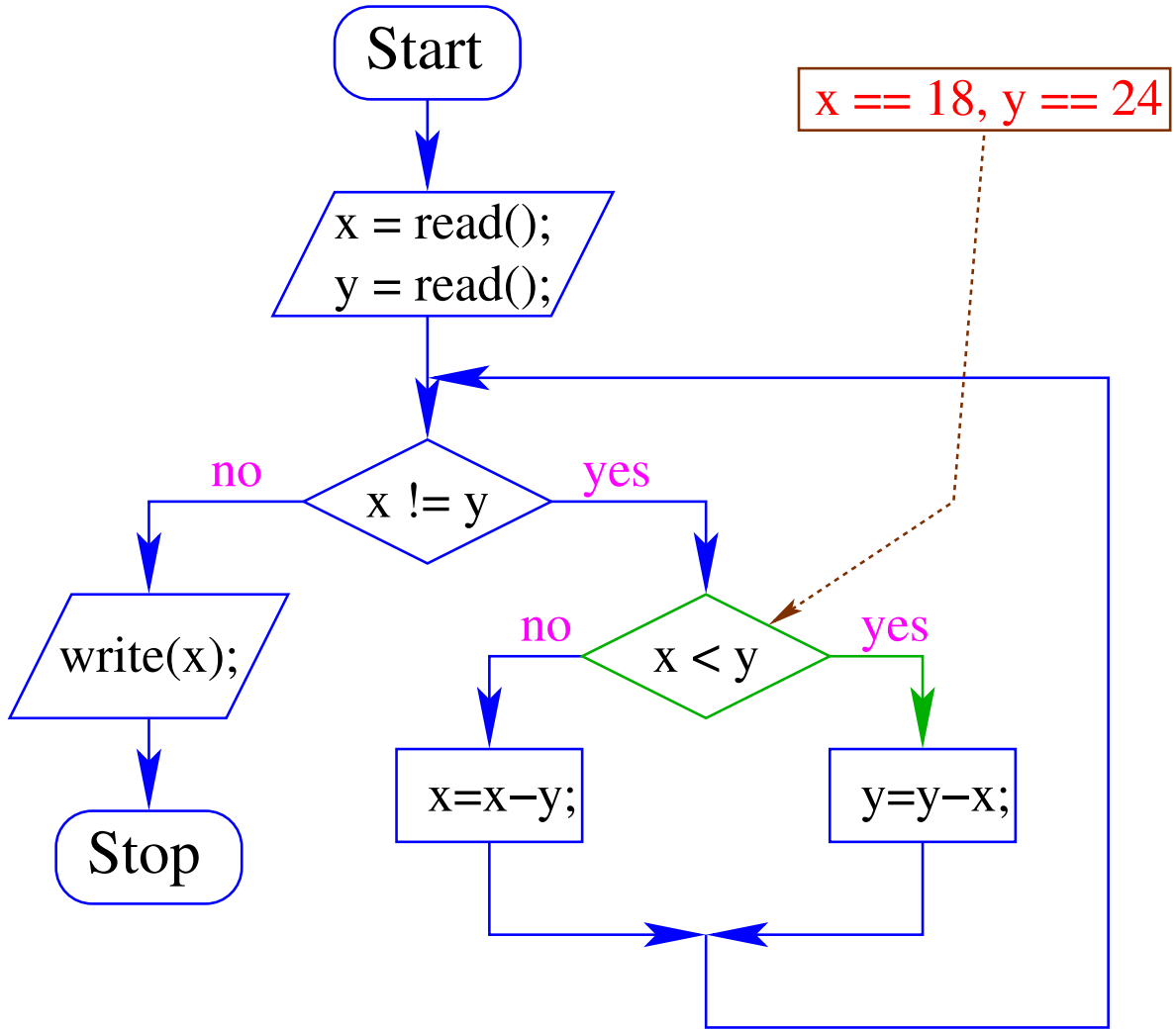
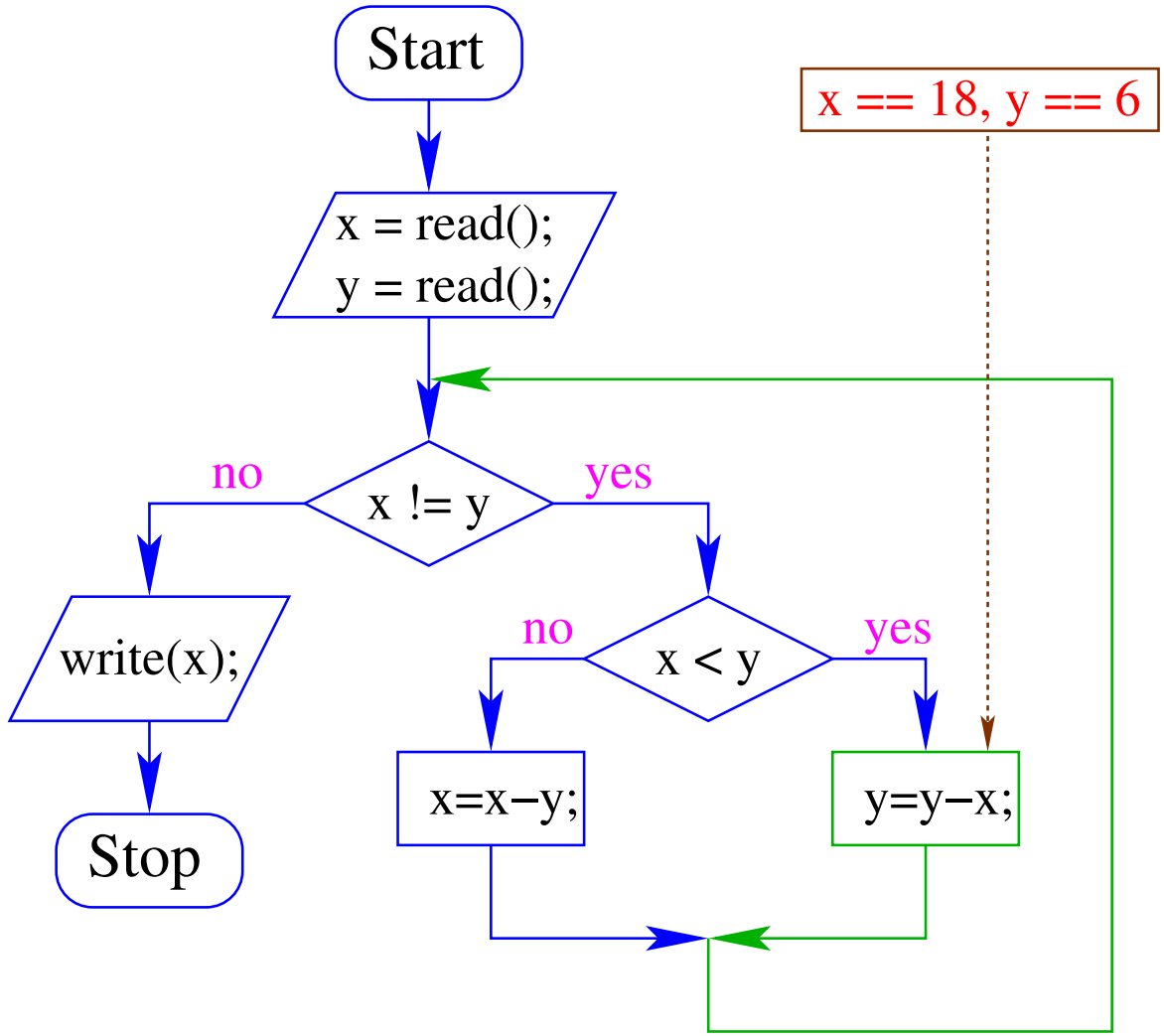
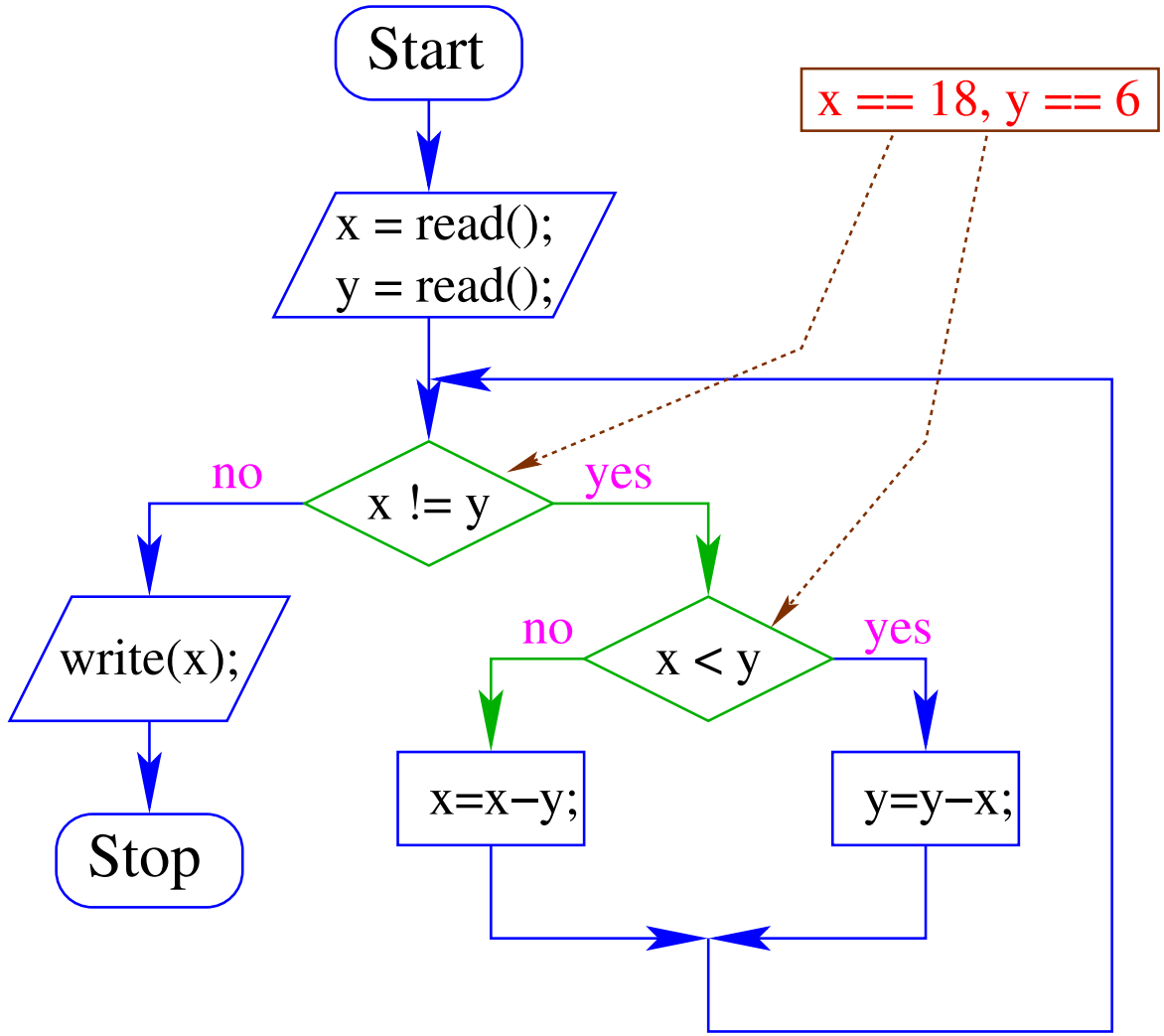
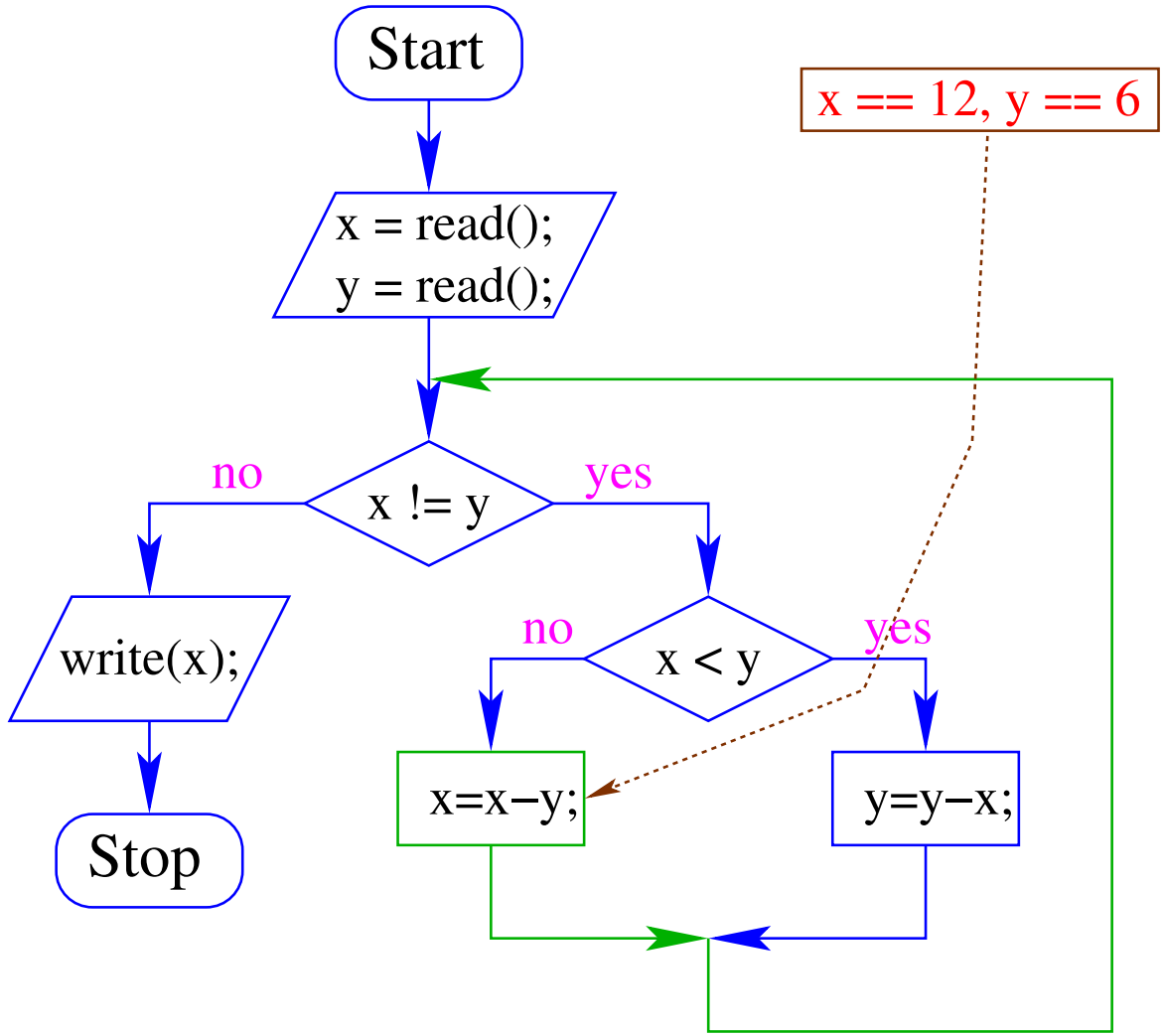
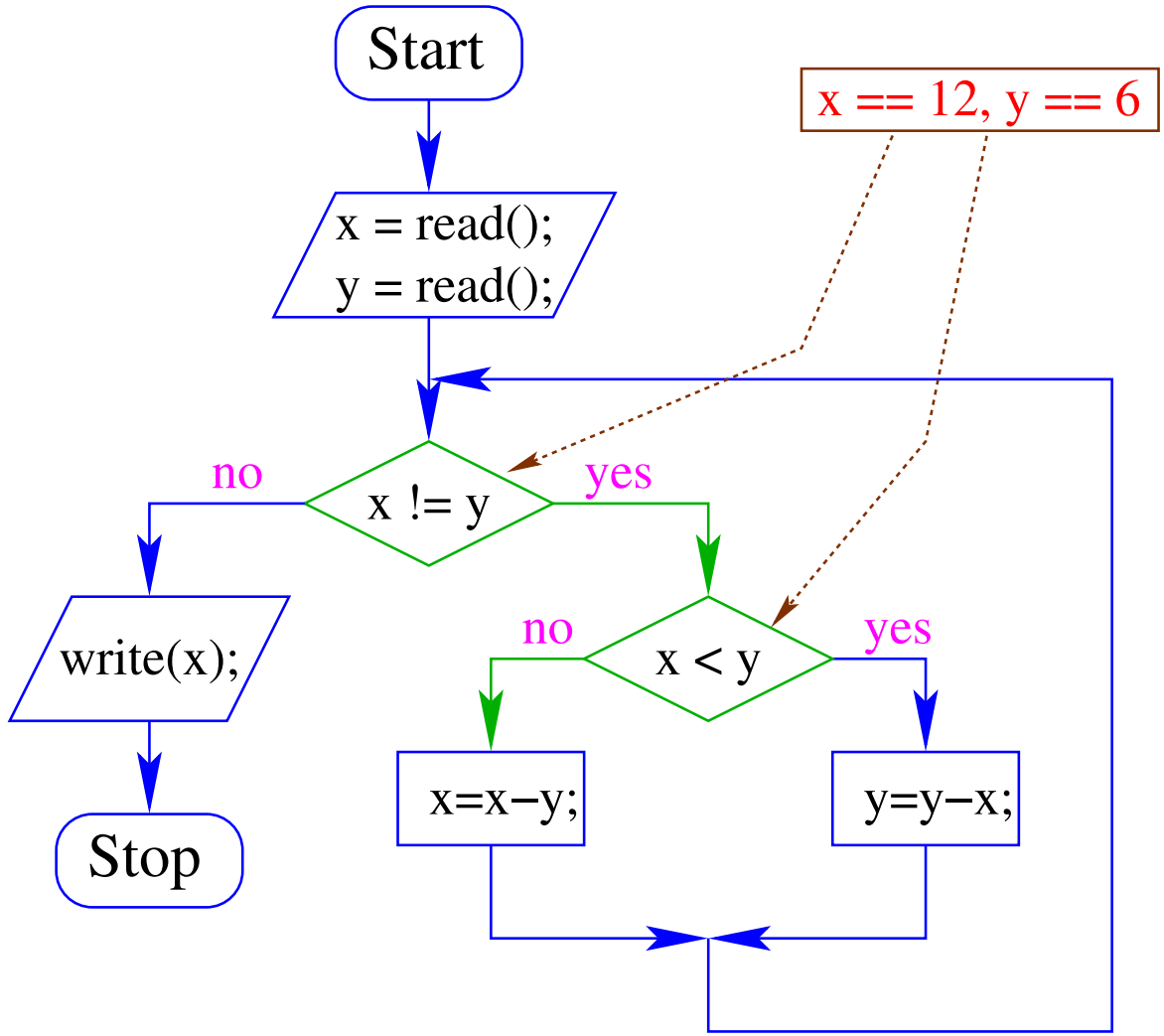
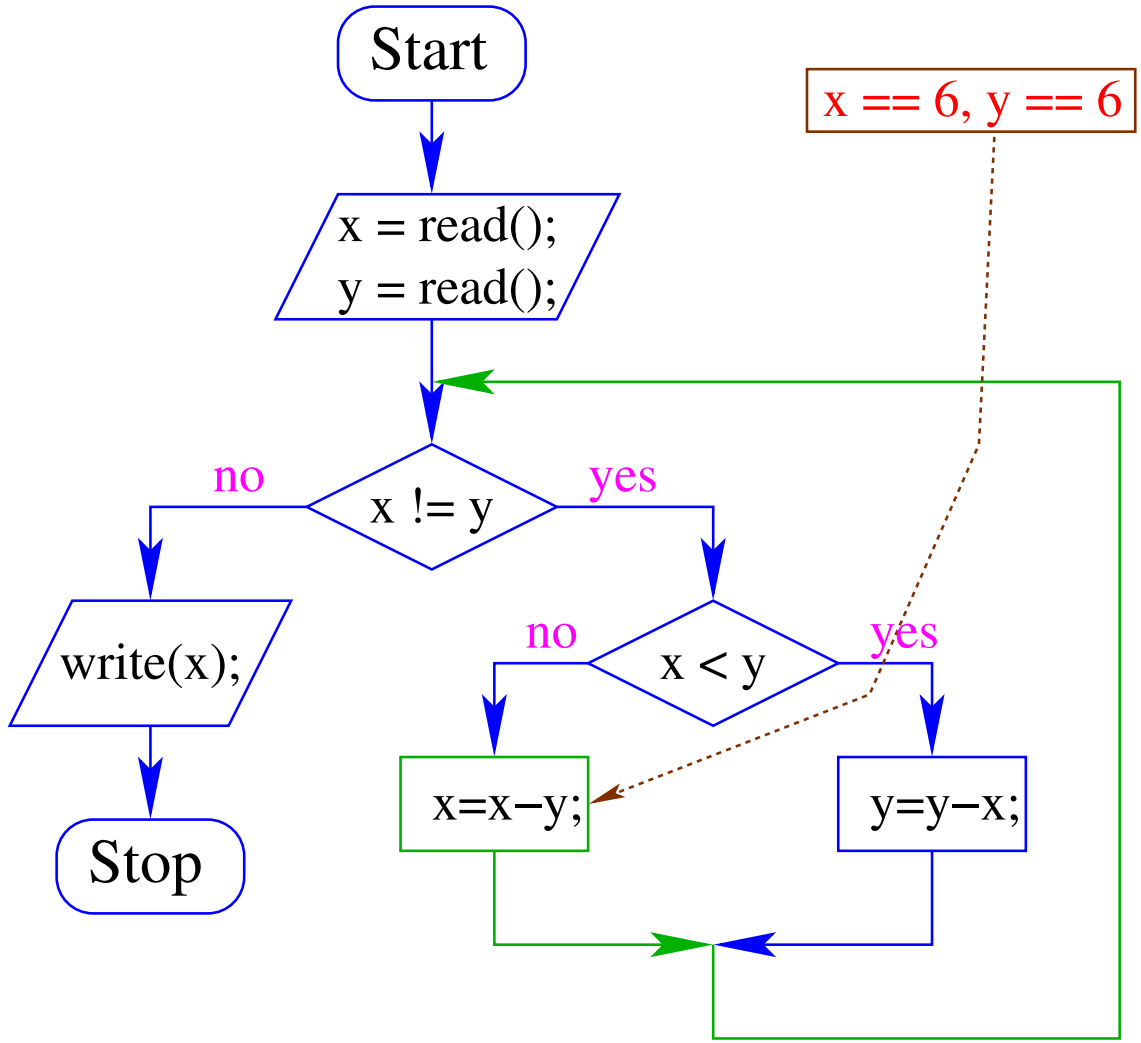
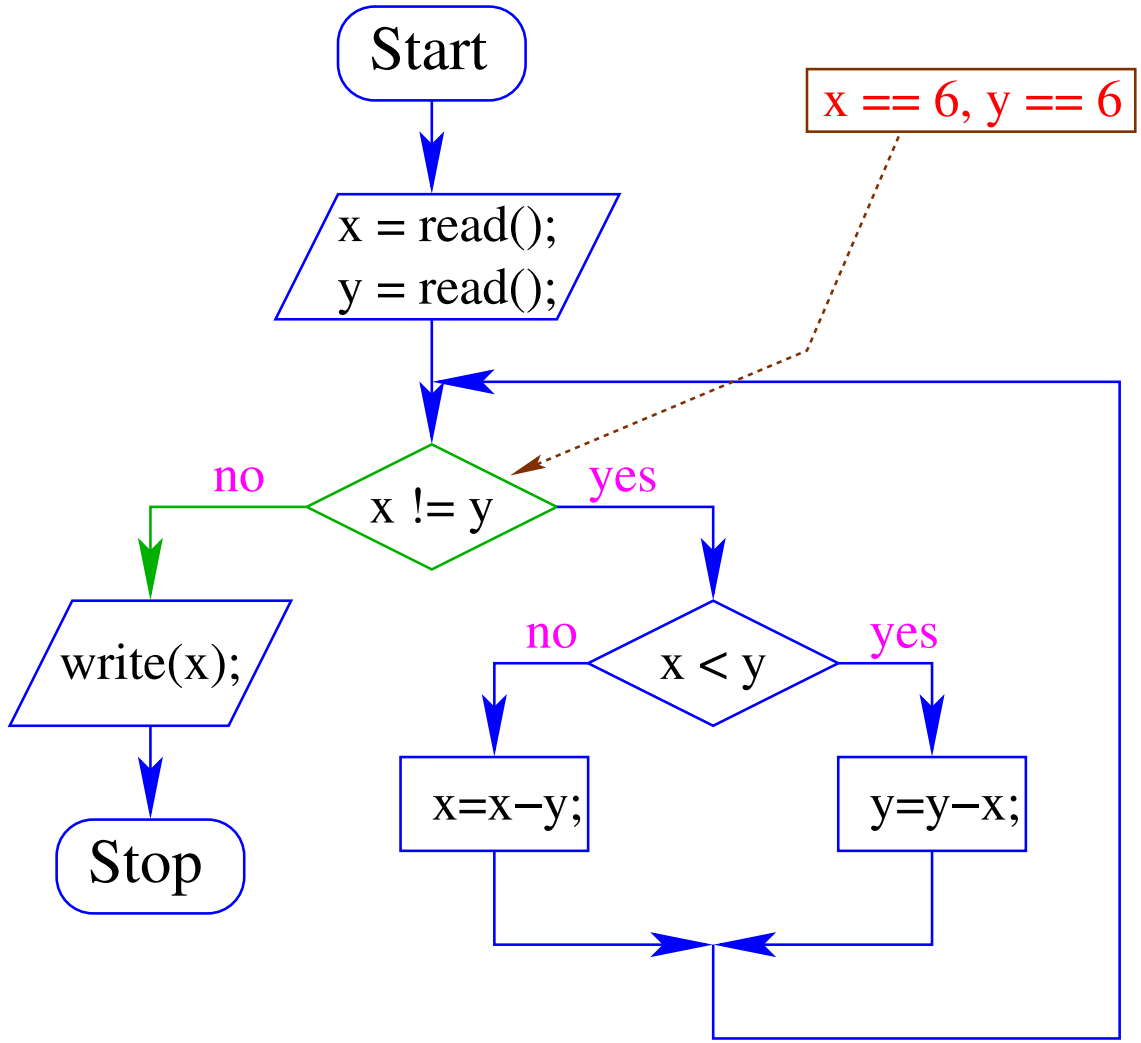
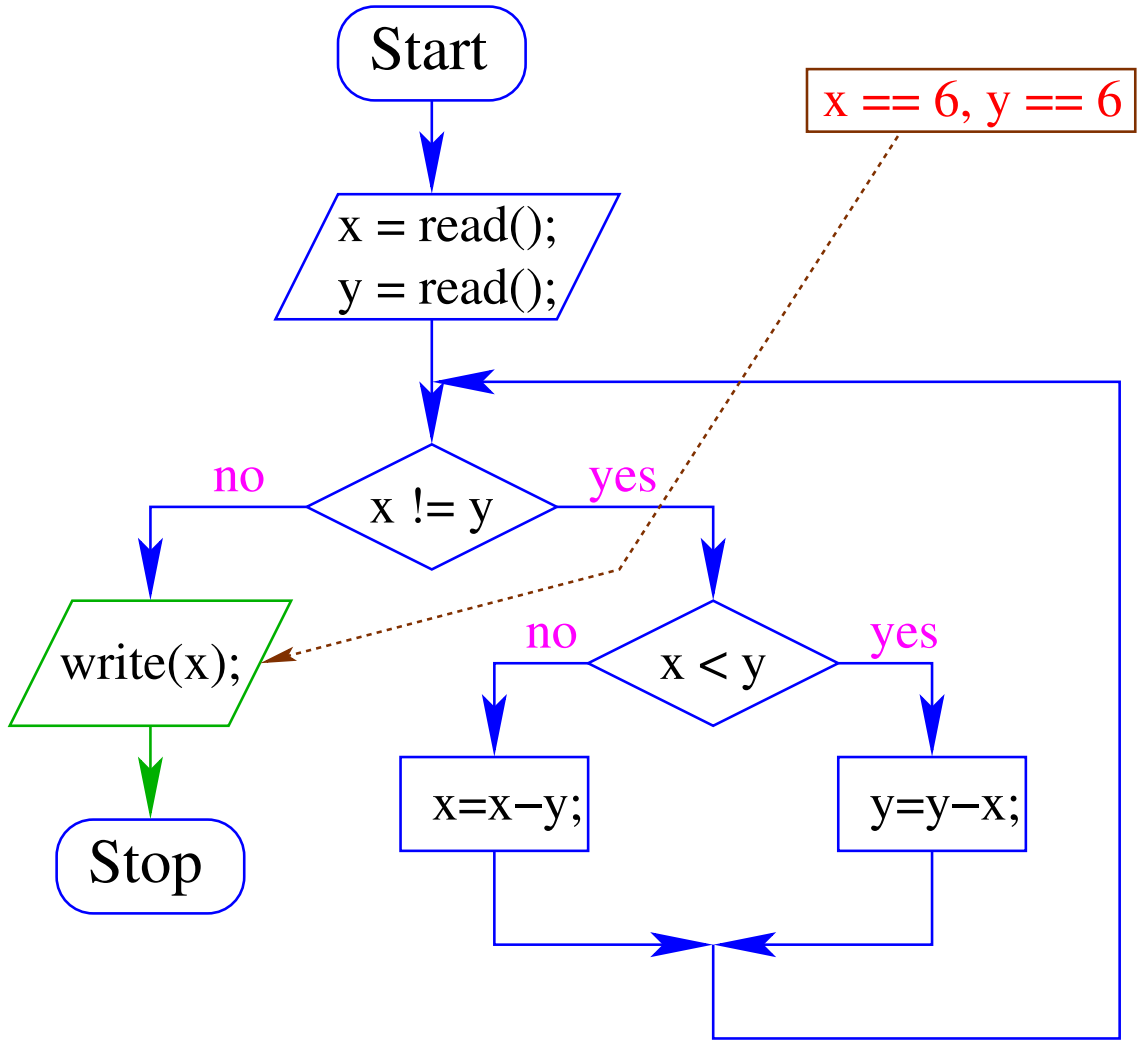
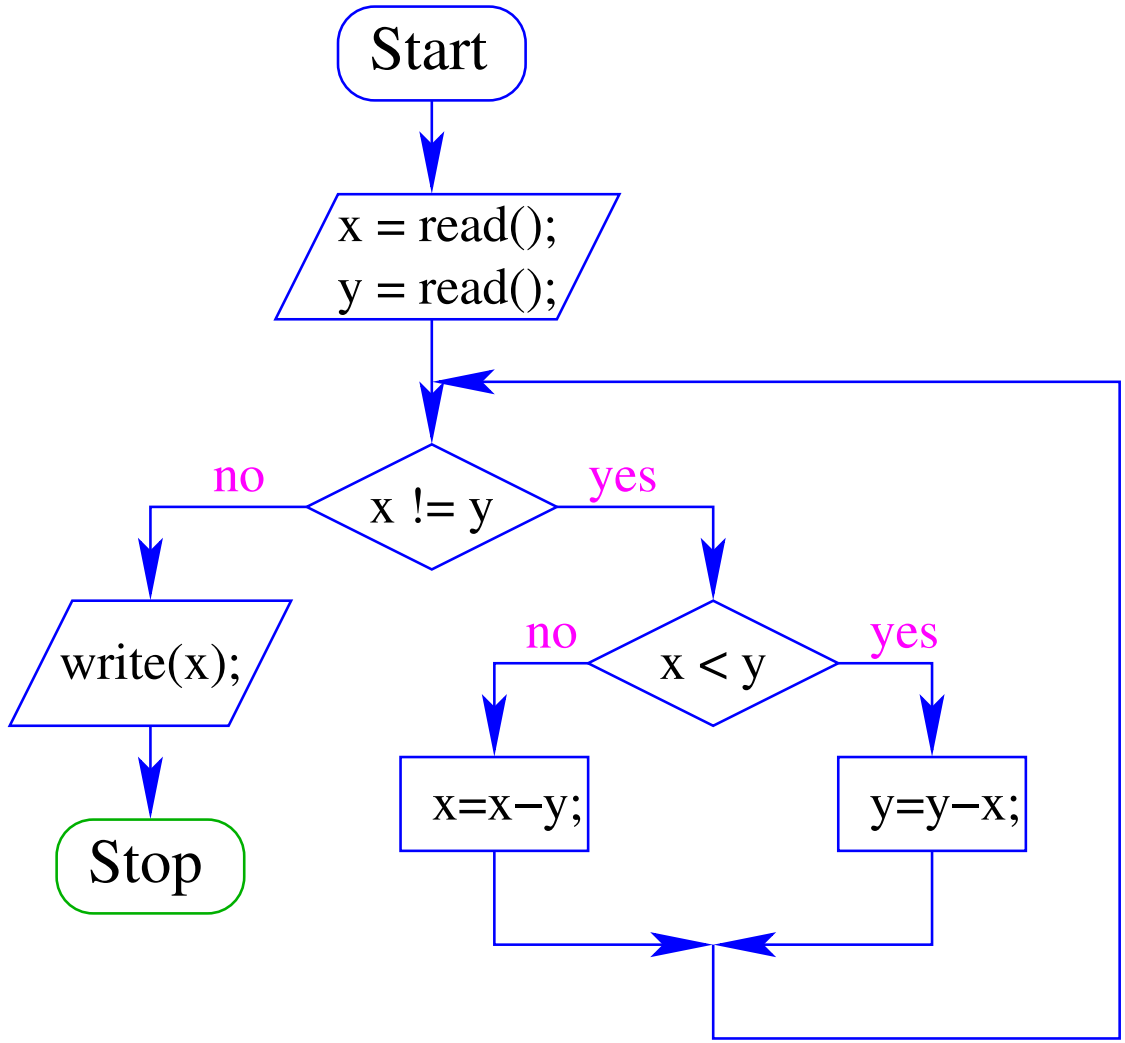
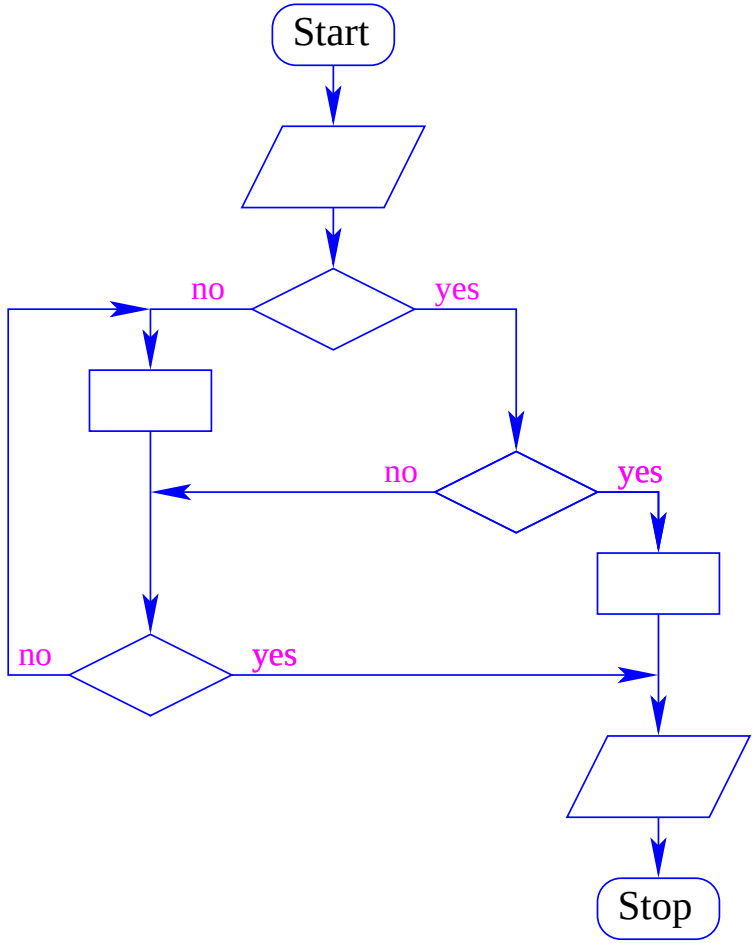
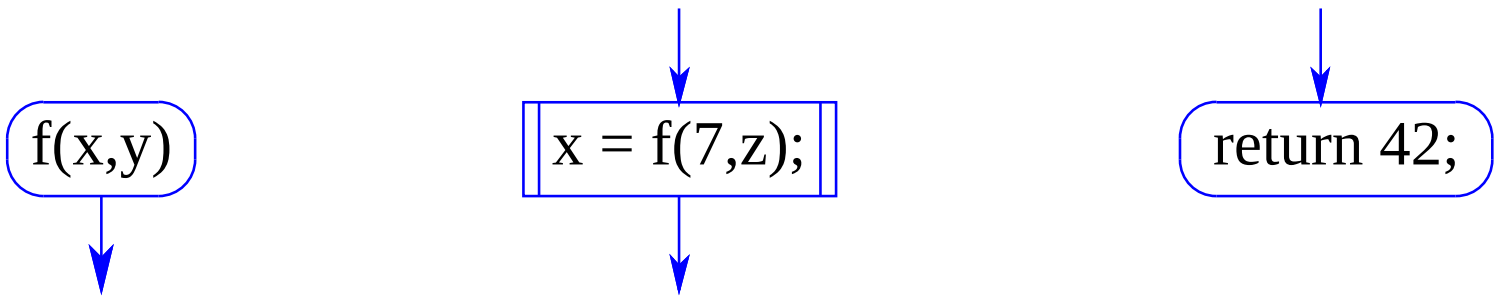
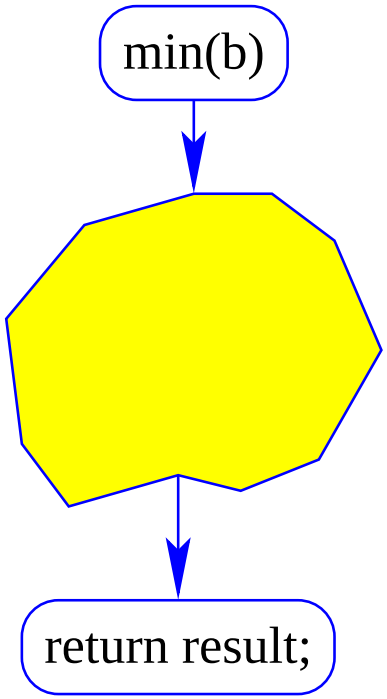
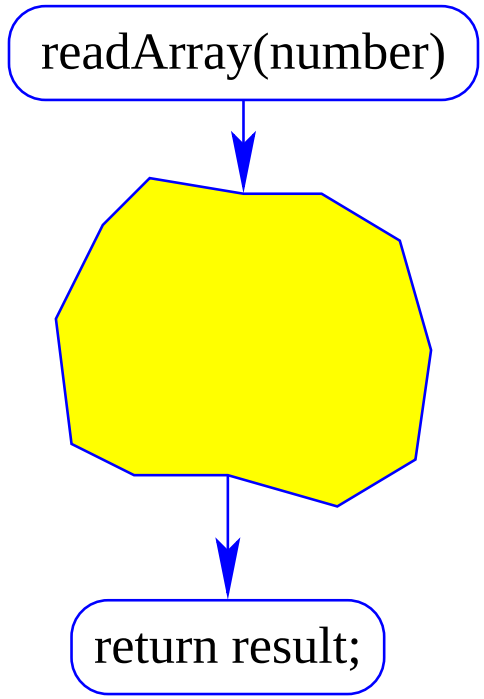
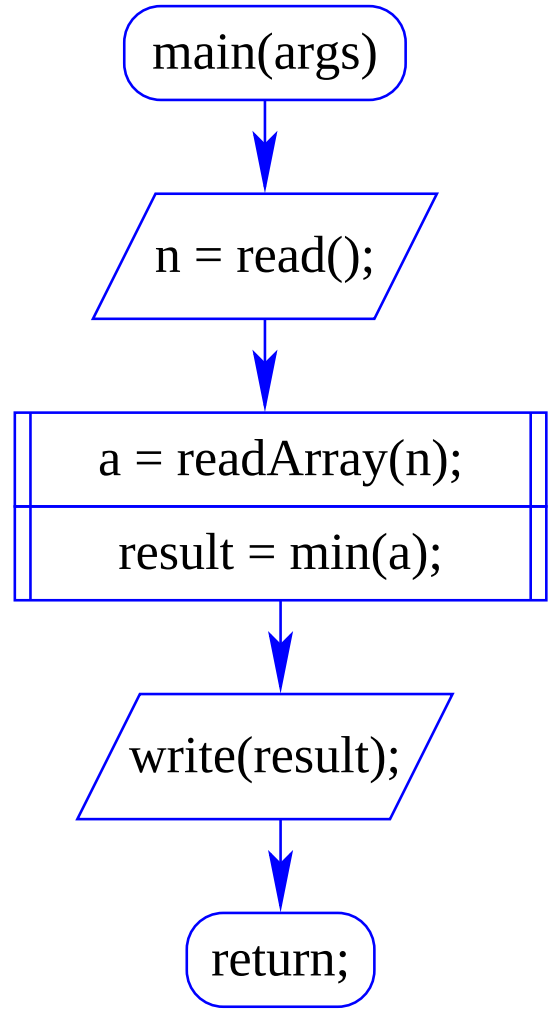
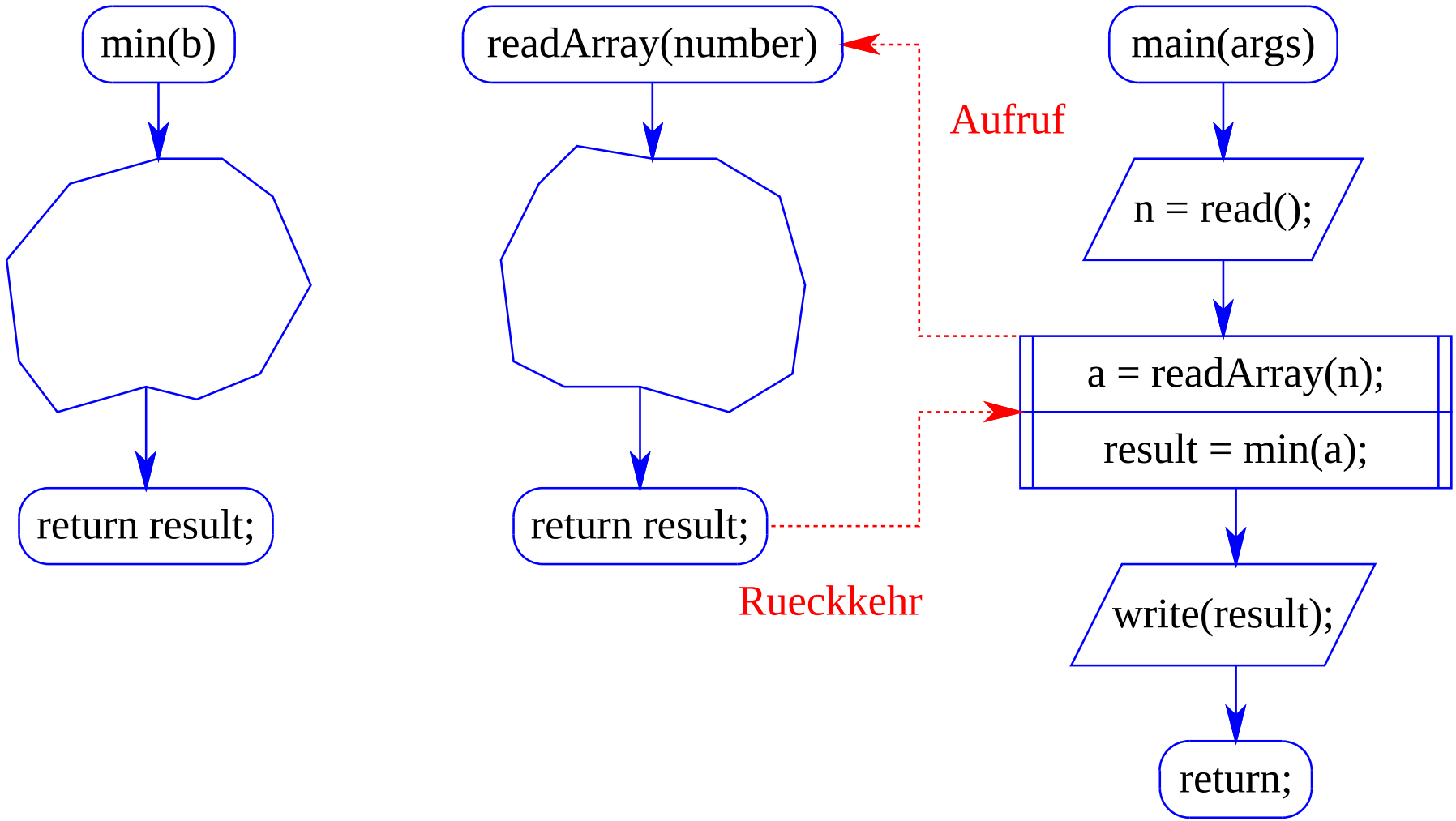
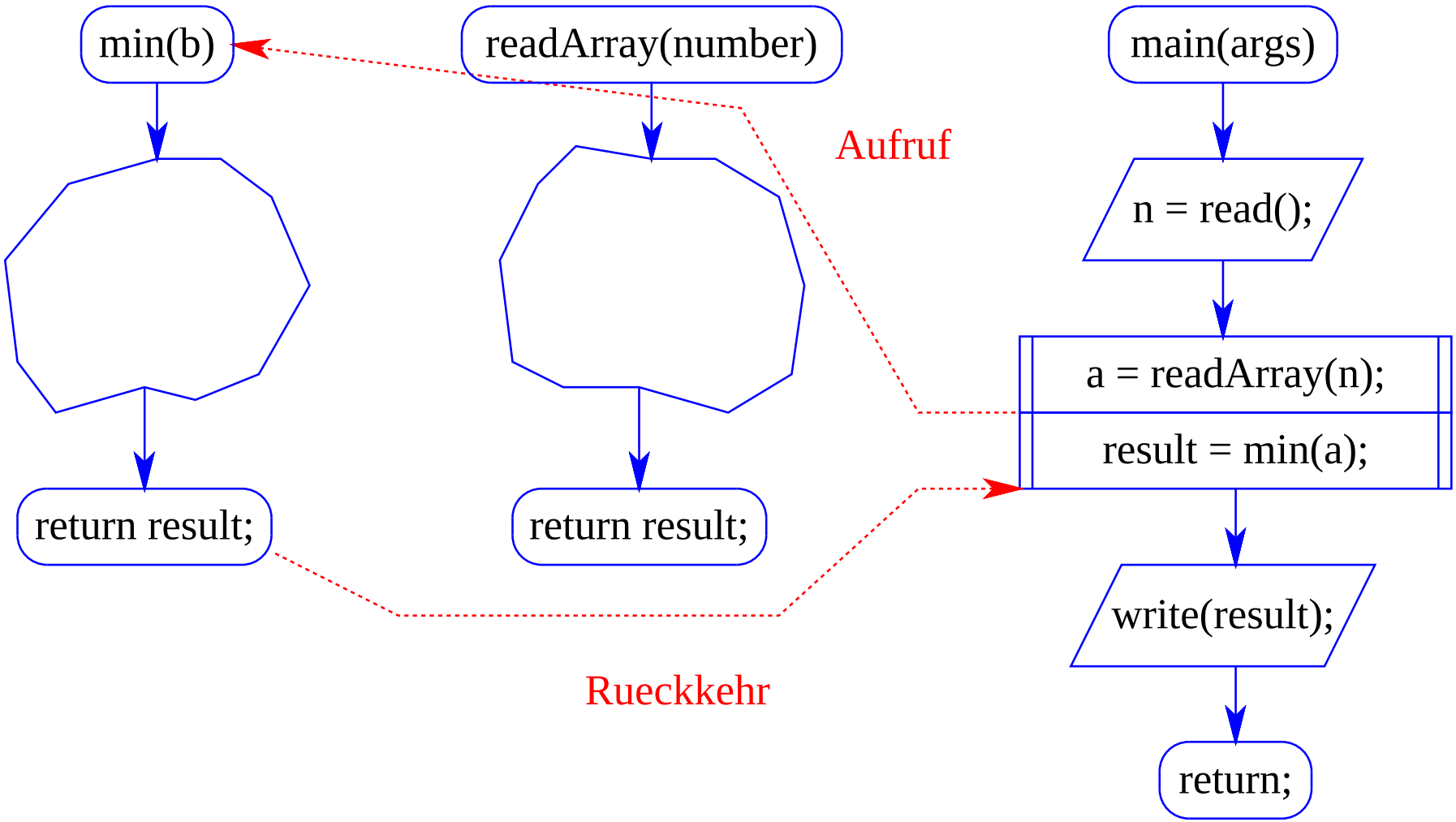
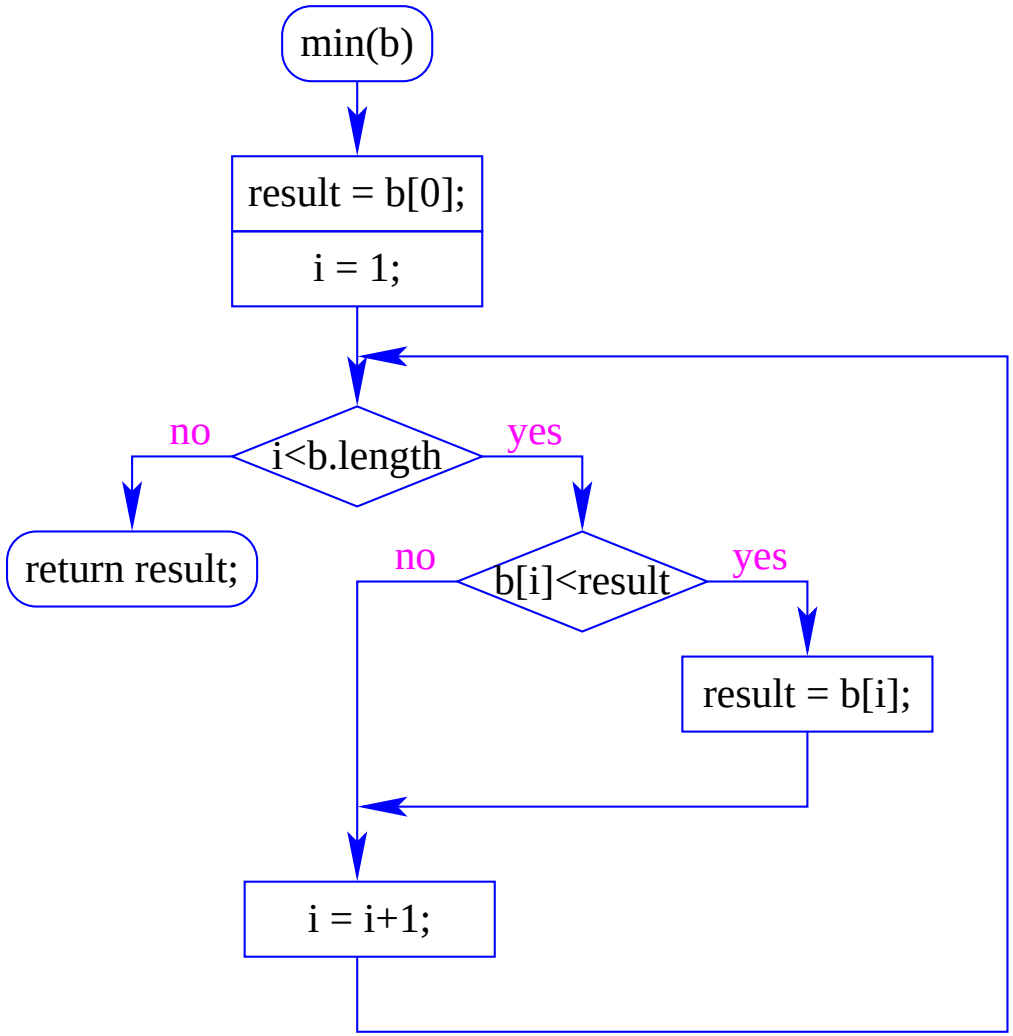
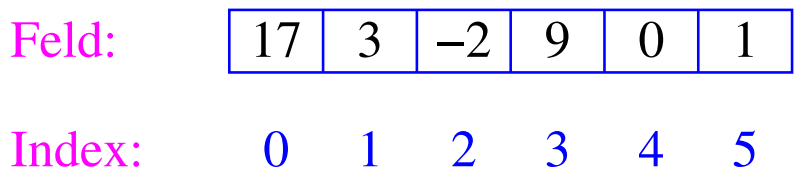
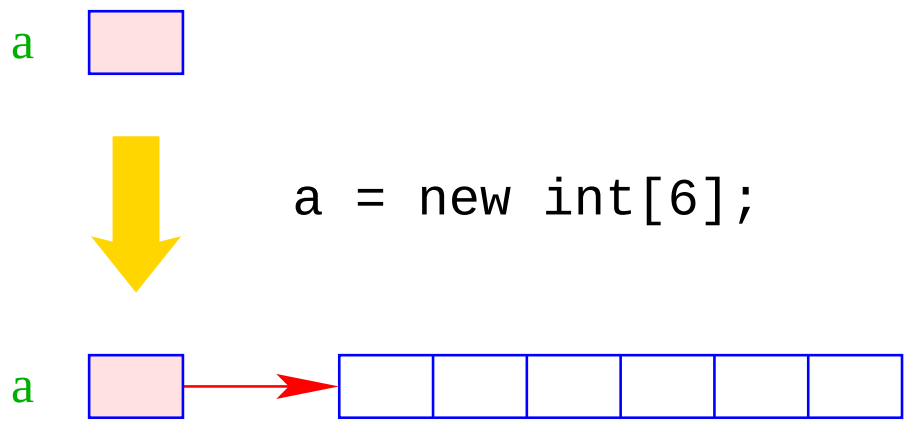
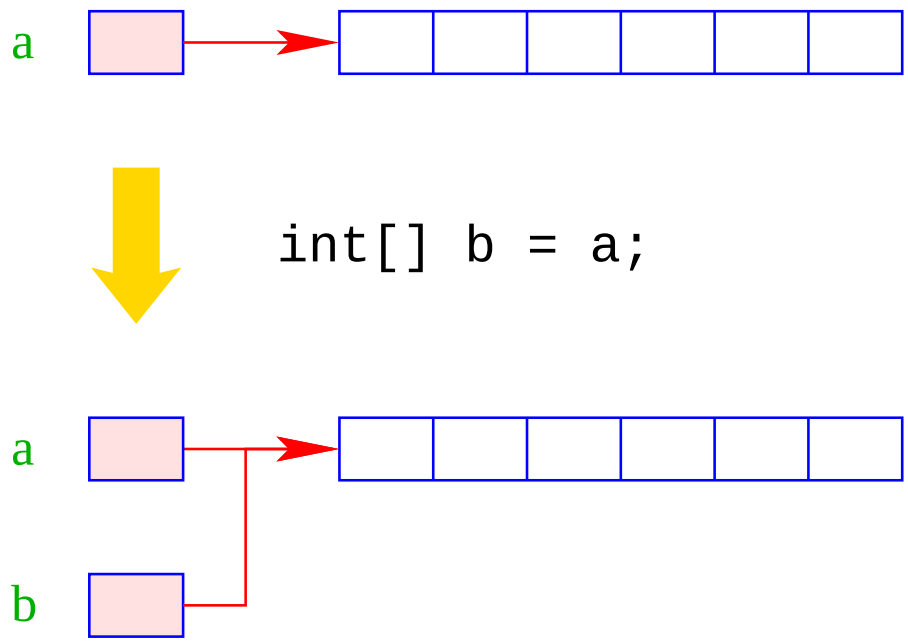
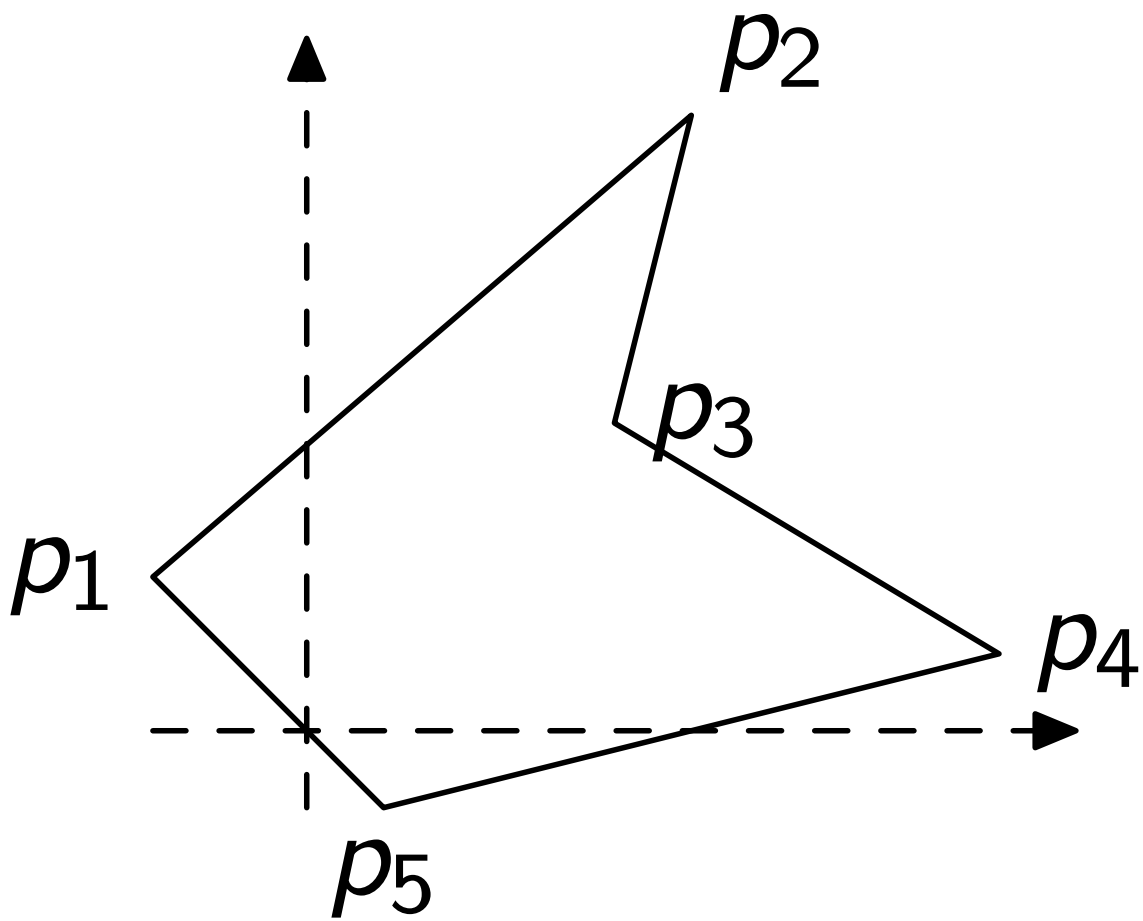

 private static void test(T x) {
   x=new T();
   x.wert=1;
   System.out.println(x.wert);
 }
}
```

Zusammenfassung

- 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 `inc` sichtbar !!

Übersicht

Suchmaschine

Objekte identifizieren



Klassenentwurf



Objekterzeugung



Objektmethoden



Suchmaschine: Objektmethoden

- Arbeiten auf den Objekten





Suchmaschine: Objektmethoden

- Arbeiten auf den Objekten



Suchmaschine: Objektmethoden

- Arbeiten auf den Objekten



Suchmaschine: Objektmethoden

- Arbeiten auf den Objekten



Suchmaschine: Objektmethoden

- Arbeiten auf den Objekten



Übersicht

Klassendiagramme: Klassen

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



Klassendiagramme—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.

Klassendiagramme—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.

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

Klassendiagramme: Referenzen

Referenzattribute werden häufig als Referenzen (Linien zwischen Klassen) notiert:



Klassendiagramme: Selbst-Referenzen



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.

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

Motivation

Eine Programmiersprache soll

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

Datenstrukturen und Operationen haben wir bereits gesehen!

Kontrollstrukturen verwenden Anweisungen und Ausdrücke.

Ausdrücke

Ausdrücke (expressions) werden in einem Programm verwendet,

- auf der rechten Seite von Zuweisungen: `x = Ausdruck;`
- als Argumente von Methoden:
  `f(Ausdruck,...,Ausdruck);`
- als Rümpfe von Funktionen: **`return`** *`Ausdruck`*`;`

Jeder Ausdruck hat einen **Typ**

Beispiele:

- „`(3.0 + y) * 4.0`“ ist ein Ausdruck vom Typ **`double`**
- „**`new`** `Point(1.0, 2.0)`“ ist ein Ausdruck vom Typ `Point`
- „`x == 2`“ ist ein Ausdruck vom Typ **`boolean`**

Einen Ausdruck vom Typ **`boolean`** nennt man *Bedingung*

Weitere Ausdrücke in Java

Ausdrücke setzen sich wie folgt zusammen

- *Variable* bzw. *Attribut*
- f(*Ausdruck*$_1$, ..., *Ausdruck*$_n$)
  (wenn Rückgabetyp von f nicht **void**)
- *Ausdruck*$_1$ $\oplus$ *Ausdruck*$_2$
- $\odot$ *Ausdruck*
- **new** ...

$\oplus$ bezeichnet dabei einen (bel.) binären Operator
$\odot$ steht für einen unären Operator

Es gibt in Java noch zwei weitere Ausdrücke;
hier nicht genauer betrachtet:

_ **instanceof** _	Klassenzugehörigkeits-Test
_ **?** _ **:** _	Bedingter Ausdruck (Wenn-Dann-Sonst)

Anweisungen

Anweisungen (statements)

- Veranlassen eine Zustandsänderung
- Haben **keinen** Typen

In Java gibt es eine Vielzahl von Anweisungen:

- Deklarationen lokaler Variablen: `int x;`
- Zuweisungen: `x = y;` (Ausnahme: hat einen Typen, wenn `a` und `b` kompatibel sind)
- Methodenaufrufe: `p(x, 1);` (wenn Rückgabetyp von `p` **`void`**)
- Block-Anweisungen:
  `{` *Anweisung*$_1$`;` `...` `;` *Anweisung*$_n$`;` `}`
- Return-Statements: **`return`** *Ausdruck*`;`
- ...

Gegenüberstellung Ausdruck – Anweisung

Unterschiede zwischen Ausdrücken und Anweisungen

	Ausdruck	**Anweisung**
Typisiert	ja	nein
Zweck	„Berechne ...“	„Mache ...“
Effekt	liefert Wert	ändert Zustand
Syntax	ohne „`;`“	mit „`;`“

- Ein Ausdruck ist immer Teil einer Anweisung
- Der Rumpf jeder Methode ist immer eine *Folge von Anweisungen*
- In Java gibt es auch sogenannte „*ExpressionStatements*“; Das sind Anweisungen, die gleichzeitig auch als Ausdruck verwendet werden können
  - Beispiel: `i++`

Sequenz ";"

```
int x, y, result;
x = Terminal.askInt("Zahl 1: ");
y = Terminal.askInt("Zahl 2: ");
result = x + y;
System.out.println(result);
```

Sequenz ";"

- 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 = Terminal.askInt("Zahl␣1:␣");
y = Terminal.askInt("Zahl␣2:␣");
if (x > y)
    result = x - y;
else
    result = y - x;
System.out.println(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 = Terminal.askInt("Zahl␣1:␣");
if (x == 0)
    System.out.println("0");
else if (x < 0)
    System.out.println("-1");
else
    write(+1);
```

Beachte

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

```
int x, y;
x = Terminal.askInt("Zahl 1: ");
if (x != 0) {
    y = Terminal.askInt("Zahl 2: ");
    if (x > y)
        System.out.println(x);
    else
        System.out.println(y);
} else
    System.out.println(0);
```

Beachte

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

```
int x, y;
x = Terminal.askInt("Zahl 1: ");
if (x != 0) {
    y = Terminal.askInt("Zahl 2: ");
    if (x > y)
        System.out.println(x);
    else
        System.out.println(y);
}
```

Die Switch-Anweisung

Die **Switch-Anweisung** (switch statement) realisiert eine weitere Form der Verzweigung

```
static final char NEW  = 'n';
static final char OPEN = 'o';
static final char SAVE = 's';
static final char QUIT = 'q';

void doCommand() {
   char command = Terminal.askChar("Kommando ?");
   switch (command) {
      case NEW  : createNewFile();
                  break;
      case OPEN : openFile();
                  break;
      case SAVE : saveFile();
                  break;
      case QUIT : exitProgram();
                  break;
      default   : System.out.println("Unbekanntes Kommando: " + command);
                  break;
   }
}
```

Die Switch-Anweisung – Allgemein

Syntax:

- **switch** (*Ausdruck*) {
  case $Wert_1$: $Anweisungen_1$
  ...
  case $Wert_n$: $Anweisungen_n$
  default : *Anweisungen*
  }
- *Ausdruck* muss dabei den Typ **char**, **byte**, **short** oder **int** haben
- Die Werte nach **case** müssen konstant sein (keine Variablen!)
- Ein „**case** *Wert*“ legt nur den Einstiegspunkt innerhalb des **switch**-Blocks fest
- Die **break**-Anweisung veranlasst das (sofortige) Verlassen des gesamten **switch**-Blocks
- Ohne **break** werden auch alle Anweisungen der nachfolgenden **case**-Blöcke abgearbeitet

Die Switch-Anweisung – Beispiel

```
int tageImMonat(int monat) {
  int tage = 0;
  switch (monat) {
    case  1: tage = 31; break;
    case  2: tage = 28; break;
    case  3: tage = 31; break;
    case  4: tage = 30; break;
    case  5: tage = 31; break;
    case  6: tage = 30; break;
    case  7: tage = 31; break;
    case  8: tage = 31; break;
    case  9: tage = 30; break;
    case 10: tage = 31; break;
    case 11: tage = 30; break;
    case 12: tage = 31; break;
  }
  return tage;
}
```

```
int tageImMonat(int monat) {
  int tage;
  switch (monat) {
    case  2: tage = 28; break;
    case  4:
    case  6:
    case  9:
    case 11: tage = 30; break;
    default: tage = 31; break;
  }
  return tage;
}
```

- Rechte Variante zwar kürzer, aber schlechter lesbar und es werden auch Monate größer 12 und kleiner 1 akzeptiert
  $\Longrightarrow$ Fehlerfindung erschwert

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

Iteration (wiederholte Ausführung)

```
int x, y;
x = Terminal.askInt("Zahl␣1:␣");
y = Terminal.askInt("Zahl␣2:␣");
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
System.out.println(x);
```

while

- Syntax: `while` (Bedingung) `{` Rumpf; `}`
- 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.

Beispiel Fakultätsfunktion

```
int i=Terminal.askInt();
int fac=1;
if(i<0) return -1;
if(i==0) return 1;
while(i > 0){
  fac=fac*i;
  i--;
  // kürzer: fac *= i--;
}
```

break revisited

- Manchmal möchte man eine Schleife verlassen, *bevor* alle Schleifendurchläufe abgearbeitet wurden
- „**break;**“ veranlasst auch das sofortige Verlassen der innersten Schleife (allgemein: des innersten Blocks)

Beispiel: Summe eingelesener Zahlen bis zur Eingabe '0'

```
void sum() {
   int i;                                    // Vorbereitung
   int sum = 0;

   while (true) {                            // Schleife
      i = Terminal.askInt("i = ");
      if (i == 0) { break; }
      sum = sum + i;
   }

   System.out.println("sum = " + sum); // Nachbereitung
}
```

- **break** sollte nur sparsam und gezielt eingesetzt werden, so dass der Code übersichtlich und verständlich bleibt

Berechenbarkeit

Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, lässt sich mit Selektion, Sequenz und Iteration berechnen !!

Beweis: ↑ Berechenbarkeitstheorie.

Idee:

Eine Turing-Maschine kann alles berechnen...

Versuche, eine Turing-Maschine zu simulieren!

(Do-)While-Schleife – Syntax

Syntax:

- **while** (*Bedingung*) {*Anweisungen*}
- **do** {*Anweisungen*} **while** (*Bedingung*);

Do-While-Schleife – Beispiel

Beispiel:
Berechne die Summe eingelesener Zahlen bis zur Eingabe '0'

```
void sum() {
  int i;                                    // Vorbereitung
  int sum = 0;

  do {                                      // Schleife
    i = Terminal.askInt("i = ");
    if (i != 0) { sum = sum + i; }
  } while (i != 0);

  System.out.println("sum = " + sum); // Nachbereitung
}
```

- Do-While-Schleifen werden mindestens einmal durchlaufen, da das Abbruchkriterium erst am Schleifenende überprüft wird

Übersicht

Kontrollfluss-Diagramme (I)

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

Ingredienzien:



Startknoten



Endknoten

Kontrollfluss-Diagramme (II)



Eingabe



Ausgabe



Zuweisung

Kontrollfluss-Diagramme (III)



bedingte Verzweigung

Kontrollfluss−Transfer



Zusammenlauf

Beispiel

```
int x, y;
x = Terminal.askInt("Zahl 1: ");
y = Terminal.askInt("Zahl 2: ");
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```



Operationelle Semantik

- 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.

Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Beispielablauf ggT



Äquivalenz

- Zu jedem Java-Programm lässt sich ein Kontrollfluss-Diagramm konstruieren;
- Die umgekehrte Richtung gilt zwar ebenfalls, liegt aber nicht so auf der Hand.

Beispiel



Funktionen in Flussdiagrammen

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



Startknoten der Funktion f

Funktionsaufruf

Endknoten

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

Beispiel


min(b)
return result;


readArray(number)
return result;


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

Beispiel


min(b)
return result;
readArray(number)
return result;
Aufruf
Rueckkehr
main(args)
n = read();
a = readArray(n);
result = min(a);
write(result);
return;

Beispiel


min(b)
readArray(number)
main(args)
Aufruf
n = read();
a = readArray(n);
result = min(a);
return result;
return result;
write(result);
Rueckkehr
return;

mit Aufruf von `min()`



Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

Felder

Oft müssen viele Werte gleichen Typs gespeichert werden.

Idee:

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



Grundlagen I

- `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:



Grundlagen II

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



Grundlagen III

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

Beispiel 1: Polygone im $\mathbb{R}^2$



Code

```
class Polygon {

  // Attribute: Array von Eckpunkten
  Point[] nodes;

  // Konstruktormethode
  Polygon(Point[] nodes) {
    this.nodes = nodes;
  }

  // andere Methoden
}
```

Polygone erzeugen (I)

- **Variante 1: Explizite Angabe der Array-Länge**

```
Point[] points = new Point[5];

points[0] = new Point(-2.0, 2.0);
points[1] = new Point(5.0, 8.0);
points[2] = new Point(4.0, 4.0);
points[3] = new Point(9.0, 1.0);
points[4] = new Point(1.0, -1.0);

Polygon poly = new Polygon(points);
```

Polygone erzeugen (II)

- **Variante 2: Explizite Angabe von Punkten**

```
Point p1 = new Point(-2.0, 2.0);
Point p2 = new Point(5.0, 8.0);
Point p3 = new Point(4.0, 4.0);
Point p4 = new Point(9.0, 1.0);
Point p5 = new Point(1.0, -1.0);
Point[] points = { p1, p2, p3, p4, p5 };
Polygon poly = new Polygon(points);
```

- **Variante 3: Anonyme Punkte**

```
Polygon poly = new Polygon(
                        new Point[] { new Point(-2.0, 2.0),
                                      new Point(5.0, 8.0),
                                      new Point(4.0, 4.0),
                                      new Point(9.0, 1.0),
                                      new Point(1.0, -1.0)
                                    });
```

Beispiel 2: Einlesen eines Felds - mit while

```
int[] a; // Deklaration
int n = Terminal.askInt("Anzahl: ");

a = new int[n];
          // Anlegen des Felds
int i = 0;
while (i < n) {
    a[i] = Terminal.askInt("nächste Zahl: ");
    i = i+1;
}
```

Muster

Typische Form der Iteration über Felder:

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

Beispiel 3: Bestimmung des Minimums mit while

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

... oder mithilfe des `for`-Statements:

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

Semantik

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

... entspricht:

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

... wobei `i++` äquivalent ist zu `i = i+1` .

++ und −

- 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;
```

Warnung

- Die Zuweisung `i = i+1` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variable `i` 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)   // ACHTUNG!
   System.out.println("Sorry! This must be an error ...");
```

Beispiel: Einlesen eines Felds mit for

```
public static int[] readArray(int number) {
        // number = Anzahl der zu lesenden Elemente
    int[] result = new int[number]; // Anlegen des Felds
    for (int i = 0; i < number; i++) {
        result[i] = Terminal.askInt("nächste␣Zahl:␣");
    }
    return result;
}
```

Modifiers

- `public` sagt, wo die Funktion verwendet werden darf (↑kommt später)
- Zur Erinnerung: `static` bezeichnet eine Klassenmethode.
- Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von "`{`" und "`}`", sind nur innerhalb dieses Blocks sichtbar, d.h. benutzbar (lokale Variablen).

Beispiel 4: Kopieren von Arrays

```
static float[] copy(float[] a) {
   float[] b = new float[a.length];

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

- Anwendung: „`float[] b = copy(a);`“
- „`float[] c = a;`“ kopiert das Array nicht!
  (Warum? → s. Referenzen)
- Schnellere Variante:
  `java.lang.System.arraycopy(...)`

Beispiel 4: Kopieren von Arrays

```
public static void main(String[] args) {
    float[] a = {2.3, 1.2, 4.8, 5.46, 1.23};

    float[] b = copy(a);   // Kopie
    float[] c = a;         // Alias

    b[2] = 0.0;
    System.out.println(a[2]);

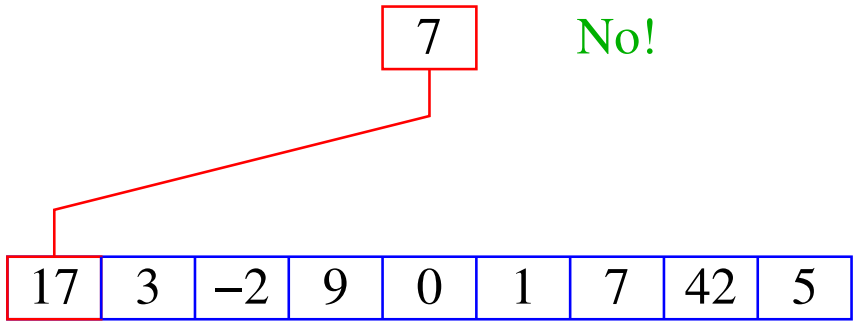
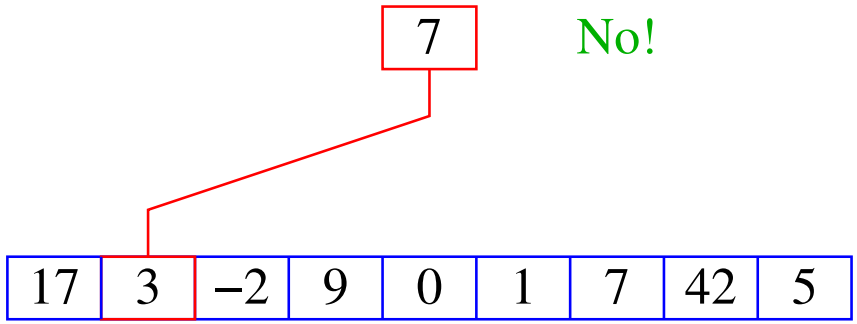
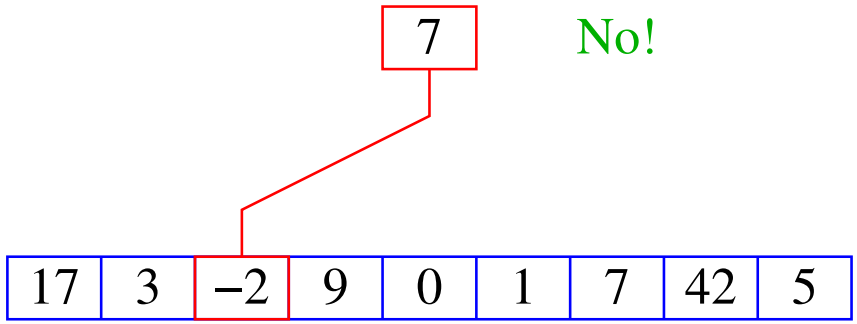
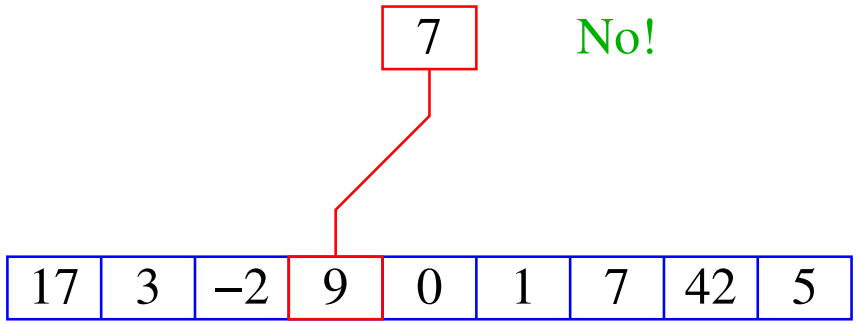
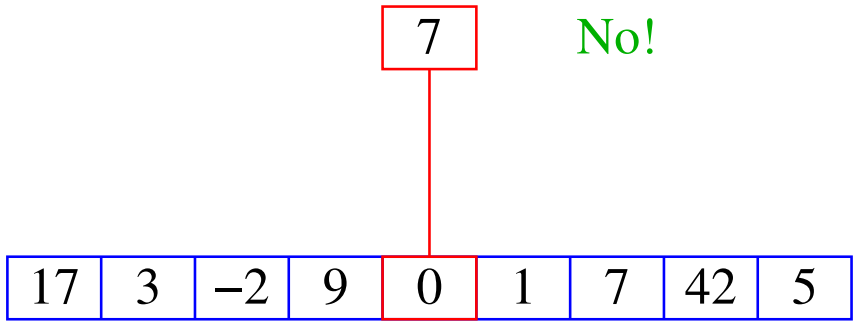
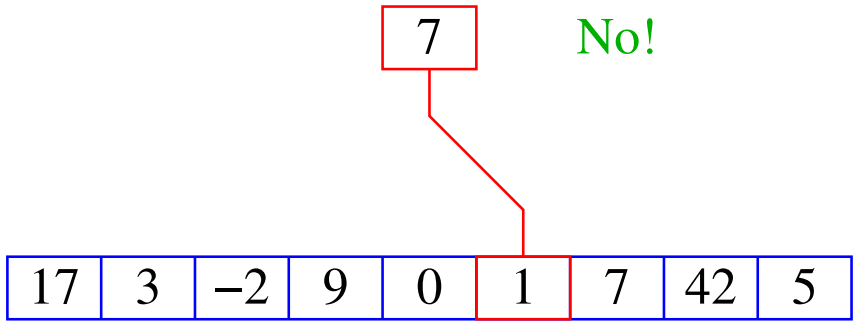
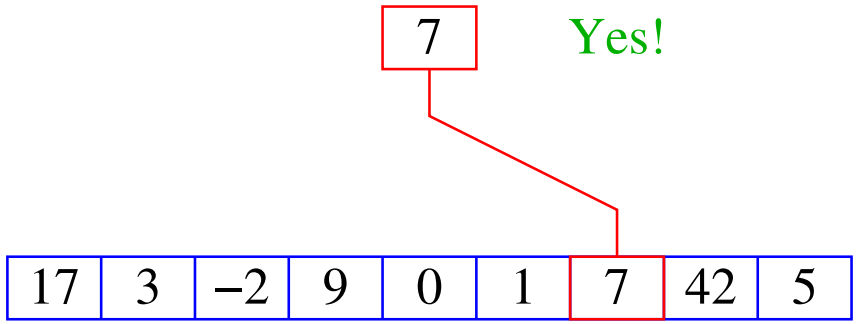
    c[2] = 1.0;
    System.out.println(a[2]);
}
```

Beispiel 4: Kopieren von Arrays

```
public static void main(String[] args) {
    float[] a = {2.3, 1.2, 4.8, 5.46, 1.23};

    float[] b = copy(a);   // Kopie
    float[] c = a;         // Alias

    b[2] = 0.0;
    System.out.println(a[2]); // Ausgabe: 4.8

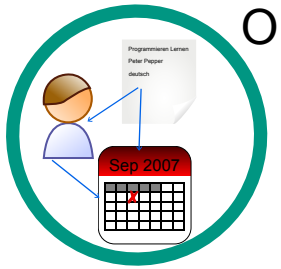
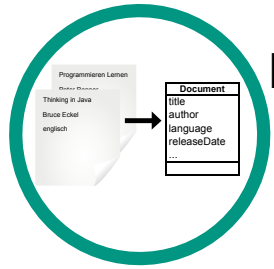
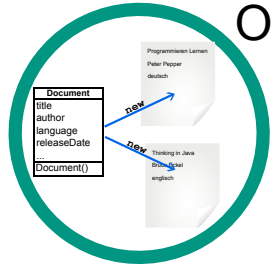
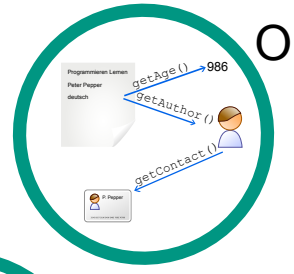
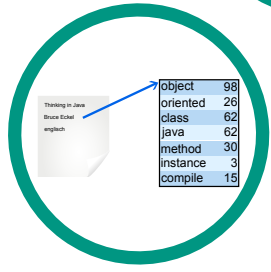
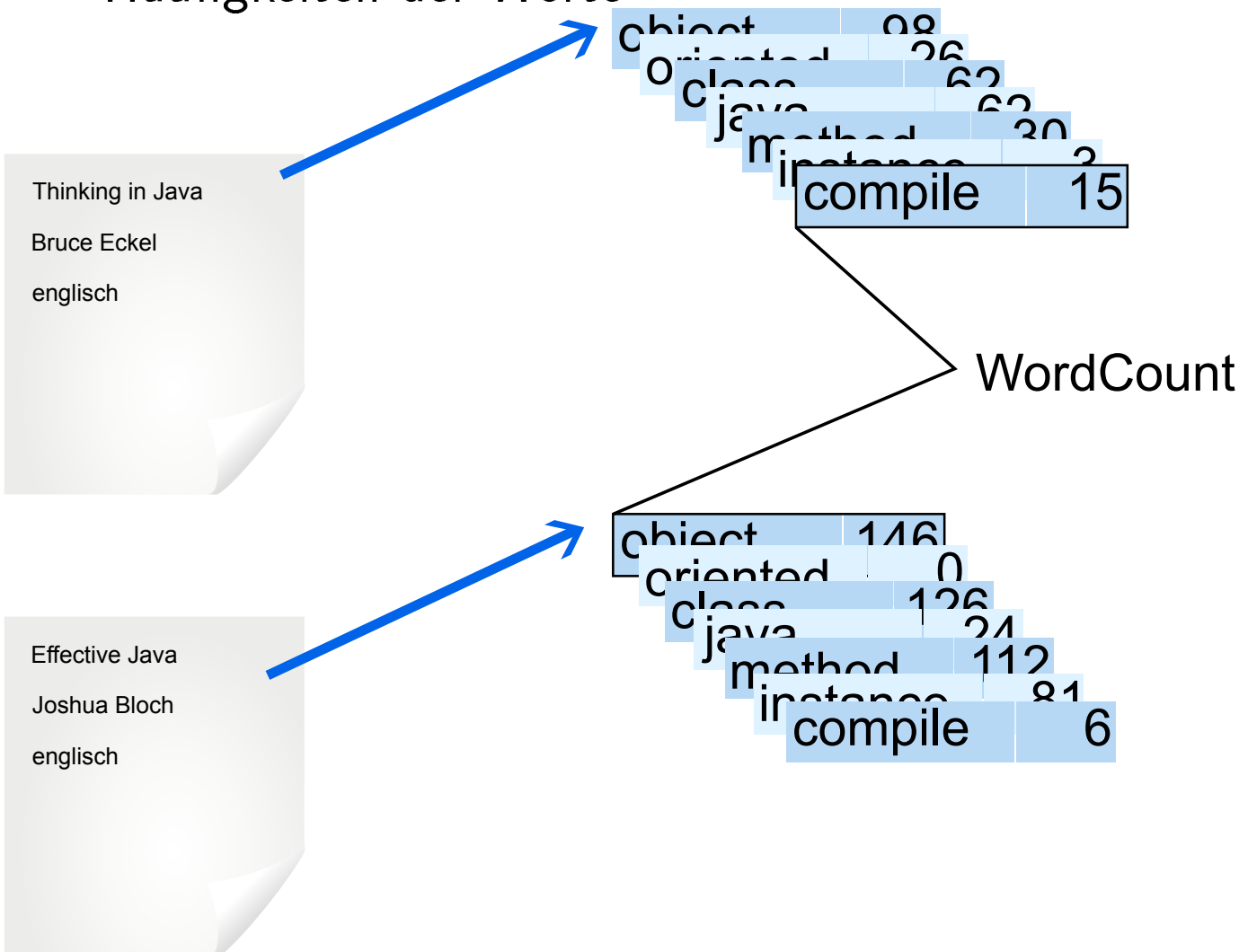
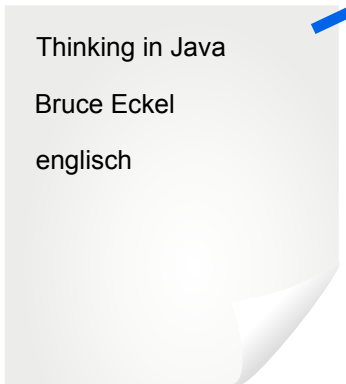
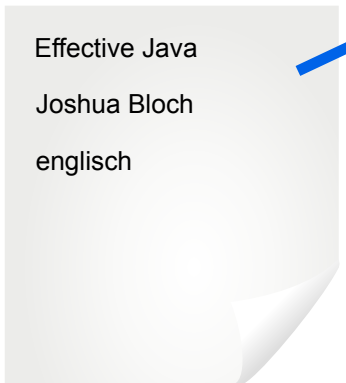
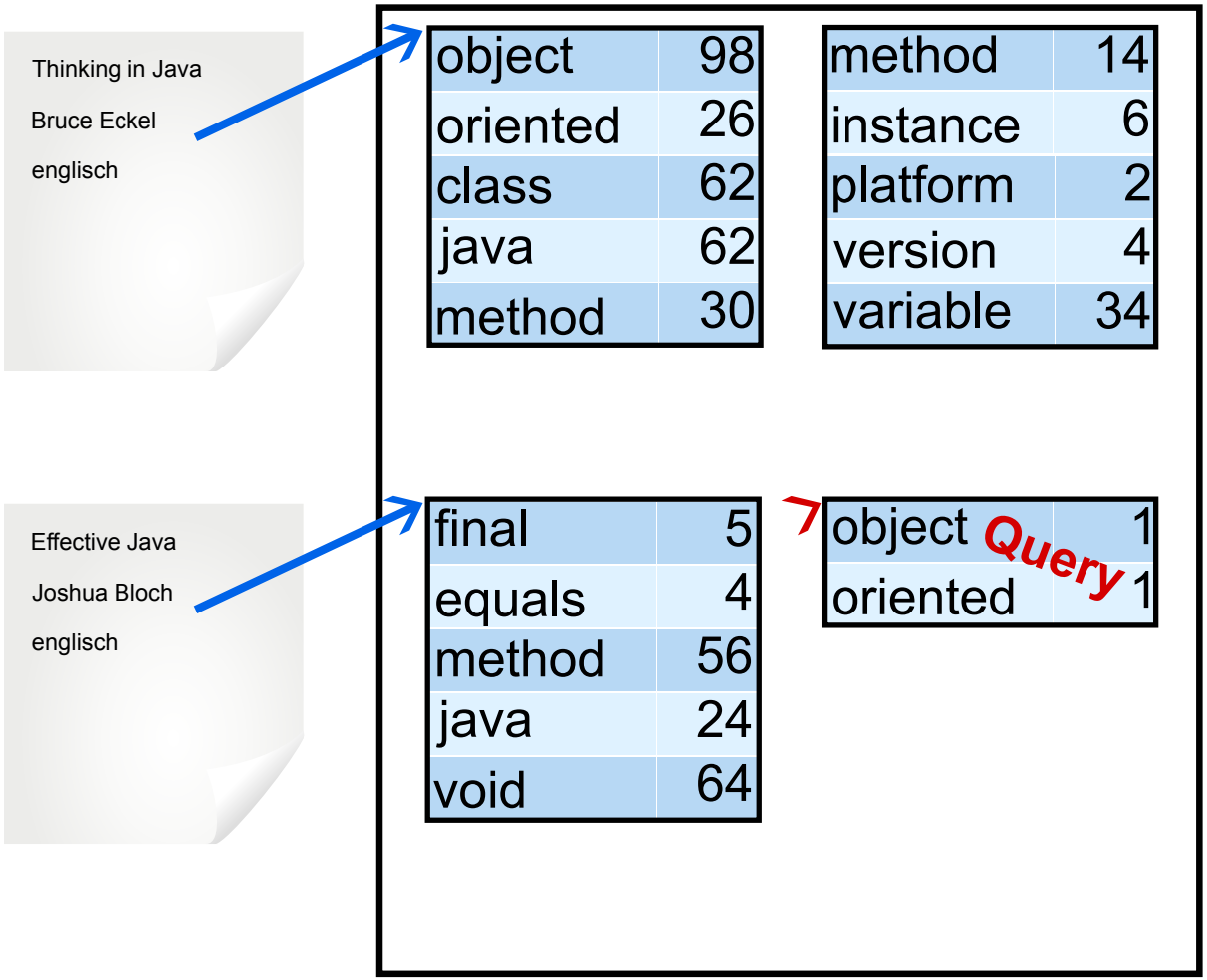
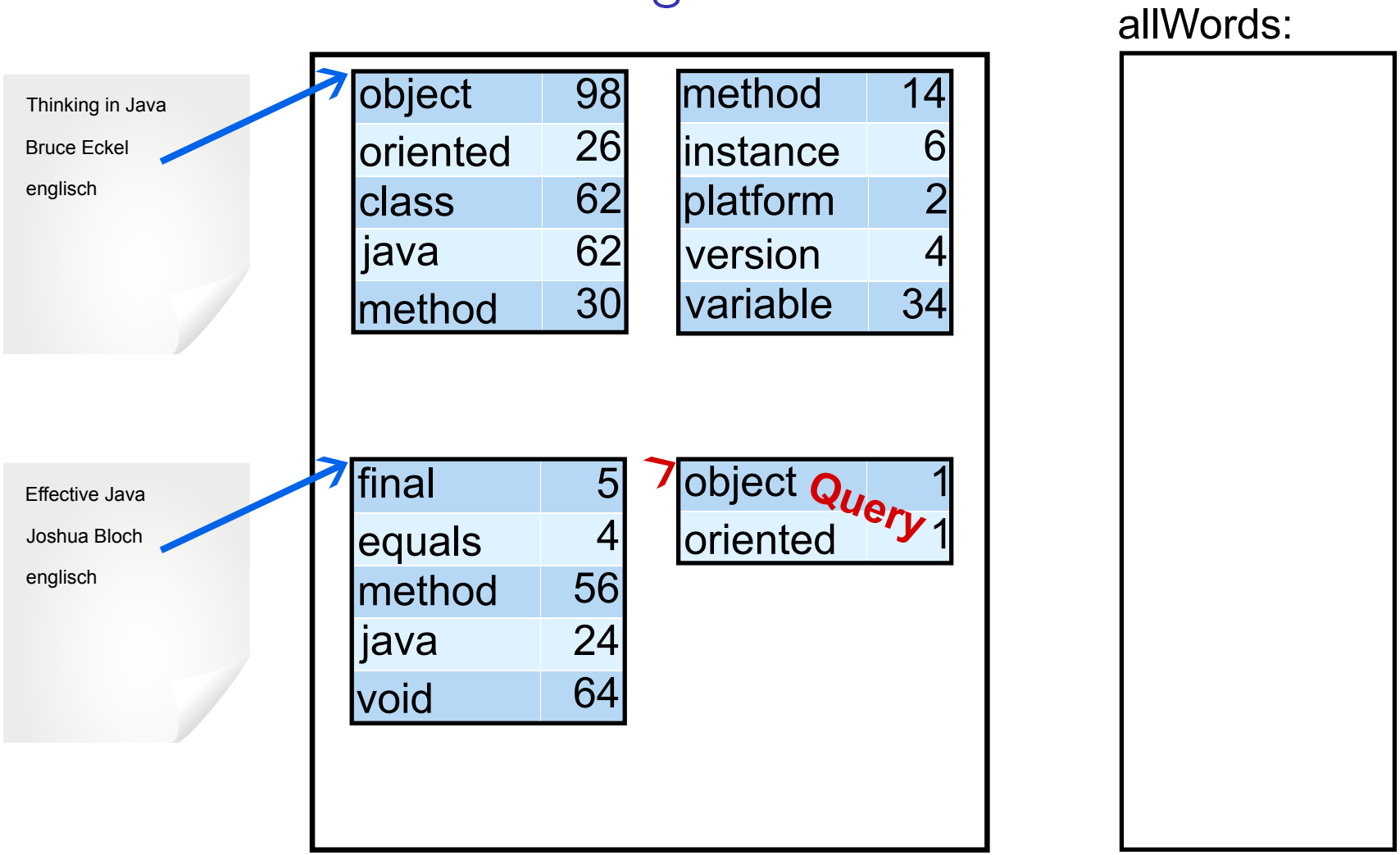
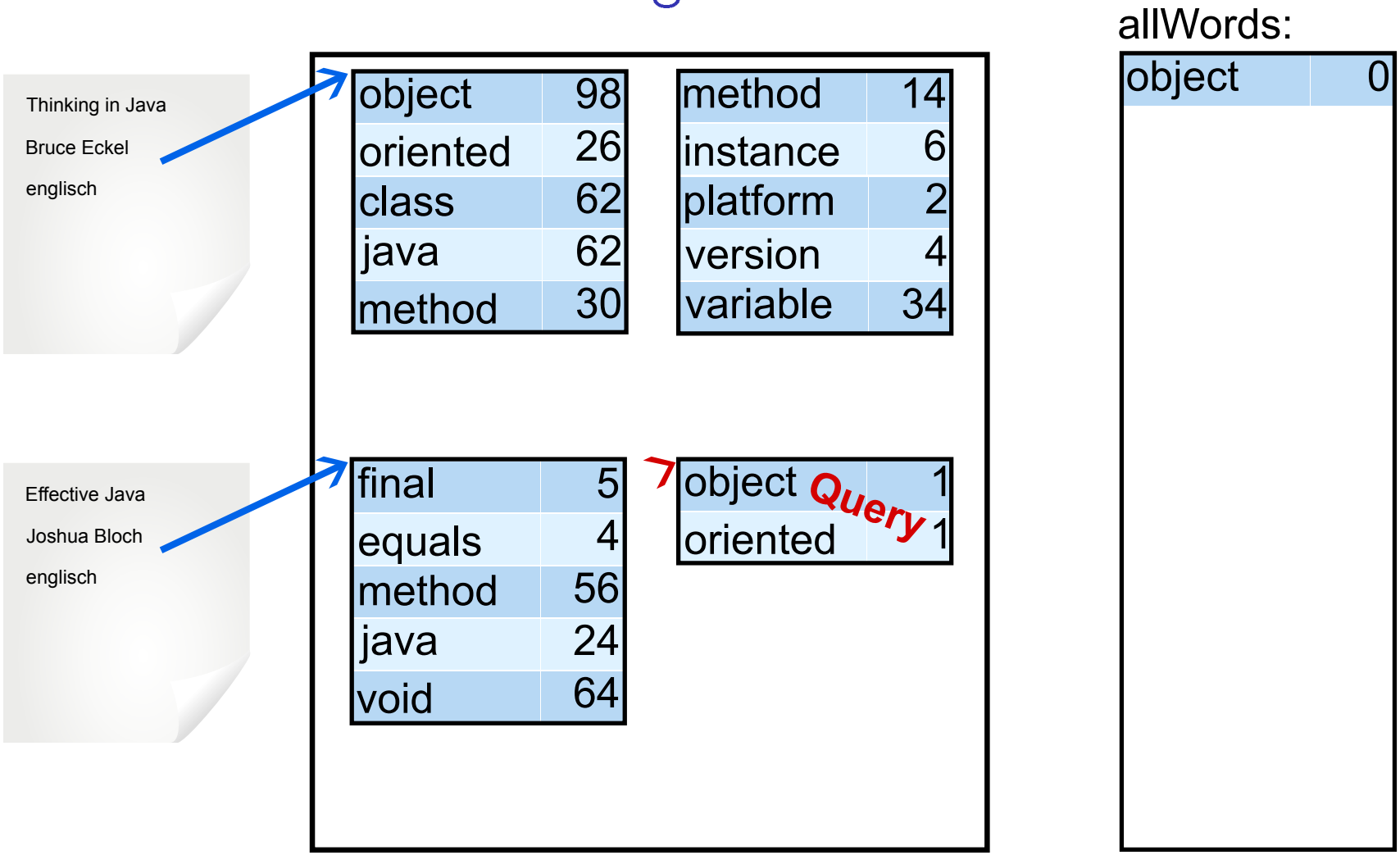
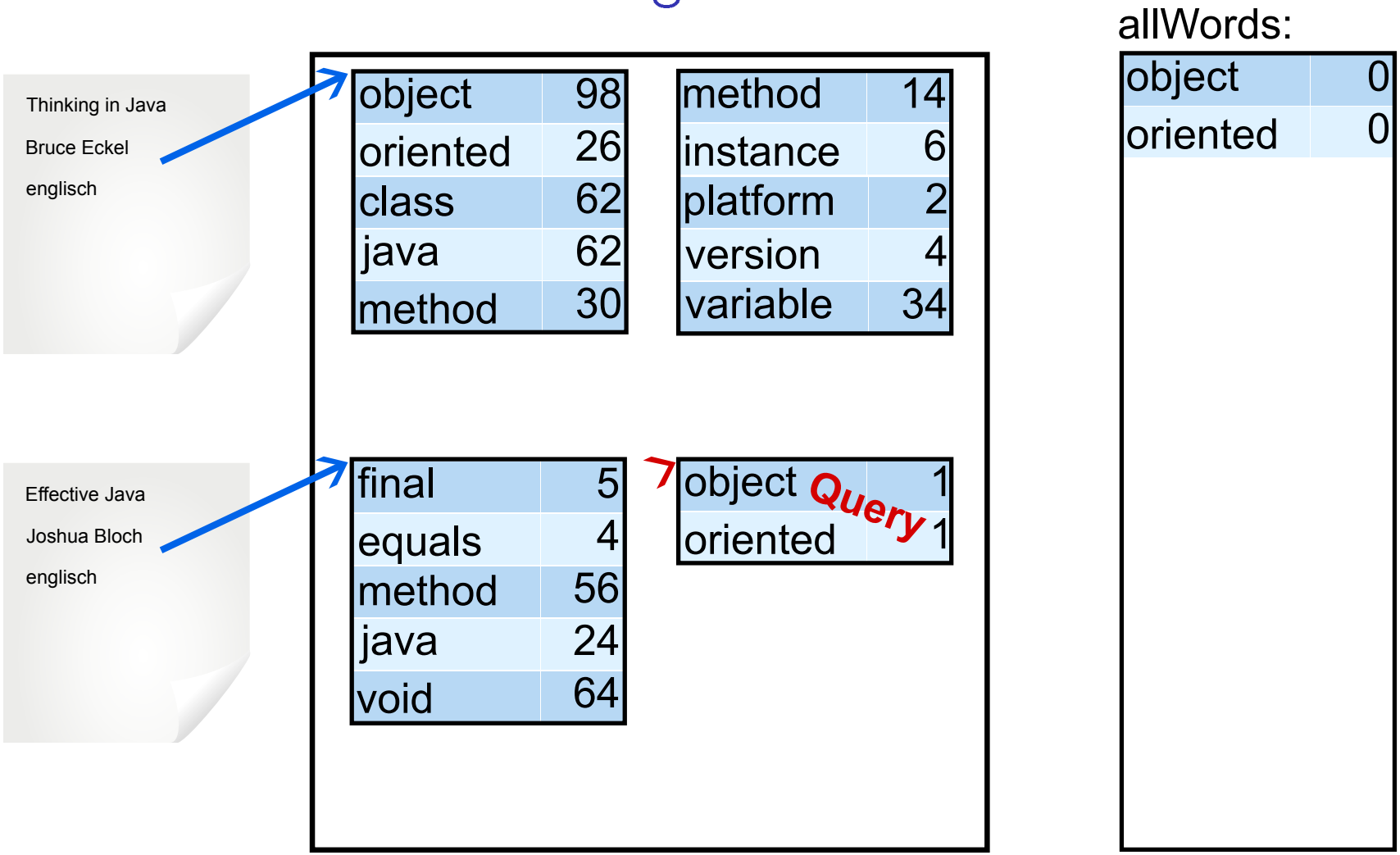
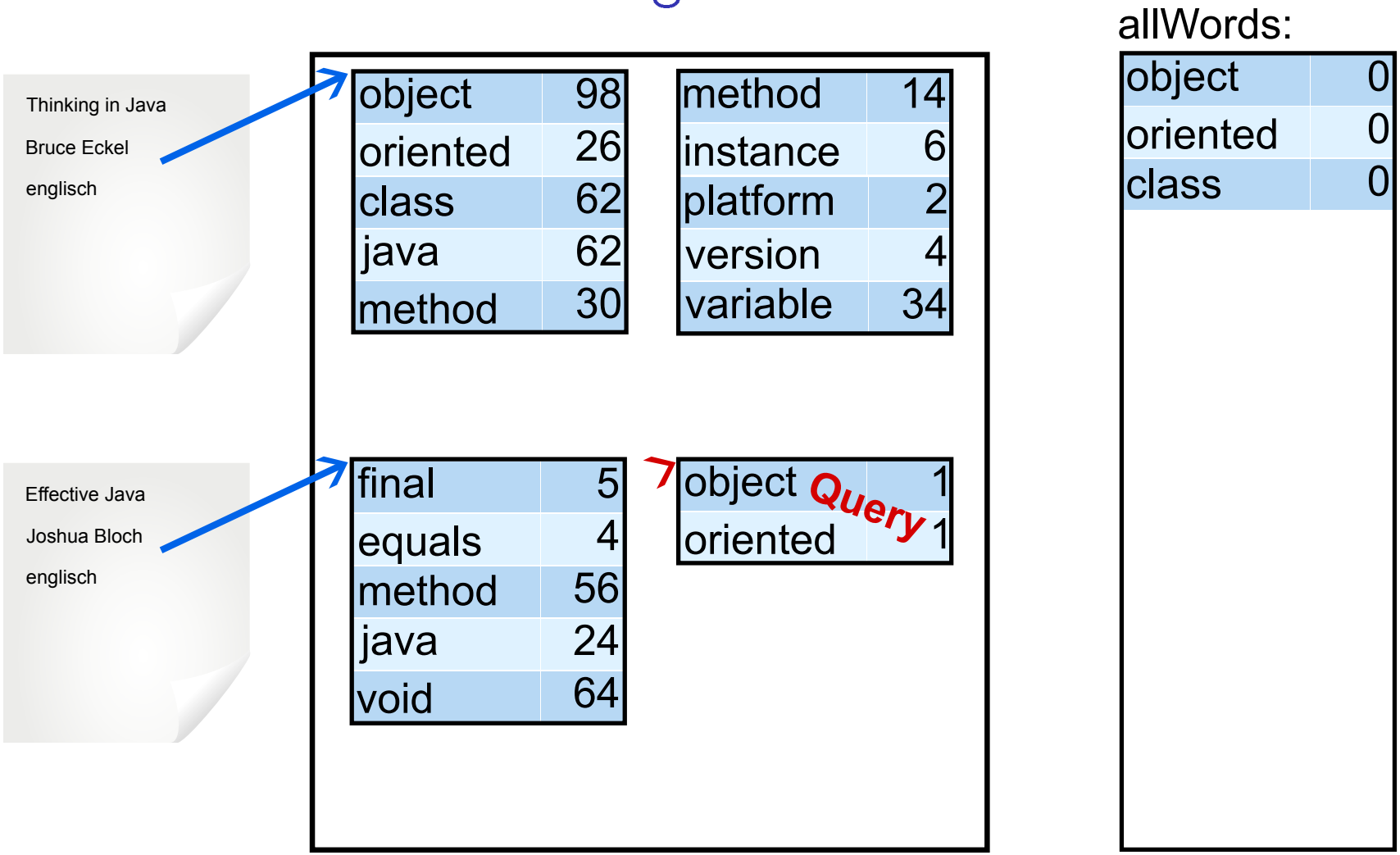
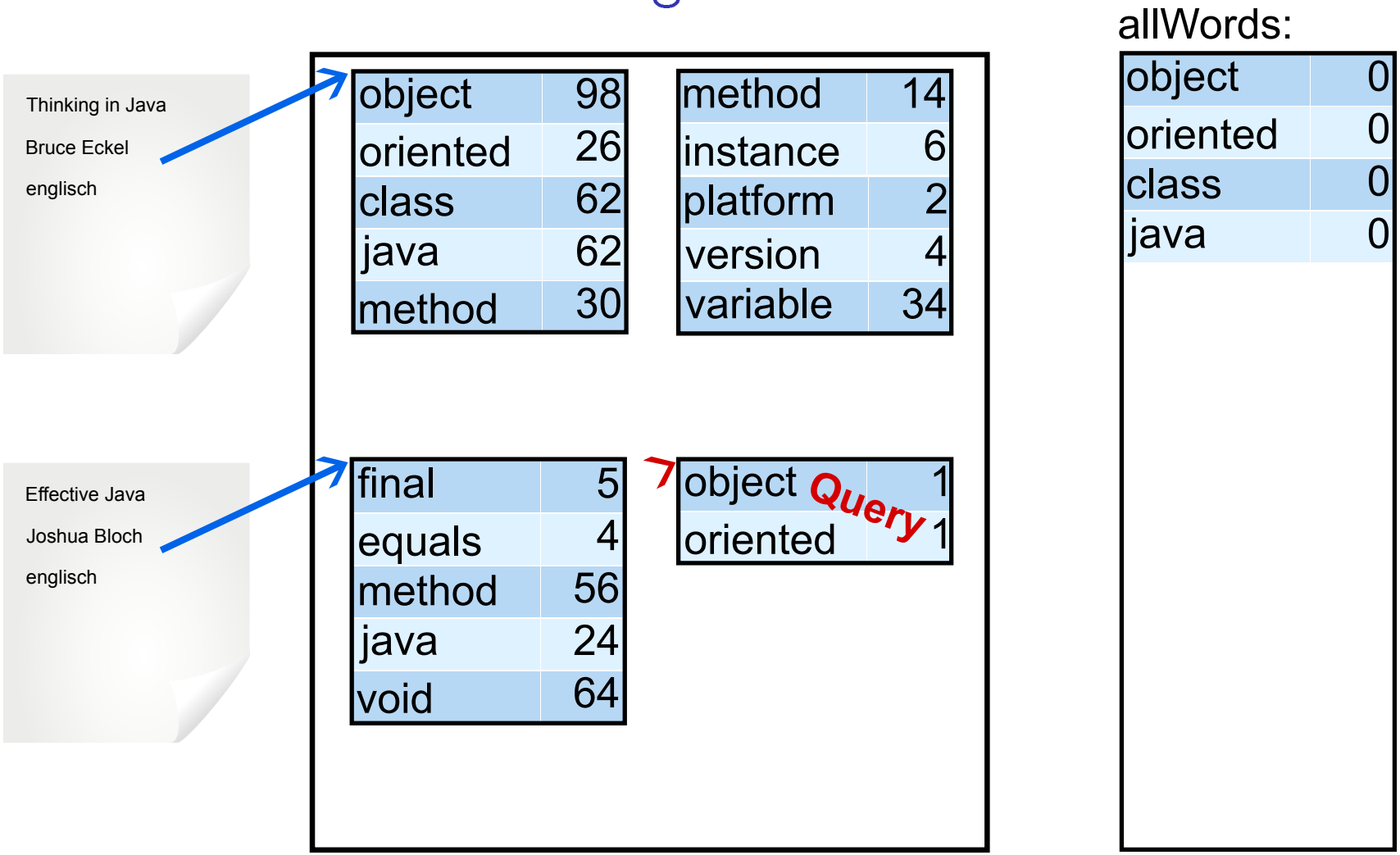
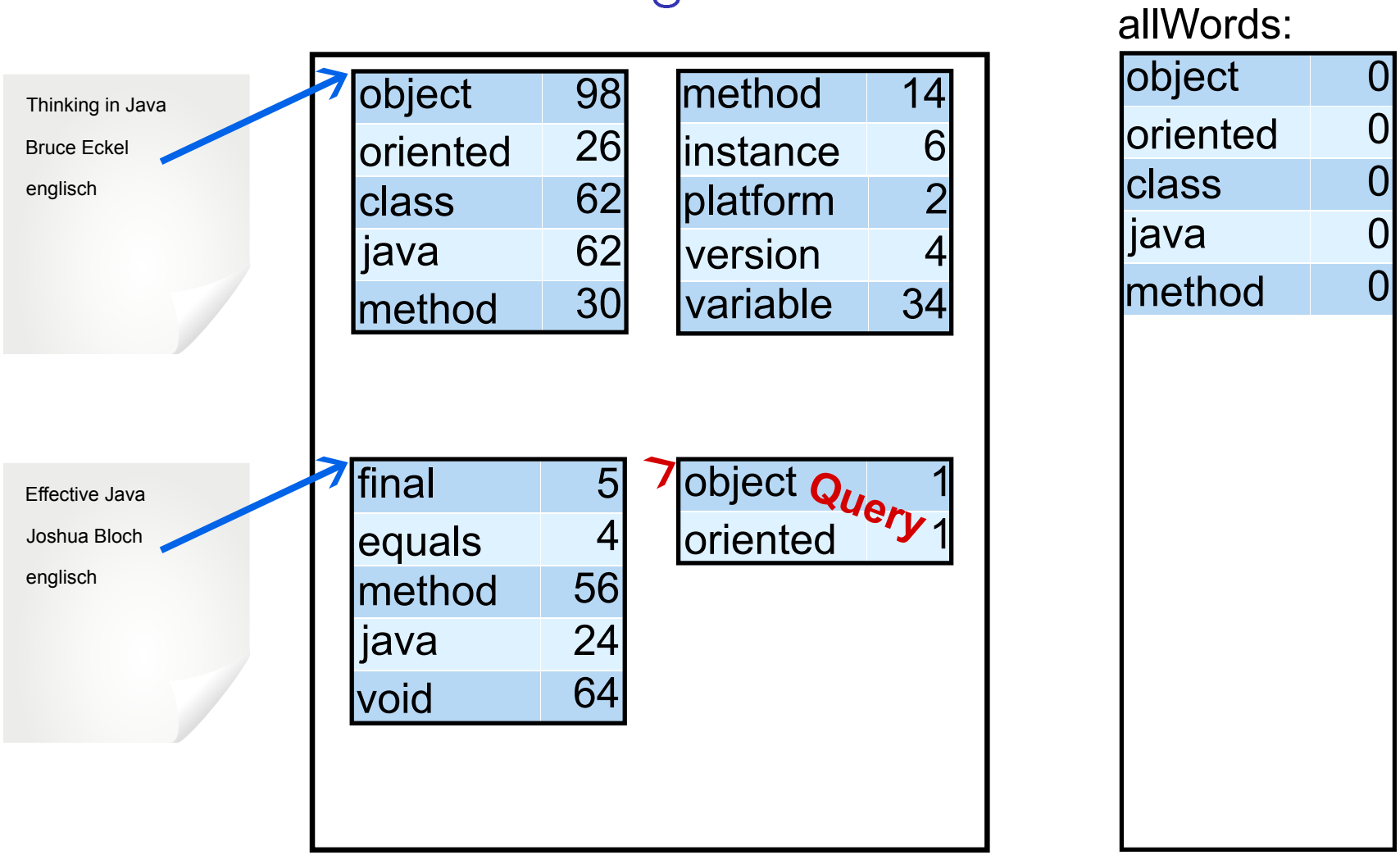
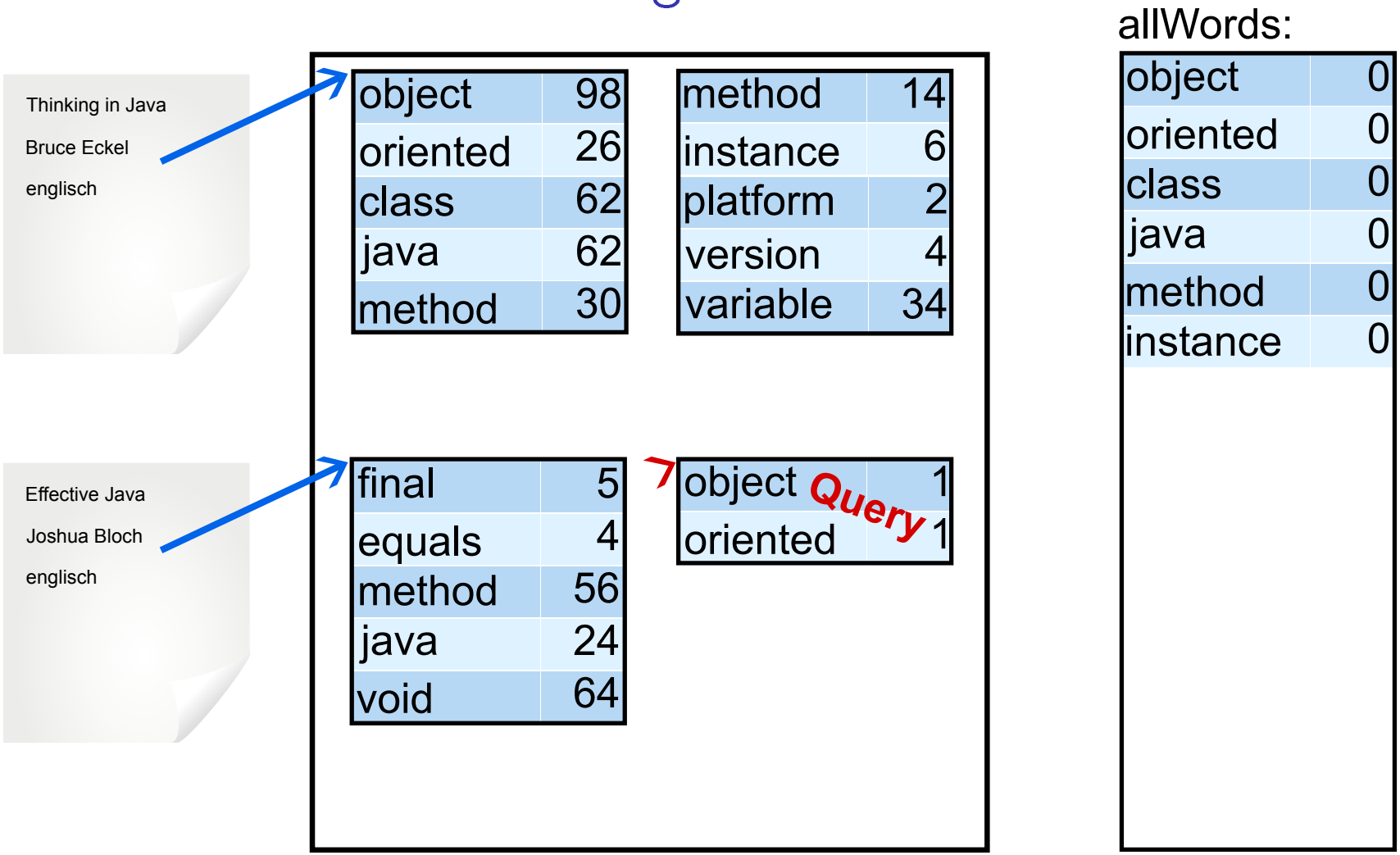
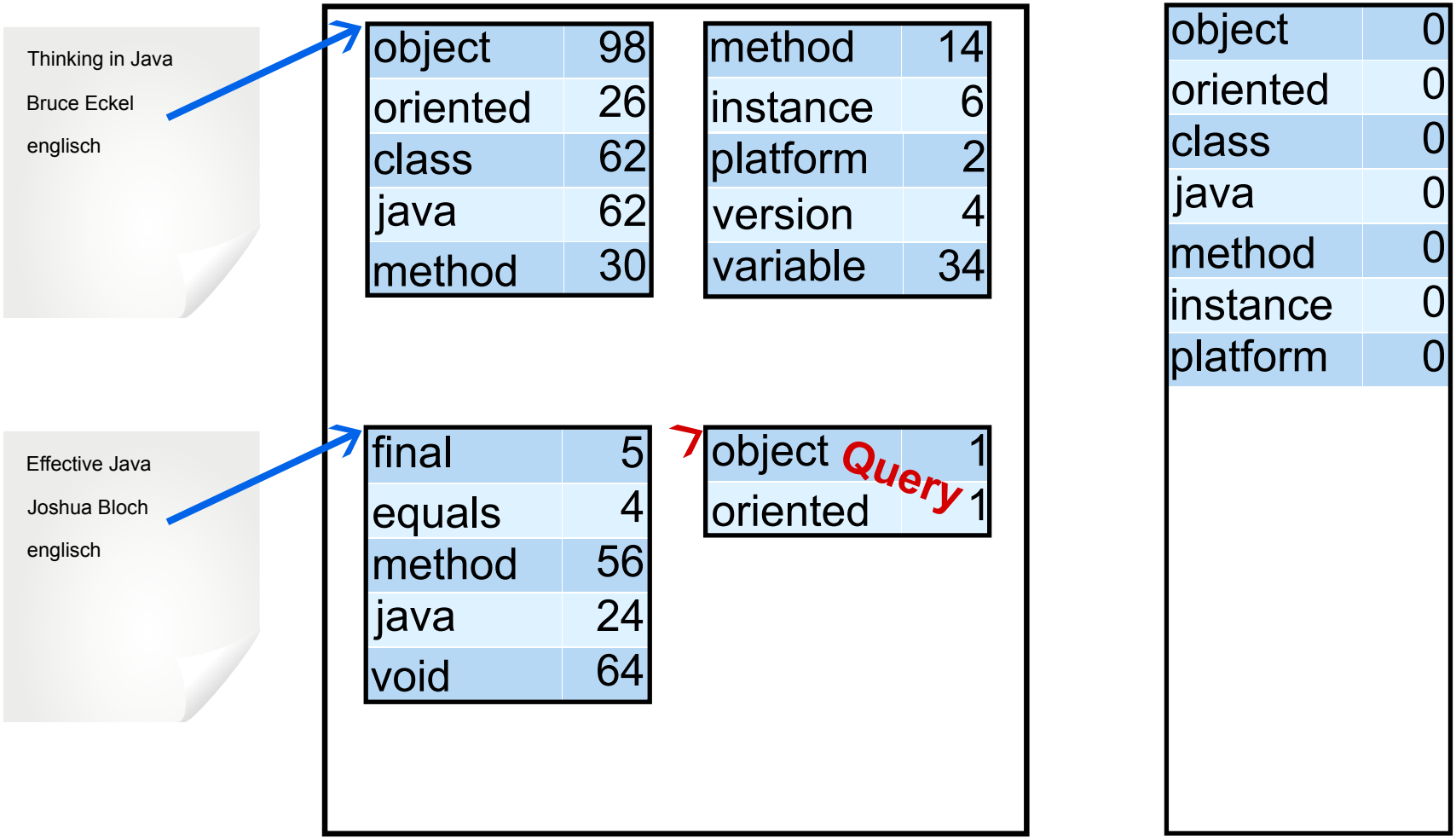
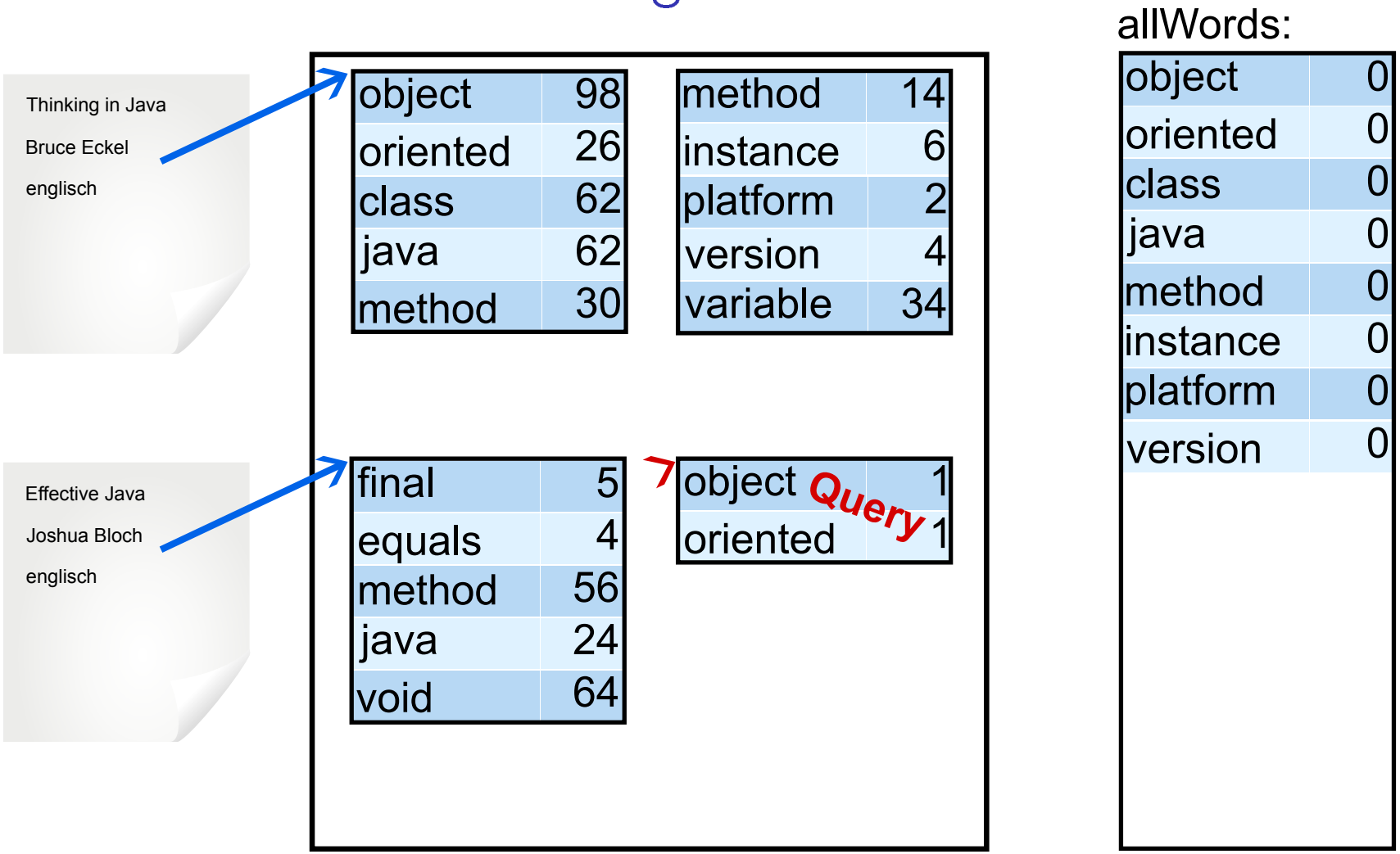
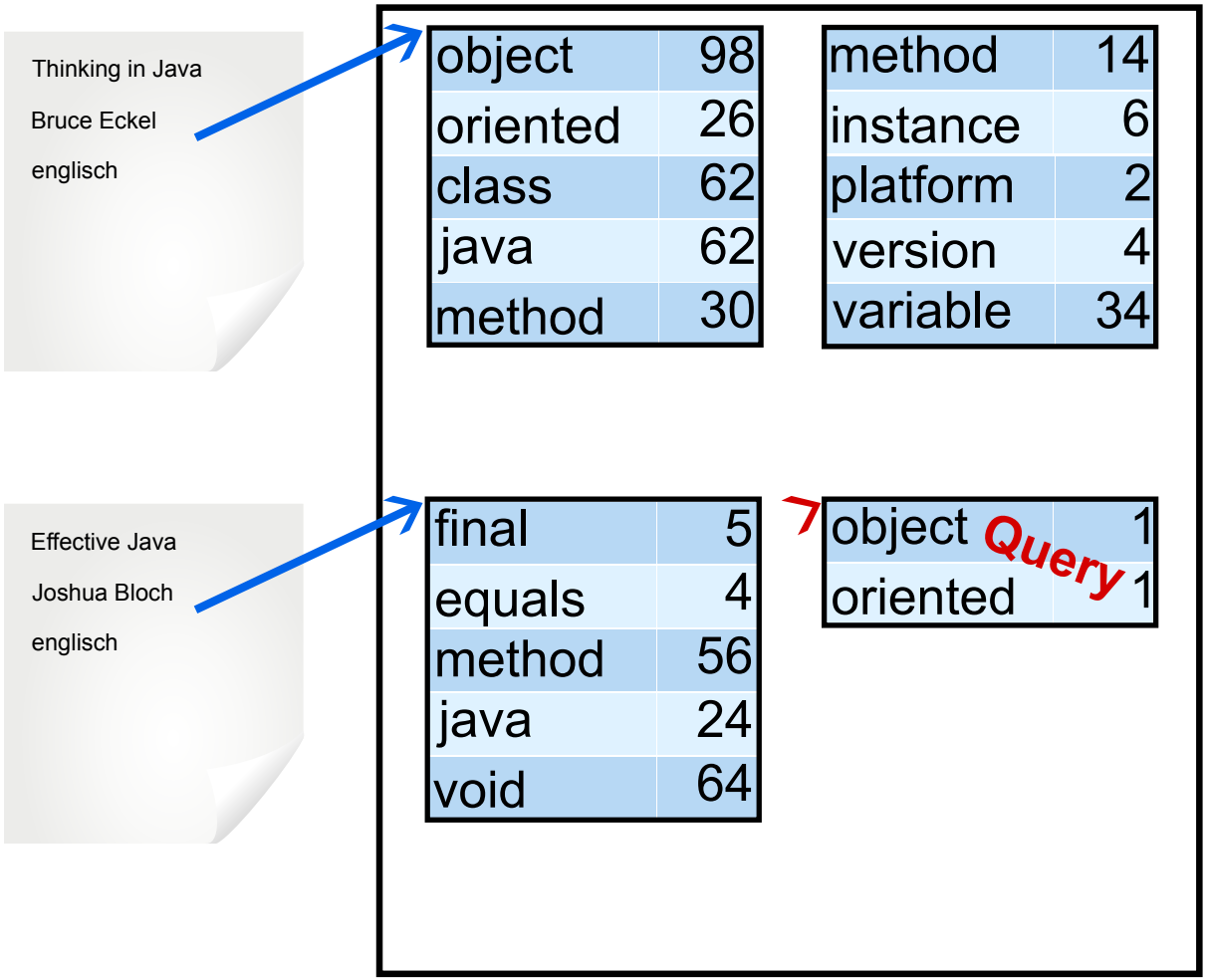
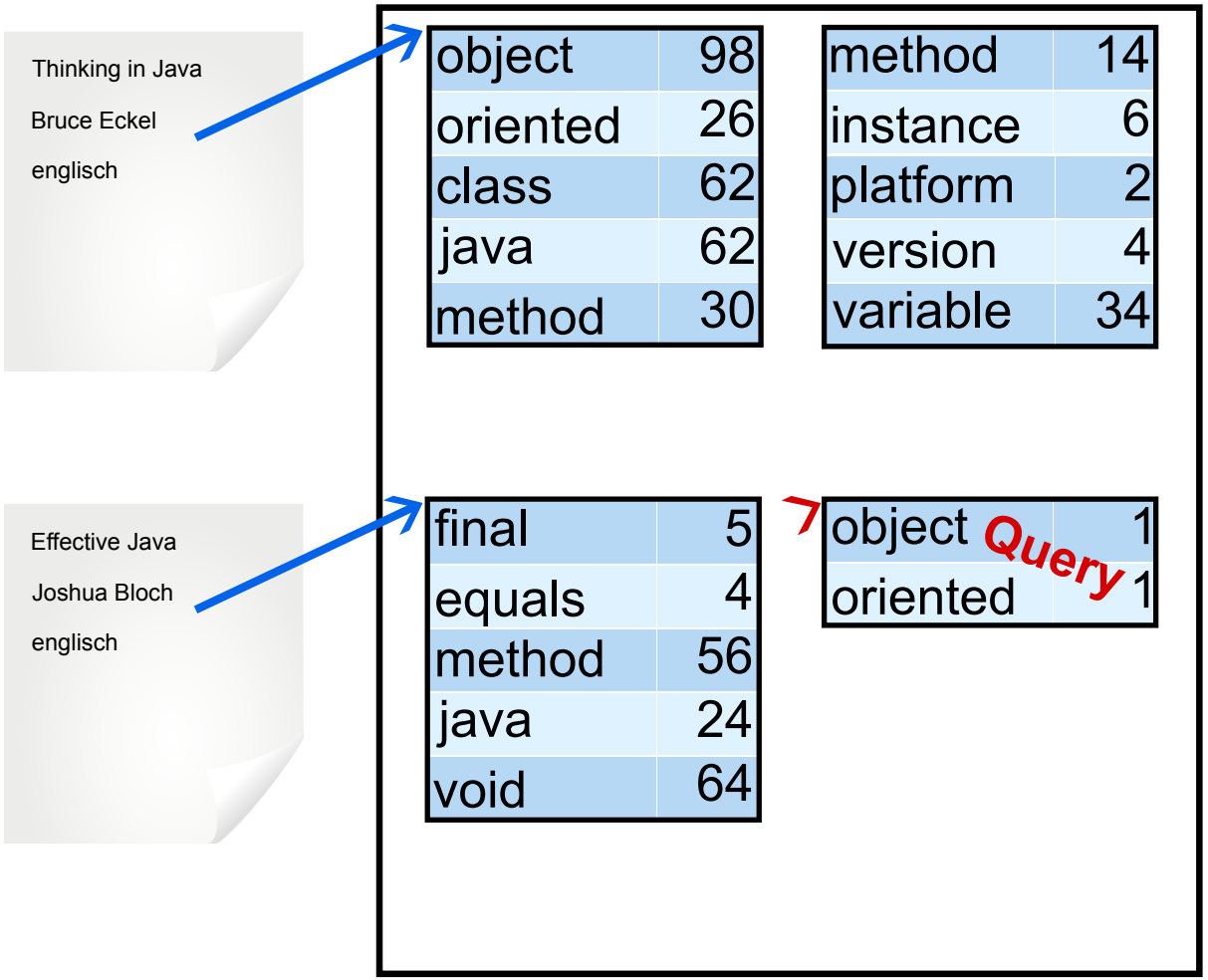
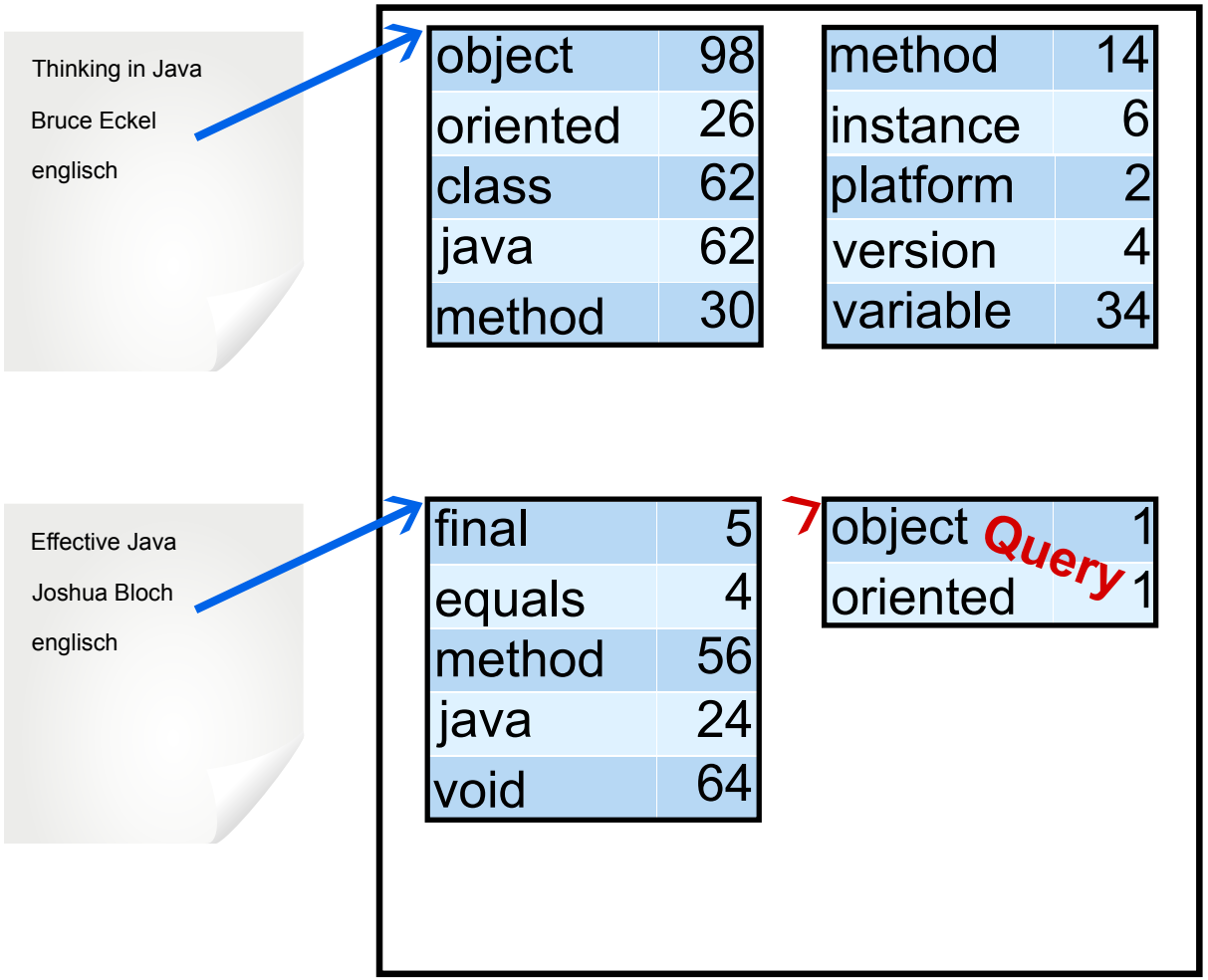
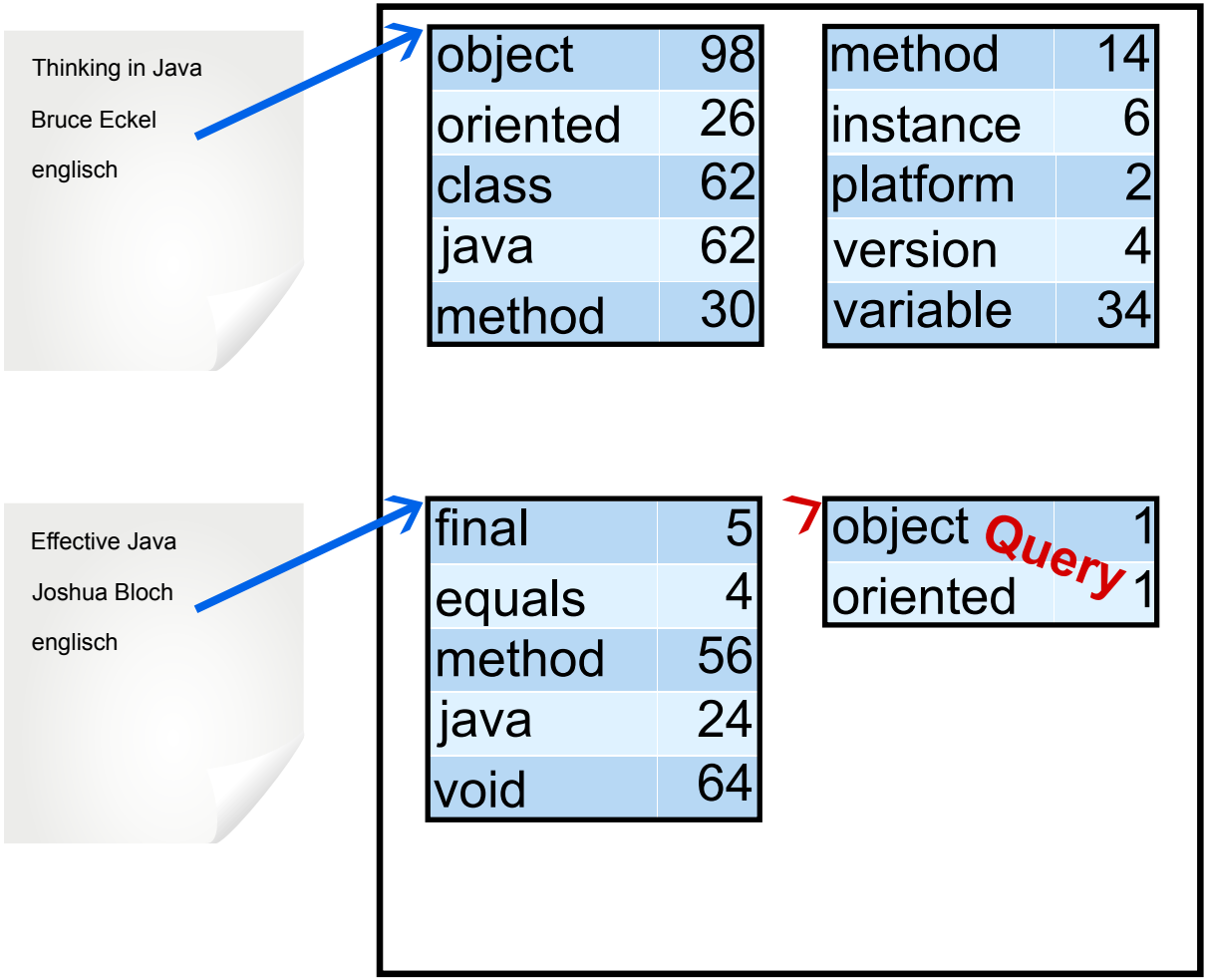
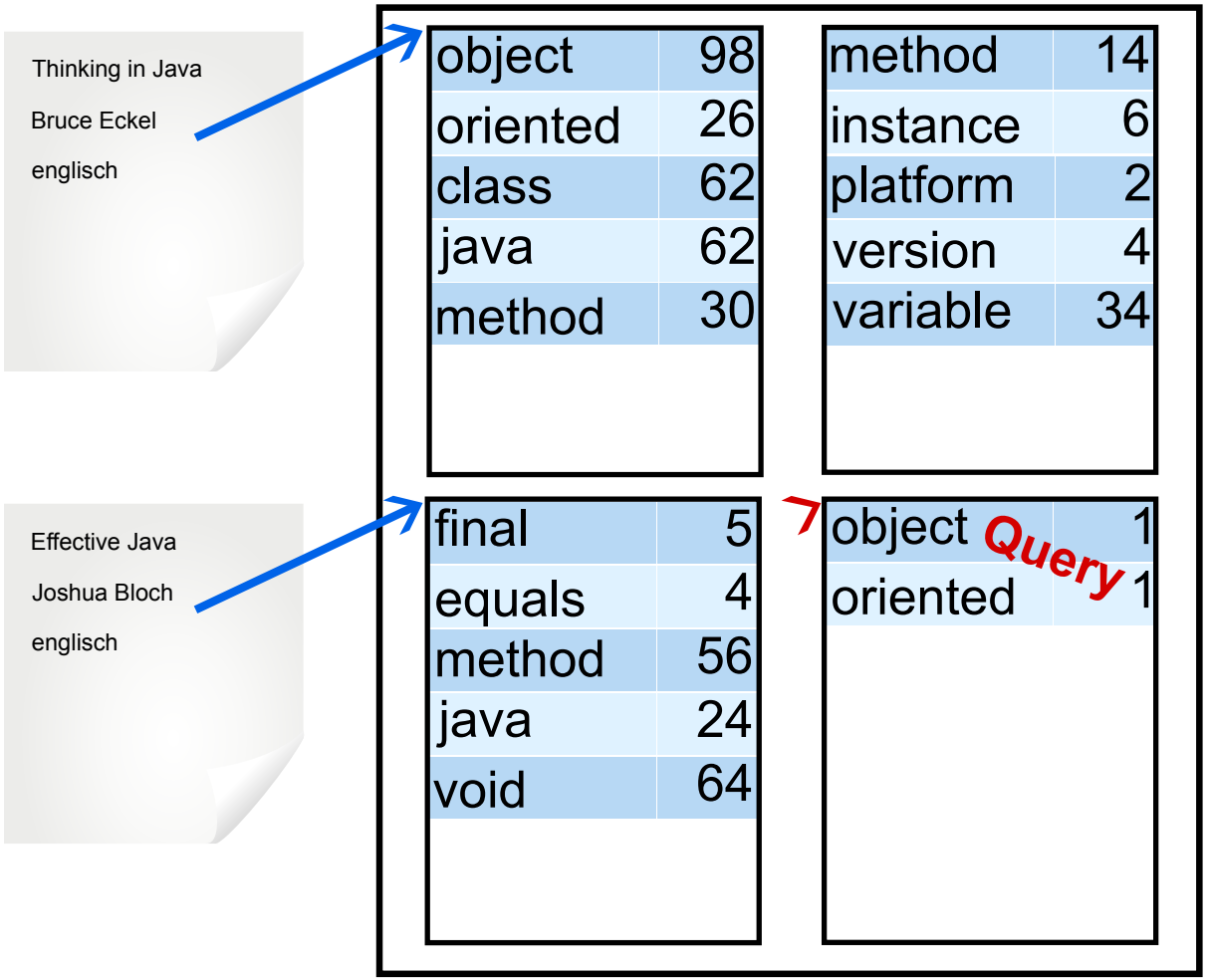
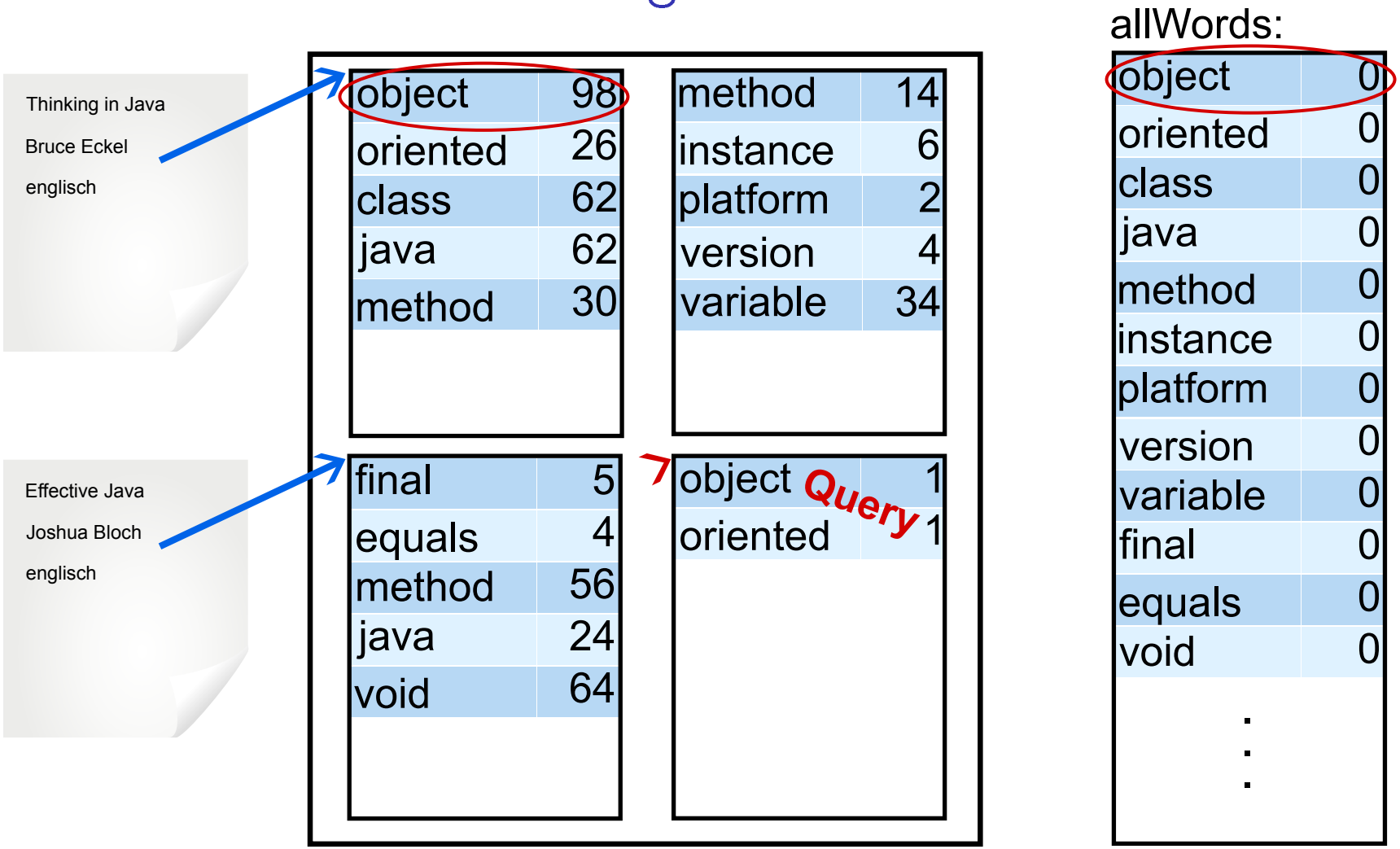
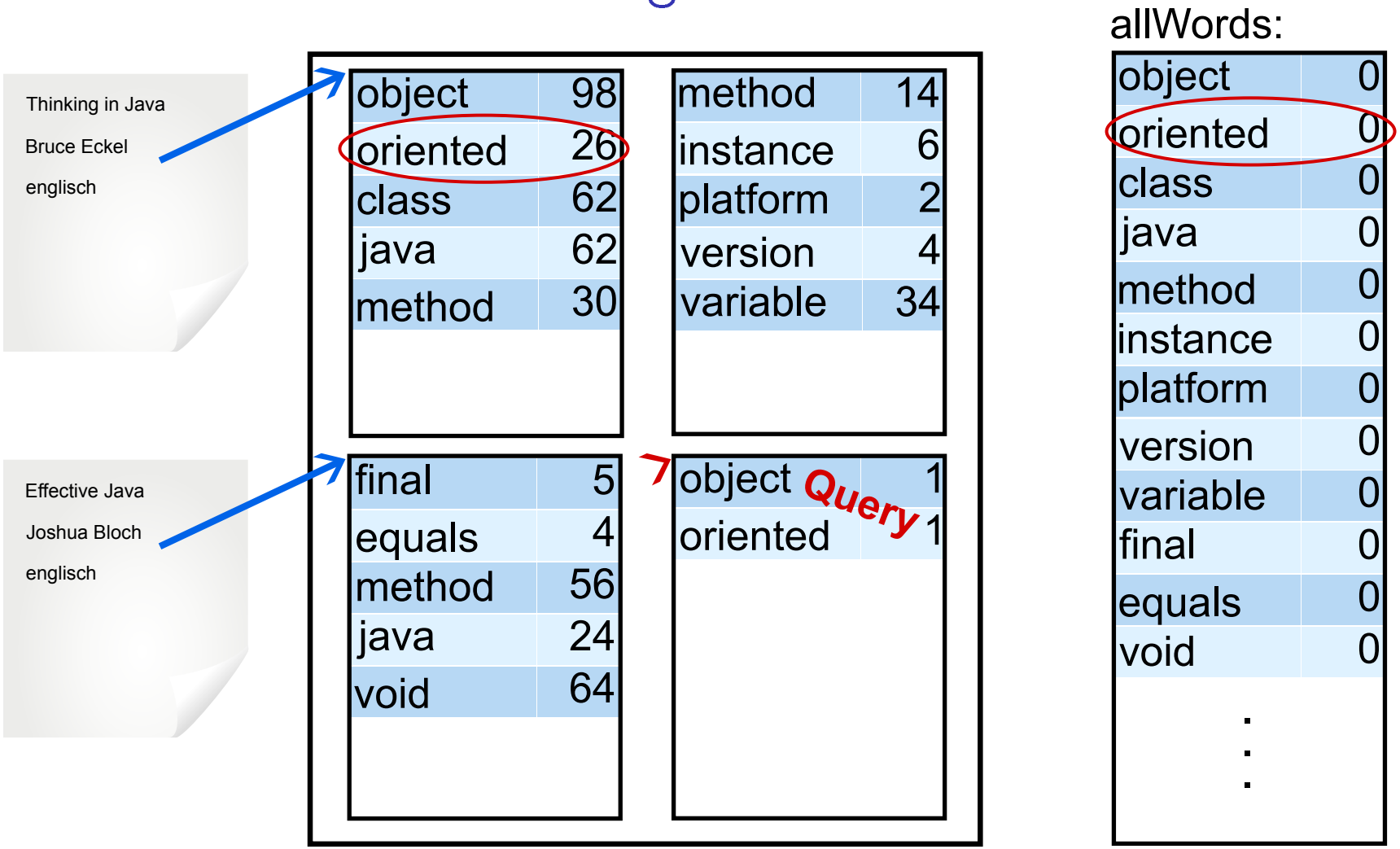
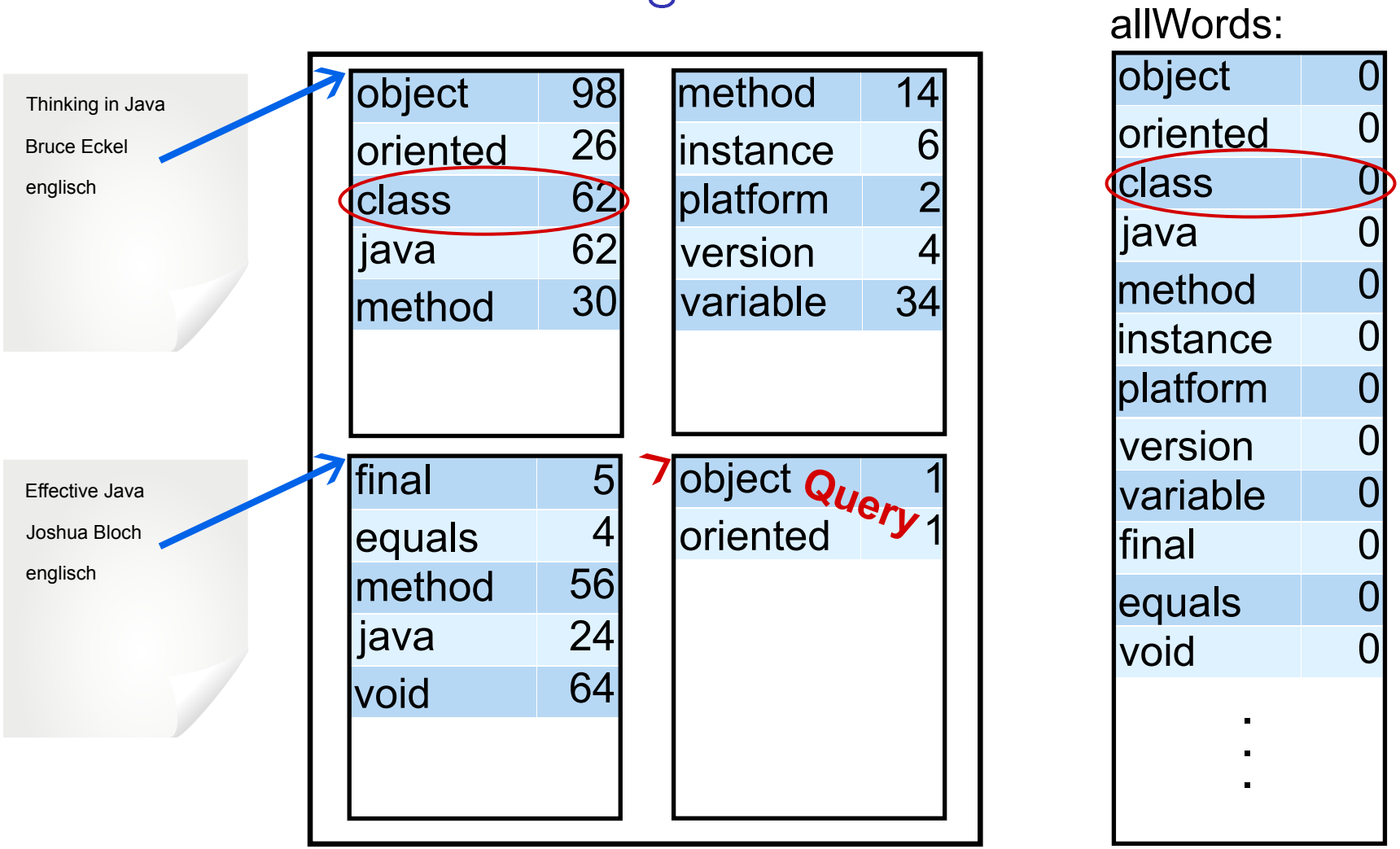
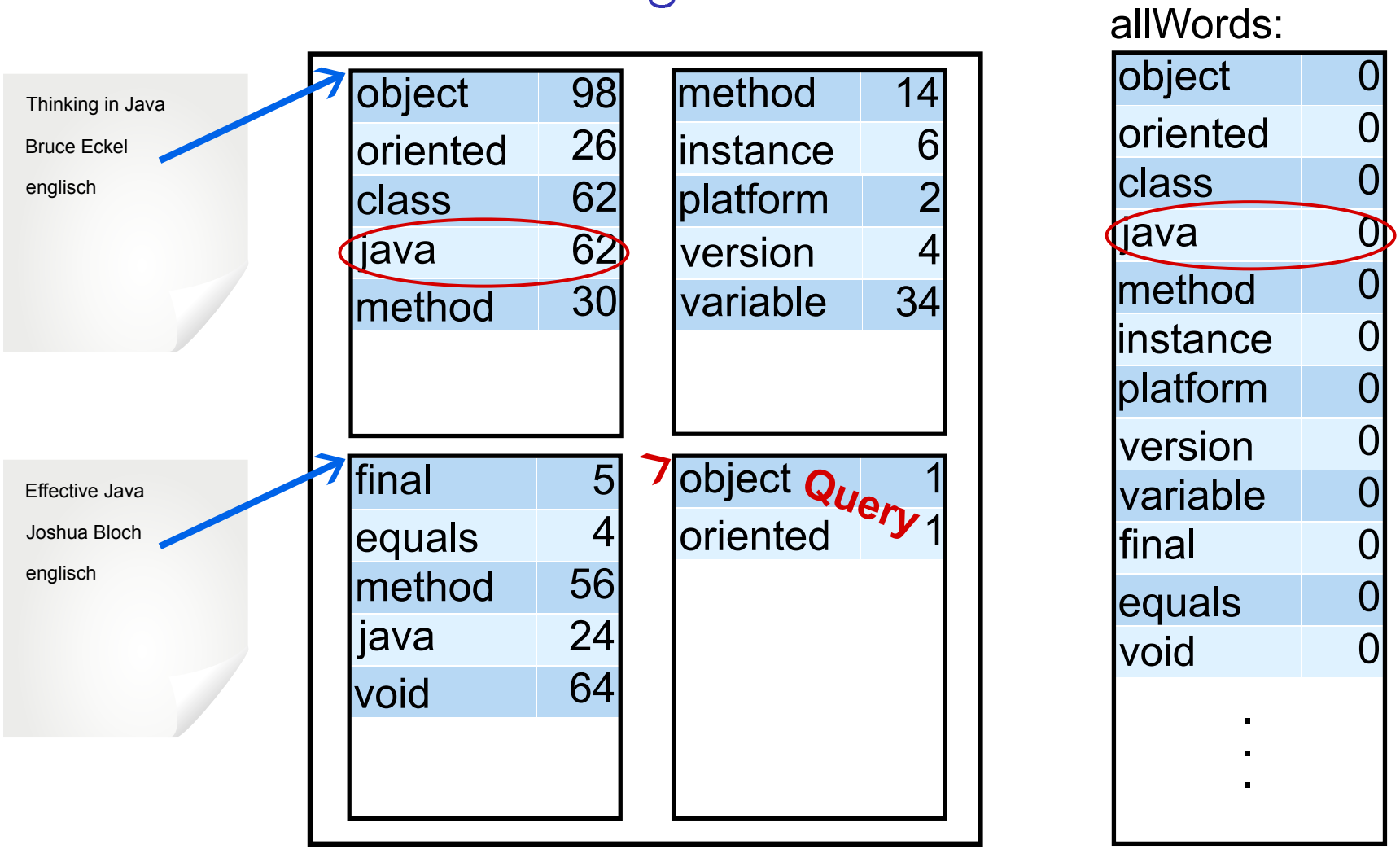
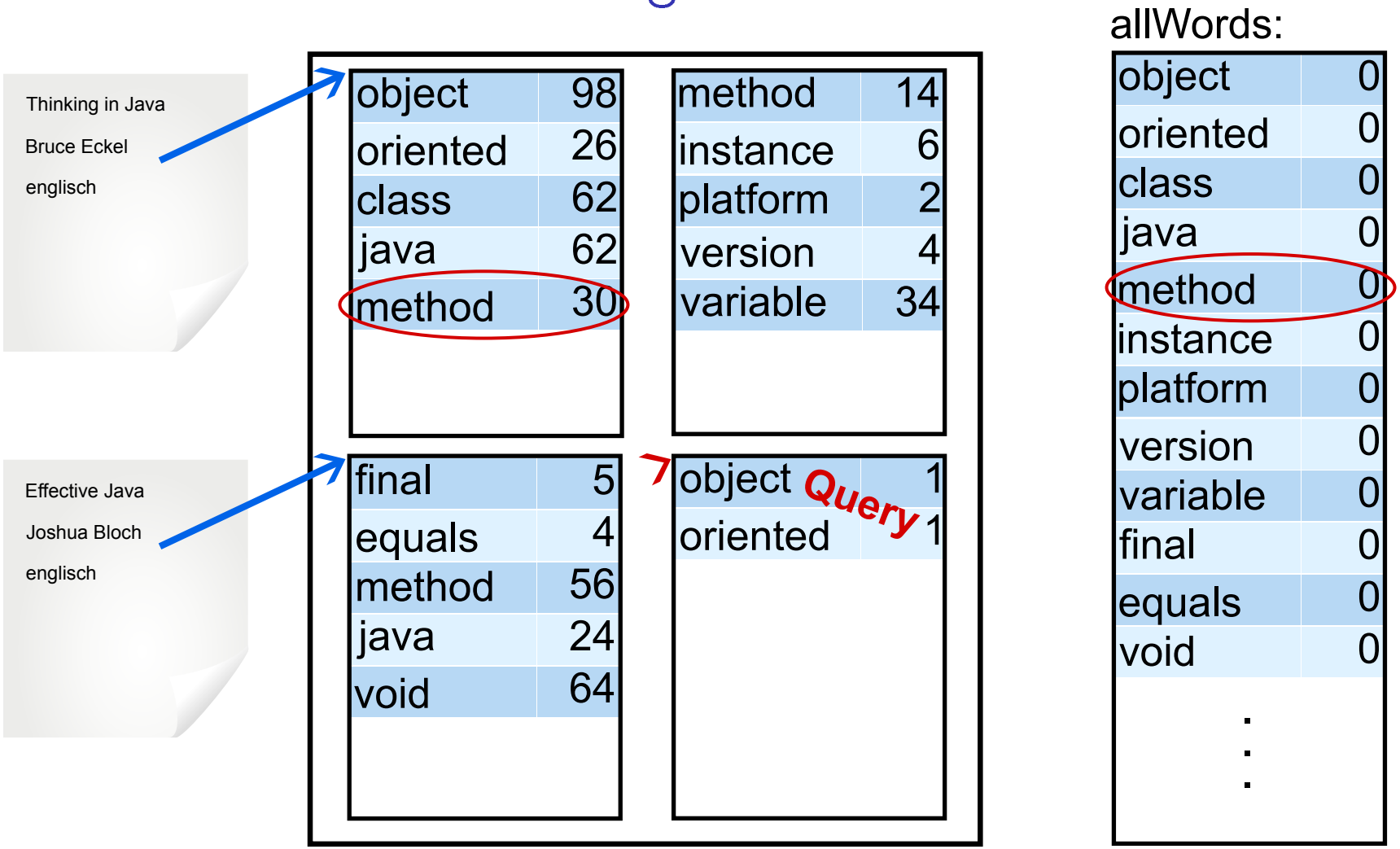
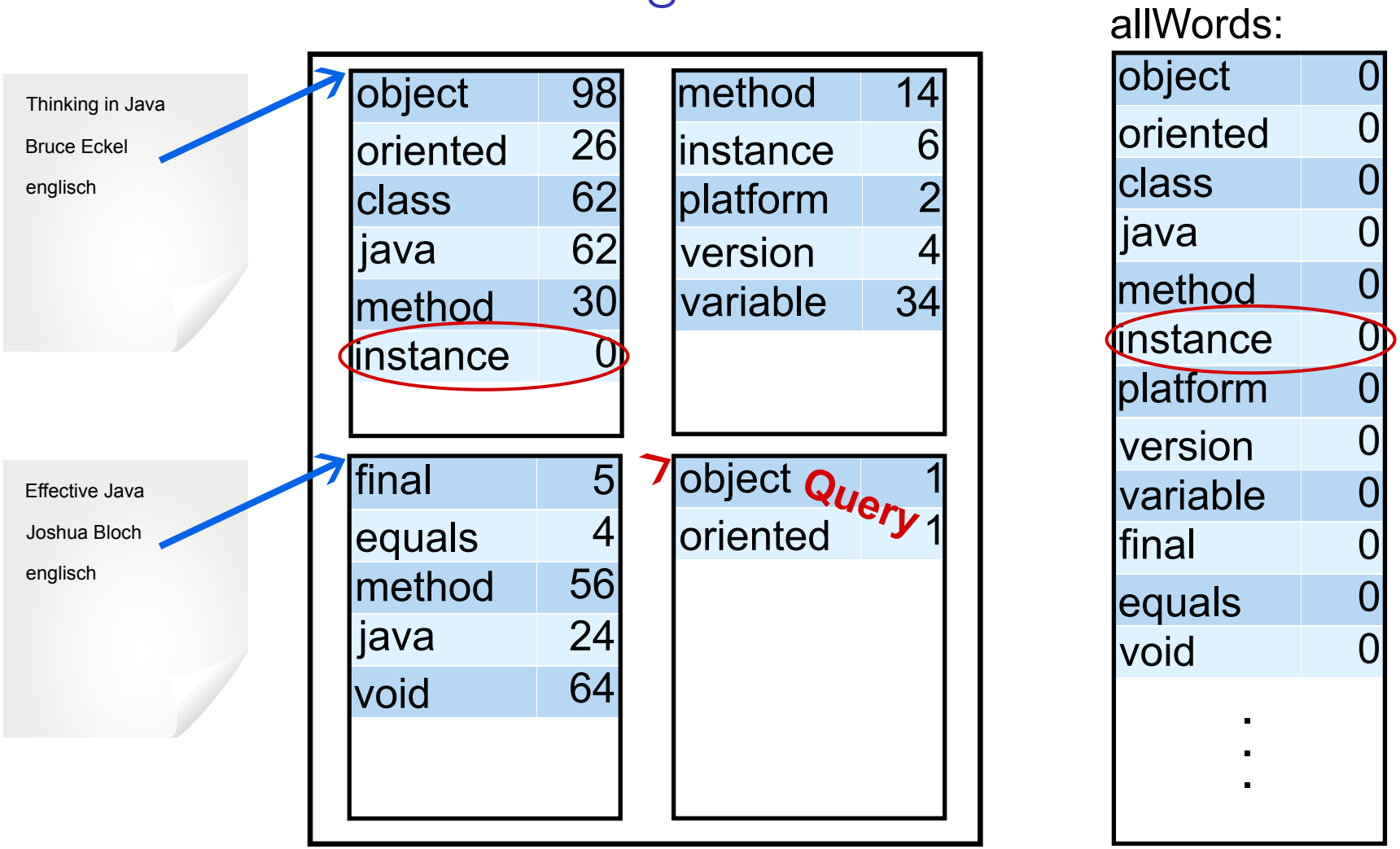
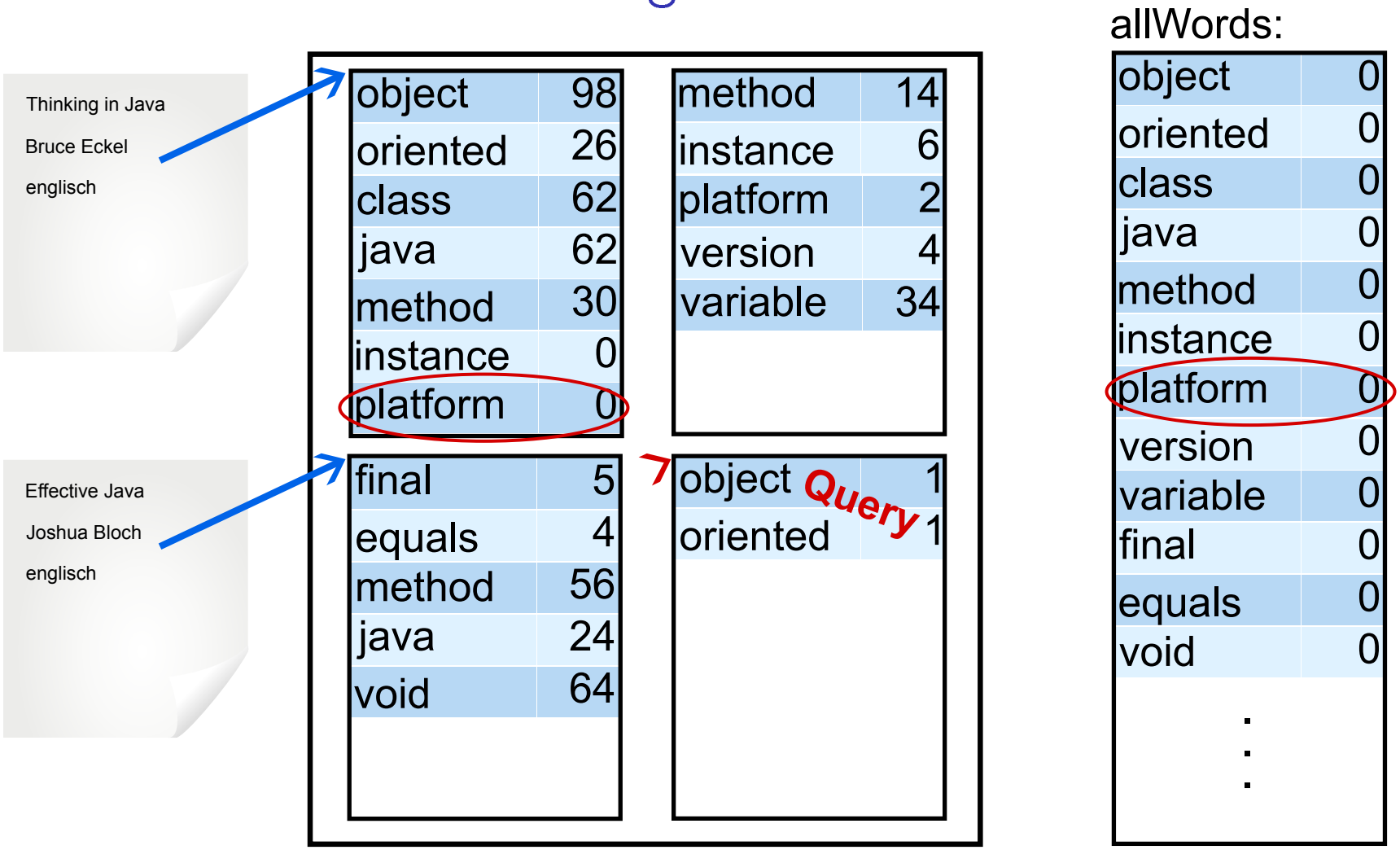
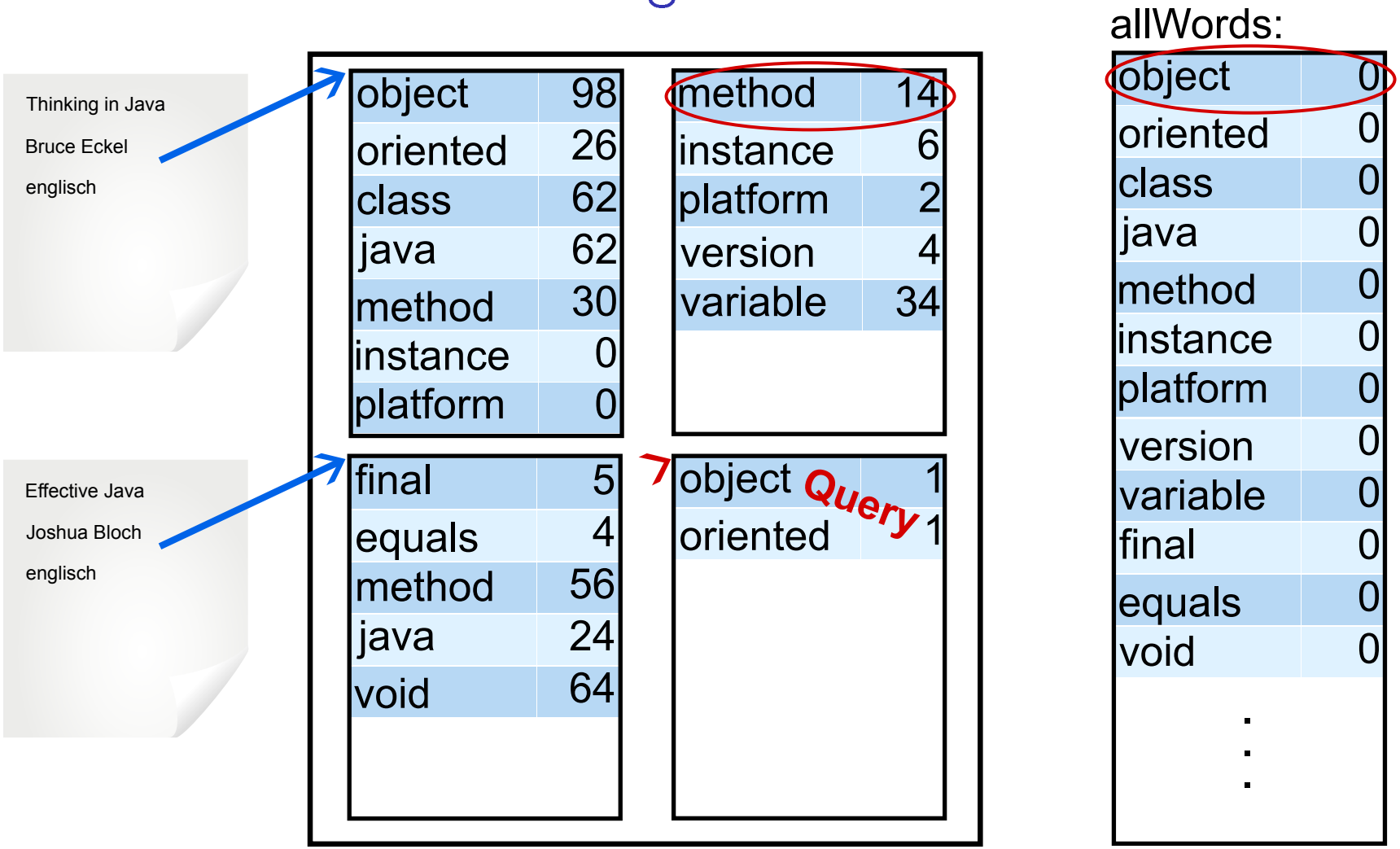
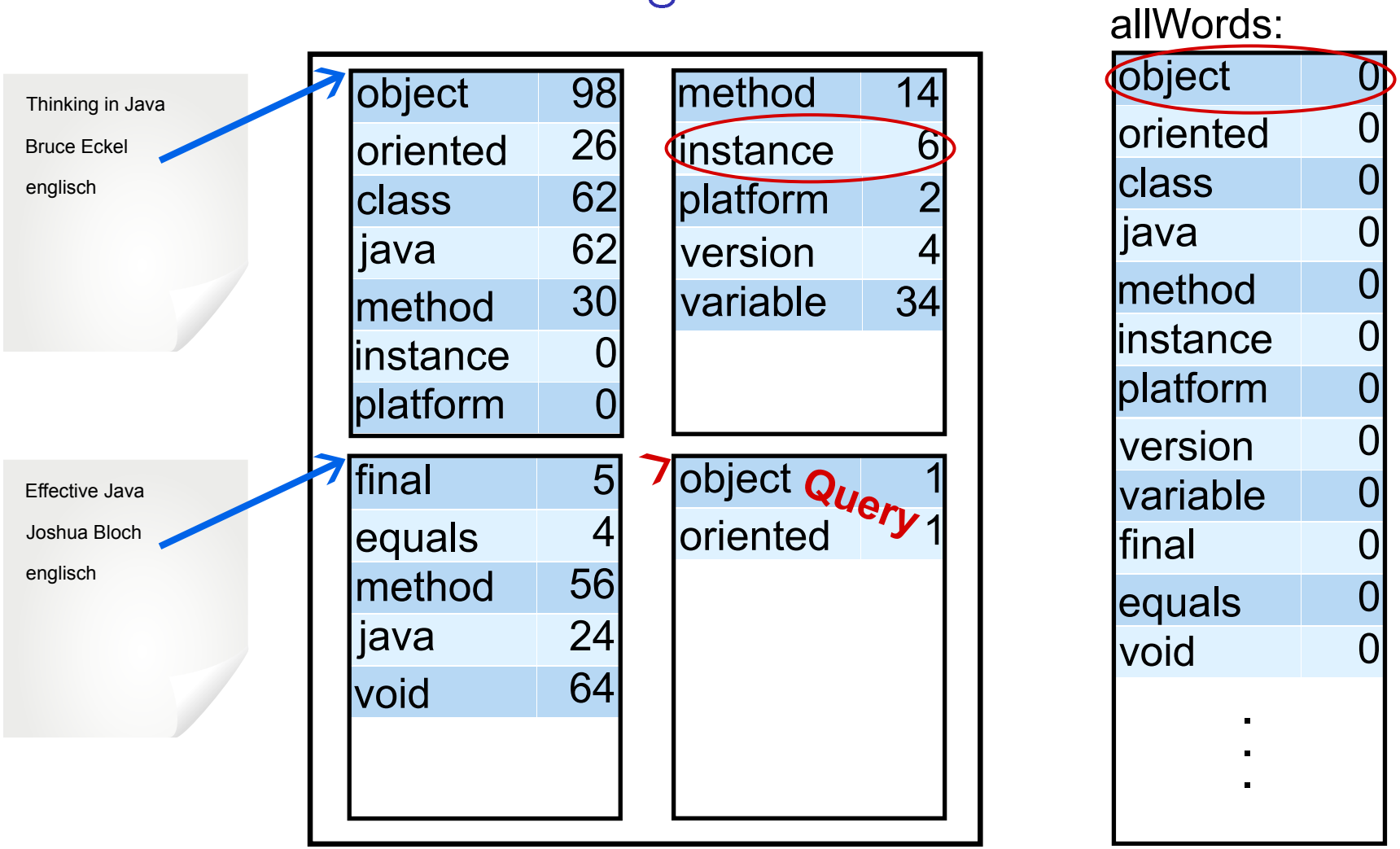
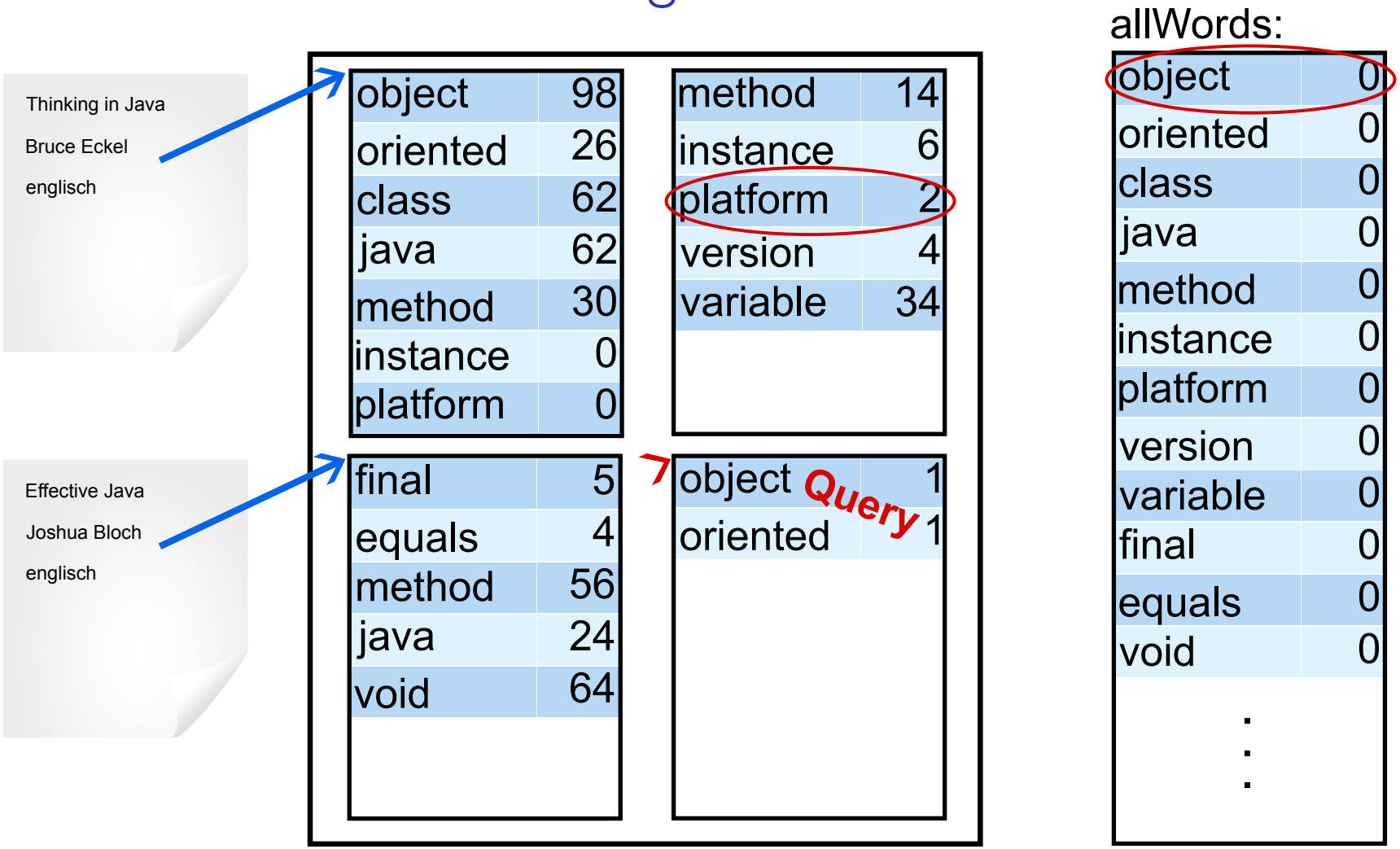
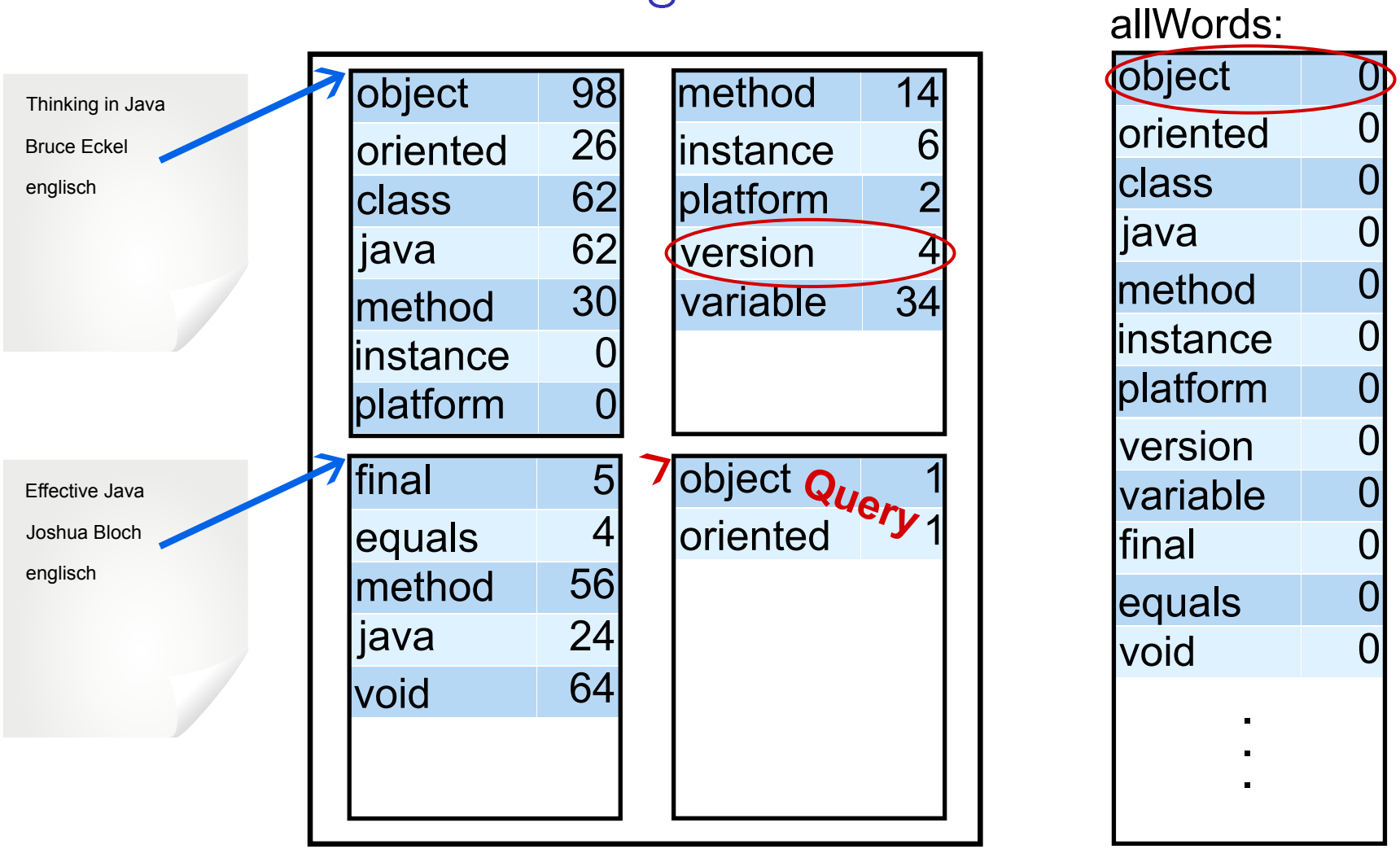
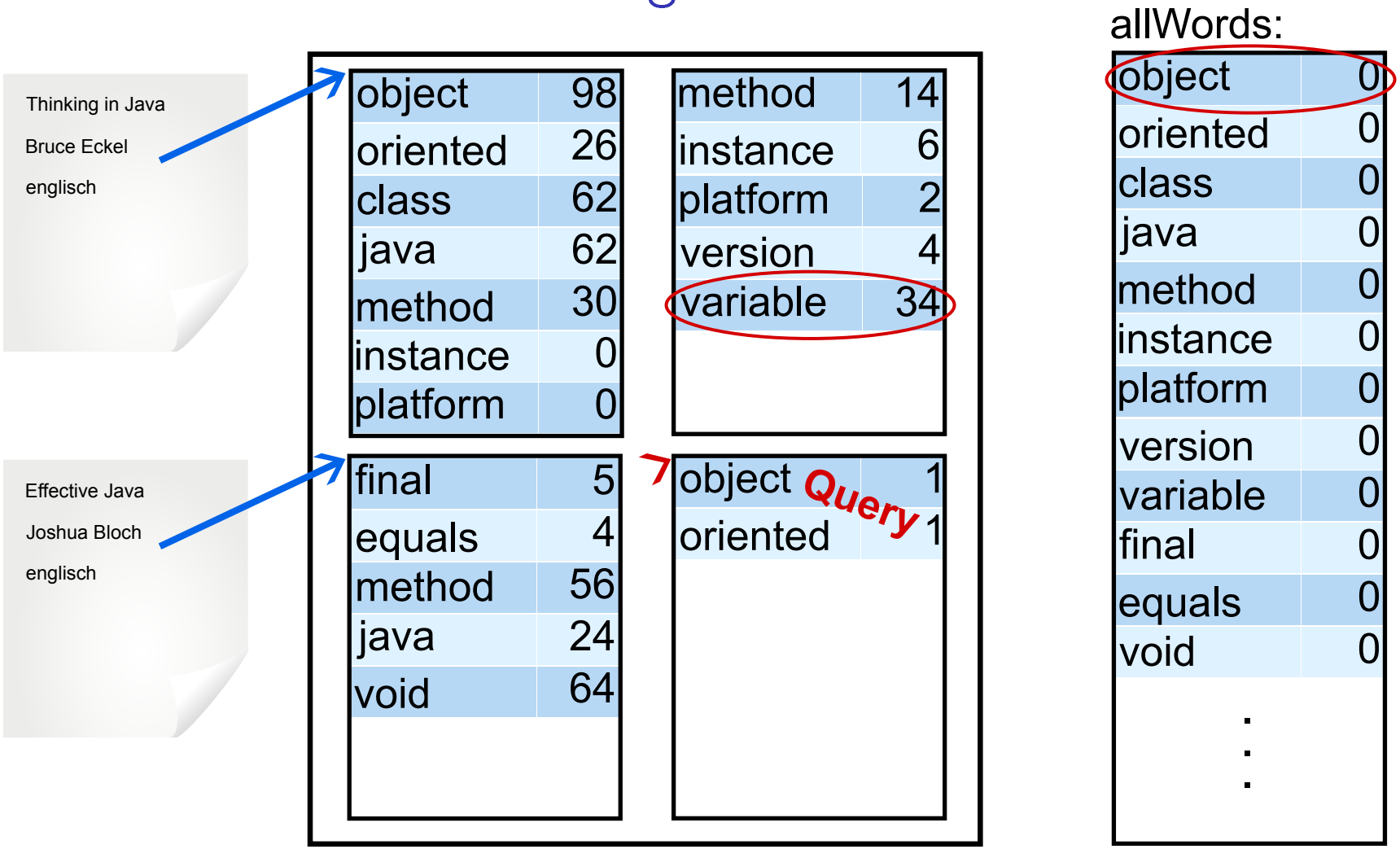
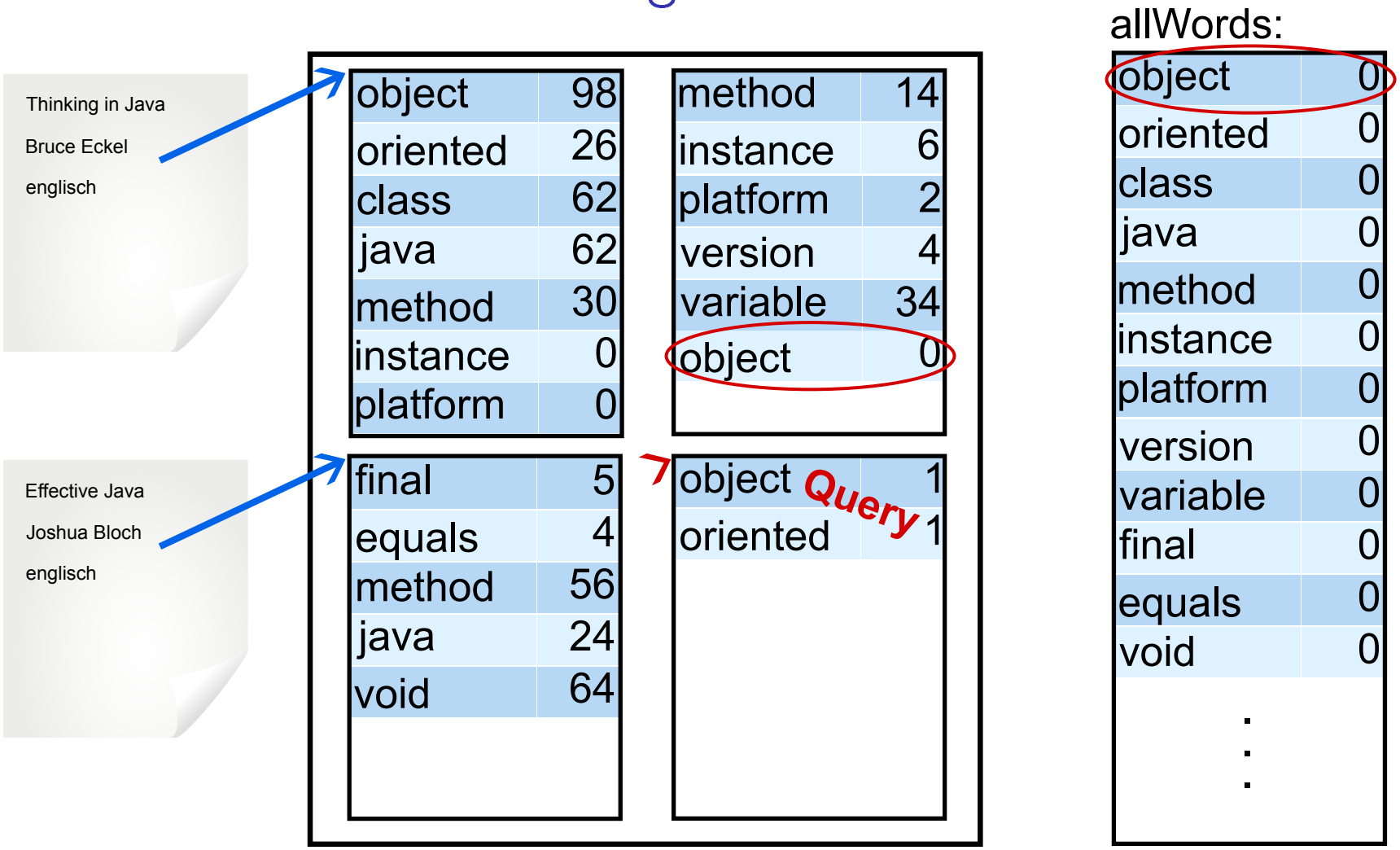
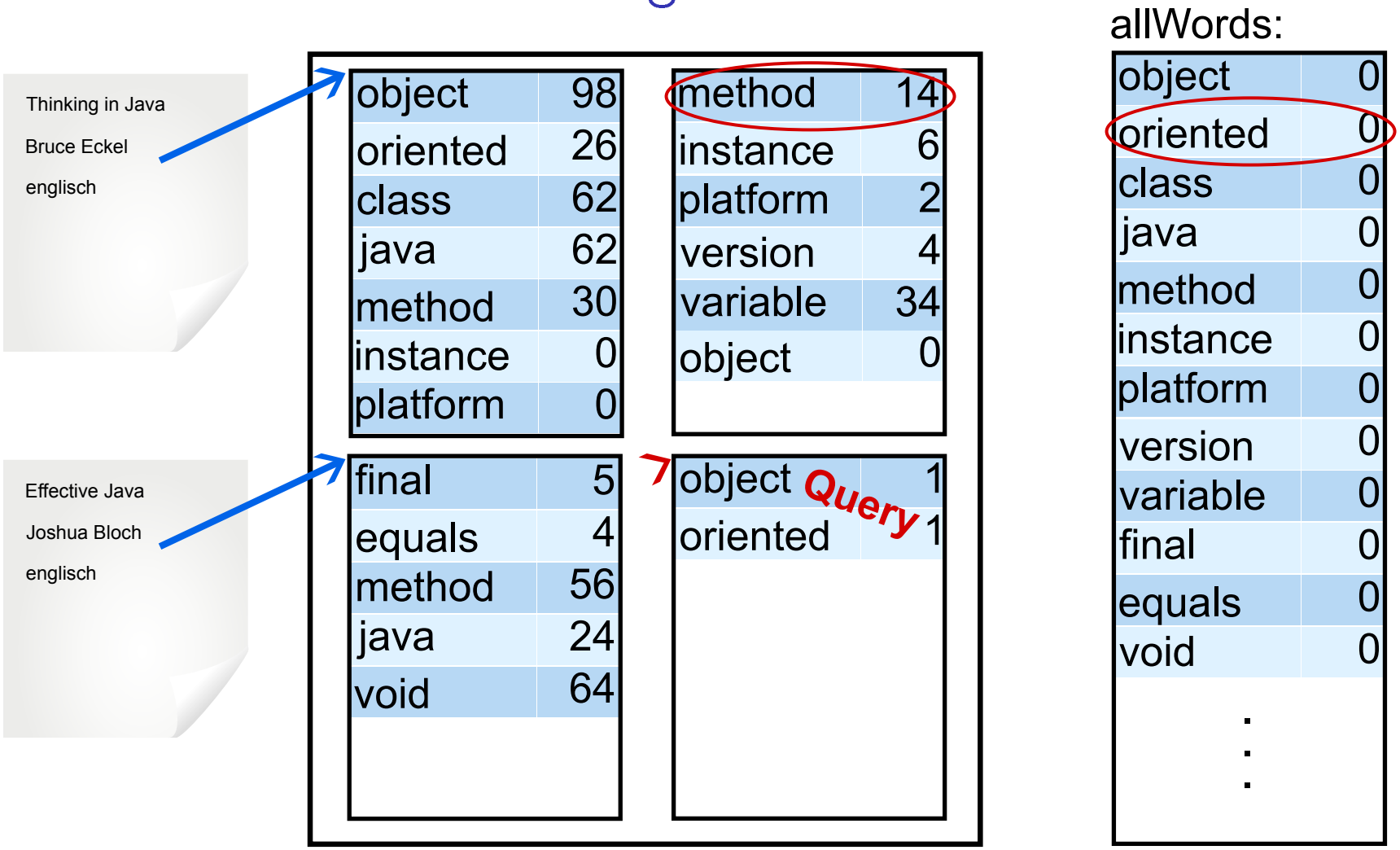
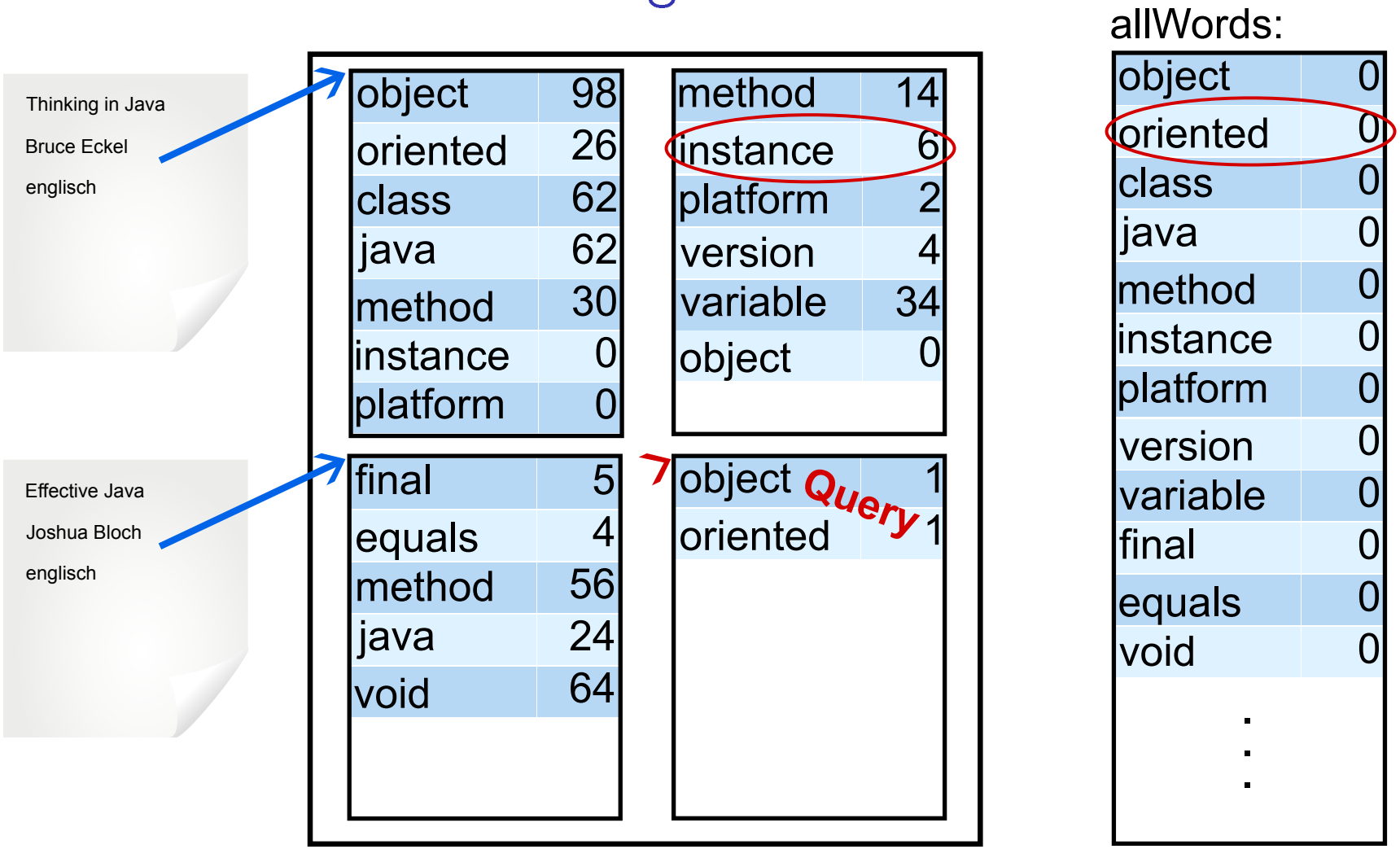
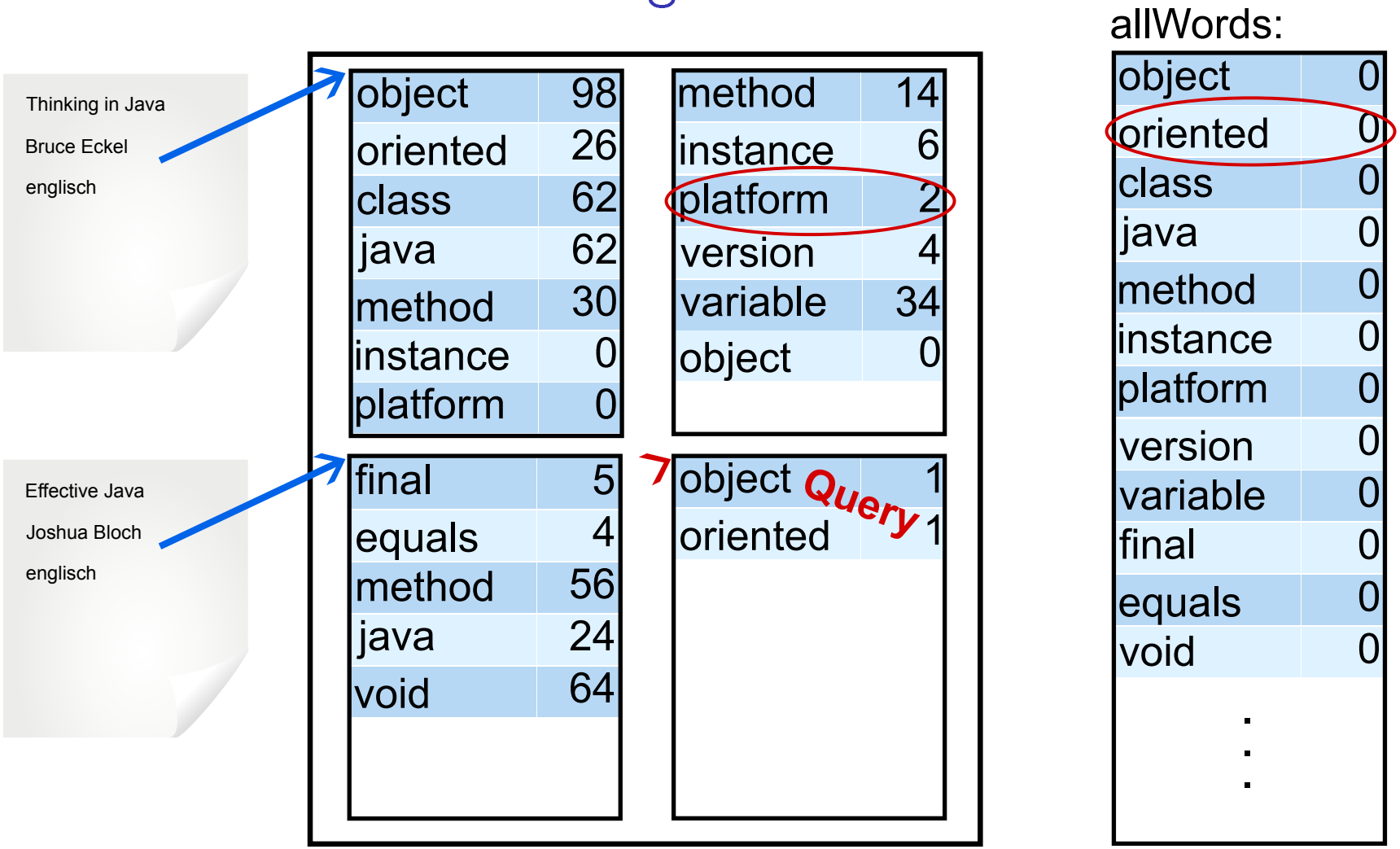
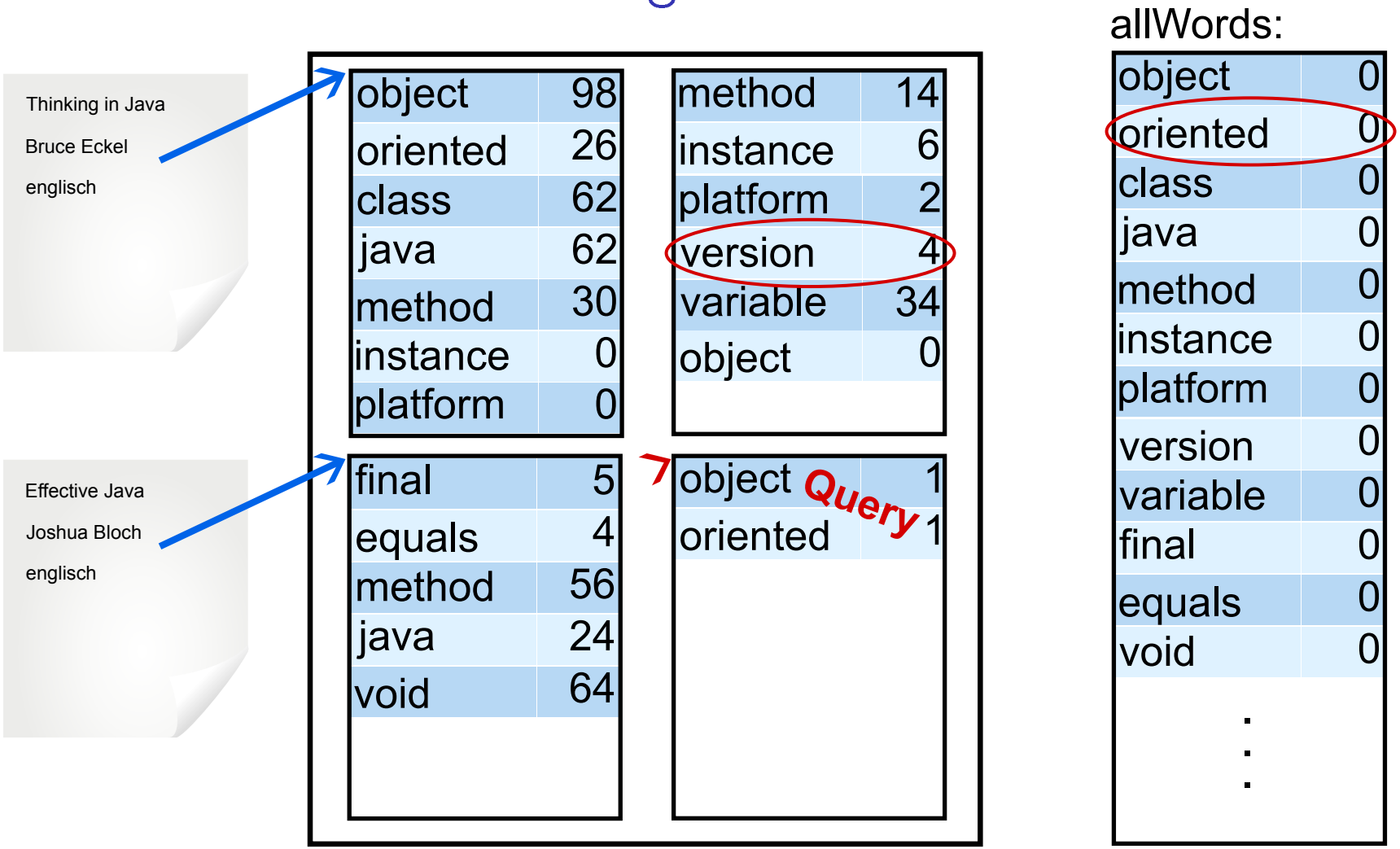
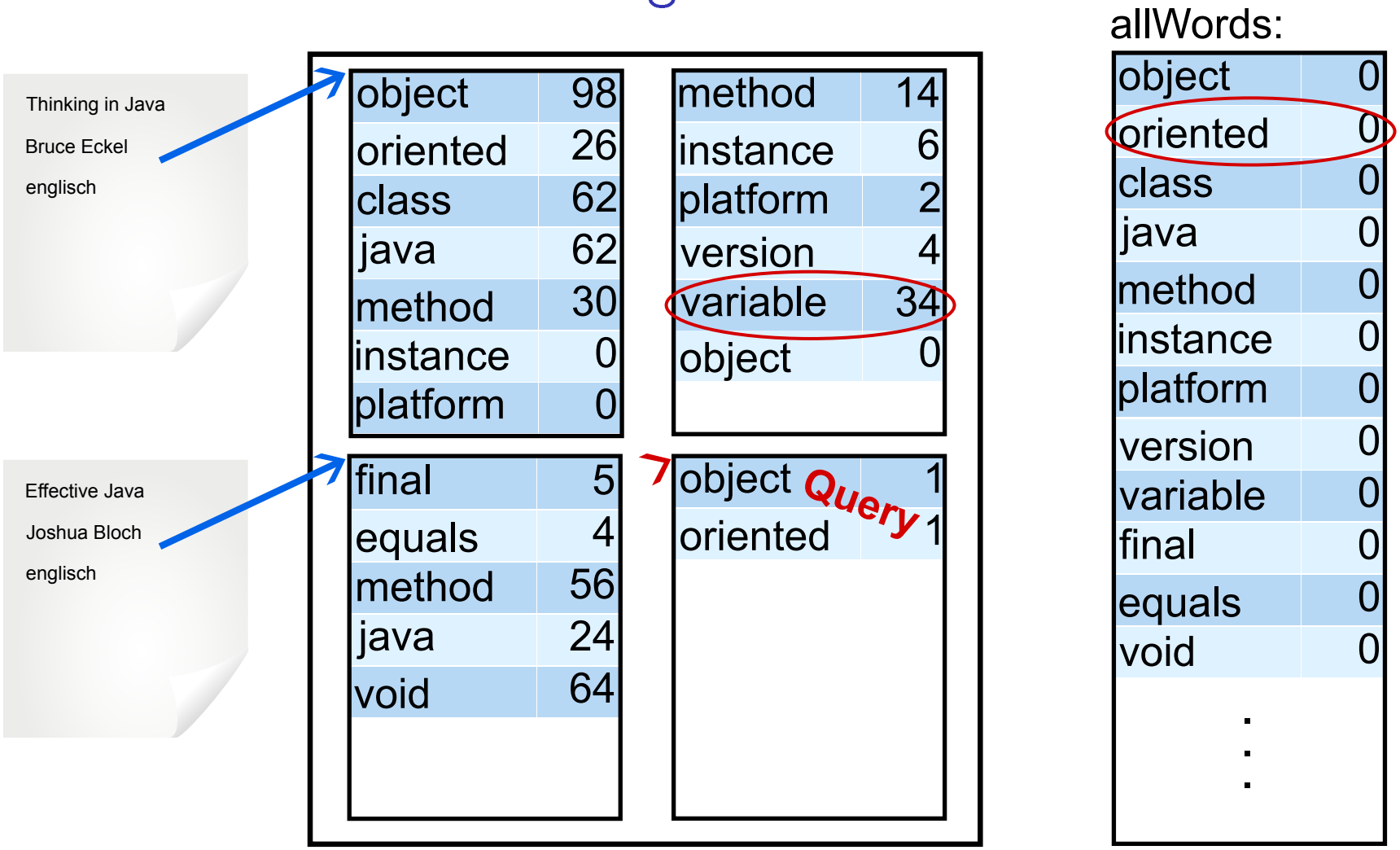
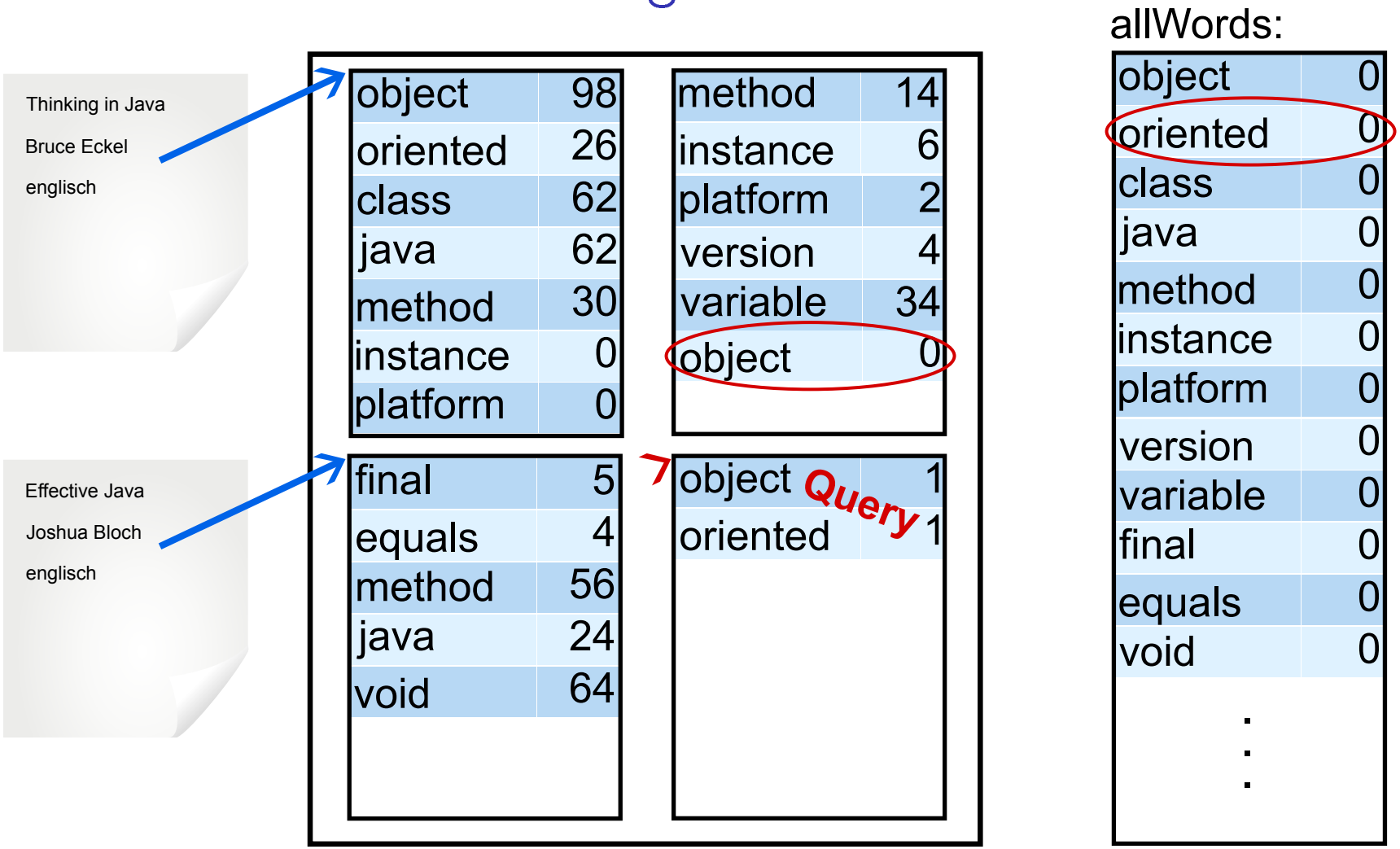
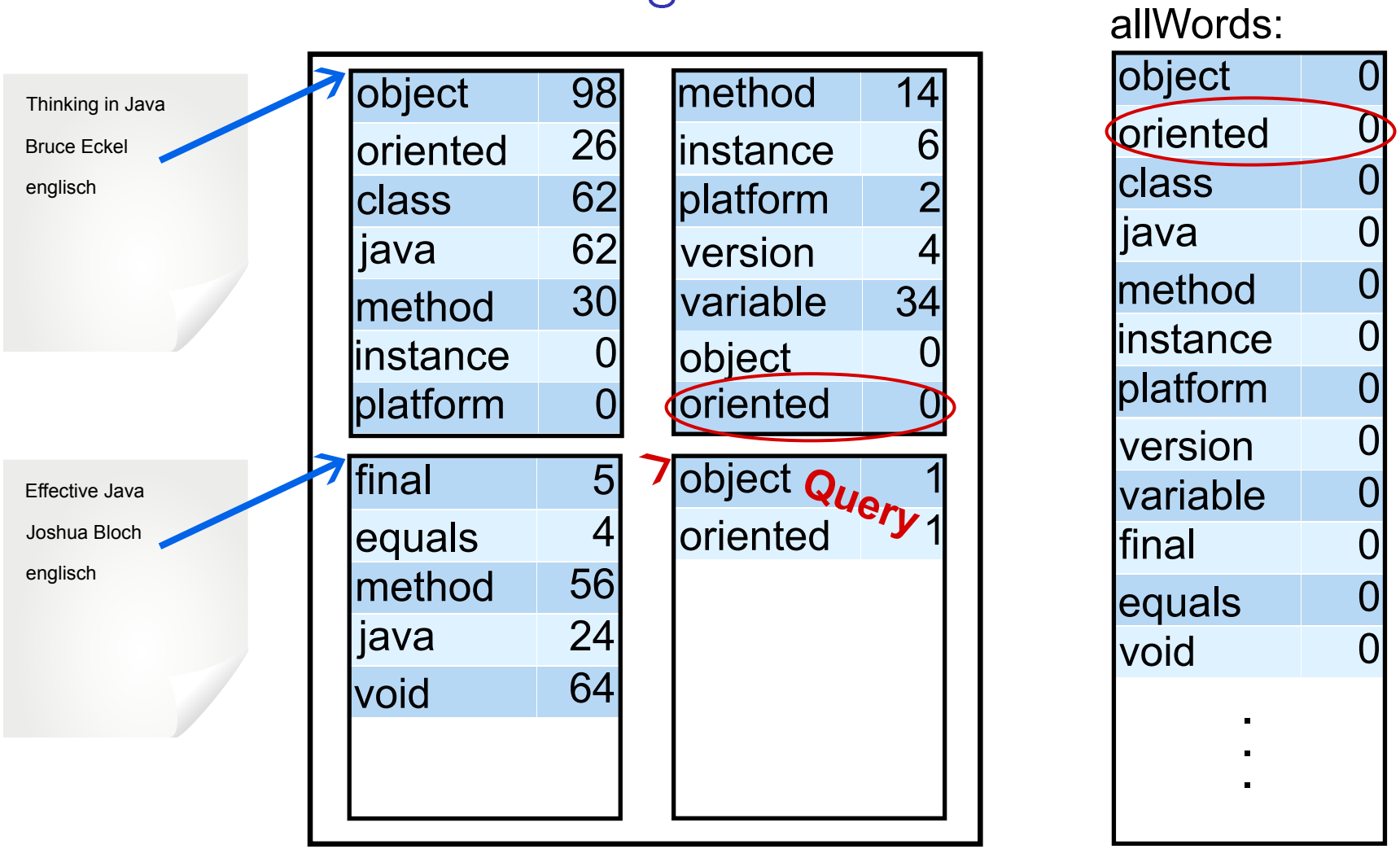
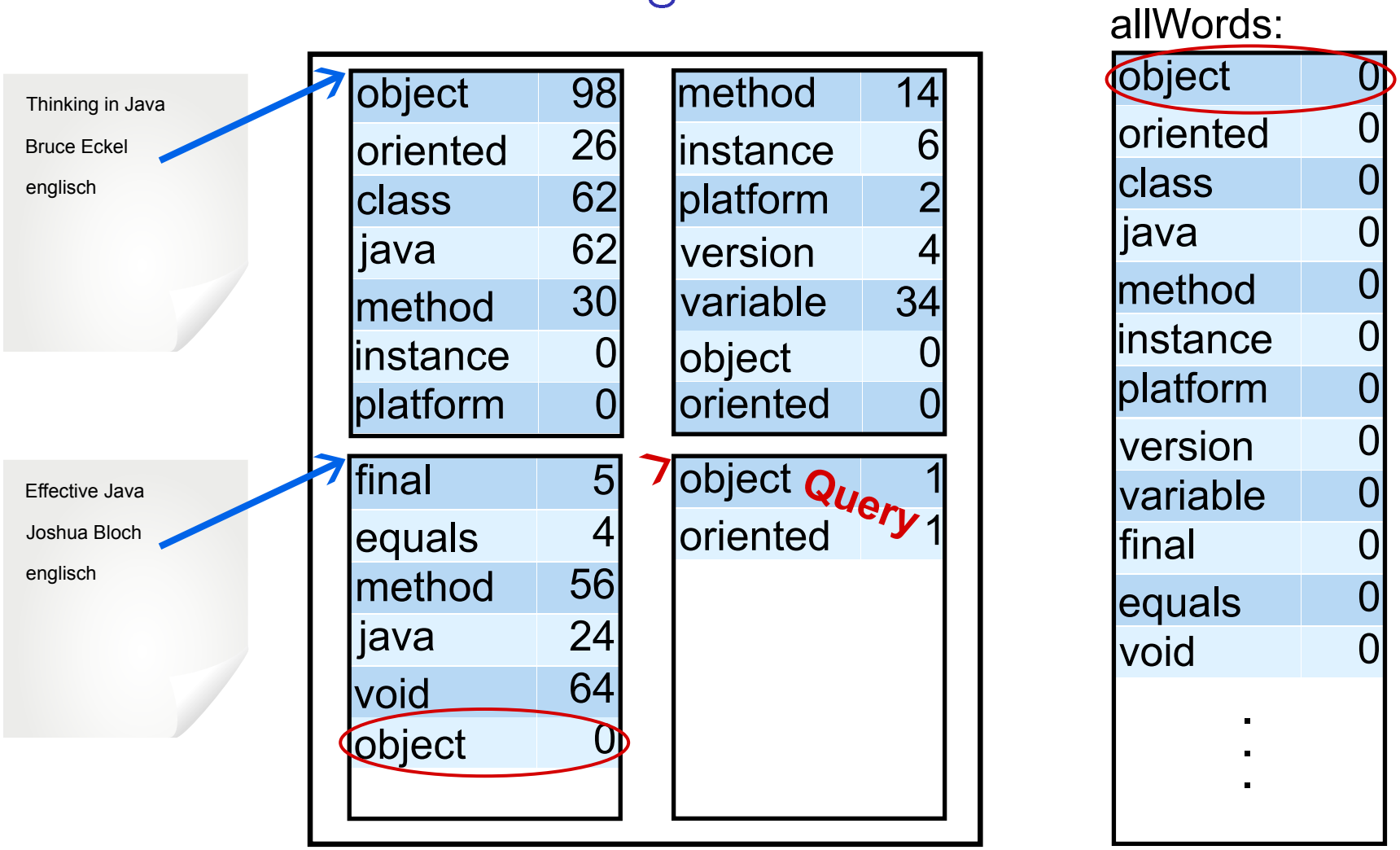
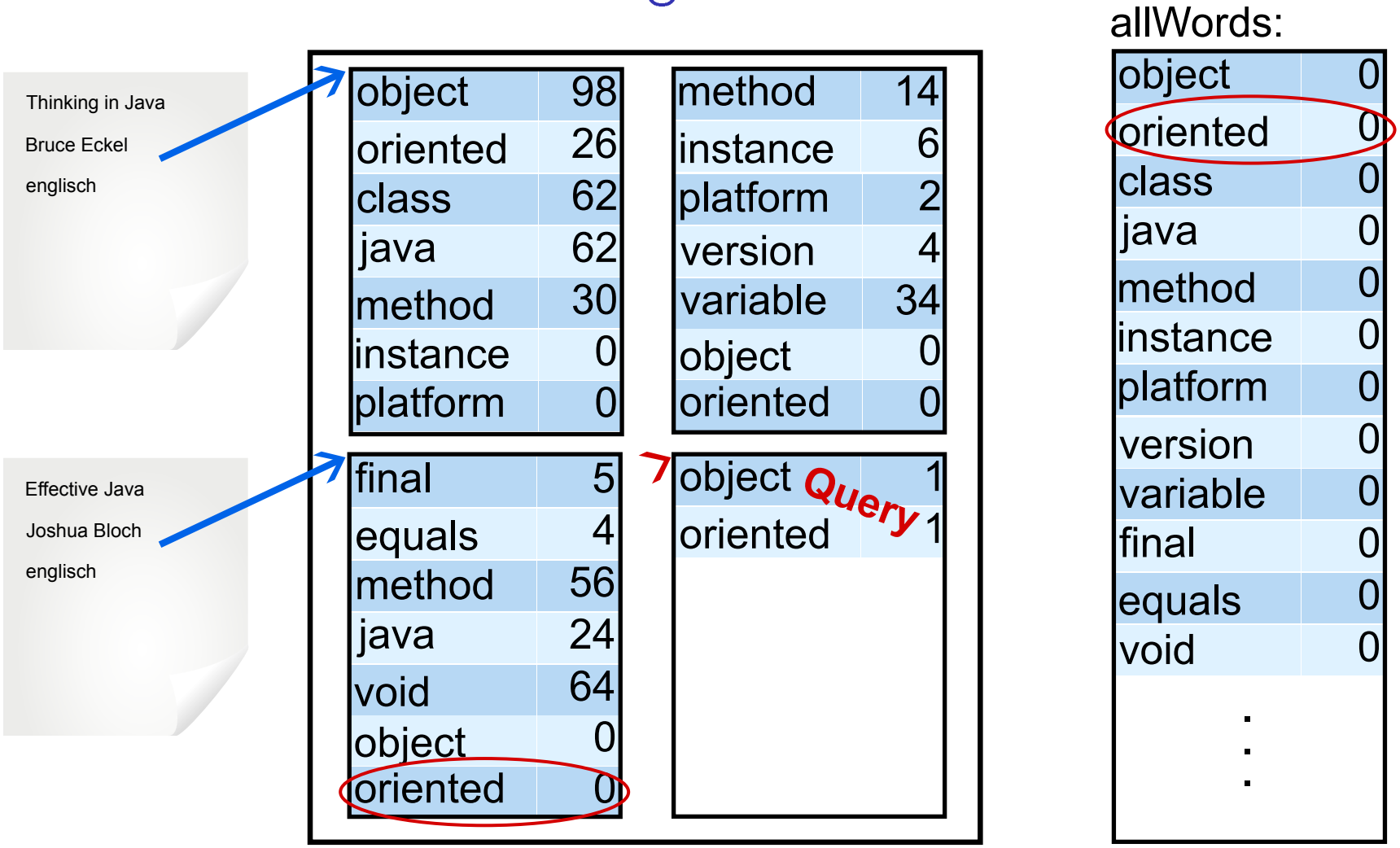
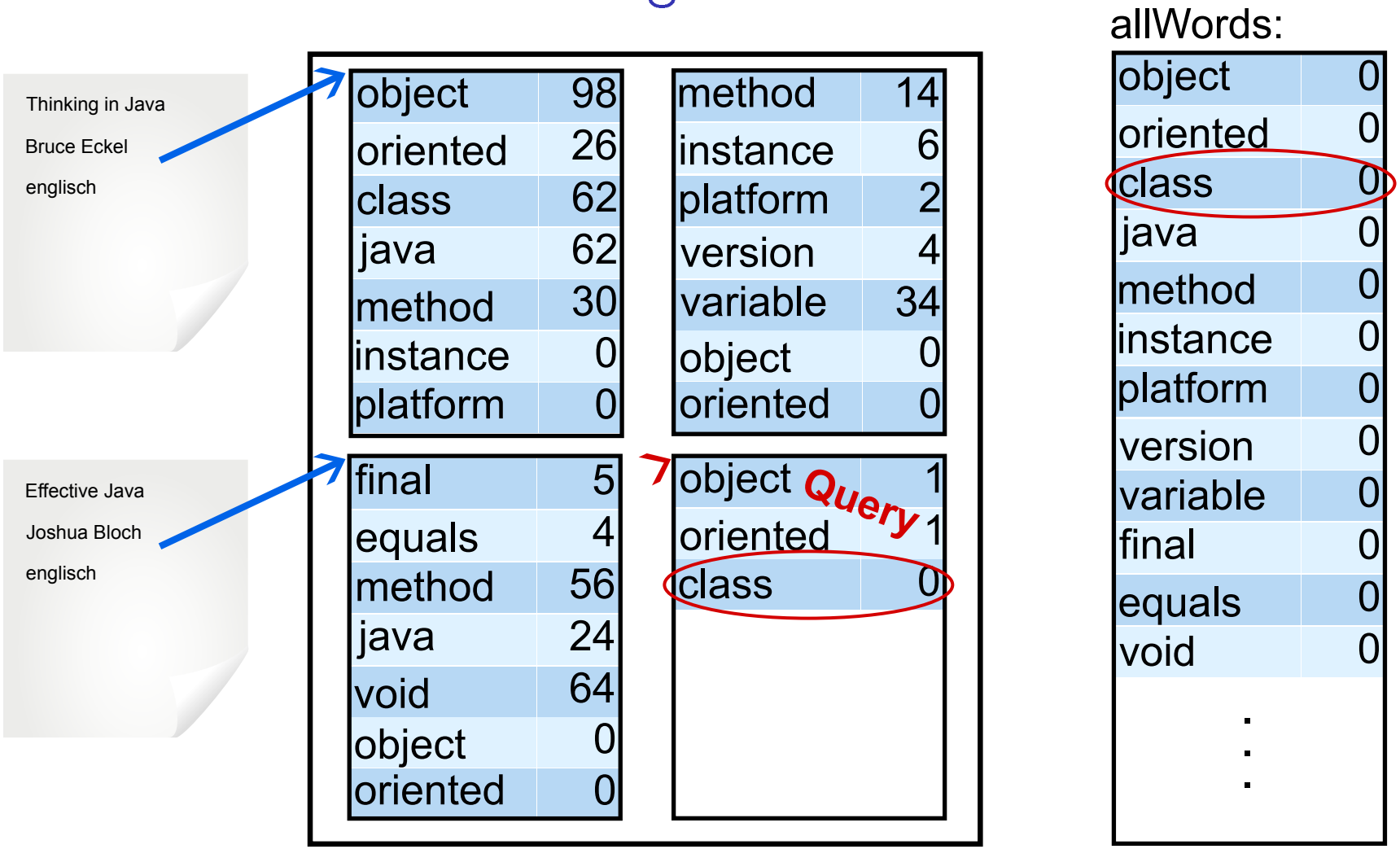
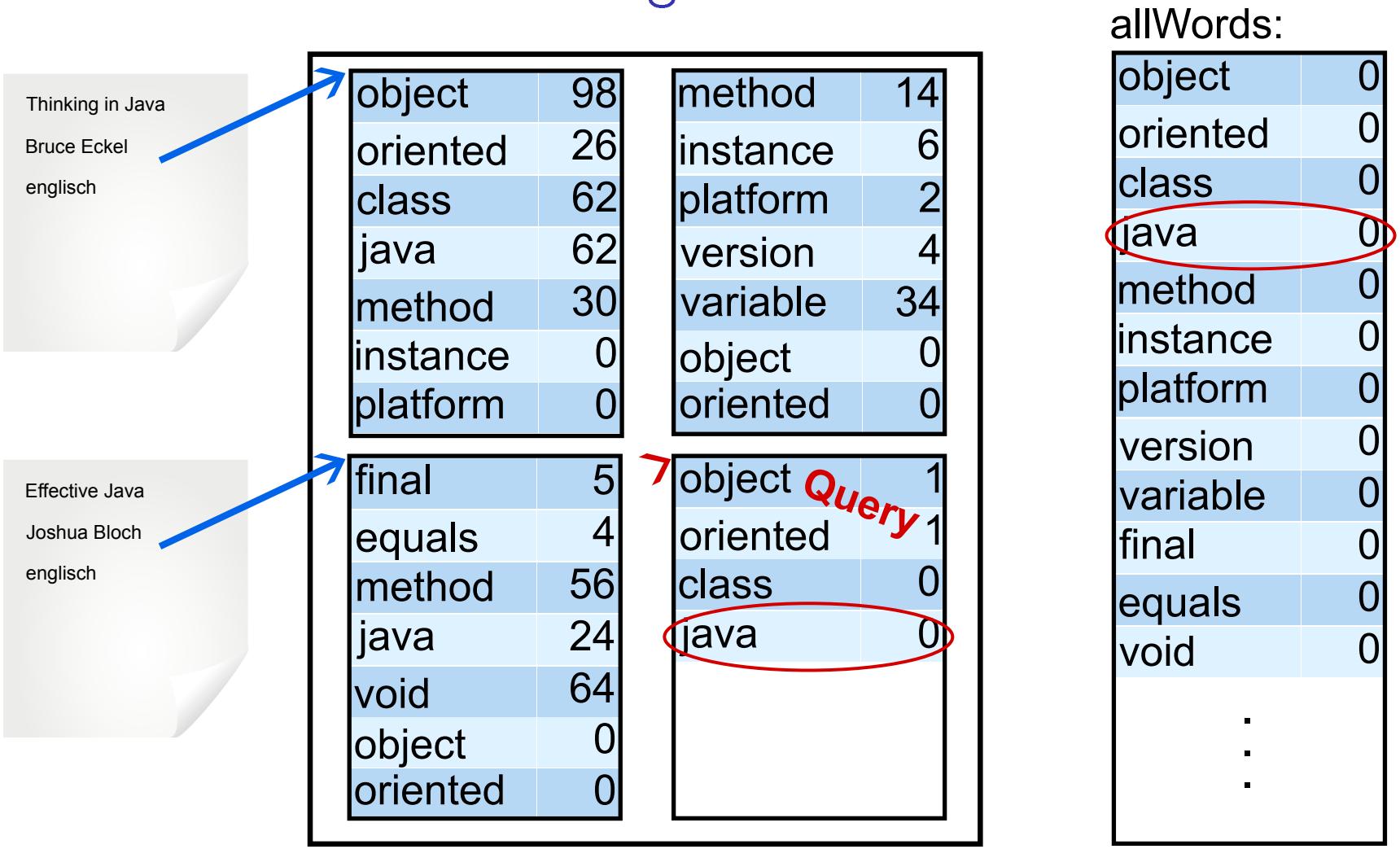
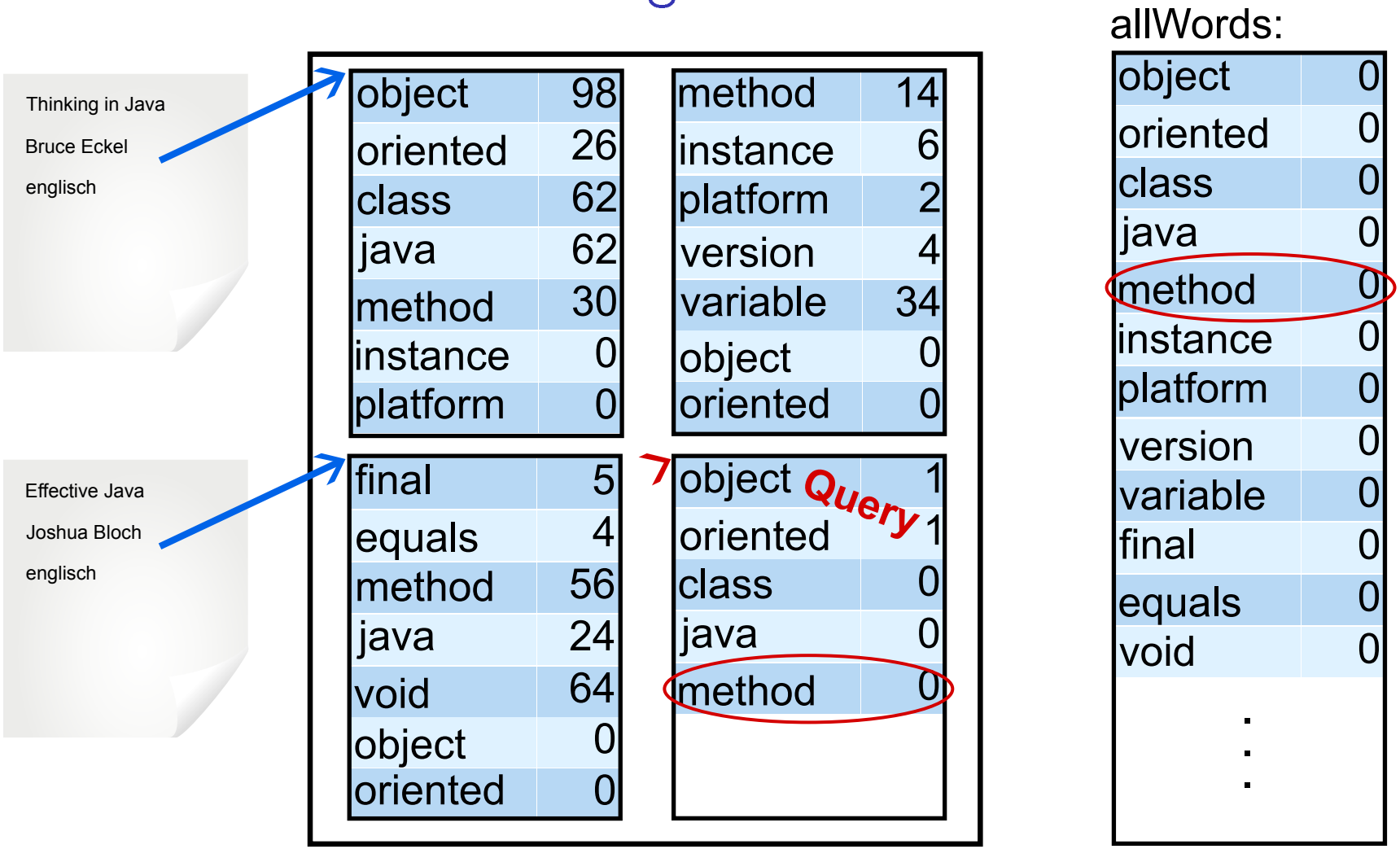
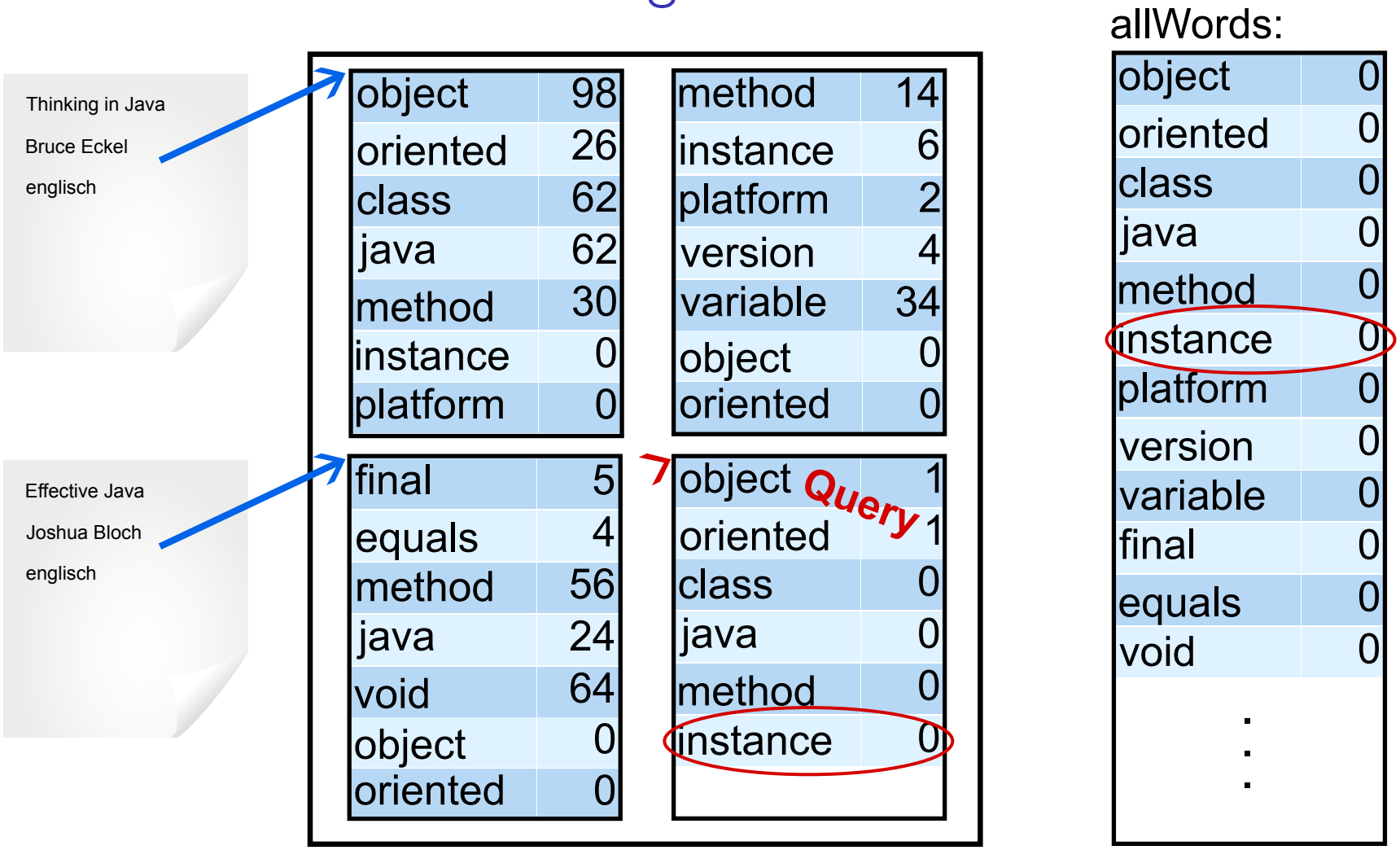
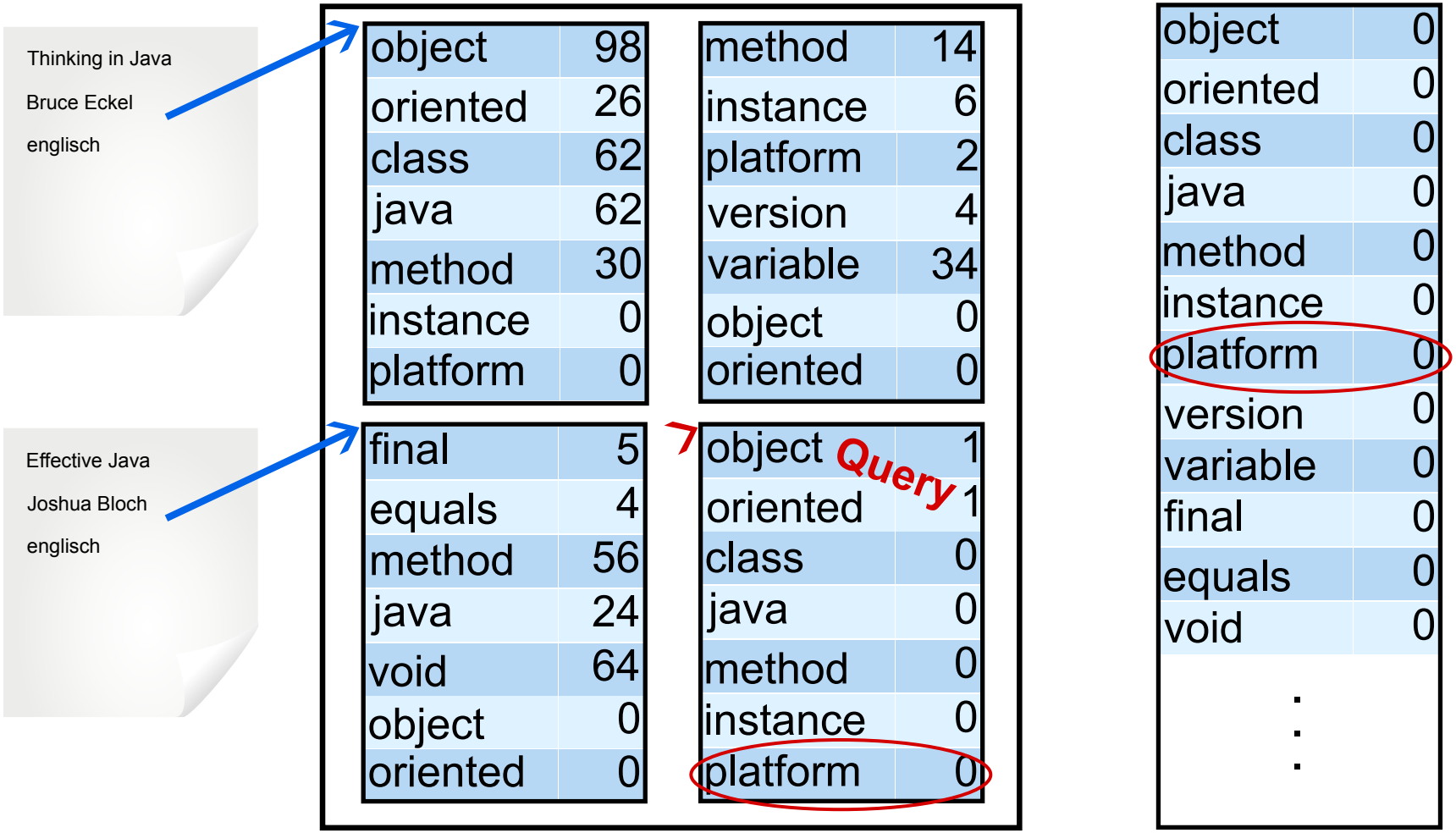
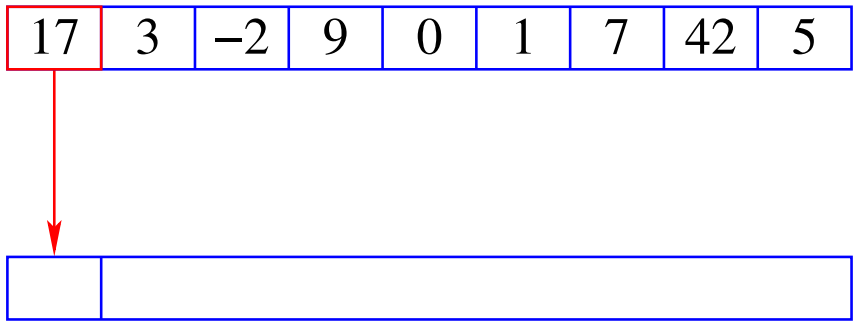
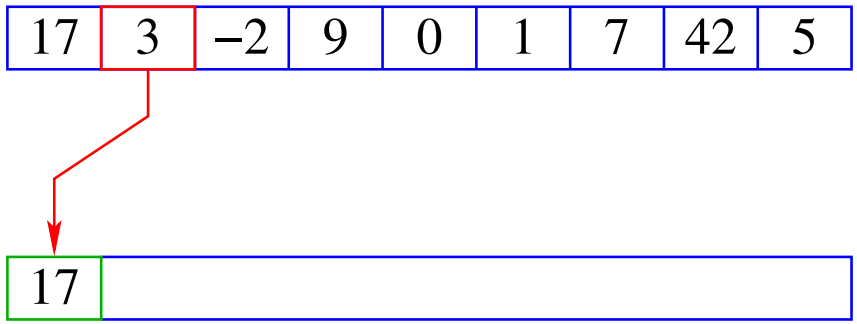
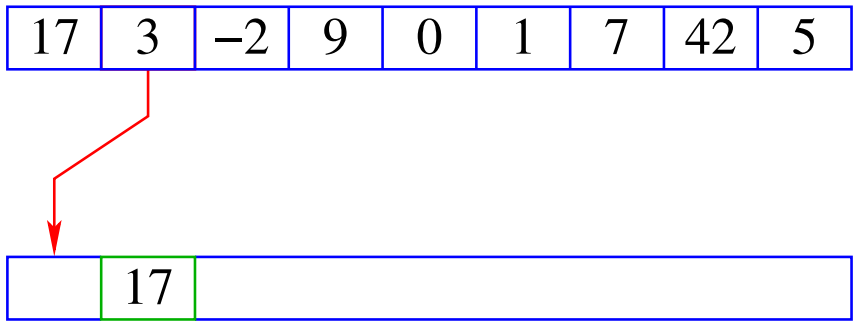
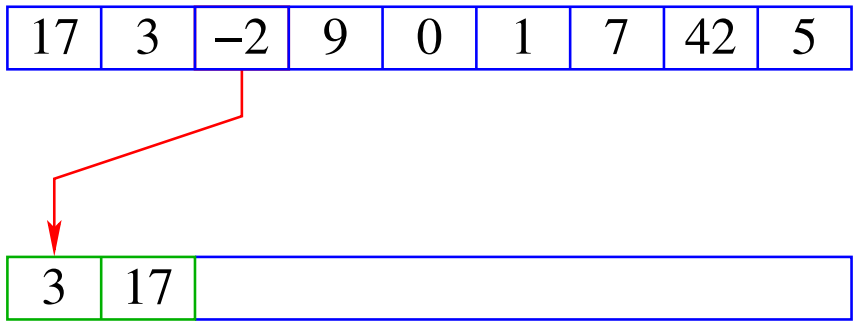
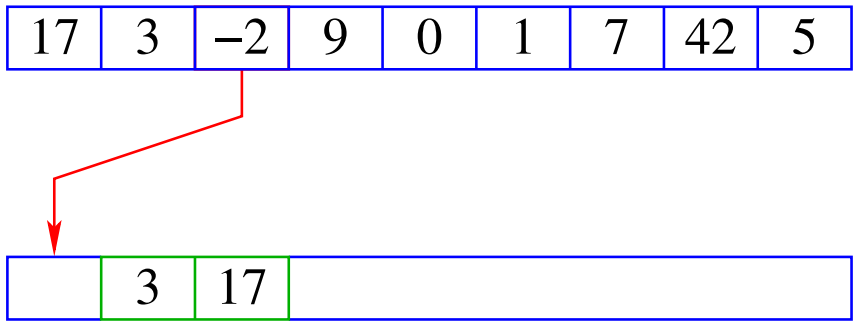
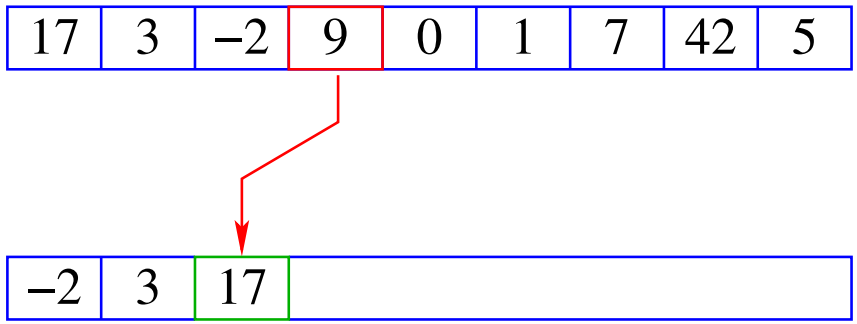
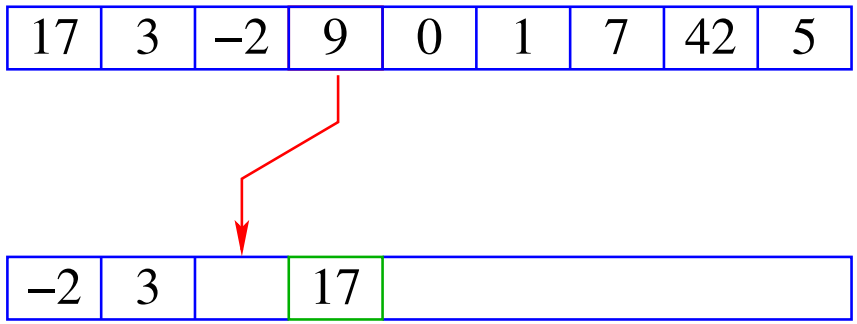
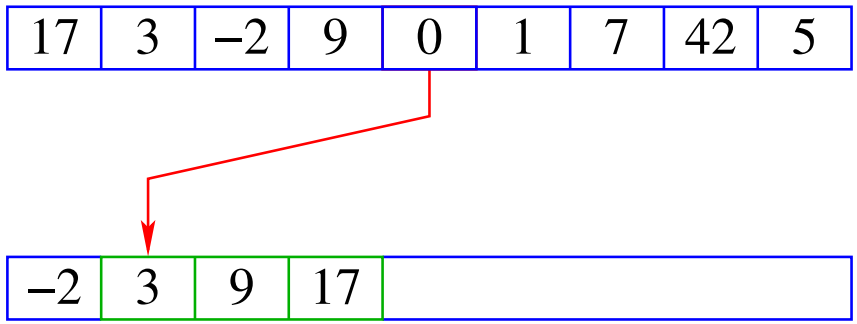
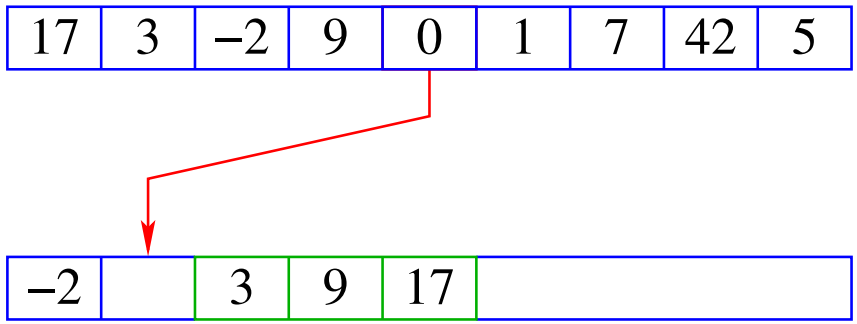
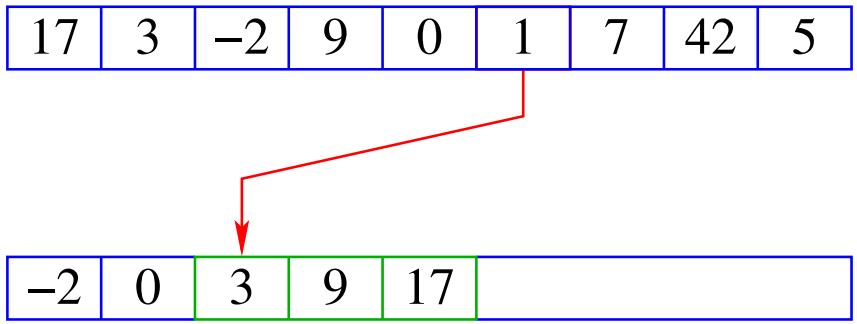
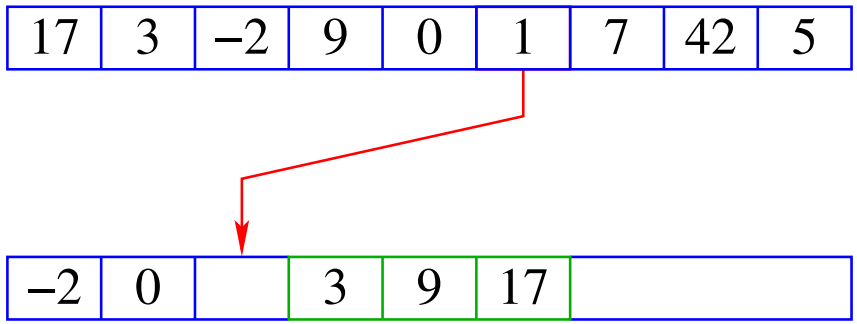
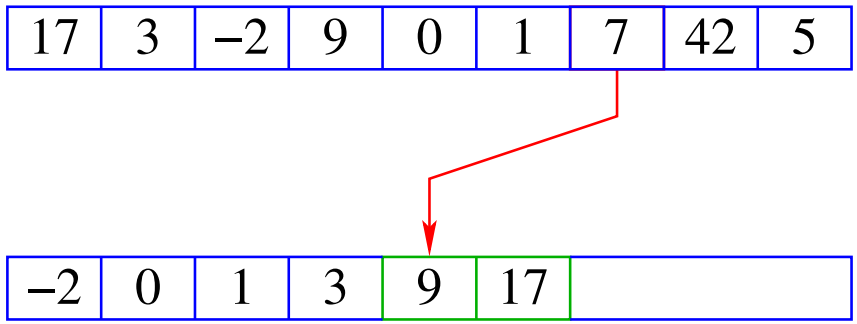
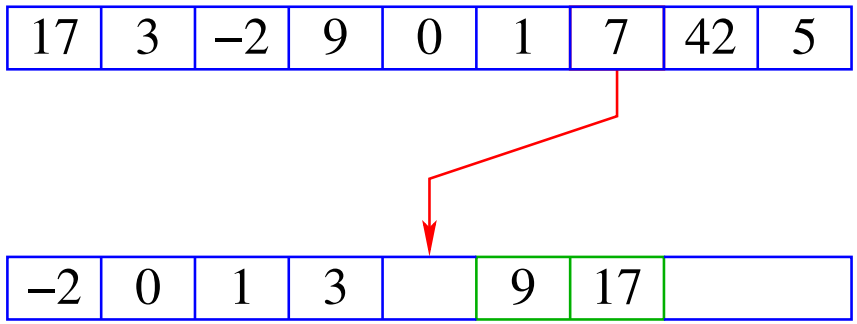
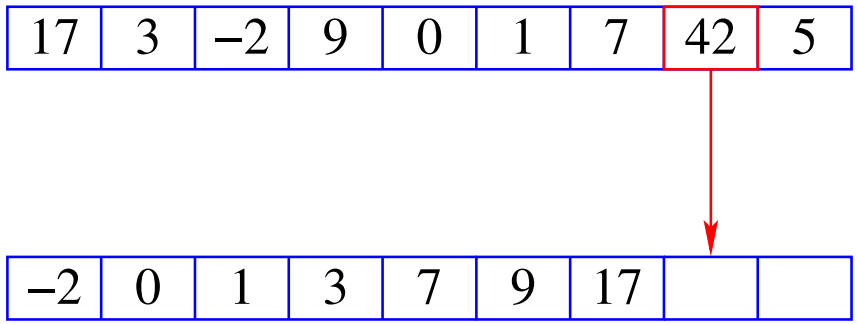
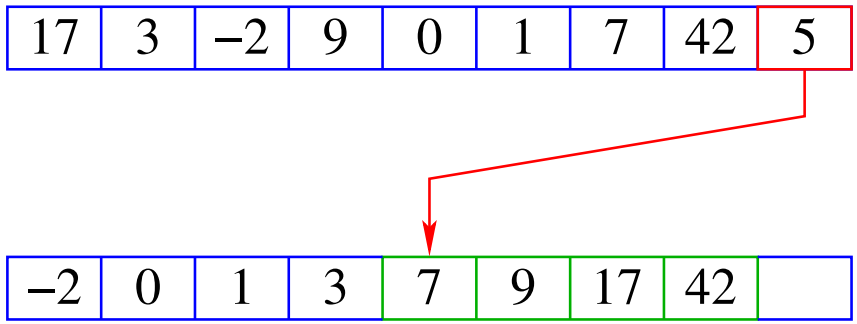
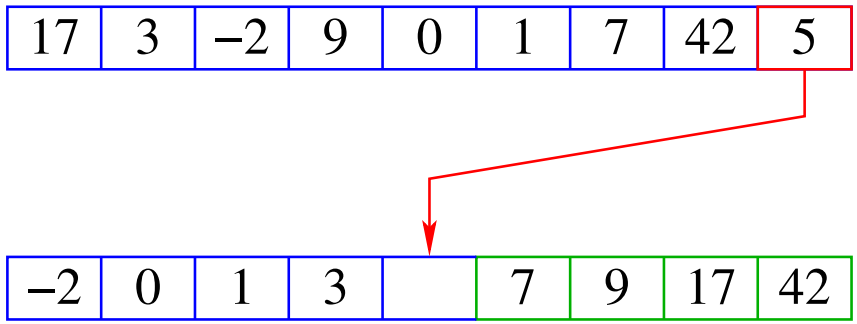
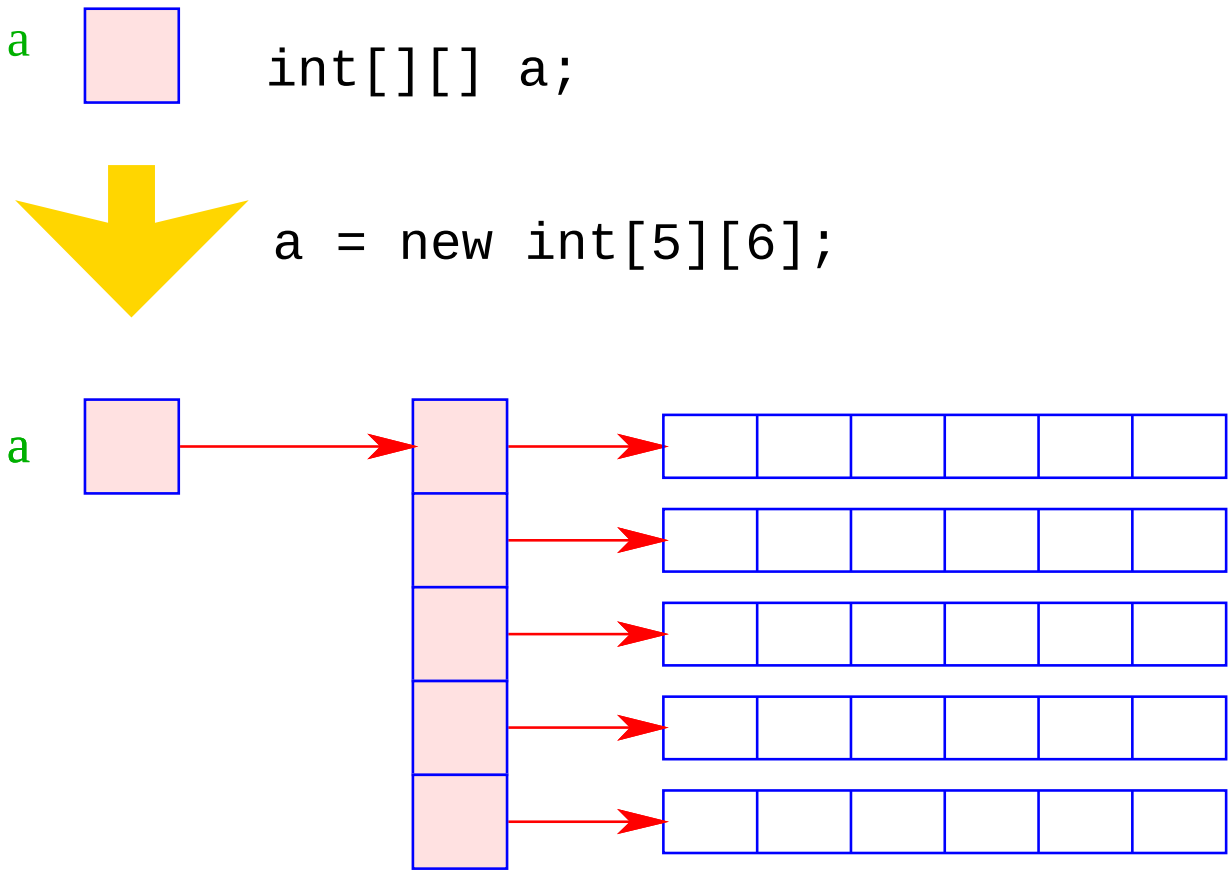
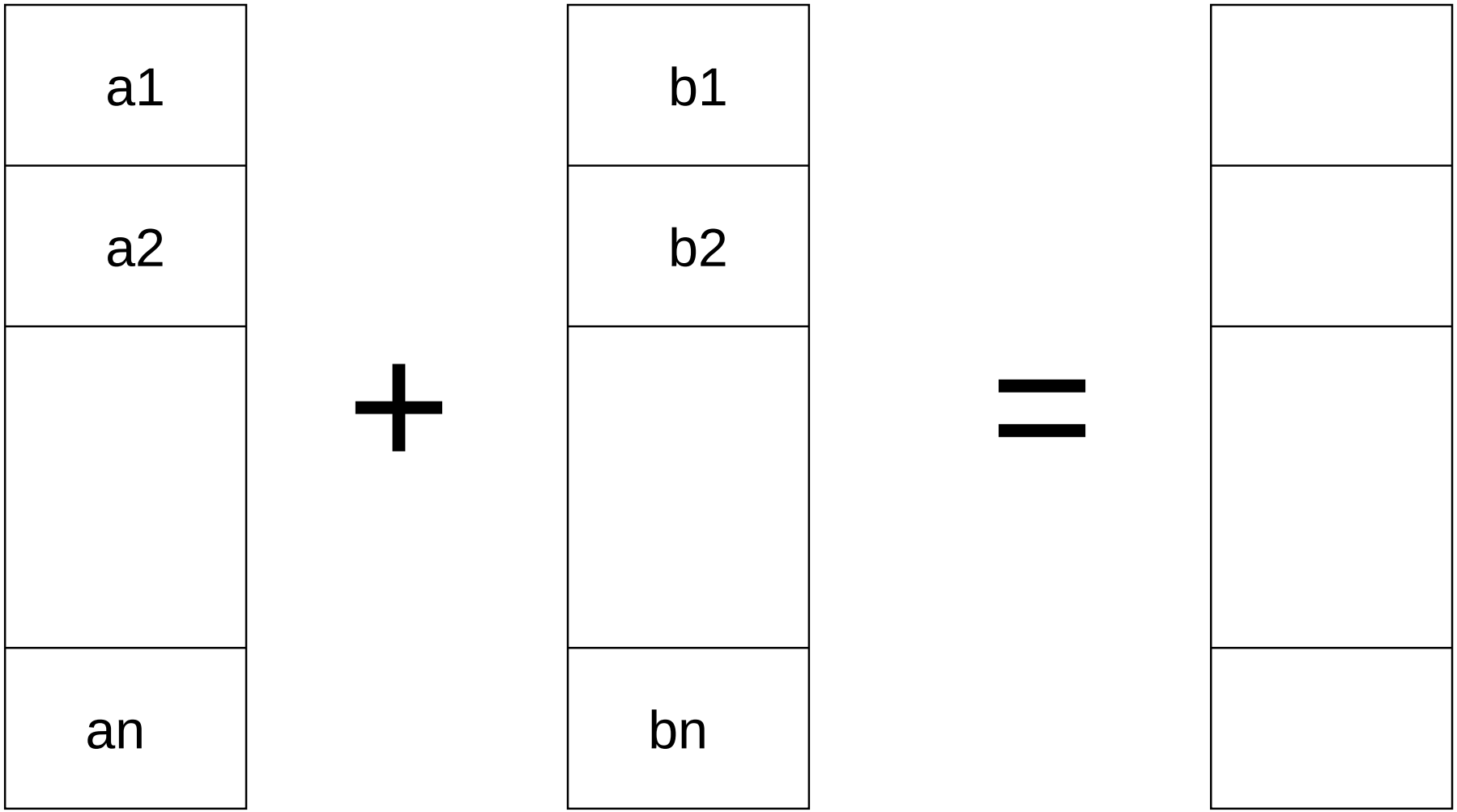
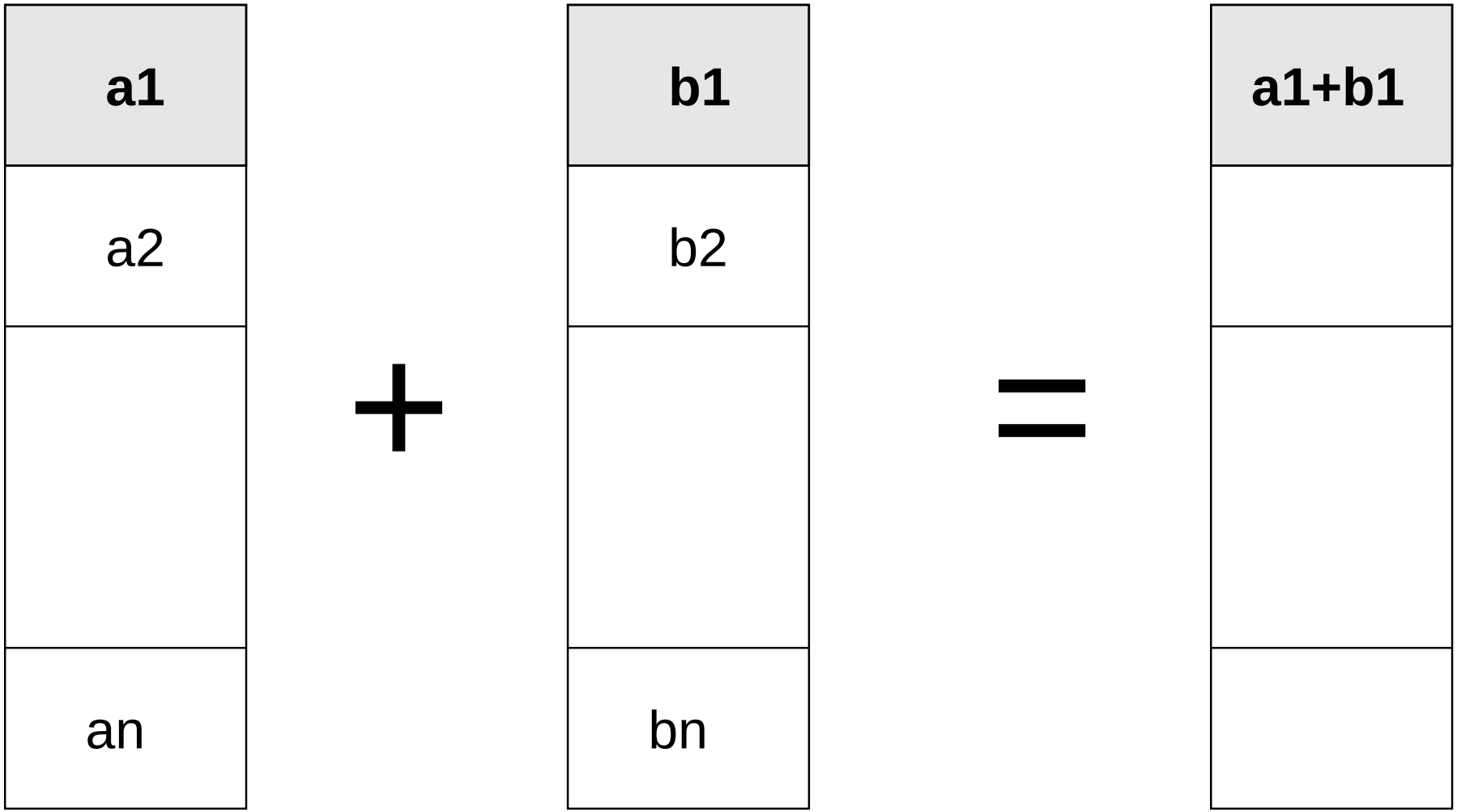
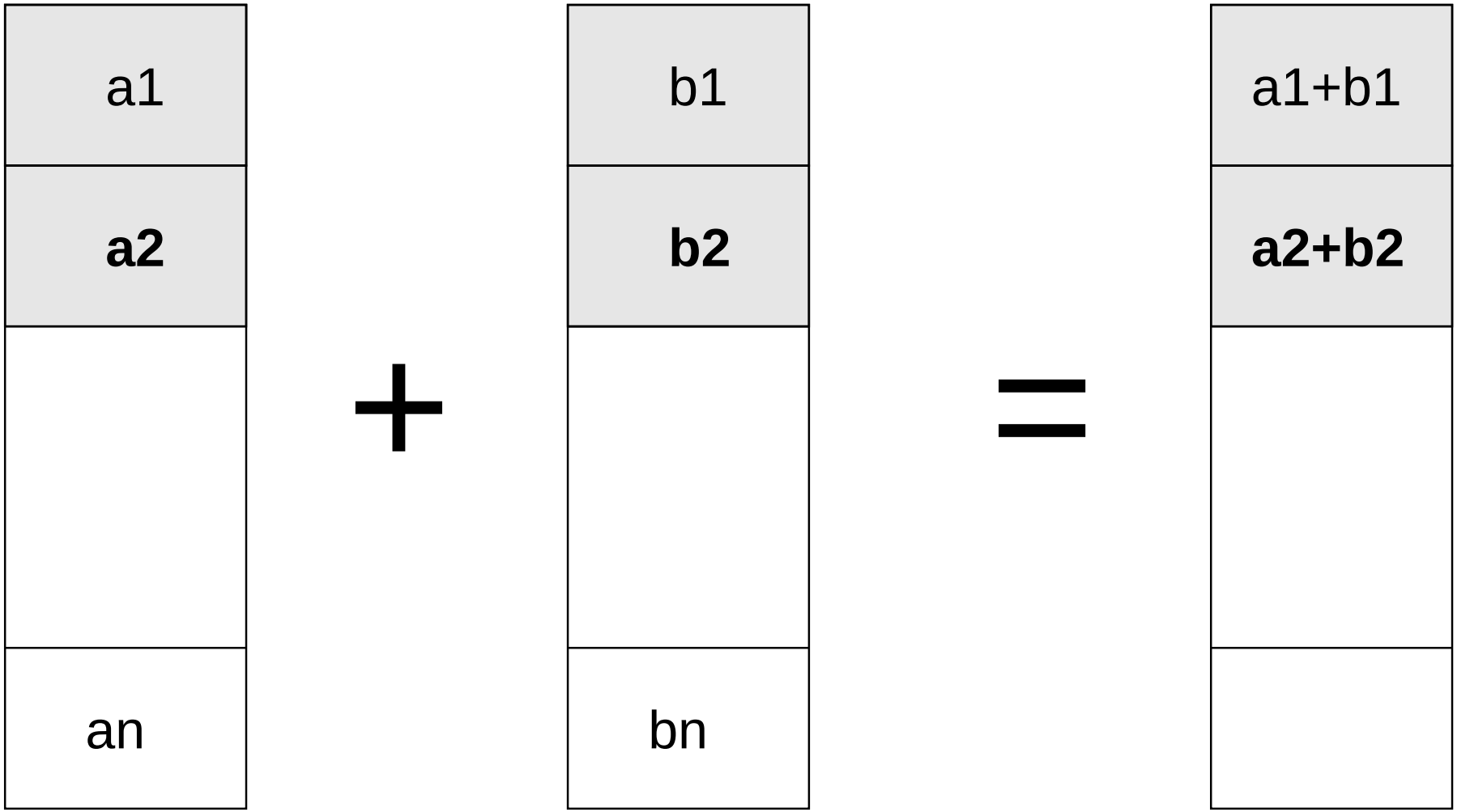
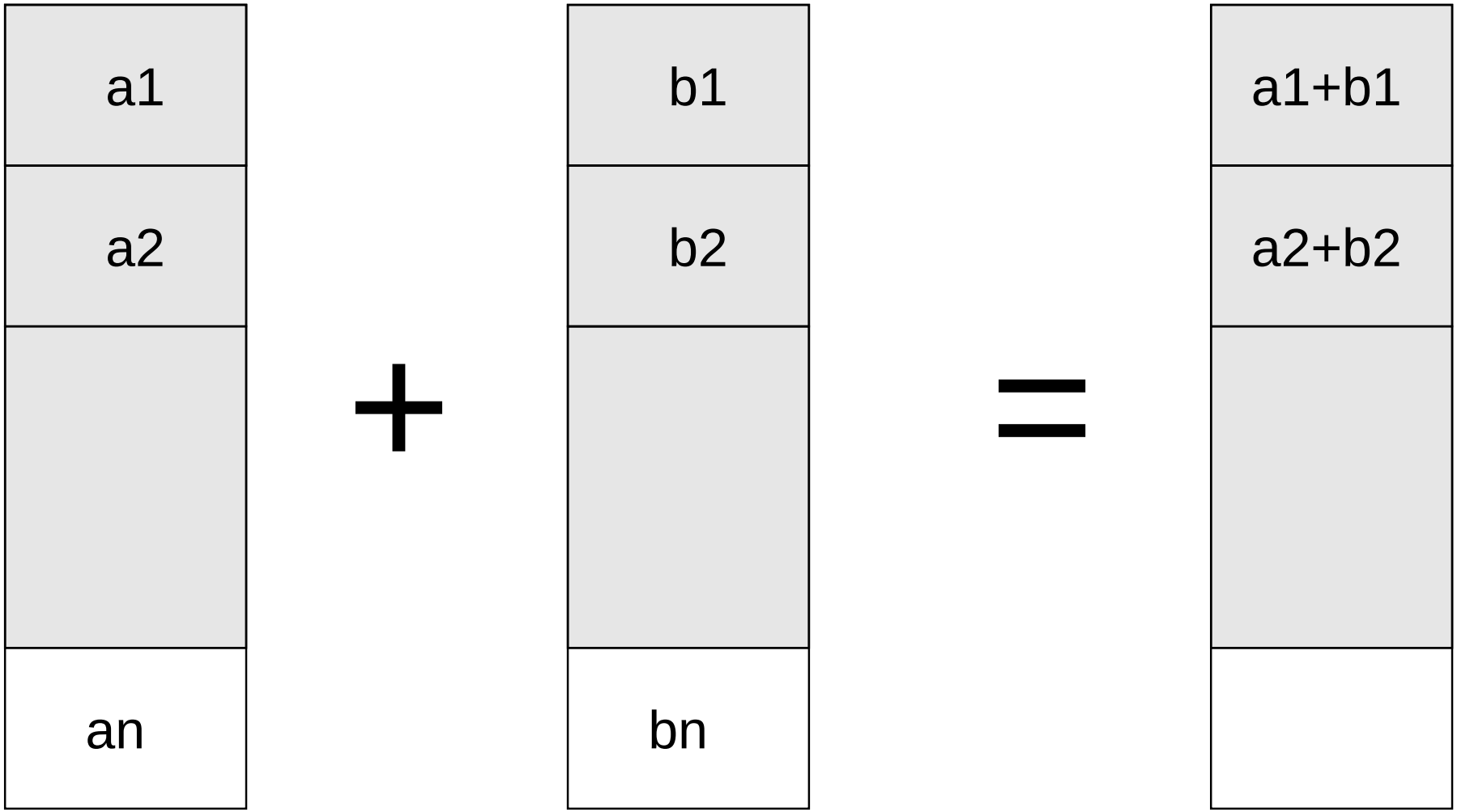
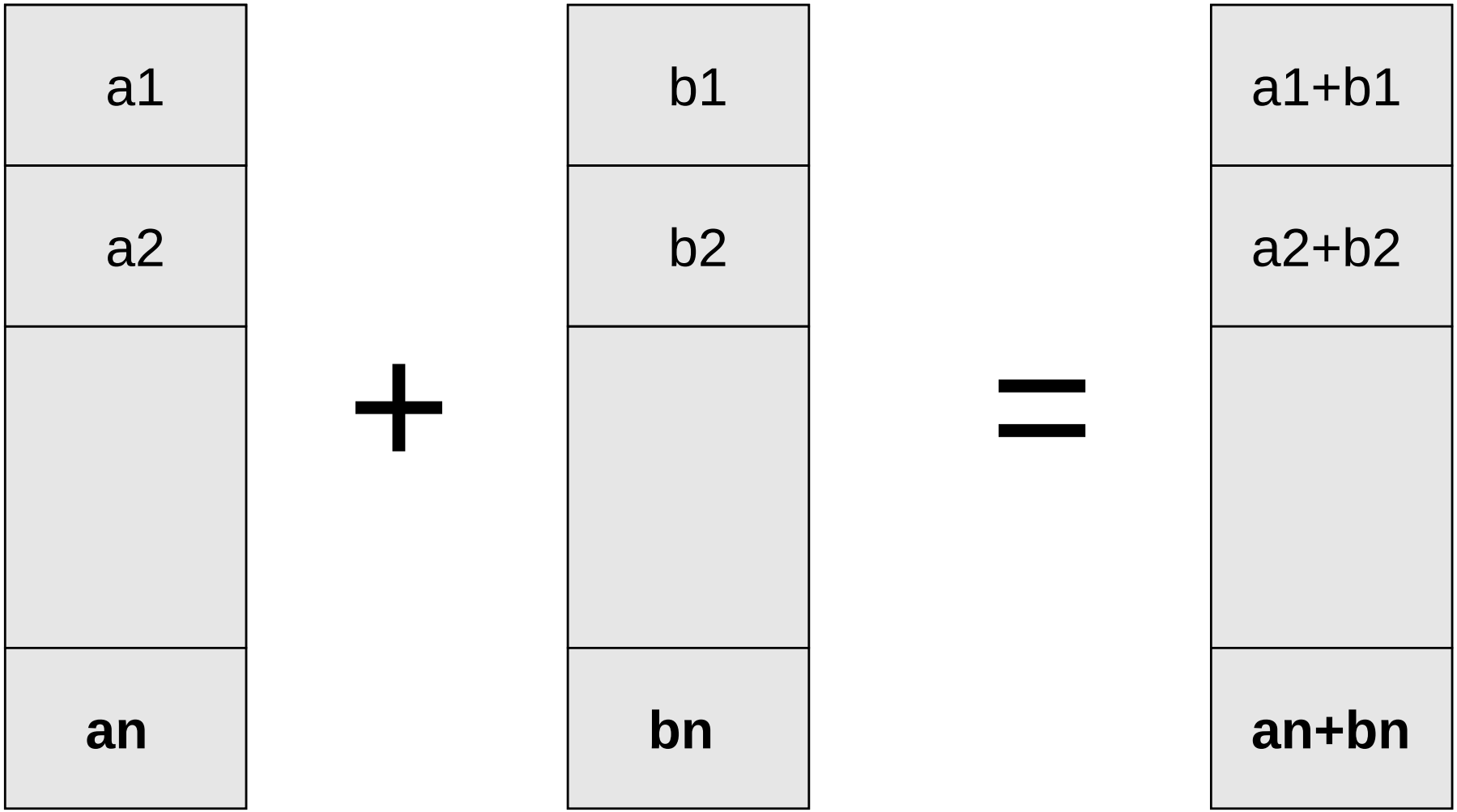
    c[2] = 1.0;
    System.out.println(a[2]); // Ausgabe: 1.0
}
```

Übersicht

Beispiel 5: Suchen in Arrays

```
boolean has(long[] a, long x) {
    int i;

    for (i = 0; i < a.length; i++) {
        if (a[i] == x) {
            break;
        }
    }

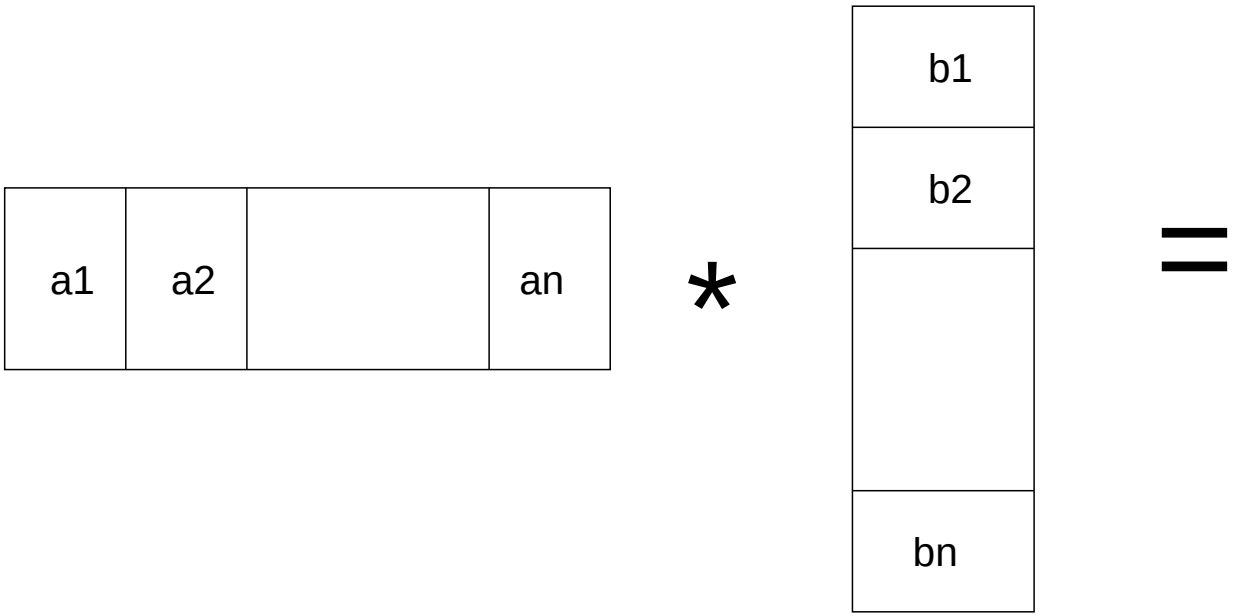
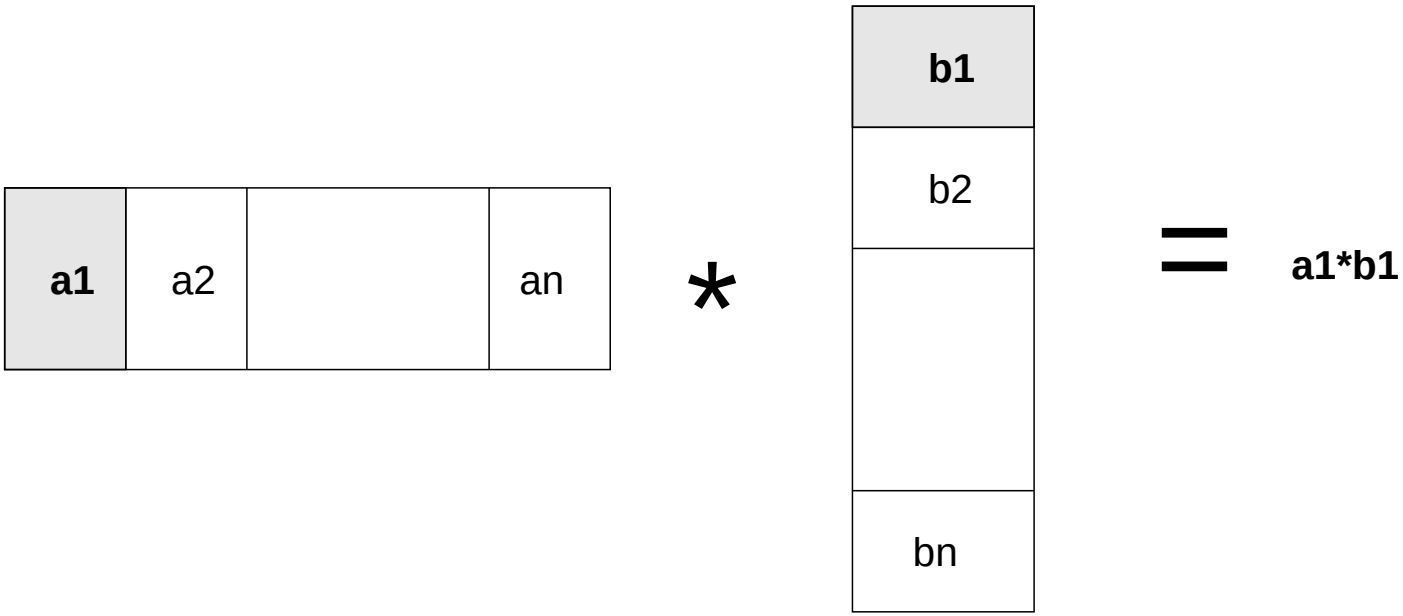
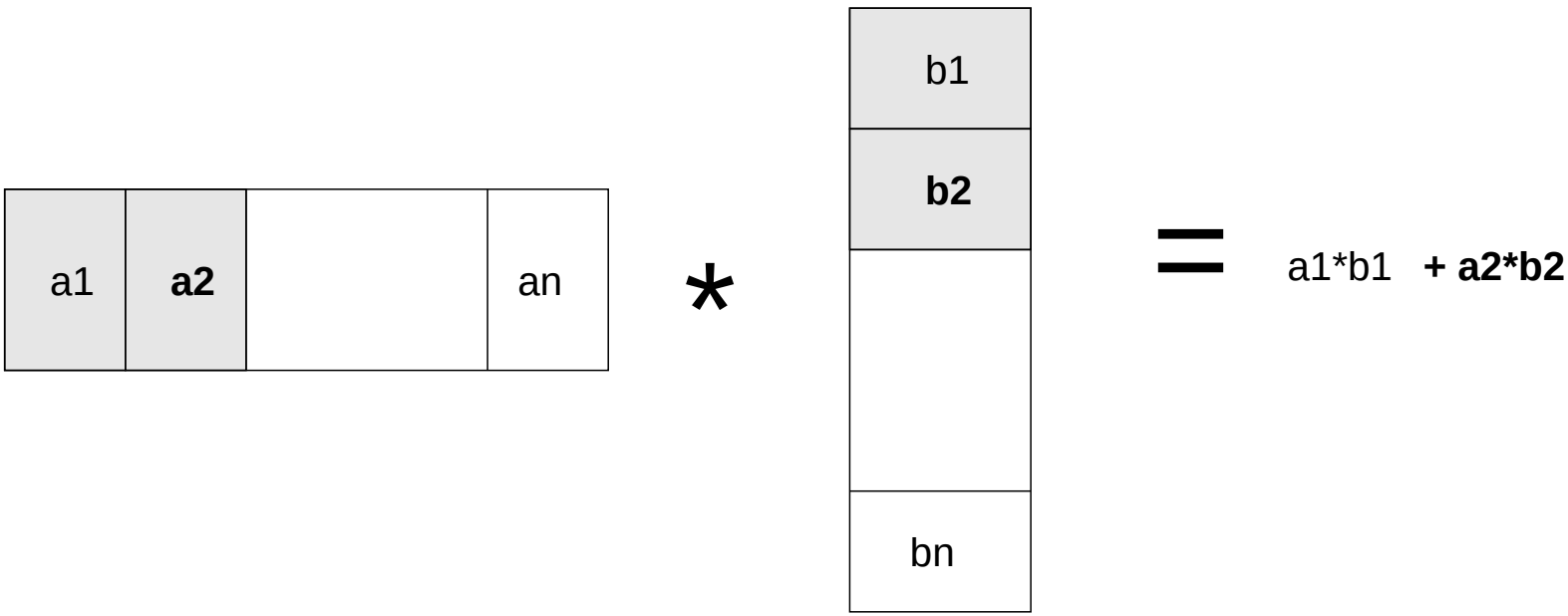
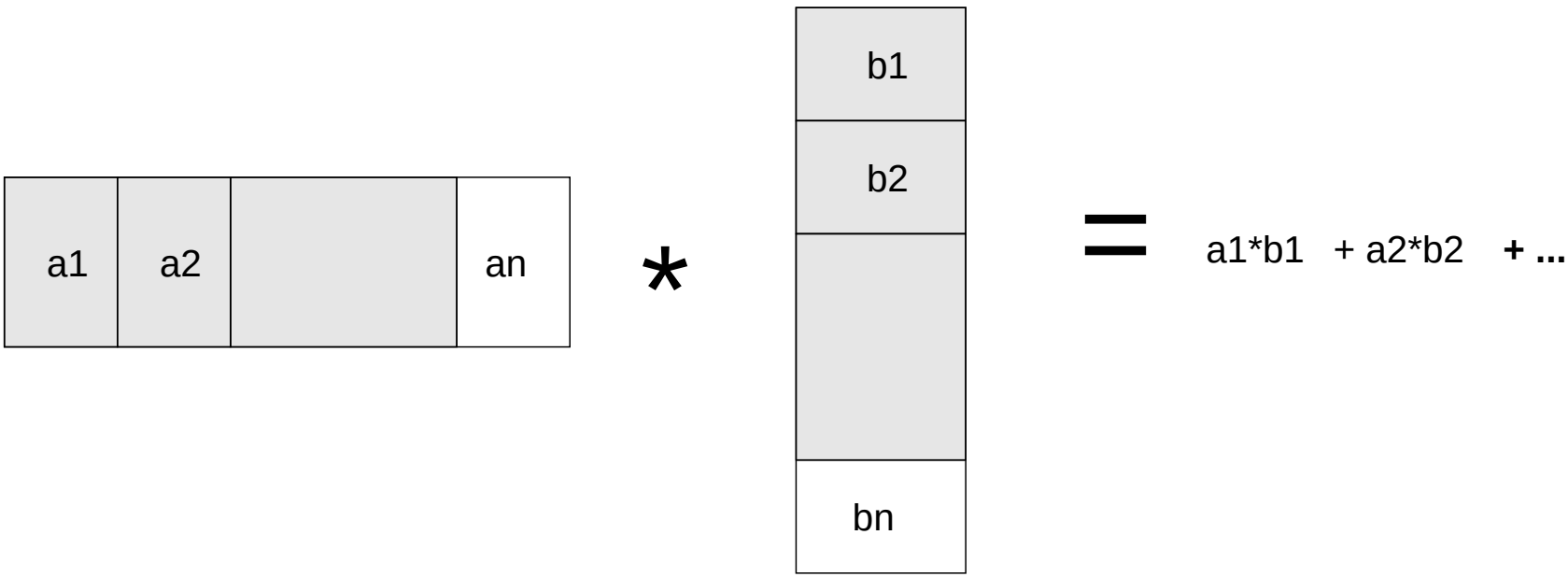
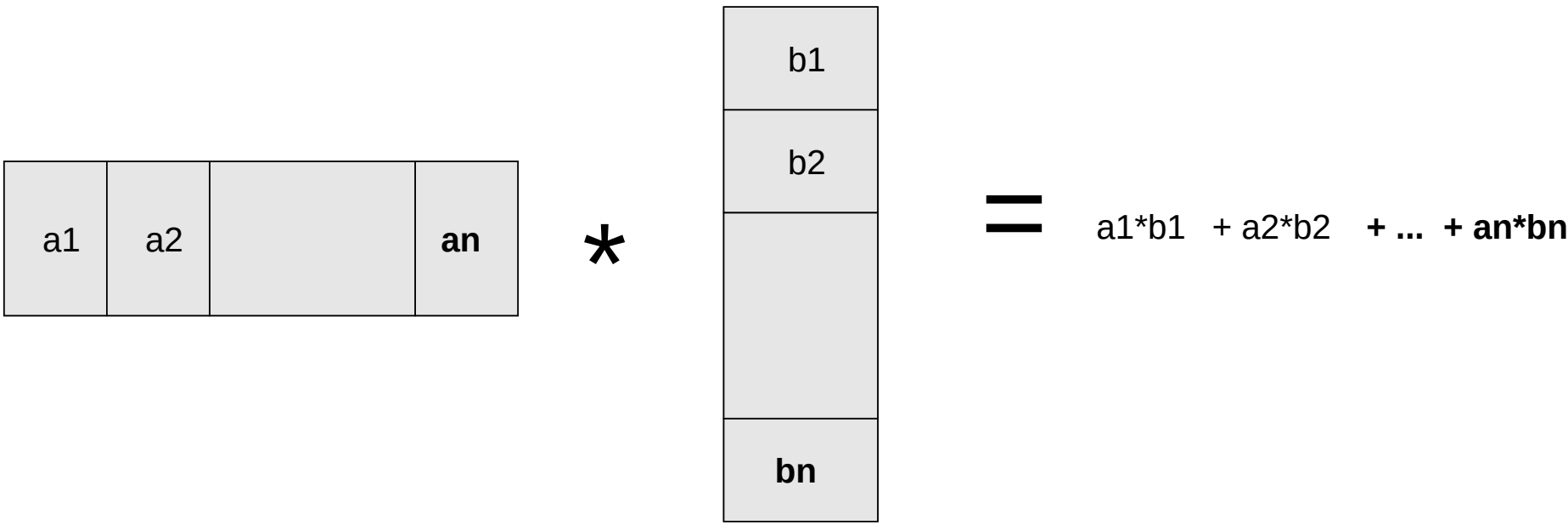
    return i != a.length;
}
```

Naive Suche



Naive Suche



Naive Suche



Naive Suche



Naive Suche



Naive Suche



Naive Suche



Beispiel 5: Suchen in Arrays

```
boolean has(long[] a, long x) {
    int i;

    for (i = 0; i < a.length; i++) {
        if (a[i] == x) {
            break;
        }
    }

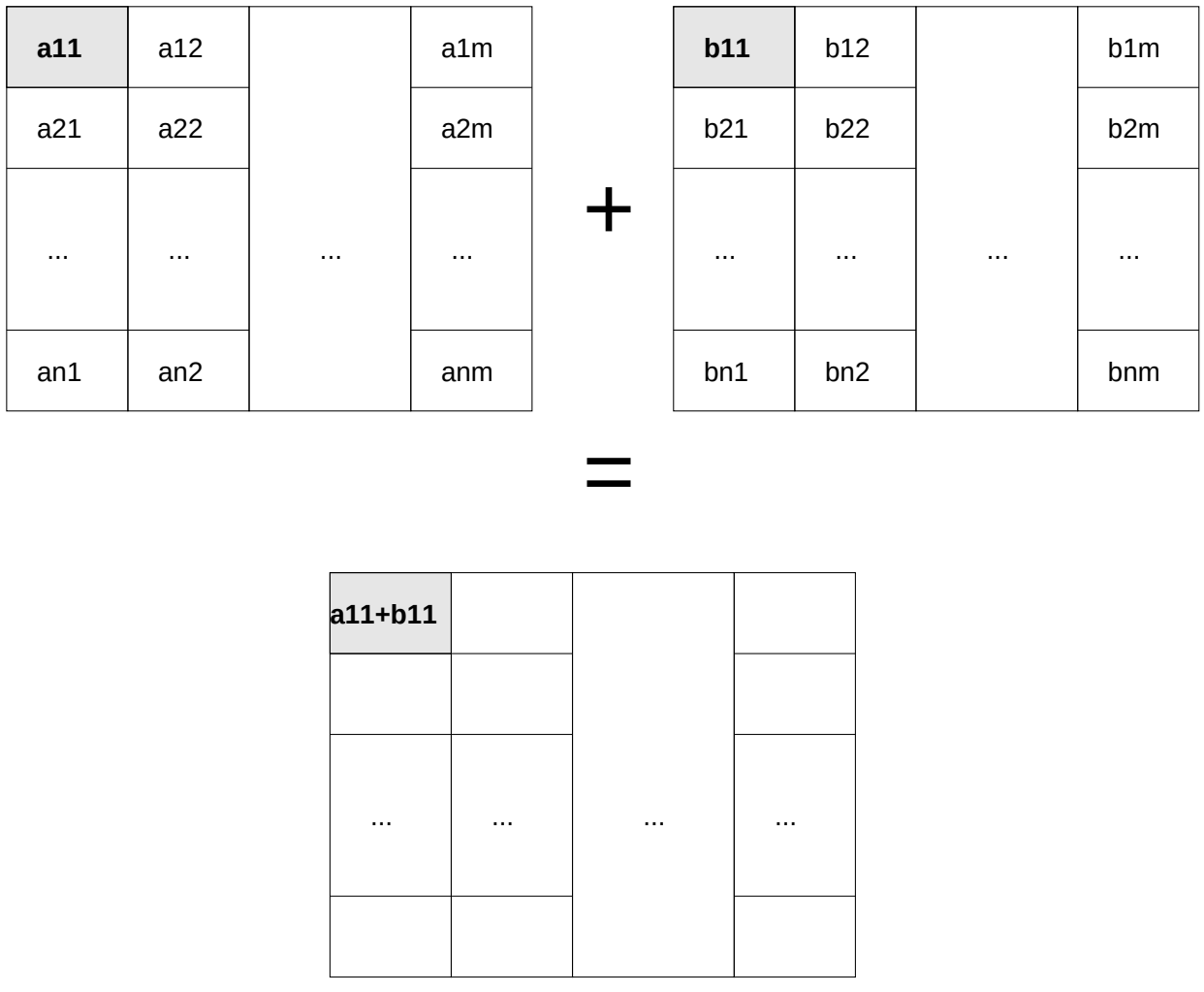
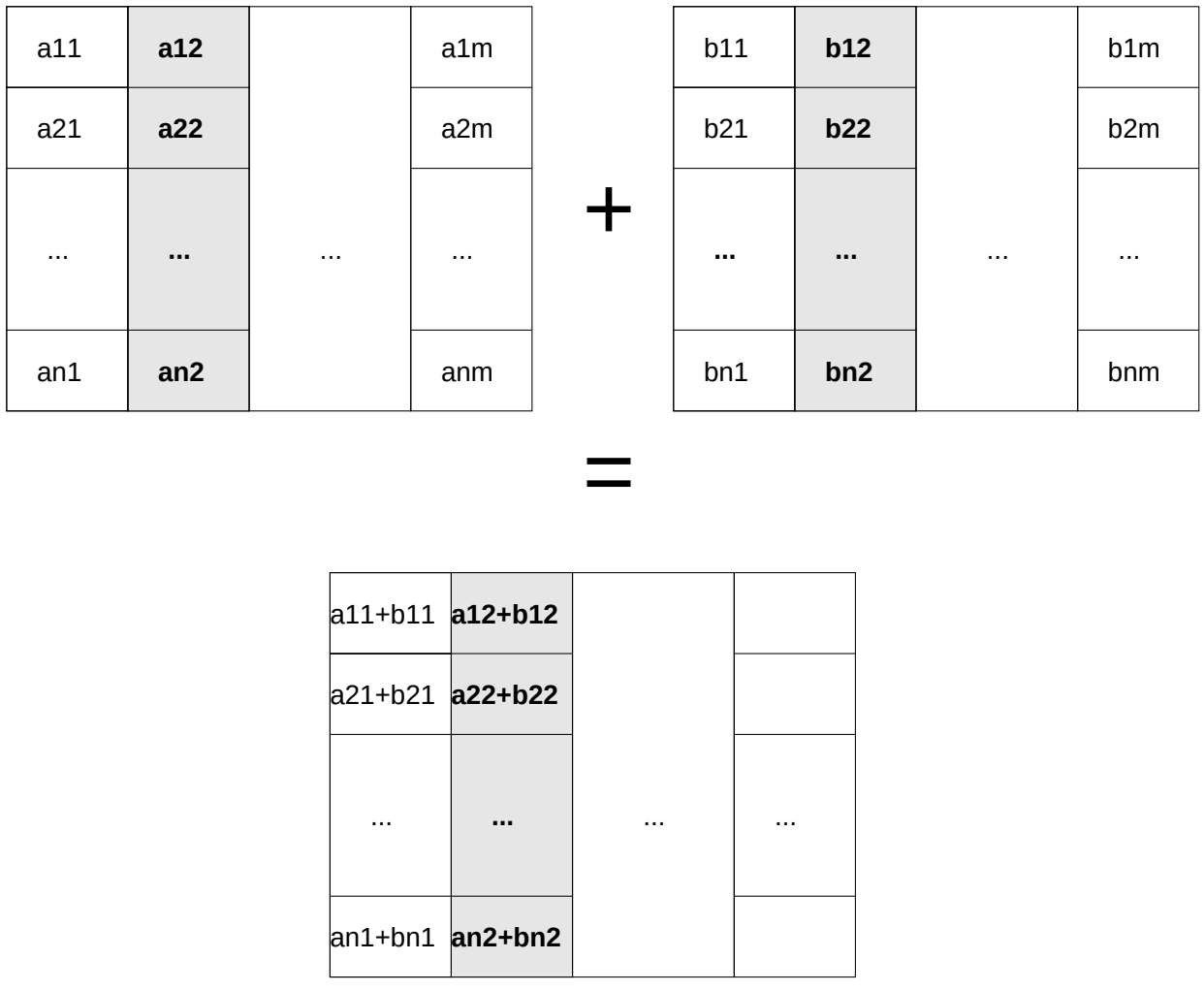
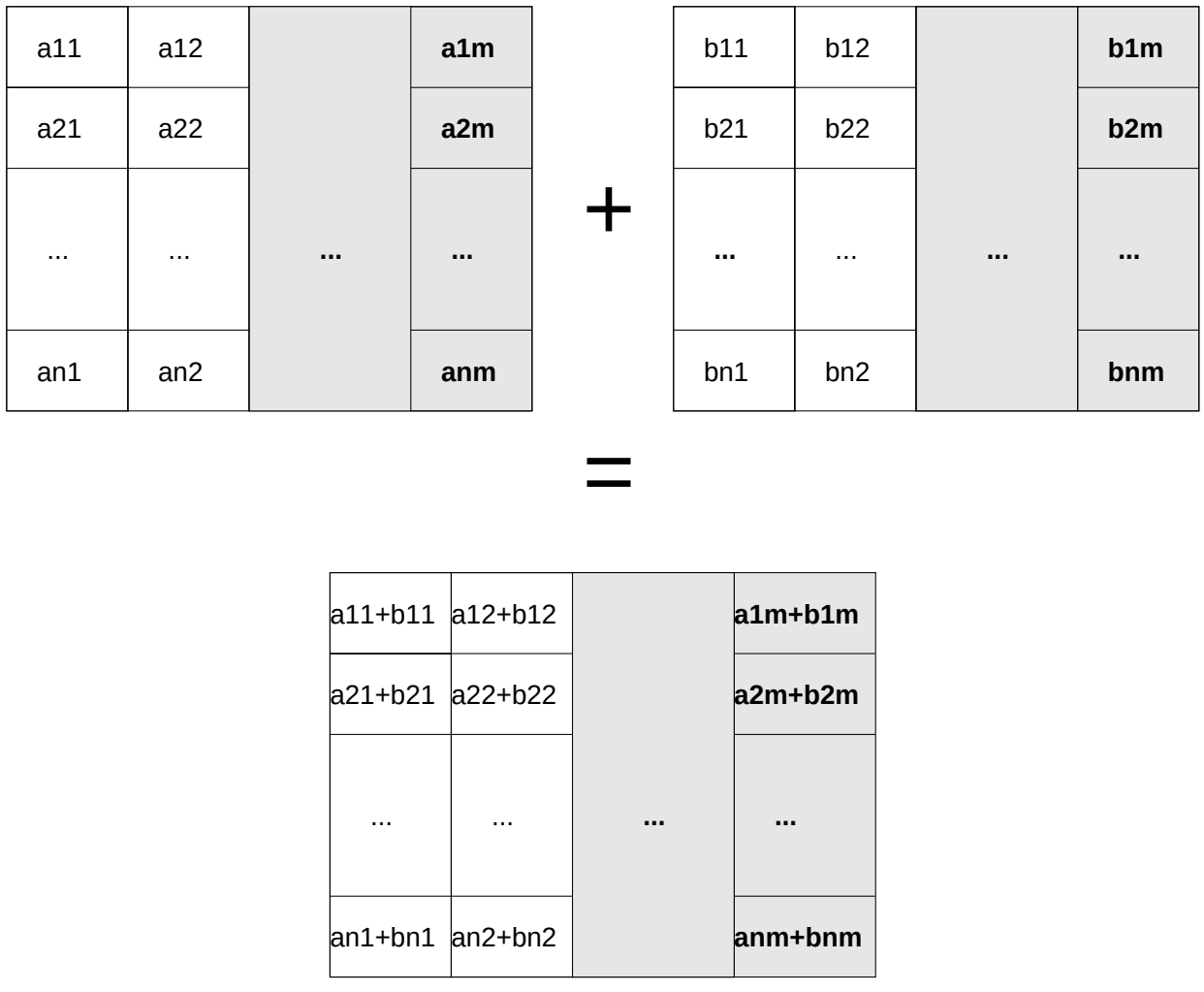
    return i != a.length;
}
```

- **Problem:** Schleife hat 2 Abbruchbedingungen

Beispiel 5: Suchen in Arrays

```
boolean has(long[] a, long x) {
    int i;
    for (i = 0; i < a.length && a[i] != x; i++);
    return i != a.length;
}
```

- **Nachteile:**
  - Verknüpfung zweier semantisch unabhänginger Abbruchbedingungen
  - leerer Schleifenrumpf $\rightarrow$ keinesfalls intuitiv
  - sehr schwer lesbar
  - sehr schwer wartbar

- Was passiert, wenn das Semikolon (`;`) vergessen wird?
- Kurzschlussauswertung von &&: Wenn der linke Operand falsch ist, wird der rechte nicht berechnet. Deshalb hier kein Laufzeitfehler!

Beispiel 5: Schöner Suchen in Arrays

```
boolean has(long[] a, long x) {
    int i = 0;
    boolean found = false;

    while (!found && i < a.length) {
        found = (a[i] == x);
        i++;
    }
    return found;
}
```

Später mehr!

Übersicht

Suchmaschine

Objekte identifizieren



Klassenentwurf



Objekterzeugung



Objektmethoder



Worthäufigkeiten



Suchmaschine: Inhalt von Dokumenten

- Häufigkeiten der Worte



Suchmaschine: Inhalt von Dokumenten

- Häufigkeiten der Worte



object	98
oriented	26
class	62
java	62
method	30
instance	3
compile	15

WordCountArray



object	146
oriented	0
class	126
java	24
method	112
instance	81
compile	6

WordCountArray

Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



allWords:

object	0
oriented	0
class	0
java	0
method	0
instance	0
platform	0
version	0
variable	0

Suchmaschine: Vektoren vergleichbar machen



allWords:

object	0
oriented	0
class	0
java	0
method	0
instance	0
platform	0
version	0
variable	0
final	0

Suchmaschine: Vektoren vergleichbar machen



allWords:

object	0
oriented	0
class	0
java	0
method	0
instance	0
platform	0
version	0
variable	0
final	0
equals	0

Suchmaschine: Vektoren vergleichbar machen



allWords:

object	0
oriented	0
class	0
java	0
method	0
instance	0
platform	0
version	0
variable	0
final	0
equals	0
void	0
...	

Suchmaschine: Vektoren vergleichbar machen



allWords:

object	0
oriented	0
class	0
java	0
method	0
instance	0
platform	0
version	0
variable	0
final	0
equals	0
void	0
...	

Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Suchmaschine: Vektoren vergleichbar machen



Übersicht

Beispiel 6: Sortieren

Gegeben: eine Folge von ganzen Zahlen.
Gesucht: die zugehörige aufsteigend sortierte Folge.

Beispiel 6: 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!

Beispiel 6: 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 ...

Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen



Einfügen

17	3	−2	9	0	1	7	42	5

−2	0	1	3	5	7	9	17	42

Sortieren: Code

```
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 ???

Einfügen: Code

```
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?

Finden und Verschieben: Code

```
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];
}
```

Finden und Verschieben: Code

```
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];
}
```

Achtung:
1. Auch hier wird das zweite Argument des Operators `&&` in `locate` nur ausgewertet, sofern das erste `true` ergibt (Kurzschluss-Auswertung). Sonst würde hier ggf. auf eine uninitialisierte Variable zugegriffen.

Finden und Verschieben: Code

```
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];
}
```

Achtung:

1. Auch hier wird das zweite Argument des Operators `&&` in `locate` nur ausgewertet, sofern das erste `true` ergibt (Kurzschluss-Auswertung). Sonst würde hier ggf. auf eine uninitialisierte Variable zugegriffen.
2. Warum laufen wir in `shift()` abwärts von `i-1` nach `j` ?

Erläuterungen

- 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 ...

Also ...

```
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).

Mehrdimensionale Felder

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



Übersicht

Vektoren und Matrizen – Beispiele

- Vektoraddition



Vektoren und Matrizen – Beispiele

- Vektoraddition



Vektoren und Matrizen – Beispiele

- Vektoraddition



Vektoren und Matrizen – Beispiele

- Vektoraddition



Vektoren und Matrizen – Beispiele

- Vektoraddition



Vektoren und Matrizen – Beispiele

- Vektoraddition – Algorithmus

```
// Vorbedingung: a.length == b.length
public static int[] addVectors(int[] a, int[] b) {
    int[] c = new int[a.length];

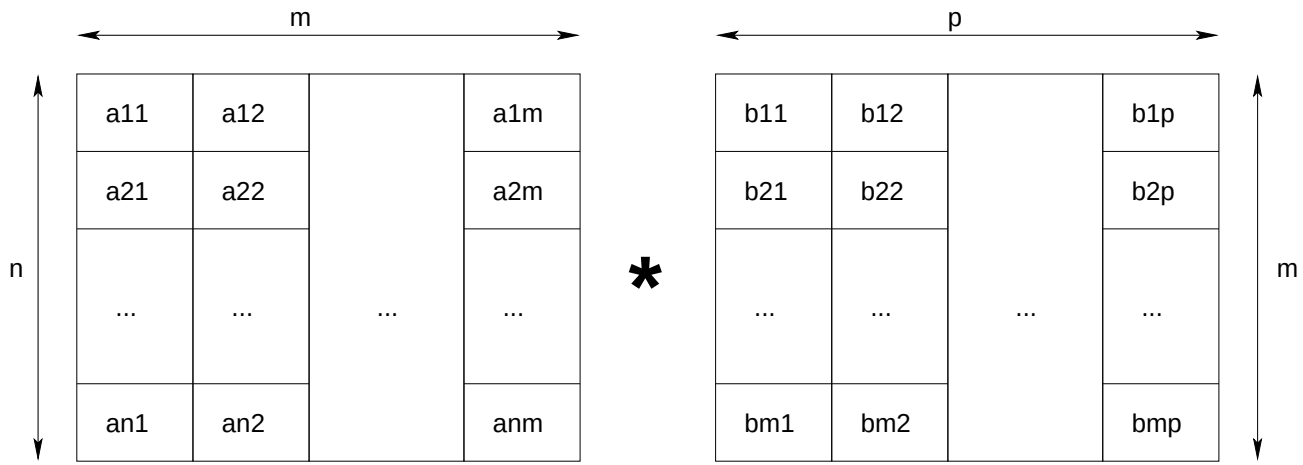
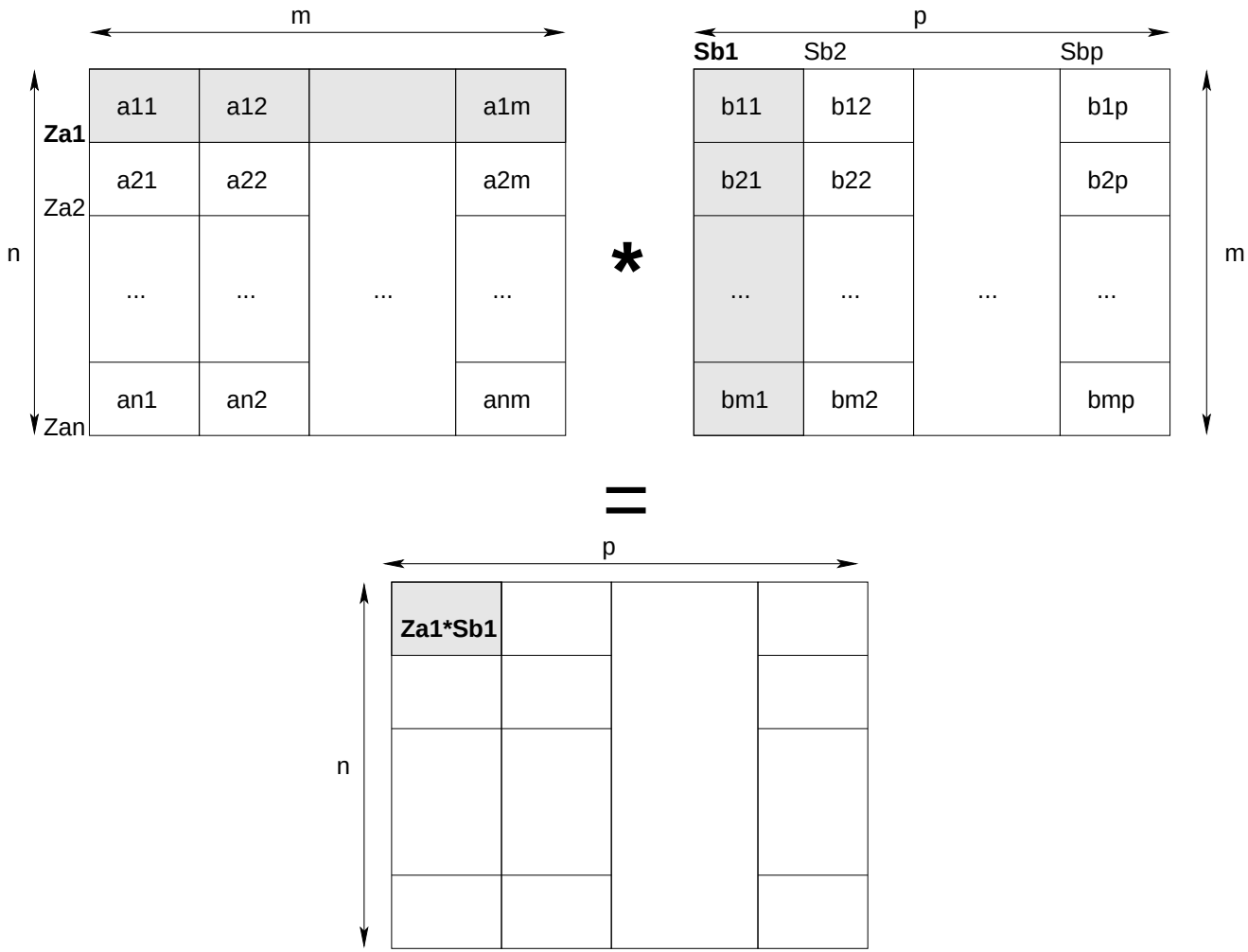
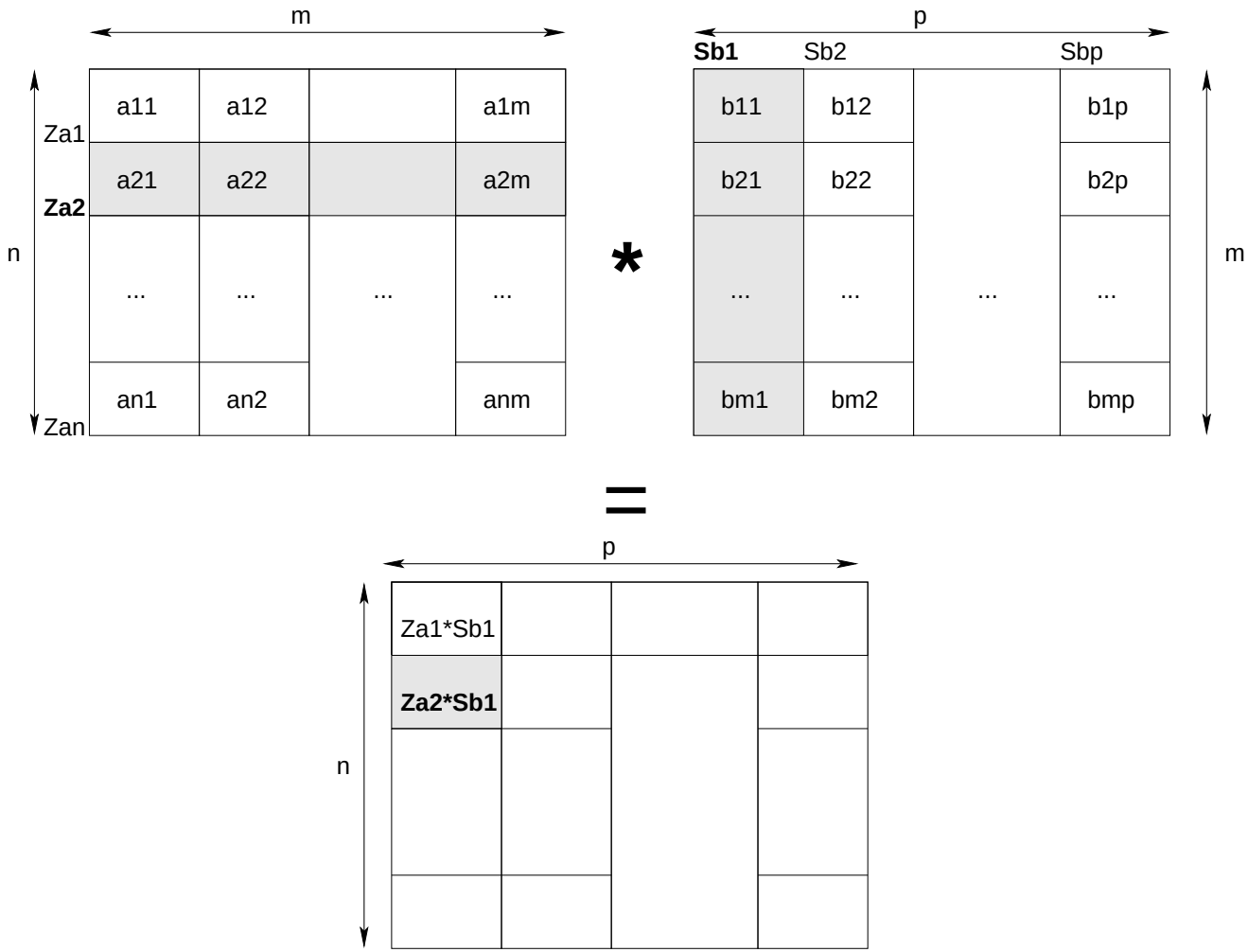
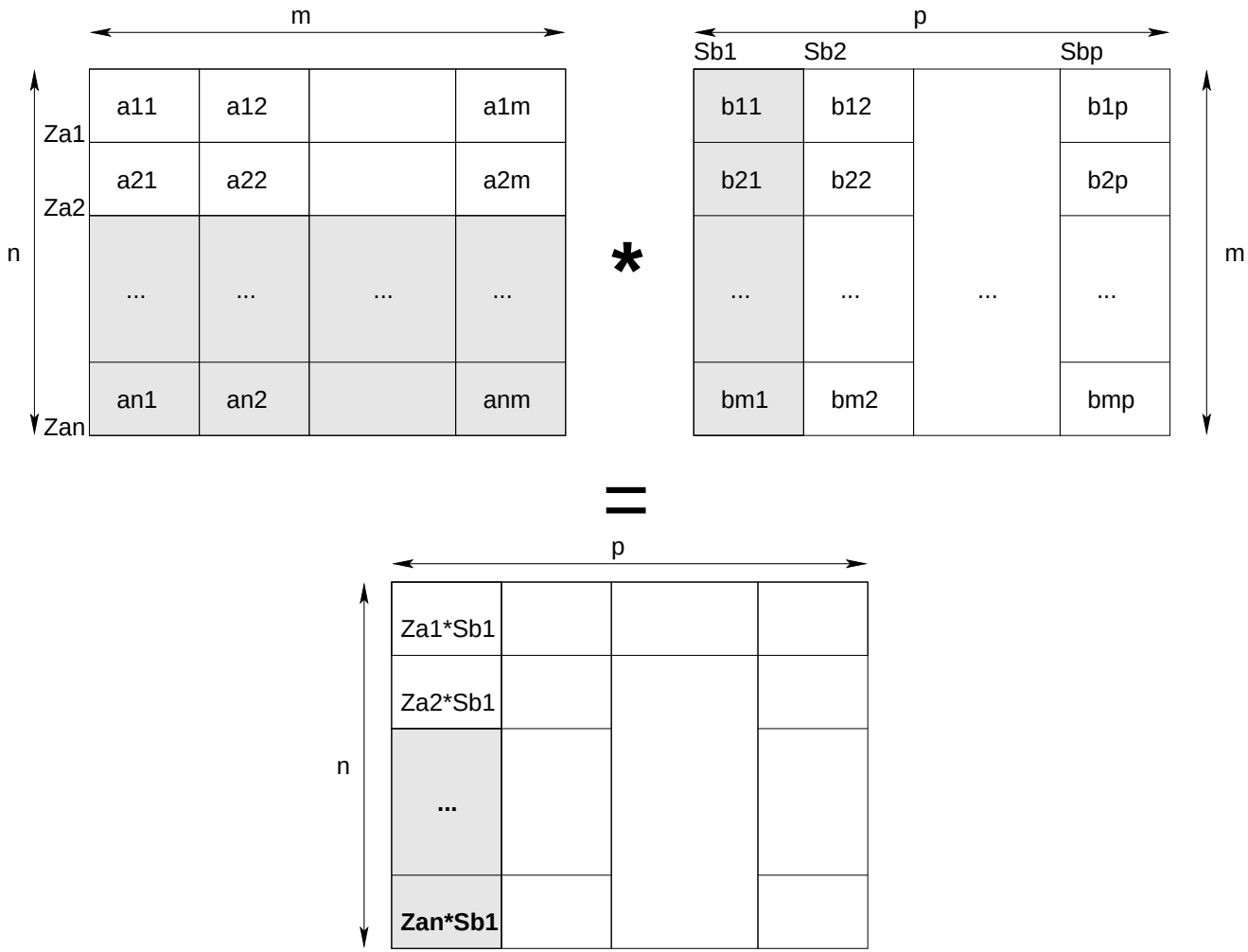
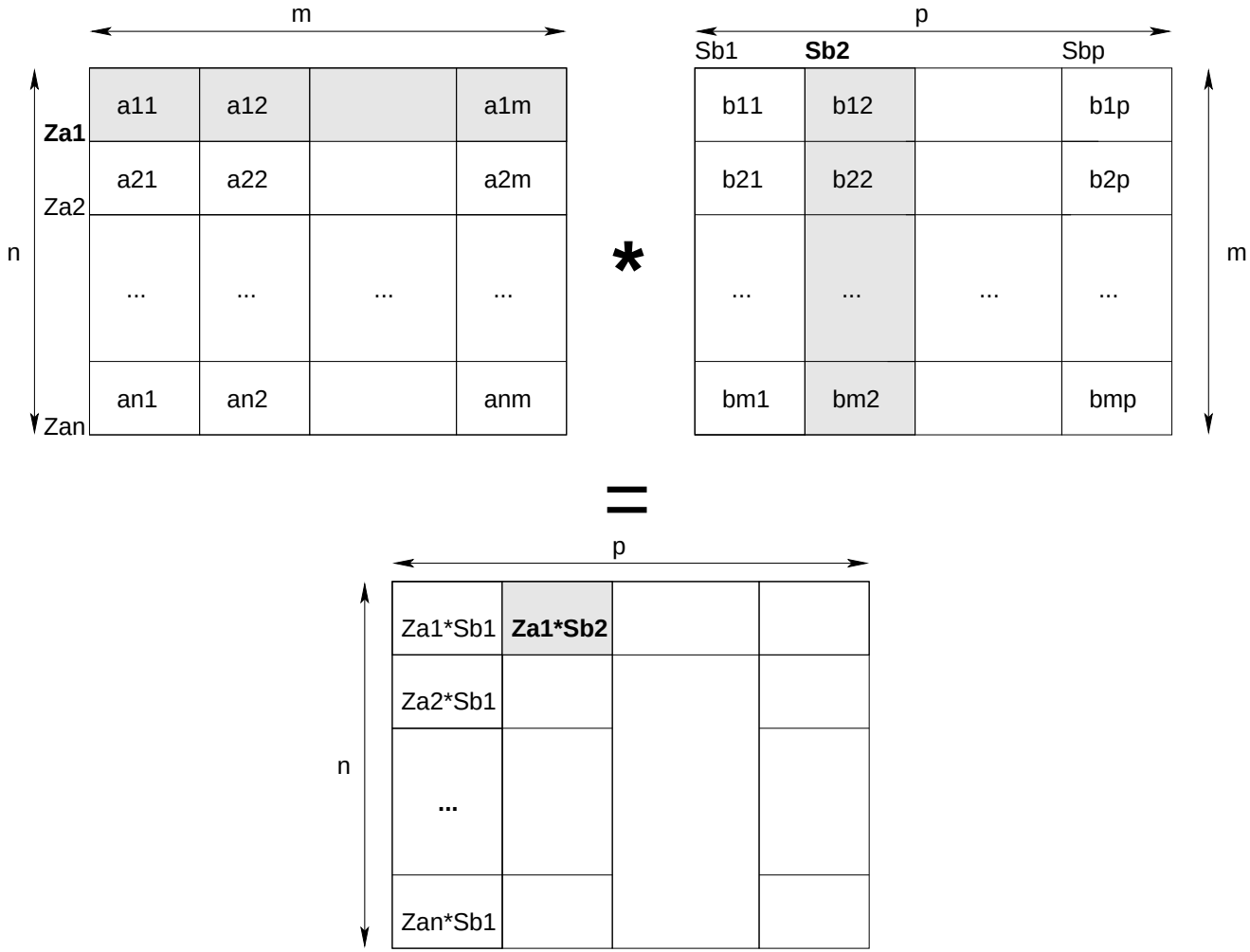
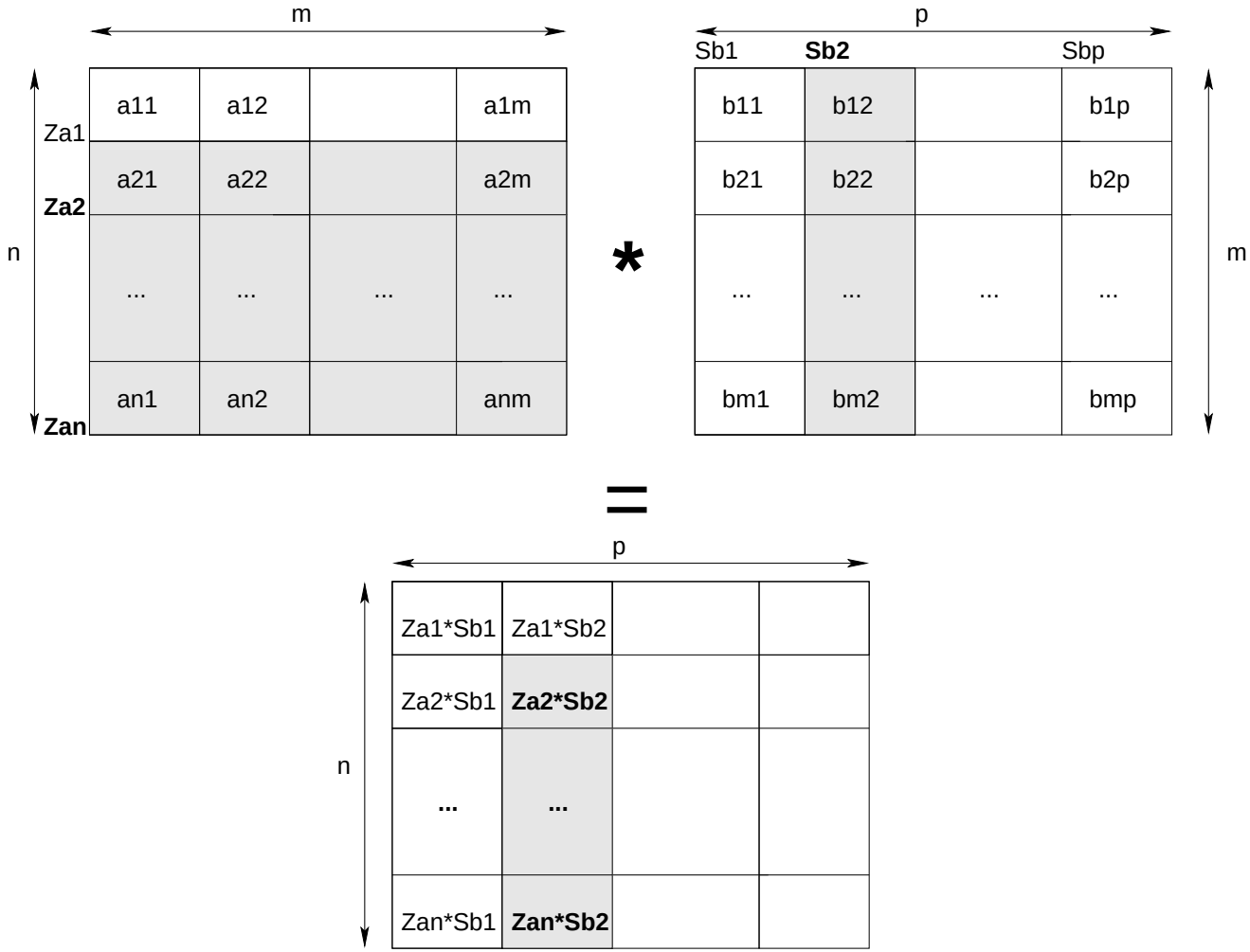
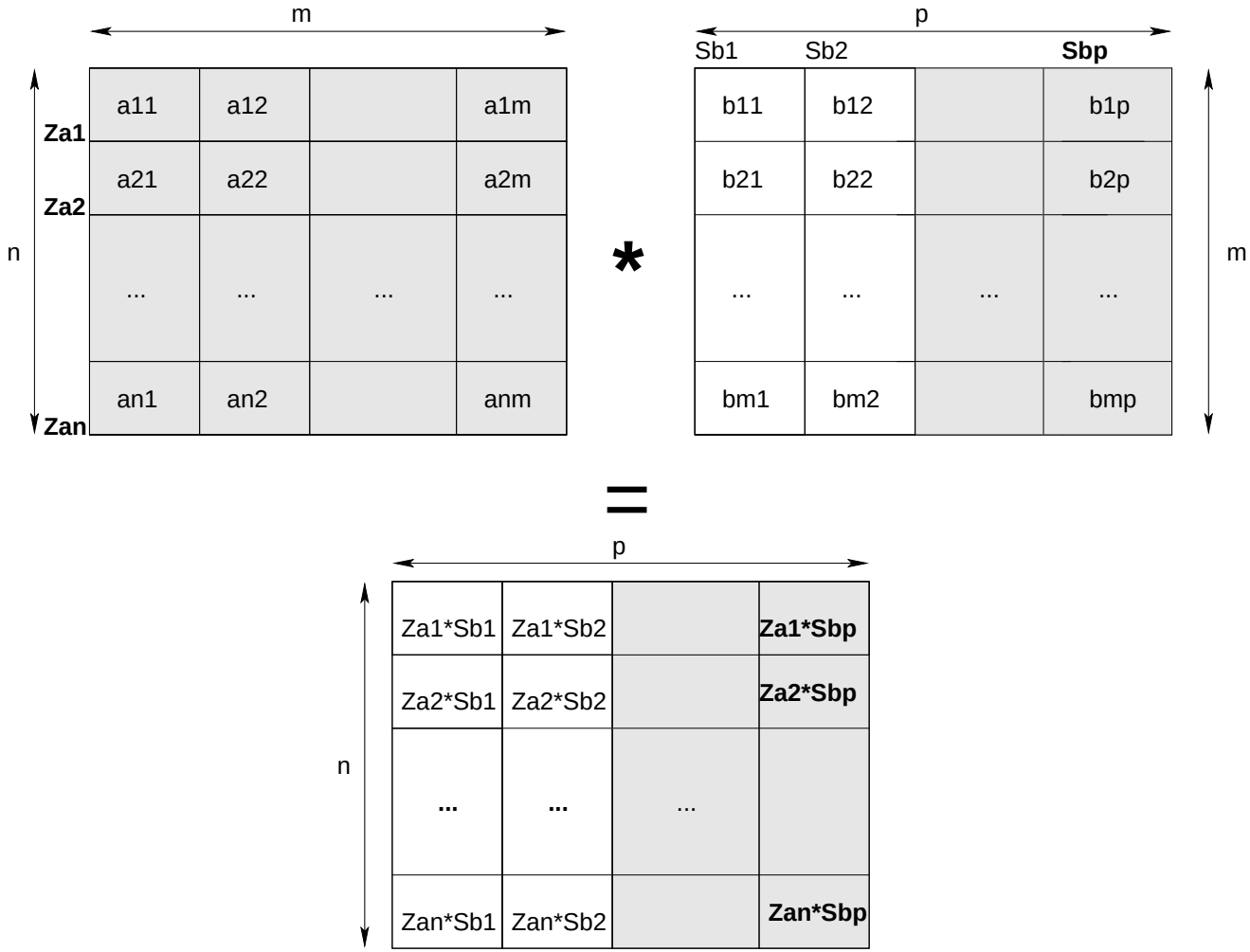
    for (int i = 0; i < a.length; i++) {
        c[i] = a[i] + b[i];
    }

    return c;
}
```

Vektoren und Matrizen – Beispiele

- Vektormultiplikation (Skalarprodukt)



Vektoren und Matrizen – Beispiele

- Vektormultiplikation (Skalarprodukt)



Vektoren und Matrizen – Beispiele

- Vektormultiplikation (Skalarprodukt)



Vektoren und Matrizen – Beispiele

- Vektormultiplikation (Skalarprodukt)



Vektoren und Matrizen – Beispiele

- Vektormultiplikation (Skalarprodukt)



Vektoren und Matrizen – Beispiele

- Vektormultiplikation – Algorithmus

```
// Vorbedingung: a.length == b.length
public static int multVectors(int[] a, int[] b) {
    int c = 0;

    for (int i = 0; i < a.length; i++) {
        c += a[i] * b[i];
    }

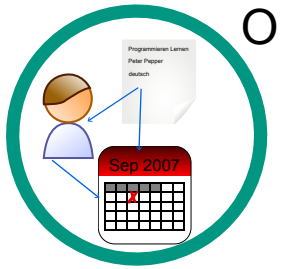
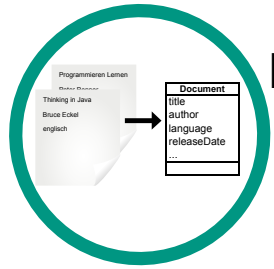
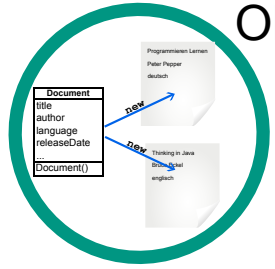
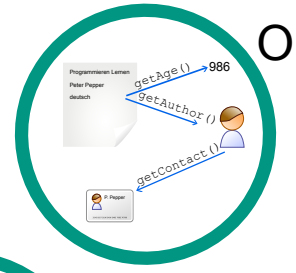
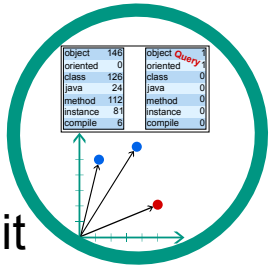
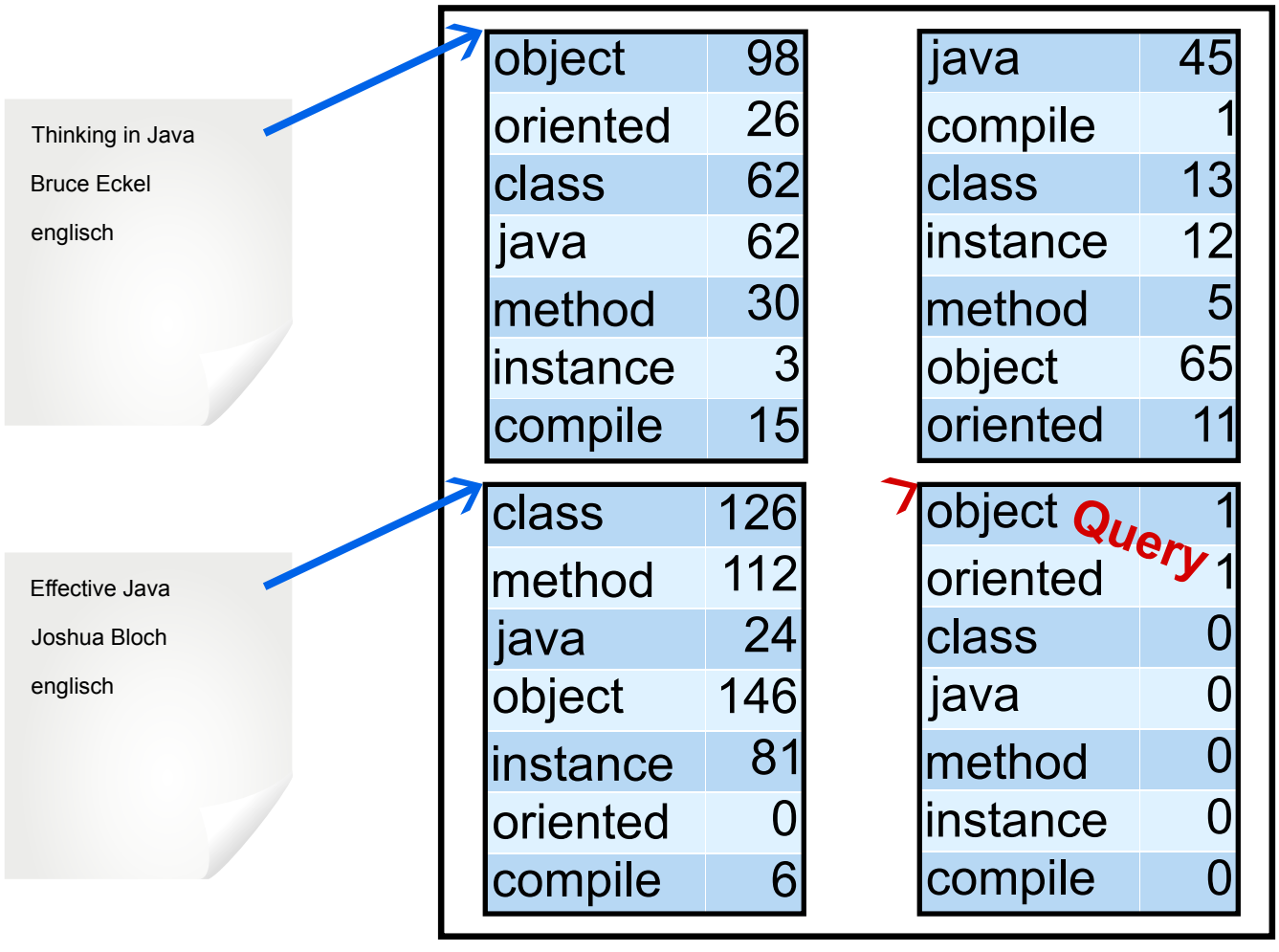
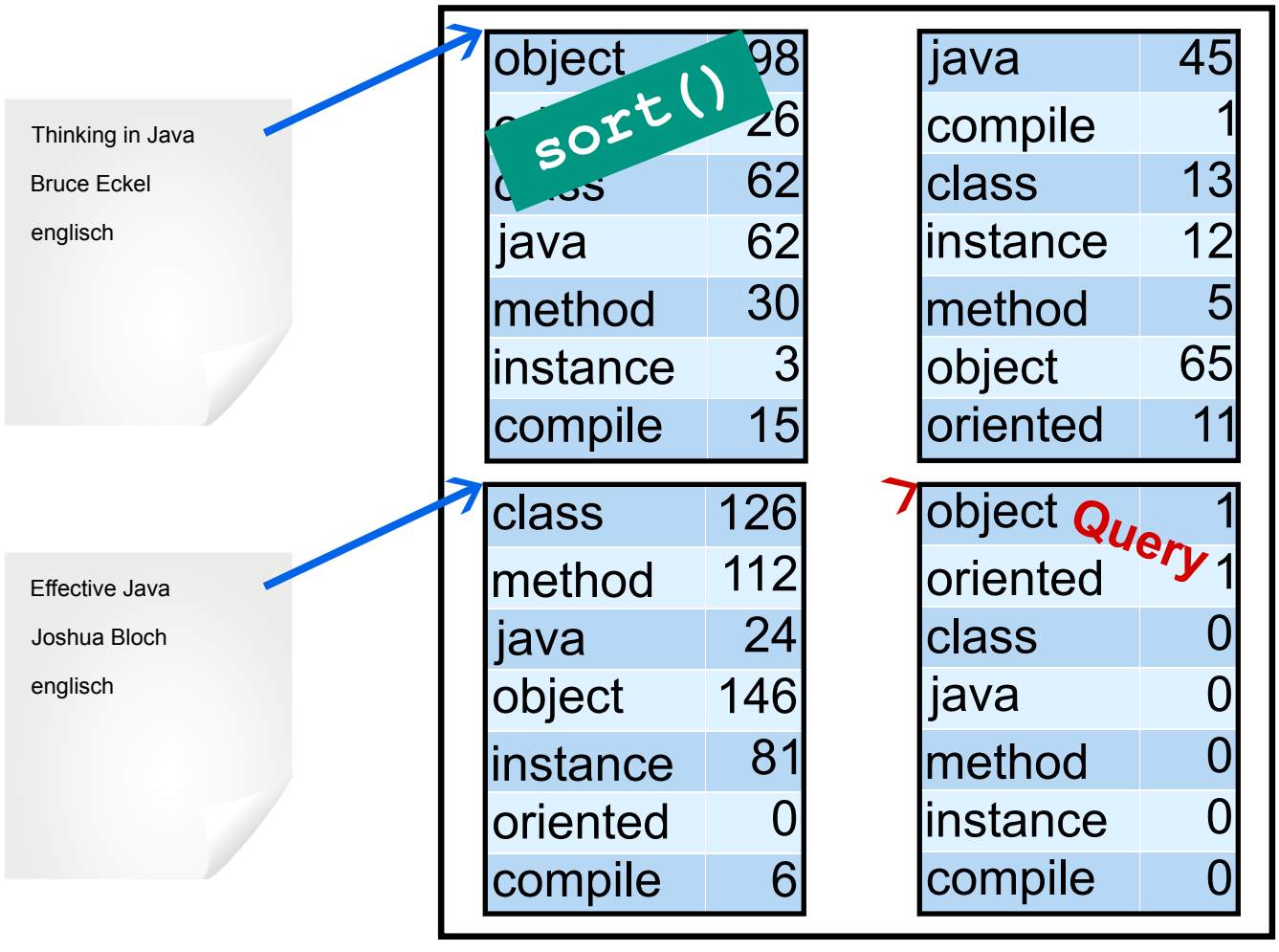
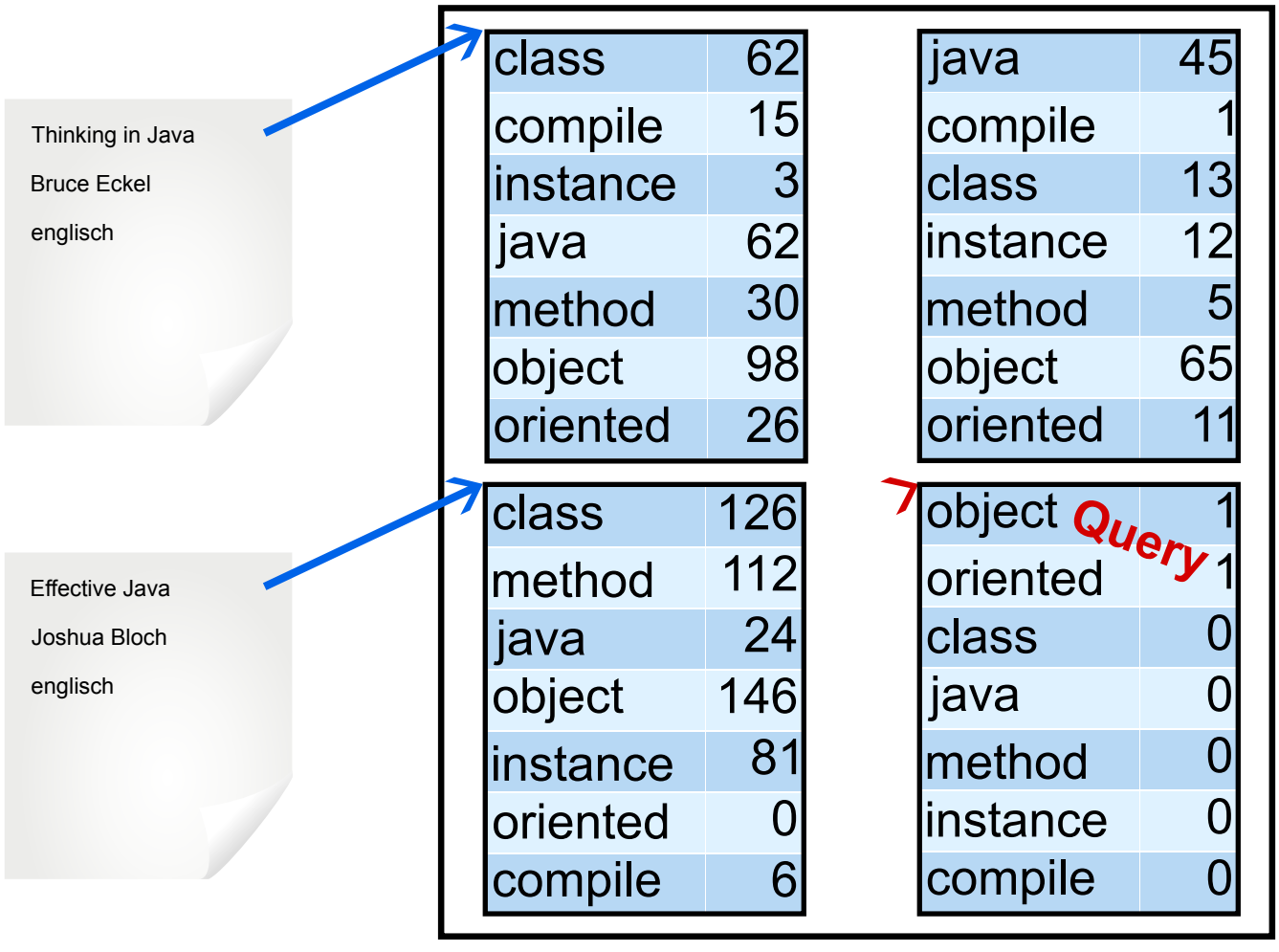
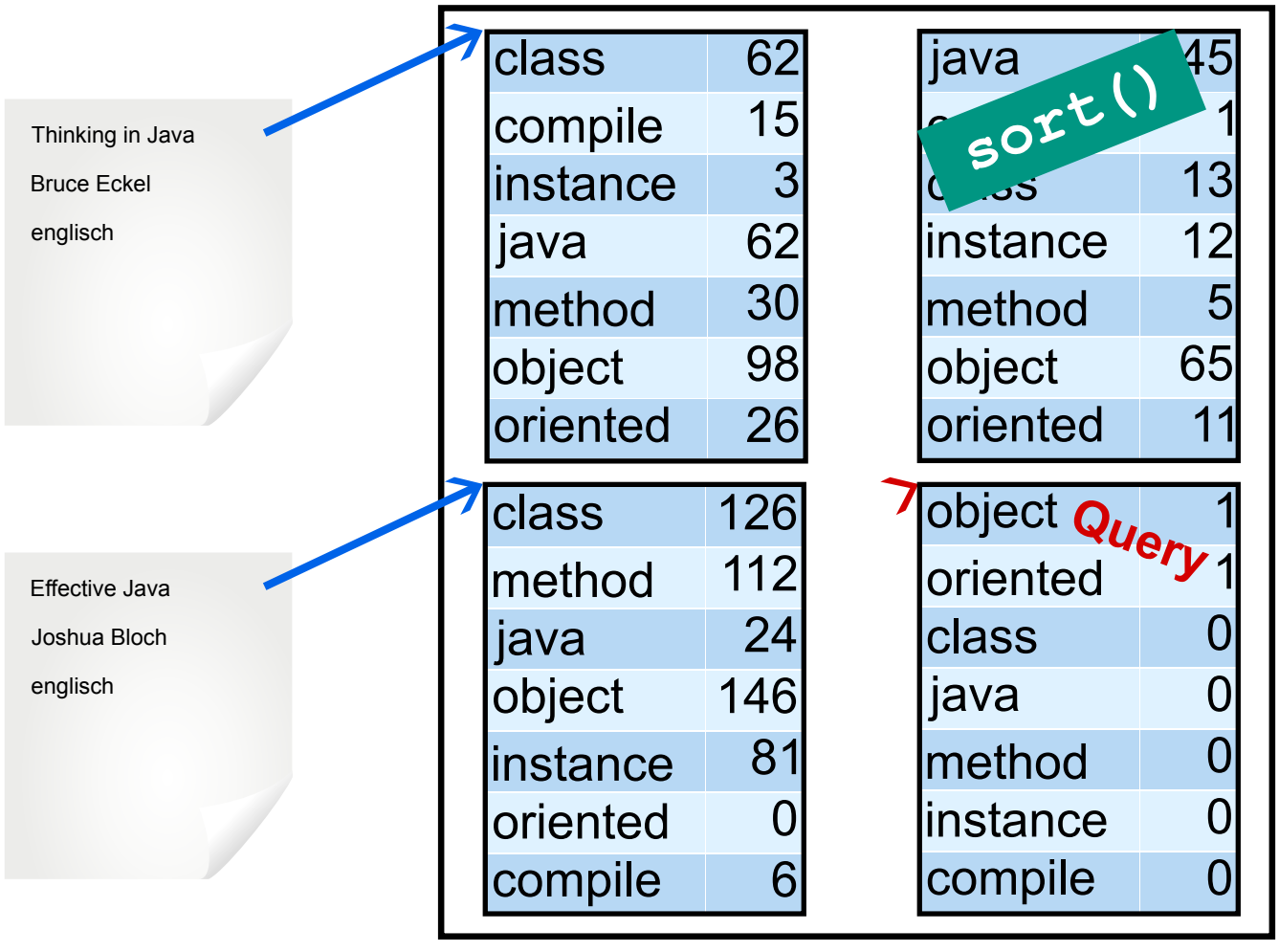
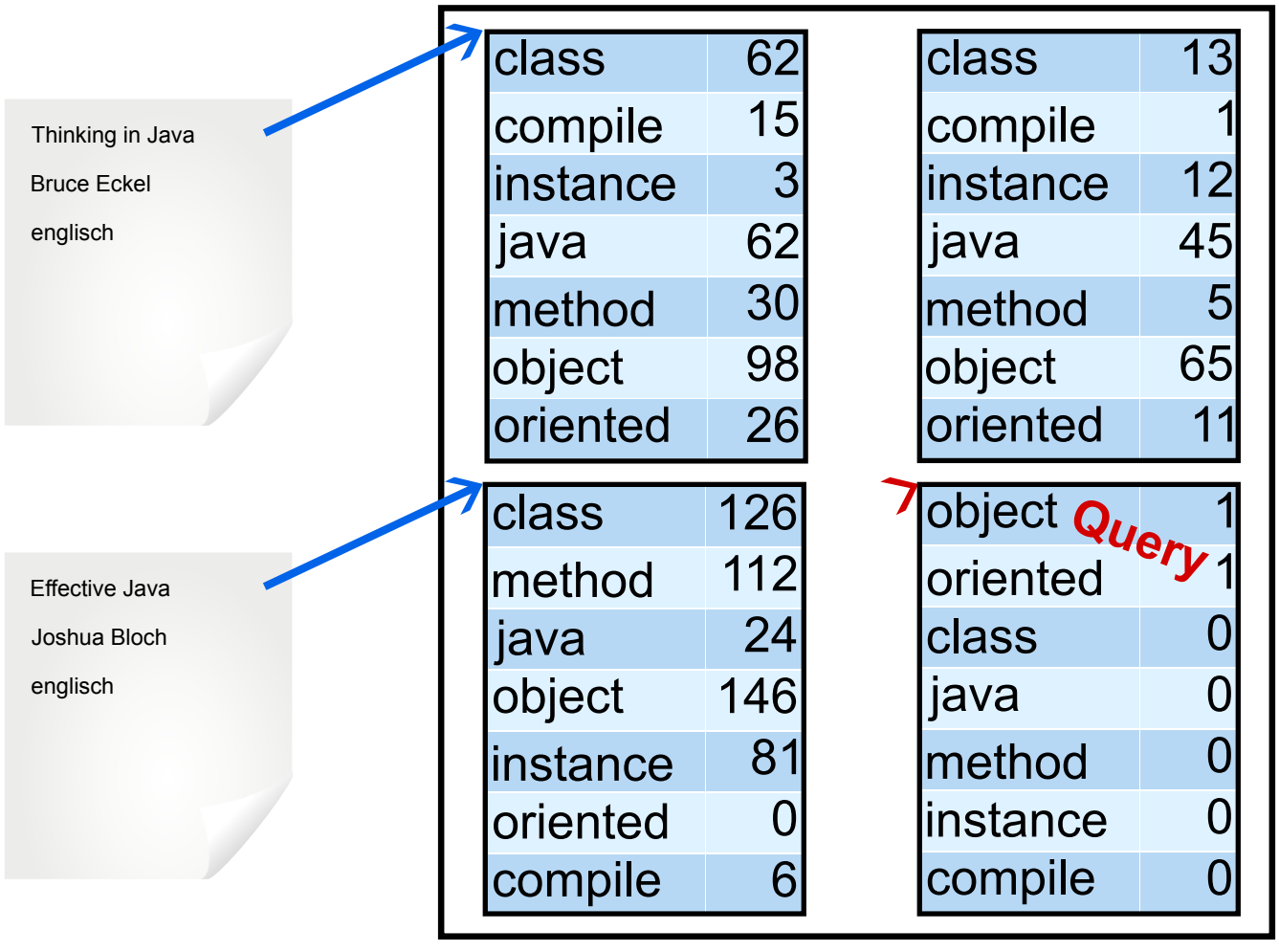
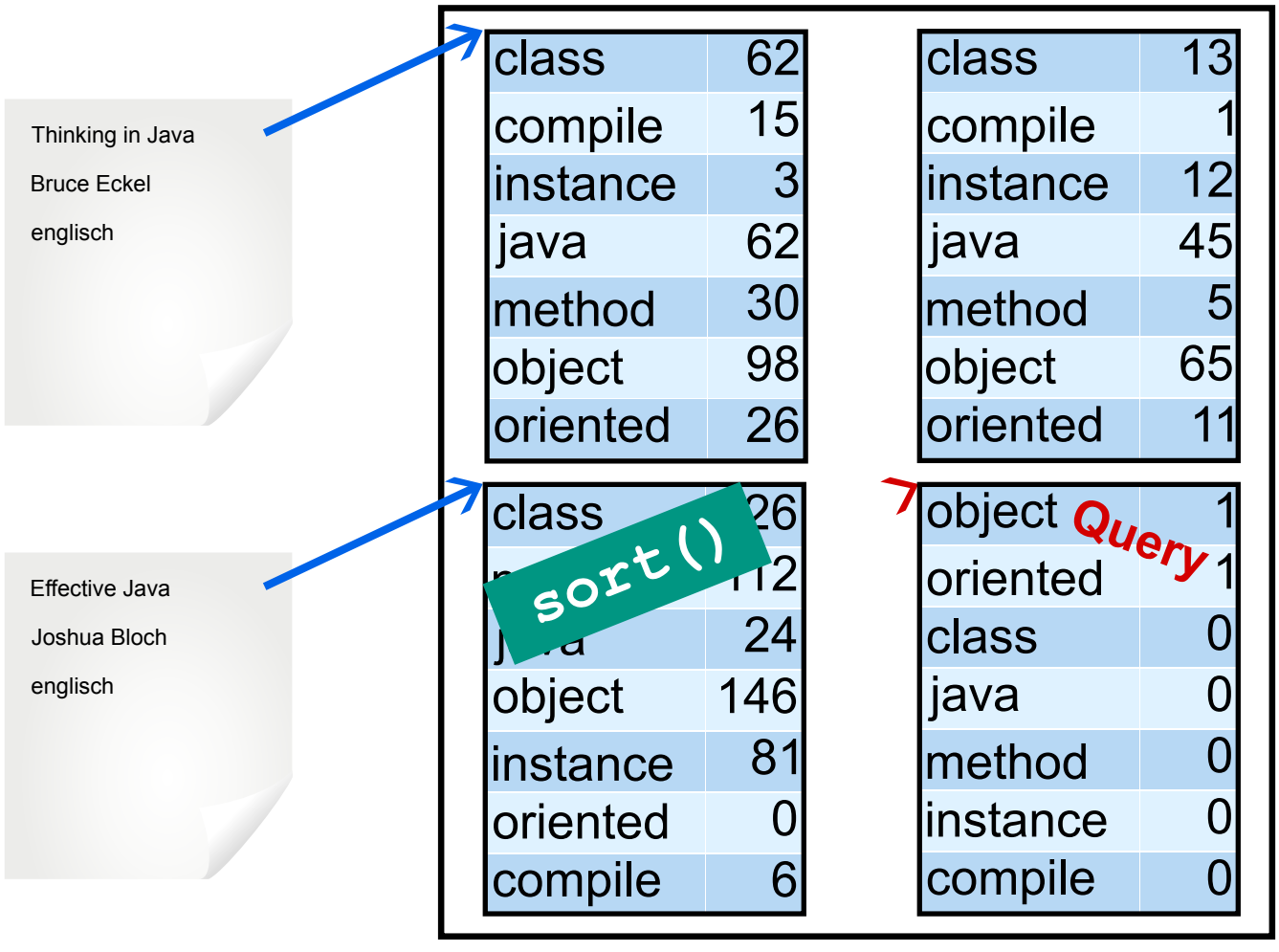
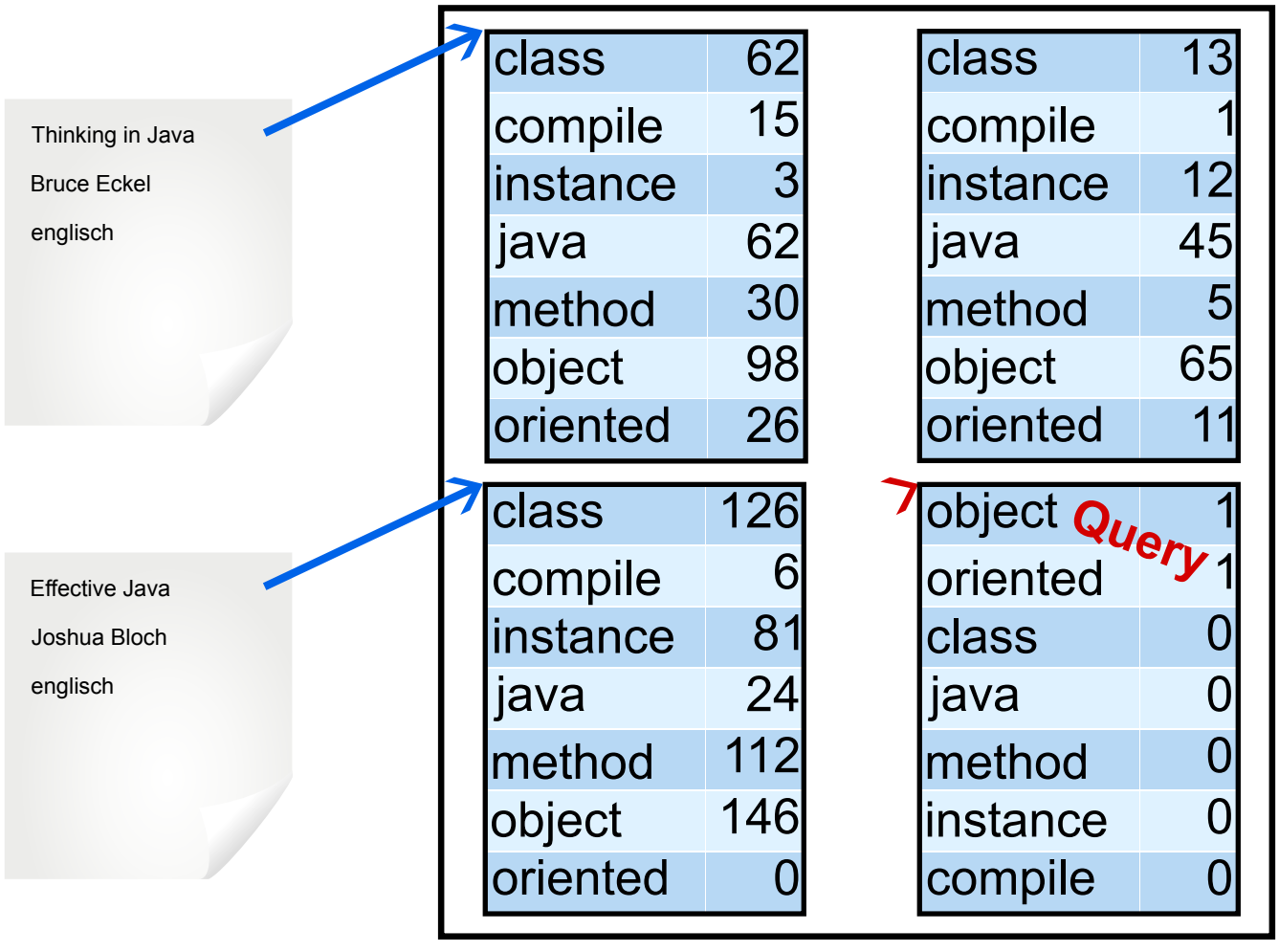
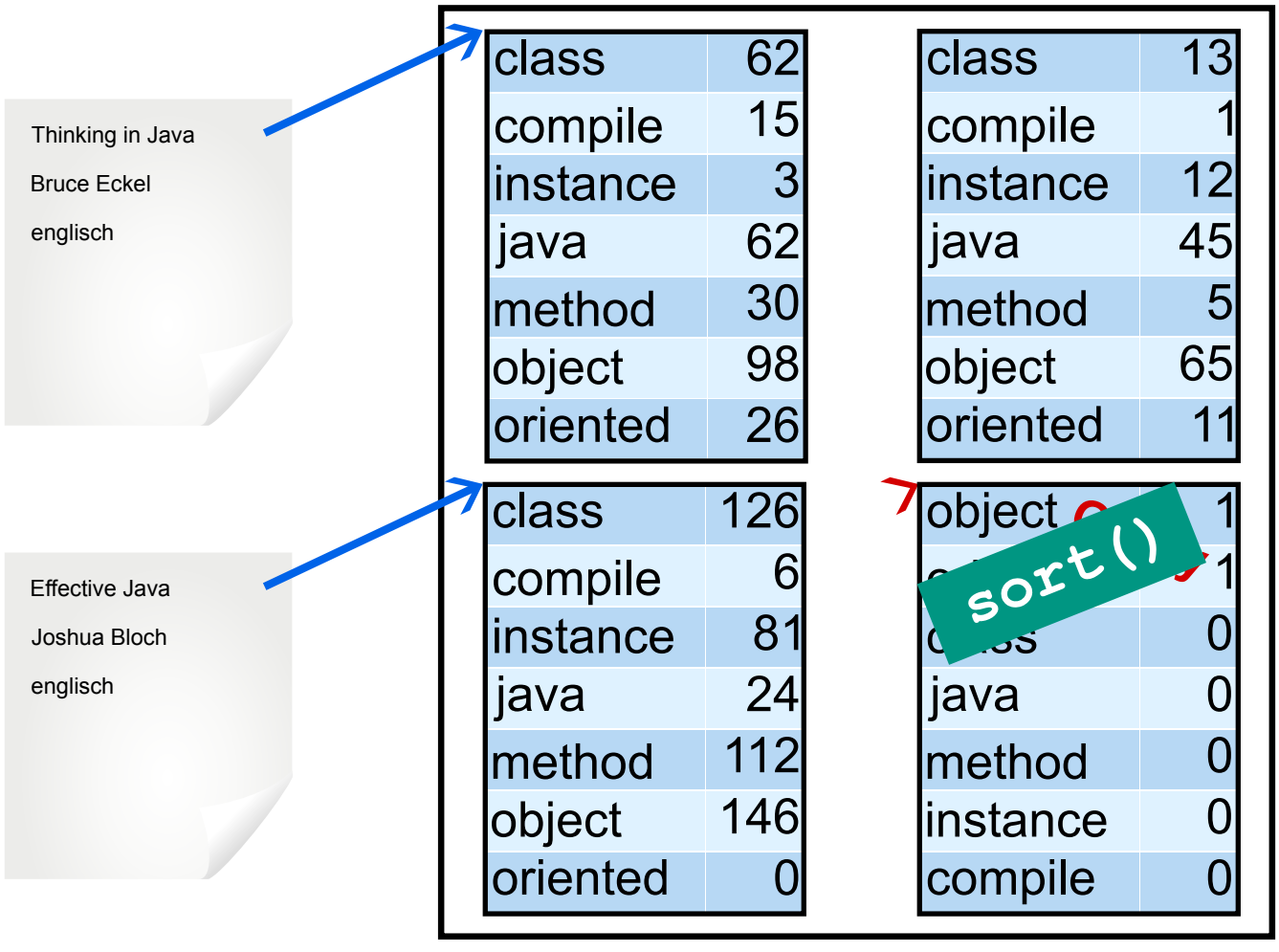
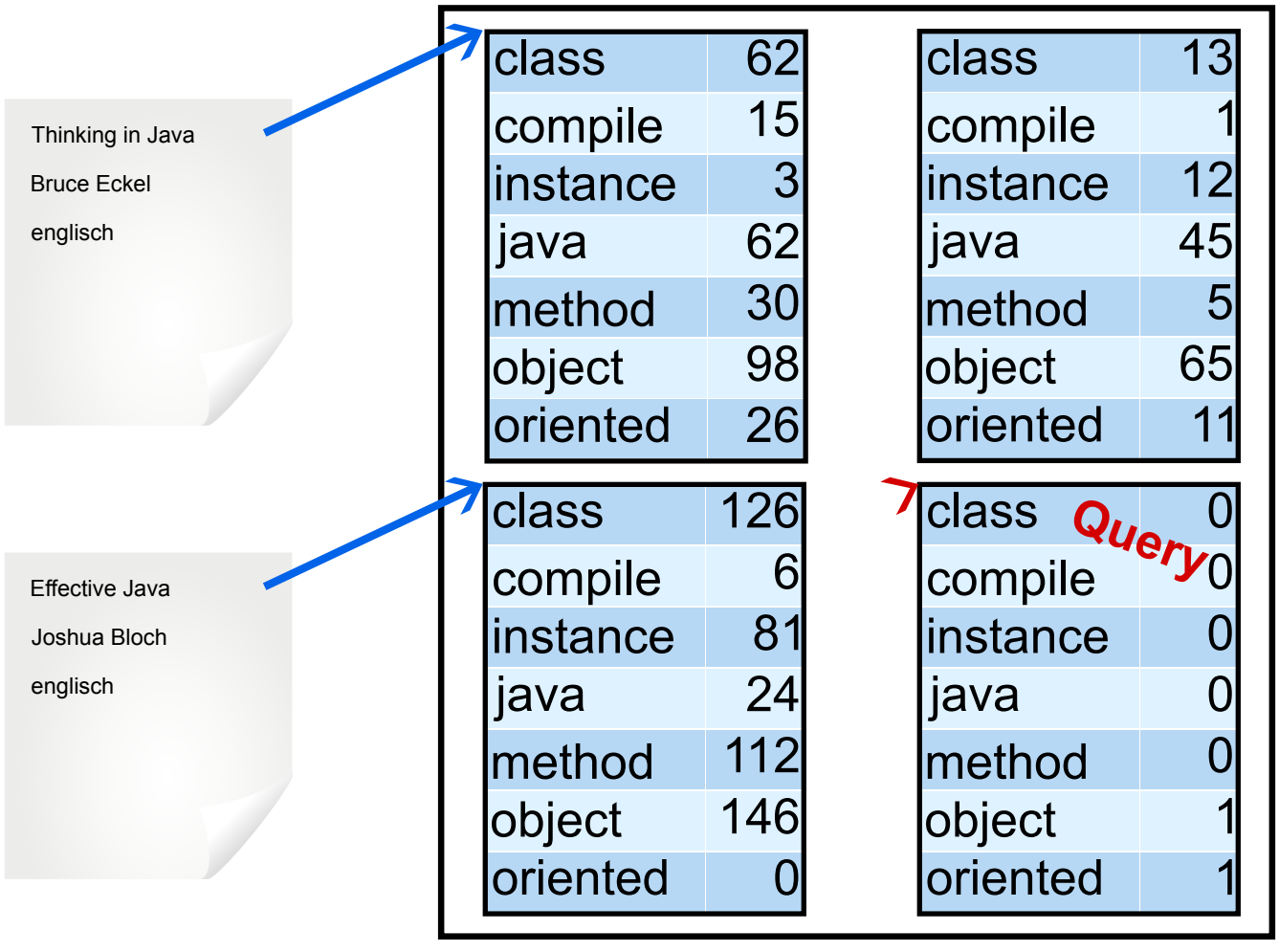
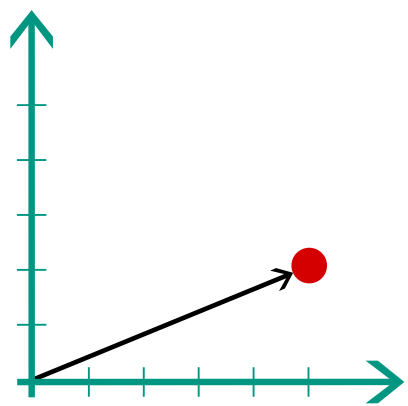
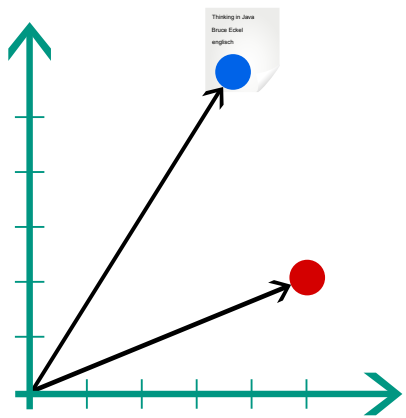
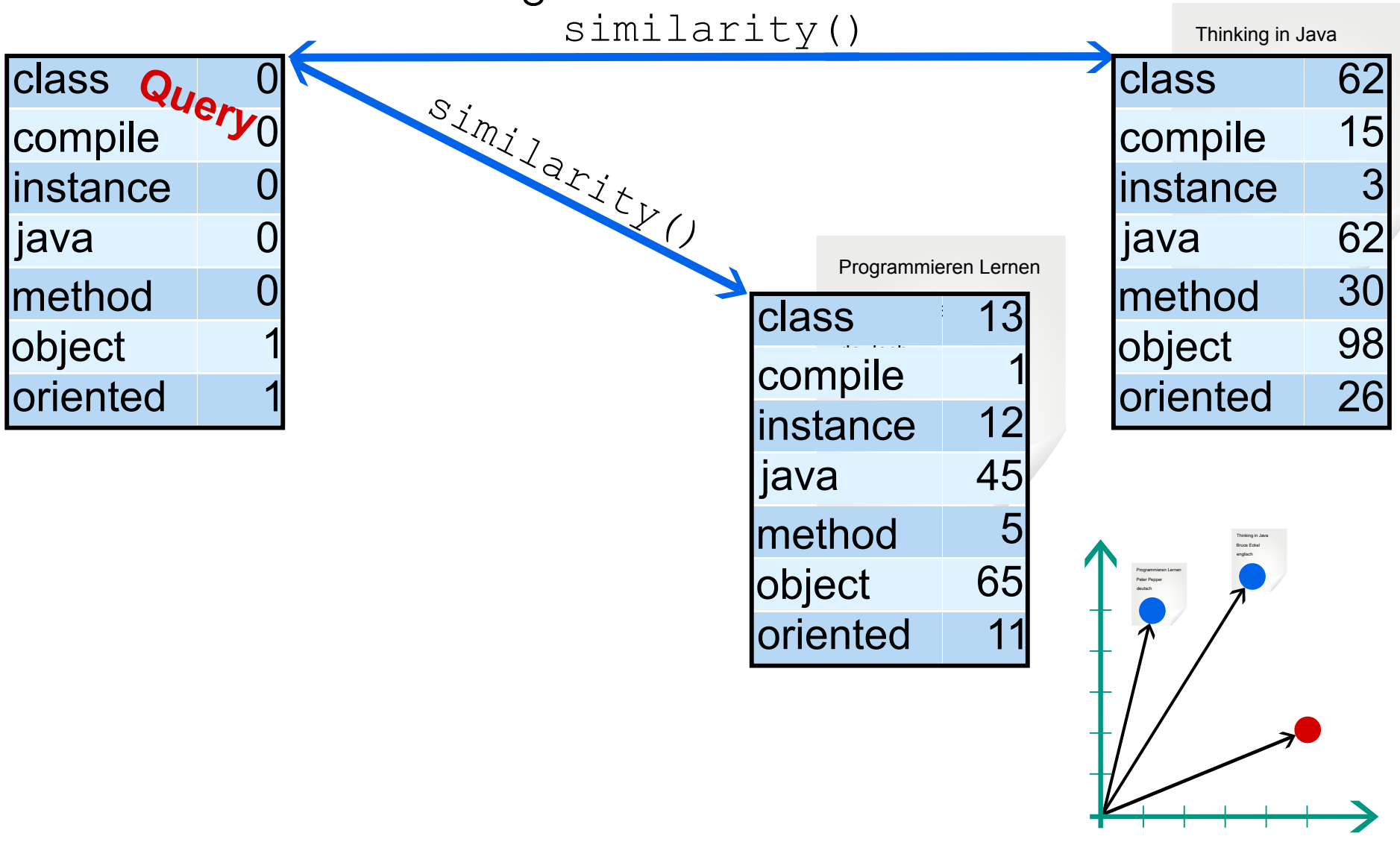
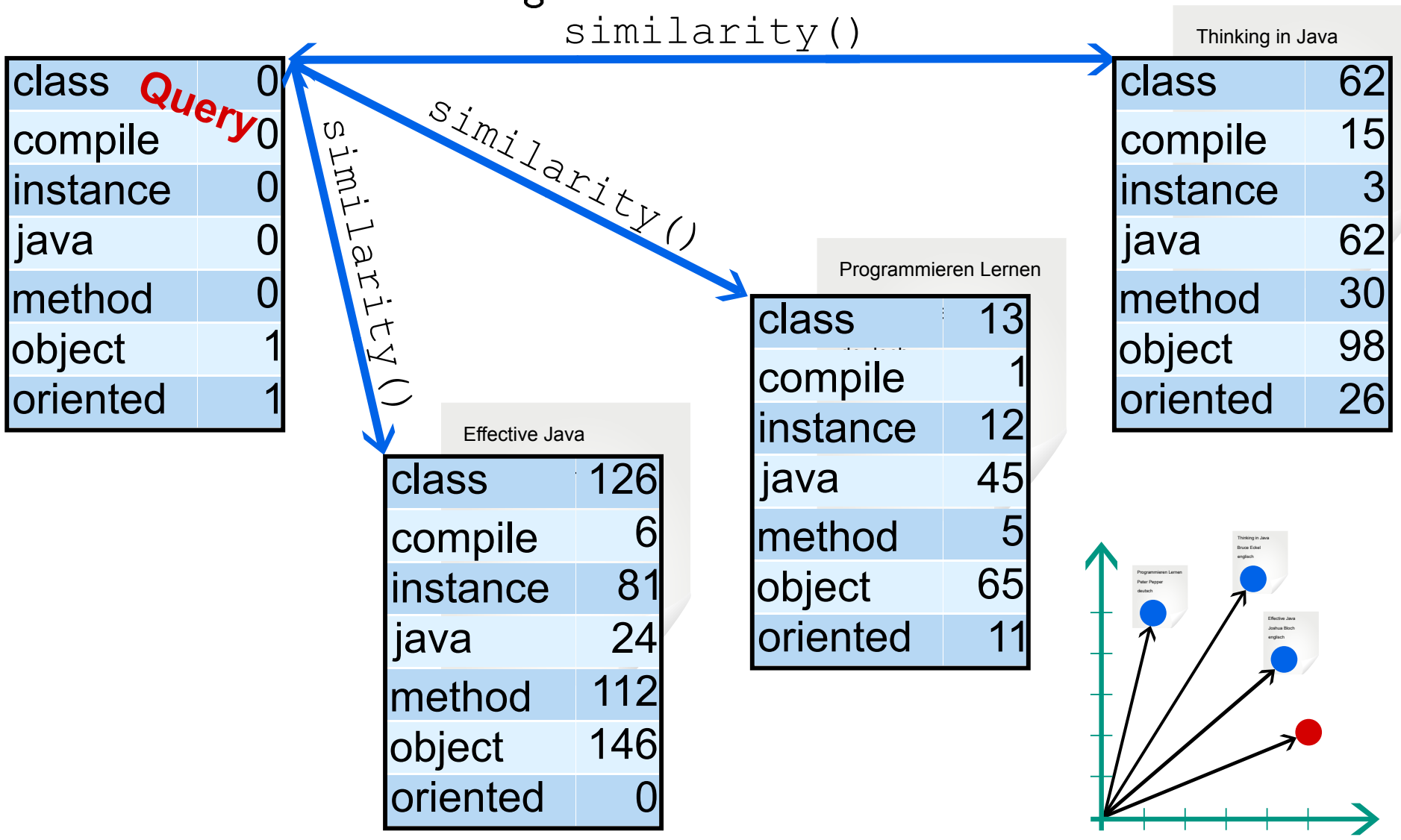
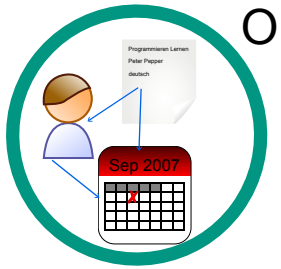
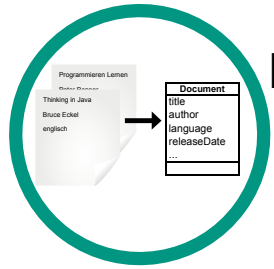
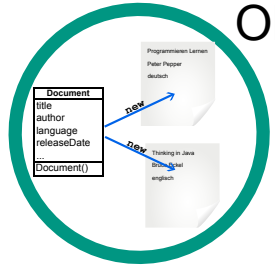
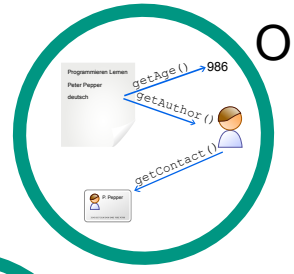
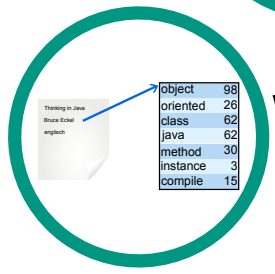
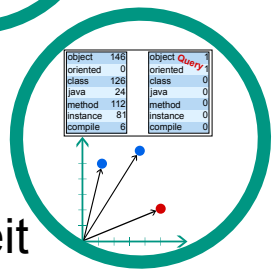
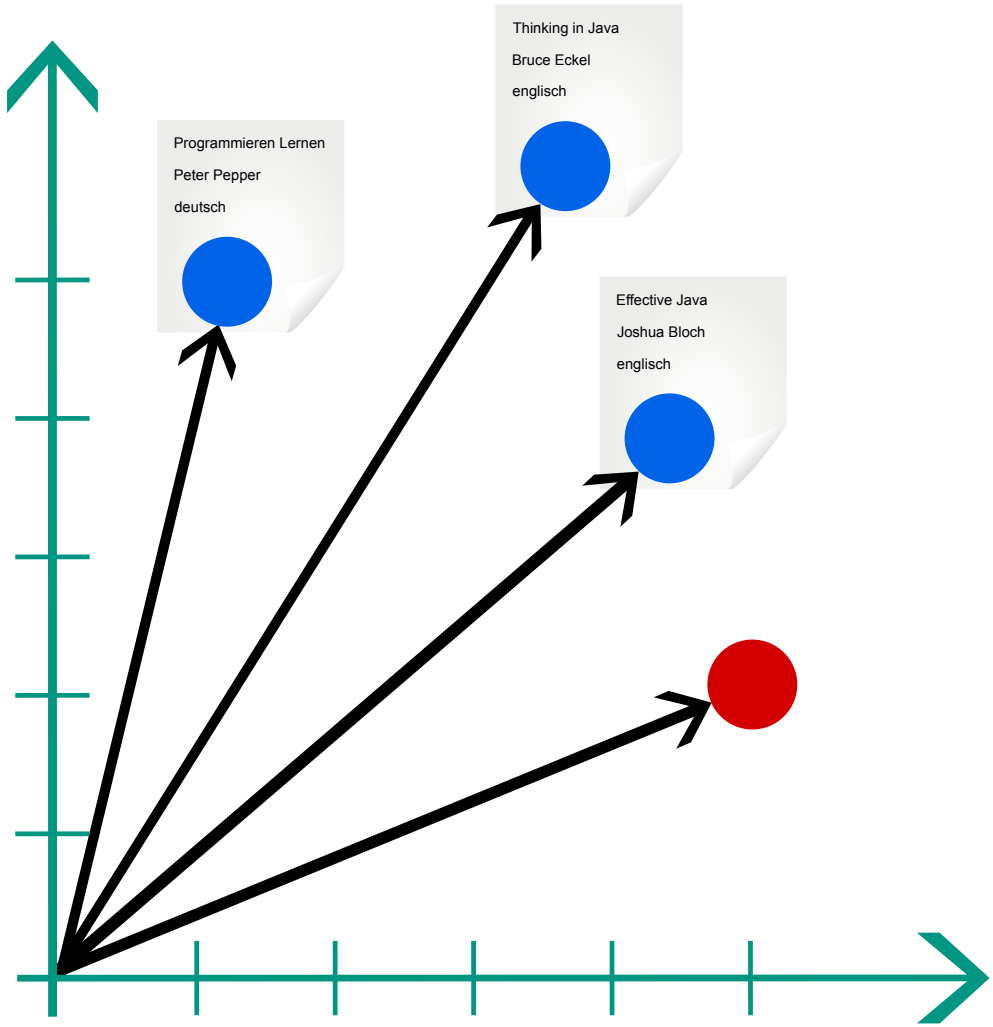
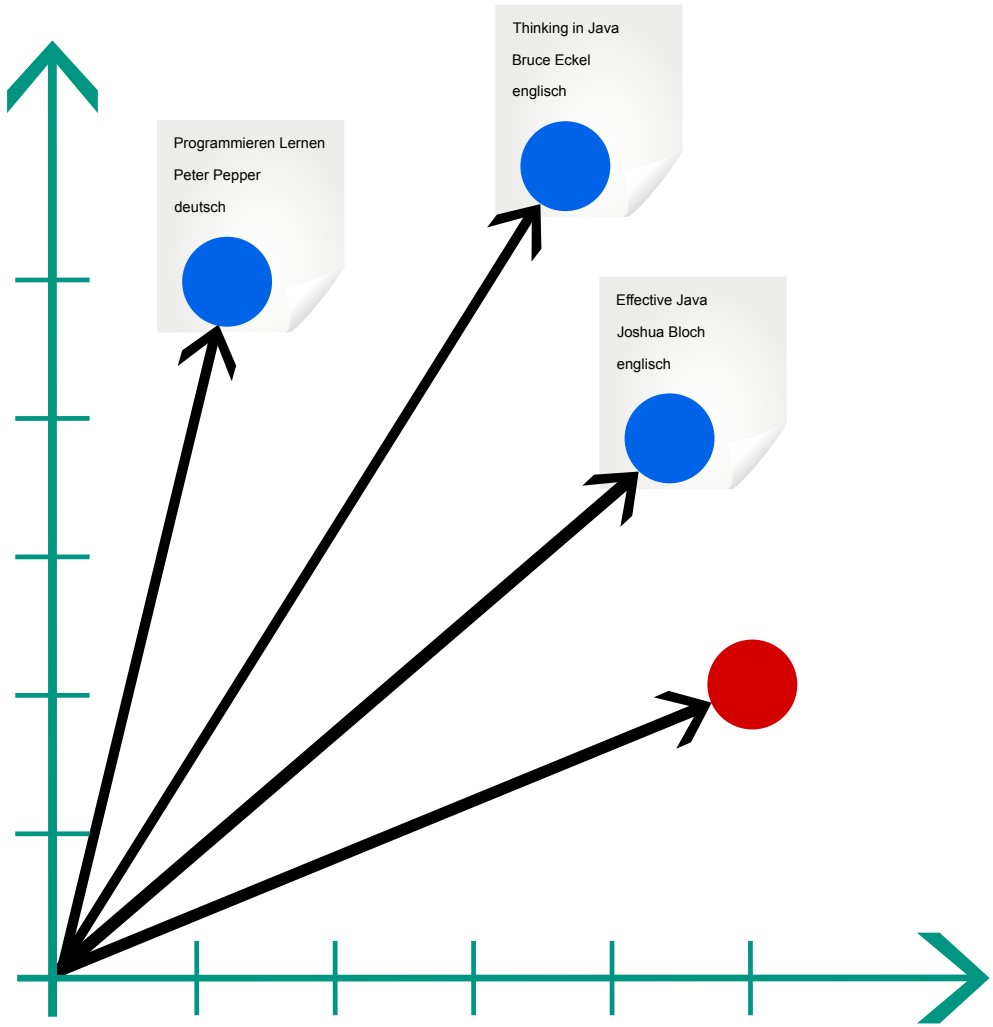
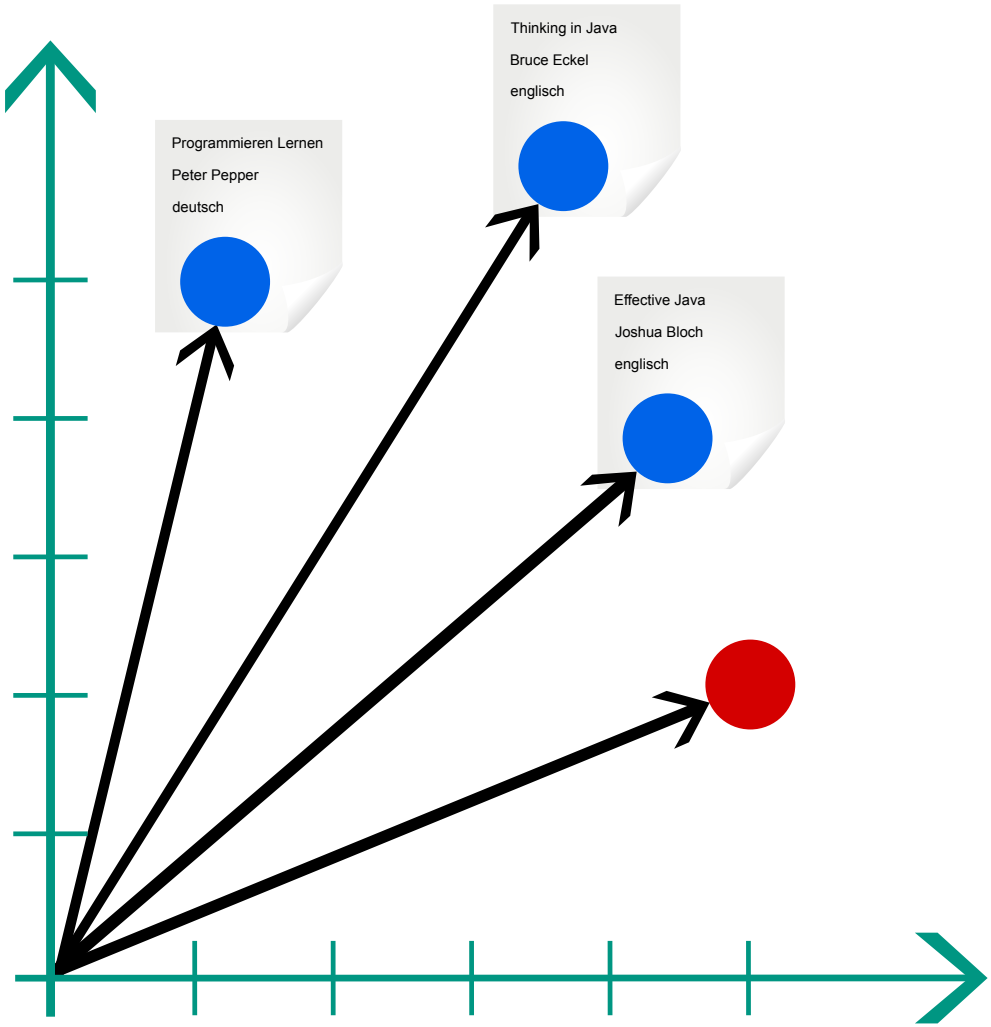
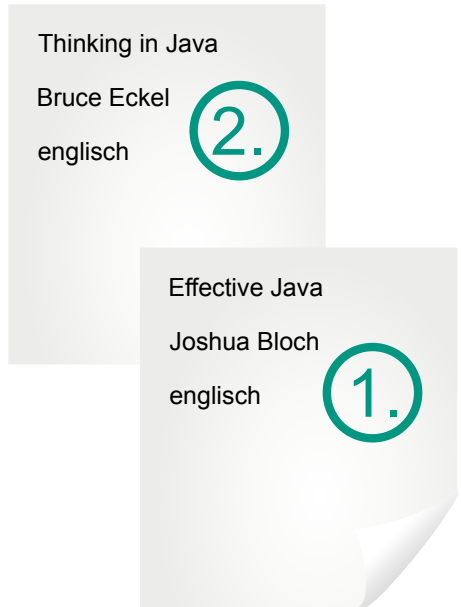
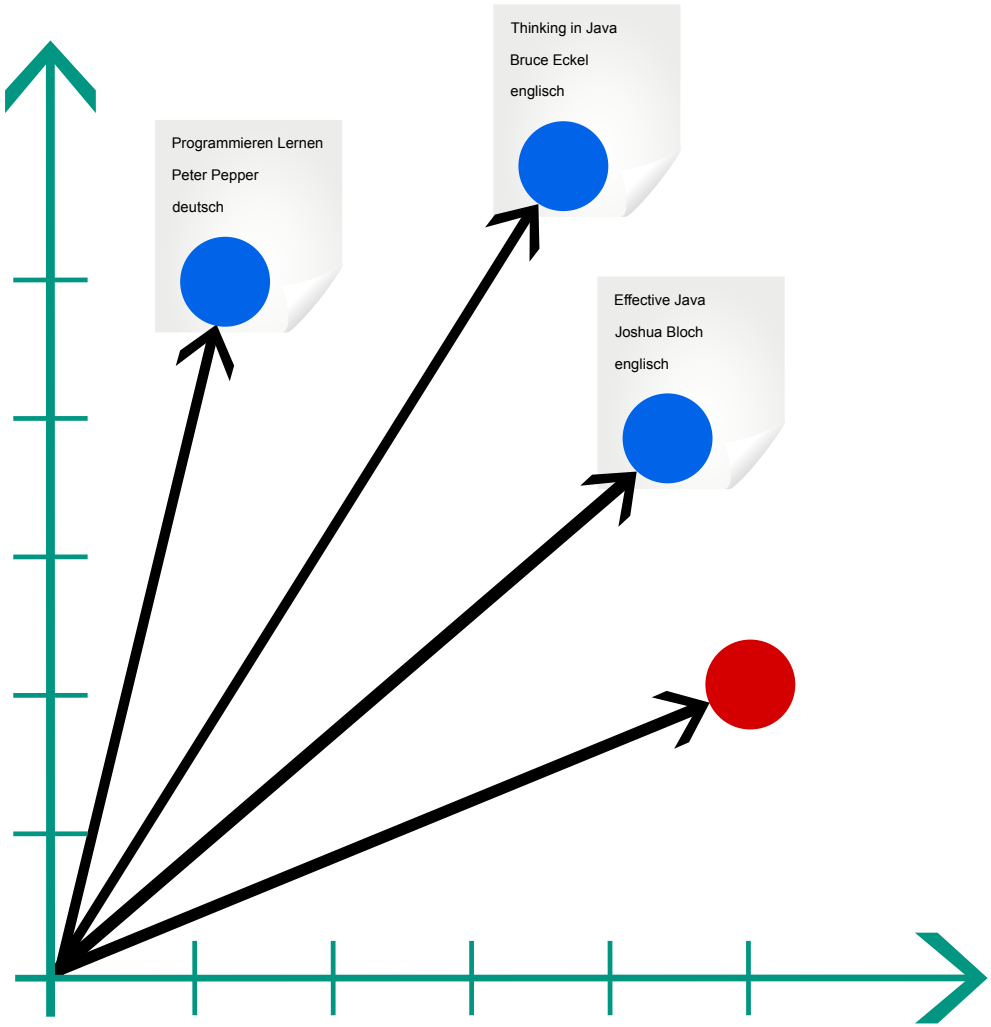
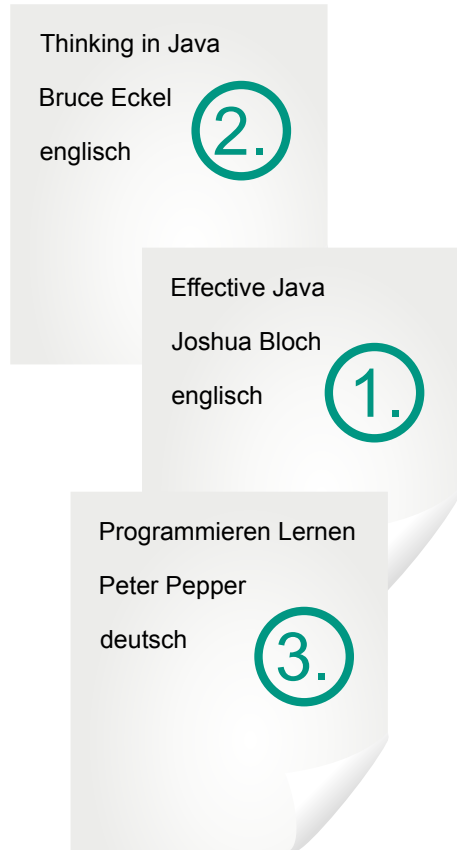
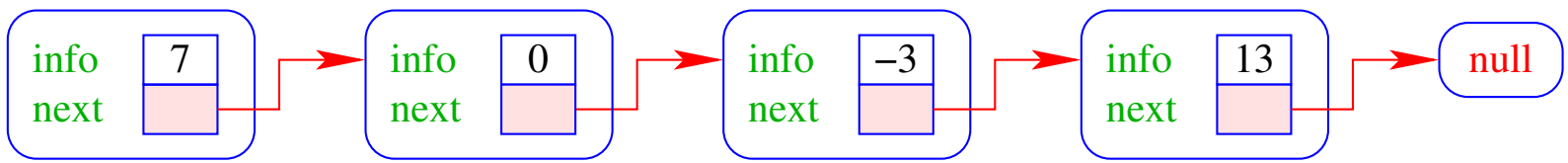
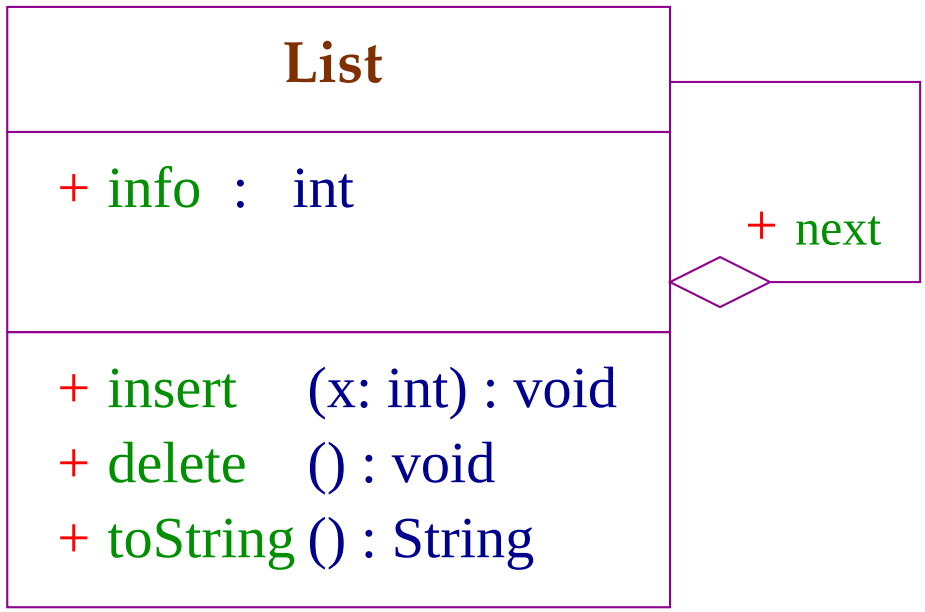
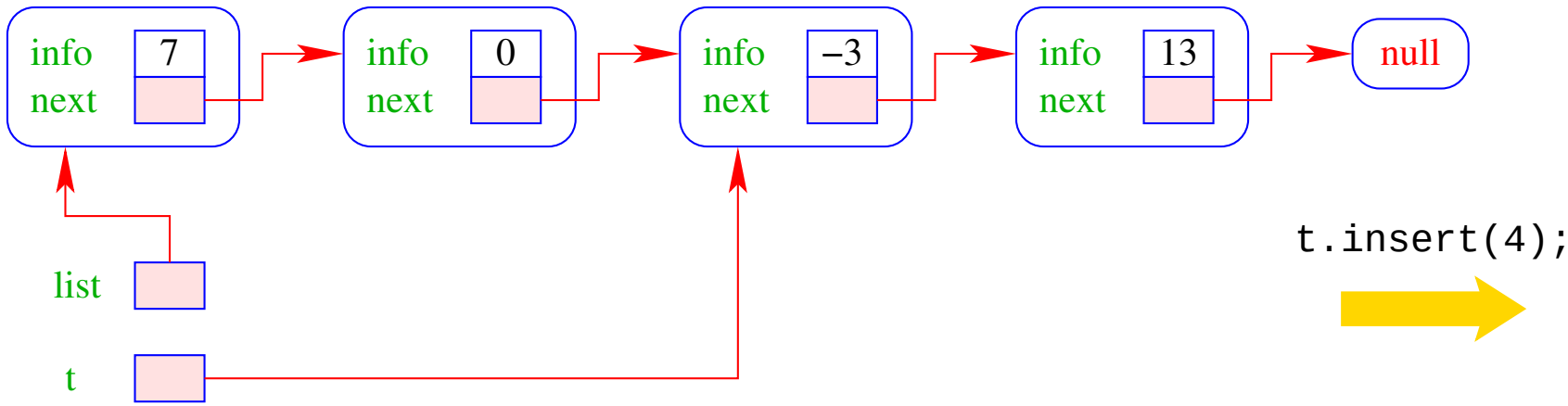
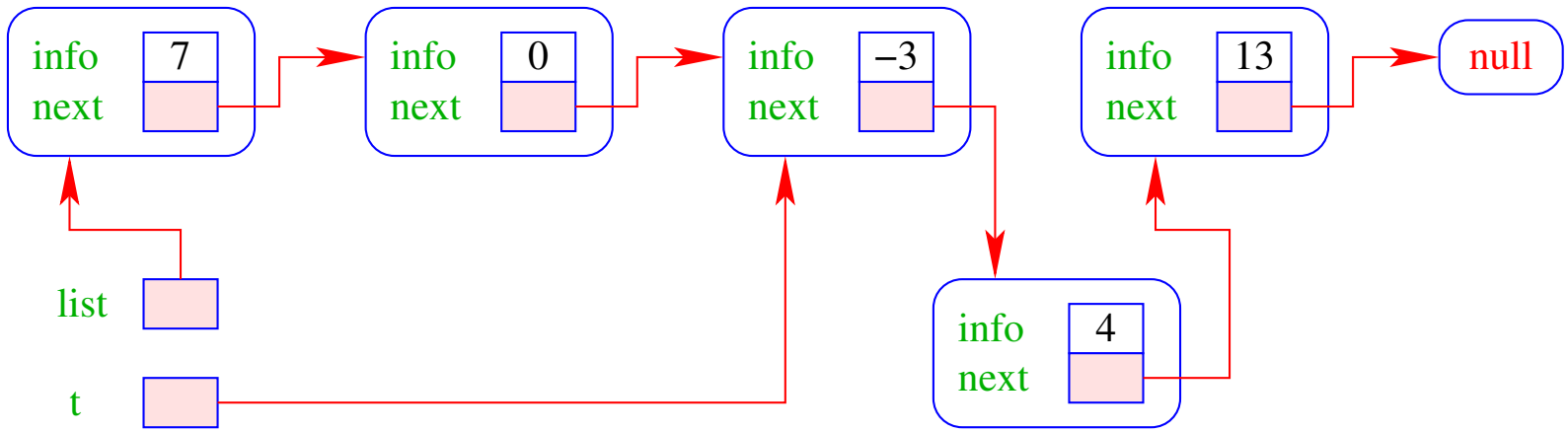
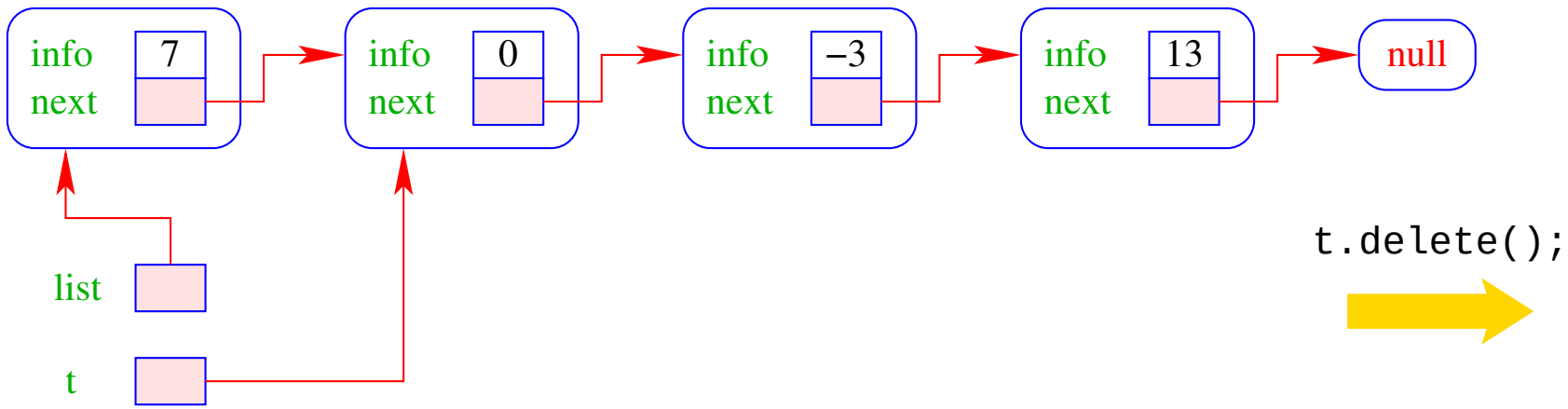
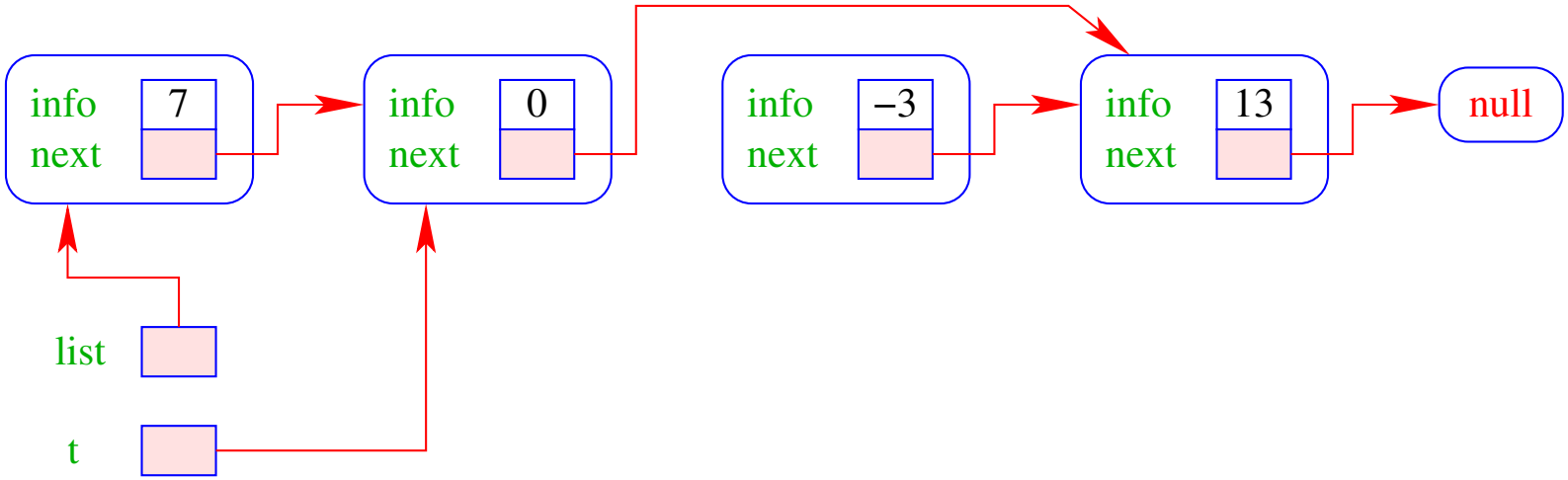
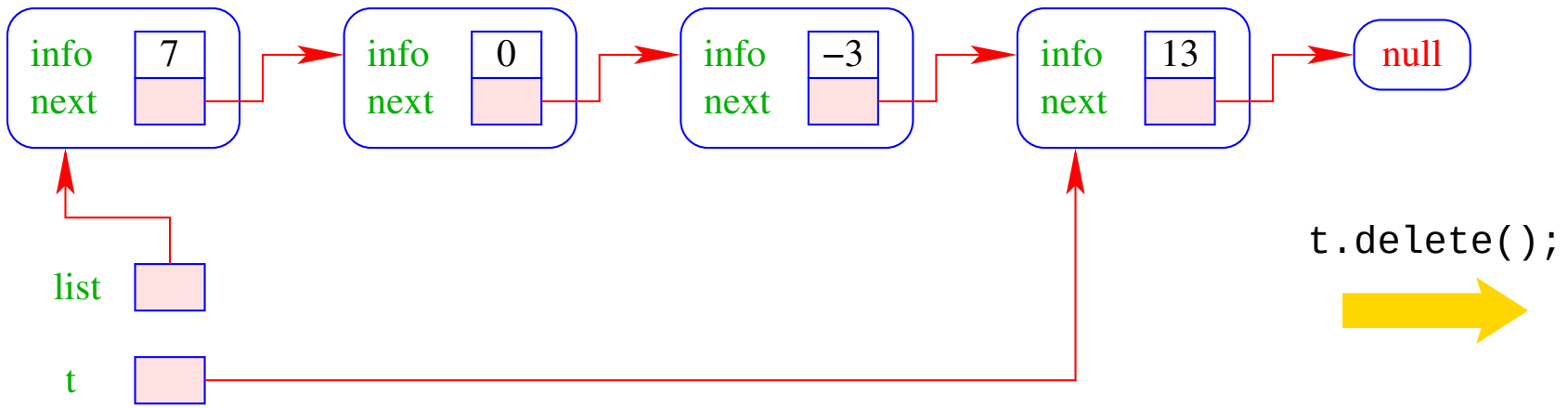
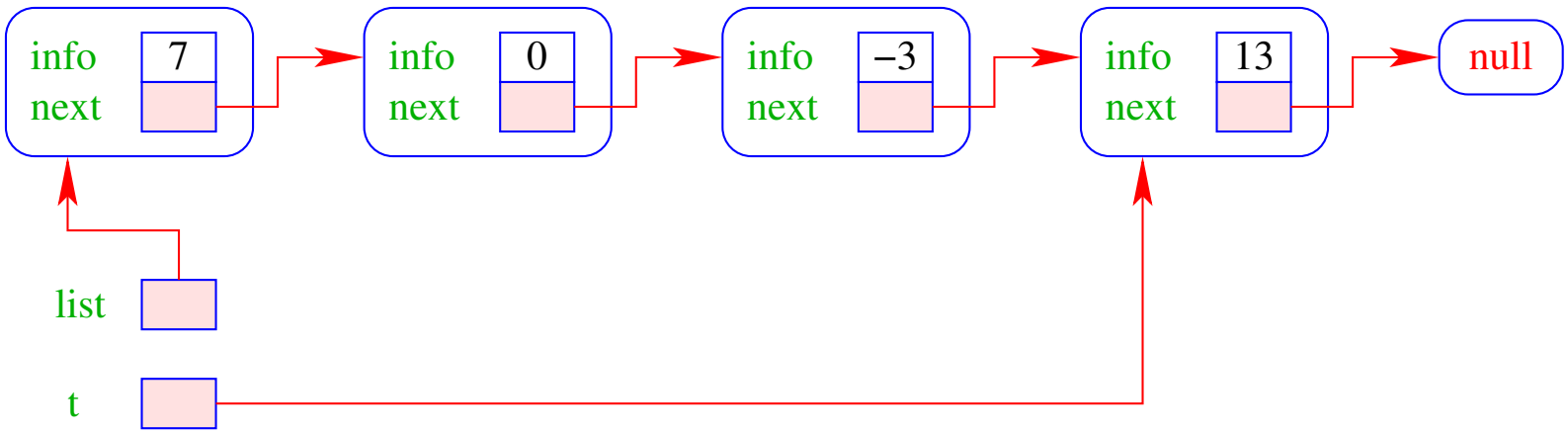
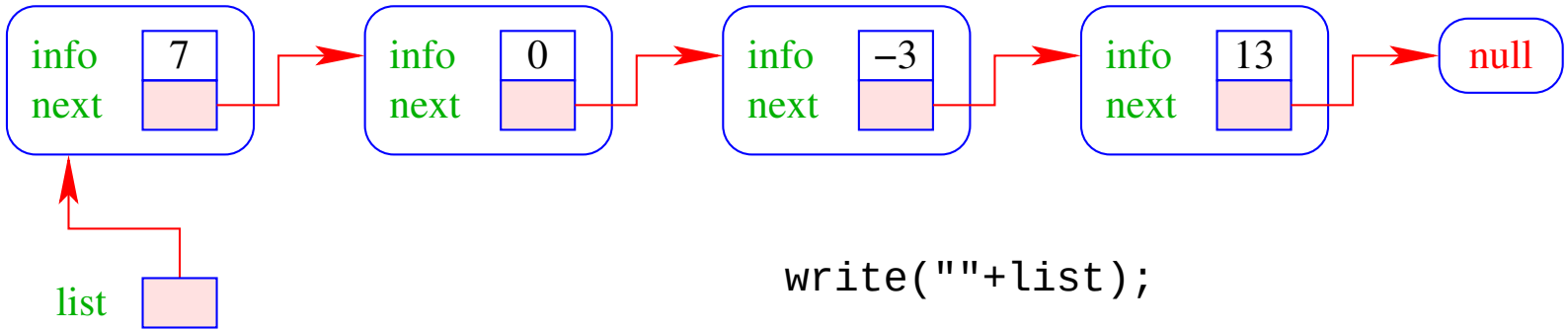
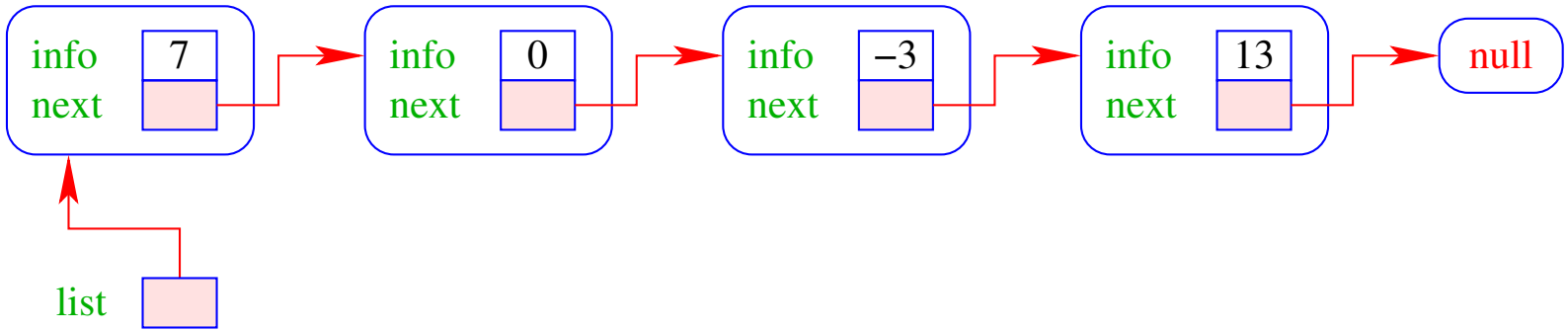
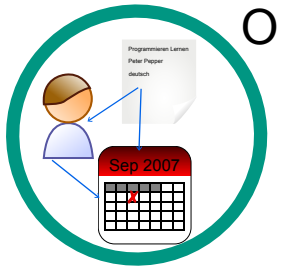
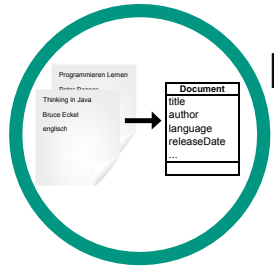
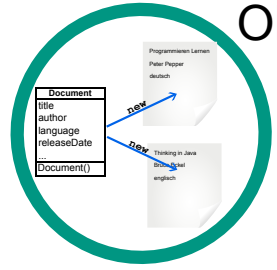
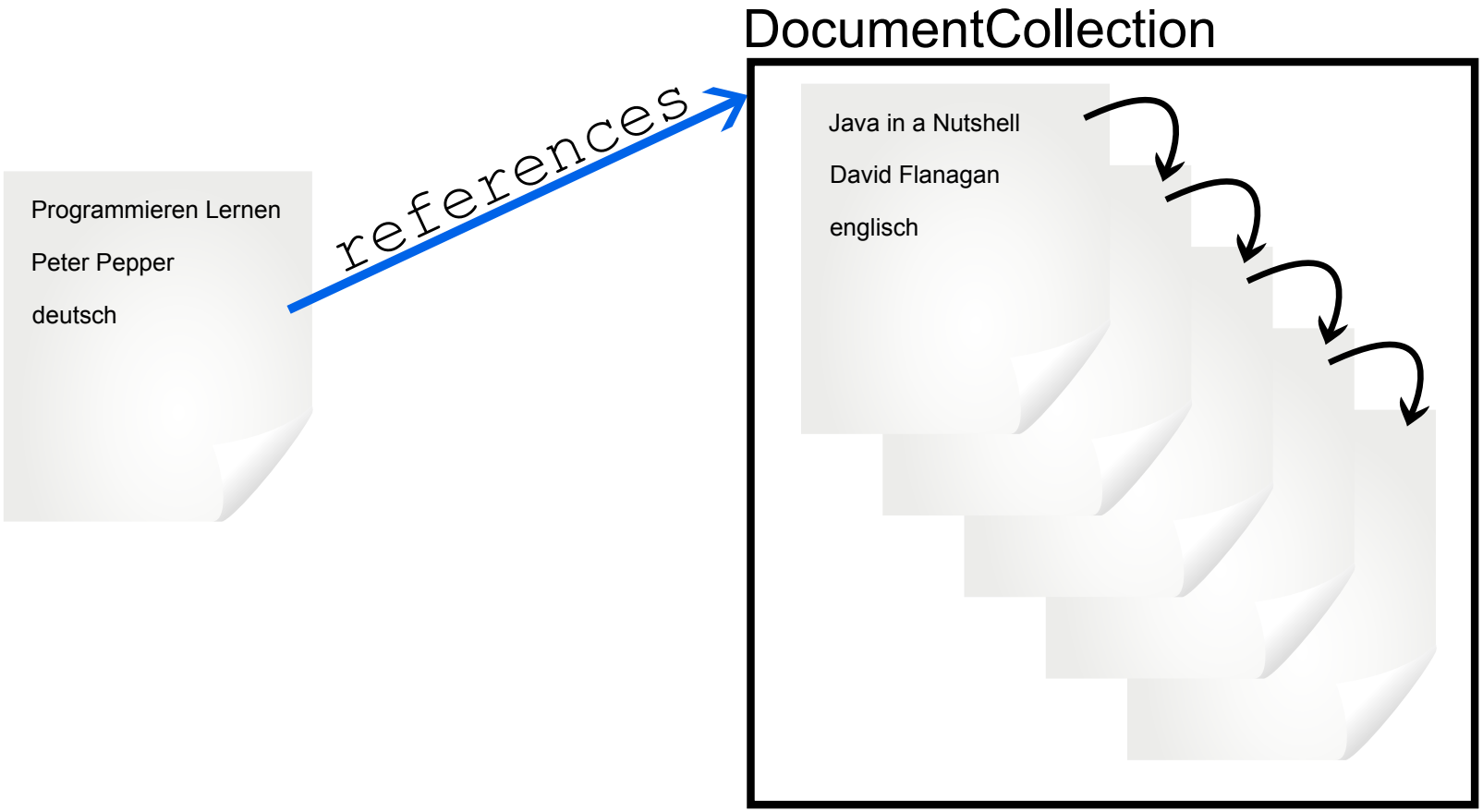
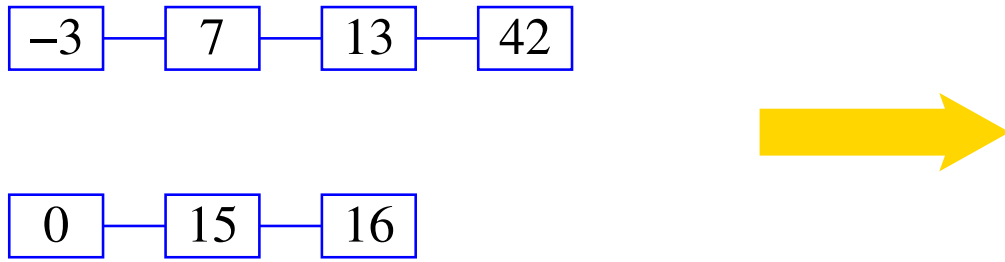
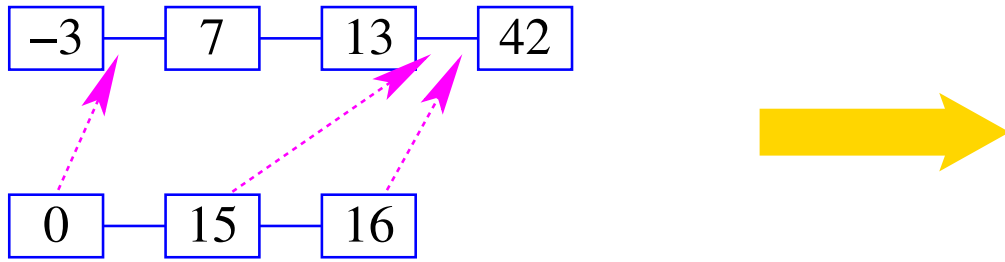
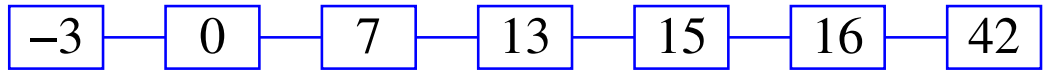
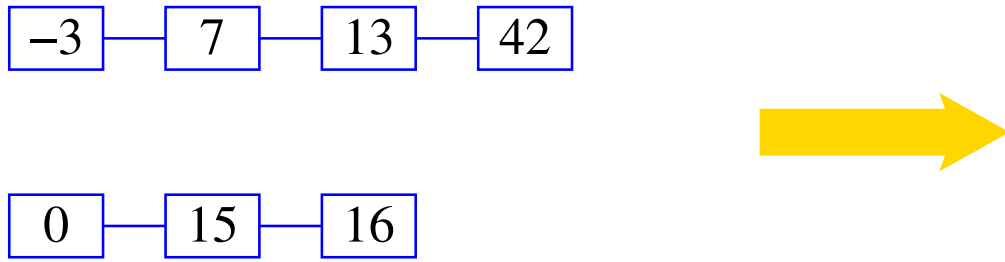
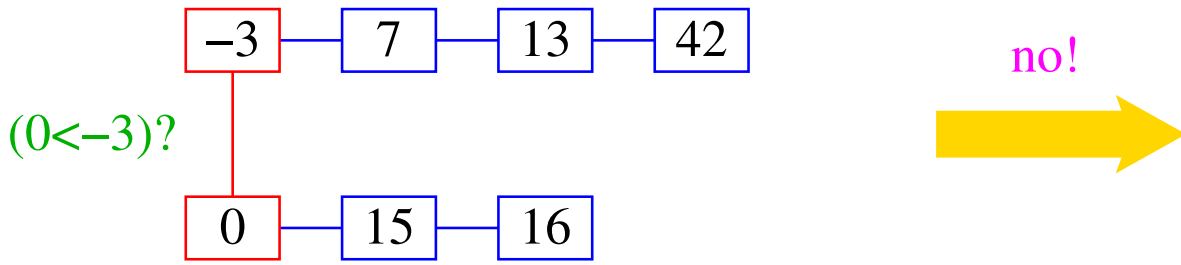
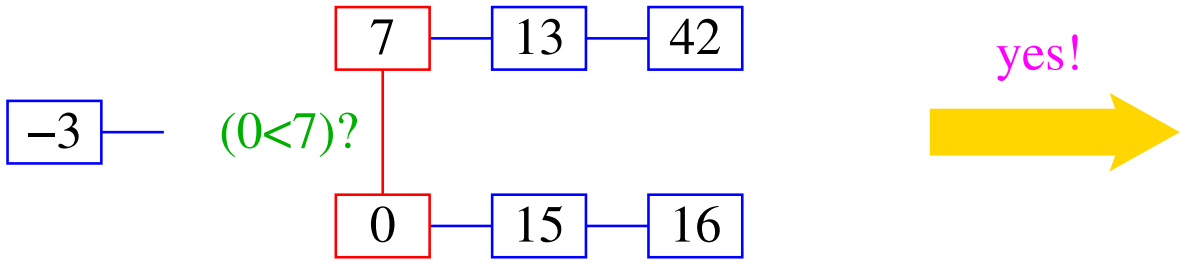
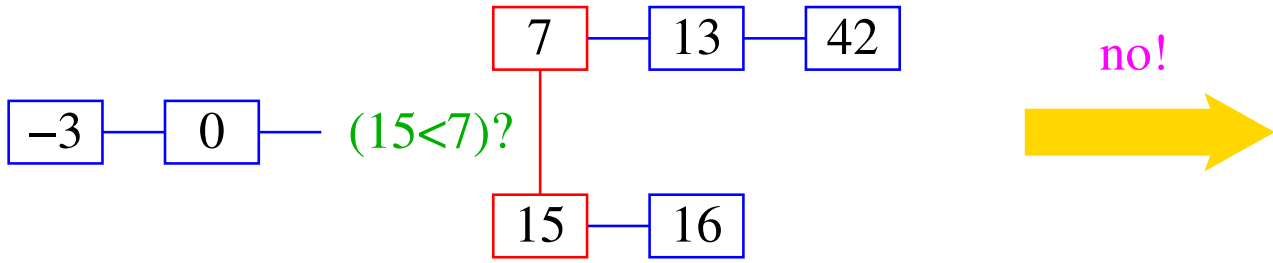
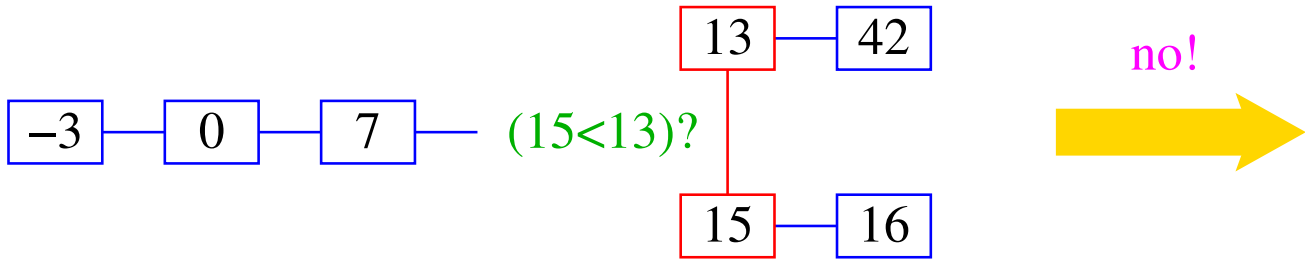
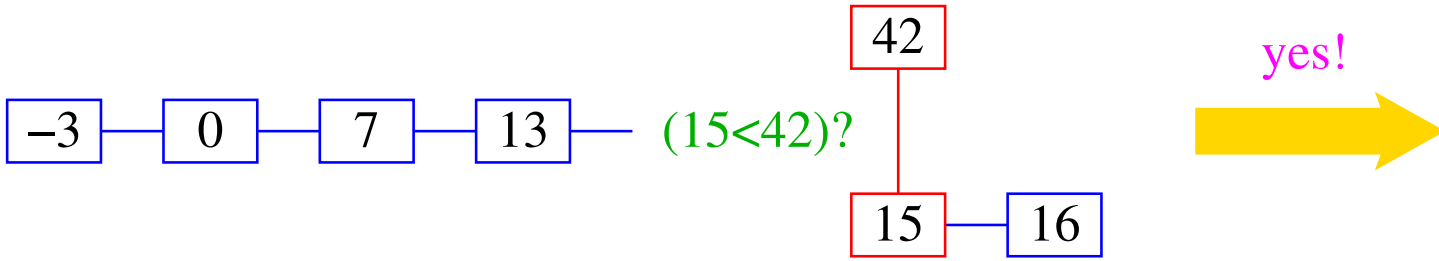
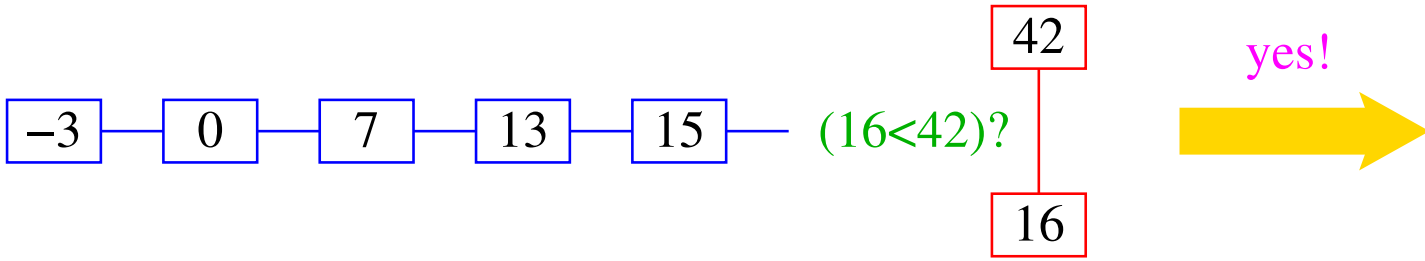
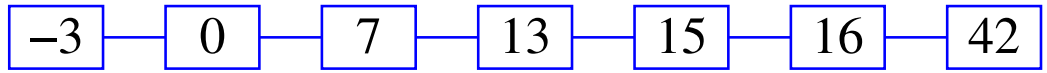
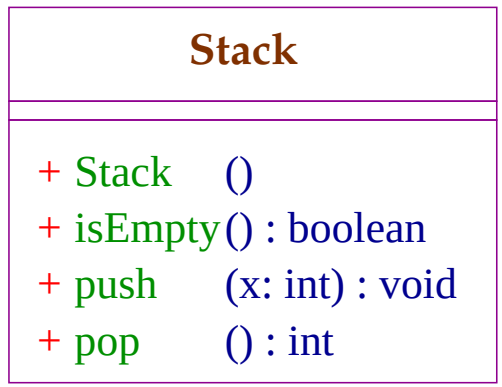
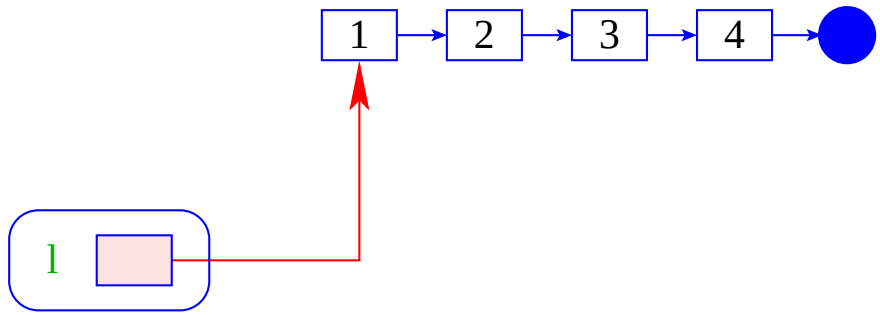
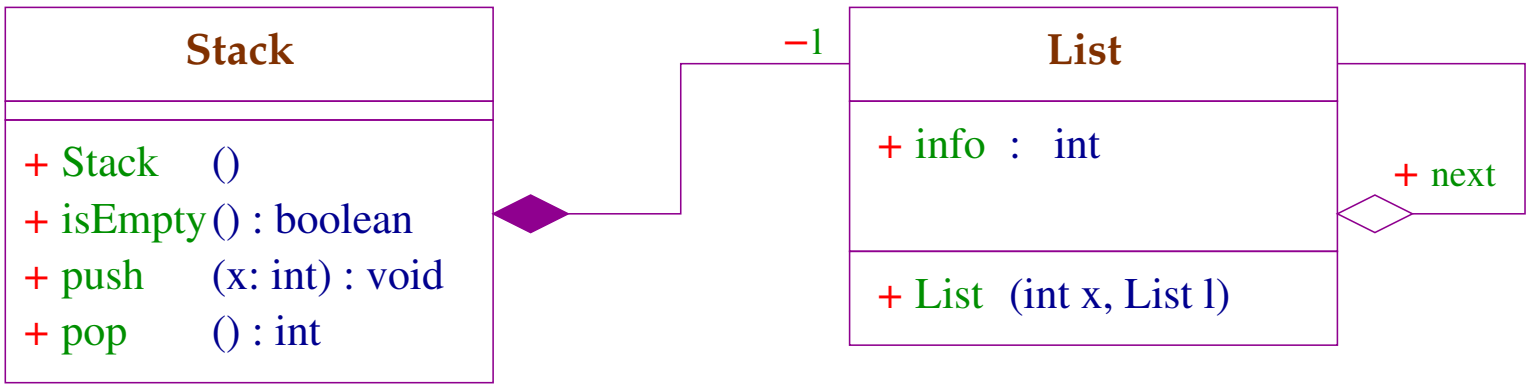
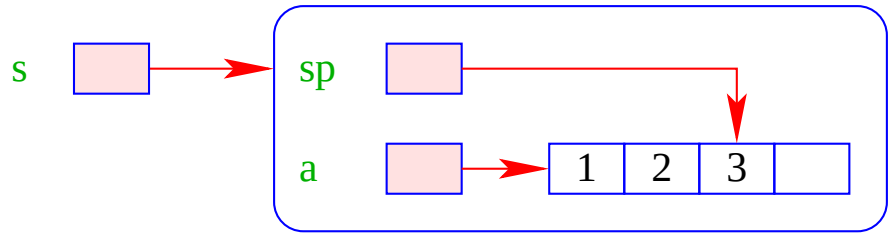
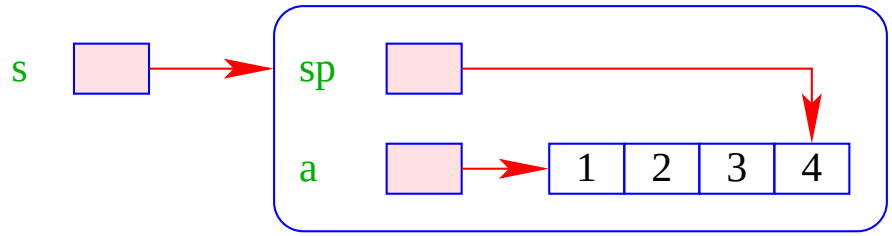
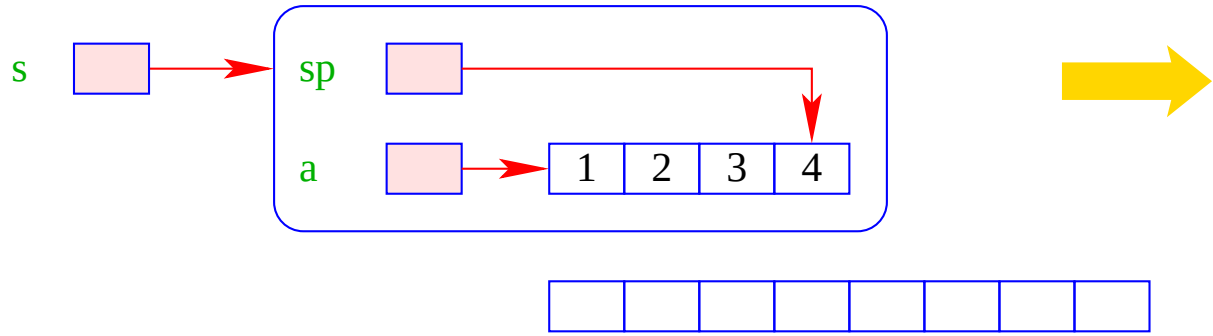
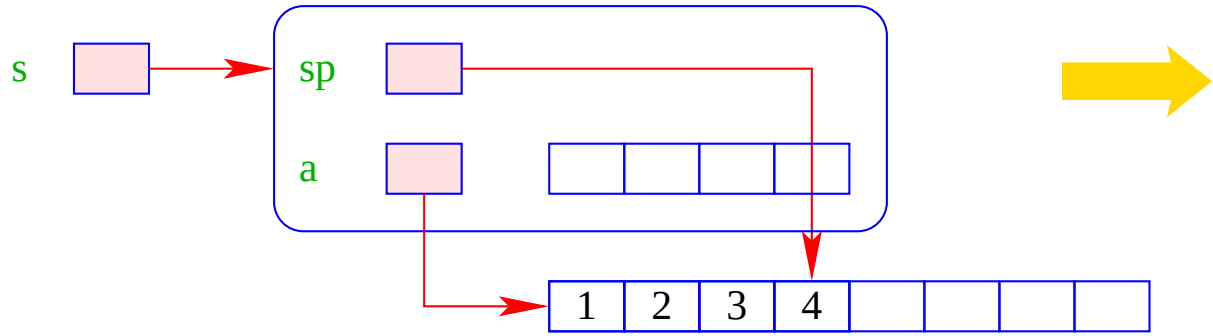
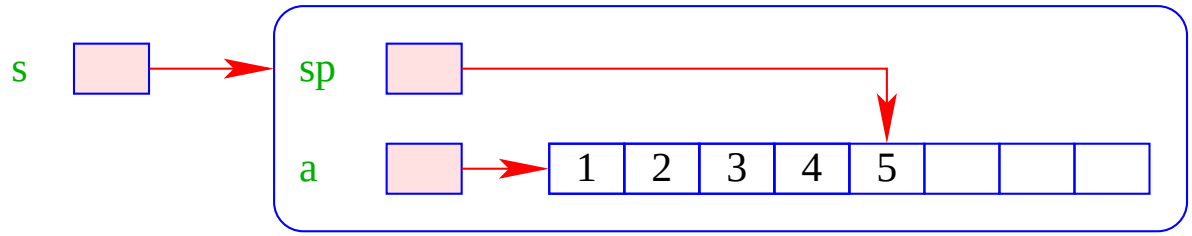
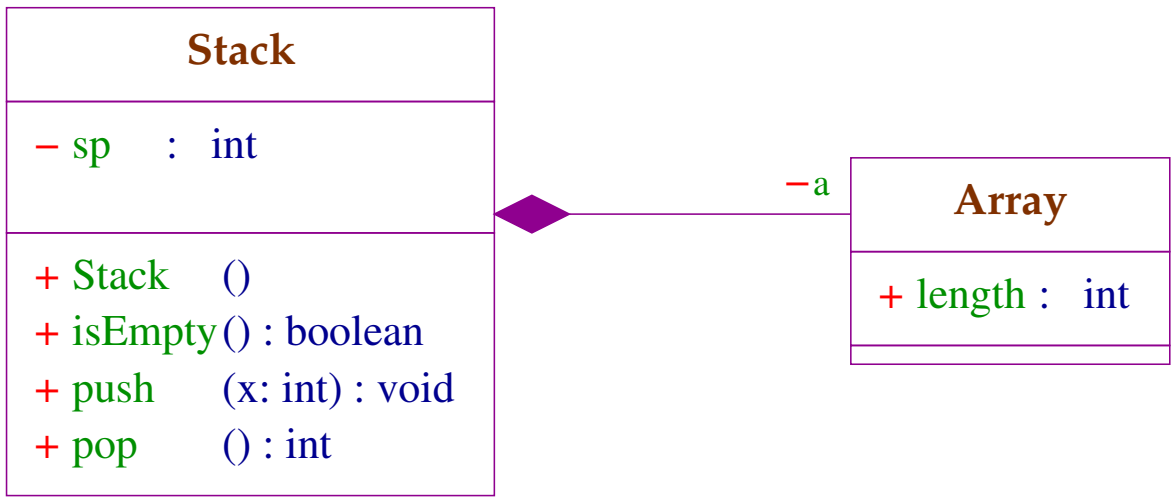
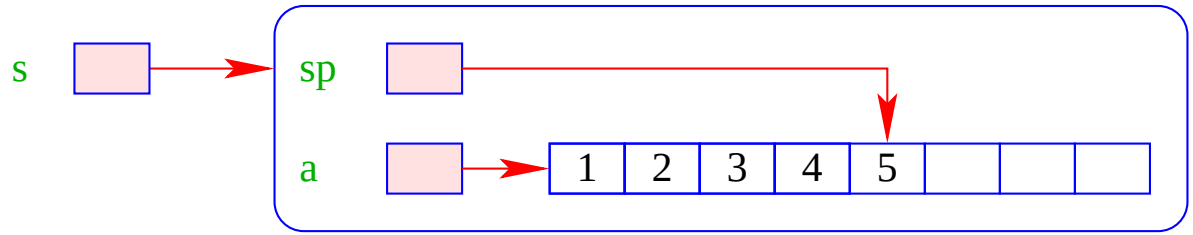
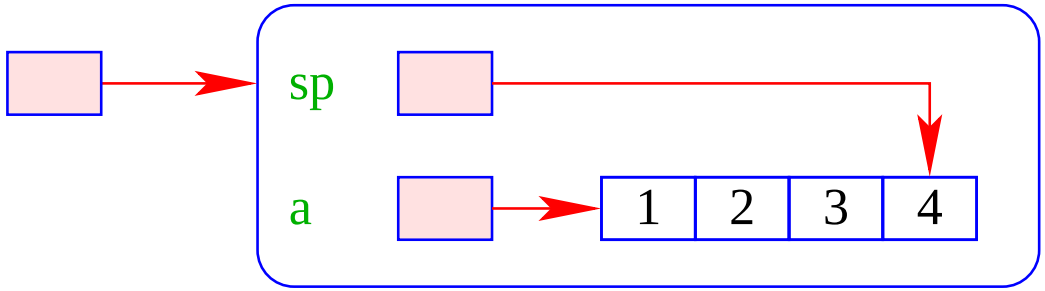
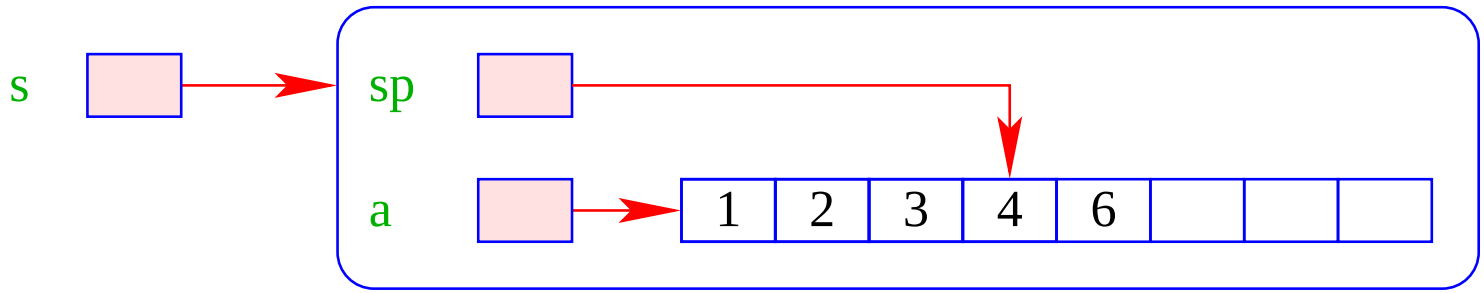
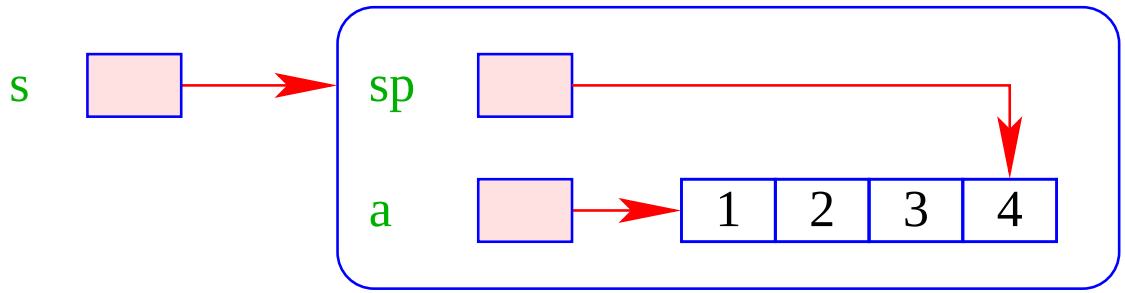
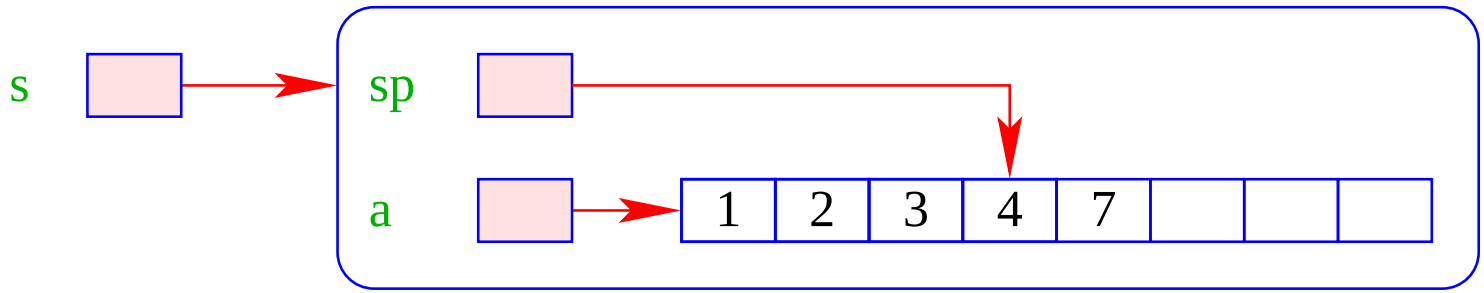
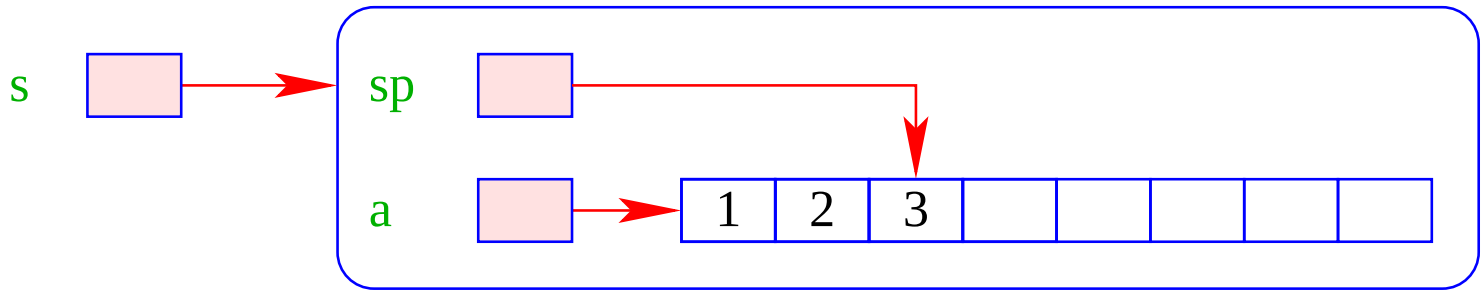
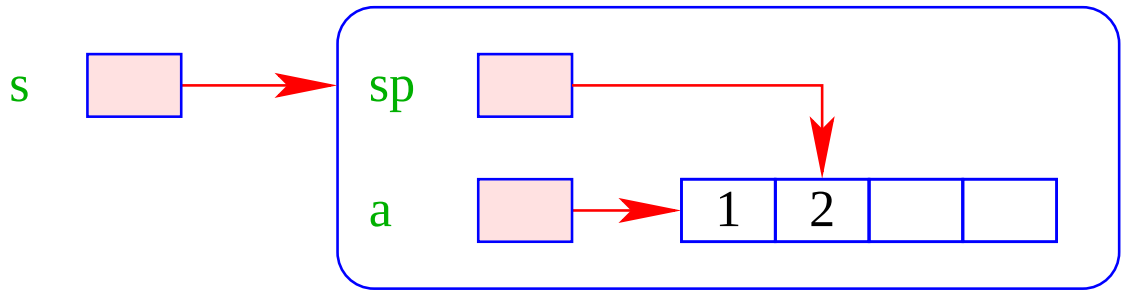
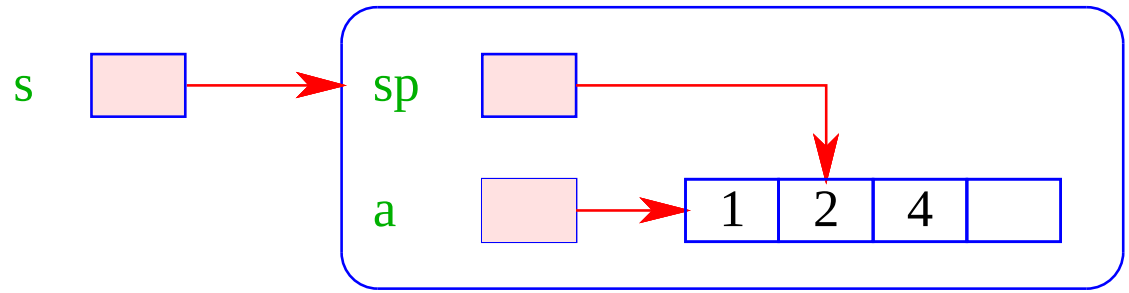
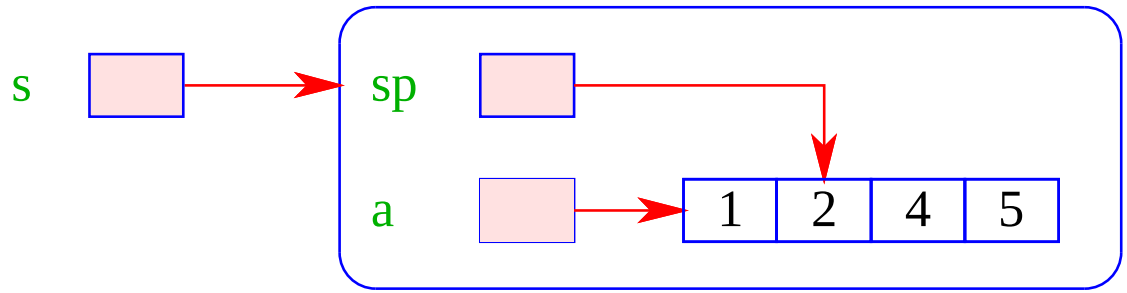
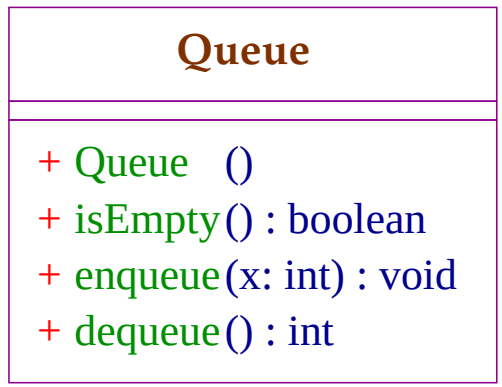
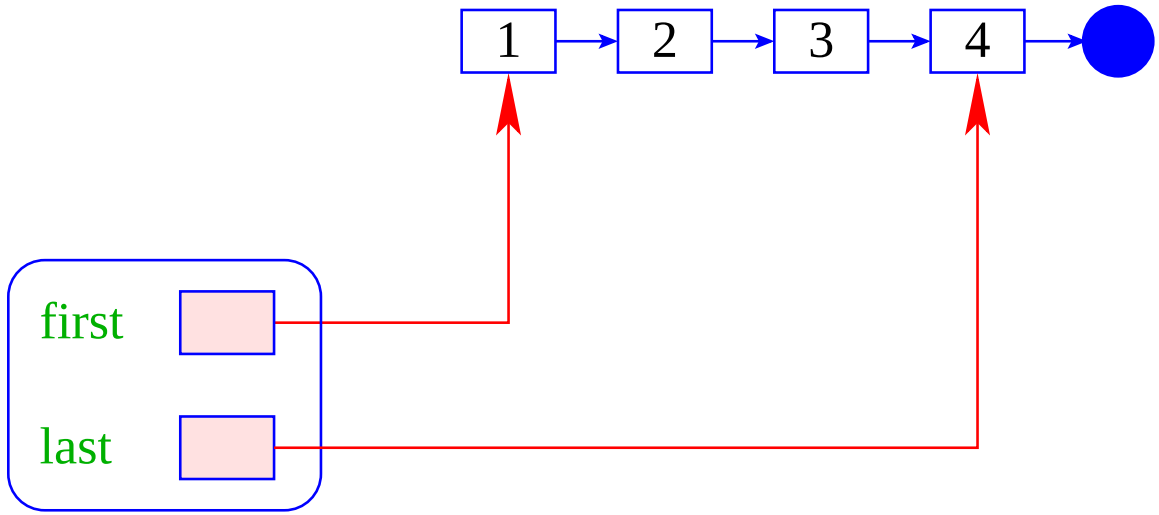
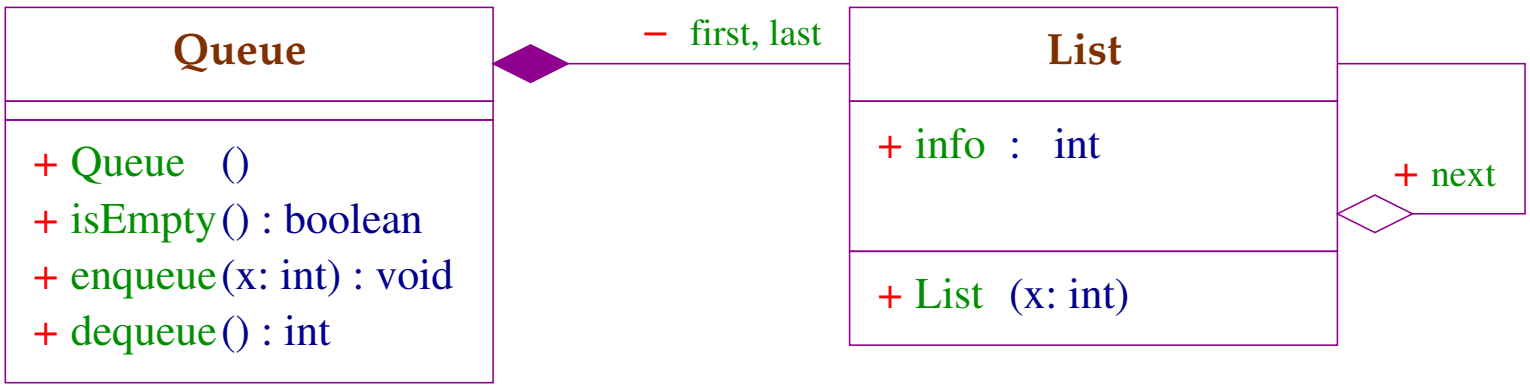
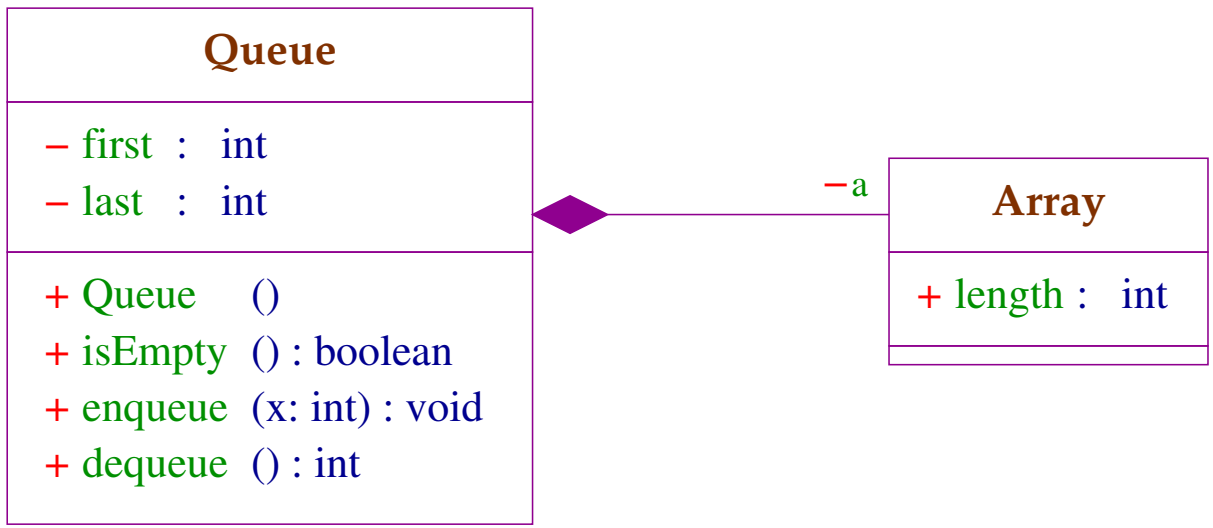
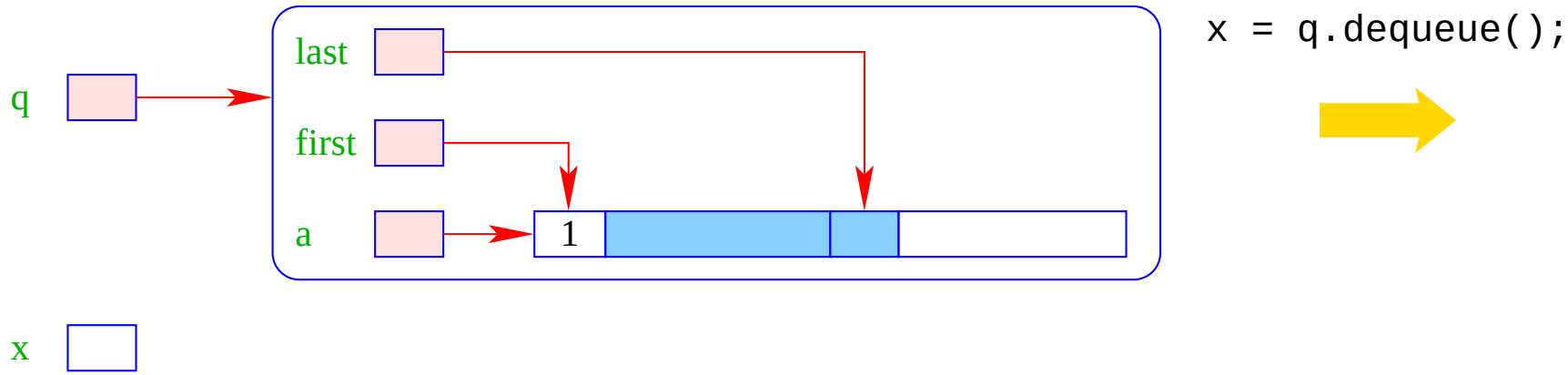
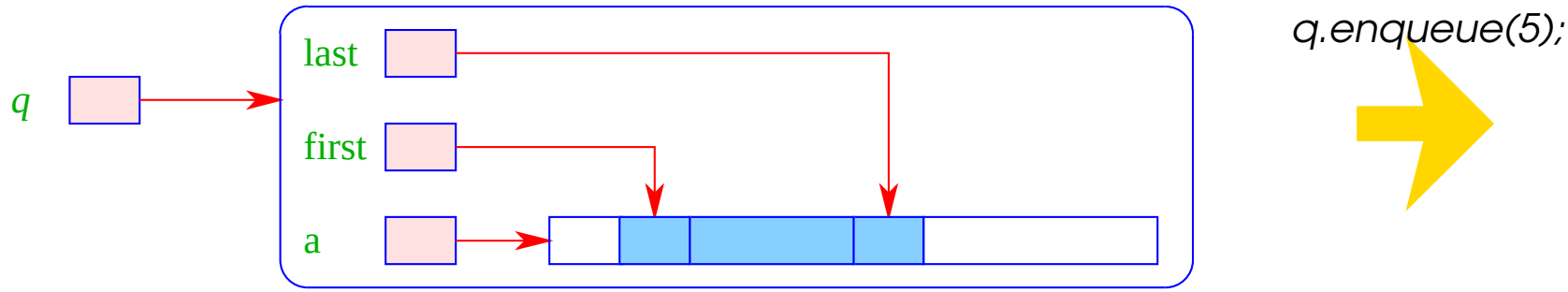
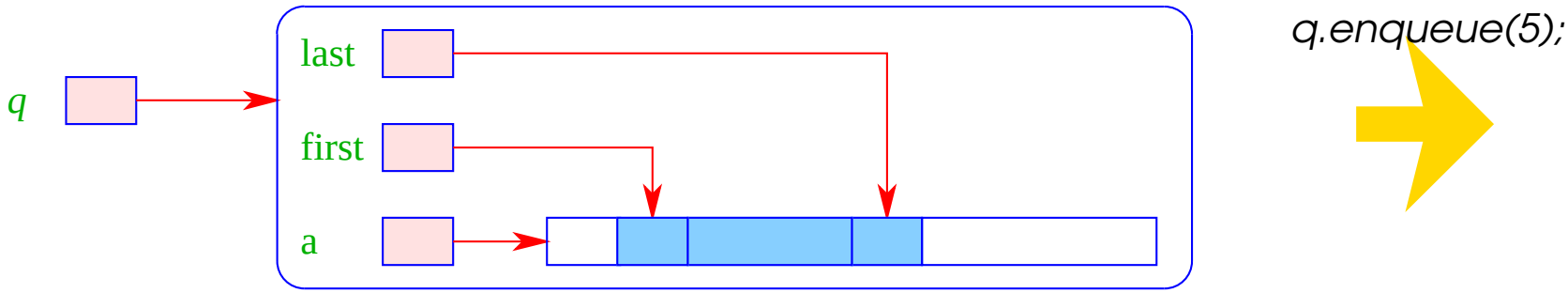
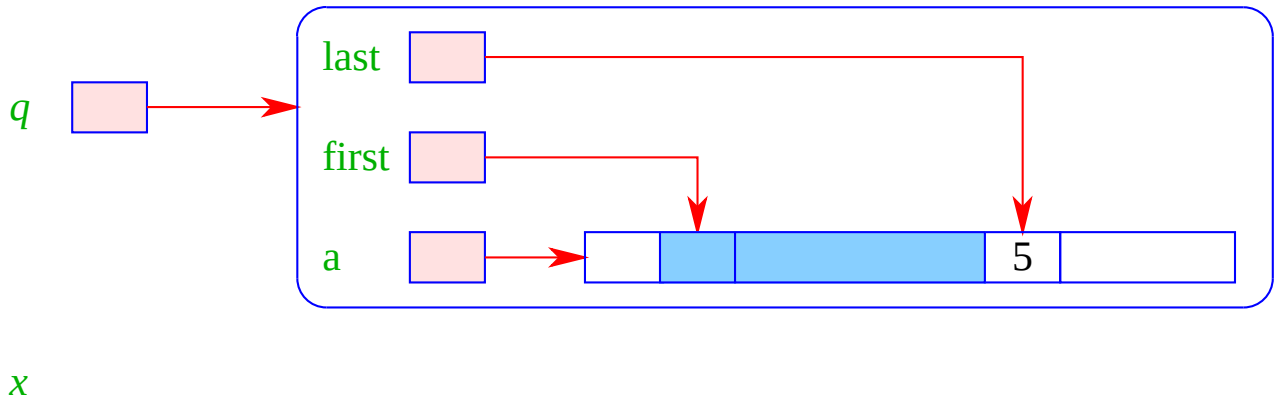
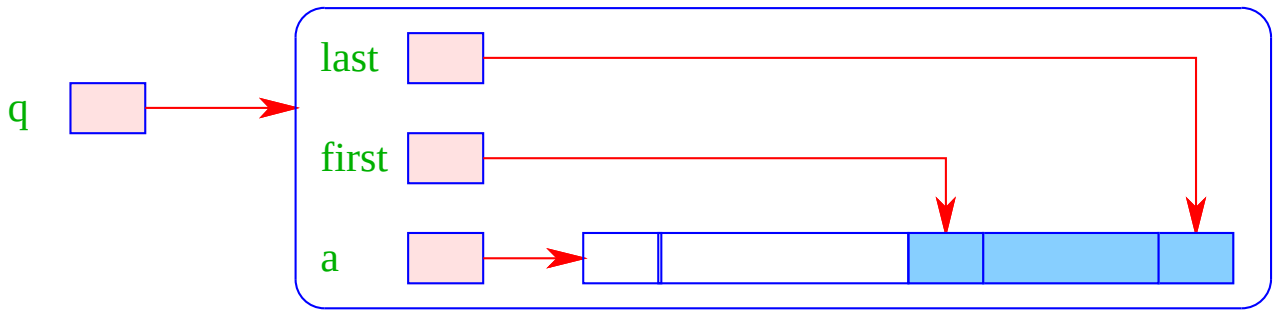
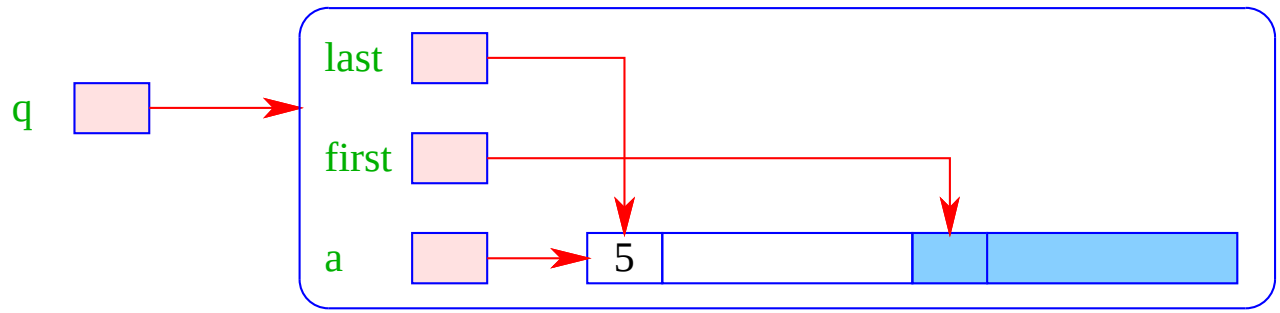
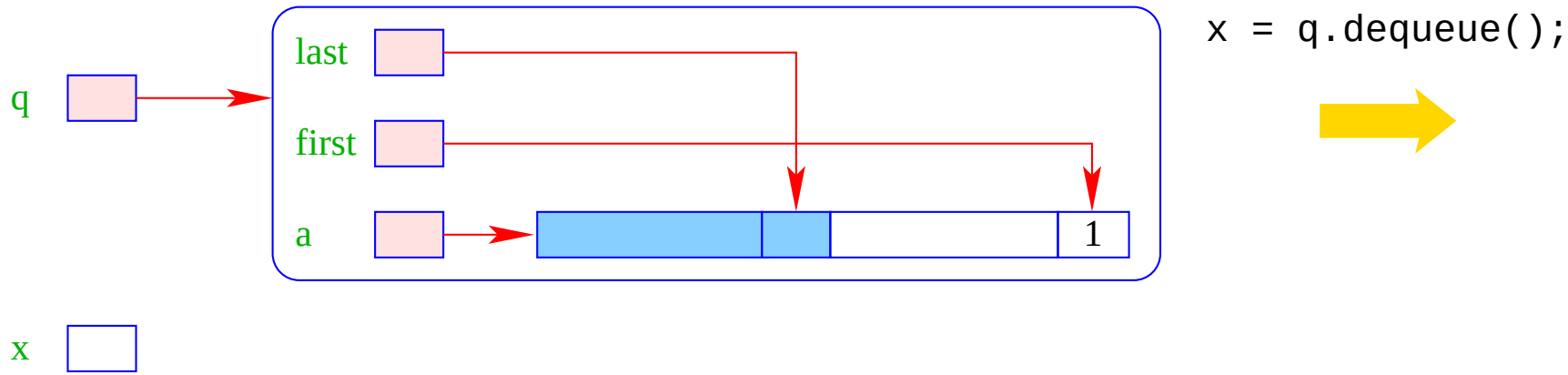
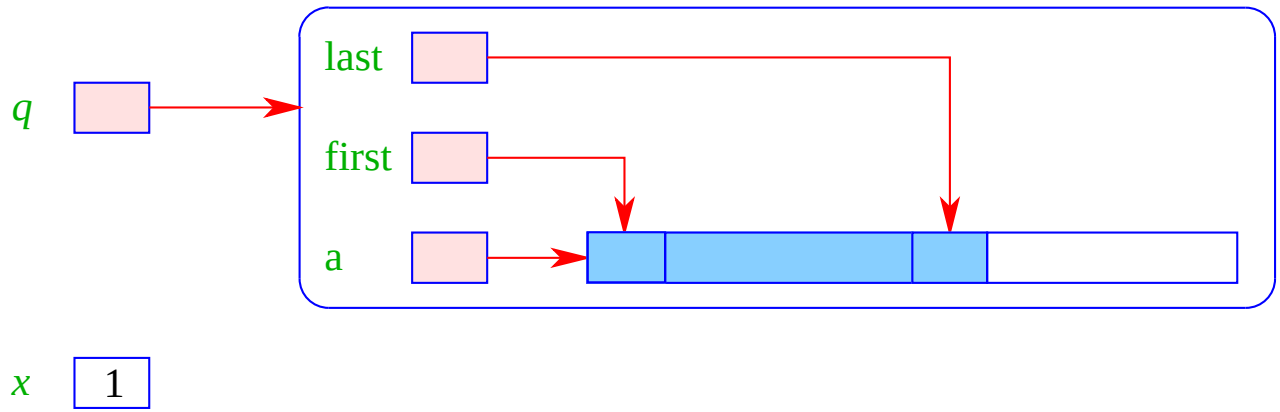
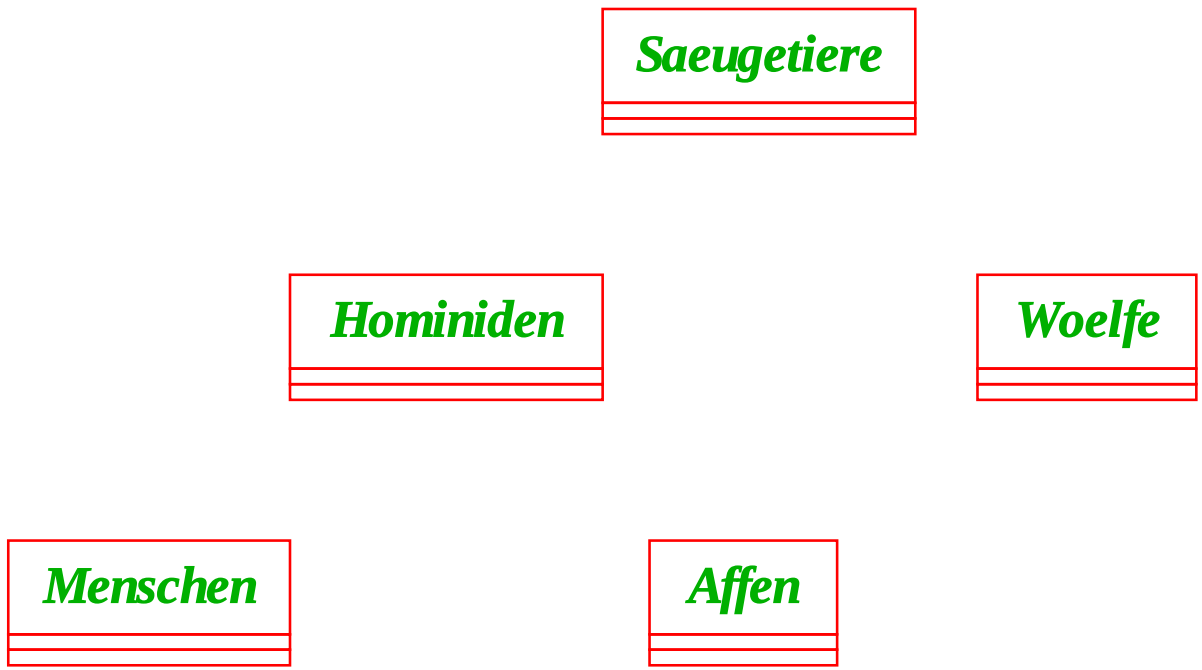
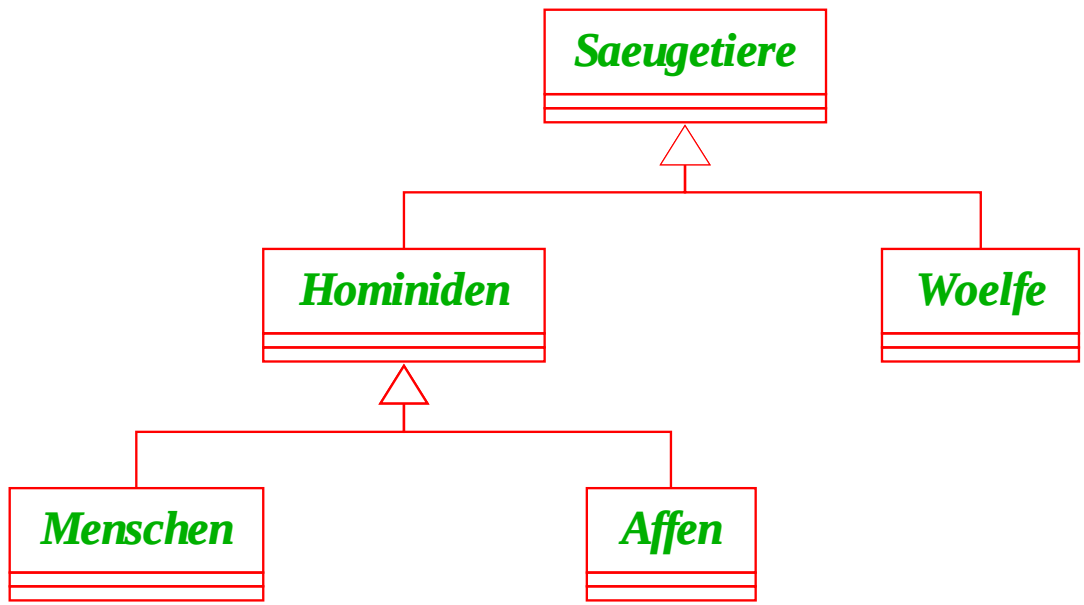
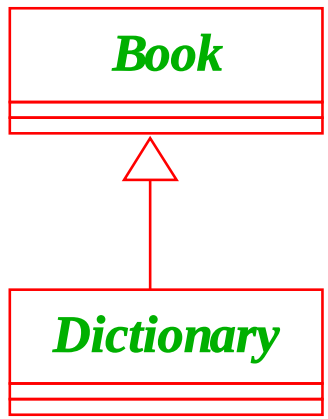
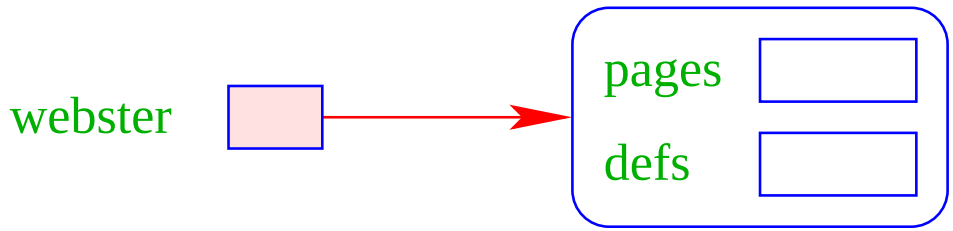
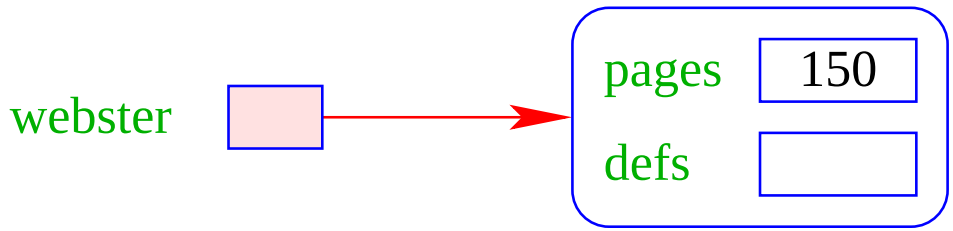
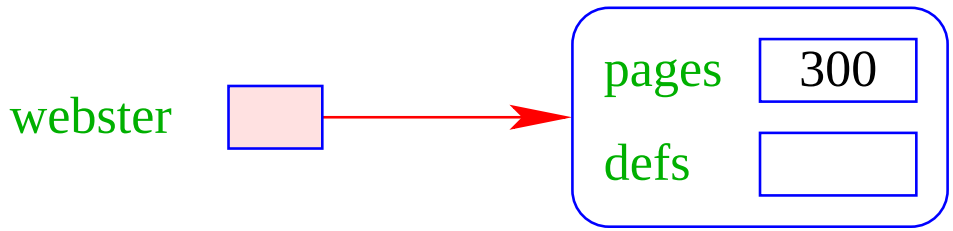
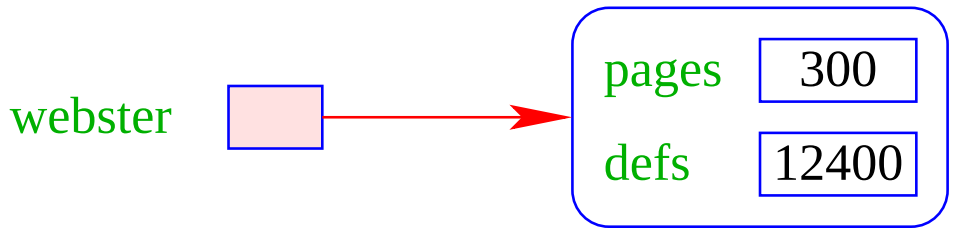
    return c;
}
```

Vektoren und Matrizen – Beispiele

- Matrixaddition

a11	a12		a1m
a21	a22		a2m
...	...	...	...
an1	an2		anm

+

b11	b12		b1m
b21	b22		b2m
...	...	...	...
bn1	bn2		bnm

Vektoren und Matrizen – Beispiele

- Matrixaddition



Vektoren und Matrizen – Beispiele

- Matrixaddition

a11	a12		a1m
a21	a22		a2m
...	...	...	...
an1	an2		anm

+

b11	b12		b1m
b21	b22		b2m
...	...	...	...
bn1	bn2		bnm

=

a11+b11			
a21+b21			
...	...	...	...
an1+bn1			

Vektoren und Matrizen – Beispiele

- Matrixaddition



Vektoren und Matrizen – Beispiele

- Matrixaddition



Vektoren und Matrizen – Beispiele

- Matrixaddition – Algorithmus

```
// Vorbedingung: Dimensionen der Matrizen sind identisch
public static int[][] addMatrices(int[][] a, int[][] b) {
   int[][] c = new int[a.length][a[0].length];

   for (int i = 0; i < a.length; i++) {
     c[i] = addVectors(a[i], b[i]);
   }

   return c;
}
```

Vektoren und Matrizen – Beispiele

- Matrixmultiplikation



Vektoren und Matrizen – Beispiele

- Matrixmultiplikation



Vektoren und Matrizen – Beispiele

- Matrixmultiplikation



Vektoren und Matrizen – Beispiele

- Matrixmultiplikation



Vektoren und Matrizen – Beispiele

- Matrixmultiplikation



Vektoren und Matrizen – Beispiele

- Matrixmultiplikation



Vektoren und Matrizen – Beispiele

- Matrixmultiplikation



Vektoren und Matrizen – Beispiele

- Matrixmultiplikation – Algorithmus

```
// Vorbedingung: Anzahl Spalten a == Anzahl Zeilen b
public static int[][] multMatrices(int[][] a, int[][] b) {
   int n = a.length;
   int m = a[0].length;              // int m = b.length;
   int p = b[0].length;

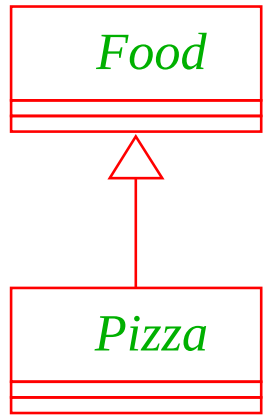
   int[][] c = new int[n][p];

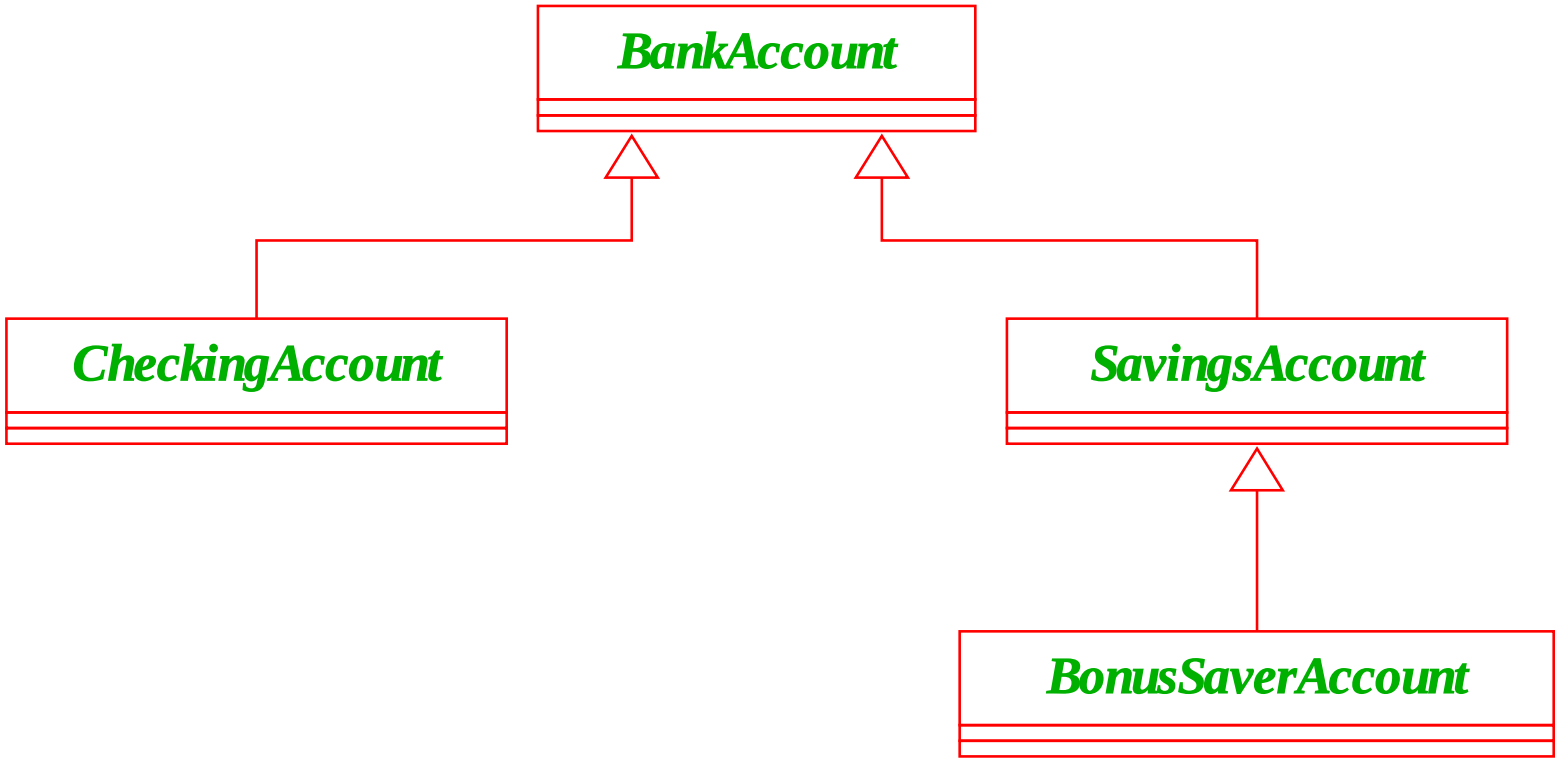
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < p; j++) {
         c[i][j] = 0;
         for (int k = 0; k < m; k++) {
            c[i][j] += a[i][k]*b[k][j];
         }
      }
   }

   return c;
}
```

Übersicht

Suchmaschine



Objekte identifizieren



Klassenentwurf



Objekterzeugung



Objektmethoden

Worthäufigkeiten



Vektorähnlichkeit

Dokumentensammlung

Suchmaschine: Sortieren der Vektoren



Suchmaschine: Sortieren der Vektoren



Suchmaschine: Sortieren der Vektoren



Suchmaschine: Sortieren der Vektoren



Suchmaschine: Sortieren der Vektoren



Suchmaschine: Sortieren der Vektoren



Suchmaschine: Sortieren der Vektoren



Suchmaschine: Sortieren der Vektoren



Suchmaschine: Sortieren der Vektoren



Suchmaschine: Vektorähnlichkeit

- Paarweise Berechnung einfacher Vektorähnlichkeit

	Query
class	0
compile	0
instance	0
java	0
method	0
object	1
oriented	1



Suchmaschine: Vektorähnlichkeit

- Paarweise Berechnung einfacher Vektorähnlichkeit

`similarity()`

Query	
class	0
compile	0
instance	0
java	0
method	0
object	1
oriented	1

Thinking in Java

class	62
compile	15
instance	3
java	62
method	30
object	98
oriented	26



Suchmaschine: Vektorähnlichkeit

- Paarweise Berechnung einfacher Vektorähnlichkeit



Suchmaschine: Vektorähnlichkeit

- Paarweise Berechnung einfacher Vektorähnlichkeit



Suchmaschine



Objekte identifizieren



Klassenentwurf



Objekterzeugung



Objektmethoden



Worthäufigkeiten

Dokumentensammlung

Sortieren nach Vektorähnlichkeit



Vektorähnlichkeit

Suchmaschine: Sortieren der Ergebnisse

- Sortieren der Ergebnisse nach Relevanz



Suchmaschine: Sortieren der Ergebnisse

- Sortieren der Ergebnisse nach Relevanz



Effective Java
Joshua Bloch
englisch
1.

Suchmaschine: Sortieren der Ergebnisse

- Sortieren der Ergebnisse nach Relevanz





Suchmaschine: Sortieren der Ergebnisse

- Sortieren der Ergebnisse nach Relevanz





Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

5. Einige Abstrakte Datentypen

Felder haben Nachteile

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

Idee: Listen



Members

- `info` == Wert des Listenelements;
- `next` == Verweis auf das nächste Element;

`null` bezeichnet das leere Objekt—und das Listenende.

Operationen:

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

Modellierung



Die Rauten-Verbindung heißt auch Aggregation.

Beispiel Einfügen



`t` ist ein Zeiger auf das "aktuelle" Listenelement. Kann (zusammen mit dem Zeiger auf den Beginn einer Liste) in einer dedizierten Wrapper-Klasse gespeichert werden!

Beispiel Einfügen



`t` ist ein Zeiger auf das "aktuelle" Listenelement. Kann (zusammen mit dem Zeiger auf den Beginn einer Liste) in einer dedizierten Wrapper-Klasse gespeichert werden!

Beispiel Löschen



`t` ist ein Zeiger auf das "aktuelle" Listenelement. Kann innerhalb oder außerhalb von Listenobjekten gespeichert werden!

Beispiel Löschen



`t` ist ein Zeiger auf das "aktuelle" Listenelement. Kann innerhalb oder außerhalb von Listenobjekten gespeichert werden!

Weitere Operationen

Weiterhin sollte man

- ... eine Liste auf Leerheit testen können;

Weitere Operationen

Weiterhin sollte man

- ... eine Liste auf Leerheit testen können;

Achtung:

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

Weitere Operationen

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;

Weitere Operationen

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.

Konstruktoren

```
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;
   }
   ...
}
```

Objektmethoden

```
// Objekt-Methoden:
   public void insert(int x) {
      next = new List(x,next);
   }
   public void delete() {
      // löscht das nächste (!) Element
      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+"]";
   }
   ...
```

Bemerkungen I

- 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 nichts.

Löschen kurz vor Listenende



Löschen kurz vor Listenende



Bemerkungen II: `null`

- 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)`.

Konvertierung in Strings




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

Konvertierung in Strings




```
"[7, 0, -3, 13]"
```

Konvertierung in Strings


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

Konvertierung in Strings

"null"

Klassenmethoden

```
// 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();
      }
   ...
```

Konvertierungsmethoden: Code

```
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;
}
...
```

Konvertierungsmethoden

- 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
```

Bemerkungen

- 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.

Übersicht

Suchmaschine



Objekte identifizieren



Klassenentwurf



Objekterzeugung

Objektmethoden

Worthäufigkeiten

Dokumentensammlung

Suchmaschine: Verweise

- Verweise auf mehrere andere Dokumente

Suchmaschine: Verweise

- Verweise auf mehrere andere Dokumente



Übersicht

Nochmals Sortieren

Anwendung: Mergesort – Sortieren durch Mischen

John von Neumann (1945)

Nochmals Sortieren: Mergesort

Eingabe: zwei sortierte Listen;
Ausgabe: eine gemeinsame sortierte Liste.

Tatsächlich funktioniert mergesort (auch) für eine unsortierte Liste. Das Vorgehen ist dann:

- Halbiere die Liste
- Sortiere beide Hälften separat
- Mische (merge) die beiden sortierten Hälften

Wir betrachten hier zunächst nur den letzten Schritt! Sortieren der beiden Hälften behandeln wir im Kapitel über Rekursion!

Funktionsweise Mergesort Schritt 3



Funktionsweise Mergesort Schritt 3



Funktionsweise Mergesort Schritt 3



Idee

- Konstruiere sukzessive die Ausgabe-Liste aus den 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.

Beispiel Mergesort Schritt 3



Beispiel Mergesort Schritt 3



Beispiel Mergesort Schritt 3



Beispiel Mergesort Schritt 3



Beispiel Mergesort Schritt 3



Beispiel Mergesort Schritt 3



Beispiel Mergesort Schritt 3



Beispiel Mergesort Schritt 3



Implementierung

- Lege einen Dummy-Listenknoten an;
- Traversiere die Listen `a` und `b`;
- Vergleiche jeweils die beiden aktuellen Knoten;
- Derjenige, der das kleinere Element enthält, wird an die Ausgabe-Liste angehängt;
- wird entweder in `a` oder in `b` das Ende erreicht, hängen wir den Rest der anderen Liste ebenfalls an die Ausgabe;
- Schließlich werfen wir den Dummy-Knoten am Anfang der Ausgabe weg.

Code Mergesort

```
public static List merge(List a, List b) {
   List result = new List(0);
   List t = result;
   while (true) {
      if (a == null) {
         t.next = b;
         break;
      }
      if (b == null) {
         t.next = a;
         break;
      }
      if (b.info > a.info) {
         t = t.next = a;
         a = a.next;
      } else {
         t = t.next = b;
         b = b.next;
      }
   } // end of while
   return result.next;
}  // end of merge
```

Nachteil

- Für jedes Element in `b` wird ein neuer Listen-Knoten angelegt.

Verbesserung:

- Verwende die gesamten Listen-Objekte wieder, d.h. modifiziere allein die `next`-Verweise!
- Benutze Rekursion (kurz vor Weihnachten!)

Diskussion

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

  Dann gilt:

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

- Dabei ist $V(n)$ der Sortieraufwand für eine Liste der Länge n (hier noch nicht betrachtet, kommt bei Rekursion!) und $2 \cdot n$ die Anzahl der Vergleiche.
- 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?)

Übersicht

Abstrakte Datentypen

- Spezifiziere nur die Operationen!
- Verbirg Details
  - der Datenstruktur;
  - der Implementierung der Operationen.
- `List` ist unser erster Abstrakter Datentyp!

$\Longrightarrow$ Information Hiding

Wozu?

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

Übersicht

Keller (Stacks)

Operationen:

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

Weiterhin müssen wir einen leeren Keller anlegen können.

Friedrich Ludwig Bauer, TUM

Modellierung



Erste Idee

- Realisiere Keller mithilfe einer Liste!



- Das Attribut `l` zeigt auf das oberste Element.

Modellierung



Die gefüllte Raute besagt, dass die Liste nur Teil eines `Stacks` sein darf und nur solange existiert, wie auch der Stack existiert.

Implementierung I

```
public class Stack {
   private List l;
   // Konstruktor:
   public Stack() {
      l = null;
   }
   // Objekt−Methoden:
   public boolean isEmpty() {
      return l == null;
   }
   ...
```

Implementierung II

```
    public int pop() {
        int result = l.info;
        l = l.next;
        return result;
    }
    public void push(int a) {
        l = new List(a,l);
    }
    public String toString() {
        return List.toString(l);
    }
} // end of class Stack
```

Bemerkungen

- Die Implementierung ist sehr einfach;
- ... nutzt 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!

Bemerkungen

- Die Implementierung ist sehr einfach;
- ... nutzt 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.

Beispielablauf




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

Beispielablauf




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

Beispielablauf



Beispielablauf



Beispielablauf



Modellierung



Implementierung I

```
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);
   }
   ...
```

Implementierung II

```
    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 alloziert, aber nie welcher freigegeben.

Erste Idee:

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

Beispielablauf



x

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

Beispielablauf



x 5

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

Beispielablauf




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


x 5

Beispielablauf



x 6

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

Beispielablauf




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

x 6

Beobachtung

- 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 !

Beispielablauf



x

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

Beispielablauf



x 3

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

Beispielablauf



s.push(5);

Beispielablauf



Analyse

- 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;
}
```

Übersicht

Operationen

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

Operationen:

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

Weiterhin müssen wir eine leere Schlange anlegen können.

Modellierung



Erste Idee

- Realisiere Schlange mithilfe einer Liste :



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

Modellierung



Objekte der Klasse `Queue` enthalten zwei Verweise auf Objekte der Klasse `List`.

Implementierung I

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

Implementierung II

```
    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
```

Analyse

- 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.

Analyse

- 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.

Modellierung



Beispielablauf `dequeue()`



Beispielablauf `dequeue()`



Beim Entfernen eines Elements wird also das `first`-Element nach rechts verschoben.
Falls `first==last` gilt, ist nach `dequeue()` die Schlange leer, und wir setzen `first=last=-1`.

Beispielablauf `enqueue()`



Beispielablauf `enqueue()`



Beim Hinzufügen eines Elements wird also der Wert an `a[++last]` eingefügt.

Feldgrenzen bei `enqueue()`

Was passiert, wenn `first` oder `last` das Ende des Feldes erreichen, also `first==a.length` oder `last==a.length` gilt?

- Erste Idee: Feldgröße verdoppeln!

Feldgrenzen bei `enqueue()`

Was passiert, wenn `first` oder `last` das Ende des Feldes erreichen, also `first==a.length` oder `last==a.length` gilt?

- Erste Idee: Feldgröße verdoppeln!
  Machen wir auch, wenn das Feld wirklich voll ist!
  Nachteil: Alle Elemente in `a` links von `first` (sofern es solche gibt) werden nie wieder befüllt werden können!

Feldgrenzen bei `enqueue()`

Was passiert, wenn `first` oder `last` das Ende des Feldes erreichen, also `first==a.length` oder `last==a.length` gilt?

- Erste Idee: Feldgröße verdoppeln!
  Machen wir auch, wenn das Feld wirklich voll ist!
  Nachteil: Alle Elemente in `a` links von `first` (sofern es solche gibt) werden nie wieder befüllt werden können!
- Deshalb zunächst Elemente in `a` links von `first` wiederverwenden!

Feldgrenzen bei `enqueue()`

Was passiert, wenn `first` oder `last` das Ende des Feldes erreichen, also `first==a.length` oder `last==a.length` gilt?

- Erste Idee: Feldgröße verdoppeln!
  Machen wir auch, wenn das Feld wirklich voll ist!
  Nachteil: Alle Elemente in `a` links von `first` (sofern es solche gibt) werden nie wieder befüllt werden können!
- Deshalb zunächst Elemente in `a` links von `first` wiederverwenden!
- Wenn also am Anfang des Felds (also ab Index 0) noch Platz ist, zunächst diese Zellen verwenden.
  Also Springen von `a.length-1` auf `0` — mit dem modulo-Operator `%`.

Feldgrenzen bei `enqueue()`

Was passiert, wenn `first` oder `last` das Ende des Feldes erreichen, also `first==a.length` oder `last==a.length` gilt?

- Erste Idee: Feldgröße verdoppeln!
  Machen wir auch, wenn das Feld wirklich voll ist!
  Nachteil: Alle Elemente in `a` links von `first` (sofern es solche gibt) werden nie wieder befüllt werden können!
- Deshalb zunächst Elemente in `a` links von `first` wiederverwenden!
- Wenn also am Anfang des Felds (also ab Index 0) noch Platz ist, zunächst diese Zellen verwenden.
  Also Springen von `a.length-1` auf `0` — mit dem modulo-Operator `%`.
- Ähnliches Phänomen bei `dequeue()`: Rechtsverschiebung von `first`, wenn die Feldgrenze bereits erreicht ist.

`enqueue()`: Erreichen des Feldendes

Unter der Annahme unbenutzter Elemente am Feldanfang:




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

enqueue(): Erreichen des Feldendes

Unter der Annahme unbenutzter Elemente am Feldanfang:



dequeue(): Erreichen des Feldendes



dequeue(): Erreichen des Feldendes



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, müssen `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.
  (Achtung: Es gilt nicht notwendigerweise `first==0`!)
- 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.

Implementierung von enqueue()

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

Implementierung von dequeue()

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

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

Diskussion

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

  Achtung:

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

Zusammenfassung

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

  $\Longrightarrow$ 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.

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

6. Objektorientierung

- Vererbung
  - Private Variablen und Methoden
  - Das Schlüsselwort `super`
  - Überschreiben von Attributen und Methoden
- Polymorphie
  - Unterklassen-Polymorphie
  - Generische Klassen
  - Wrapper-Klassen
- Abstrakte Klassen, finale Klassen und Interfaces
- Beispiel Suchmaschine

Beobachtung

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



Idee

- Finde Gemeinsamkeiten heraus und organisiere sie in einer Hierarchie!
- Implementiere zuerst das, was allen gemeinsam ist!
- Implementiere dann nur noch den Unterschied!

$\Longrightarrow$ inkrementelles Programmieren

$\Longrightarrow$ Software Reuse



Prinzip

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

Beispiel:



Implementierung I

```
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
...
```

Implementierung II

```
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
```

Bemerkungen

- `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.

new Dictionary(12400)

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

liefert:

new Dictionary(12400)

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

liefert:



new Dictionary(12400)

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

liefert:



new Dictionary(12400)

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

liefert:



new Dictionary(12400)

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

liefert:



Methodenverwendung

```
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
```

Sichtbarkeit und Modifier

Die Sichtbarkeit von *Attributen* und *Methoden* wird durch *Modifiers* bestimmt:

	Klasse	Package	Unterklasse	Welt
`private`	+	–	–	–
Default[1]	+	+	–[2]	–
`protected`	+	+	+	–
`public`	+	+	+	+

[1]: ohne Schlüsselwort
[2]: außer, wenn Unterklassen im gleichen Package

$\Longrightarrow$ Mittel zur Realisierung des Geheimnisprinzips

Sichtbarkeit und Modifier – Beispiel

```
class Shape {
  public double area() { ...
    // aufwaendige Berechnung der Flaeche (Integral)
  }
}

class Circle extends Shape {
  private double radius;

  public Circle(double r) {
    radius = r;
  }

  @Override
  public double area() {
    return Math.PI * radius * radius;
  }
}
```

Private Variablen und Methoden



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

Code 1

```
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
```

Code 2

```
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
```

Code 3 und Sichtbarkeiten

```
public class Pizza extends Food {
    public Pizza (int amount_fat) {
        super(amount_fat,8);  // kommt gleich! Geduld!
    }
} // 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 (wieso?)

Code 3 und Sichtbarkeiten

```
public class Pizza extends Food {
    public Pizza (int amount_fat) {
        super(amount_fat,8);  // kommt gleich! Geduld!
    }
} // 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 (wieso?)

Ausgabe des Programms: `Calories per serving: 309`

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 erneut das Schlüsselwort `super`.

... im Dictionary-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` I: Haben wir schon gesehen ...

- `super(...);` ruft den entsprechenden Konstruktor der Oberklasse auf.
- Analog gestattet `this(...);` es, den entsprechenden Konstruktor der eigenen Klasse aufzurufen.
- Ein solcher expliziter Aufruf muss stets ganz am Anfang eines Konstruktors stehen.

super II

Angenommen, eine Klasse A deklariert einen member memb desselben Namens wie in einer Oberklasse von A:

- Falls memb eine Methode mit demselben Rückgabetypen und denselben Argumenttypen in derselben Reihenfolge ist, so ist in A zunächst nur memb aus A sichtbar (Überschreiben).
- Falls memb eine Methode mit unterschiedlichen Argumenttypen oder -anzahl ist, so sind in A sowohl memb der Oberklasse als auch memb aus A sichtbar (Überladen).
- (Weitere Fälle folgen!)
- Wieso "zunächst" im ersten Fall?
  `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.

Anwendung von `super` im Beispiel

```
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` und `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
```

Überschreiben und Überladen

- Falls memb ein Attribut ist, so ist in A zunächst nur noch der Member memb aus A sichtbar.
- Falls memb eine Methode mit demselben Rückgabetypen und denselben Argumenttypen in derselben Reihenfolge ist, so ist in A zunächst ebenfalls nur memb aus A sichtbar (Überschreiben).
- Falls memb eine Methode mit unterschiedlichen Argumenttypen oder -anzahl ist, so sind in A sowohl memb der Oberklasse als auch memb aus A sichtbar (Überladen; kennen wir schon von Konstruktoren und von Methoden innerhalb einer Klasse).
- Falls memb eine Methode mit unterschiedlichem Rückgabetypen, aber denselben Argumenttypen in derselben Reihenfolge ist, so gibt es einen Compilerfehler.
- Mit `super` kann auf members der Oberklasse zugegriffen werden.

Verschatten von Attributen I

Wenn Attribute desselben Namens und Types in einer Kindklasse redefiniert werden, verschatten sie das Attribut der Elternklasse, so wie das für Methoden ebenfalls passiert:

```
public class Parent {
  protected int a;
  public int getA() {
    return a;  //diese Klasse!
  }
}
public class Child extends Parent {
  protected int a;
  public void setA(int arg) {
    a=arg;   // bezieht sich auf diese Klasse
  }
  public void setParentA(int arg) {
    super.a=arg;
  }
}
public class TestClass {
  public static void main(String[] argv) {
    Child c = new Child();
    c.setA(2);
    c.setParentA(3);
    System.out.println(c.getA()+"--"+c.a);
  }
}
```

Verschatten von Attributen I

Wenn Attribute desselben Namens und Types in einer Kindklasse redefiniert werden, verschatten sie das Attribut der Elternklasse, so wie das für Methoden ebenfalls passiert:

```
public class Parent {
  protected int a;
  public int getA() {
    return a;  //diese Klasse!
  }
}
public class Child extends Parent {
  protected int a;
  public void setA(int arg) {
    a=arg;   // bezieht sich auf diese Klasse
  }
  public void setParentA(int arg) {
    super.a=arg;
  }
}
public class TestClass {
  public static void main(String[] argv) {
    Child c = new Child();
    c.setA(2);
    c.setParentA(3);
    System.out.println(c.getA()+"--"+c.a);
  }
}
```

... liefert 3−−2 als Ausgabe !!!

Verschatten von Attributen II: getA() in Child

```
public class Parent {
  protected int a;
  public int getA() {
    return a;  //diese Parent–Klasse!
  }
}
public class Child extends Parent {
  protected int a;
  public void setA(int arg) {
    a=arg;   // bezieht sich auf diese Klasse
  }
  // NEUE METHODE!
  public int getA() {
    return a;  //a dieser Child–Klasse!
  }
  public void setParentA(int arg) {
    super.a=arg;
  }
}
public class TestClass {
  public static void main(String[] argv) {
    Child c = new Child();
    c.setA(2);
    c.setParentA(3);
    System.out.println(c.getA()+"--"+c.a);
  }
}
```

Verschatten von Attributen II: getA() in Child

```
public class Parent {
  protected int a;
  public int getA() {
    return a;  //diese Parent–Klasse!
  }
}
public class Child extends Parent {
  protected int a;
  public void setA(int arg) {
    a=arg;   // bezieht sich auf diese Klasse
  }
  // NEUE METHODE!
  public int getA() {
    return a;  //a dieser Child–Klasse!
  }
  public void setParentA(int arg) {
    super.a=arg;
  }
}
public class TestClass {
  public static void main(String[] argv) {
    Child c = new Child();
    c.setA(2);
    c.setParentA(3);
    System.out.println(c.getA()+"--"+c.a);
  }
}
```

... liefert mit der neuen `getA()`-Methode in `Child` jetzt `2--2` als Ausgabe!

Verschatten von Attributen: Unterschiedliche Typen

```
public class one {
  public static void main(String[] argv) {
    H h = new H();
    h.setA(17.3f);
    float a_in_H = h.getA();
    System.out.println("a in H: "+a_in_H);
  }
}
class G {
  int a;
}
class H extends G {
  float a;
  void setA(float arg) {
    a=arg;       // a is this.a
    super.a=(int) arg*2;
  }
  float getA() {
    System.out.println("a in G: "+super.a);
    return a;
  }
}
```

Verschatten von Attributen: Unterschiedliche Typen

```
public class one {
  public static void main(String[] argv) {
    H h = new H();
    h.setA(17.3f);
    float a_in_H = h.getA();
    System.out.println("a in H: "+a_in_H);
  }
}
class G {
  int a;
}
class H extends G {
  float a;
  void setA(float arg) {
    a=arg;      // a is this.a
    super.a=(int) arg*2;
  }
  float getA() {
    System.out.println("a in G: "+super.a);
    return a;
  }
}
```

... liefert, ganz ähnlich, als Ausgabe

```
a in G: 34
a in H: 17.3
```

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) {
      // ID 4321, initiale Summe 5028.45, 2% Zinsen
      Savings_Account savings =
         new Savings_Account (4321, 5028.45, 0.02);
      // ID 6543, initiale Summe 1475.85, 2% Zinsen
      Bonus_Saver_Account big_savings =
         new Bonus_Saver_Account (6543, 1475.85, 0.02);
      // ID 9876, initiale Summe 269.93, gehört zu savings
      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.

Code

```
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");
   }
   ...
```

Erläuterungen

- 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.

Erläuterungen

- 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.

(Finden Sie das Sicherheitsproblem?)

Abheben: Code

```
    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
```

Abheben

- 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;
   }
   ...
```

Modifizierte Abhebung

```
// 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
```

Erläuterungen

- 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
```

Erläuterungen

- 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);
   }
// Zinsen
   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
```

Bekannte Konten und Bewegungen

```
public class Bank {
   public static void main(String[] args) {
      // ID 4321, initiale Summe 5028.45, 2% Zinsen
      Savings_Account savings =
         new Savings_Account (4321, 5028.45, 0.02);
      // ID 6543, initiale Summe 1475.85, 2% Zinsen
      Bonus_Saver_Account big_savings =
         new Bonus_Saver_Account (6543, 1475.85, 0.02);
      // ID 9876, initiale Summe 269.93, gehört zu savings
      Checking_Account checking =
         new Checking_Account (9876, 269.93, savings);
      ...
      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
```

Beispielausgabe

```
Deposit into account 4321
Amount:         148.04
New balance:    5176.49

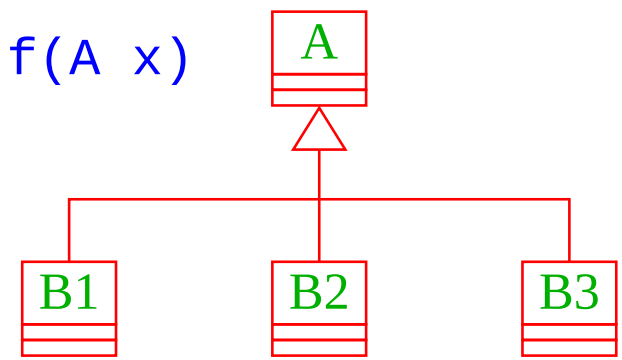
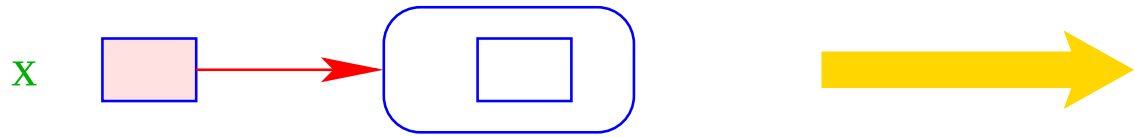
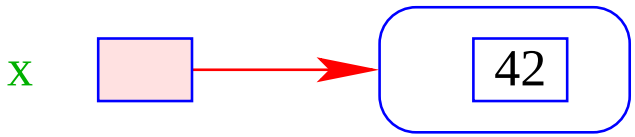
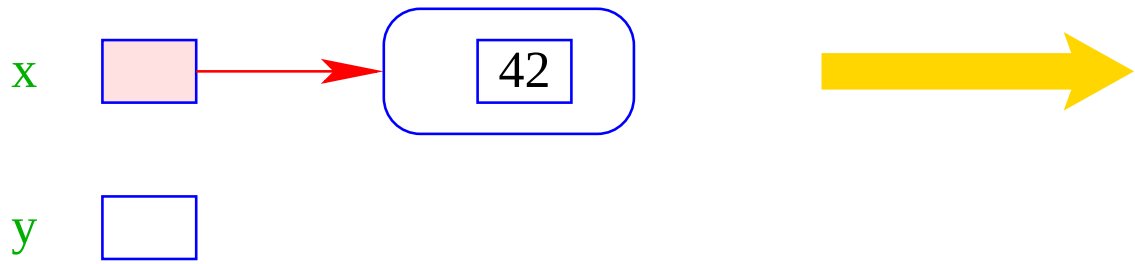
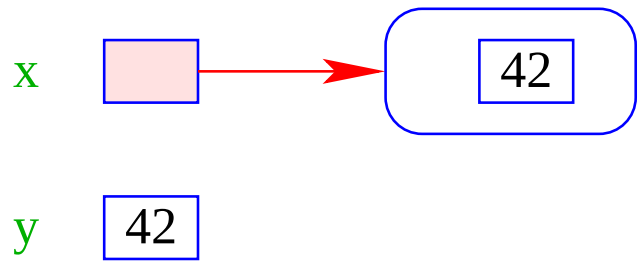
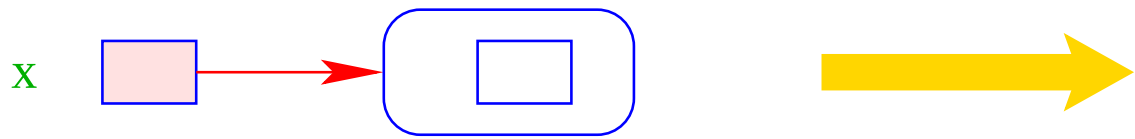
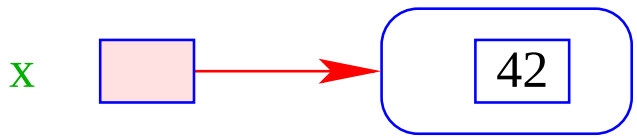
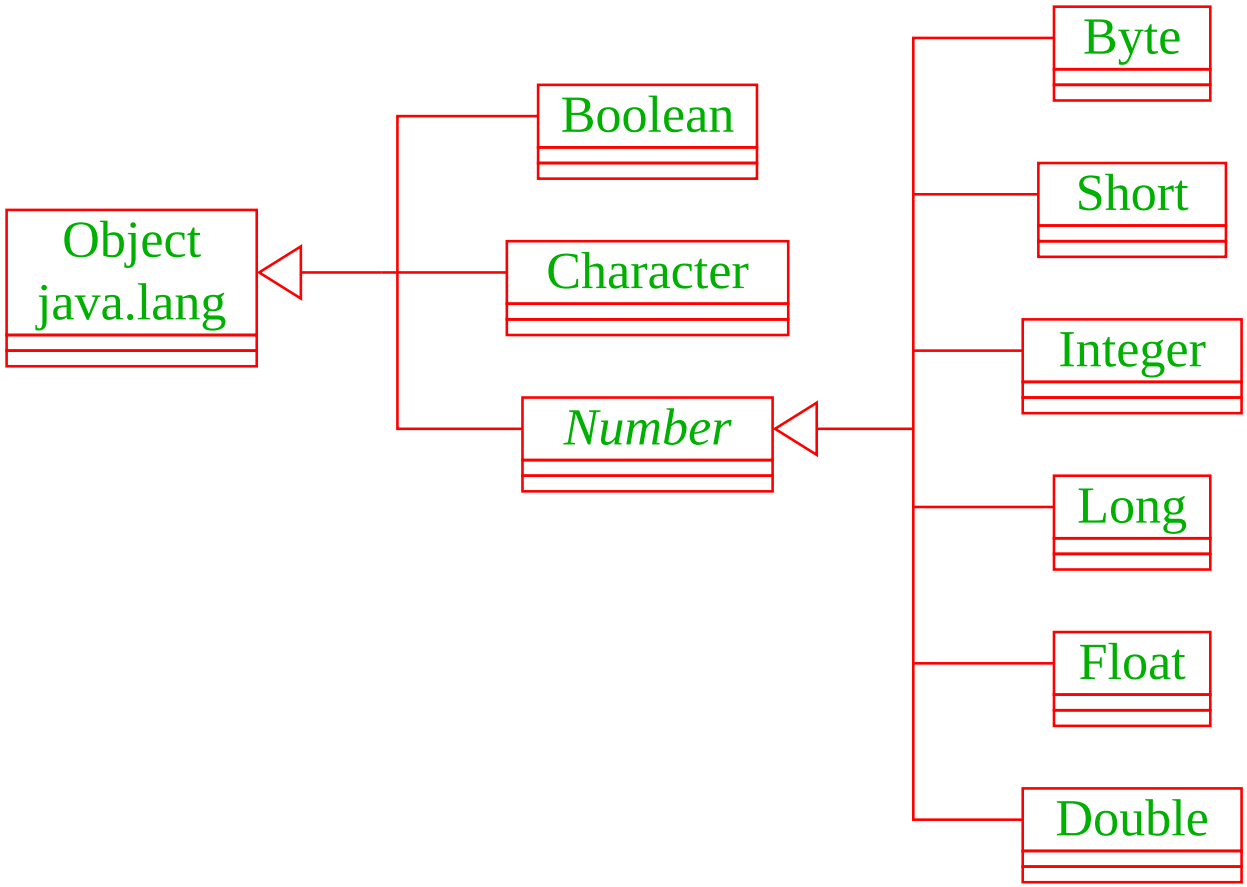
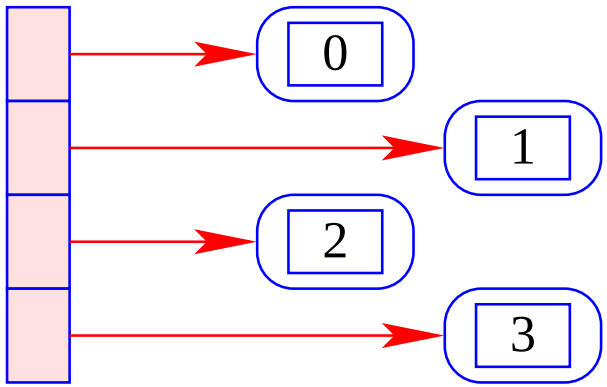
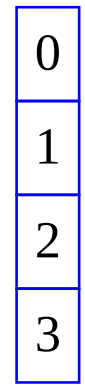
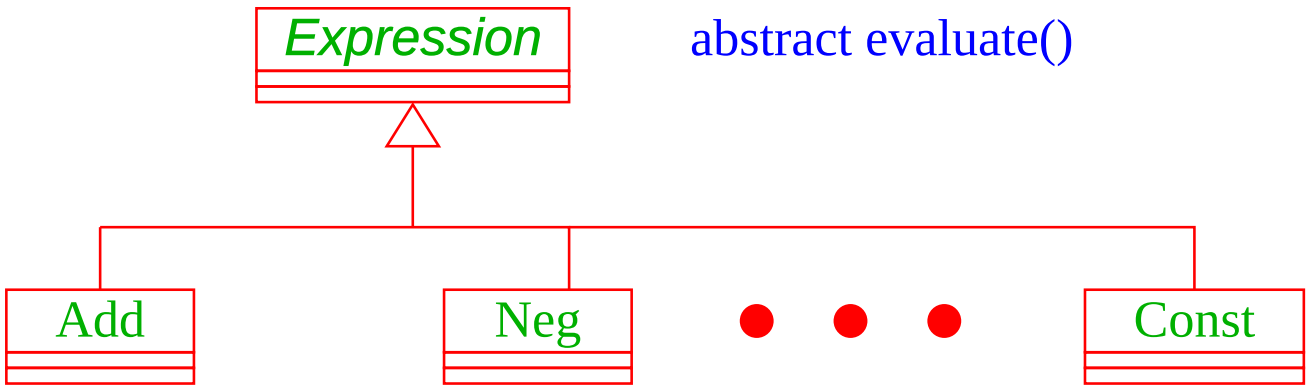
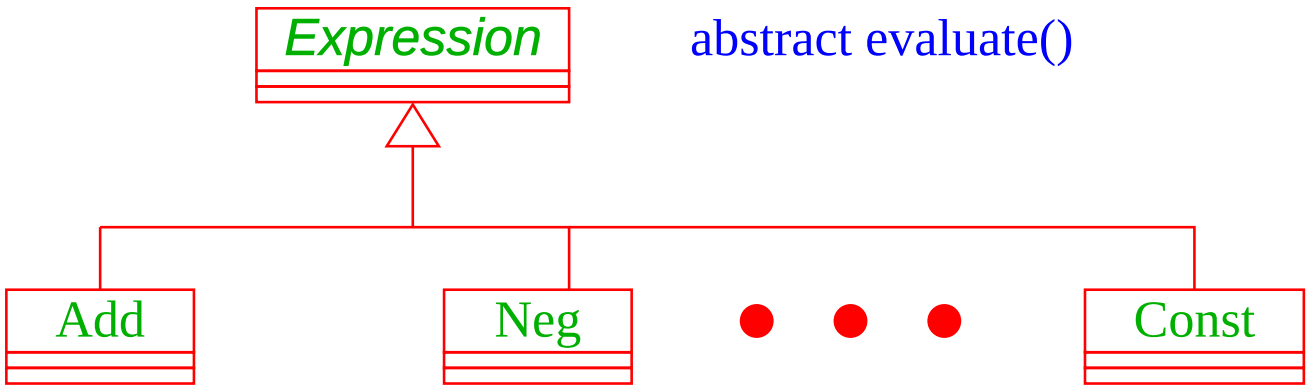
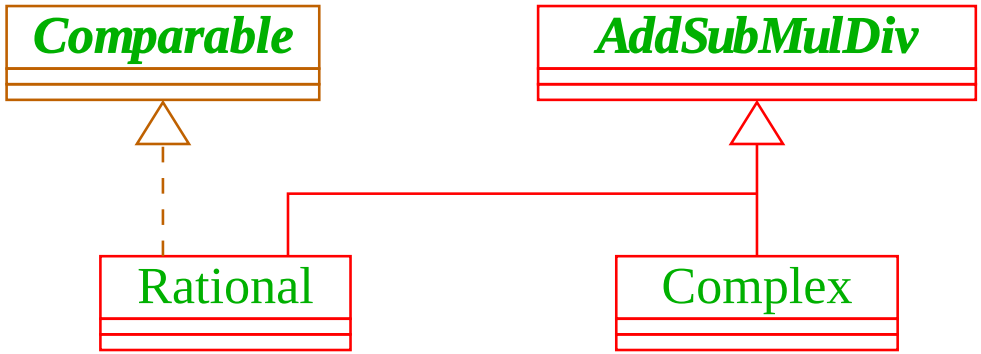
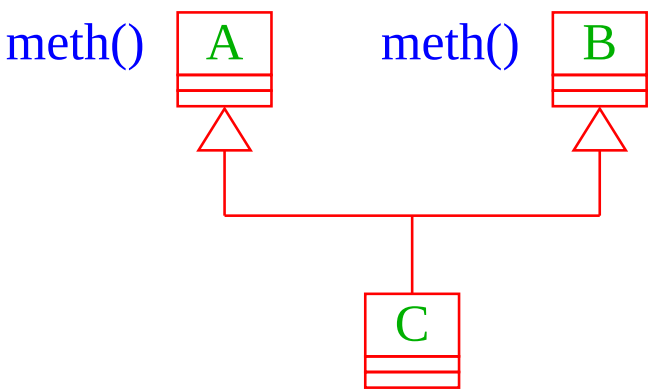
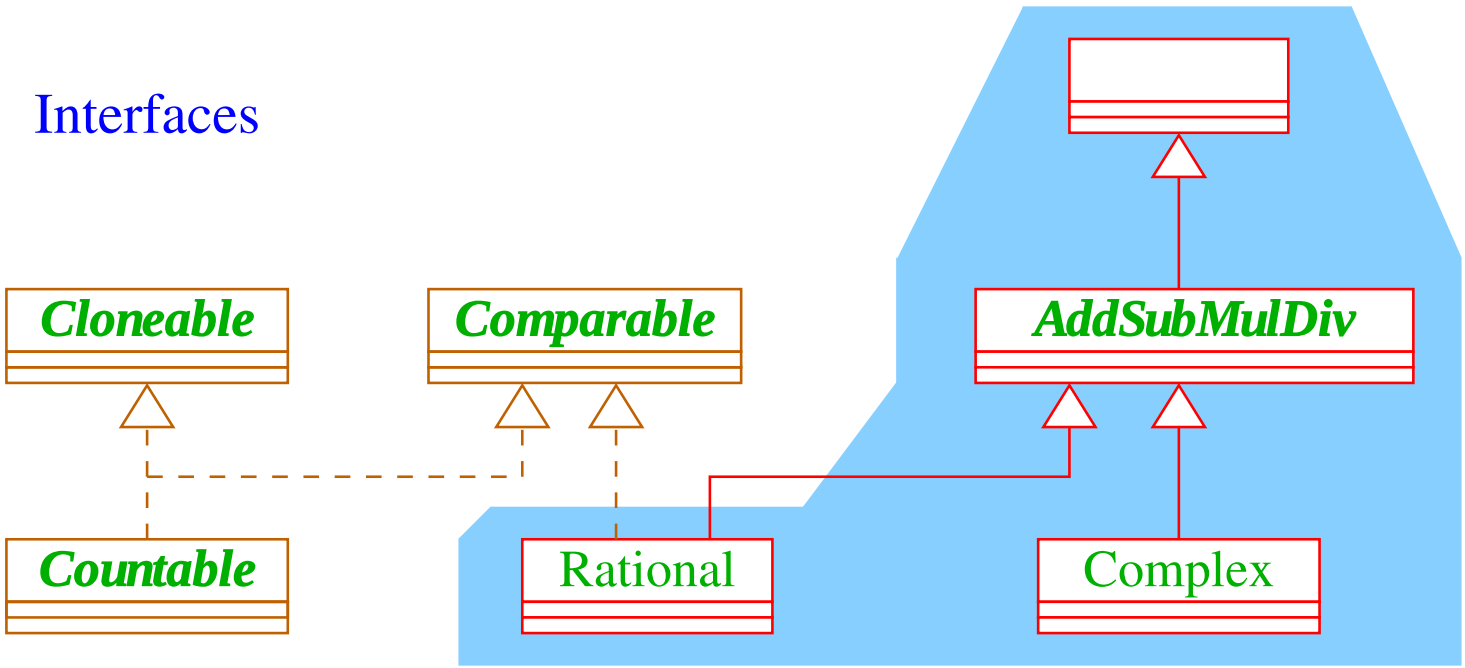
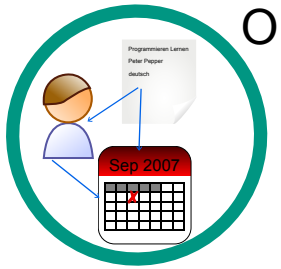
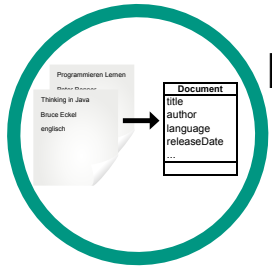
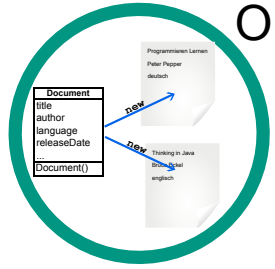
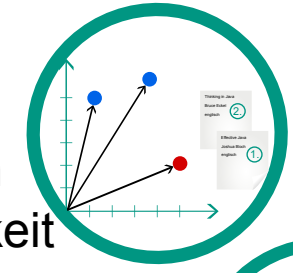
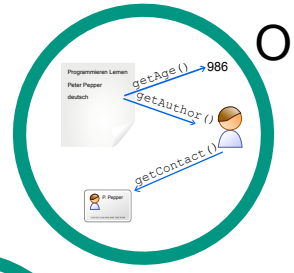
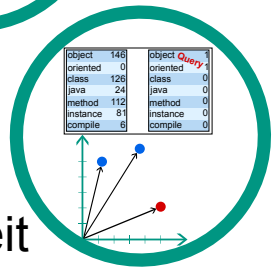
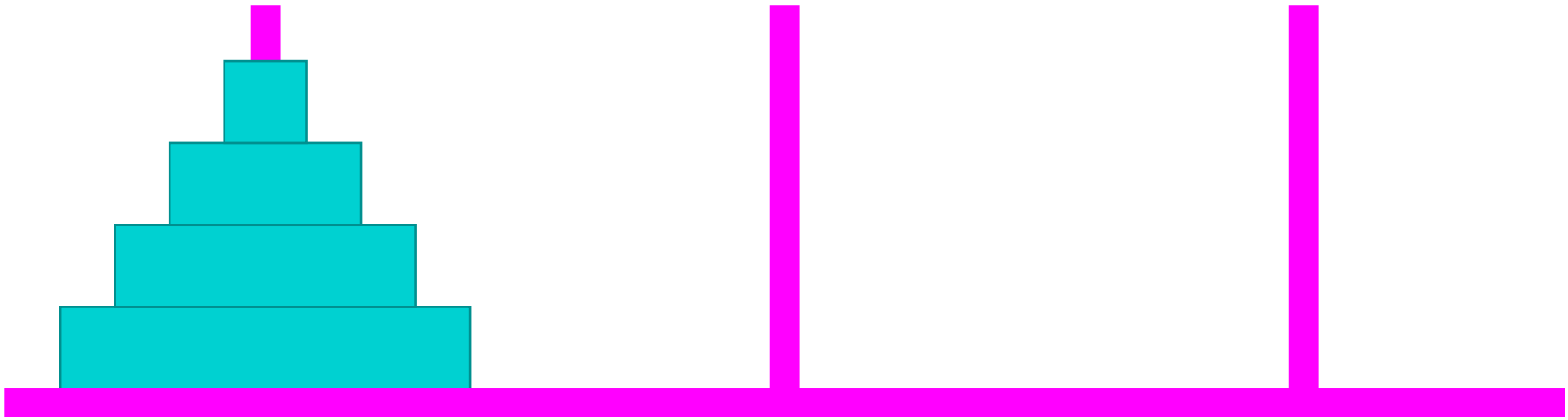
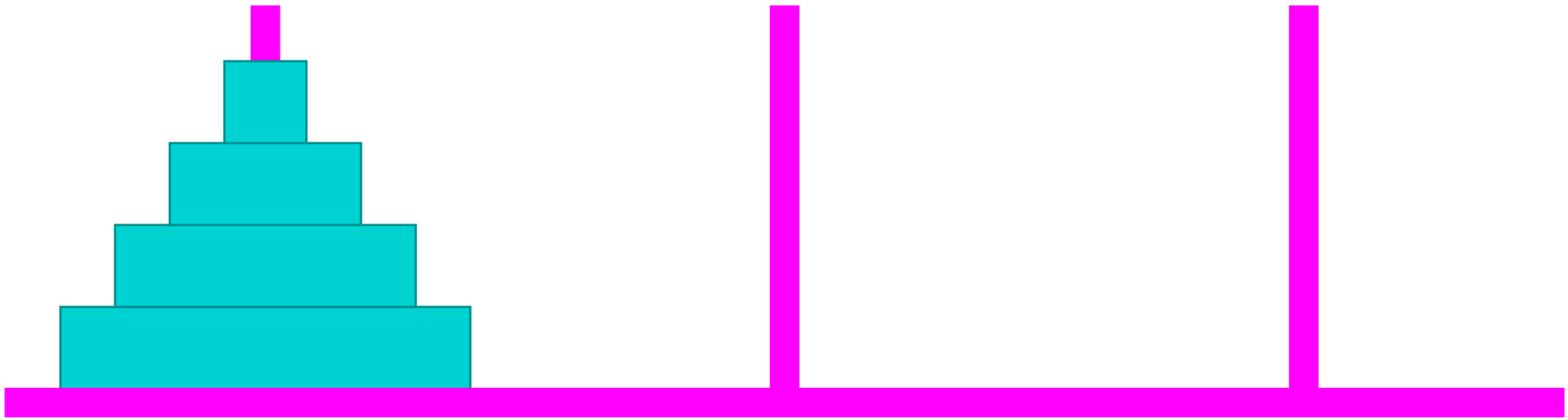
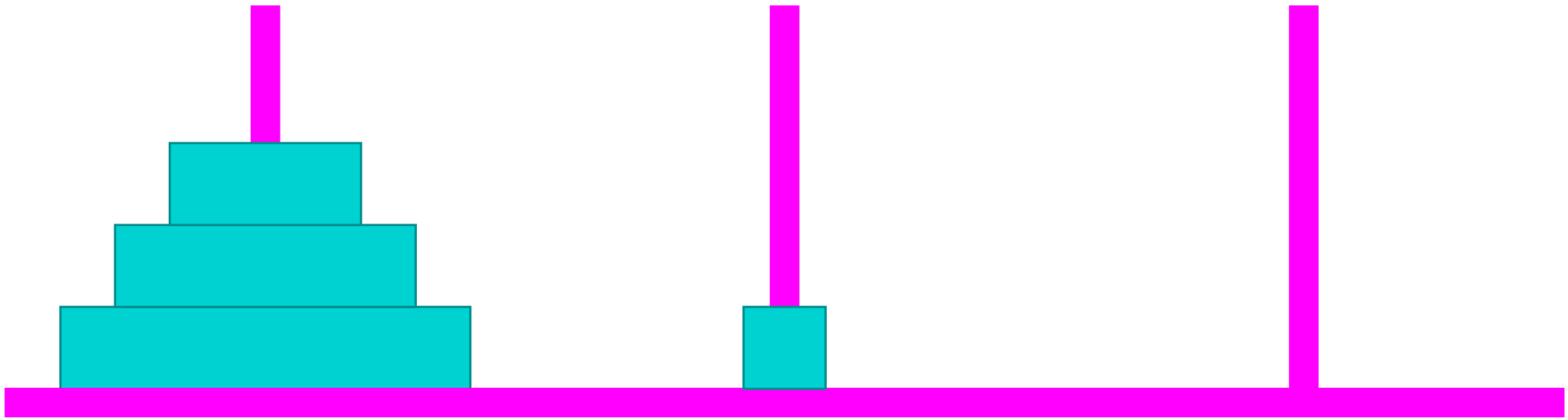
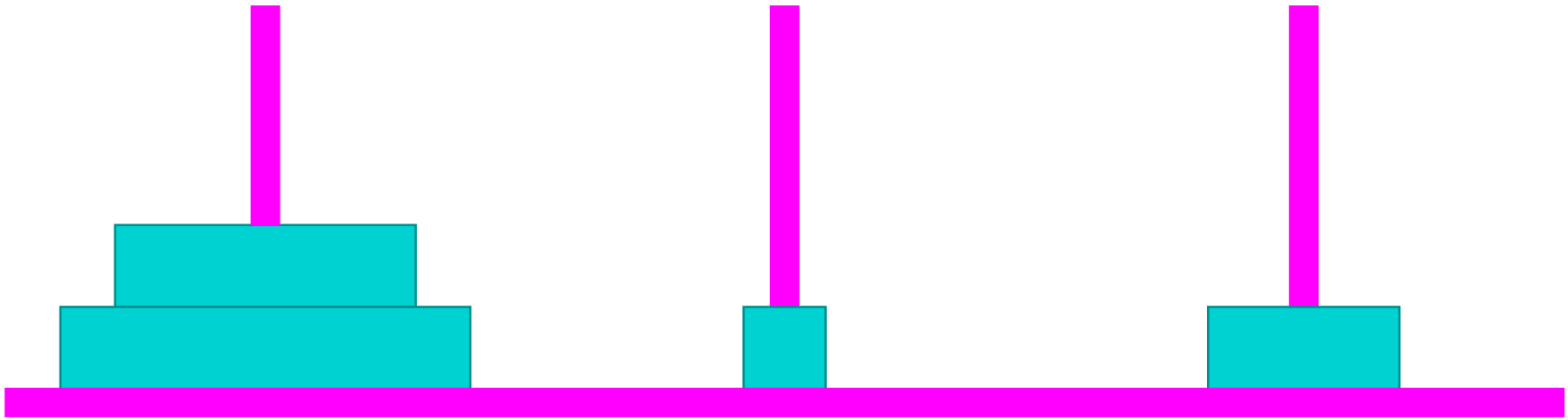
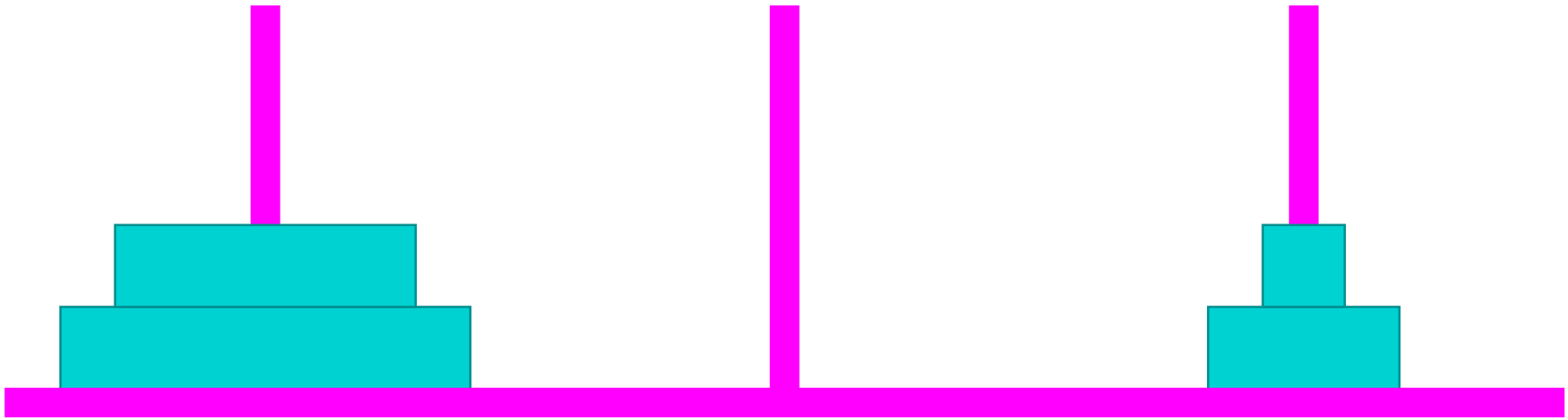
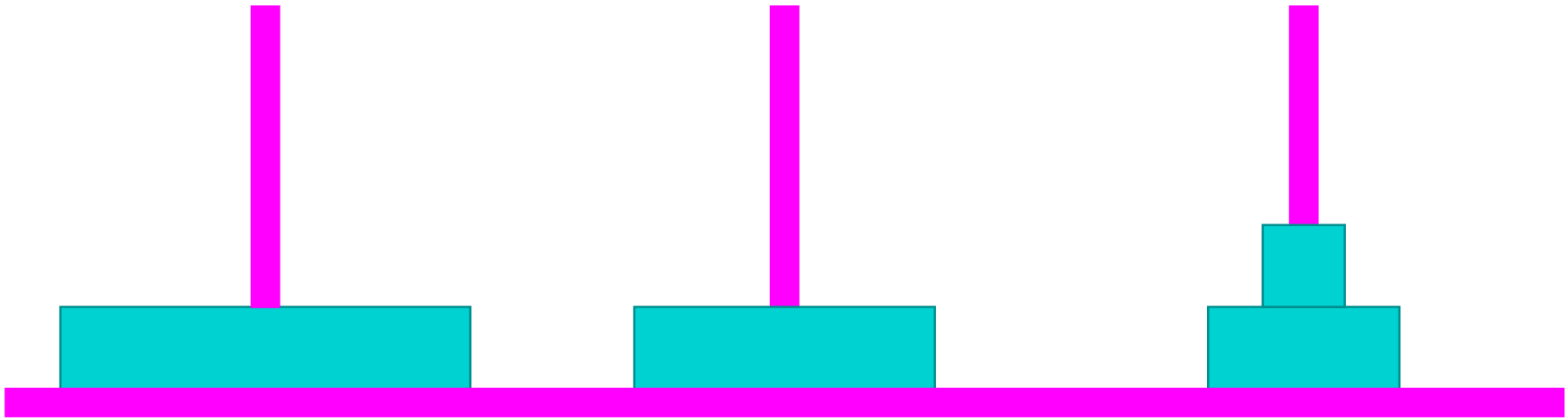
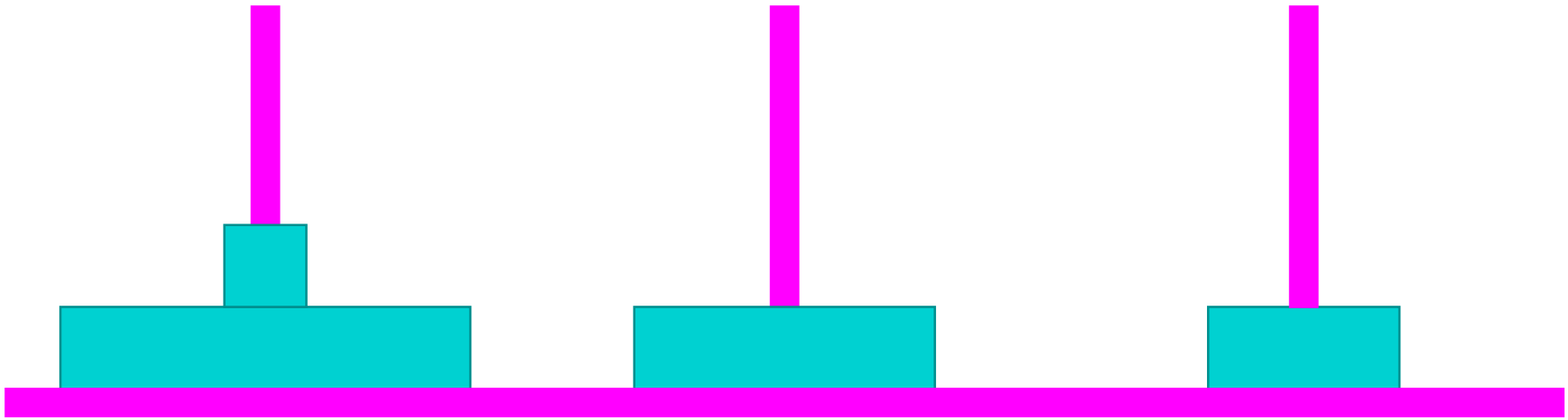
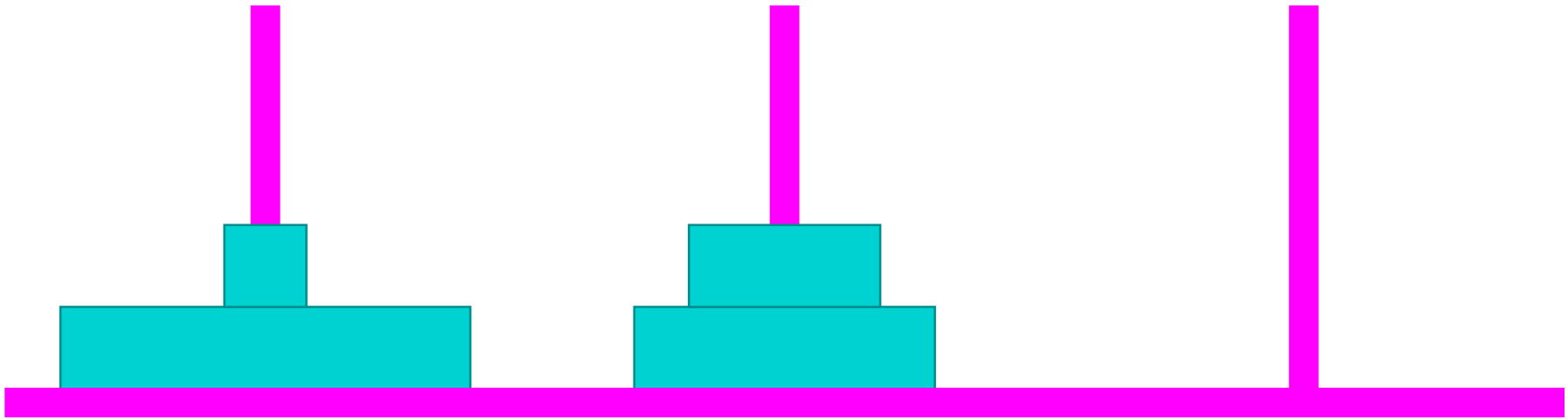
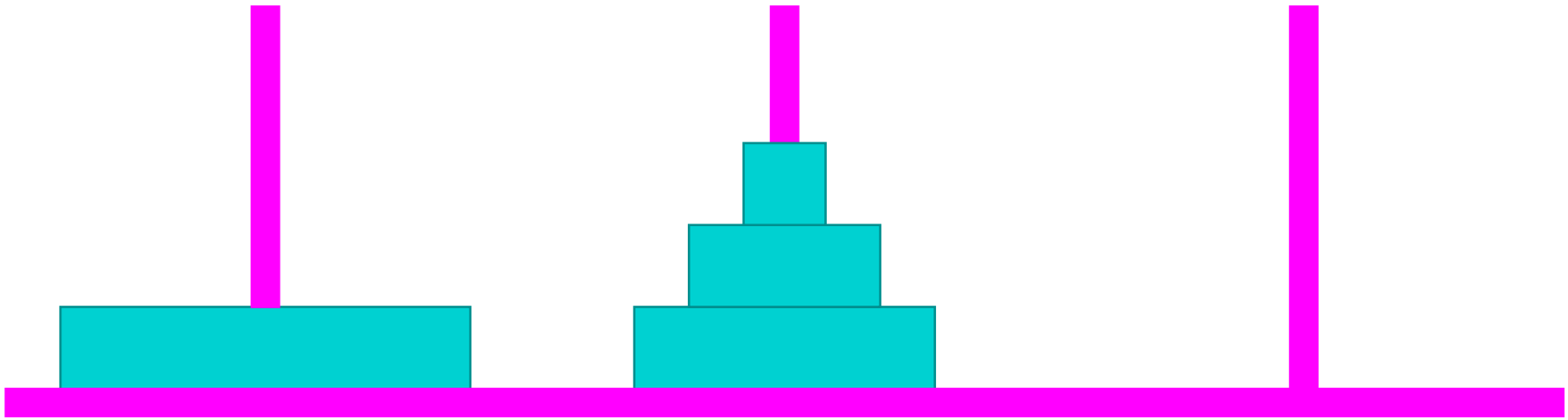
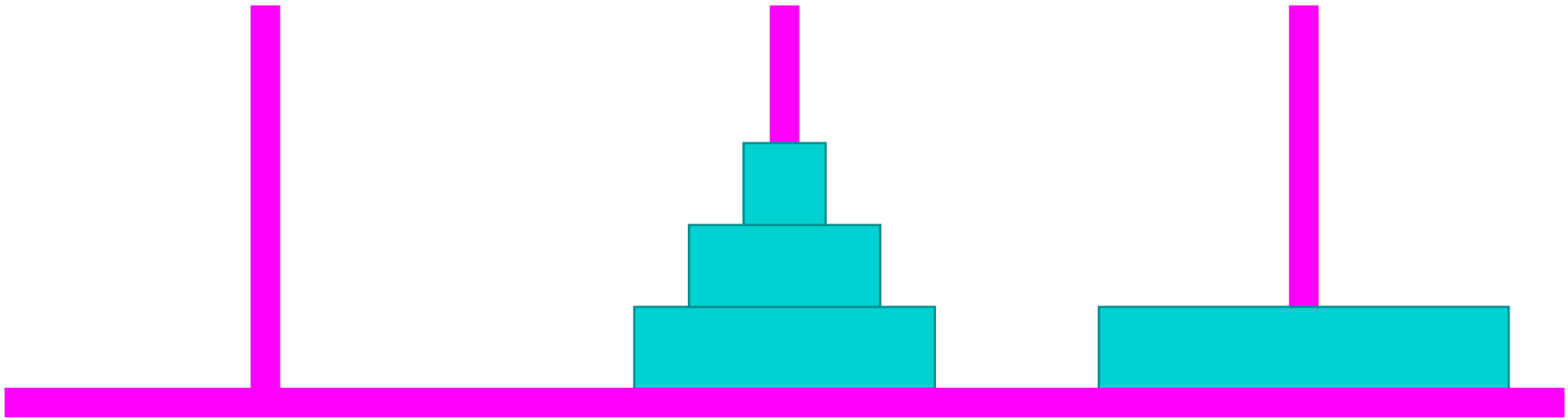
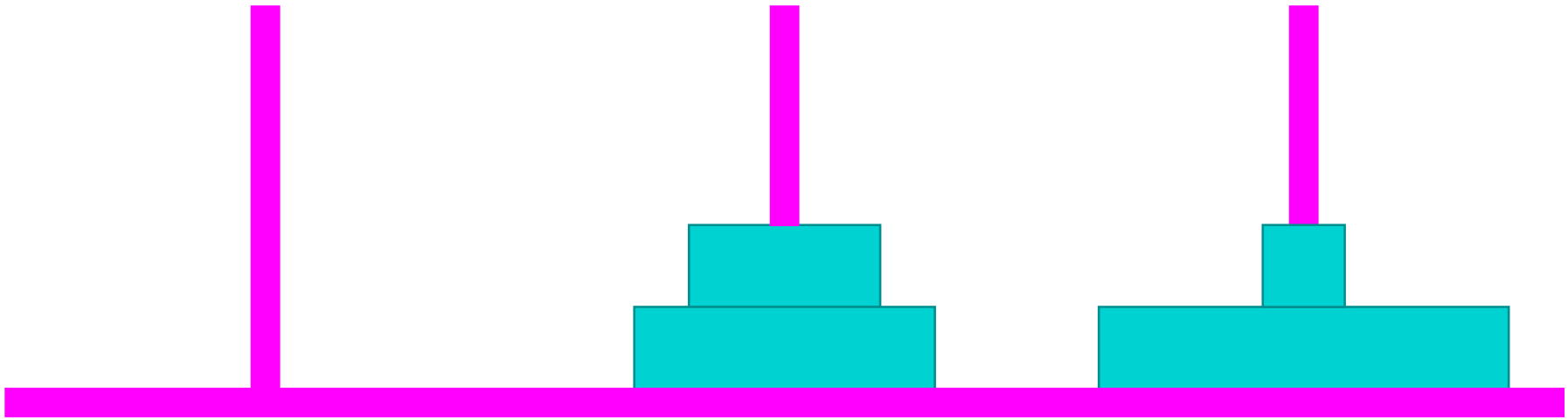
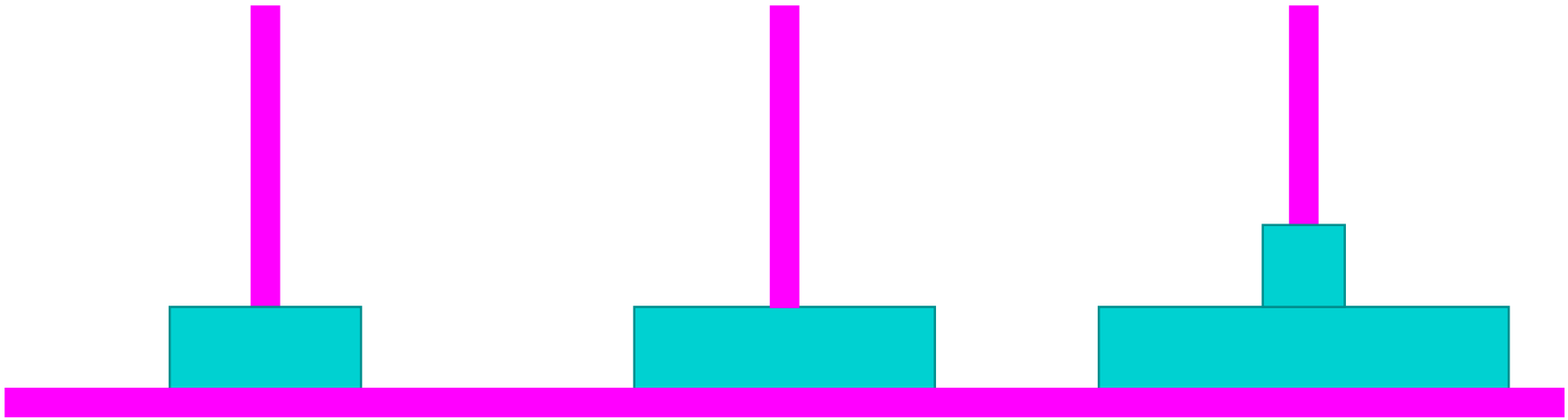
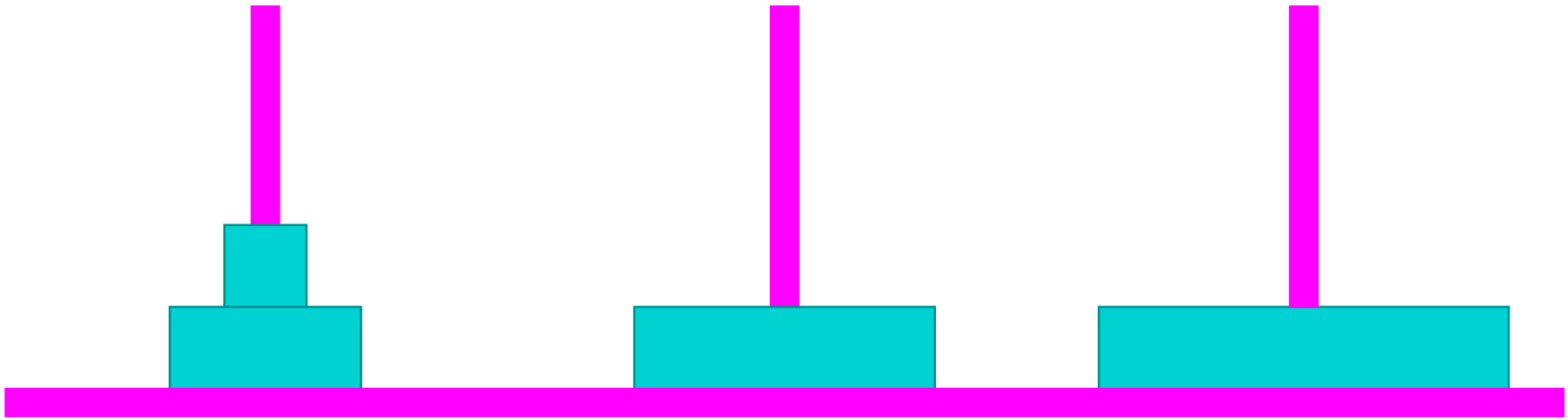
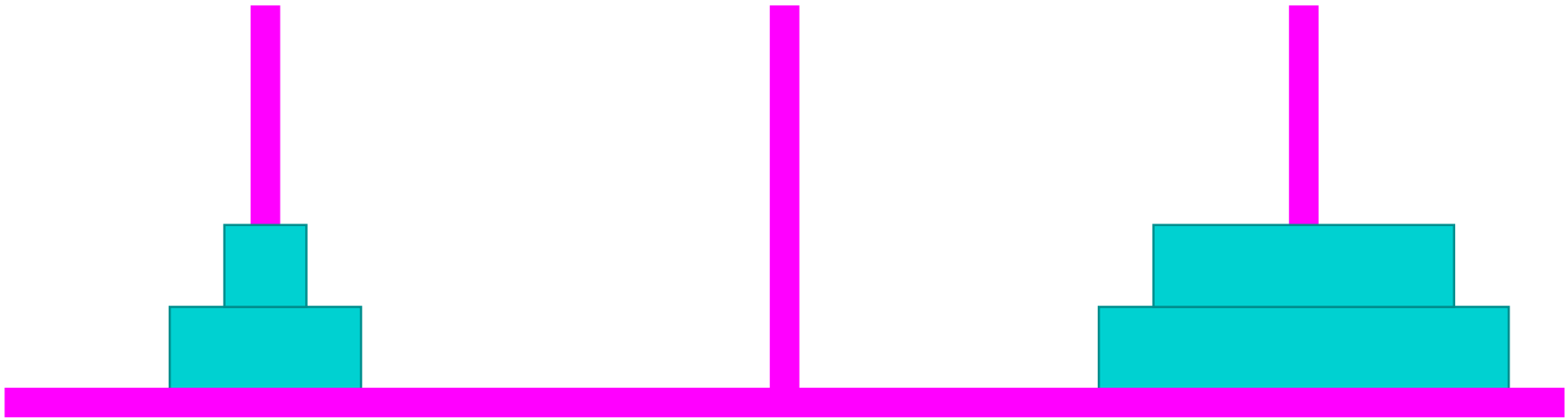
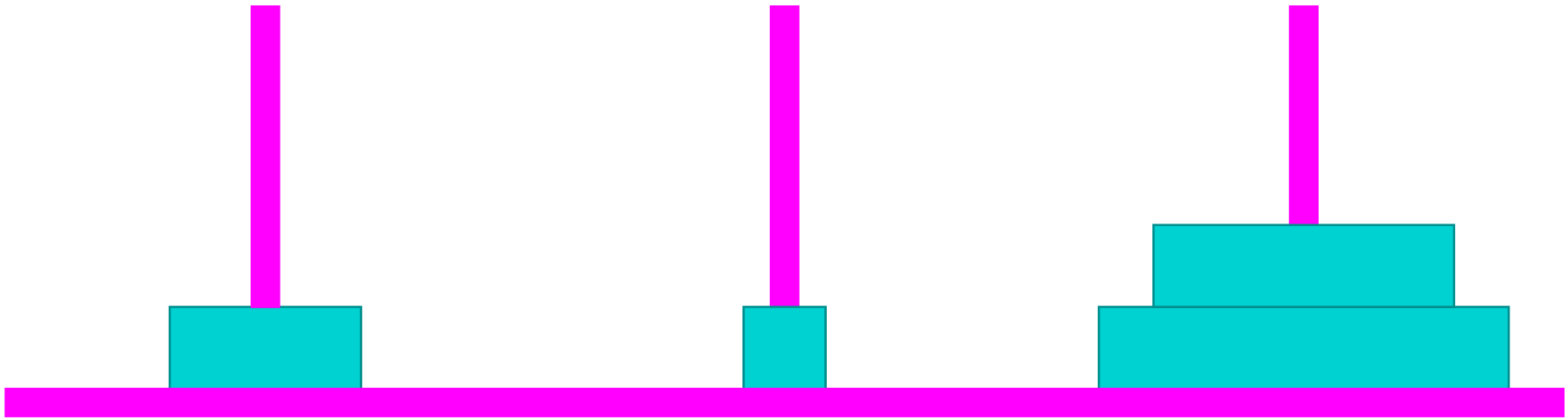
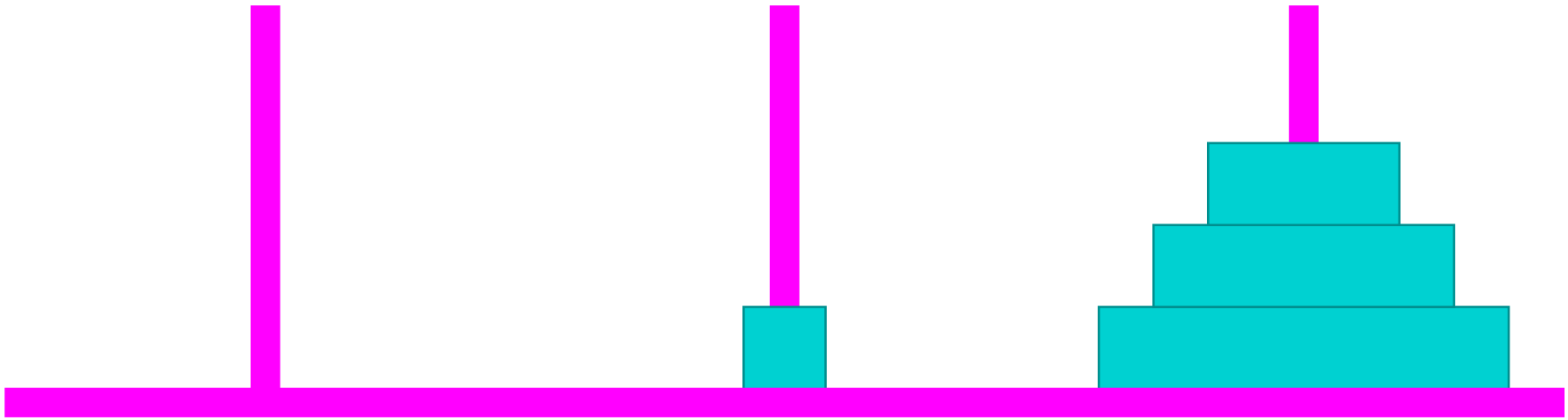
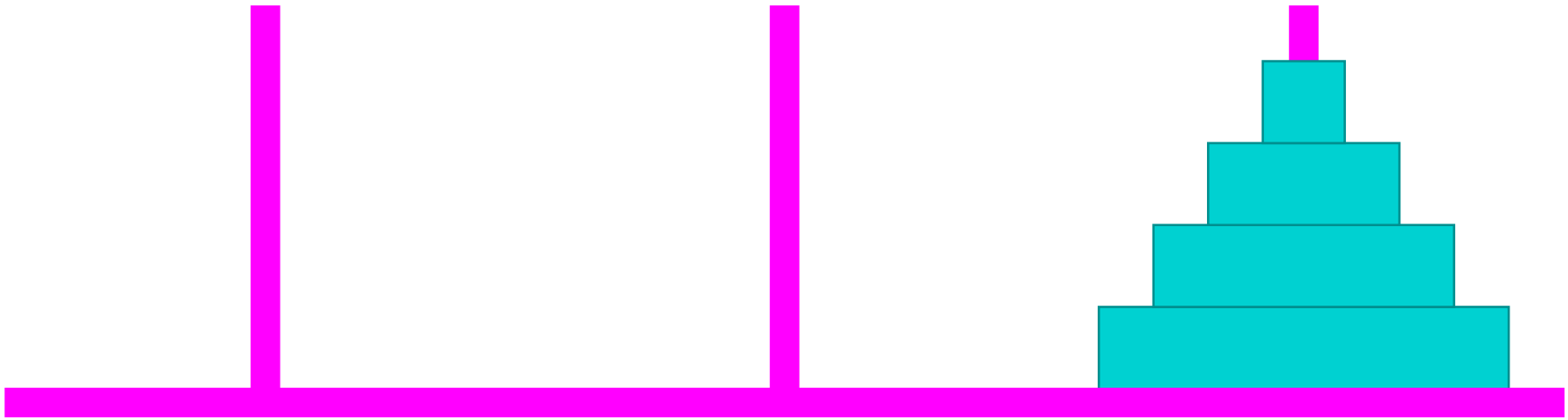
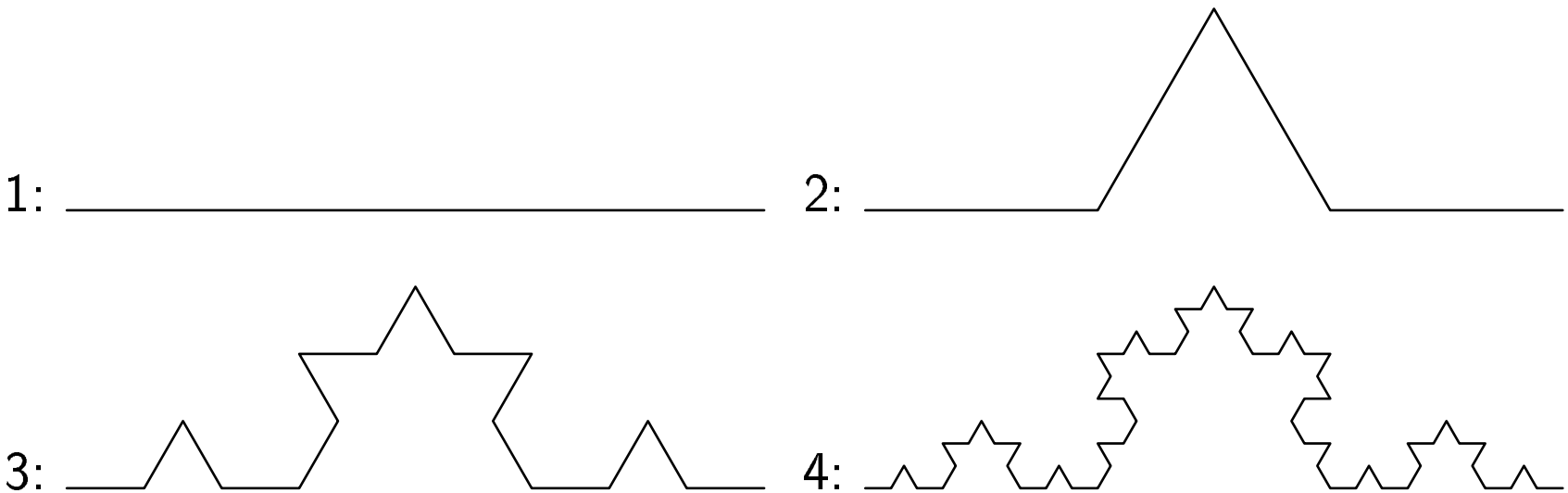
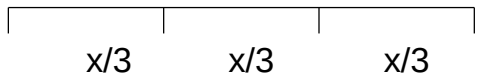
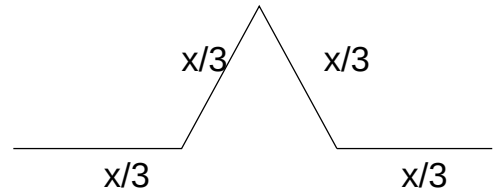
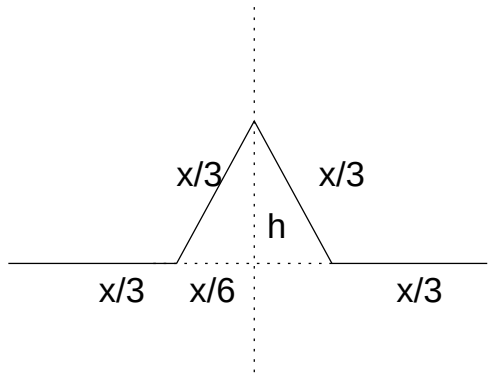
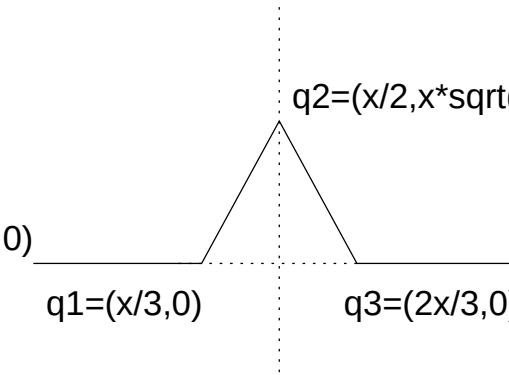
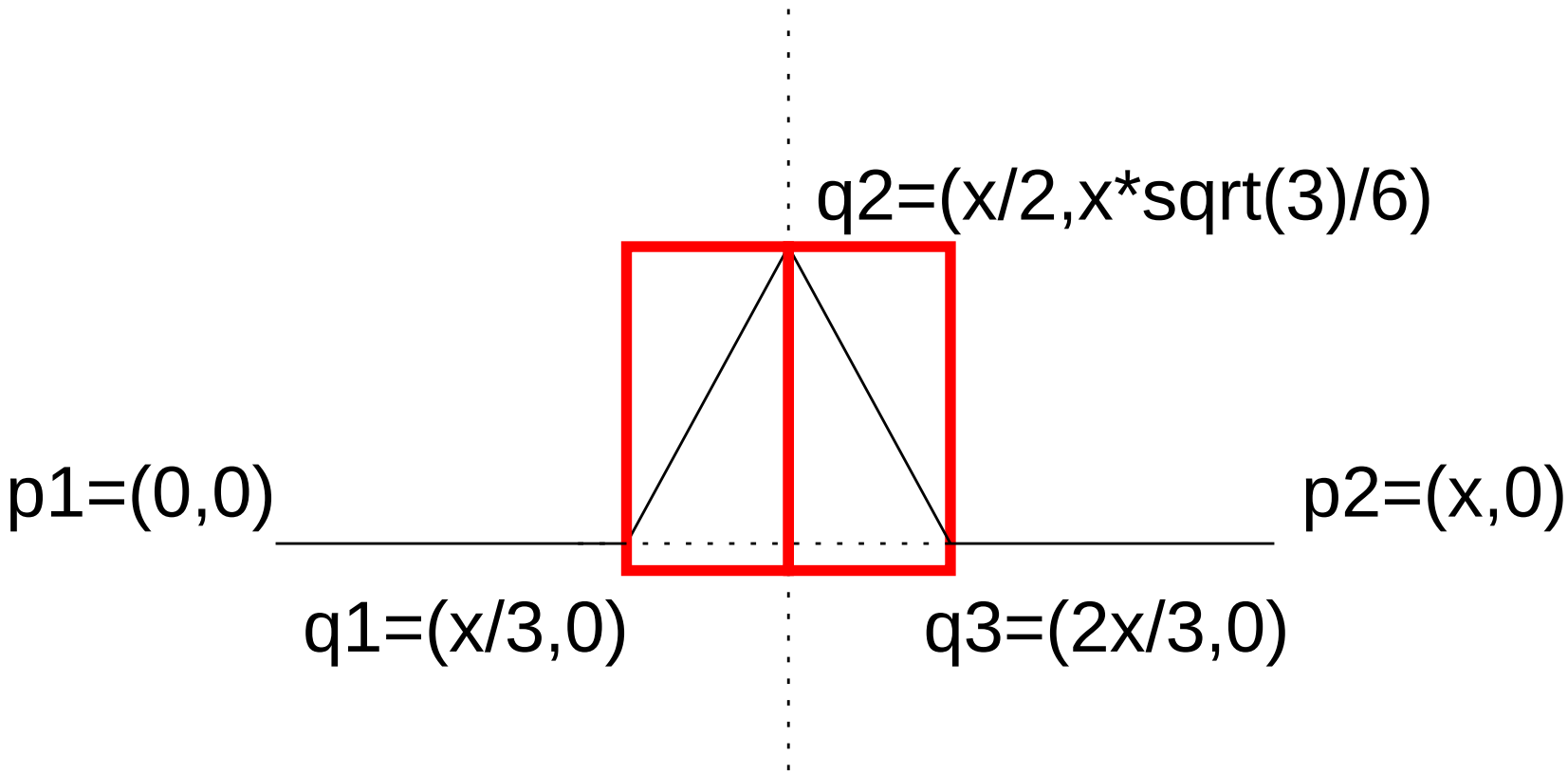
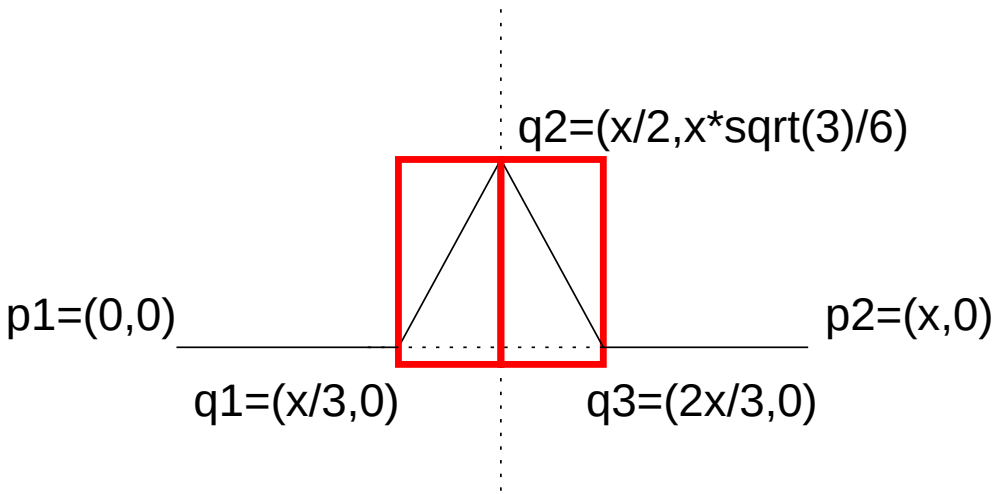
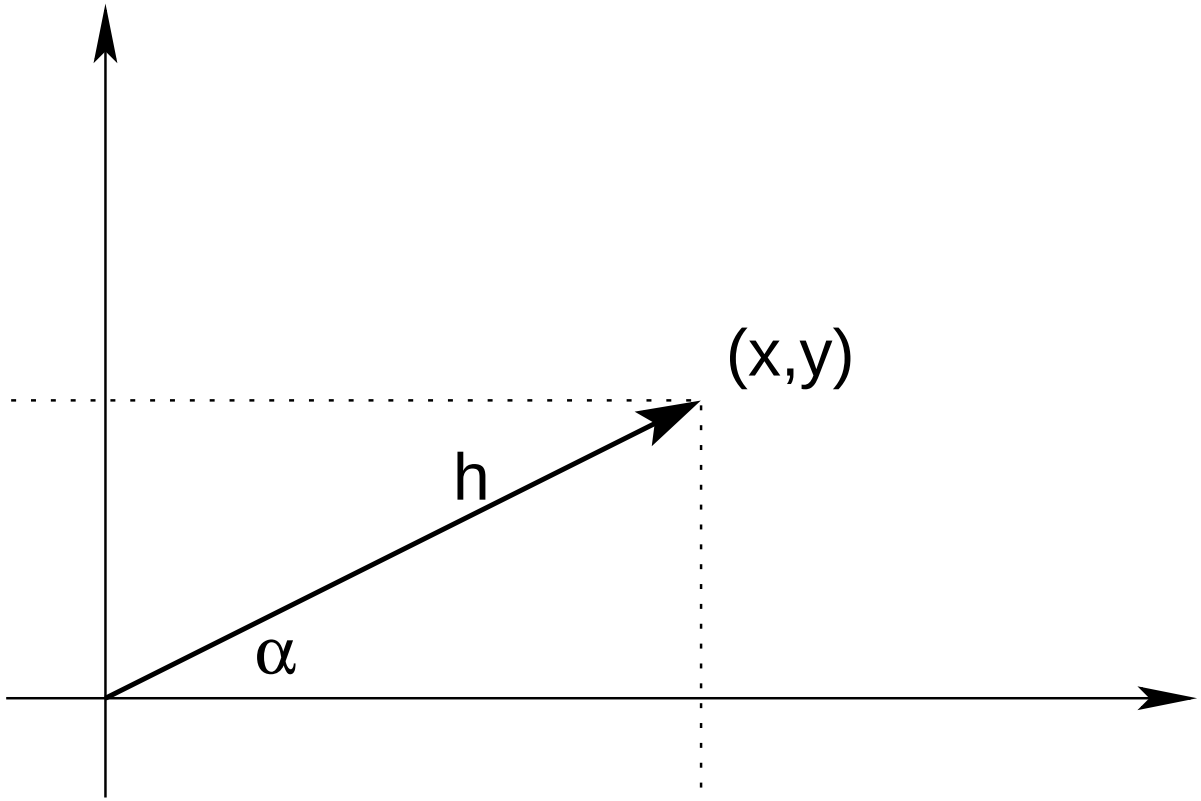
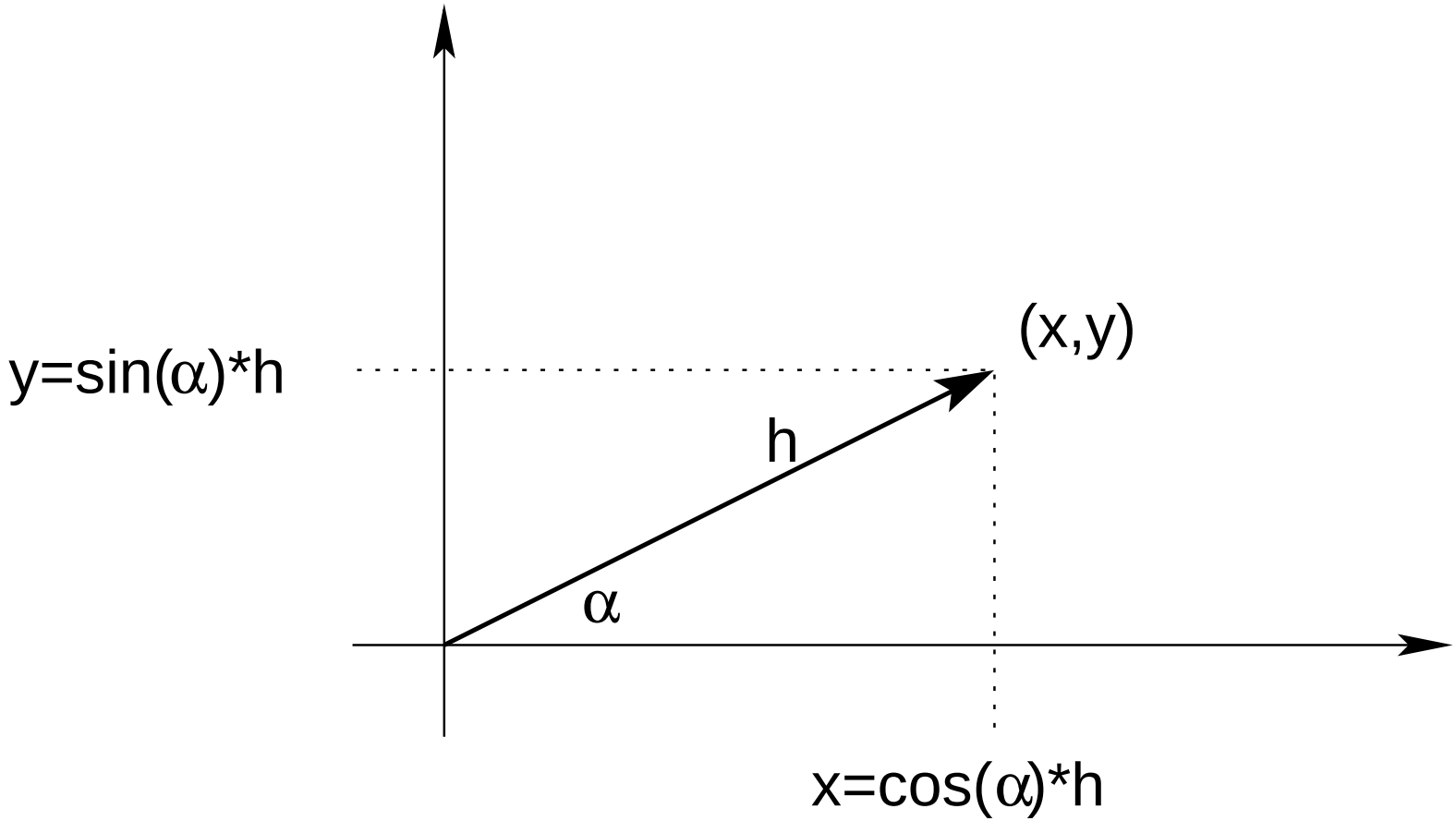
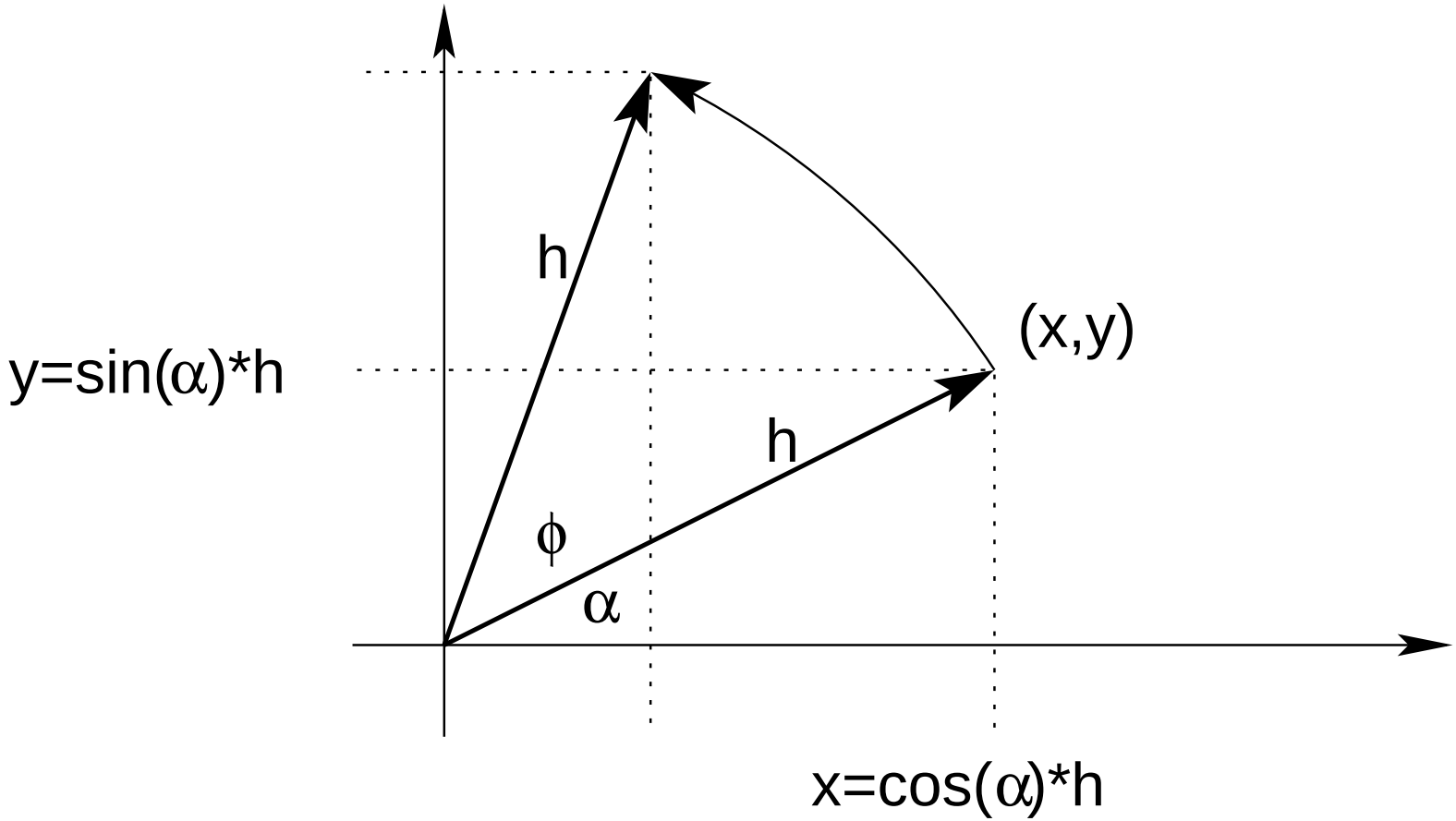
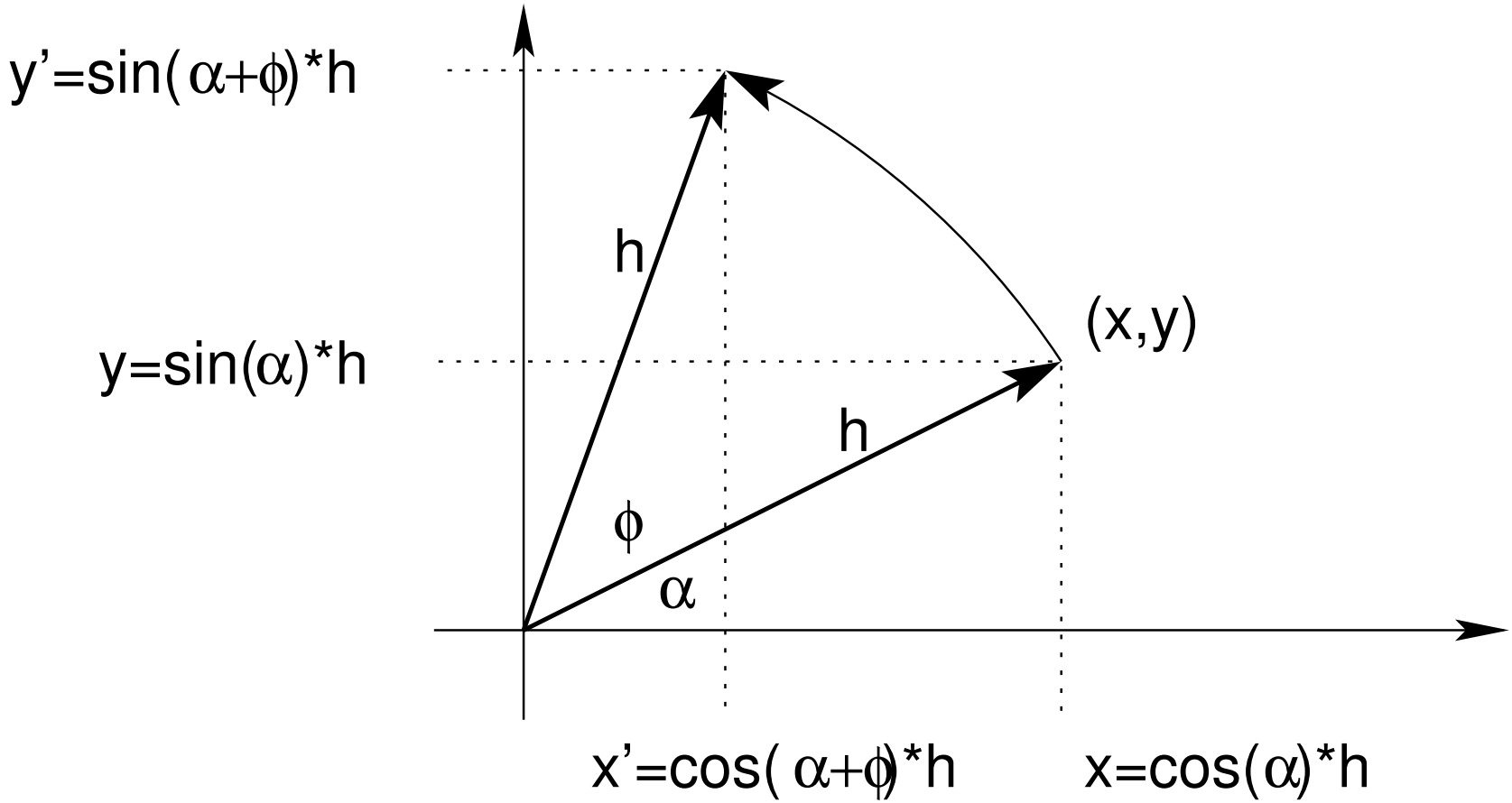
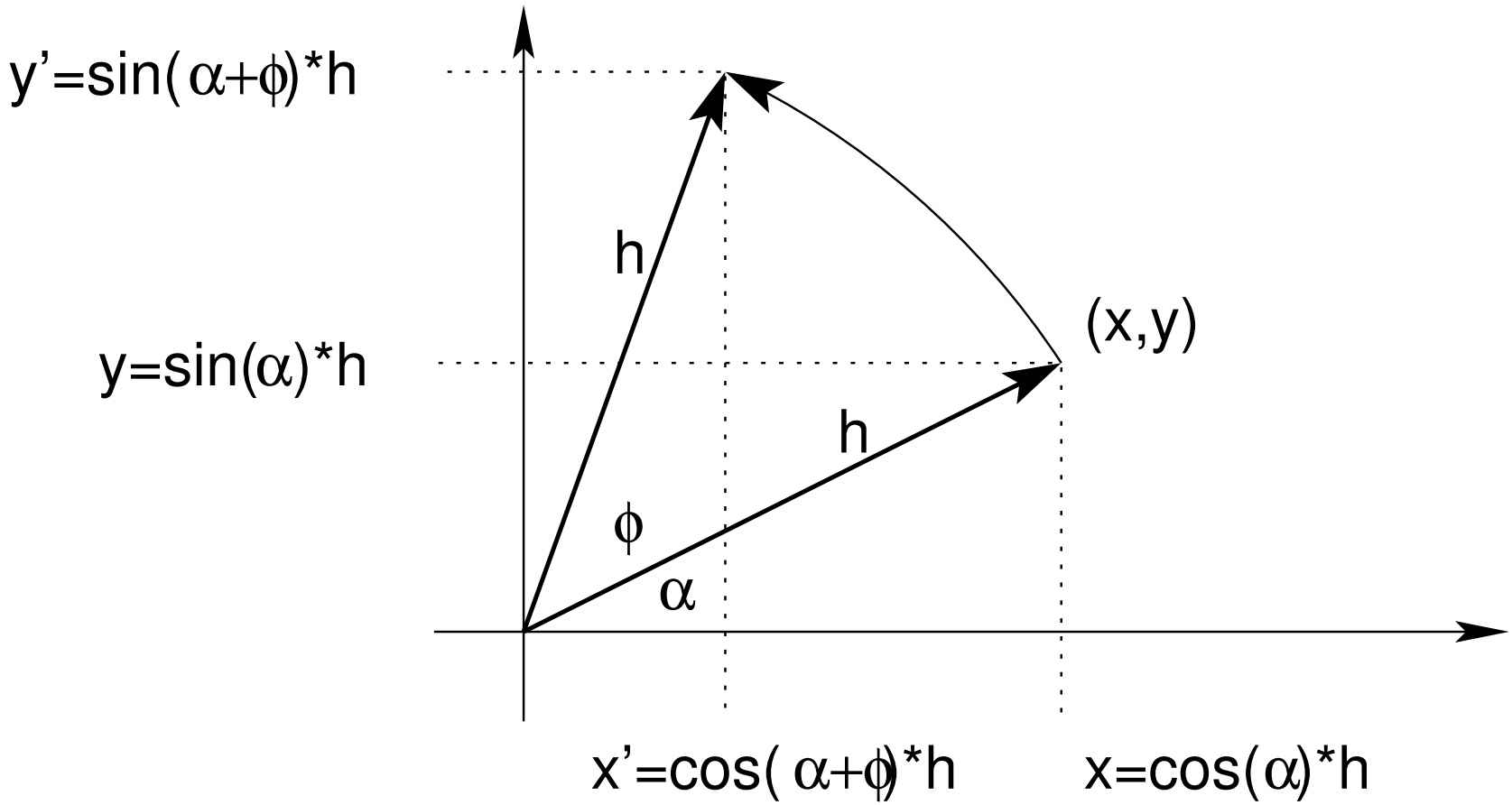
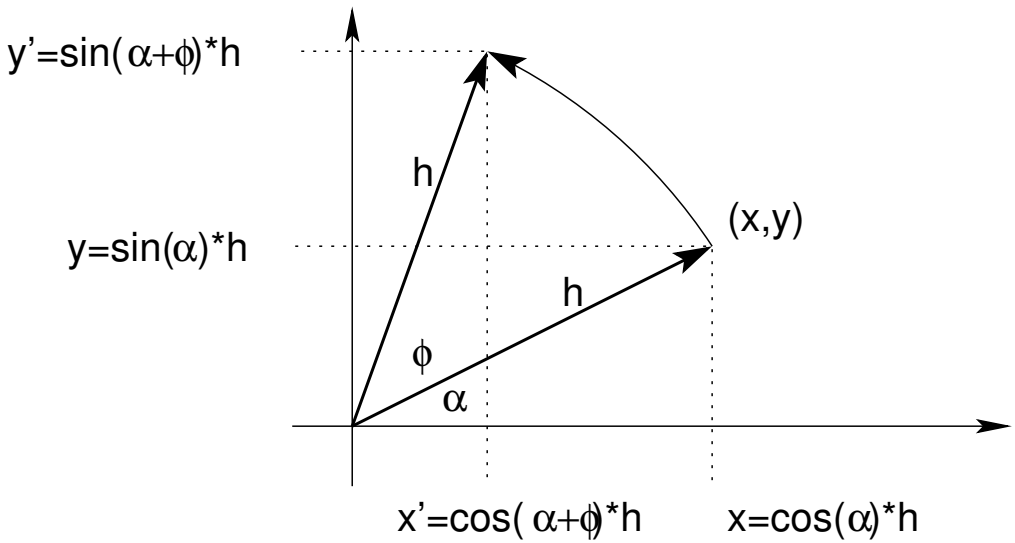
Deposit into account 6543
Amount:         41.52
New balance:    1517.37

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...

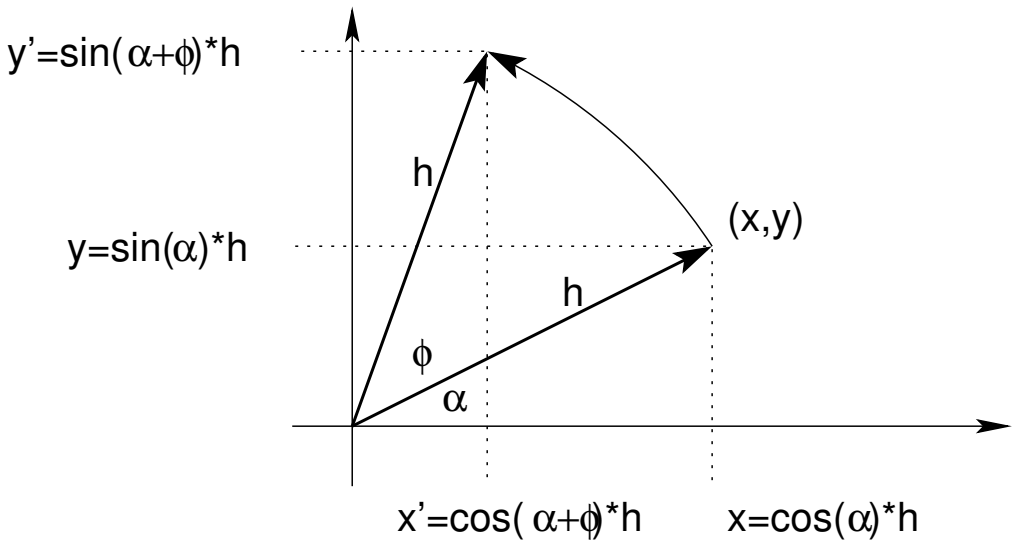
Using overdraft...
Withdrawal from account 4321
Amount:         50.25
New balance:    4400.69

New balance on account 9876: 0
```

Übersicht

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 erneut definieren.

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.

Idee

Statt:

... besser:



Ermittlung der aufgerufenen Methode

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

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

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

Die Klasse `Object`

- 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;
}
```

...

Die Klasse `Object` II

- `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");
   }
}
```

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!`

Bemerkungen

- 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.

Zur Erinnerung:

- 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 List(Object x) {
      info = x; next = null;
   }
   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 ...

Compilerfehler

... einen Compiler-Fehler, da der Variablen `x` nur Objekte (ggf. 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 ???

Generische Klassen

- 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;
   }
   ...
```

Verwendung

```
    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
```

Verwendung

```
    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>` !!!

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.

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.

Gibt es also Basistypen als Typparameter?

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.

Gibt es also Basistypen als Typparameter?

Ausweg:

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

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

Beispiel

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

Beispiel

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



Beispiel

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



Unwrapping

Eingewickelte Werte können auch wieder ausgewickelt werden.
Seit Java 1.5 erfolgt bei einer Zuweisung `int y = x;`
eine automatische Konvertierung:



Unwrapping

Eingewickelte Werte können auch wieder ausgewickelt werden.
Seit Java 1.5 erfolgt bei einer Zuweisung `int y = x;`
eine automatische Konvertierung:



Wrapping

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

Wrapping

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



Wrapping

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



Nutzen

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



Eigenschaften

- 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.

Integer vs. int



`Integer []`



`int []`

\+ `Integer`s können in polymorphen Datenstrukturen hausen.

– Sie benötigen mehr als doppelt so viel Platz.

– Sie führen zu vielen kleinen (evt.) über den gesamten Speicher verteilten Objekten $\Longrightarrow$ schlechteres Cache-Verhalten.

Übersicht

Abstrakte Methoden und Klassen

- Eine abstrakte Objekt-Methode ist eine Methode, für die keine Implementierung bereitgestellt 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.

Abstrakte Methoden und Klassen

Beispiel: Auswertung von Ausdrücken

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

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

Abstrakte Methoden und Klassen

- 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 (später!)
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

Im Beispiel ...

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

Beispiel für einen Ausdruck:

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

final

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

Andere Ausdrücke

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

```
public final class Neg extends Expression {
   private Expression arg;
   public Neg(Expression a) { arg = a; }
   protected int evaluate() { return -arg.getValue(); }
}    // end of class Neg
```

main() in Klasse `TestExp`:

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

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

Hierarchie

Die abstrakte Klasse *Expression*:



Hierarchie

Die abstrakte Klasse *Expression*:



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

Beispiel



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

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

Mehrere Oberklassen

- 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 ?



Interfaces

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

Interfaces

- 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 (kommt später!).
- 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 = (long)zaehler * (long)fraction.nenner;
     long right = (long)nenner * (long)fraction.zaehler;
     if (left == right) return 0;
     else if (left < right) return -1;
     else return 1;
  } // end of compareTo
    ...
}    // end of class Rational
```

Erläuterungen

- `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 untereinander

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

Beispiel (Forts.):

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

Beispiel

- 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
Cloneable
Comparable
Countable
Rational
AddSubMulDiv
Complex
Klassen–Hierarchie

Generics Revisited

- Für einen Typ-Parameter *T* kann man auch eine Oberklasse oder ein Interface angeben, das *T* auf jeden Fall implementieren 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 ();
   }
}
```

Generics Revisited II

- 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; }
}
```

Übersicht

Suchmaschine



Objekte identifizieren



Klassenentwurf

Komplexe Vektorähnlichkeit



Objekterzeugung



Sortieren nach Vektorähnlichkeit



Objektmethoden



Vektorähnlichkeit

Worthäufigkeiten

Dokumentensammlung

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

Divide and Conquer

Teile und Herrsche ist ein nützliches Grundprinzip der Algorithmik:

- Um ein Problem zu lösen, teile es in mehrere Teilprobleme
- Löse die einzelnen Teilprobleme
- Vereine die Teillösungen zu einer Gesamtlösung

Rekursion

Prinzip der Rekursion

> *Man führe das gleiche Berechnungsmuster immer wieder mit einfacheren bzw. kleineren Eingabedaten aus, bis man zu einer trivialen Eingabe gelangt.*

Realisierung:
Methoden, die sich *direkt* oder *indirekt* selbst aufrufen

Rekursion ist die Standard-Implementierung von Divide and Conquer

Rekursive Methoden

Definition

- Eine Methode `f` heißt (direkt) **rekursiv**, wenn im Rumpf von `f` Aufrufe von `f` vorkommen.
- Die Methode `f` heißt **indirekt rekursiv**, wenn im Rumpf von `f` eine Methode `g` aufgerufen wird, die ihrerseits direkt oder indirekt zu Aufrufen von `f` führt.

Bei jedem rekursiven Aufruf wird eine **neue Instanz** der jeweiligen Methode gestartet.

$\Longrightarrow$ Jede Instanz hat ihre eigenen lokalen Variablen und Parameter, welche „von außen“ nicht sichtbar sind.

Beispiel – Fakultätsfunktion (I)

Die Fakultätsfunktion $n!$ berechnet das Produkt der Zahlen $1, 2, ..., n$

Rekursiv lässt sich $n!$ daher so berechnen

$$0! = 1, \qquad n! = n \cdot (n-1)! \quad \text{für } n \geq 1$$

```
public static int fac(int n) {
    if (n > 0)
      return n * fac(n - 1);
    else
      return 1;
}
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
 = 4*3*fac(2)                                      // Auswerten
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
 = 4*3*fac(2)                                      // Auswerten
 = 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
 = 4*3*fac(2)                                      // Auswerten
 = 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
 = 4*3*2*fac(1)                                    // Auswerten
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
 = 4*3*fac(2)                                      // Auswerten
 = 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
 = 4*3*2*fac(1)                                    // Auswerten
 = 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } )    // Einsetzen
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
 = 4*3*fac(2)                                      // Auswerten
 = 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
 = 4*3*2*fac(1)                                    // Auswerten
 = 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } )    // Einsetzen
 = 4*3*2*1*fac(0)                                  // Auswerten
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
 = 4*3*fac(2)                                      // Auswerten
 = 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
 = 4*3*2*fac(1)                                    // Auswerten
 = 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } )    // Einsetzen
 = 4*3*2*1*fac(0)                                  // Auswerten
 = 4*3*2*1*( if (0>0) { 0*fac(0-1) } else { 1 } )  // Einsetzen
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
 = 4*3*fac(2)                                      // Auswerten
 = 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
 = 4*3*2*fac(1)                                    // Auswerten
 = 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } )    // Einsetzen
 = 4*3*2*1*fac(0)                                  // Auswerten
 = 4*3*2*1*( if (0>0) { 0*fac(0-1) } else { 1 } )  // Einsetzen
 = 4*3*2*1*1                                       // Auswerten
```

Beispiel – Fakultätsfunktion (II)

Was passiert nun beim Aufruf von `fac`?

```
fac(4)
 = ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
 = 4*fac(3)                                        // Auswerten
 = 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
 = 4*3*fac(2)                                      // Auswerten
 = 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
 = 4*3*2*fac(1)                                    // Auswerten
 = 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } )    // Einsetzen
 = 4*3*2*1*fac(0)                                  // Auswerten
 = 4*3*2*1*( if (0>0) { 0*fac(0-1) } else { 1 } )  // Einsetzen
 = 4*3*2*1*1                                       // Auswerten
 = 24
```

Ausblick: Rekursive Herausforderungen

- Ein Kreter sagt: Alle Kreter lügen (Epimenides)
- Problematischer: Ich spreche jetzt nicht die Wahrheit (Russell)
- Der Hilbertsche Barbier rasiert alle Männer des Dorfes, die sich nicht selbst rasieren können.

$$\forall x. \quad \mathit{rasiert}(\mathit{barbier}, x) \Longleftrightarrow \neg \mathit{rasiert}(x, x)$$

Rasiert der Barbier sich selbst?

```
boolean rasiert(Mann x, Mann y) {
  if (barbier.equals(x)) {
    return !rasiert(y, y);
  } else {
    return x.equals(y) && !rasiert(barbier, x);
  }
}
```

Was passiert bei `rasiert(barbier, barbier)`?

Übersicht

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.

Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



Scheiben bewegen



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 $h - 1$.

Code

```
public static void move (int h, byte a, byte b) {
  // h Scheiben von Stapel a nach Stapel b
  if (h > 0) {
     byte c = free (a,b);    //c ist der freie Platz
     move (h-1,a,c);         //h-1 Scheiben von a nach c
     // dann die verbleibende Scheibe von a nach b
     System.out.print ("\tmove␣"+a+"␣to␣"+b+"\n");
     move (h-1,c,b); //und h-1 Scheiben von c nach b
  }
}
```

Code

```
public static void move (int h, byte a, byte b) {
  // h Scheiben von Stapel a nach Stapel b
  if (h > 0) {
     byte c = free (a,b);    //c ist der freie Platz
     move (h-1,a,c);         //h-1 Scheiben von a nach c
     // dann die verbleibende Scheibe von a nach b
     System.out.print ("\tmove␣"+a+"␣to␣"+b+"\n");
     move (h-1,c,b); //und h-1 Scheiben von c nach b
  }
}
```

Bleibt die Ermittlung des freien Platzes ...

Freier Stapel?

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

(lies: M(x,y) ist frei, wenn x und y belegt sind.)

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

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

Implementierung

Diese Tabellen implementieren wir elegant mit `switch`:

```
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;
   }
}
```

... oder noch eleganter

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

... oder noch eleganter

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

Damit erhalten wir insgesamt:

Türme von Hanoi

```
class Hanoi {
  public static byte free (byte a, byte b) {
     return (byte) (3-(a+b));
  }
  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);
       }
    }
  static void main (String[] args) {
    int h = Terminal.askInt("Höhe␣des␣Turms:␣");
    System.out.print("\n\n");
    move (h,(byte) 0,(byte) 2);
    System.out.print("\n\n");
    }     // end of main()
}     // end of class Hanoi
```

Übersicht

Die Kochsche Schneeflockenkurve

Die Koch-Kurve ist ein **Fraktal**.

Sie wird – beginnend mit einer geraden Linie – iterativ gebildet, indem aus jedem geraden Linienstück das mittlere Drittel entfernt und dafür ein gleichseitiges Dreieck aufgesetzt wird.



Flockenmathe 2

Bildet man aus den drei Seiten eines gleichseitigen Dreiecks je eine Koch-Kurve, entsteht die Kochsche Schneeflocke.

DEMO

Flockenbildung: Horizontale Grundlinie

Aus jedem geraden Linienstück wird das mittlere Drittel entfernt und dafür ein gleichseitiges Dreieck aufgesetzt.

A



B



C



x^2/36 + h^2 = x^2/9
h=x*sqrt(3)/6

D



Flockenbildung: Beliebige Grundlinie

Verfahren wird rekursiv angewendet, also ist

1. die Grundlinie ggf. nicht horizontal und
2. der Ausgangspunkt meistens nicht (0,0).



Lösung: Translation und Rotation

Um trotzdem die einfache Berechnung der Punktkoordinaten verwenden zu können,

1. verschieben wir die Strecke in den Ursprung durch Addition der negativen Koordinaten des ersten Punkts, also von $(-x_1, -y_1)$,
2. rotieren wir die verschobene Strecke um einen Winkel ϕ so, dass sie horizontal liegt,
3. nehmen wir die einfache Berechnung vor und
4. rotieren den neu erzeugten Teil der Flocke in umgekehrter Richtung um $-\phi$ und verschieben ihn zurück durch Addition von (x_1, y_1).



Flockenmathe: Rotation



Flockenmathe: Rotation



Flockenmathe: Rotation



Flockenmathe: Rotation



Flockenmathe: Rotation 4

Additionstheorem 1: $\sin(\alpha + \phi) = \sin(\alpha) \cdot \cos(\phi) + \sin(\phi) \cdot \cos(\alpha)$

Additionstheorem 2: $\cos(\alpha + \phi) = \cos(\alpha) \cdot \cos(\phi) - \sin(\phi) \cdot \sin(\alpha)$



Flockenmathe: Rotation 5

Additionstheorem 1: $\sin(\alpha + \phi) = \sin(\alpha) \cdot \cos(\phi) + \sin(\phi) \cdot \cos(\alpha)$

Additionstheorem 2: $\cos(\alpha + \phi) = \cos(\alpha) \cdot \cos(\phi) - \sin(\phi) \cdot \sin(\alpha)$



Einsetzen: $x' = x \cdot \cos(\phi) - y \cdot \sin(\phi)$ und

$y' = y \cdot \cos(\phi) + x \cdot \sin(\phi)$

Realisierung in Java

- Verwende die `Point`-Klasse
- Realisierung der Iteration durch Rekursion über die einzelnen Teilstrecken

  Basisfall: Zeichne die aktuelle Strecke

  Rekursionsfall: Berechne die Zwischenpunkte der aktuellen Strecke und iteriere über alle so entstandenen Teilstrecken
- Berechnung der Zwischenpunkte mit Hilfe der Operationen aus der Klasse `Point`

Elementare grafische Operationen in Java

Zur grafischen Ausgabe verwenden wir die Klasse `Pad`
(Pepper-Buch S. 73 ff.)

- `Pad p = new Pad()` erzeugt ein neues (noch nicht sichtbares) Fenster
- `p.setPadSize(width, height)` setzt die Größe entsprechend
- `p.setVisible(true)` macht das Fenster sichtbar
- `p.drawLine(p1, p2)` zeichnet eine Linie von Punkt `p1` nach `p2`
- Der Ursprung des Koordinatensystems befindet sich in der linken oberen Ecke
- Alle Koordinaten sind in Pixel

ACHTUNG:
Die Klasse `Pad` erfüllt *nicht* unsere Vorgaben
bzgl. Programmiermethodik!

Implementierung: Hauptklasse

```
import Terminal;
class Koch {
  public static void main(String[] args) {
    Point2 p1,p2;
    Flocke f;
    PointListElement pointer;
    Pad p=new Pad();
    p.setPadSize(1000, 450);
    p.setBackground(Pad.white);
    p.setTitle("Kochsche␣Schneeflockenkurve");
    p.setVisible(true);
    p.setColor(Pad.black);

    p1=new Point2(0.0,0.0);
    p2=new Point2(1000.0,0.0);
    f=new Flocke(p1,p2);
    // f.schneien(Integer.valueOf(args[0]));
    f.rekursivSchneien(f.first,f.last,Integer.valueOf(args[0]));
```

Implementierung: Hauptklasse 2

```
        // zeichnen!
        pointer=f.first;
        while(pointer!=null && pointer.getNext()!=null) {
          p.drawLine((int)pointer.getPoint().getX(),
                     400-(int)pointer.getPoint().getY(),
                     (int)pointer.getNext().getPoint().getX(),
                     400-(int)pointer.getNext().getPoint().getY());
          pointer=pointer.getNext();
      }
    }
  }
```

Implementierung: Punkte

```
class Point2 {
    private double x,y;
    Point2(double x, double y) { ... }
    void set ...
    double get ...

    // changes object on which method is executed AND returns this altered object
    Point2 rotate(double phi){
      set(x*Math.cos(phi)-y*Math.sin(phi), y*Math.cos(phi)+x*Math.sin(phi));
      return this;
    }

    // changes object on which method is executed AND returns this altered object
    Point2 translate(double x, double y) {
      set(this.x+x, this.y+y);
      return this;
    }
    Point2 rotateThenTranslate(double angle, double x, double y) {
      return this.rotate(angle).translate(x,y);
    }

}
```

Erinnerung: Rotation

Additionstheorem 1: $\sin(\alpha + \phi) = \sin(\alpha) \cdot \cos(\phi) + \sin(\phi) \cdot \cos(\alpha)$

Additionstheorem 2: $\cos(\alpha + \phi) = \cos(\alpha) \cdot \cos(\phi) - \sin(\phi) \cdot \sin(\alpha)$



Einsetzen: $x' = x \cdot \cos(\phi) - y \cdot \sin(\phi)$ und

$y' = y \cdot \cos(\phi) + x \cdot \sin(\phi)$

Implementierung: Punktliste

```
class PointListElement {
    private Point2 p;
    private PointListElement next;

    PointListElement(Point2 p) {
       setPoint(p);
       setNext(null);
    }
    void setPoint(Point2 p) {
       this.p=p;
    }
    void setNext(PointListElement p) {
       this.next=p;
    }

    Point2 getPoint() {
       return p;
    }

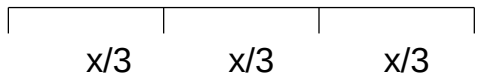
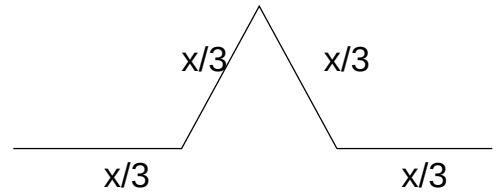
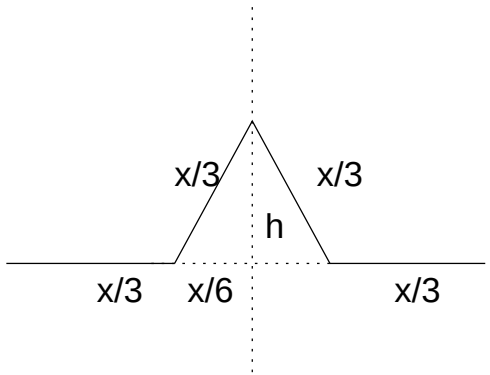
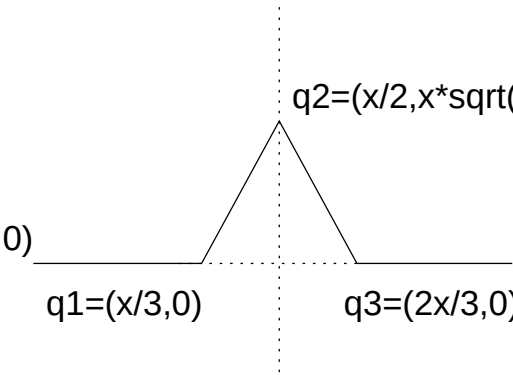
    PointListElement getNext() {
       return next;
    }
}
```

Implementierung: Flocke 1

```
class Flocke {
  PointListElement first,last;
  Flocke(Point2 p1, Point2 p2) {
    first=new PointListElement(p1); last=new PointListElement(p2);
    first.setNext(last);
  }

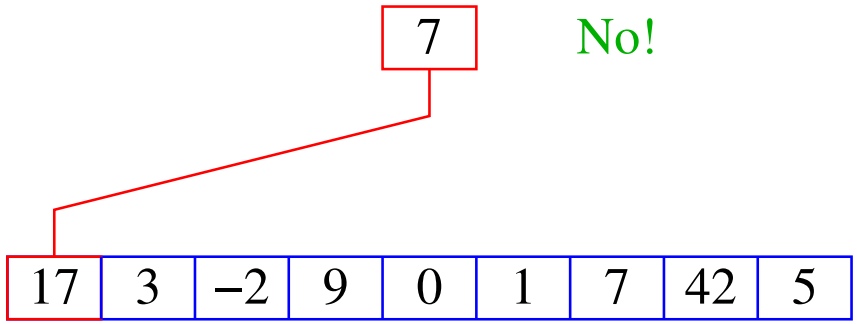
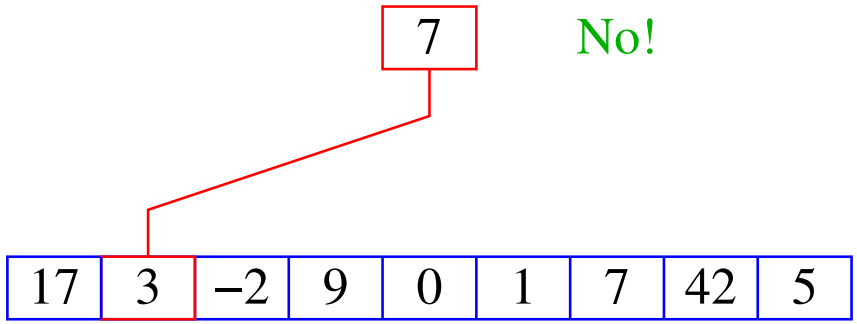
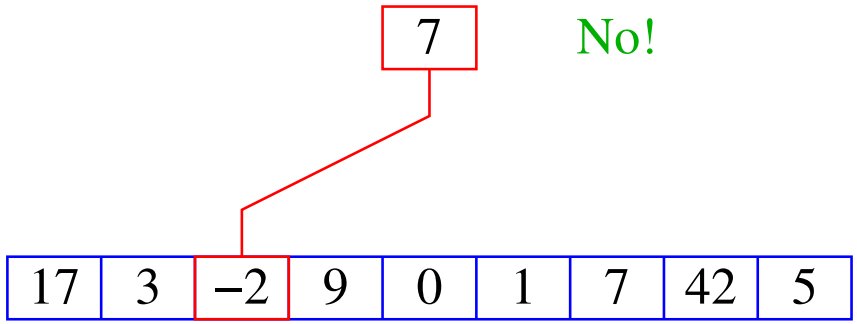
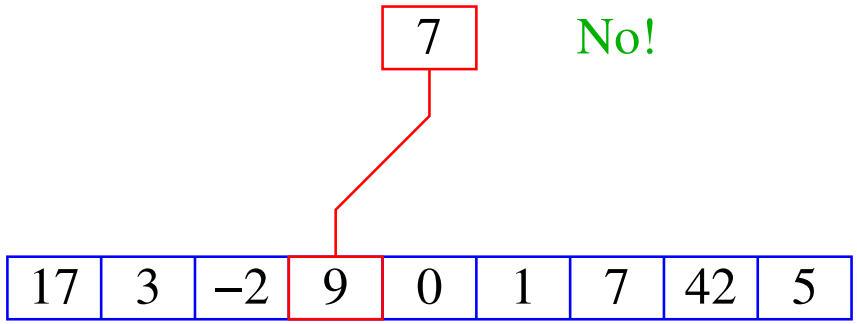
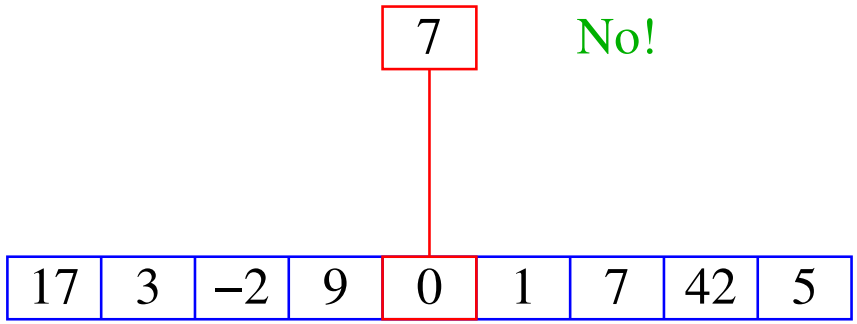
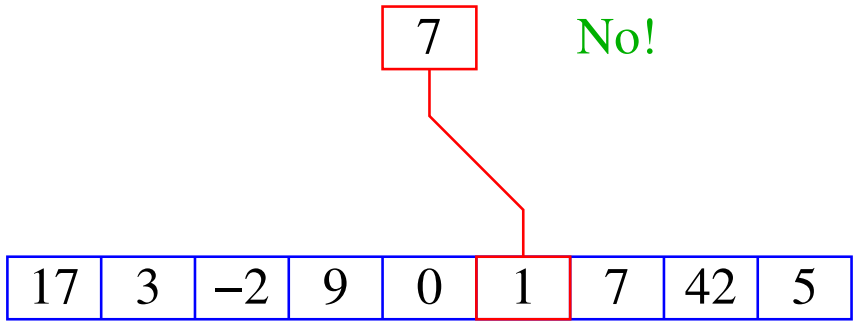
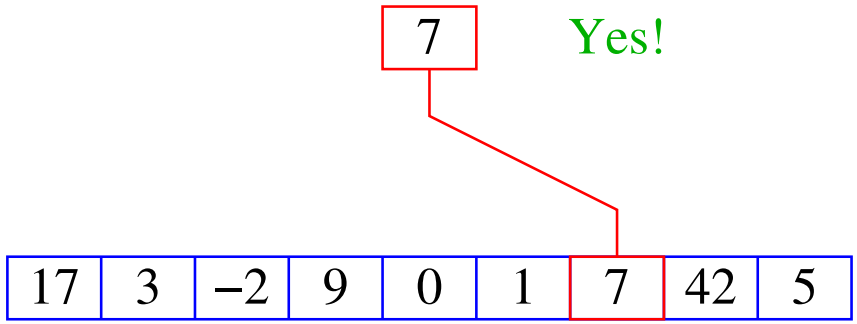
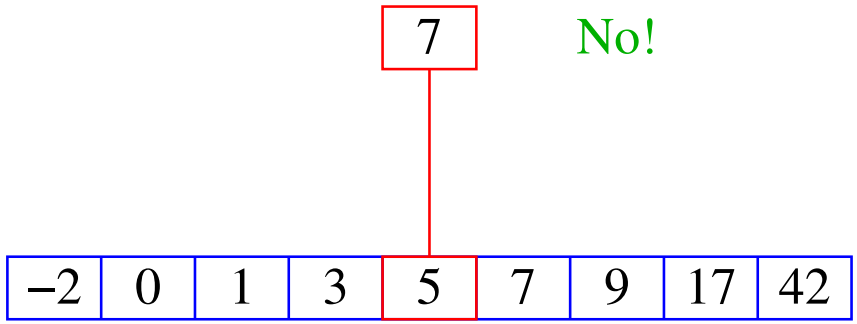
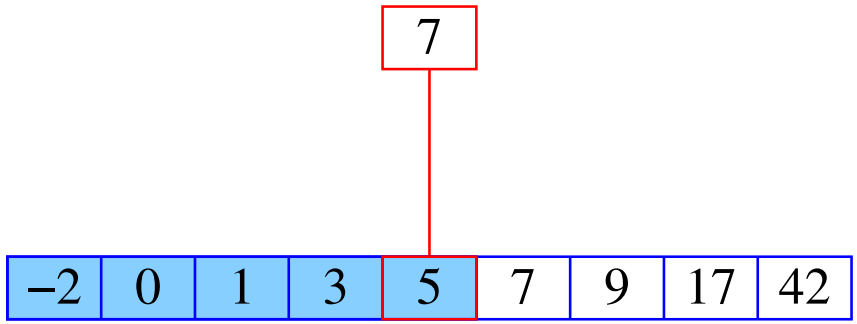
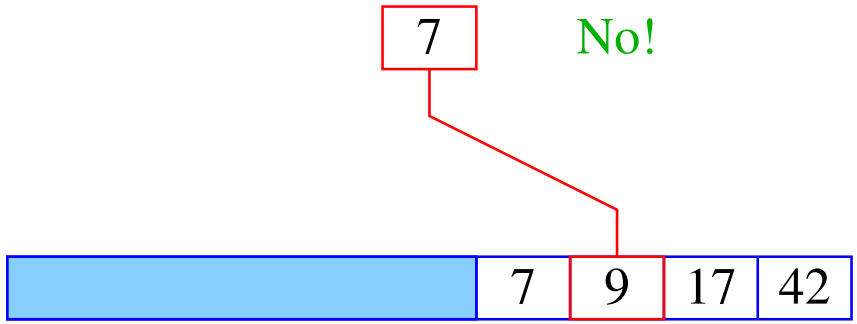
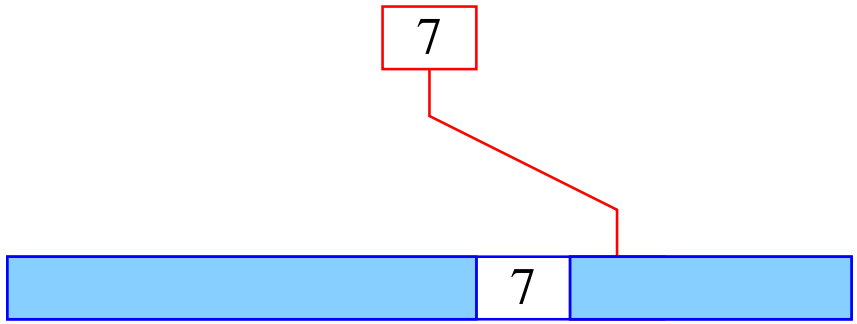
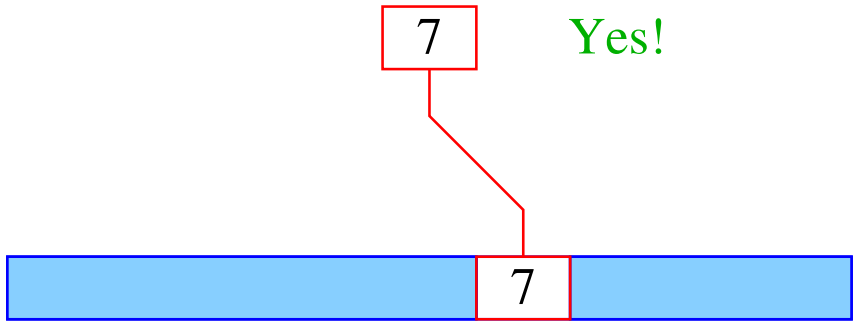
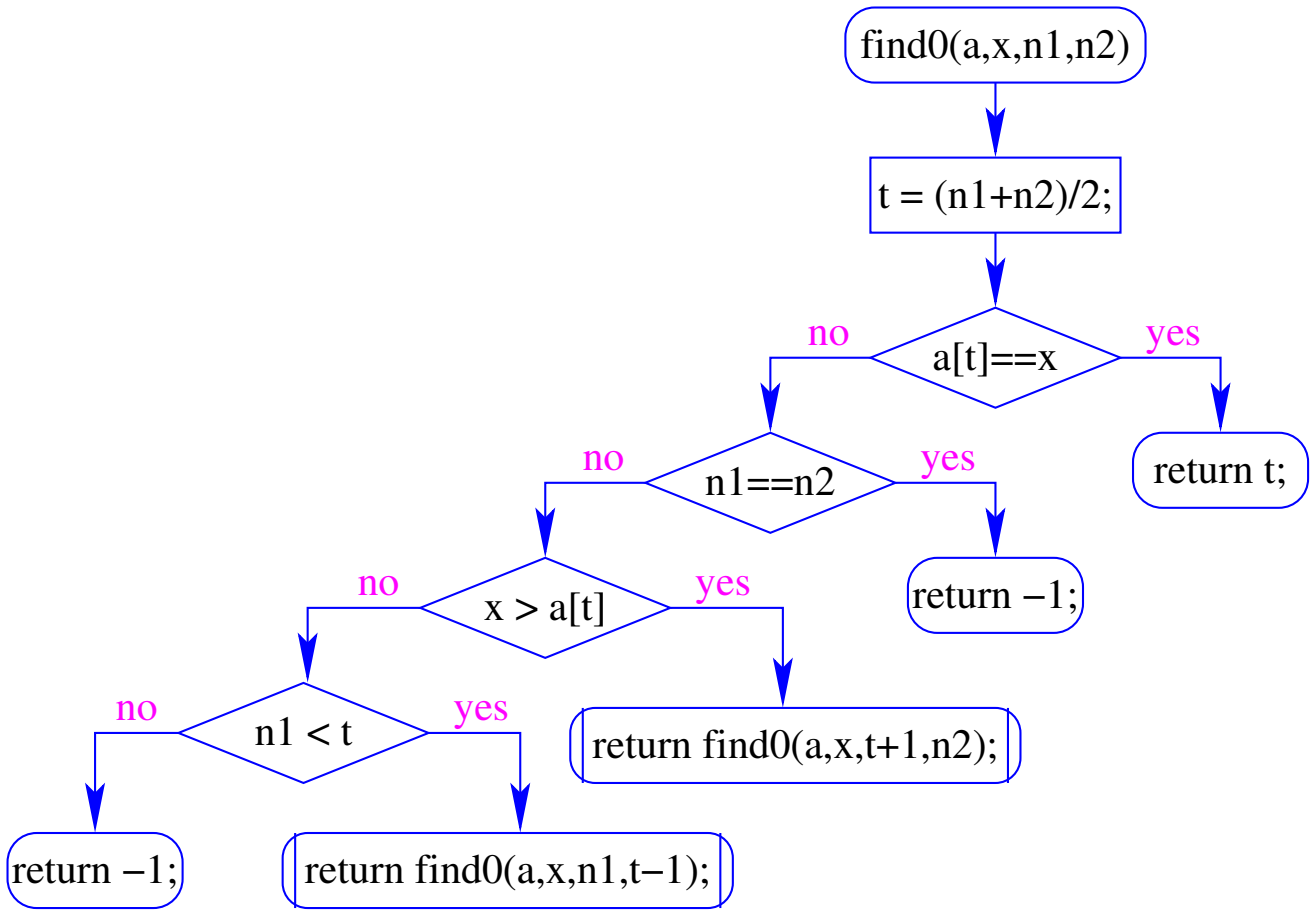
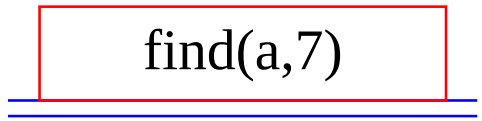
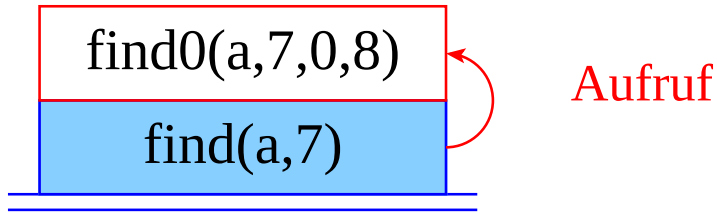
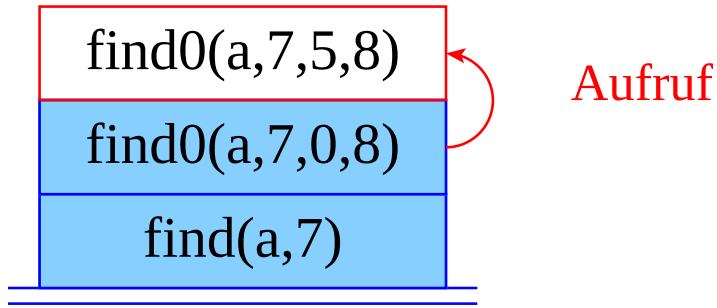
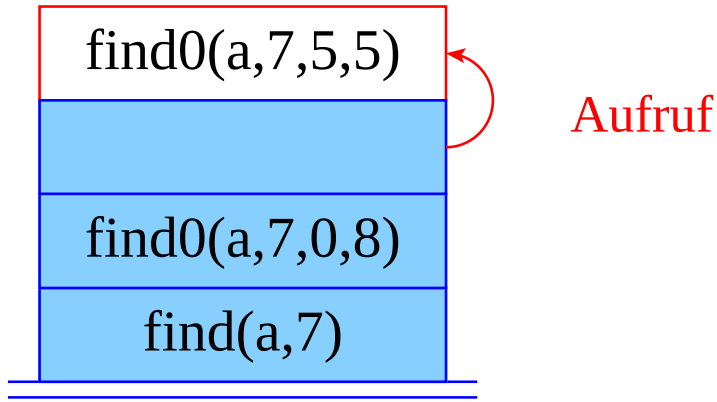
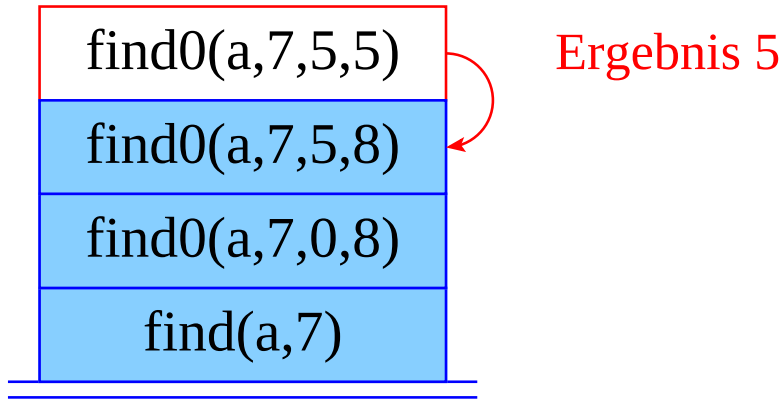
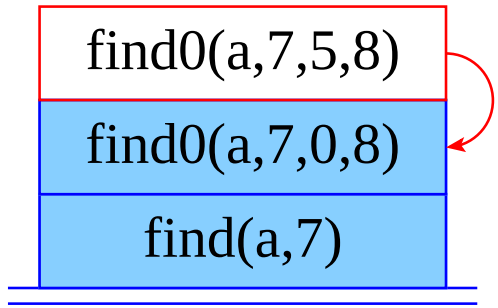
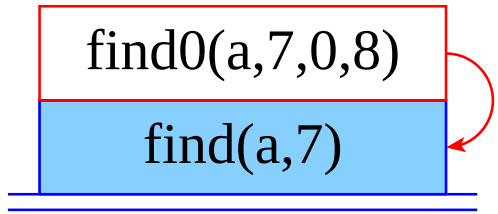
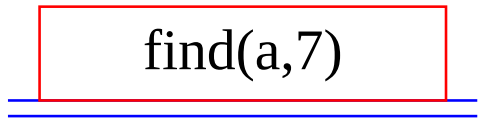
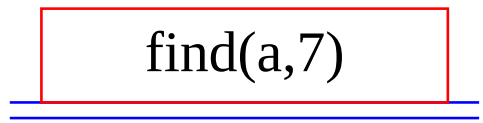
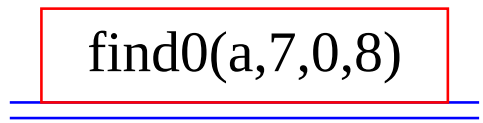
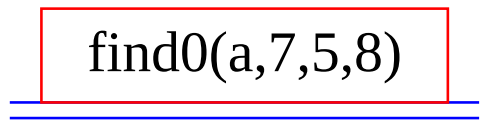
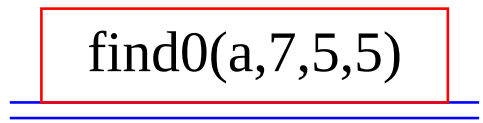
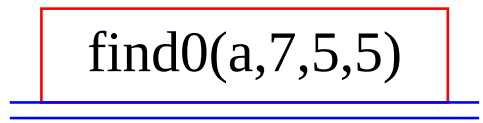
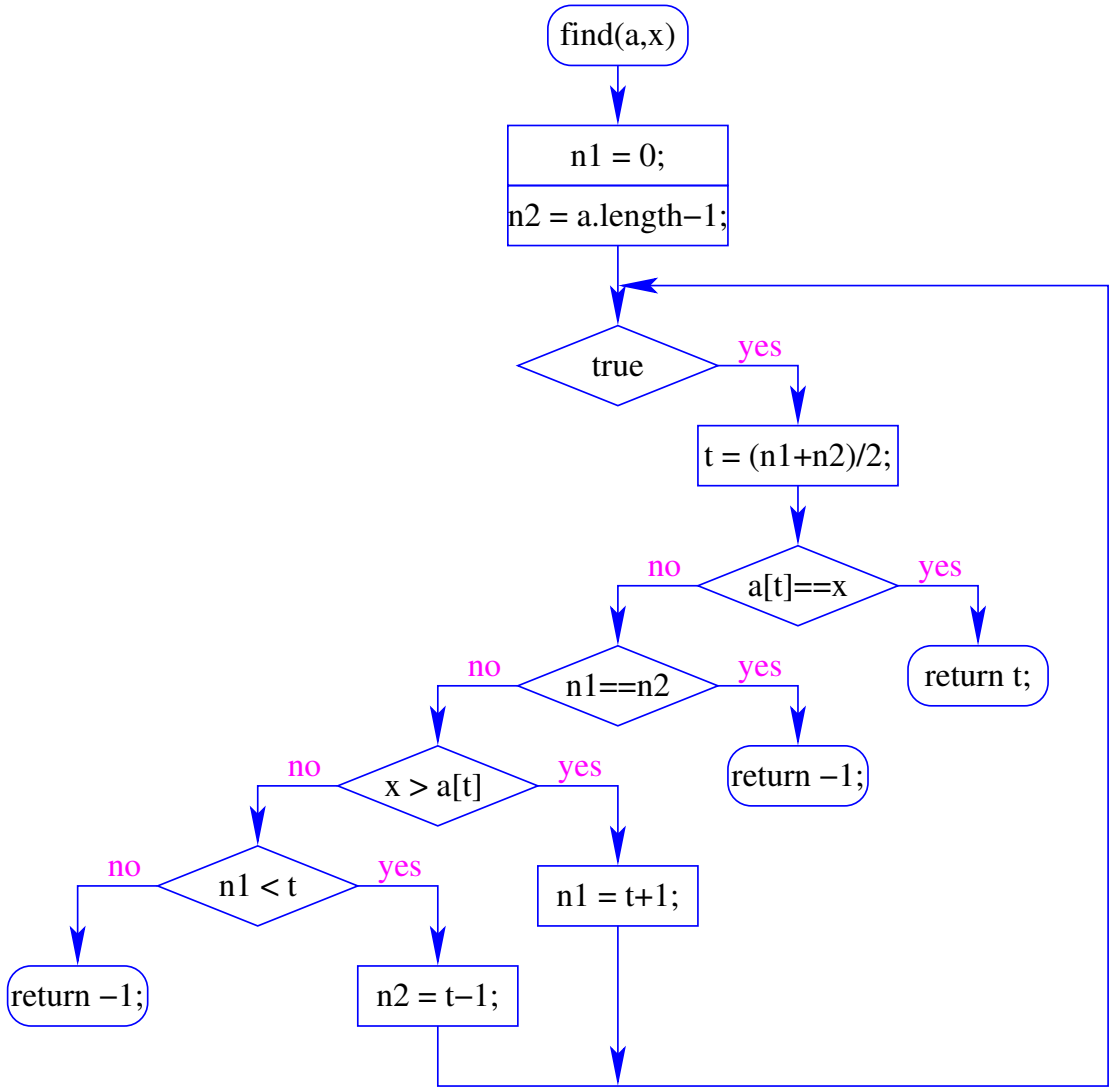
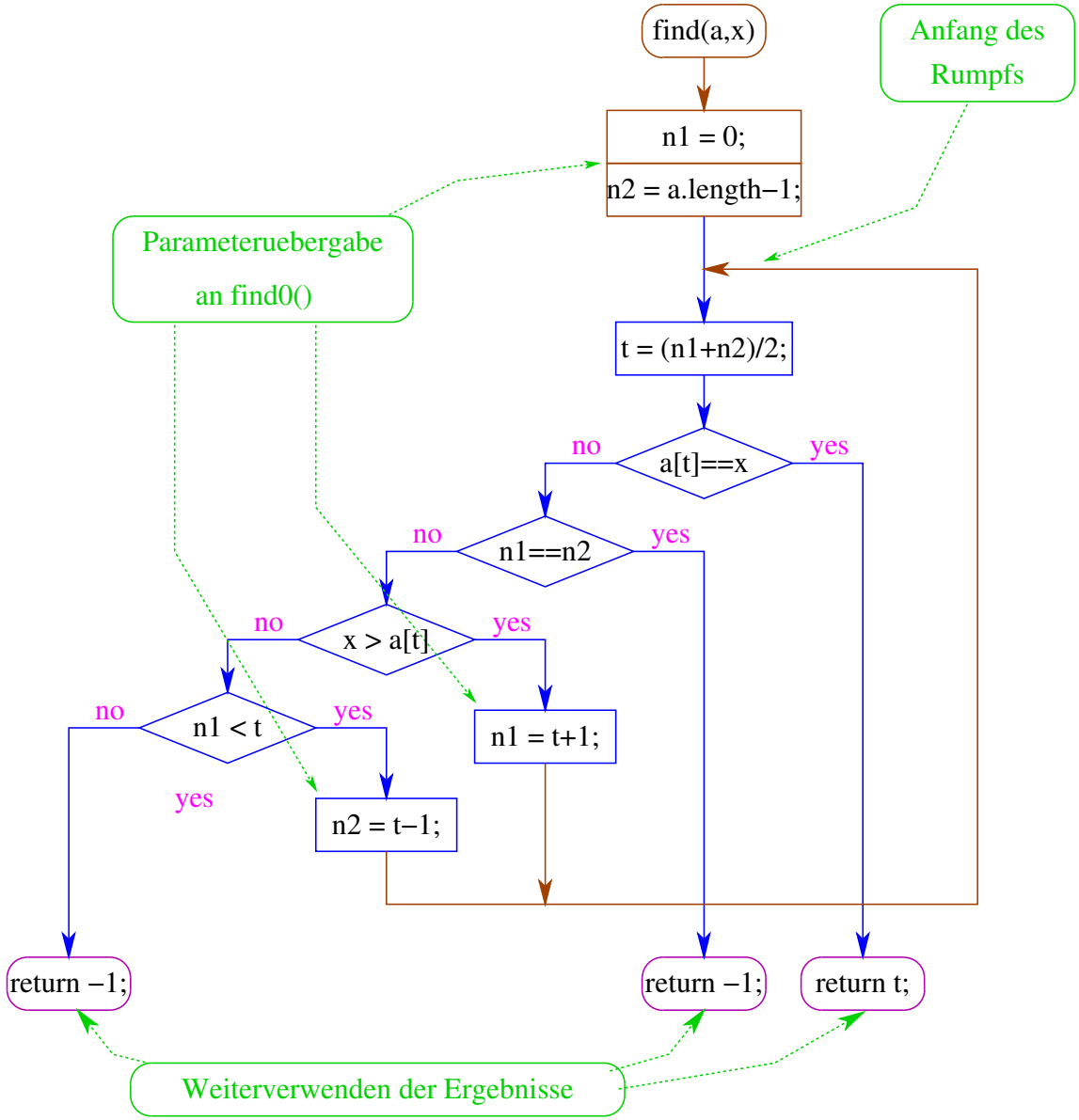
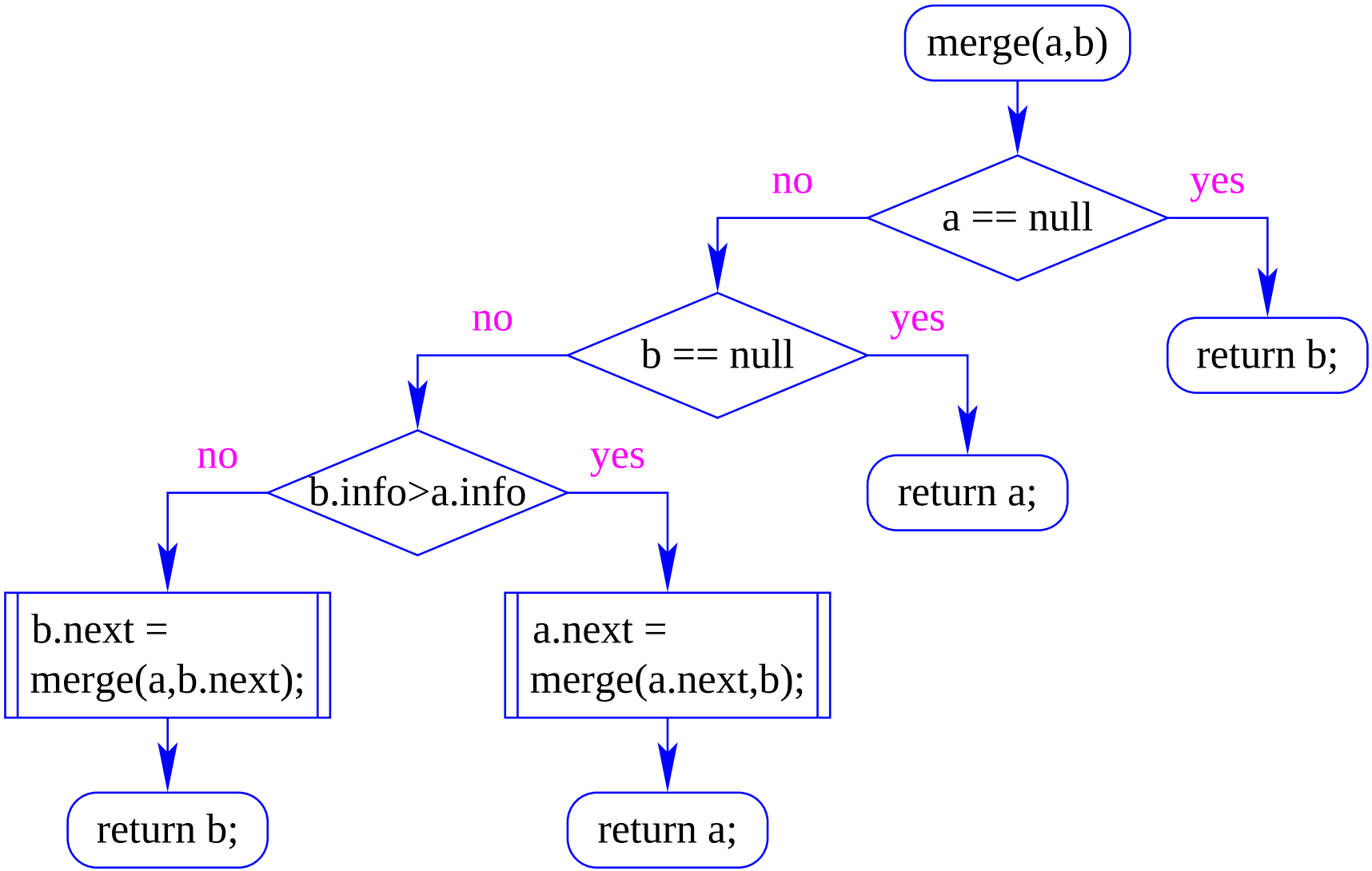
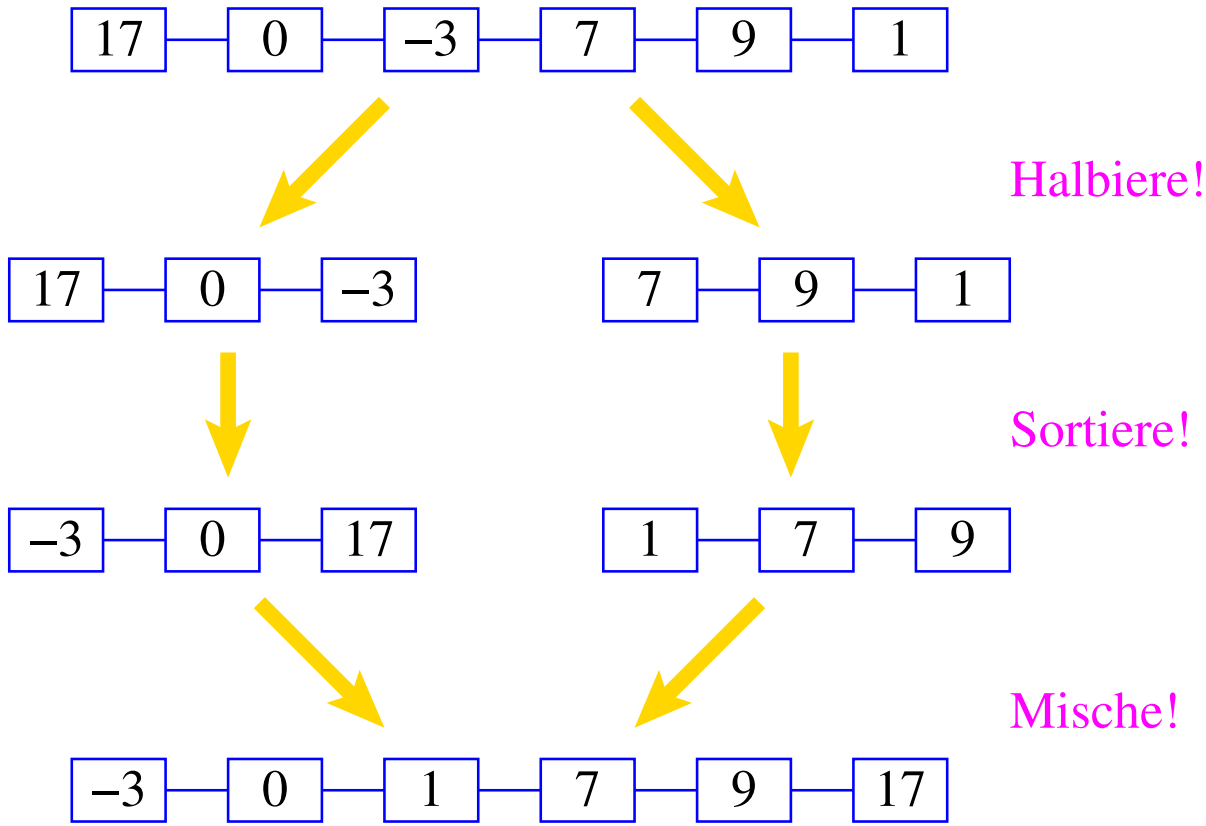
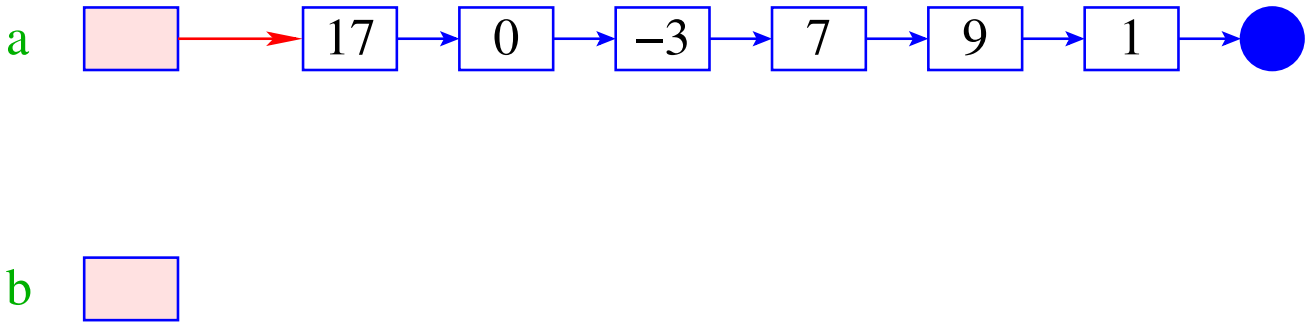
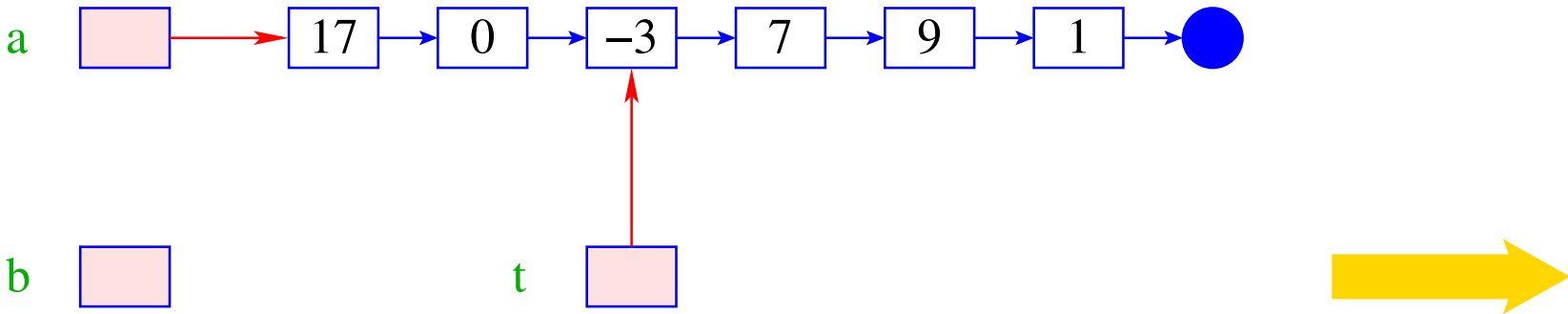
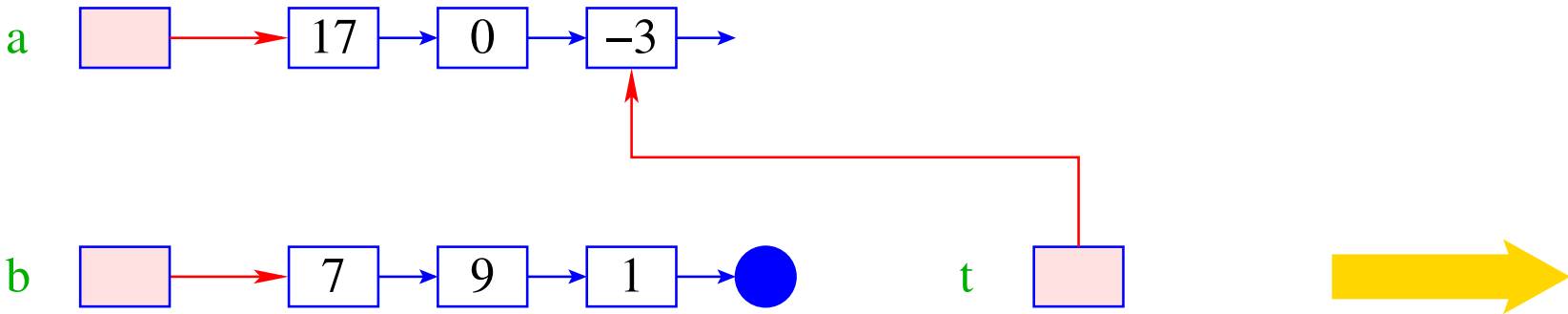
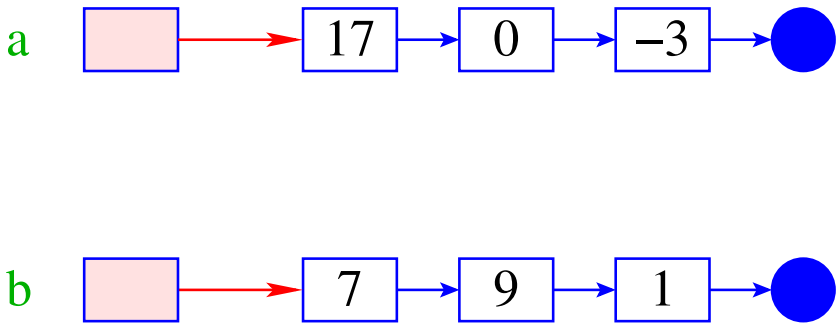
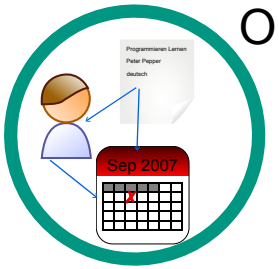
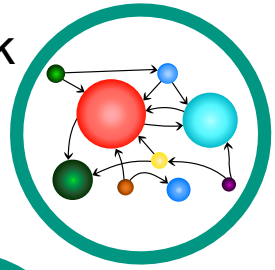
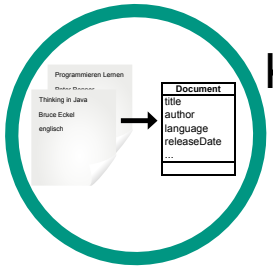
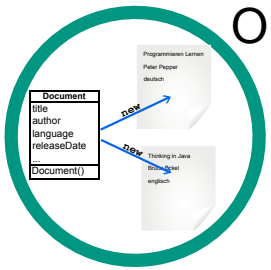
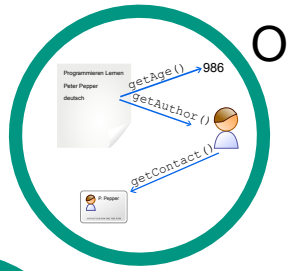
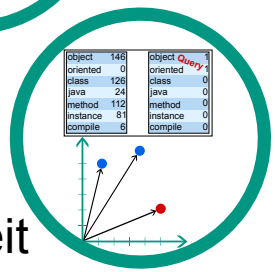
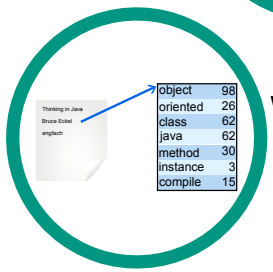
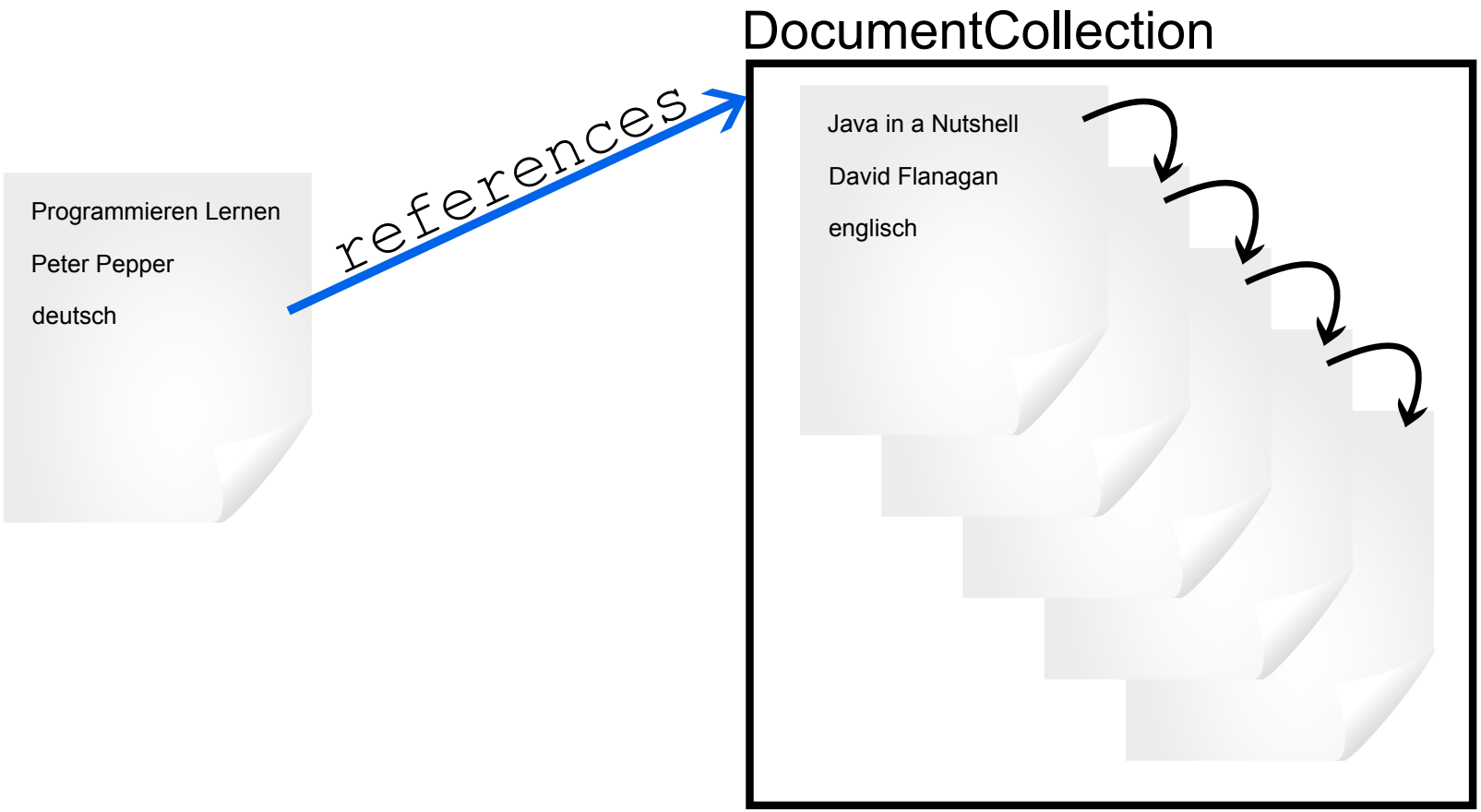
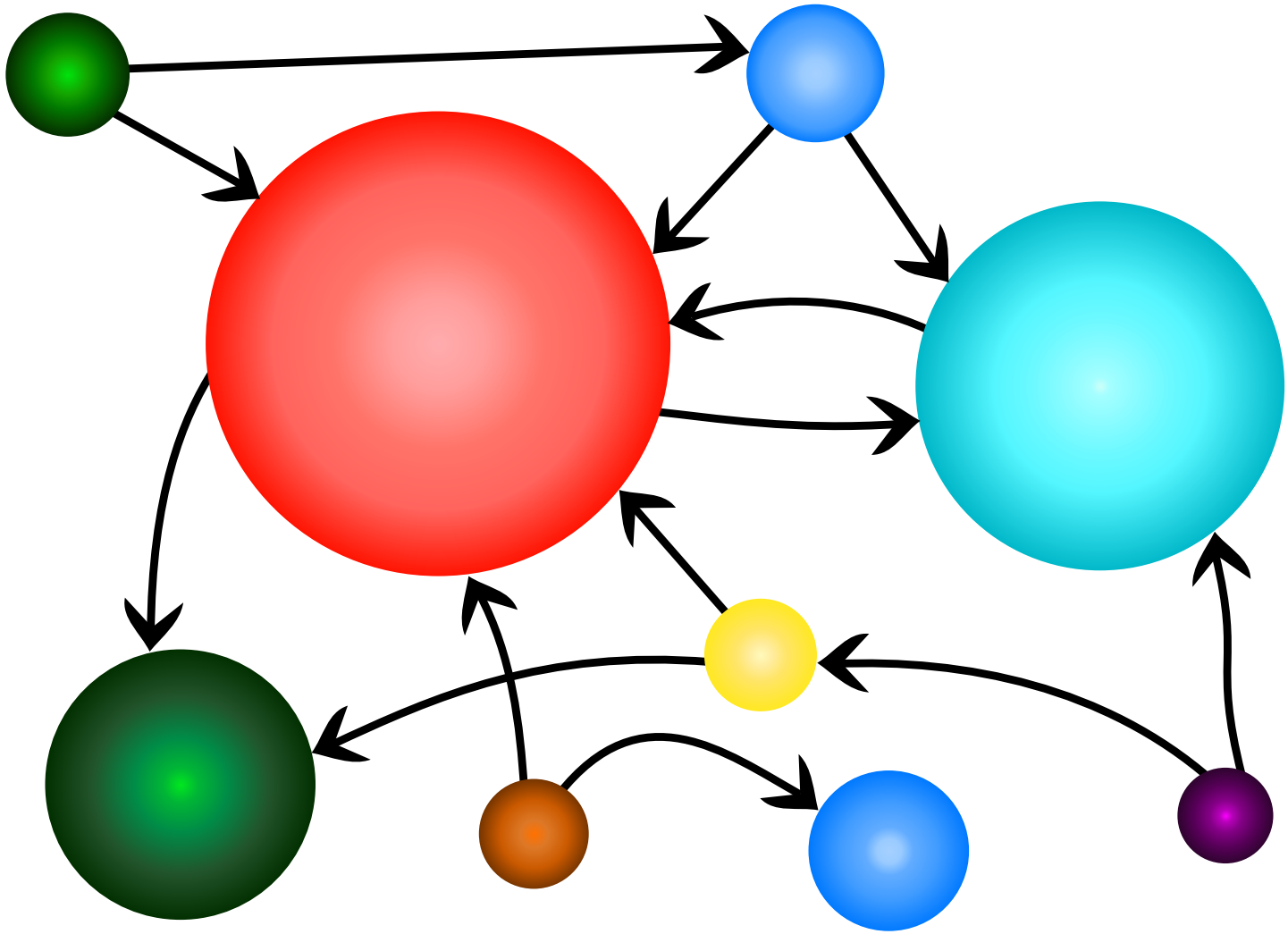
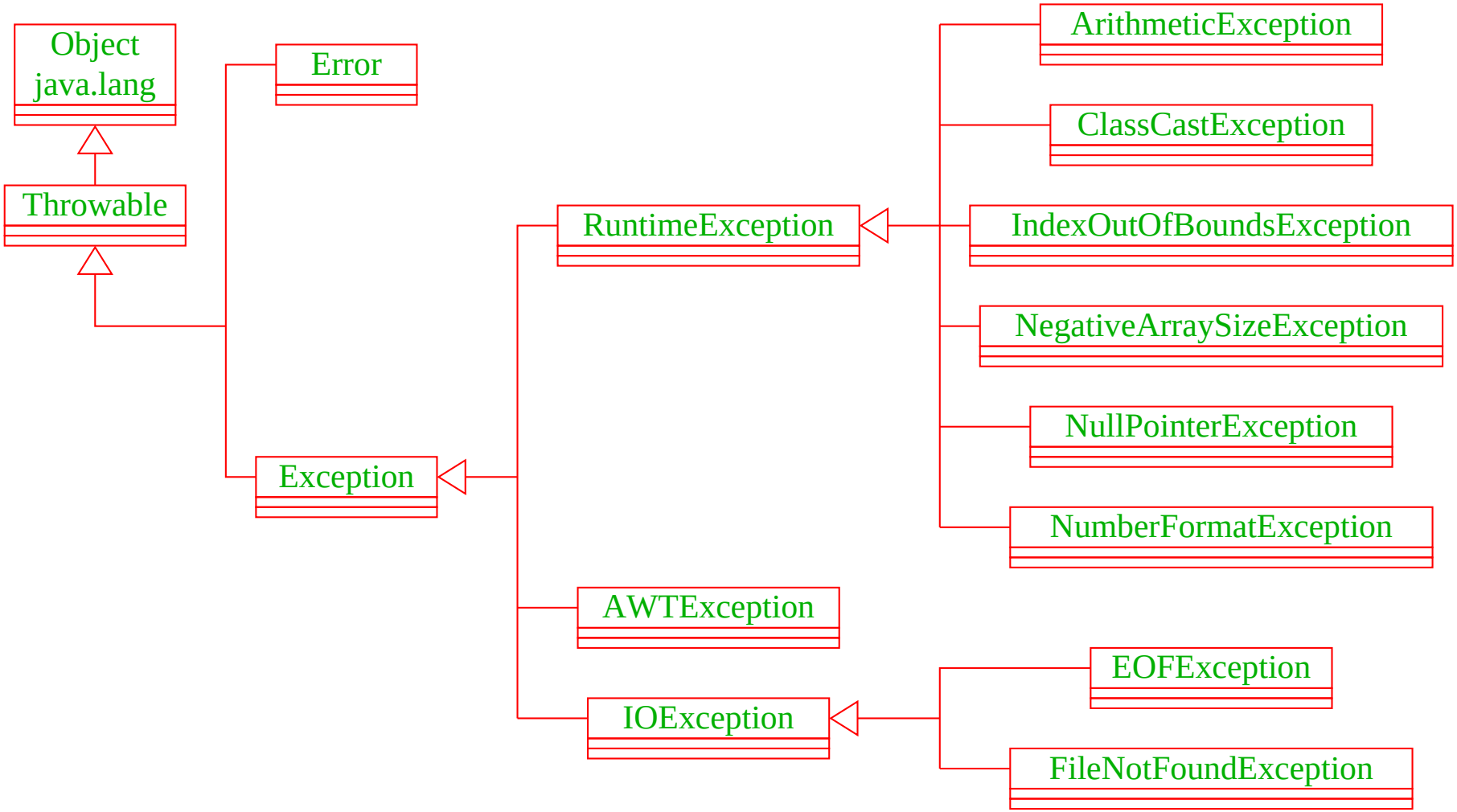
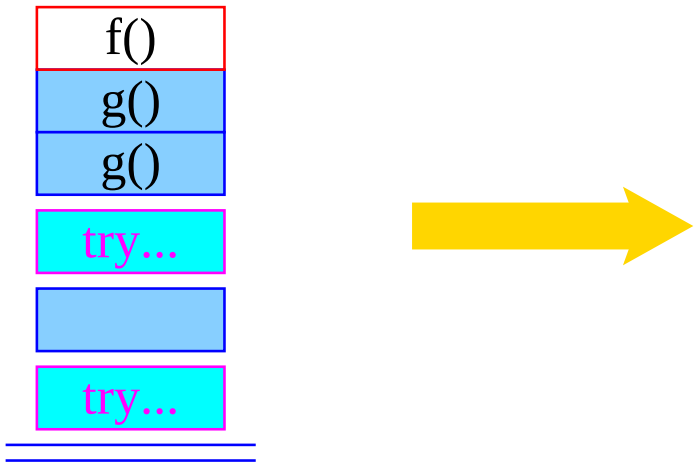
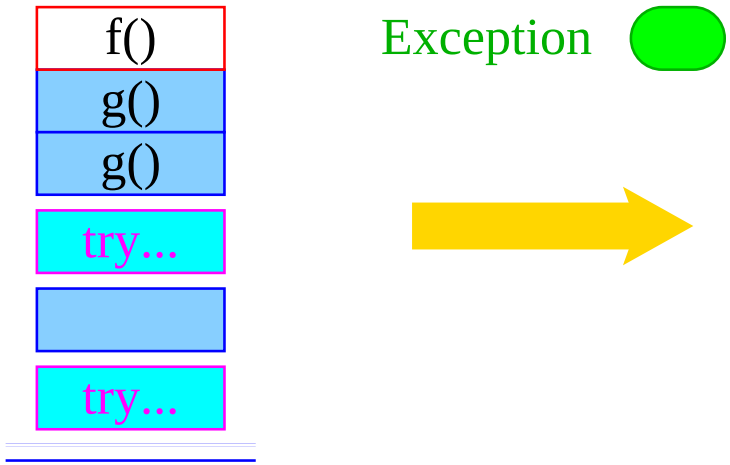
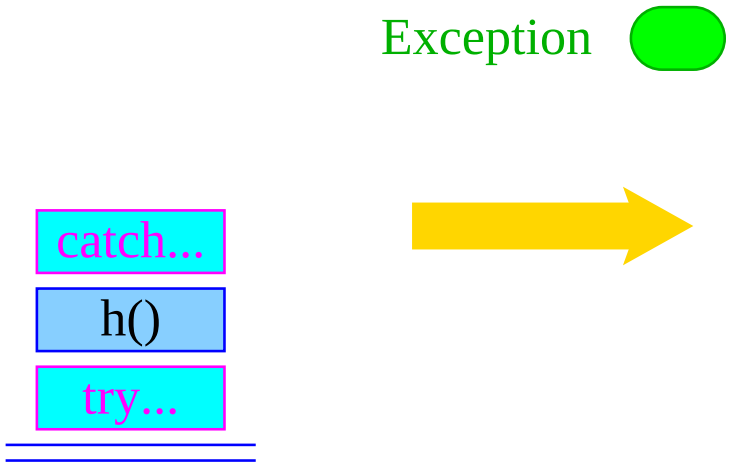
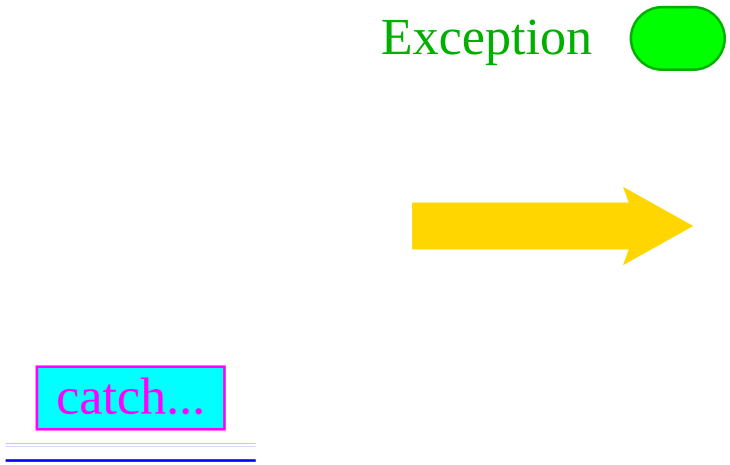
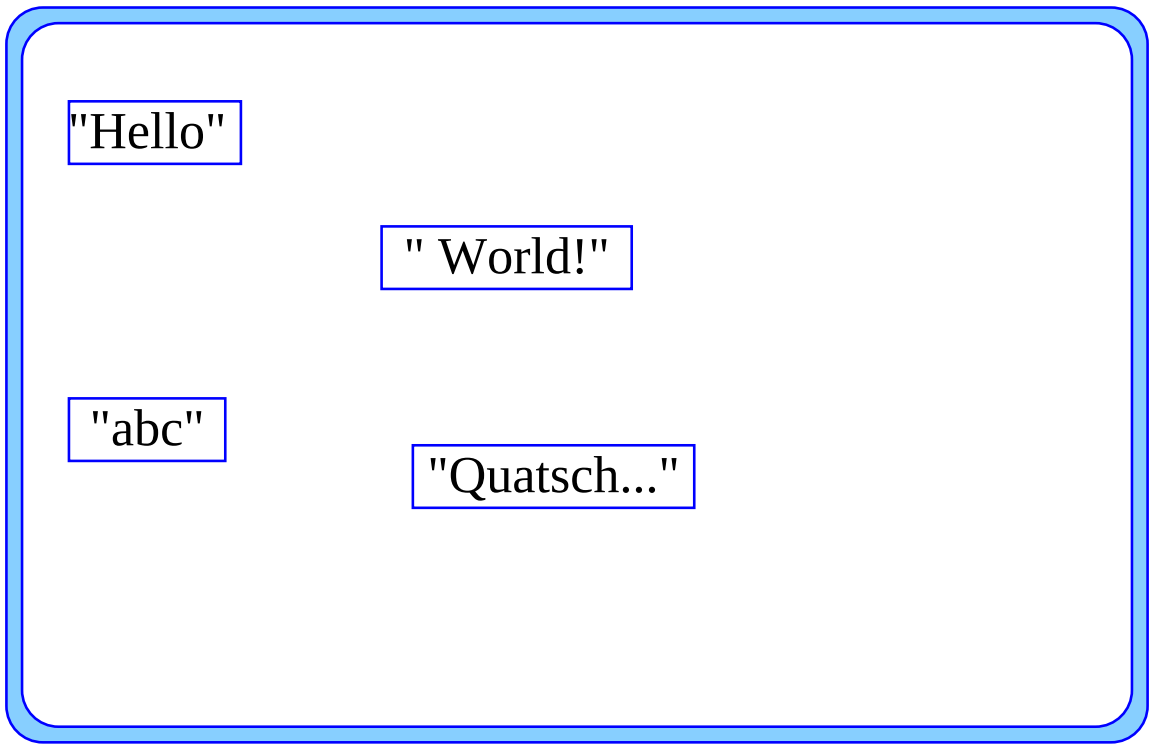
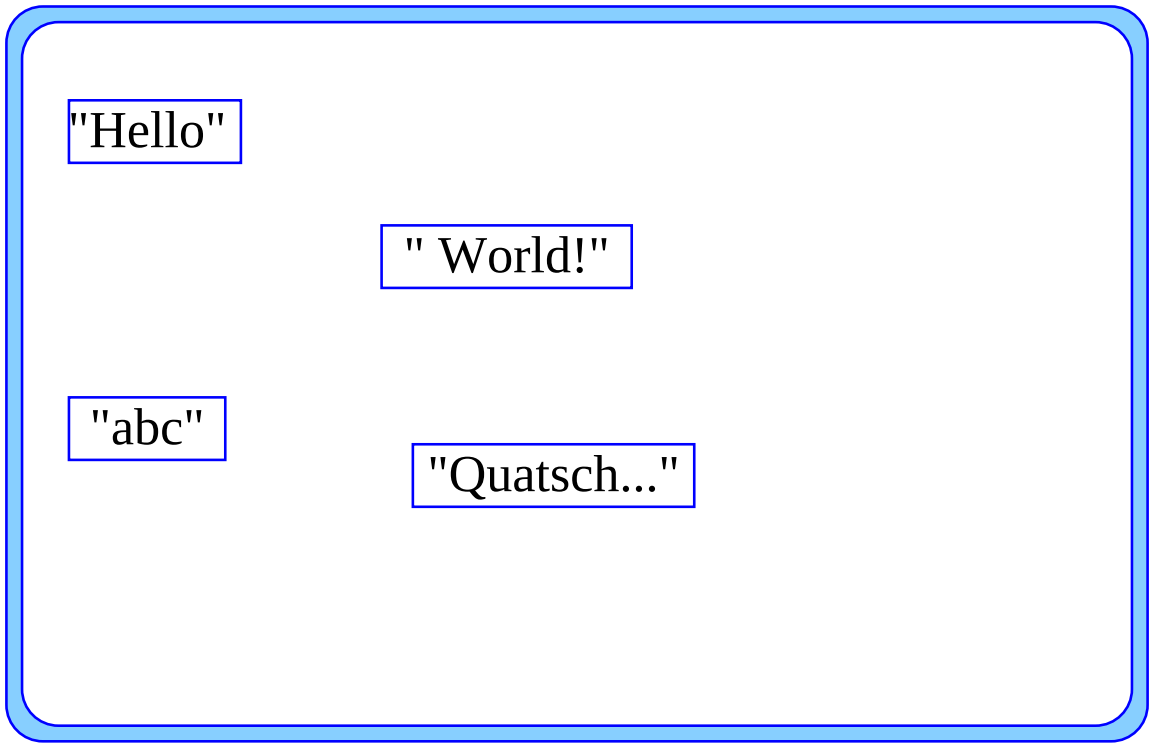
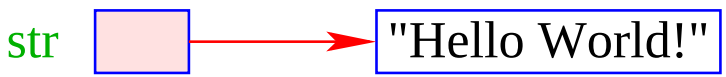
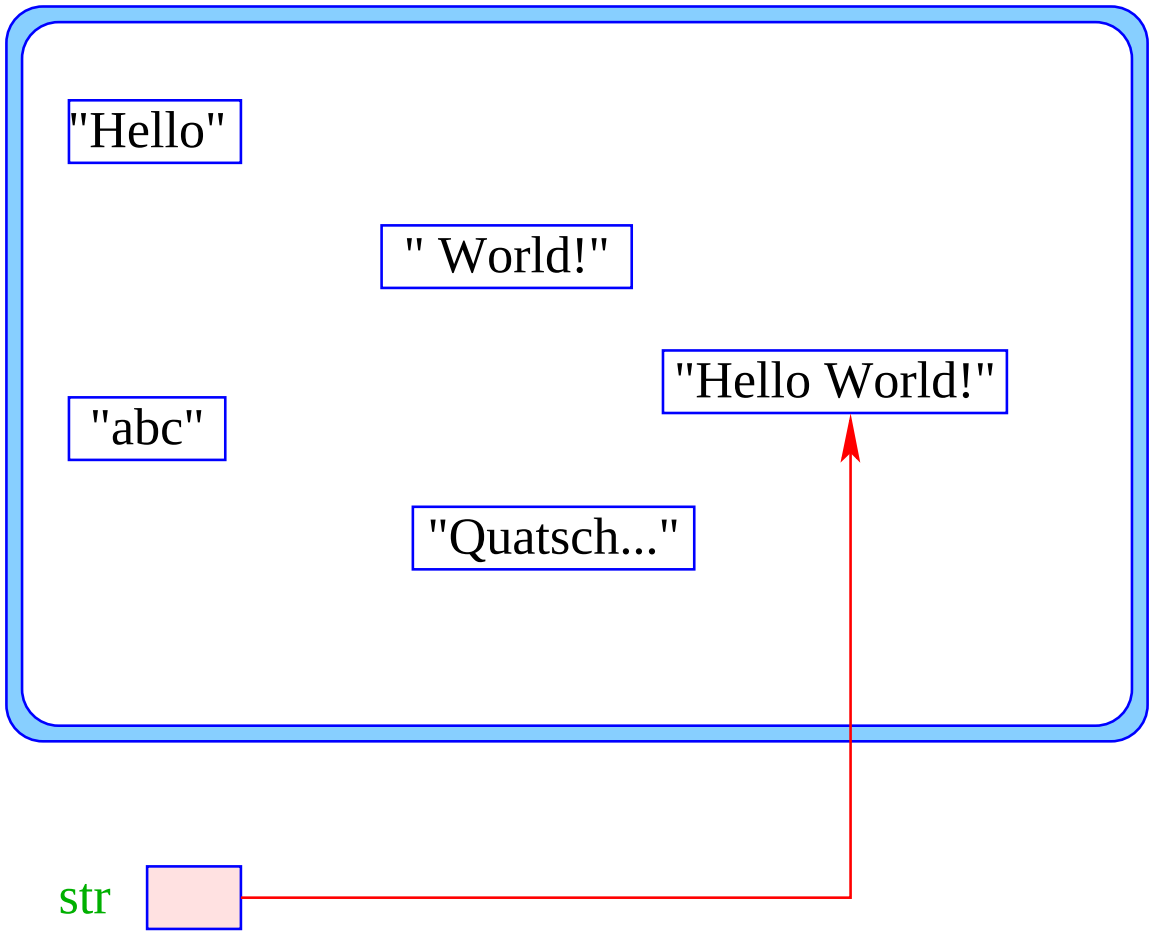
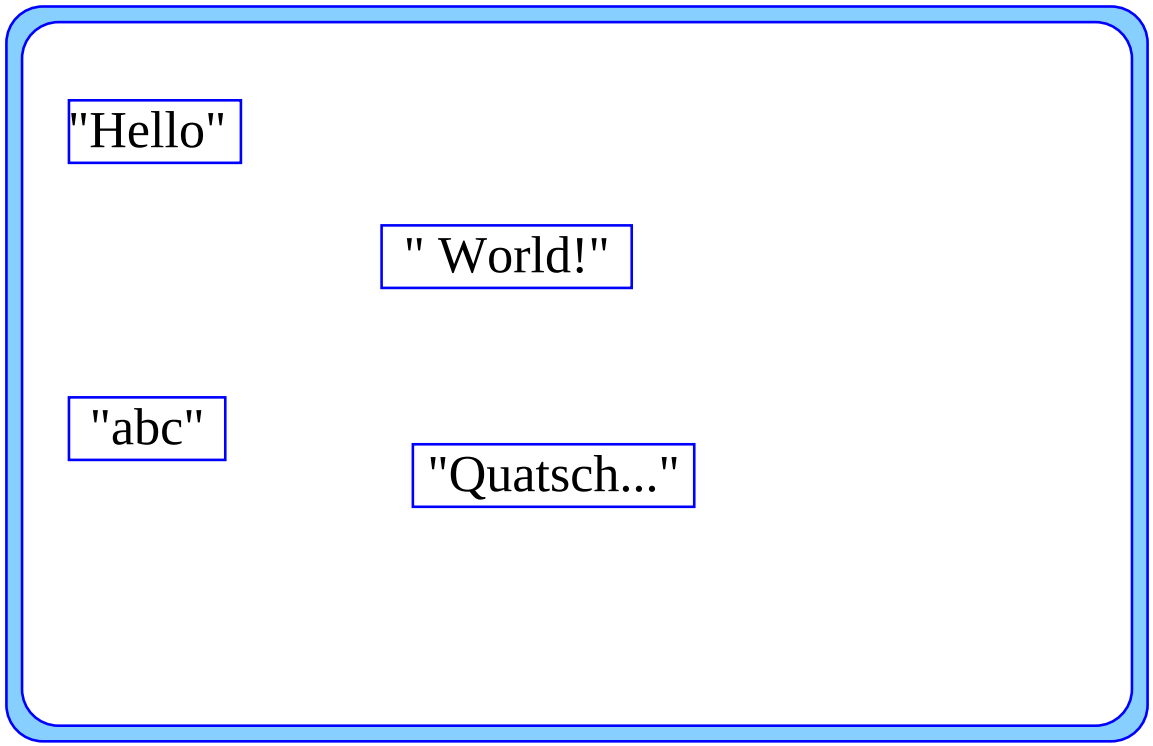
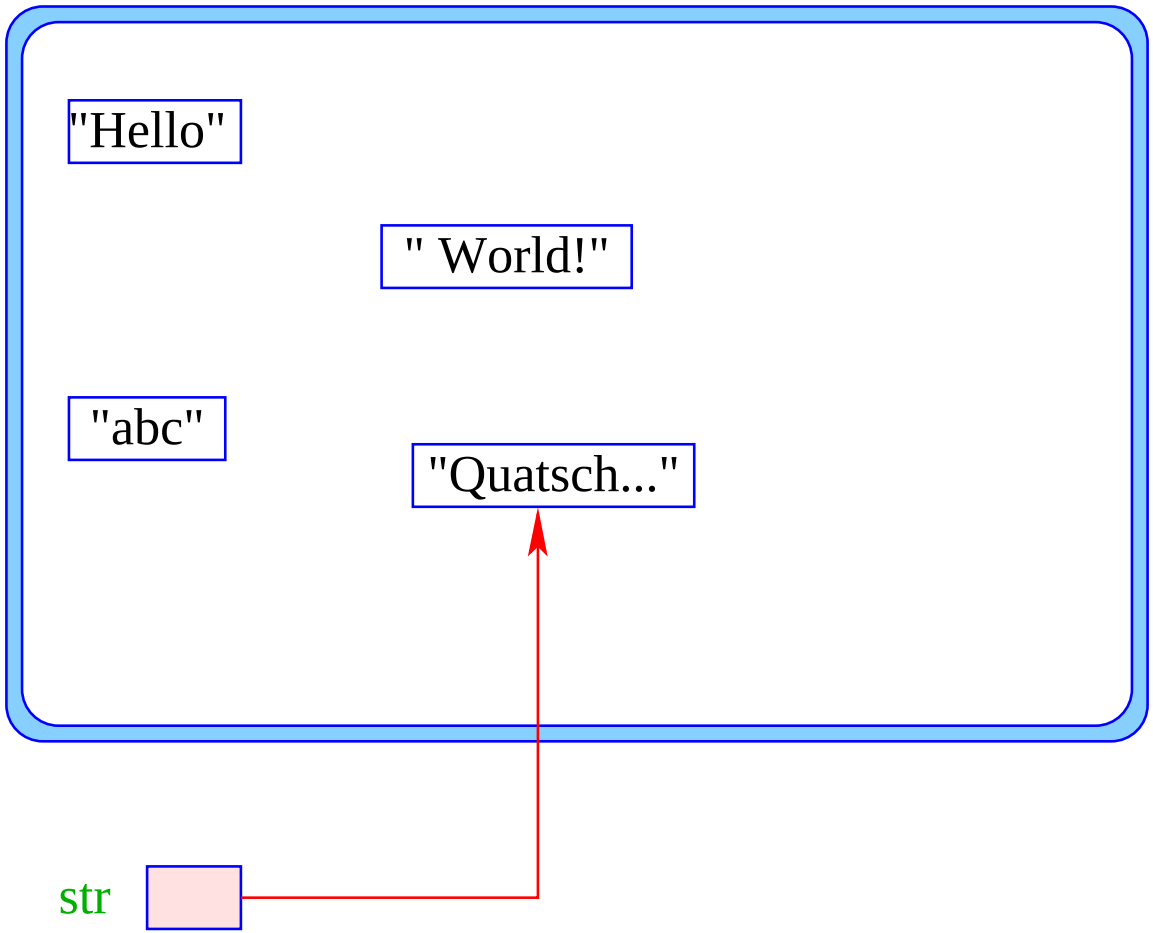
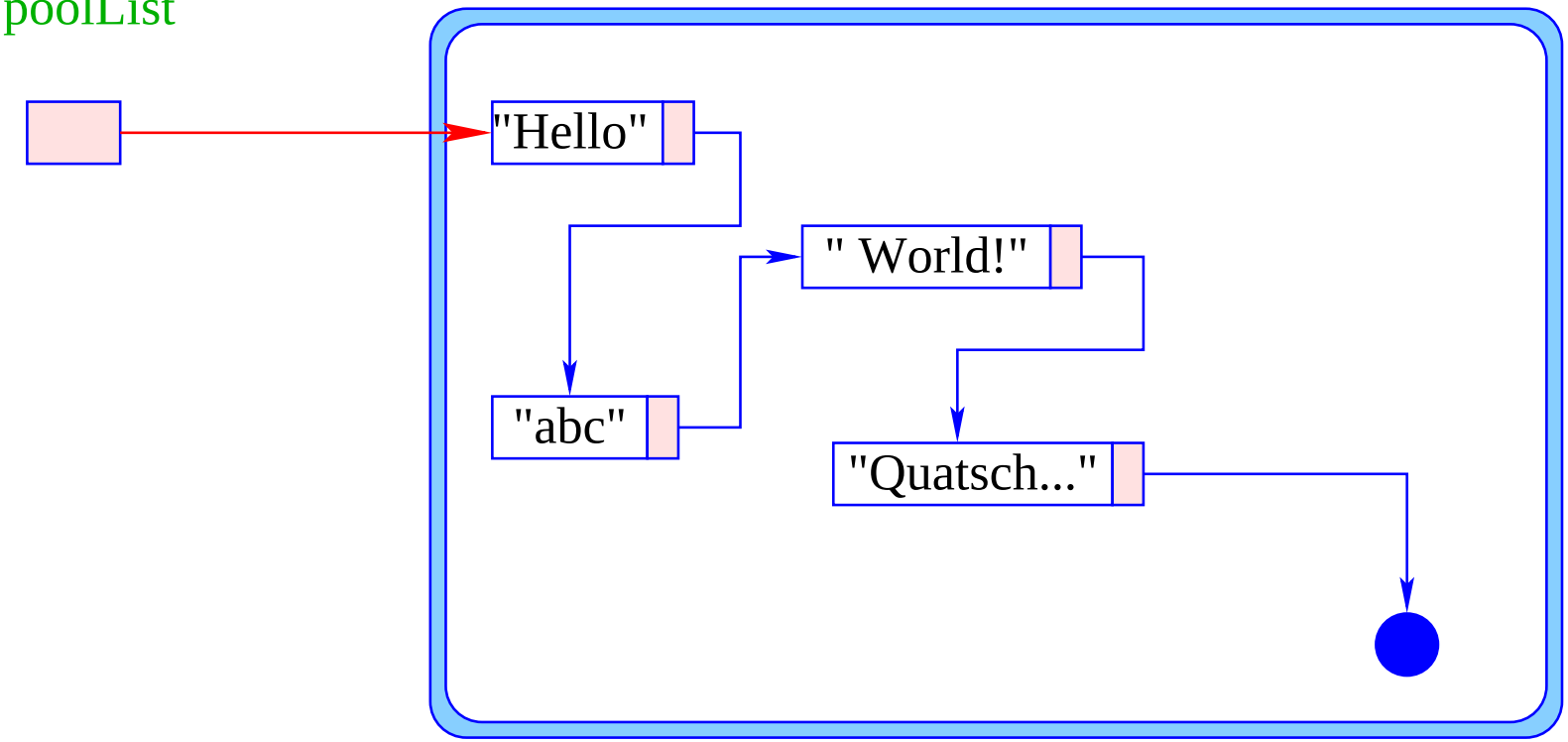
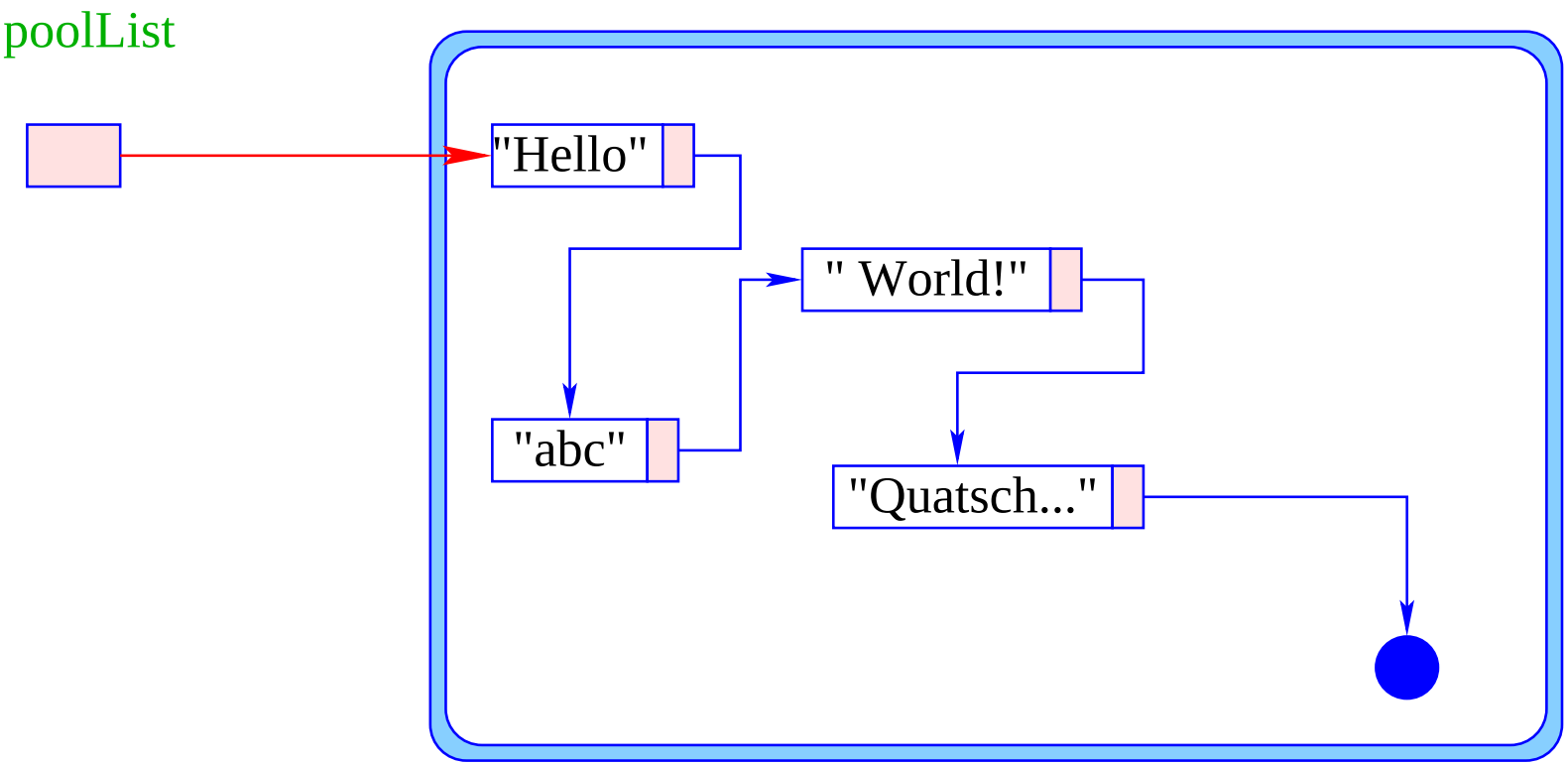
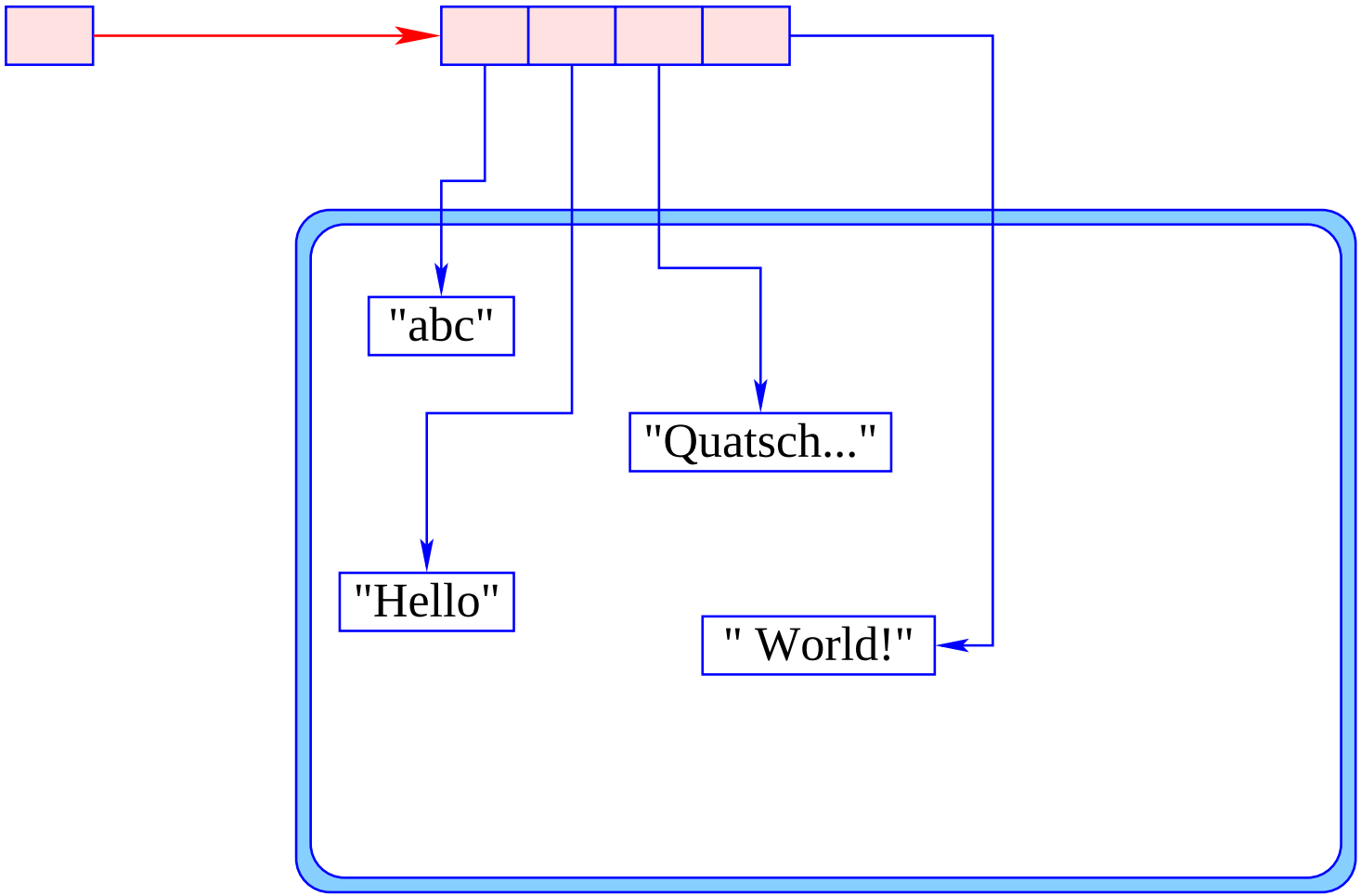
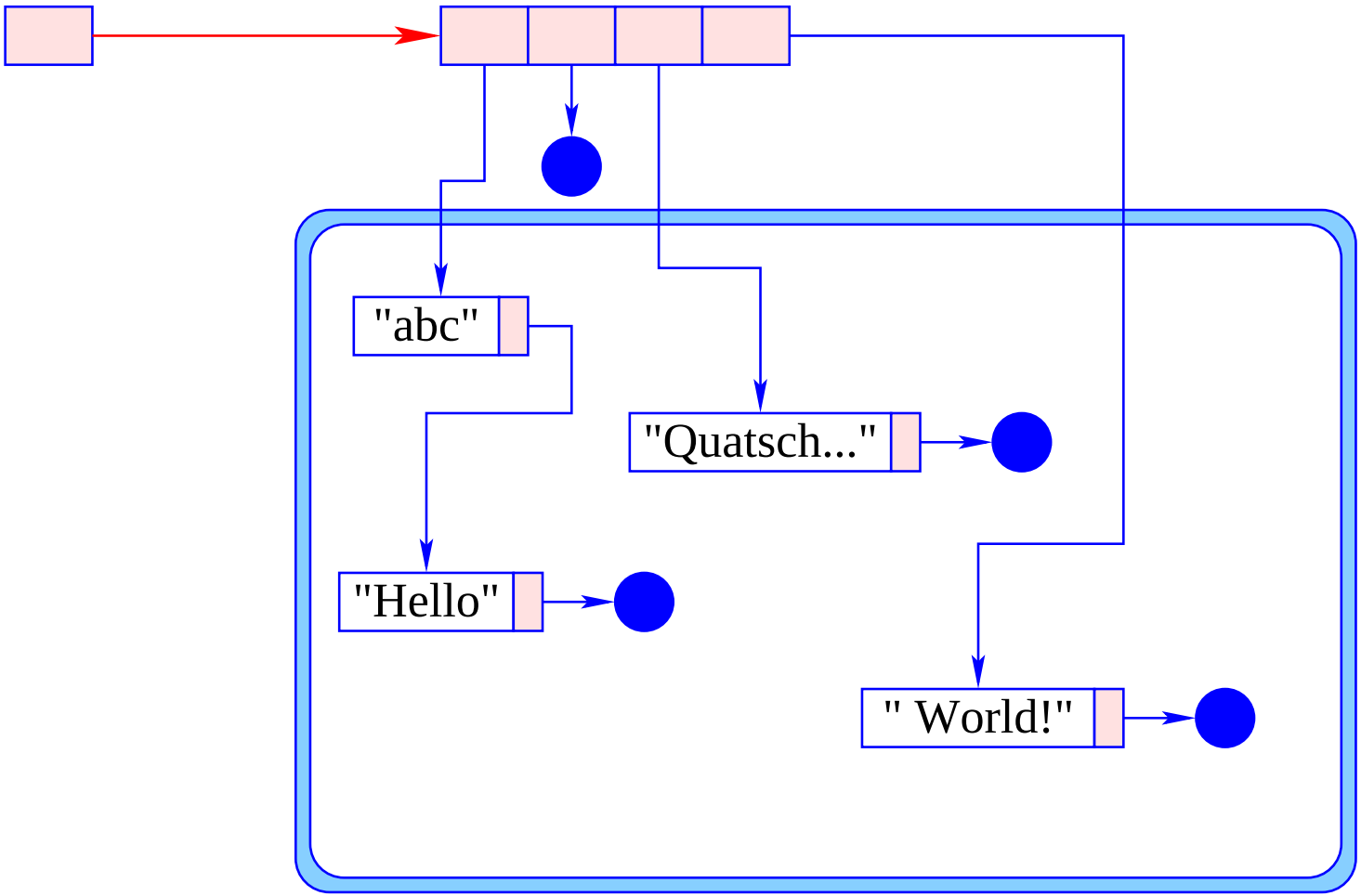
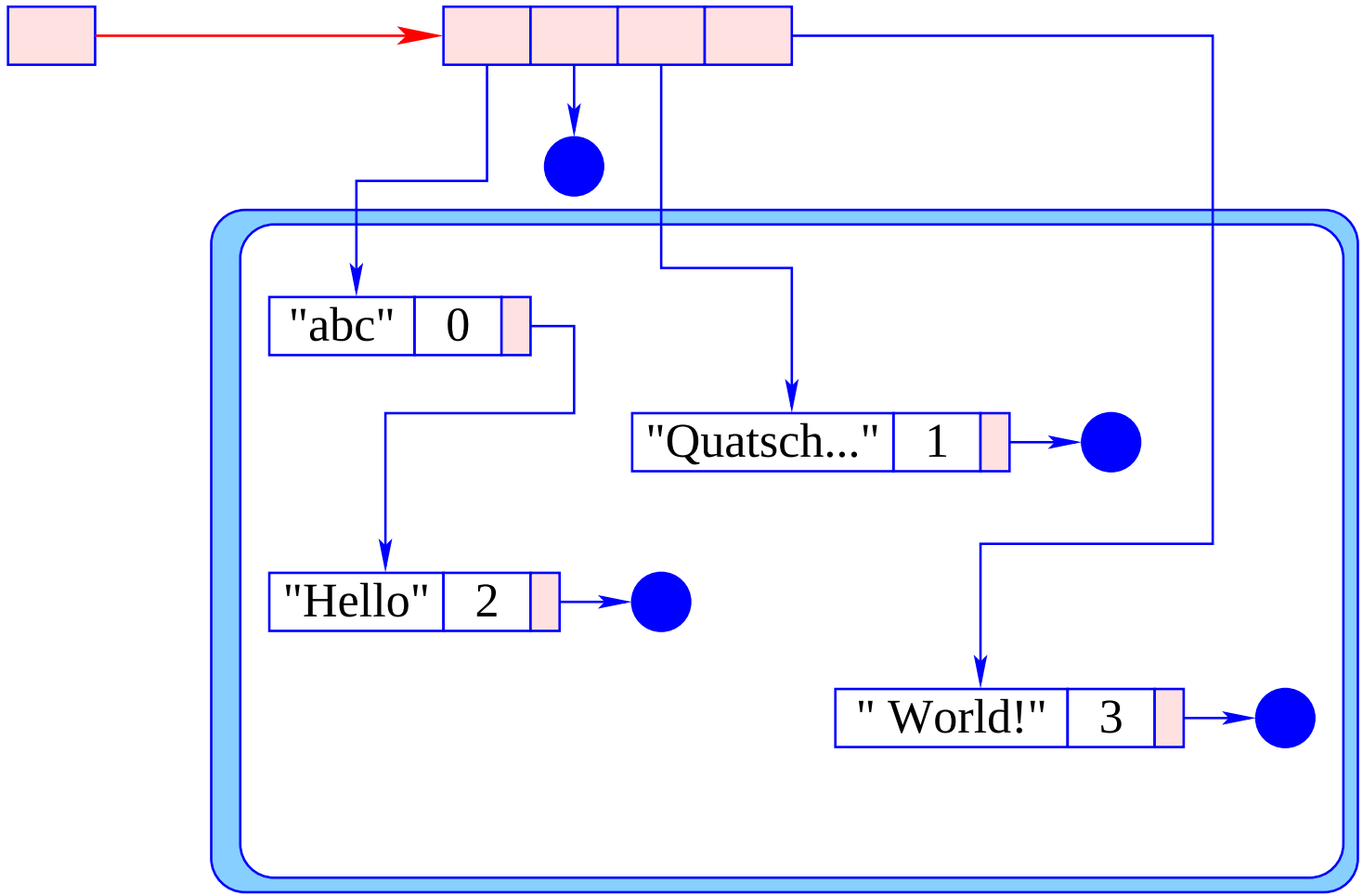
  void rekursivSchneien(PointListElement p1, PointListElement p2, int depth) {
    double angle, shiftx, shifty;
    Point2 hp;                    // helper
    PointListElement q1,q2,q3;  // turn p1–p2 into p1–q1–q2–q3–p2

    if(depth>0) {
      // move points to origin
      shiftx=p1.getPoint().getX();
      shifty=p1.getPoint().getY();
      p1.getPoint().set(0.0,0.0);
      p2.getPoint().translate(-shiftx, -shifty);

      // rotate to horizontal
      angle=Math.atan(p2.getPoint().getY()/p2.getPoint().getX());
      p2.getPoint().rotate(-angle);      // p1 is (0,0)
```

Implementierung: Flocke 2

```
// compute new points; then rotate; then translate
hp=new Point2(p2.getPoint().getX()/3.0,0.0);
q1=new PointListElement(hp.rotateThenTranslate(angle,shiftx,shifty));
hp=new Point2(p2.getPoint().getX()/2.0,
              p2.getPoint().getX()/6.0*Math.sqrt(3.0));
q2=new PointListElement(hp.rotateThenTranslate(angle,shiftx,shifty));
hp=new Point2(2.0*p2.getPoint().getX()/3.0,0.0);
q3=new PointListElement(hp.rotateThenTranslate(angle,shiftx,shifty));

// create new polygon
p1.setNext(q1);
q1.setNext(q2);
q2.setNext(q3);
q3.setNext(p2);

// undo translation and rotation
p2.getPoint().rotateThenTranslate(angle,shiftx,shifty);
p1.getPoint().translate(shiftx,shifty);
```

Erinnerung Flockenbildung: Horizontale Grundlinie

Aus jedem geraden Linienstück wird das mittlere Drittel entfernt und dafür ein gleichseitiges Dreieck aufgesetzt.

A



B



C



x^2/36 + h^2 = x^2/9
h=x*sqrt(3)/6

D



Implementierung: Flocke 3

```
        // create new flake for each segment
        rekursivSchneien(p1,q1,depth-1);
        rekursivSchneien(q1,q2,depth-1);
        rekursivSchneien(q2,q3,depth-1);
        rekursivSchneien(q3,p2,depth-1);
    }   // if depth>0
}
```

Alternative Implementierung: Flocke iterativ

```
void schneien(int depth) {
  double angle, shiftx, shifty;
  PointListElement p1, p2, q1,q2,q3, h;

  for (int i=0; i<depth; i++) {
    p1=first;
    while(p1.getNext()!=null) {
      p2=p1.getNext();
     // move points to origin
     ...
     // rotate to horizontal
     ...
     // compute new points; then rotate; then translate
     ...
     // create new polygon
     ...
     // undo translation and rotation
     ...
     p1=p2;
    }
  }
}
```

Übersicht

Beispiel 3: Binäre Suche

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 !"
- Haben wir schon gesehen!

Naive Implementierung

```
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;
   }
```

Suchen



Suchen



Suchen



Suchen



Suchen



Suchen



Suchen



Analyse

- 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 ...

Binäre Suche



Binäre Suche



Binäre Suche



Binäre Suche



Binäre Suche



Analyse

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

Idee:

Wir führen eine Hilfsfunktion

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

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

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

Implementierung find0

```
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;
}
```

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.

Ausführung



Ausführung



Ausführung



Ausführung



Ausführung



Ausführung



Ausführung



Ergebnis 5

Ausführung



Ergebnis 5

Beobachtung

- 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.
- ... Und die kennen wir bereits!
- Aktiv ist jeweils nur der oberste/letzte Aufruf.
- Achtung: es kann zu einem Zeitpunkt mehrere weitere inaktive Aufrufe der selben Funktion geben !!!

Terminierung

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, sondern ersetze den bereits dort liegenden Aufruf !

End-Rekursion



End-Rekursion



End-Rekursion



End-Rekursion



End-Rekursion



Ergebnis: 5

End-Rekursion

$\Longrightarrow$ 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
```

Kontrollfluss



Schleifen verlassen

- Die Schleife wird hier alleine durch die `return`-Anweisungen verlassen.
- Offenbar können Schleifen mit mehreren Ausgängen Sinn ergeben.
- 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:
  - Code zur Parameter-Übergabe;
  - einen Sprung an den Anfang des Rumpfs.
- Aber Achtung, wenn die Funktion an mehreren Stellen benutzt wird !!!
  (Was ist das Problem ?)
- Nicht unbedingt der sauberste Programmierstil!


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

Übersicht

Beispiel 4: Mergesort rekursiv

- Zur Erinnerung: Eingabe zwei (bisher: sortierte) Listen, Ausgabe: eine sortierte Liste mit exakt den Elementen der Eingabelisten
- Rekursive Implementierung mit Listen

```
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;
   }
   ...
```

Beispiel 4: Mergesort rekursiv

- 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 ...

Code merge()

```
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;
   }
}
```

Kontrollfluss



Sortieren durch Mischen

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

Beispiel


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

Code sort

```
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);
}
```

Code half

```
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;
}
```

Listen halbieren




```
b = a.half();
```

Listen halbieren



Listen halbieren



Listen halbieren



Übersicht

Suchmaschine

Objekte identifizieren



PageRank



Klassenentwurf



Komplexe Vektorähnlichkeit

Objekterzeugung



Objektmethoden



Sortieren nach Vektorähnlichkeit

Vektorähnlichkeit



Worthäufigkeiten



Dokumentensammlung

Suchmaschine:
Spidern/Crawlen der Dokumentenstruktur

- Suchen nach neuen Dokumenten durch Dokumentenverweise



Suchmaschine: PageRank

- Bewertung der Dokumente an Hand der Dokumentenstruktur



Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

8. Fortgeschrittene Programmierkonstrukte

Übersicht

Laufzeitfehler

- Tritt während der Programmausführung ein Fehler auf, wird die normale Ausführung abgebrochen und ein Fehlerobjekt erzeugt (geworfen).
- Die Klasse `Throwable` fasst alle Arten von Laufzeitfehlern 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 vordefinierte Fehler-Klassen



Throwable

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

Keine Fehlerbehandlung?

Tritt ein Fehler auf und wird nicht behandelt, bricht die Programmausführung 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
```

Fehlermeldung

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.

Fehlerbehandlung I

Soll die Programmausführung nicht beendet werden, muss der Fehler abgefangen werden.

Beispiel: `NumberFormatException`

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

Fehlerbehandlung II

- 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 ...

Fehlerbehandlung III: Code

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

Beispielablauf

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

Ausnahmebehandlung I

- 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 Programmausführung direkt hinter dem Handler fort.
- Wird eine Exception ausgelöst, durchsucht der Handler mithilfe des geworfenen Fehler-Objekts sequentiell die `catch`-Regeln.

Ausnahmebehandlung II

- 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.

Zur Laufzeit



Zur Laufzeit



Zur Laufzeit



Zur Laufzeit



Zur Laufzeit

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

Achtung

- Die Folge fss 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 fss 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();
   }
   ...
```

Beispiel

```
    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
```

Spezialisierte Fehler

- 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.
- Exceptions dürfen nicht eingesetzt werden, um Programmierfehler zu beheben!

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) { }`

Übersicht

Strings

- 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.

Strings

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);
```

Strings

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

Einige Konstruktoren

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

Einige Objekt-Methoden

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

... und 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:



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.

String-Pool




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




String-Pool



String-Pool




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


String-Pool



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.

1. Idee

poolList



1. Idee



+ 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 ...

2. Idee

poolArray



2. Idee

+ Auffinden eines Worts im Pool ist einfacher (logarithmisch mit binärer Suche).
- 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.

3. Idee

hashSet



Auffinden der richtigen Liste

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

Beispiele

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 $p = 31$.)

Beispiele

`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
...		...	

Beispiele

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

Mögliche Implementierung von intern()

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

Erläuterungen

- 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`)

Stringpool mit Hashes

hashTable



Weitere Klassen zur Manipulation von Zeichenketten

- `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.

Übersicht

Motivation

Es gibt viele Möglichkeiten, geordnete Sammlungen von Objekten (oder Basiswerten) zu implementieren:

- Strings sind Sammlungen von Characters
- Listen und Arrays sind geordnete Sammlungen von Objekten eines definierten Typs
- usw.

Wenn man auf die einzelnen Elemente zugreift, möchte man oft

- Wissen, ob es noch ein weiteres Element gibt
- Zum nächsten Element springen
- Ein Element nach Abarbeitung entfernen

Das wird durch das Interface `Iterable<T>` mit entsprechenden Konstruktoren sowie Methoden `hasNext()`, `next()` und `remove()` definiert.

Das Interface `Iterable<T>`

Das Interface `Iterable<T>` definiert die Methode **`public`** `Iterator<T> iterator()`.

Die implementierende Klasse muss somit die Methode `iterator()` zur Verfügung stellen, die einen `Iterator<T>` als Ergebnis hat.

Beispiel: Eine Klasse `IterableString`, bei der über die einzelnen `Character` des `String` iteriert werden kann.

```
class IterableString implements Iterable<Character> {
   private String str;
   public IterableString(String str) {
      this.str = str;
   }

   public Iterator<Character> iterator() {
      return new IterableStringIterator(str);
   }
}
```

Iterable<T> und Iterator<T> (I)

- Die zugehörige Klasse IterableStringIterator:

```
class IterableStringIterator implements Iterator<Character> {
   private String str;
   private int count = 0;

   public IterableStringIterator(String str) { this.str = str; }

   public boolean hasNext() { return count < str.length(); }

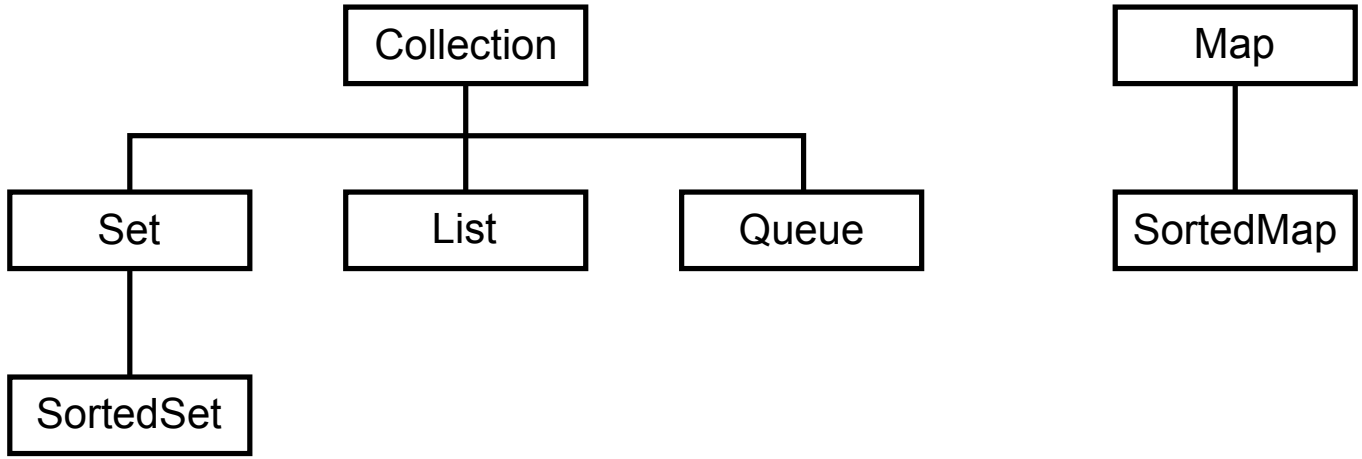
   public Character next() {
      if (count == str.length()) {
         throw new NoSuchElementException();
      }
      return str.charAt(count++);
   }

   public void remove() {
      throw new UnsupportedOperationException();
   }
}
```

Iterable<T> und Iterator<T> (II)

- Verwendung 1:

```
IterableString s = new IterableString("super␣duper␣Iteratoren");
Iterator<Character> it = s.iterator();

while (it.hasNext()) {
   System.out.print(it.next());
}

System.out.println();
```

- Verwendung 2:

```
IterableString s = new IterableString("super␣duper␣Iteratoren");

for (Character c : s) {
   System.out.print(c);
}

System.out.println();
```

Übersicht

Das Java Collections Framework in java.util

- Collection = Sammlung von Objekten
- Einheitliche Architektur zur Repräsentation und Manipulation von Collections
- Abstraktion von der Implementierung der Collections
- Verringerung des Programmieraufwandes
- Erhöhung der Performanz von Operationen auf Collections
- Interoperabilität voneinander unabhängiger Collections

http://download.oracle.com/javase/tutorial/collections/index.html

Java Collections Framework – Interfaces

- Haupt-Interfaces des Collection-Frameworks:



- Die Haupt-Interfaces kapseln verschiedene Arten von Collections
- Ermöglichen die implementierungsunabhängige Manipulation der Collections
- Haupt-Interfaces bilden eine Hierarchie

Das Interface `Collection<E>` (I)

- Alle Collection-Interfaces sind generisch:

  ```
  public interface Collection<E> { ...
  ```

- Bei Deklaration einer Collection kann und sollte angegeben werden, welche Elemente enthalten sind:

  ```
  Collection<Point> c;
  ```

  $\Longrightarrow$ Compiler kann prüfen, ob die der Collection hinzugefügten Elemente vom Typ `Point` sind.

- `Collection<E>` trifft beispielsweise keine Aussage darüber, ob
  - die Elemente der Collection geordnet sind
  - die Collection Duplikate enthält

- `Collection<E>` wird verwendet, wenn möglichst wenig Einschränkungen gelten sollen bzw. bekannt sind.

- `Set`, `List` und `Queue` sind spezifischere Collections

Das Interface `Collection<E>` (II)

```
public interface Collection<E> extends Iterable<E> {
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(E element);           //optional
   boolean remove(Object element); //optional
   Iterator<E> iterator();

   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c); //optional
   boolean removeAll(Collection<?> c);        //optional
   void clear();                              //optional

   Object[] toArray();
   <T> T[] toArray(T[] a);
}
```

Das Interface `List<E>` (I)

- Eine `List<E>` ist eine geordnete `Collection<E>`.
- Eine `List<E>` kann Duplikate enthalten.
- Zugriff auf einzelne Elemente mittels index-Position.
- `List<E>` definiert, unter anderem, folgende zusätzliche Methoden:

```
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);      //optional
    boolean add(E element);           //optional
    void add(int index, E element);   //optional
    E remove(int index);              //optional
    boolean addAll(int index,
                   Collection<? extends E> c); //optional

    int indexOf(Object o);
    int lastIndexOf(Object o);
}
```

Das Interface `List<E>` (II)

- Die Java-API stellt unter anderem die `List<E>`-Implementierungen `ArrayList` und `LinkedList` zur Verfügung.

Beispiel:

```
Collection<Point> c = new ArrayList<Point>();
c.add(new Point(3.0, 5.0));
c.add(new Point(-1.2, 3.1));
c.add(new Point(3.0, 5.0));
System.out.println(c.size());                    // Ausgabe: 3

List<Point> l = new LinkedList<Point>(c);
l.addAll(c);
System.out.println(c.size());                    // Ausgabe: 3
System.out.println(l.size());                    // Ausgabe: 6

c.addAll(l);
System.out.println(c.size());                    // Ausgabe: 9
```

Das Interface Map<K,V>

- Eine Map<K,V> bildet Schlüssel vom Typ K auf Werte vom Typ V ab.
- Eine Map<K,V> modelliert daher eine mathematische Funktion.
- Eine Map<K,V> enthält keinen Schlüssel mehrmals.
- Jeder Schlüssel bildet auf höchstens einen Wert ab.

```
public interface Map<K,V> {
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   void putAll(Map<? extends K, ? extends V> m);
   void clear();

   public Set<K> keySet();
   public Collection<V> values();
}
```

Die Klasse `Collections`: Algorithmen (I)

```
public static boolean disjoint(Collection<?> c1, Collection<?> c2)
// true, falls kein Element sowohl in c1 als auch in c2 enthalten ist

public static int frequency(Collection<?> c, Object o)
// Anzahl der Elemente in c, die, gemaess equals, identisch zu o sind

public static void reverse(List<?> l)
// Kehrt die Reihenfolge der Elemente in l um

public static <T> boolean replaceAll(List<T> l, T oldV, T newV)
// Ersetzt jedes zu oldV identische Element in l mit newV

public static <T extends Comparable<? super T>> void sort(List<T> l)
// Sortiert l gemaess der Methode compareTo des Typs T

public static <T> void sort(List<T> list, Comparator<? super T> c)
// Sortiert l gemaess dem Comparator c
```

Die Klasse `Collections`: Algorithmen (II)

Beispiel:

```
List<Integer> l = new ArrayList<Integer>();
l.add(18); l.add(46); l.add(18); l.add(12);
Set<Integer> s = new TreeSet<Integer>();
s.addAll(l);

System.out.println(l);                                  // [18, 46, 18, 12]
Collections.reverse(l);
System.out.println(l);                                  // [12, 18, 46, 18]
Collections.sort(l);
System.out.println(l);                                  // [12, 18, 18, 46]

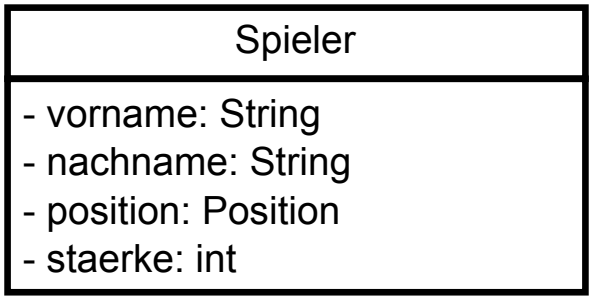
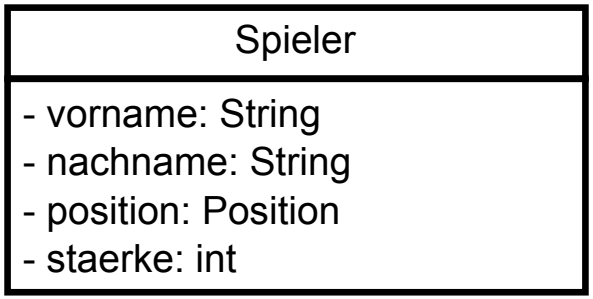
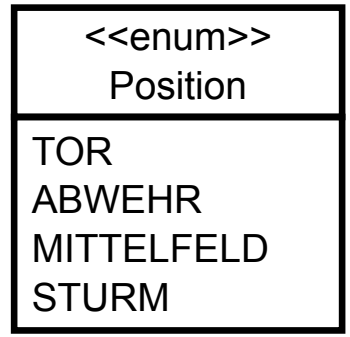
System.out.println(s);                                  // [12, 18, 46]
System.out.println(Collections.disjoint(l, s));         // false
System.out.println(Collections.frequency(l, 18));       // 2
System.out.println(Collections.frequency(s, 18));       // 1

Collections.replaceAll(l, 18, 22);
System.out.println(l);                                  // [12, 22, 22, 46]
```

Das Interface `Comparator<T>` (I)

- Das Interface `Comparator<T>` definiert die Methode
  `public int` `compare(T o1, T o2)`
  und damit eine totale Ordnung auf Objekten vom Typ `T`.
- Rückgabewert analog zu `compareTo` des Interfaces `Comparable`:
  - `< 0`, falls `o1` kleiner ist als `o2`
  - `== 0`, falls `o1` gleich groß ist wie `o2`
  - `> 0`, falls `o1` größer ist als `o2`

Beispiel: Inverse Sortierung von `Integer`.

```
class ReverseIntegerOrdering implements Comparator<Integer> {
   public int compare(Integer i1, Integer i2) { return i2 - i1; }
}

List<Integer> l = new ArrayList<Integer>();
l.add(18); l.add(46); l.add(18); l.add(12);

Collections.sort(l, new ReverseIntegerOrdering());
System.out.println(l);                              // [46, 18, 18, 12]
```

Das Interface `Comparator<T>` (II)

- Üblicherweise Verwendung mittels einer anonymen Klasse:

```
List<Integer> l = new ArrayList<Integer>();
l.add(18); l.add(46); l.add(18); l.add(12);

Collections.sort(l, new Comparator<Integer>() {
                        public int compare(Integer i1,
                                           Integer i2) {
                           return i2 - i1;
                        }
                    });

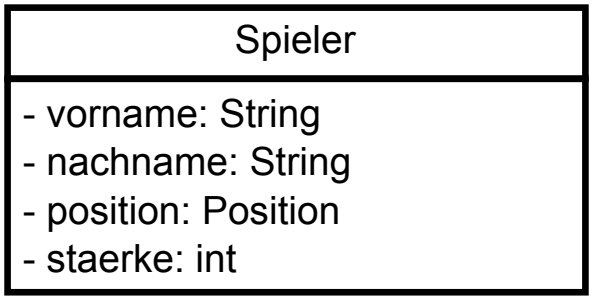
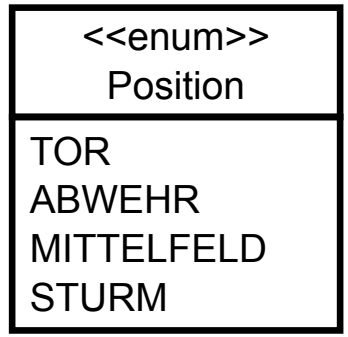
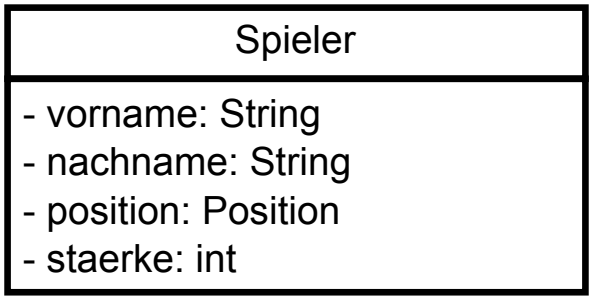
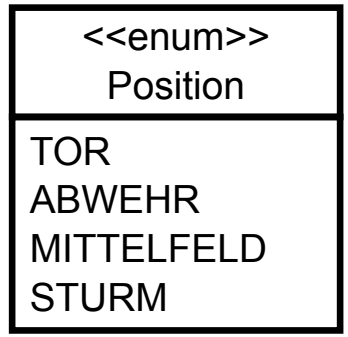
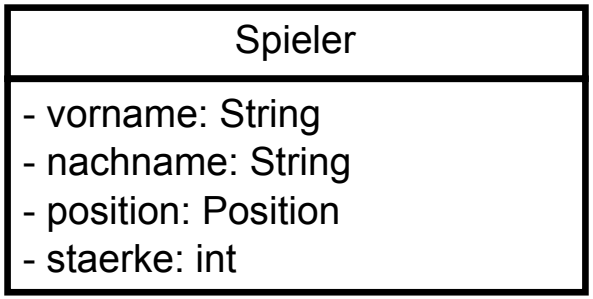
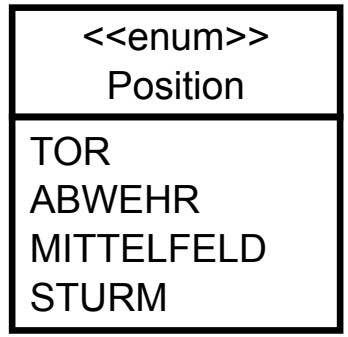
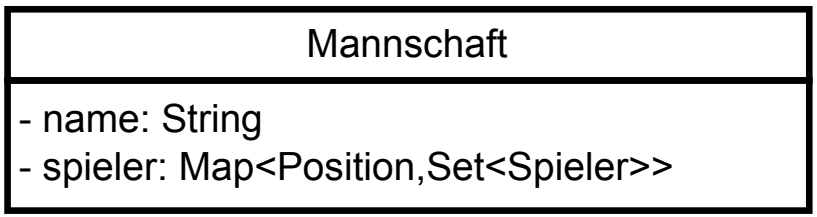
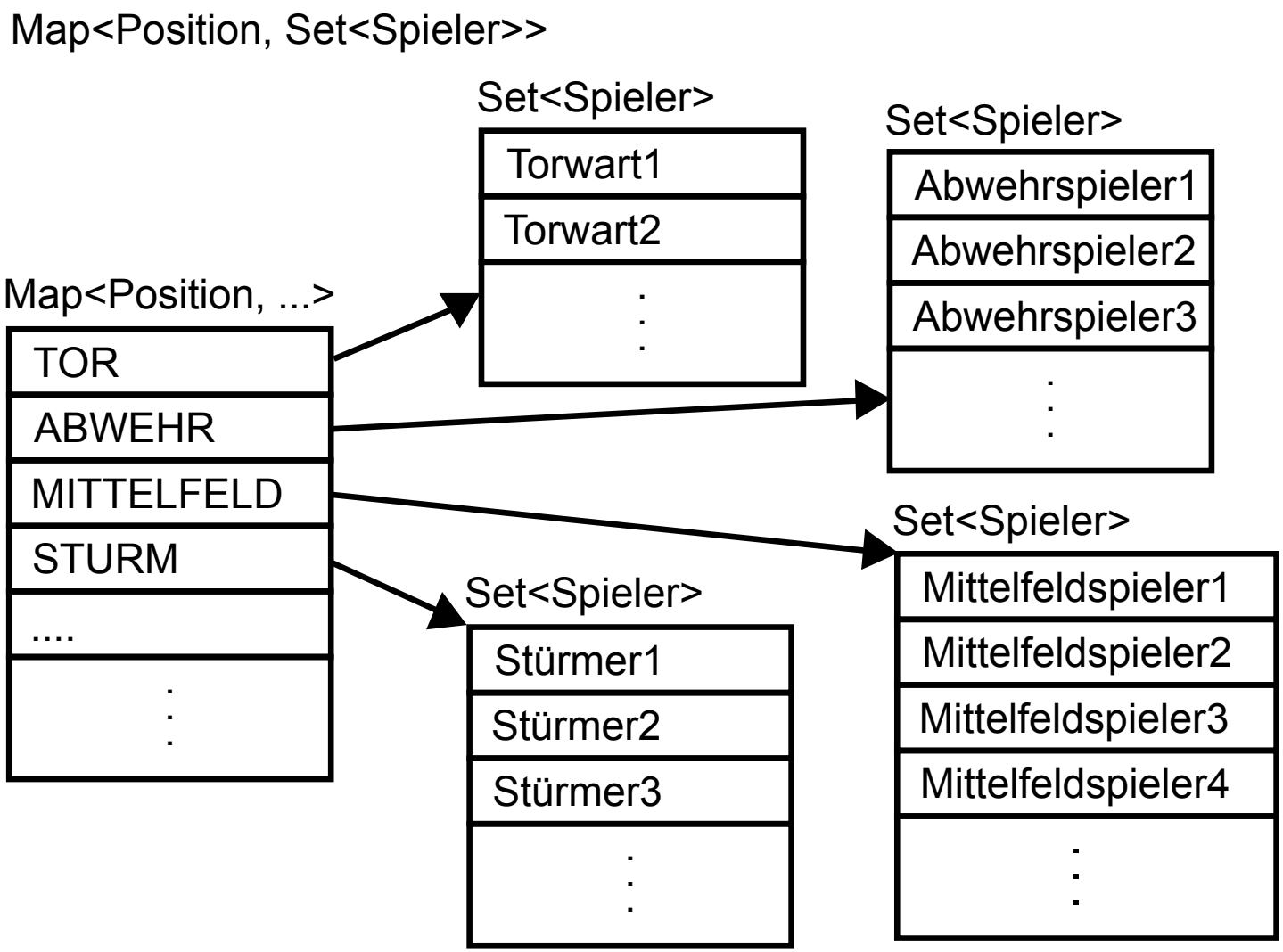
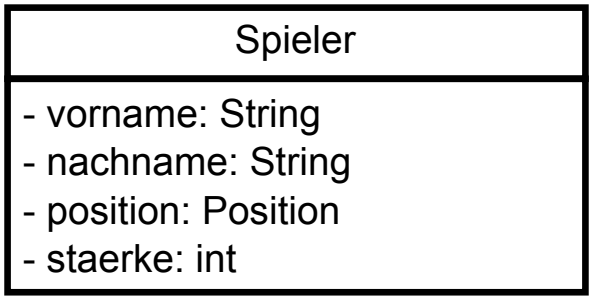
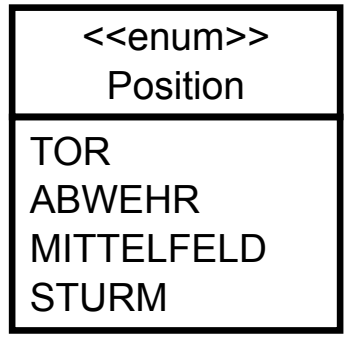
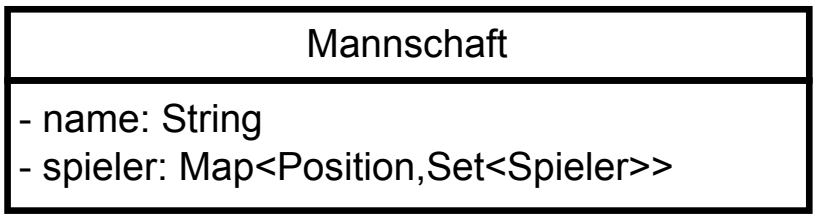
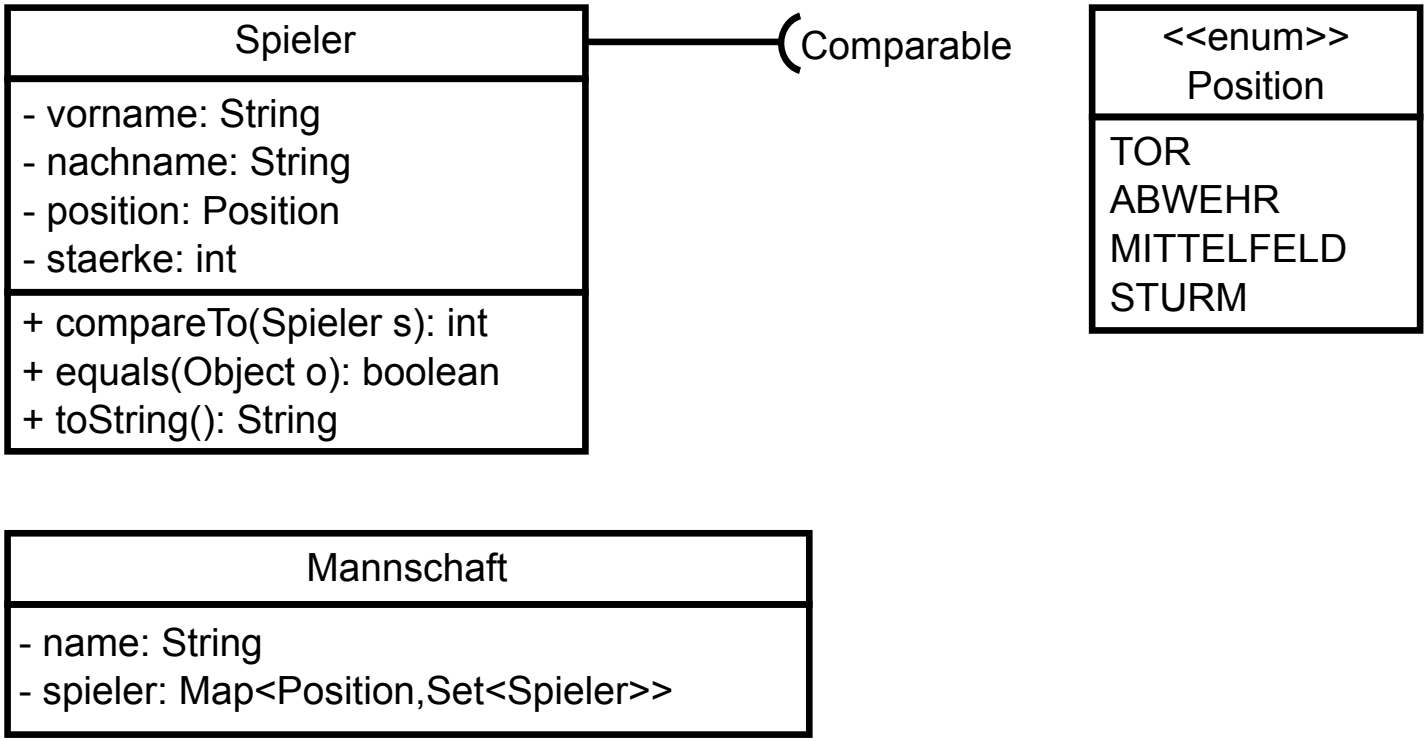
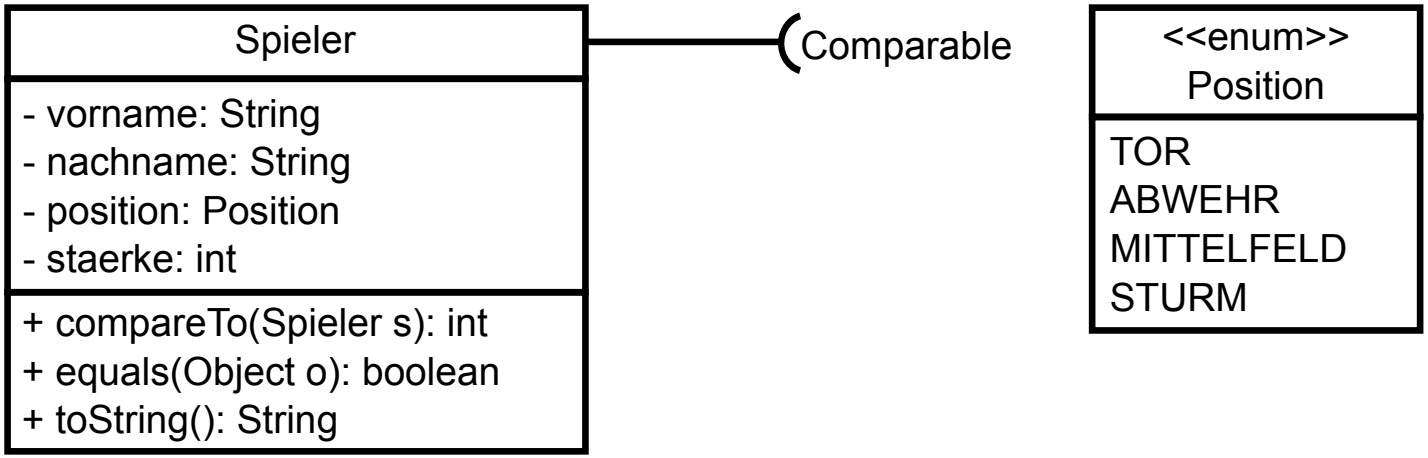
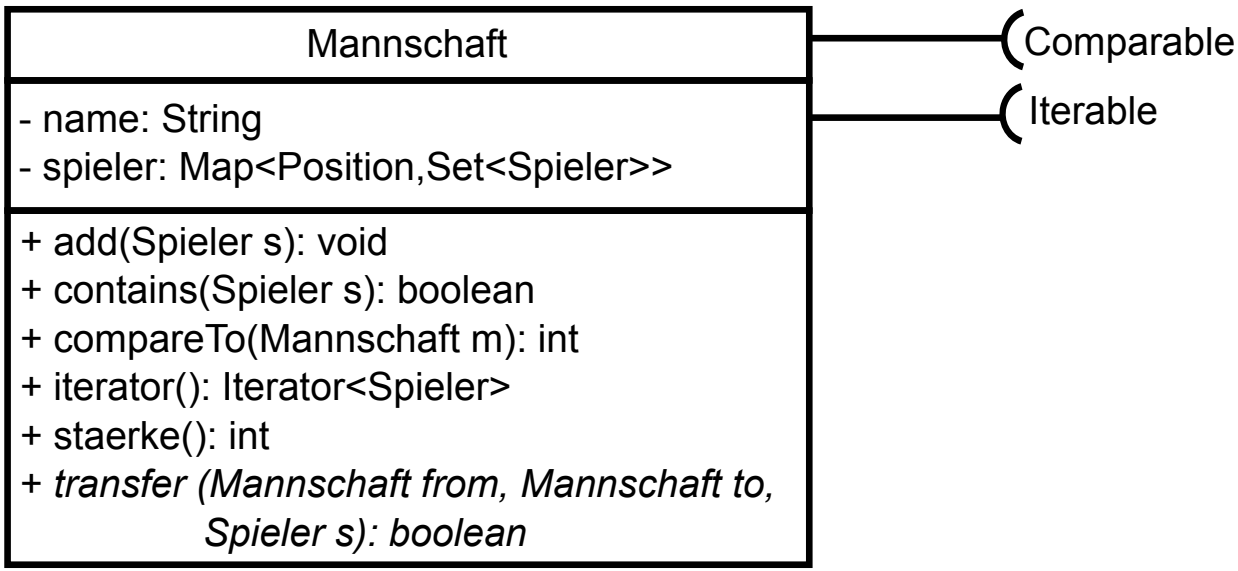
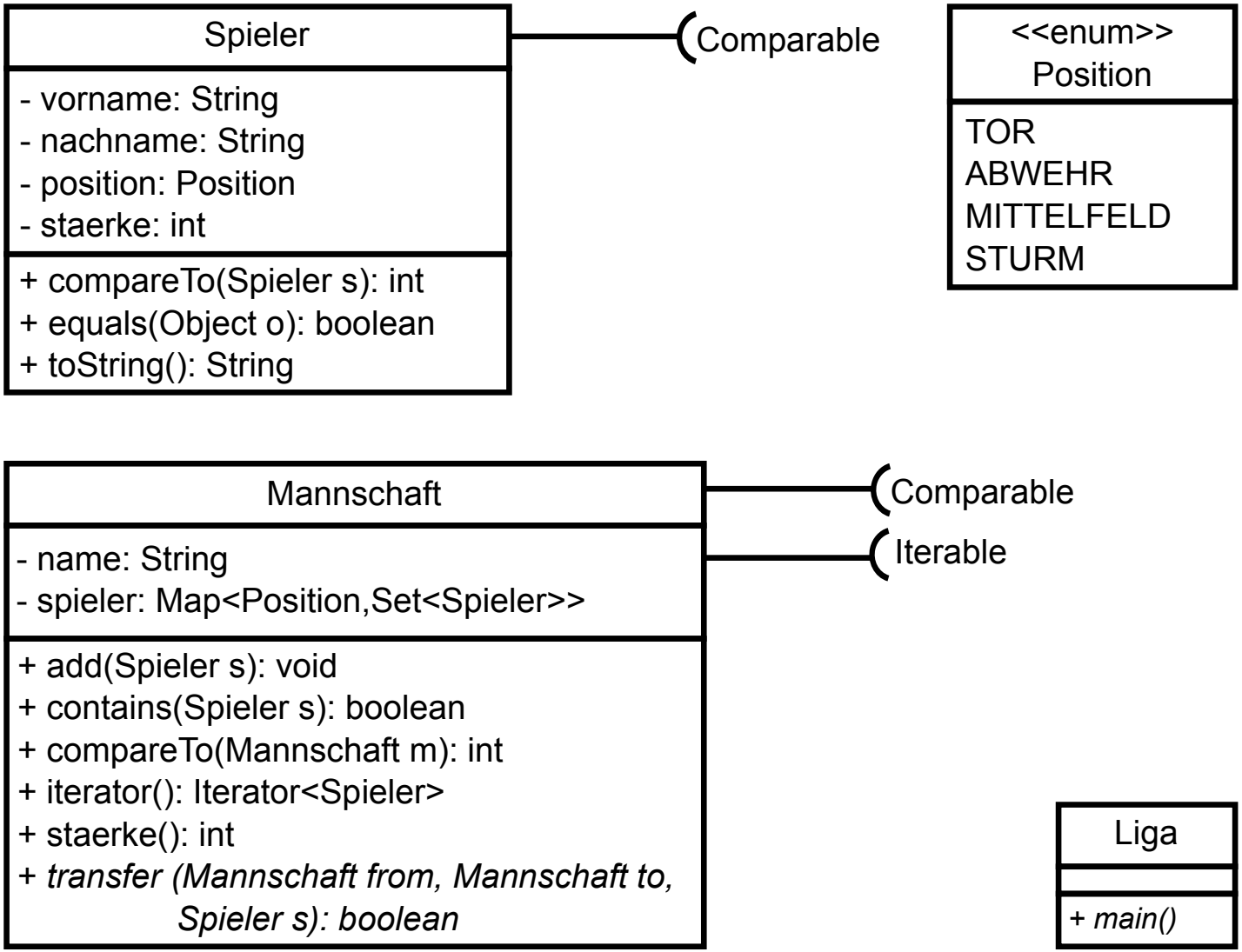
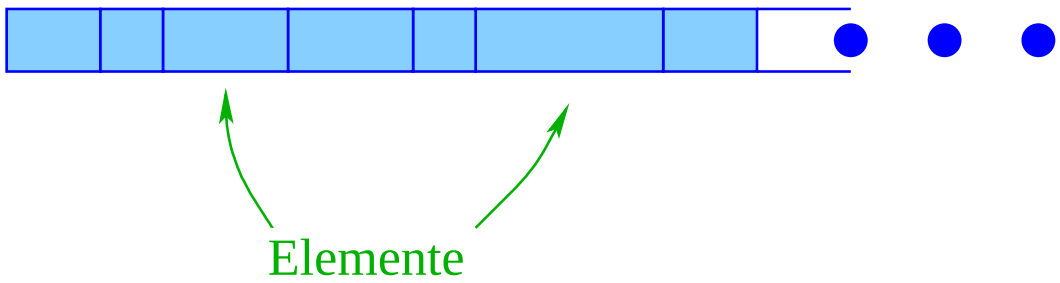
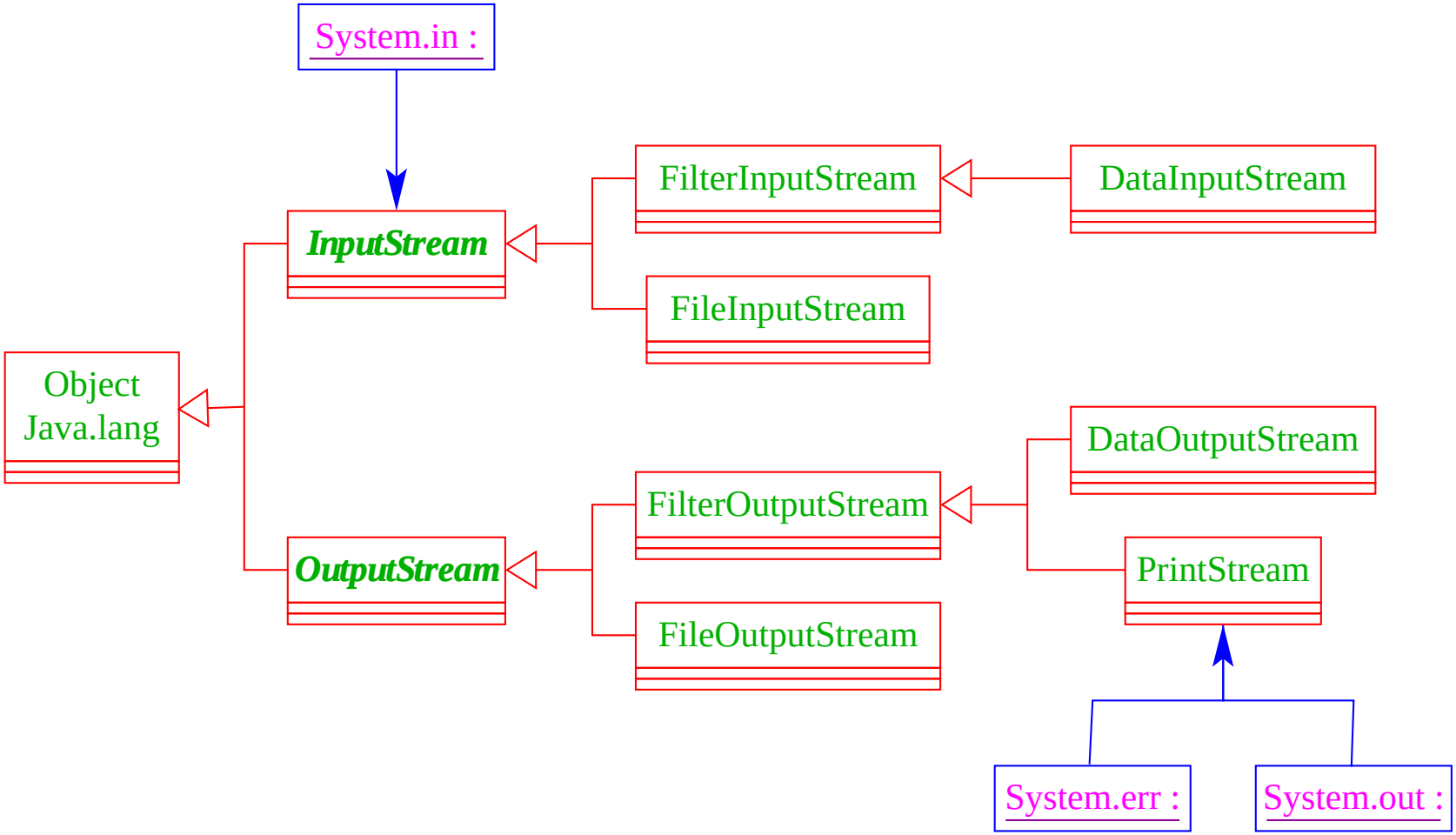
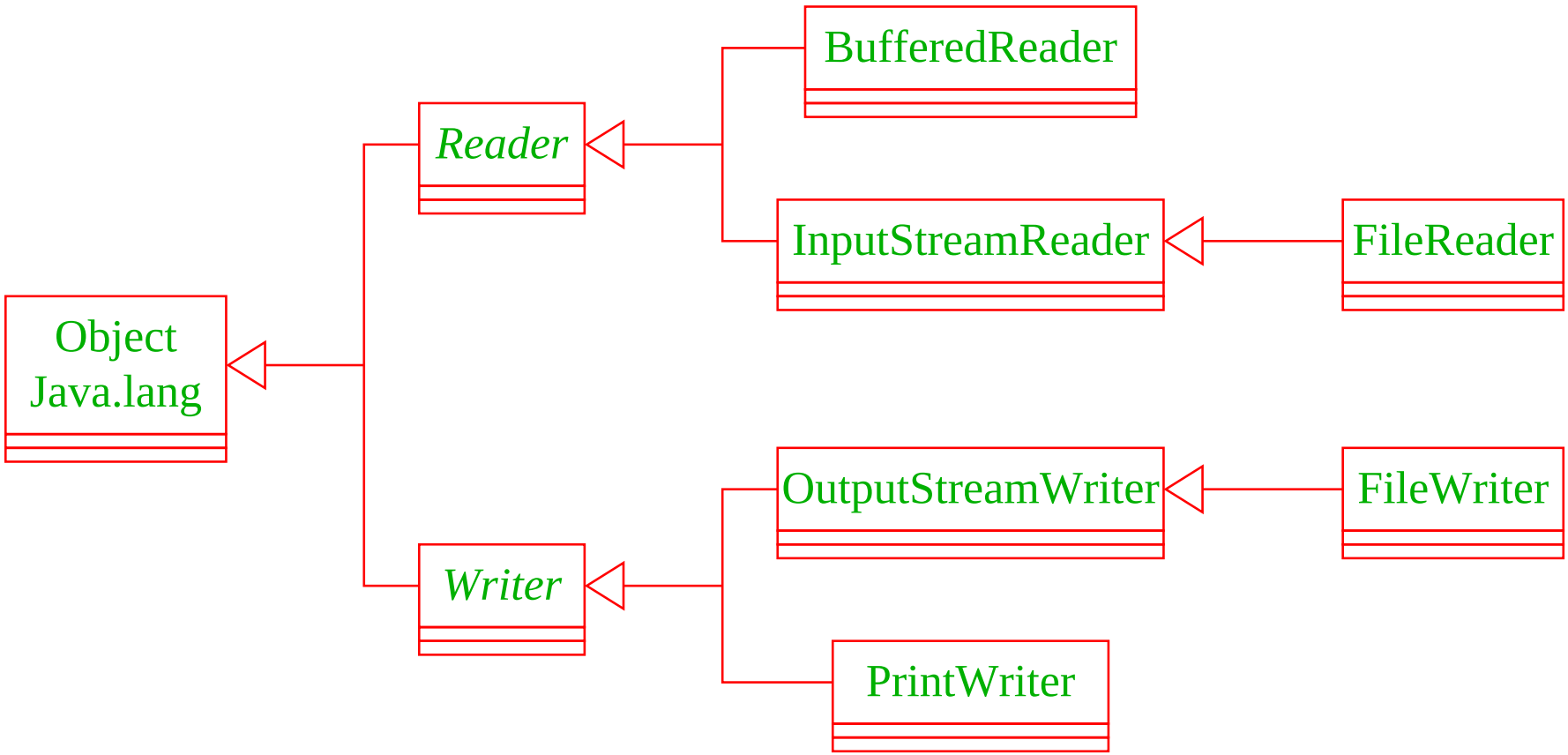
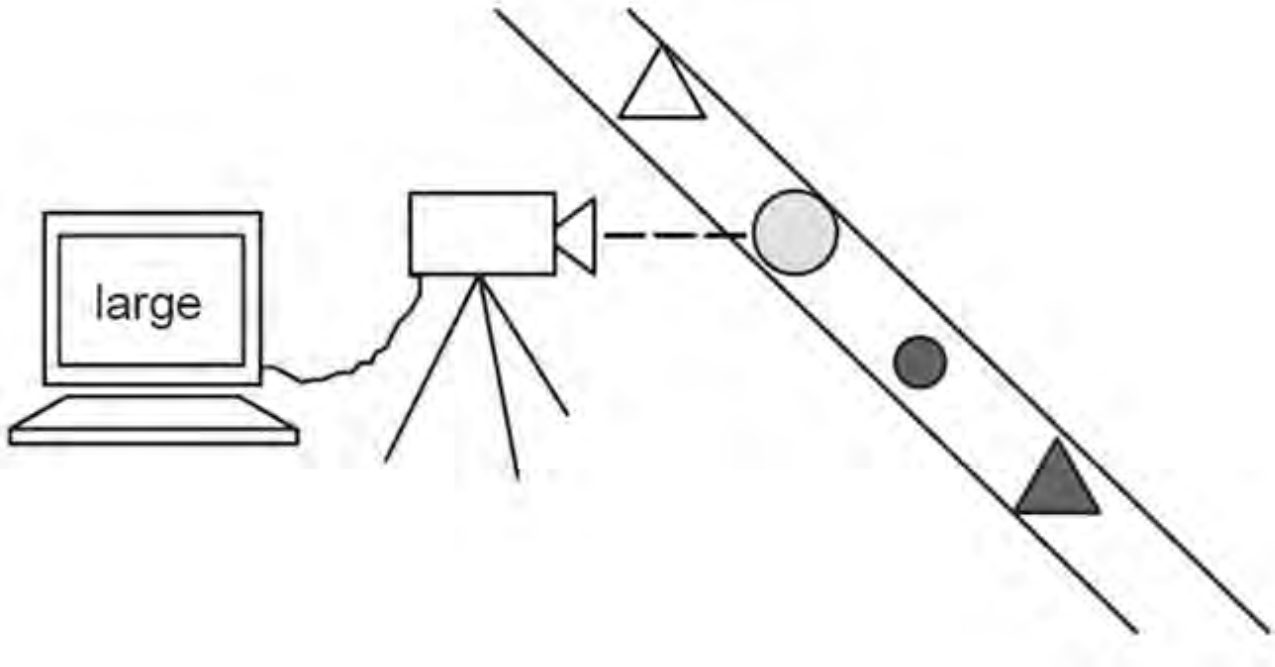
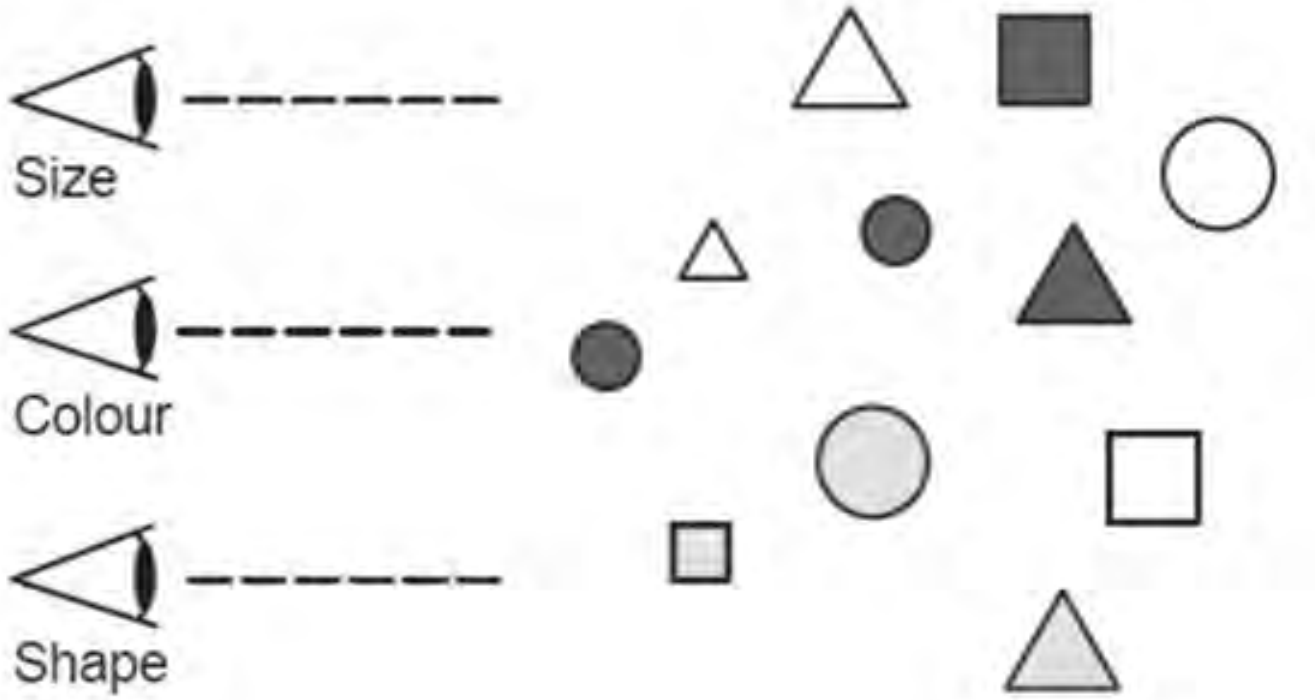
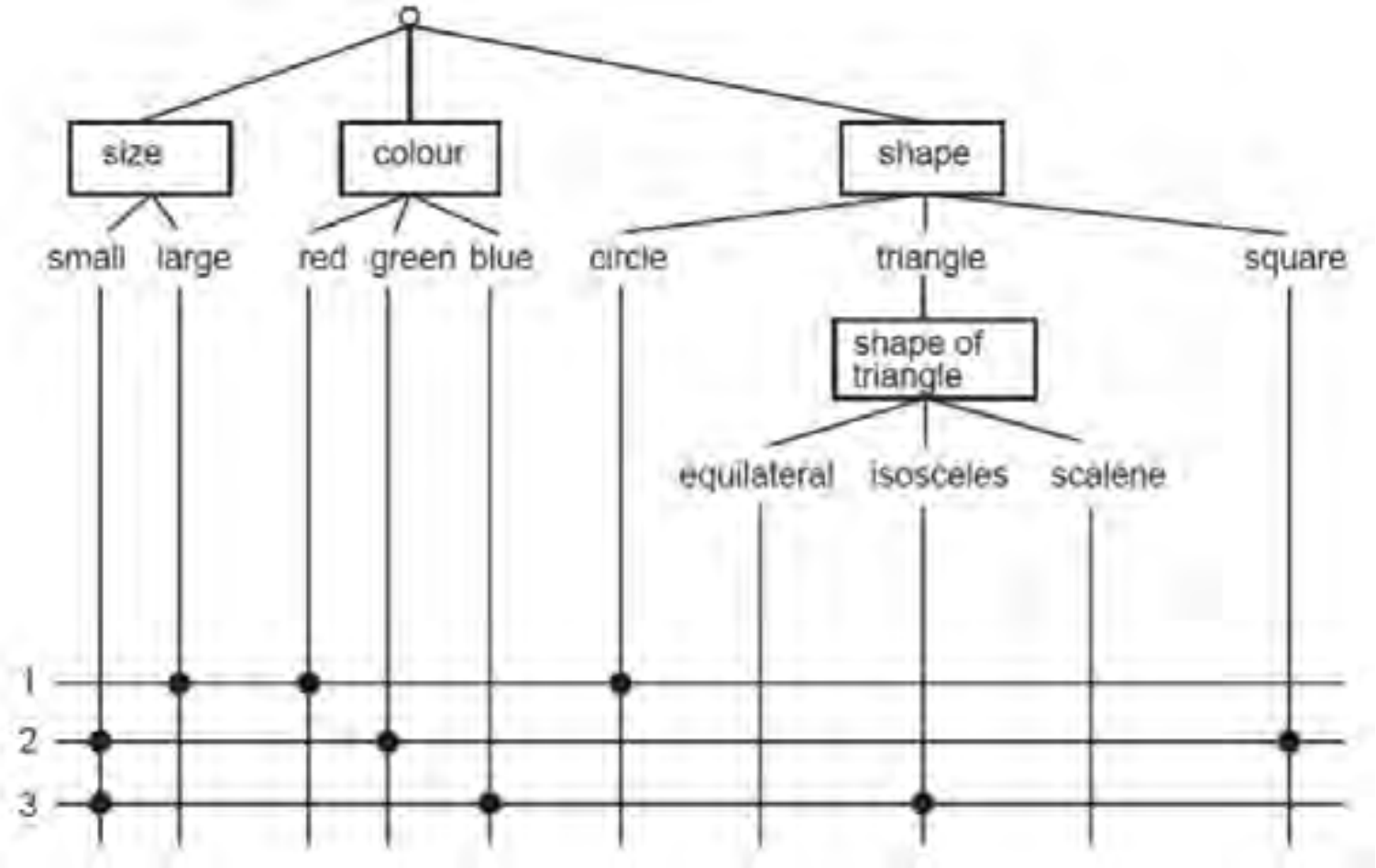
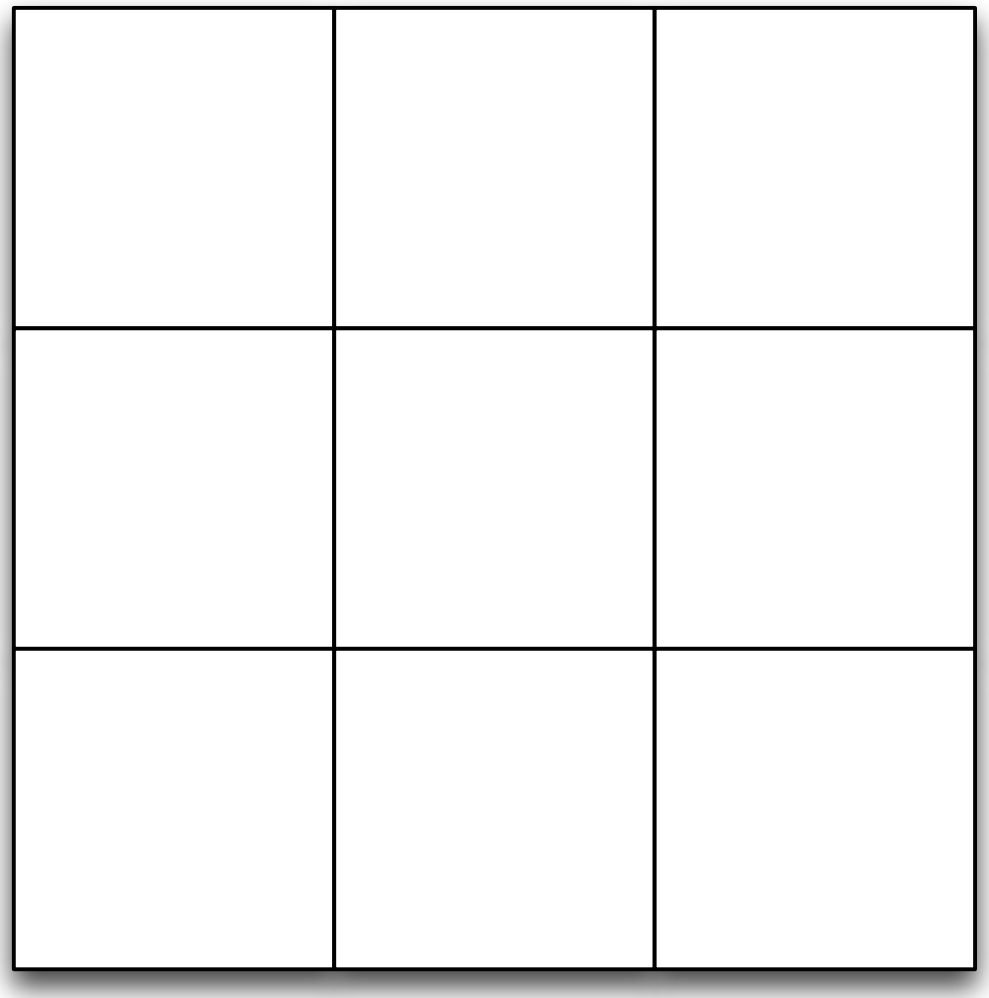
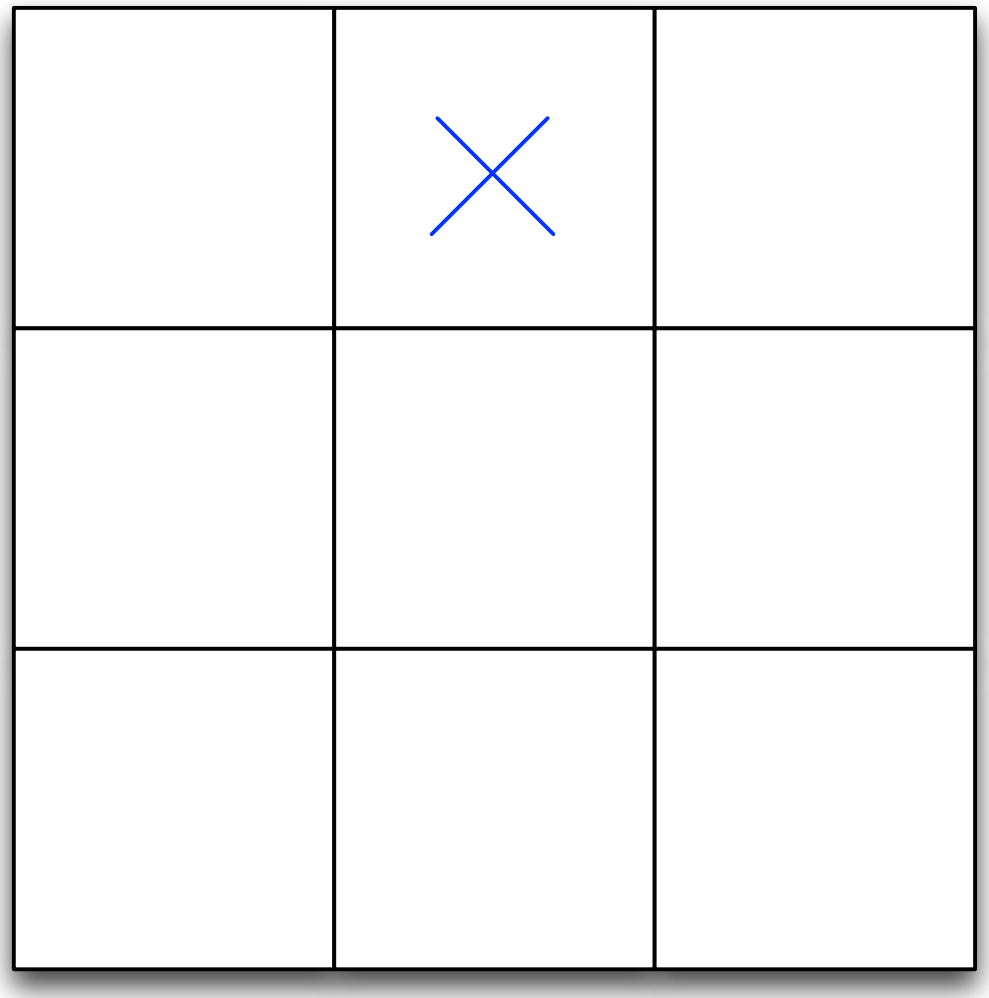
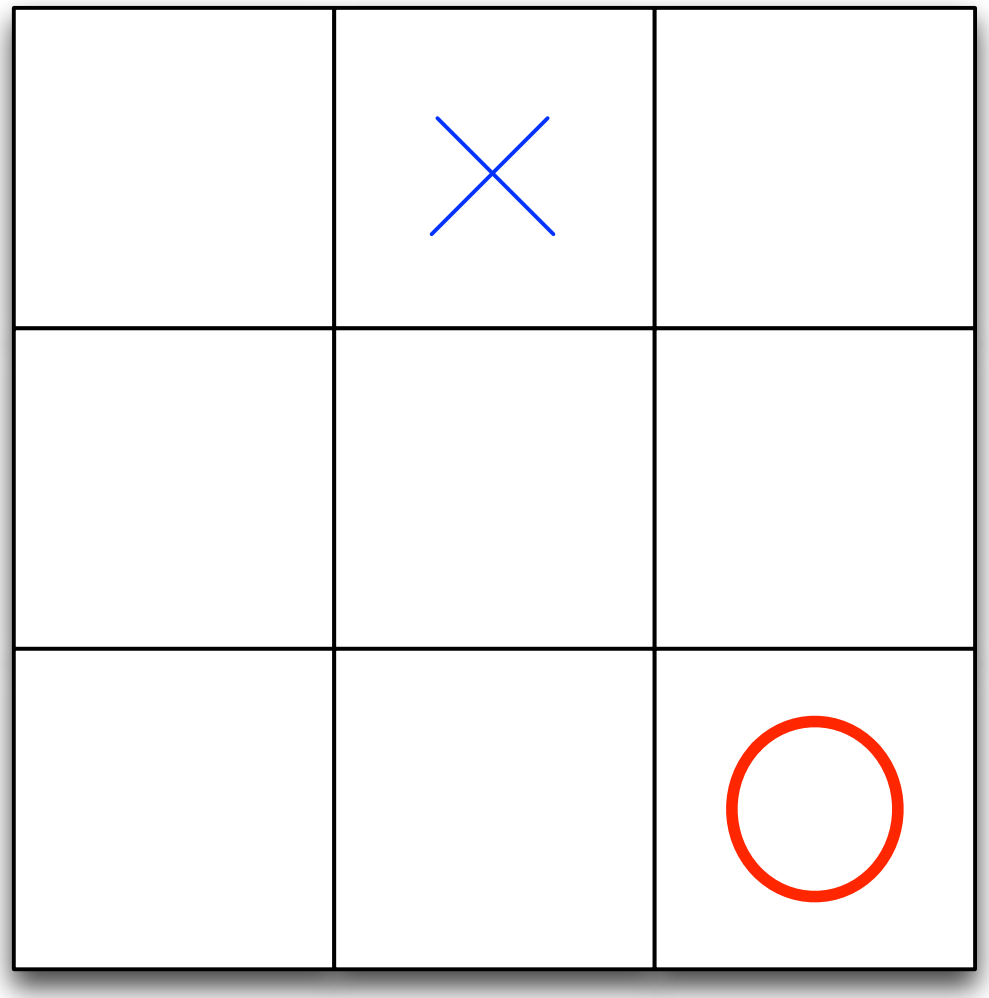
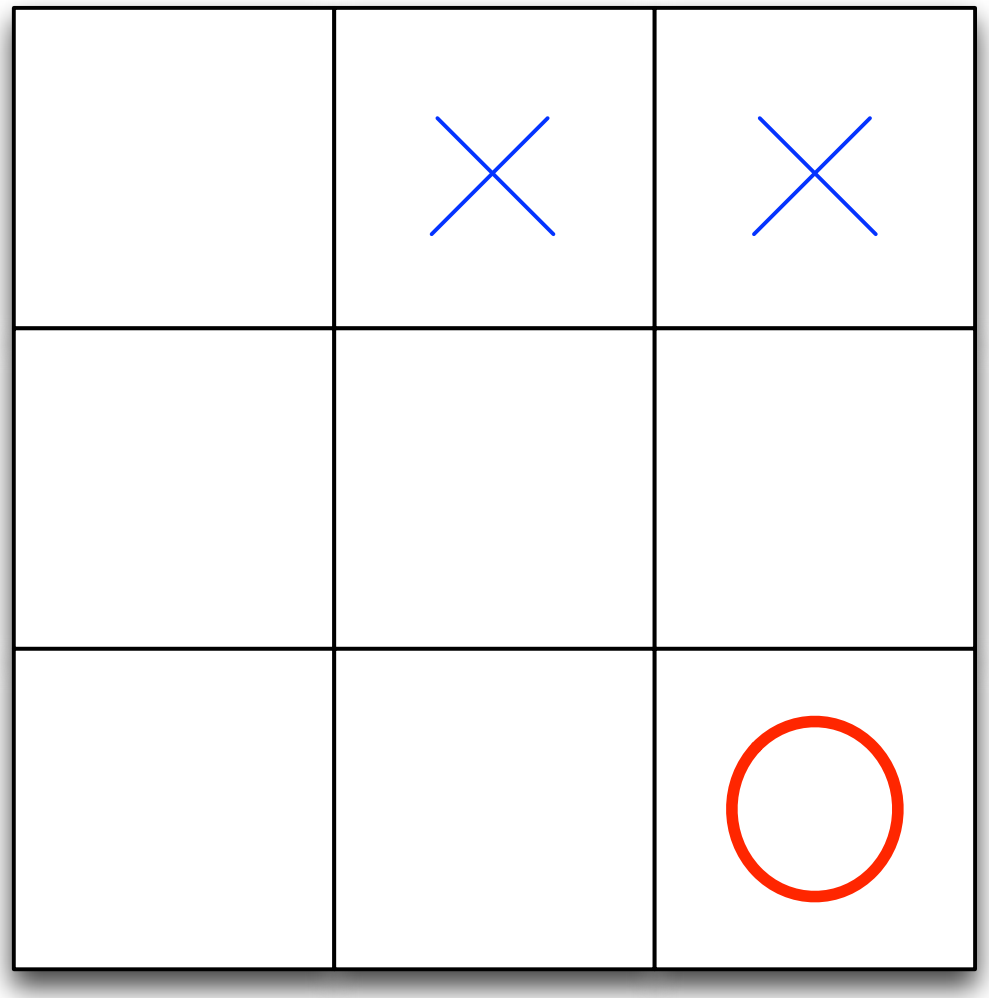
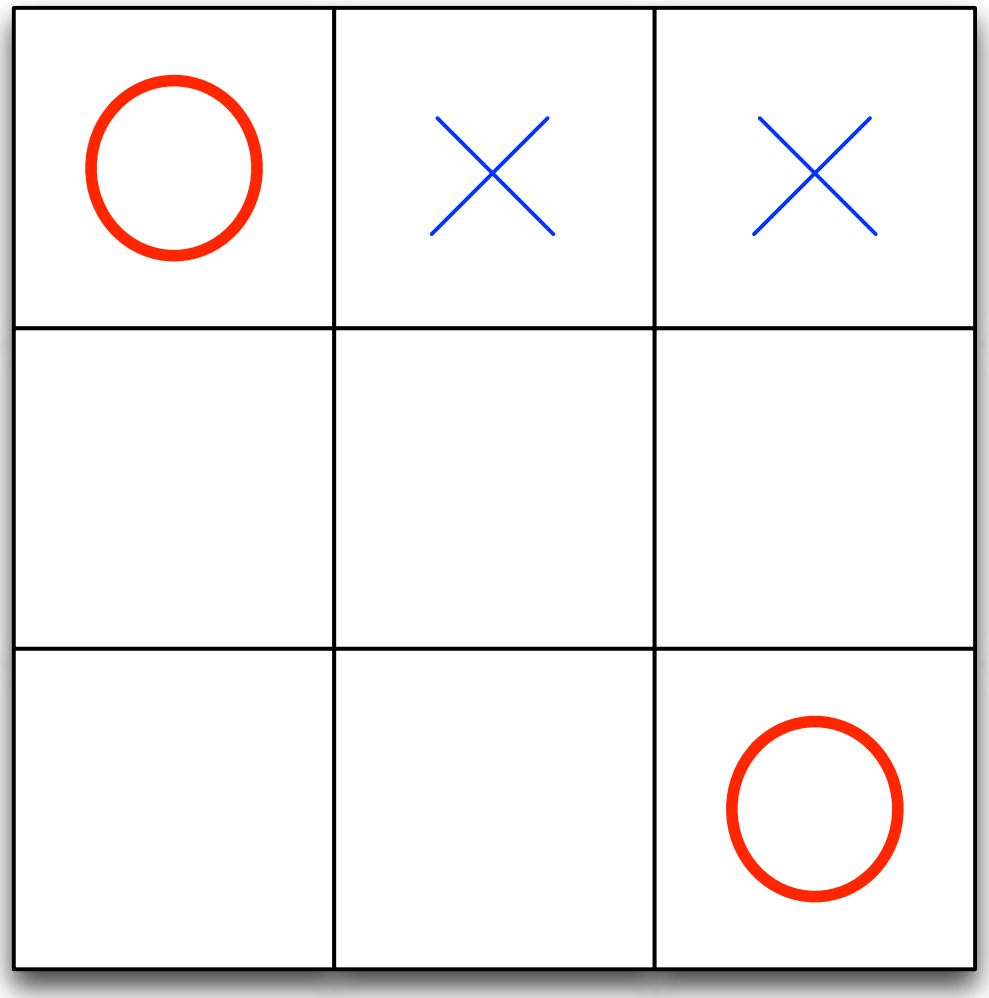
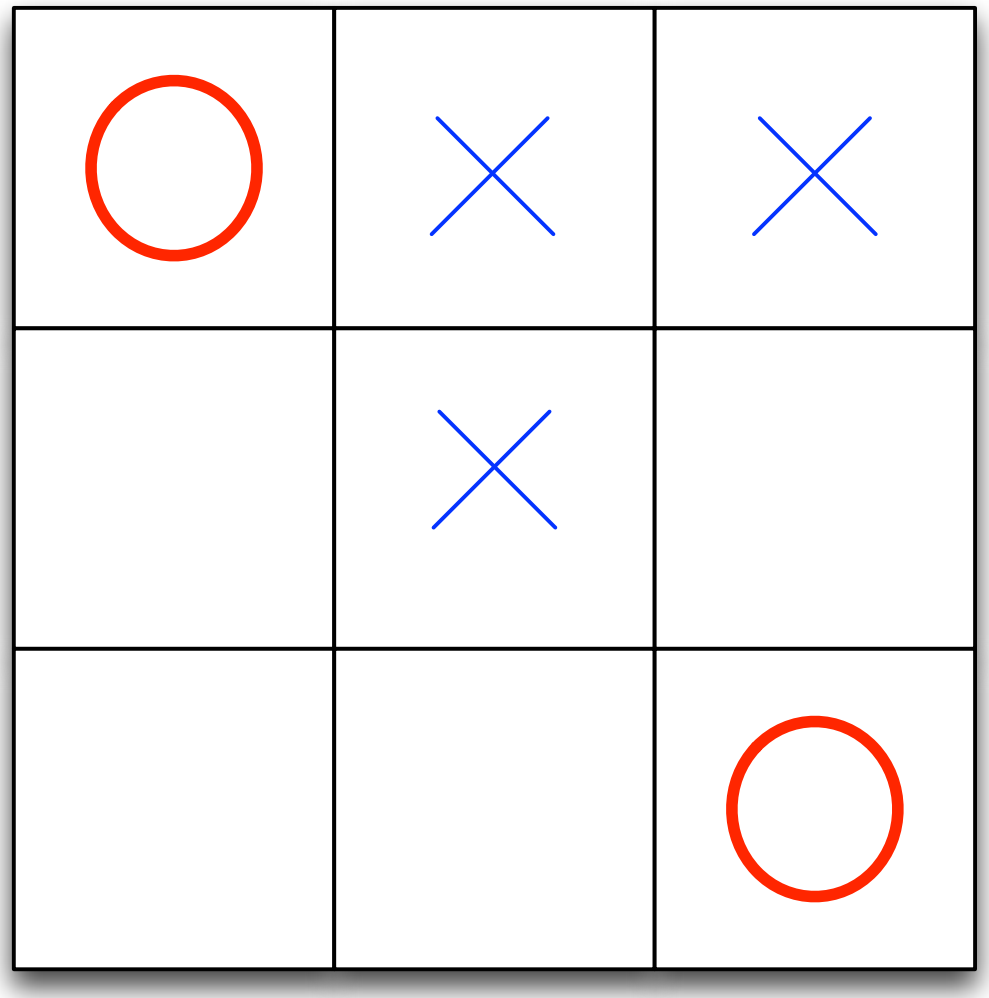
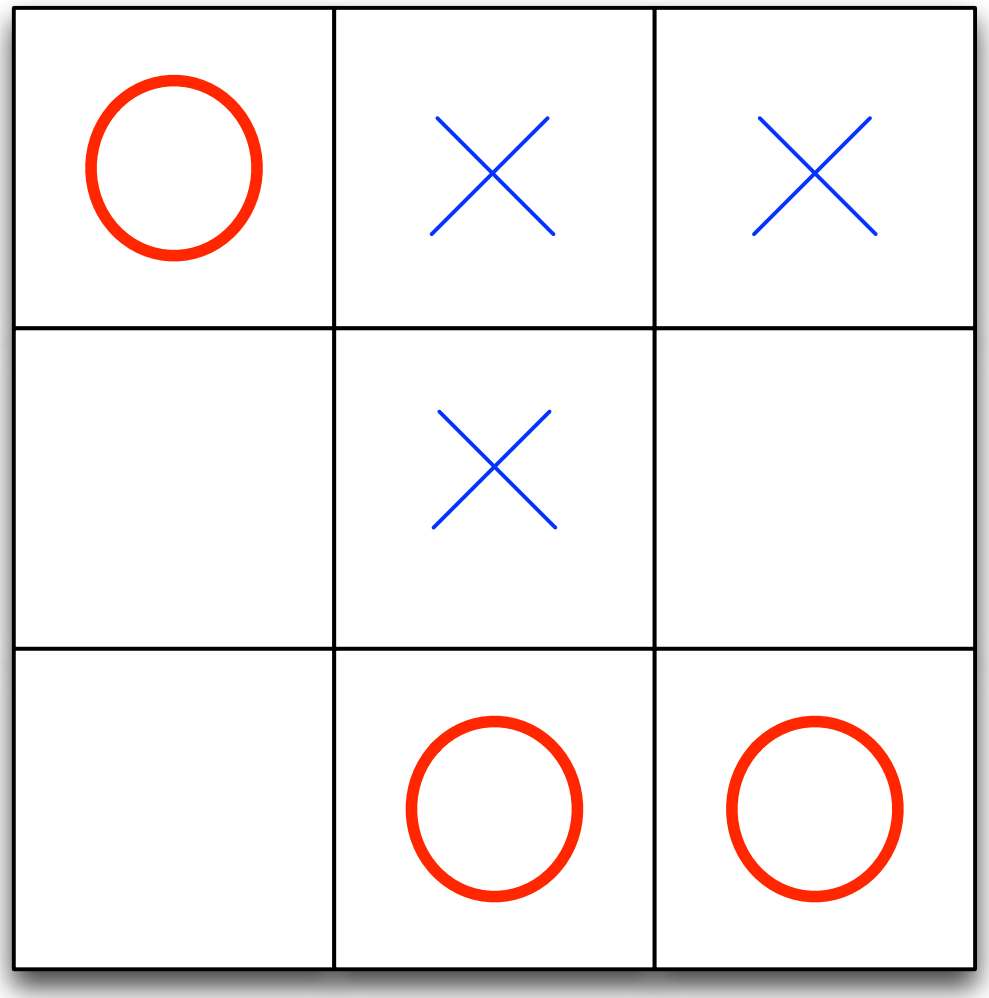
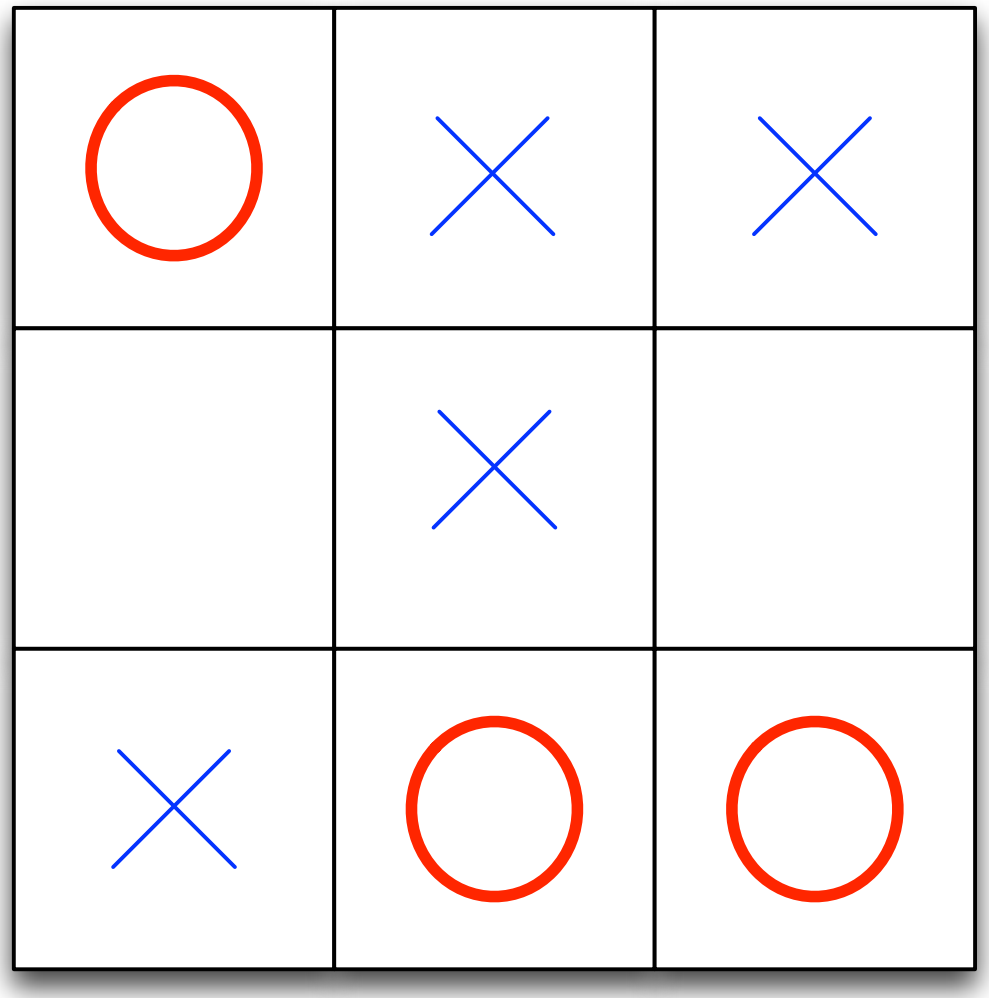
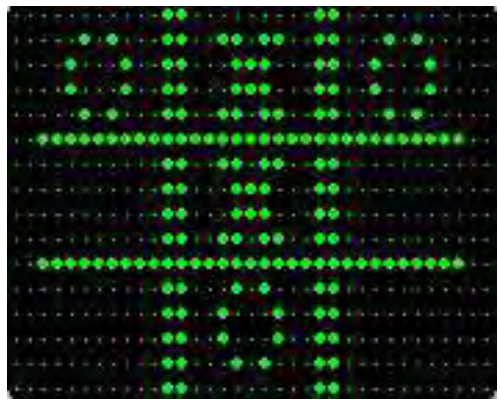
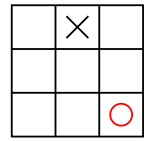
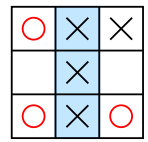
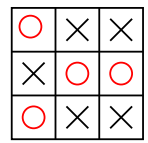
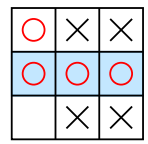
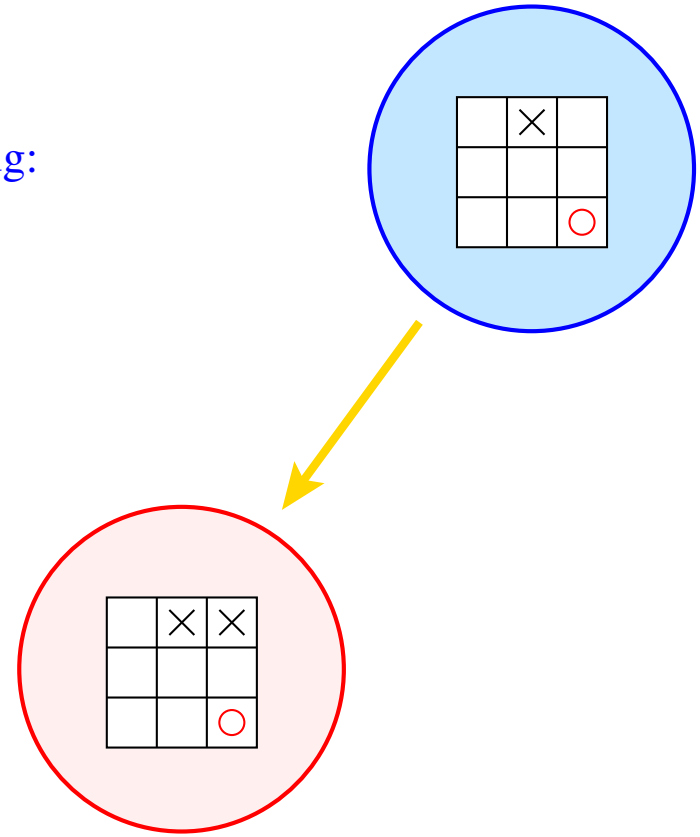
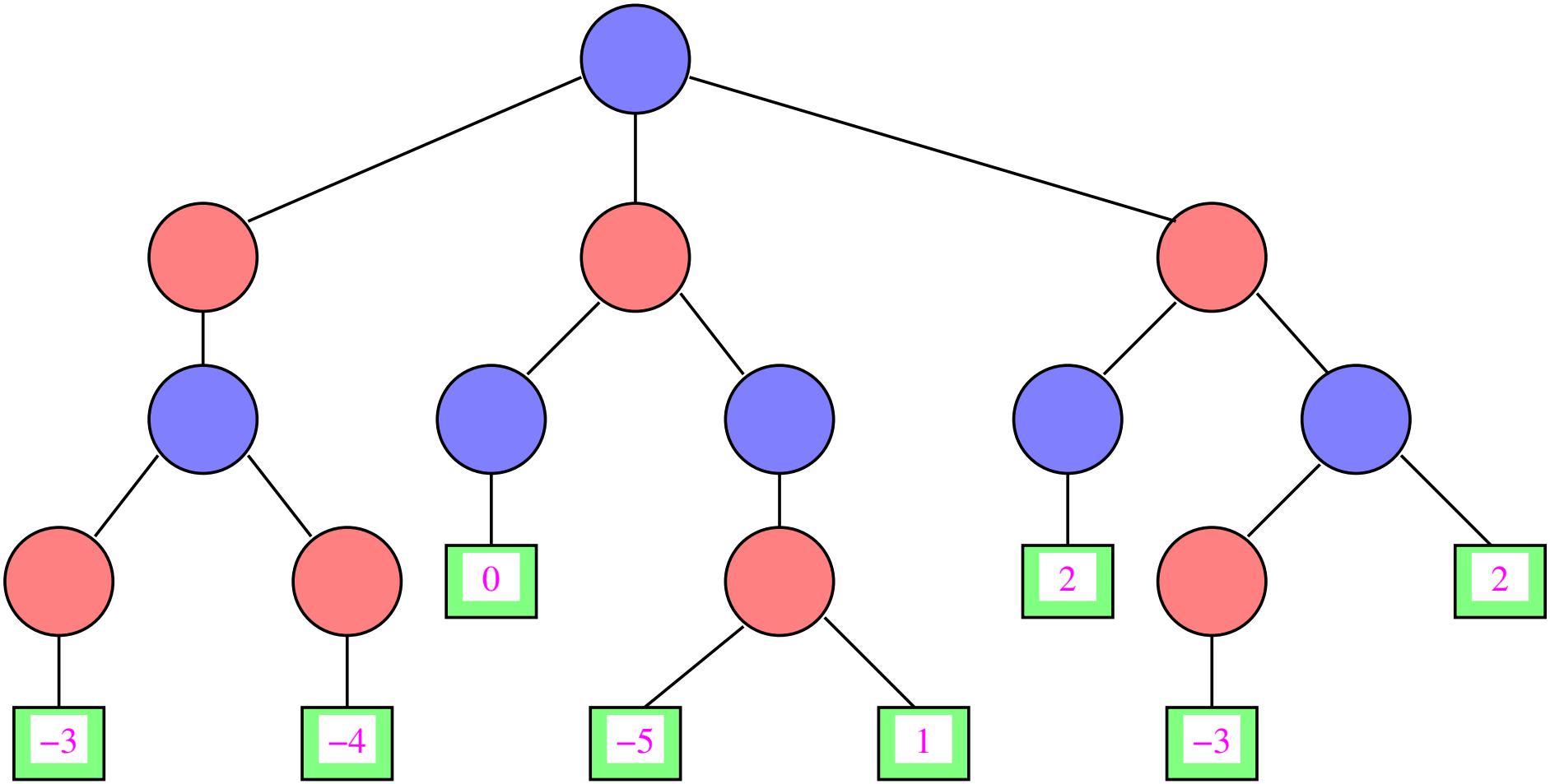
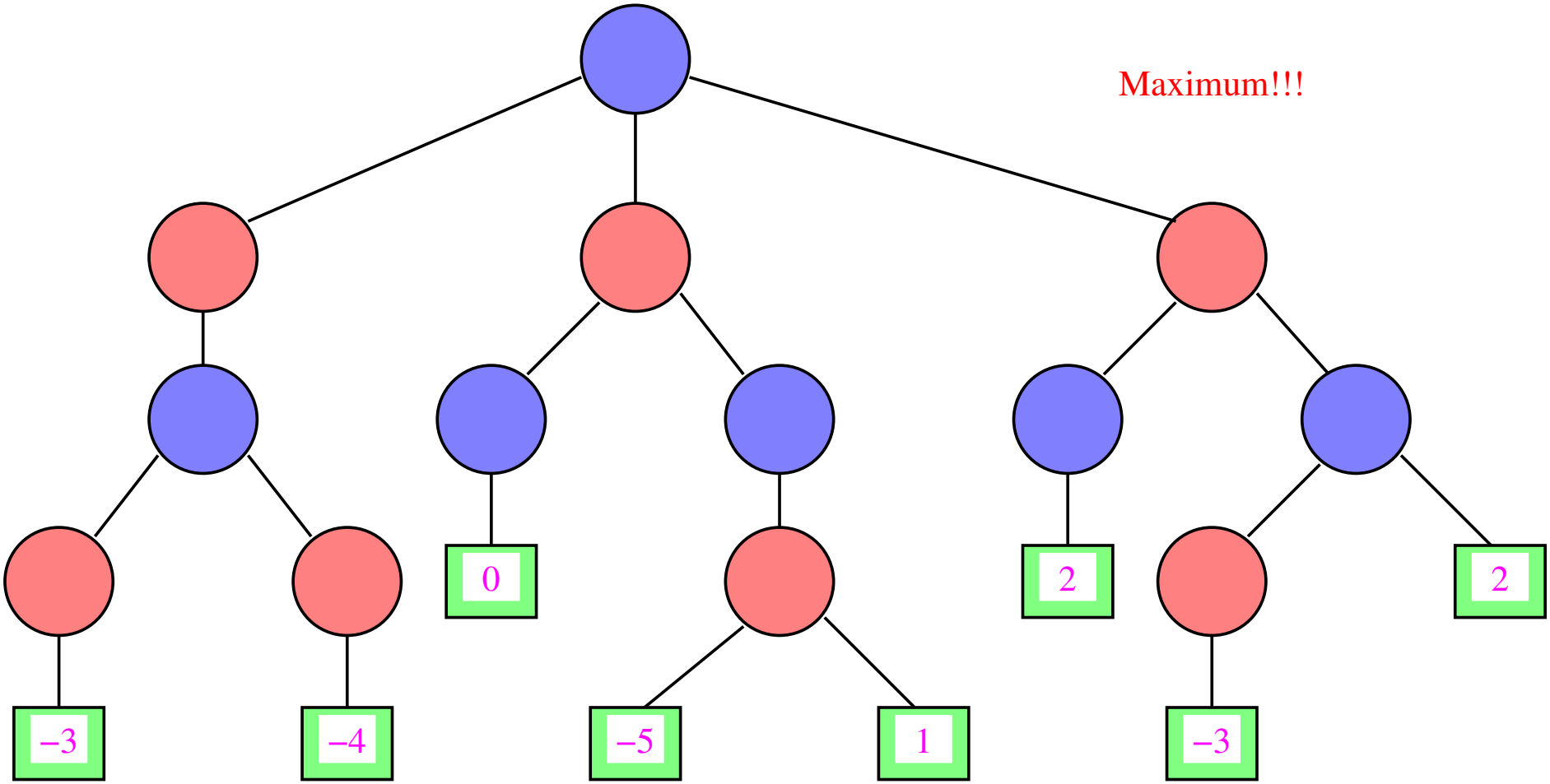
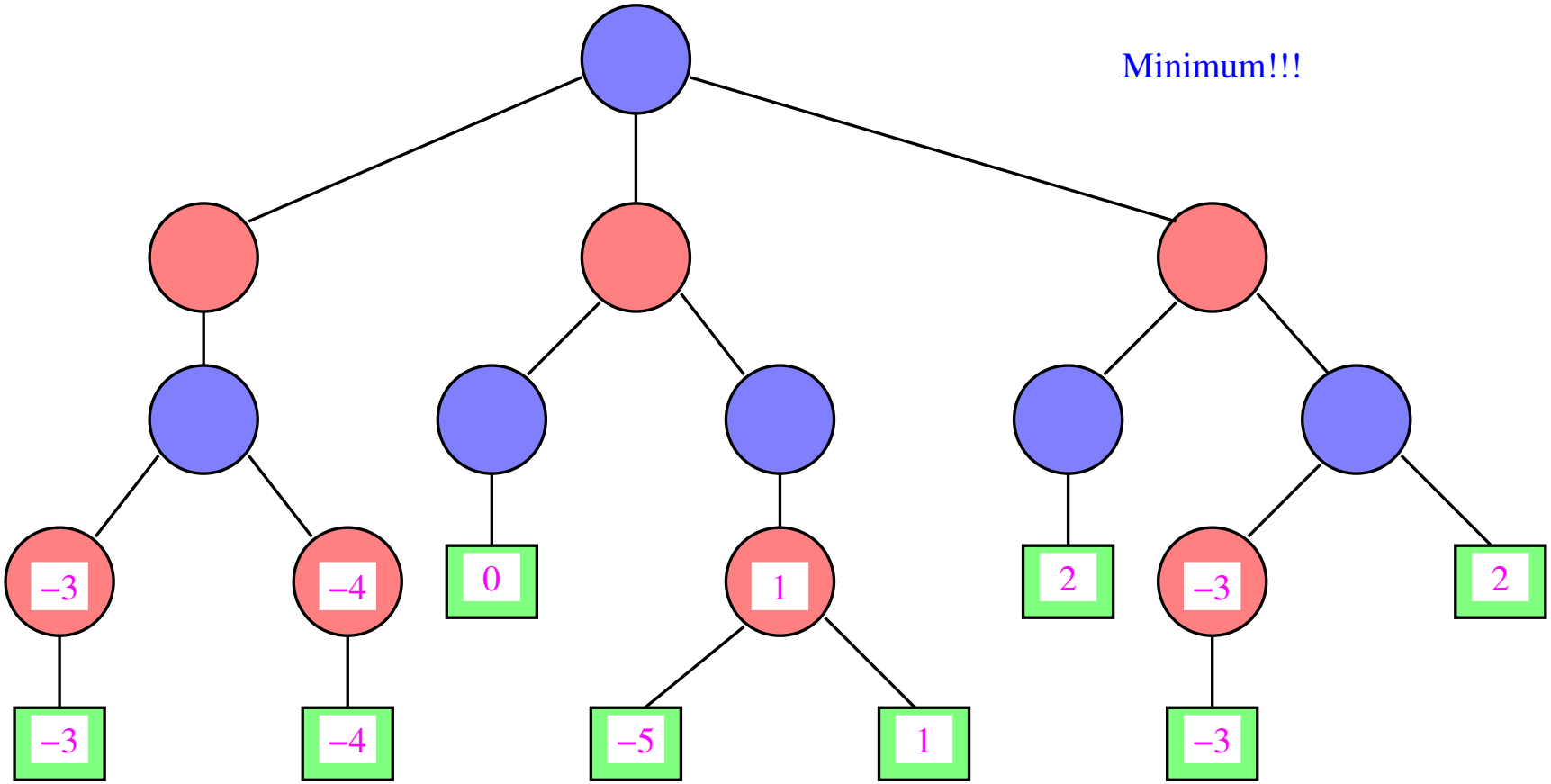
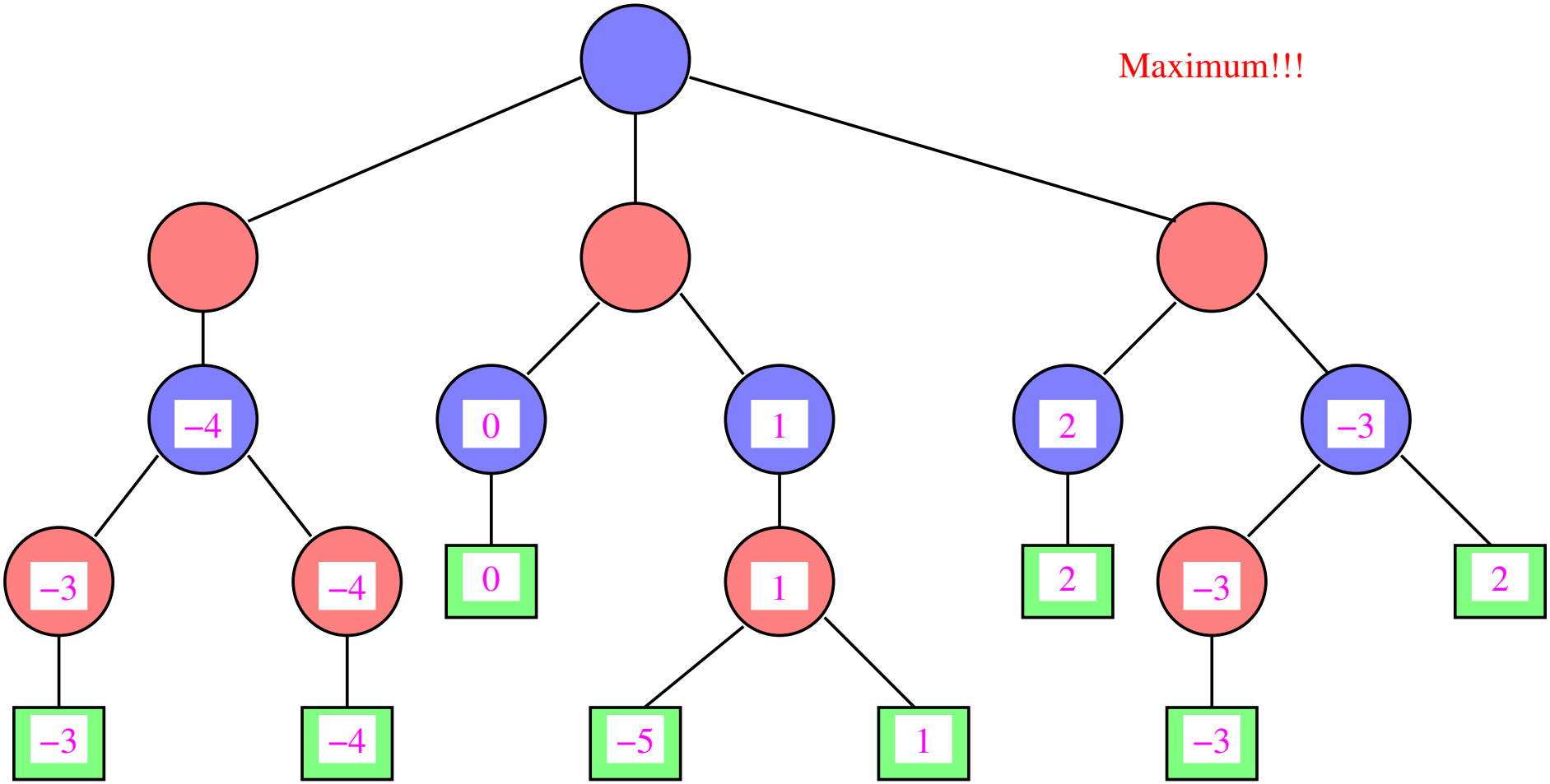
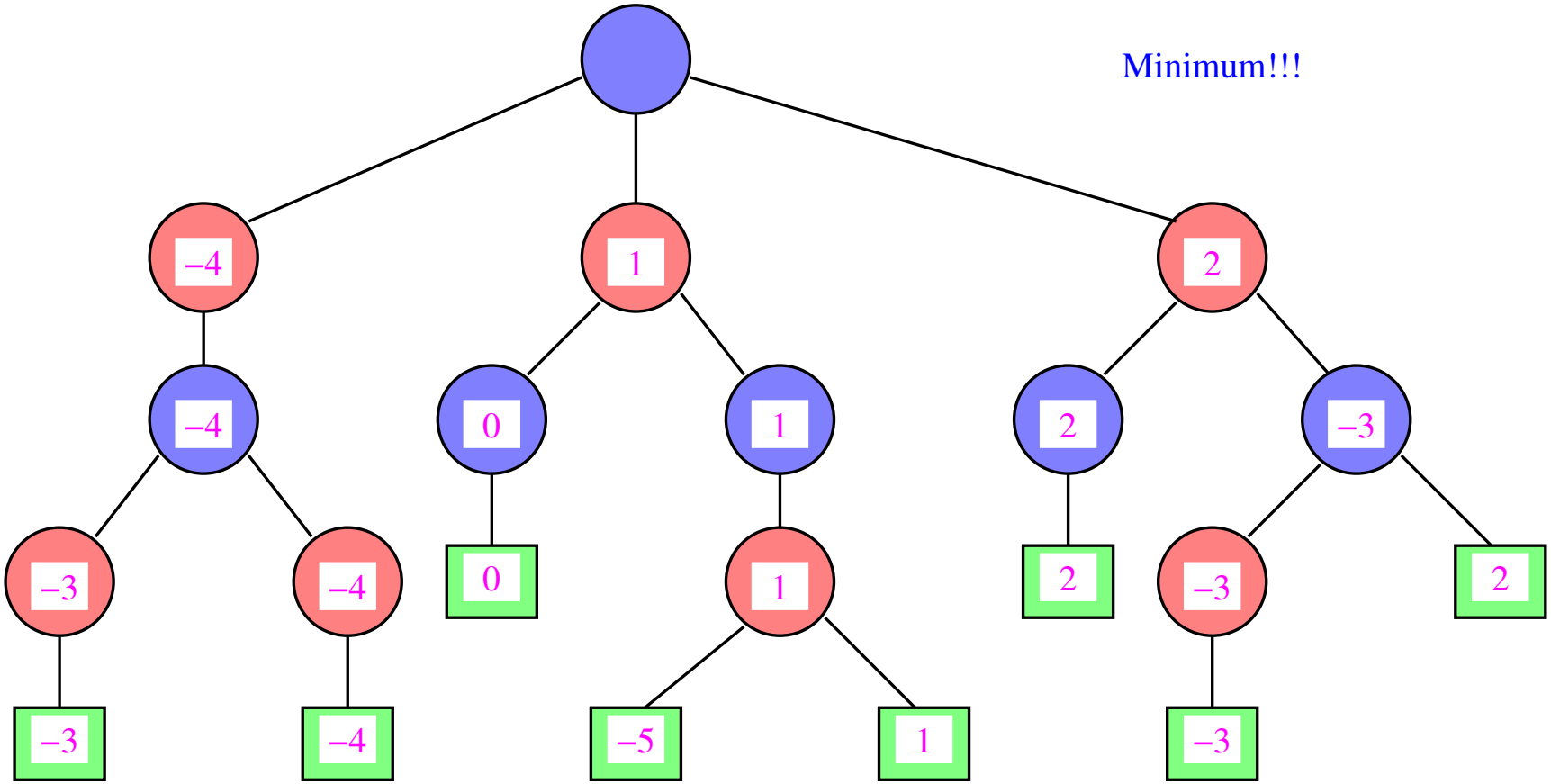
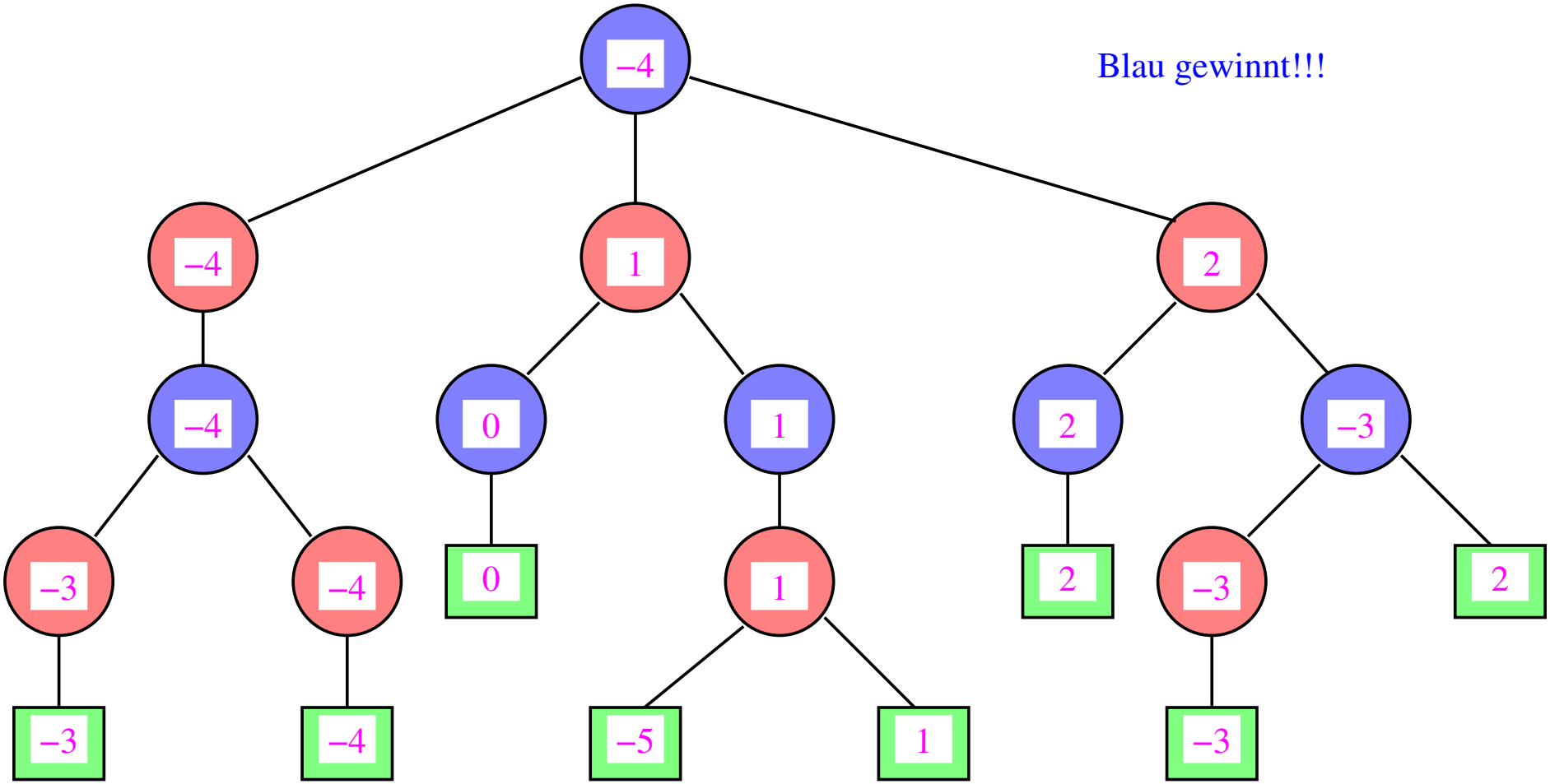
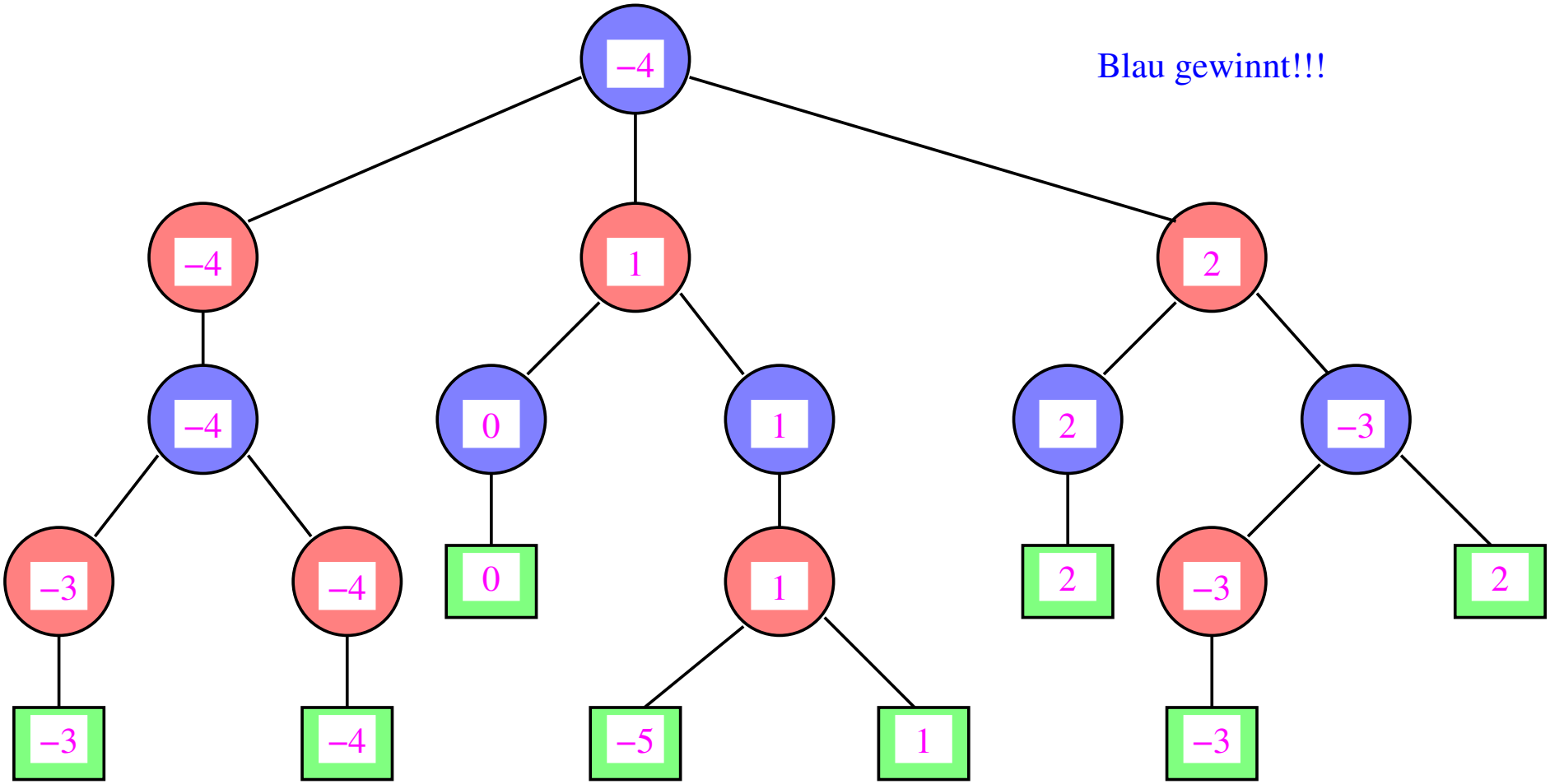
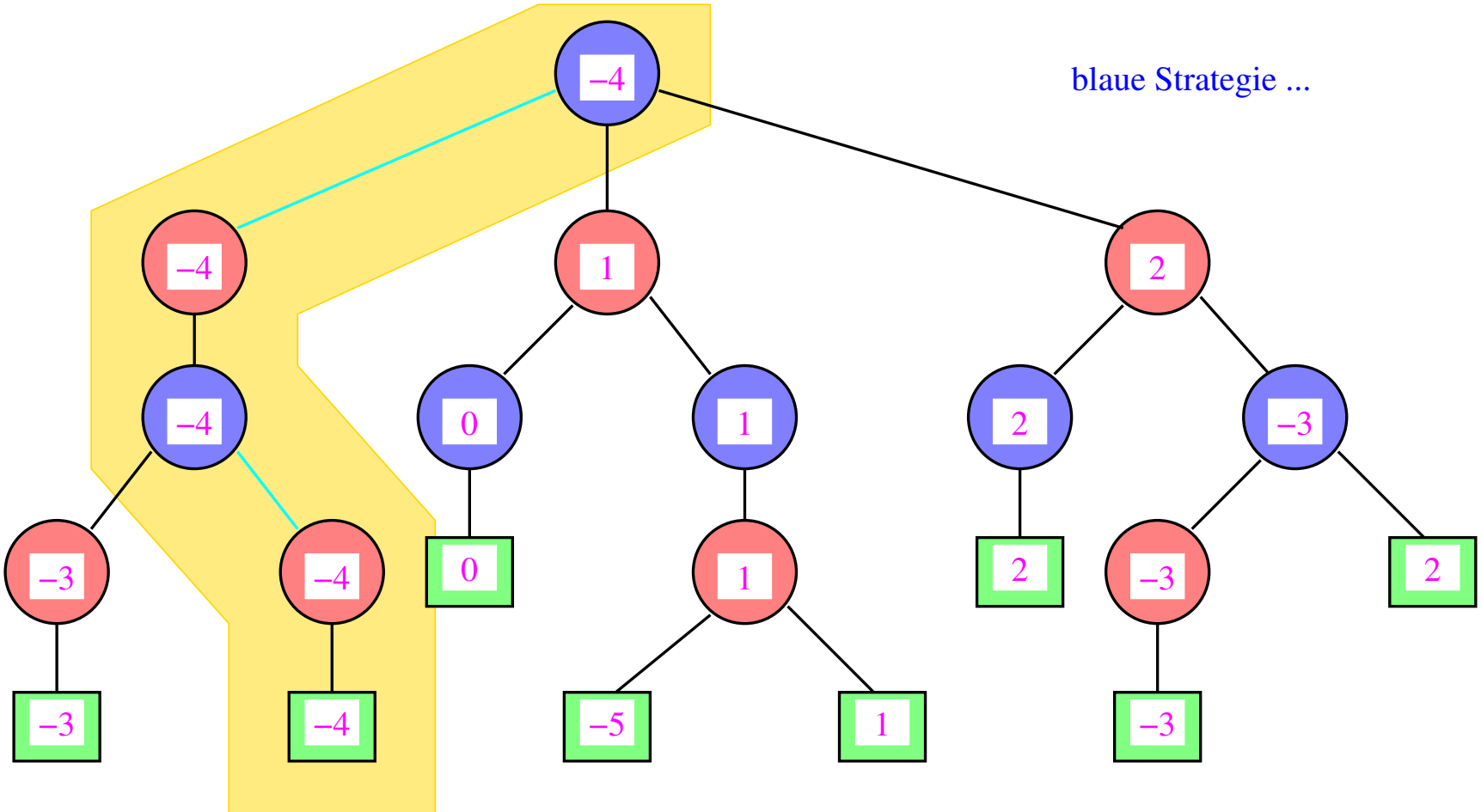
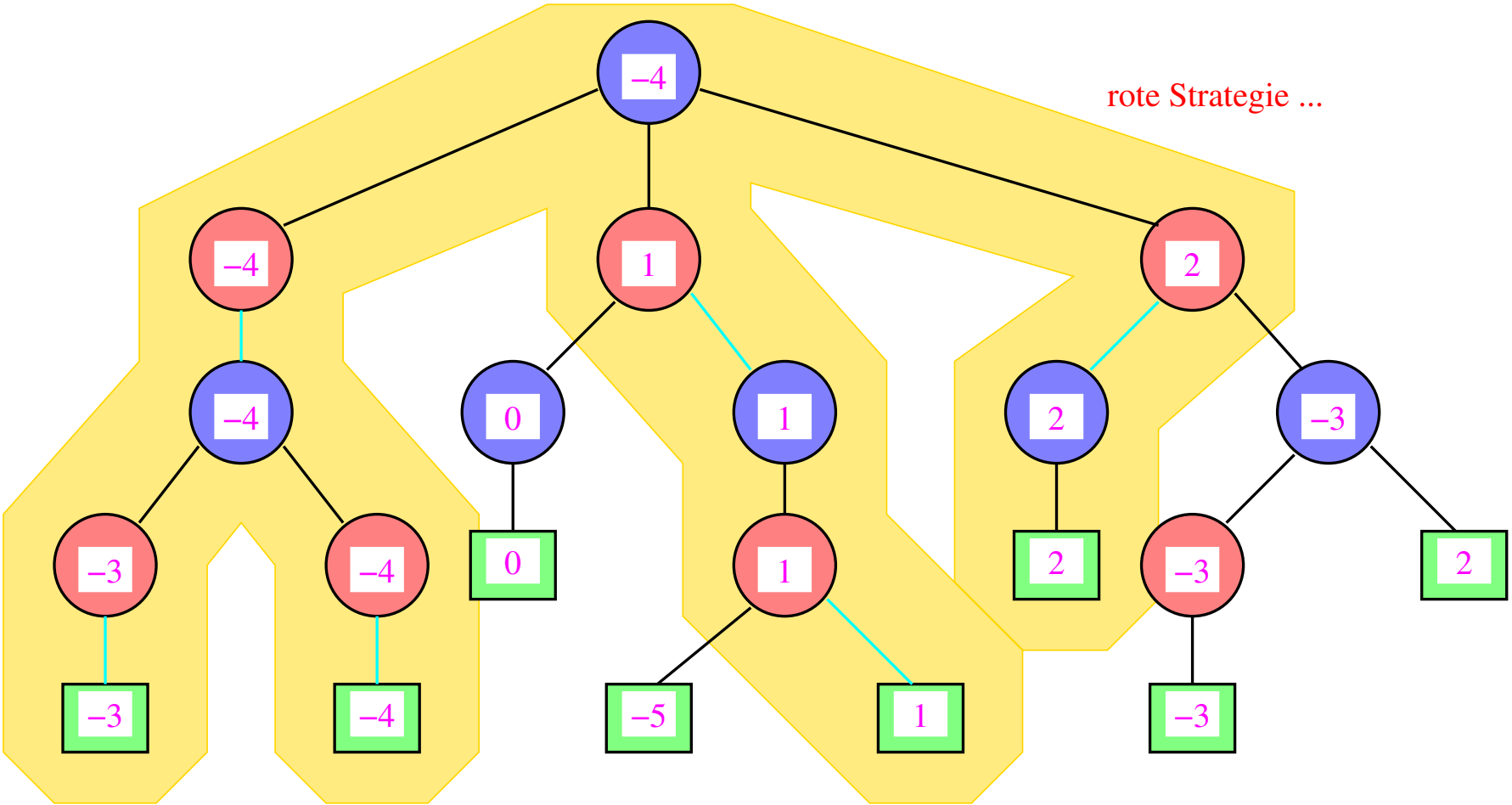
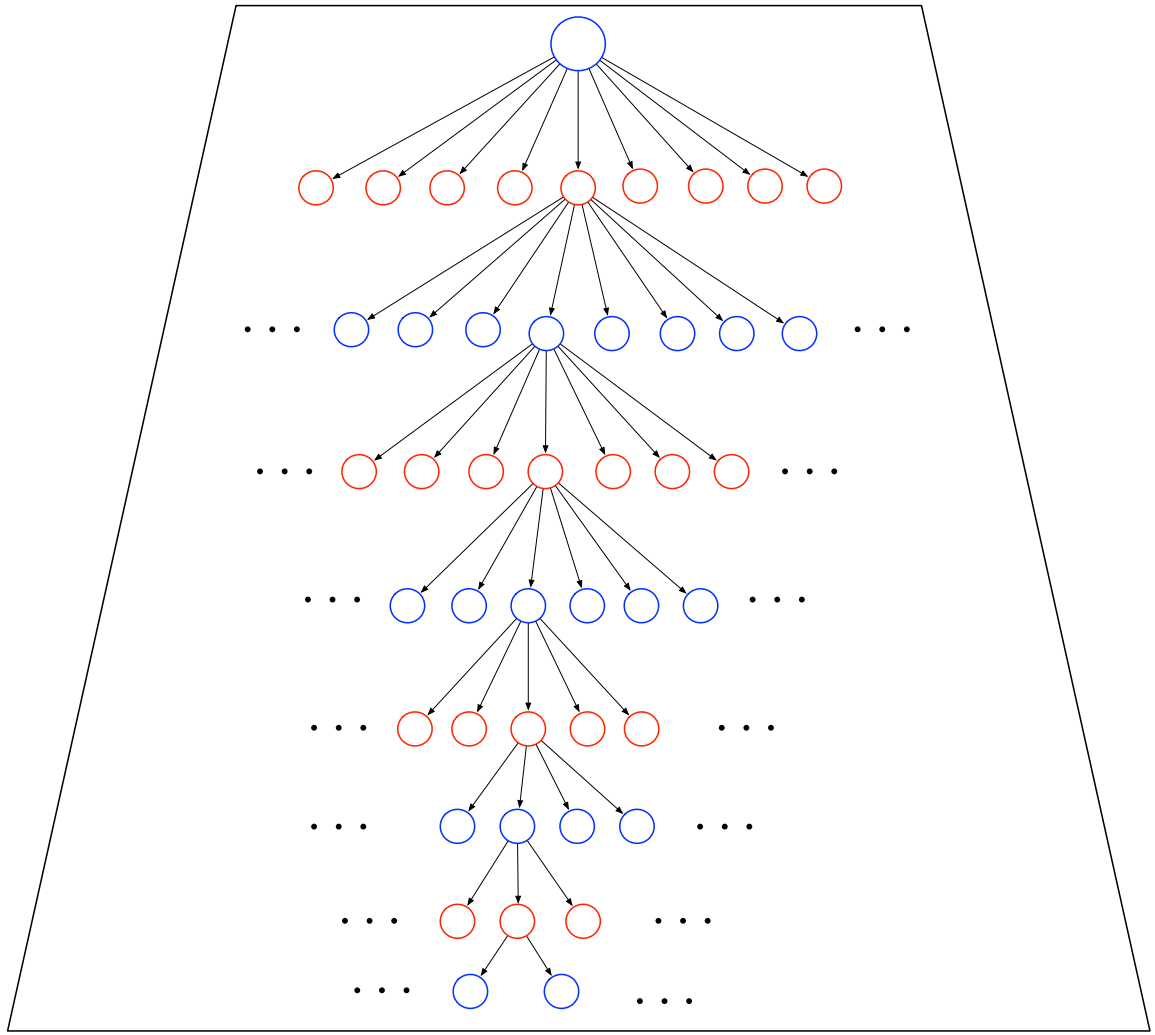
System.out.println(l);                    // [46, 18, 18, 12]
```

- Erzeugt eine anonyme Klasse, die `Comparator<Integer>` implementiert und instanziert diese Klasse einmalig.

Alles zusammen I: Beispiel Fußball

Alles zusammen I: Beispiel Fußball



Alles zusammen I: Beispiel Fußball





Der Typ Enum (I)

Der Typ Enum ist ein Aufzählungstyp mit einem endlichen Wertebereich:

Beispiel:

```
enum Position {
    TOR, ABWEHR, MITTELFELD, STURM, RECHTSAUSSEN
}

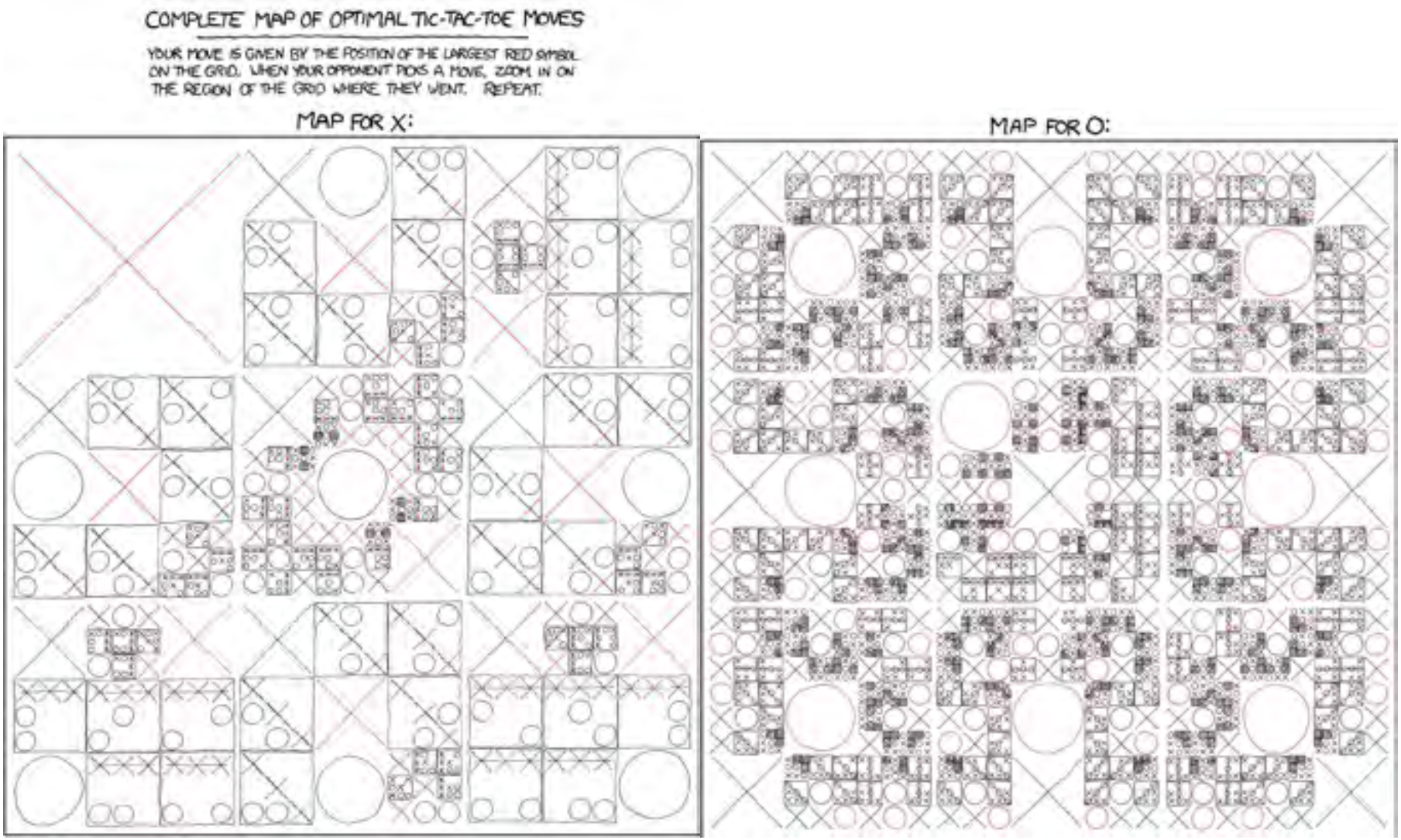
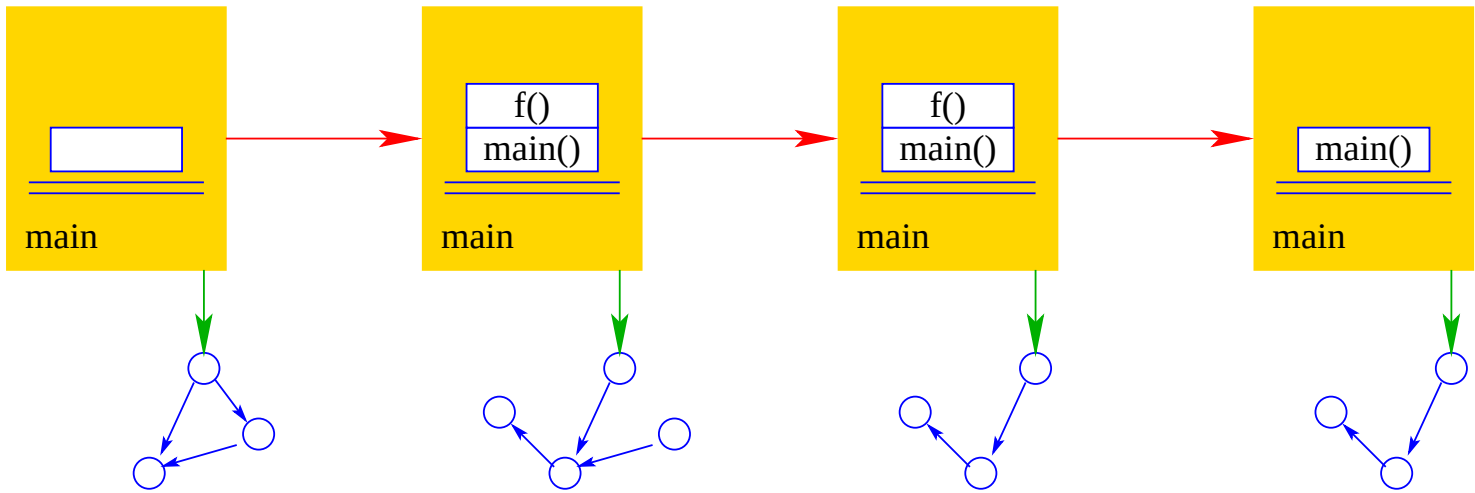
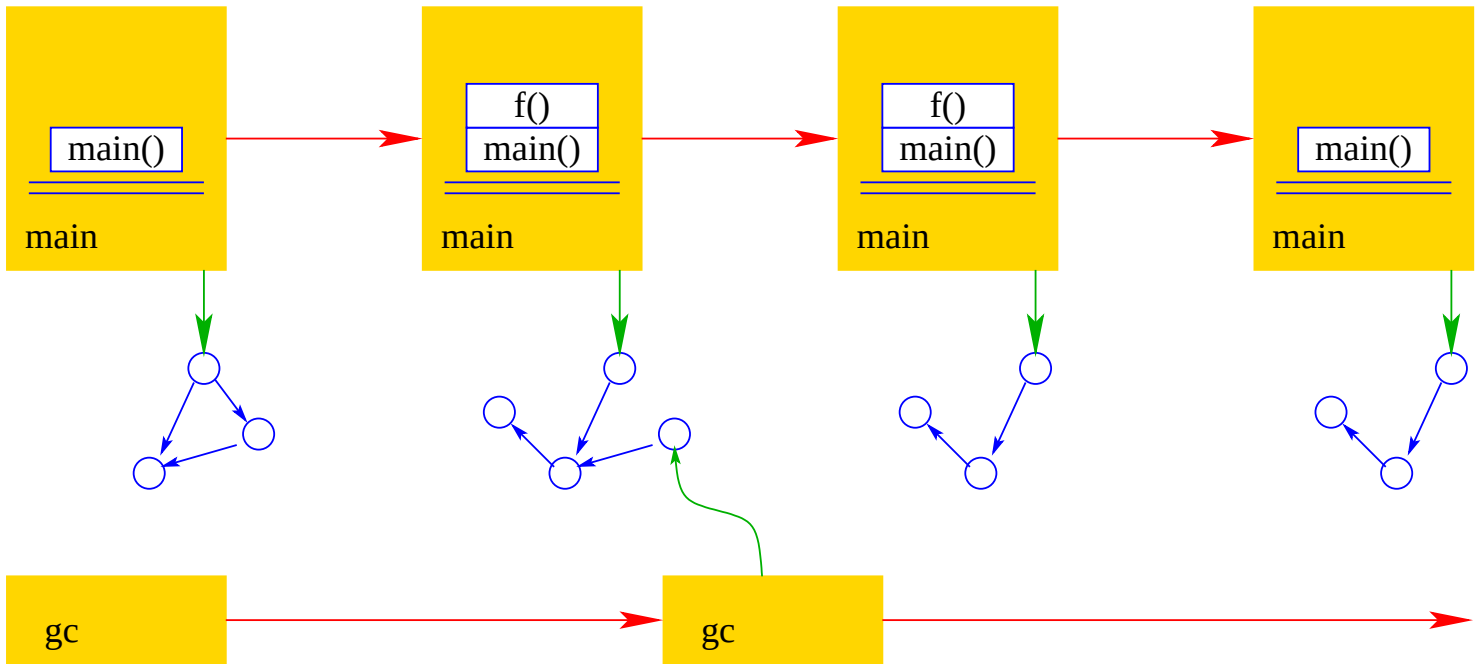
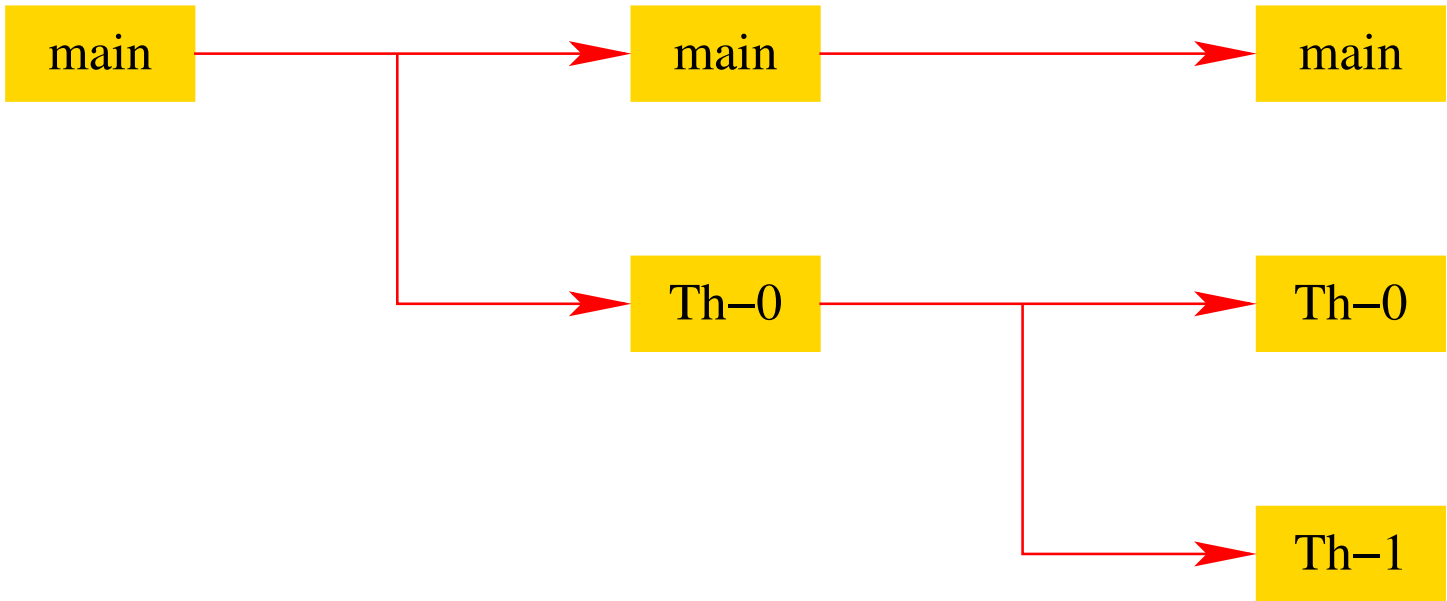
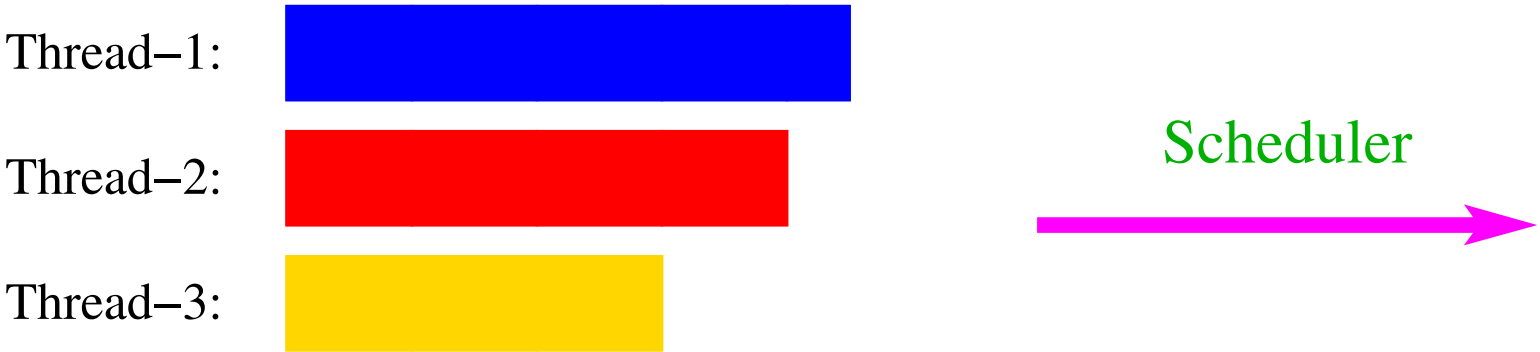
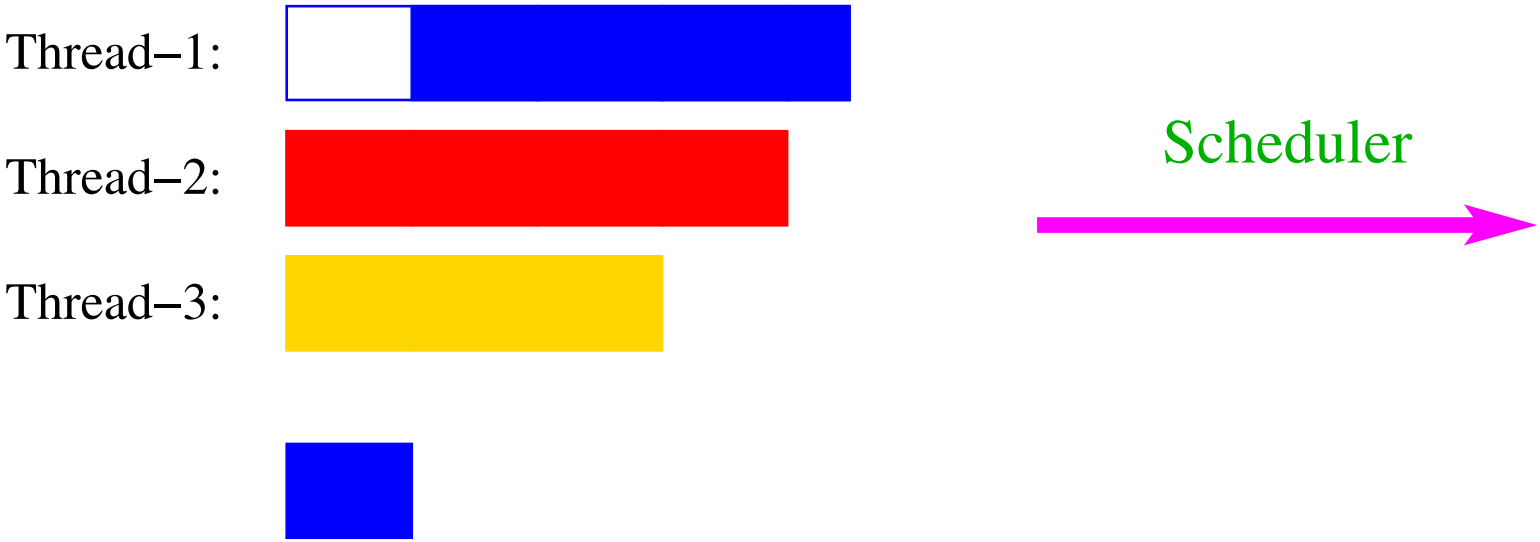
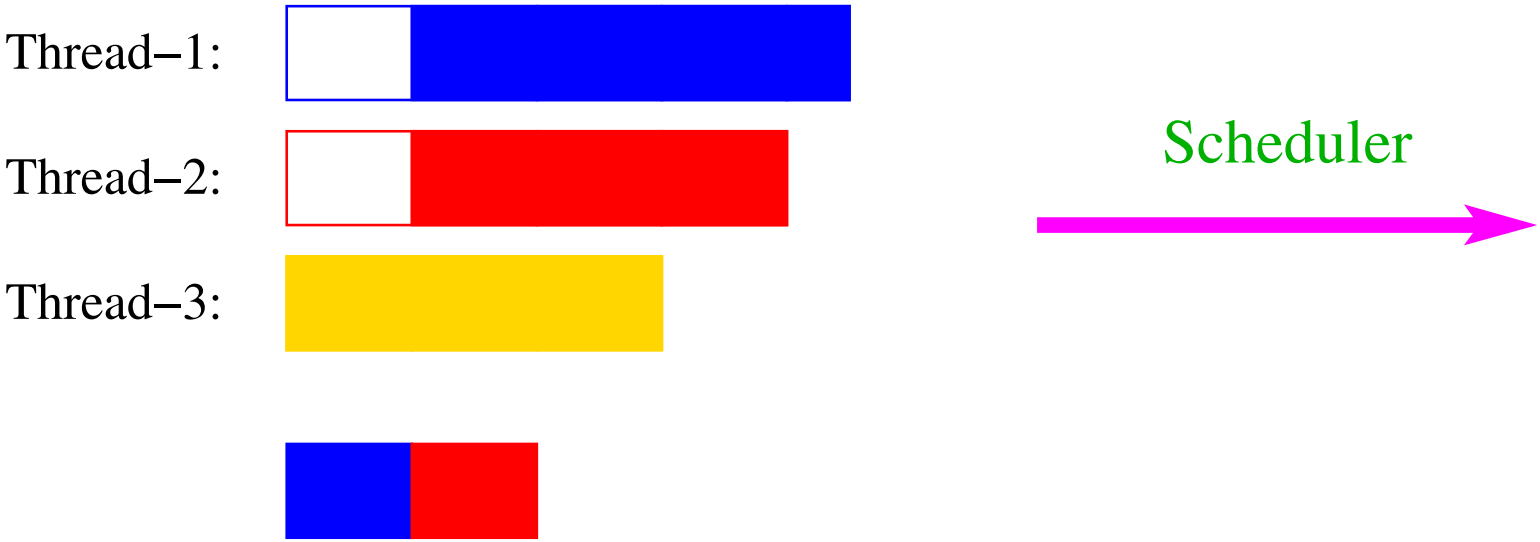
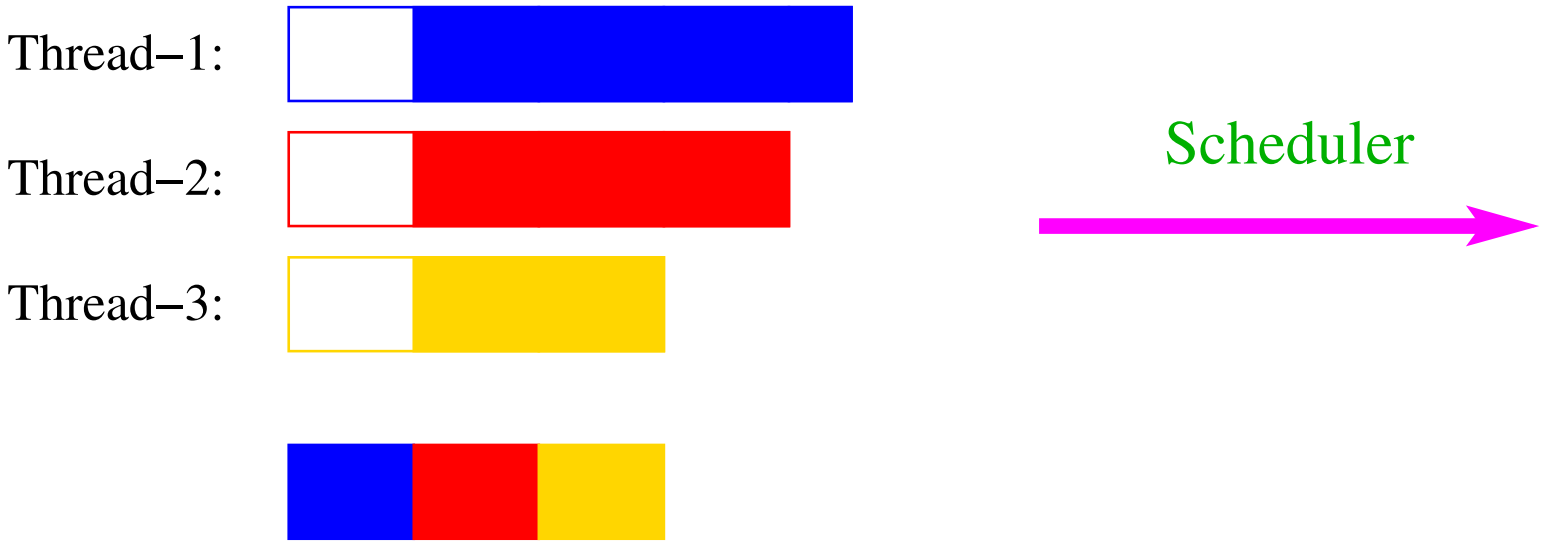
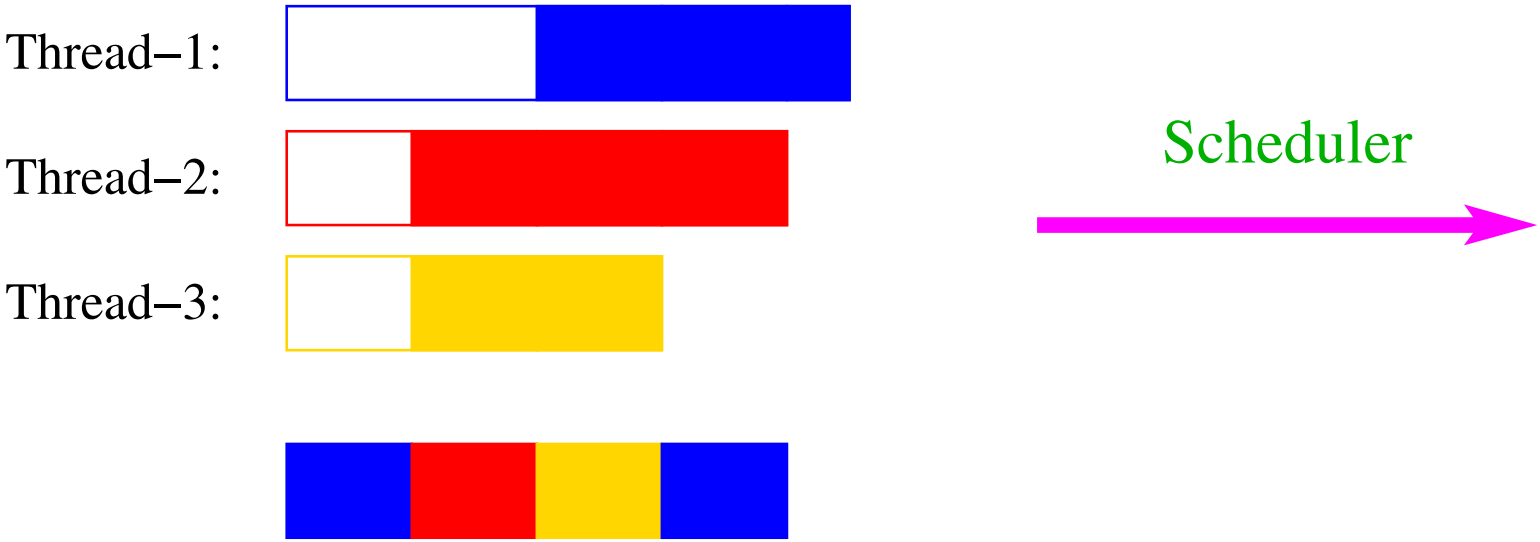
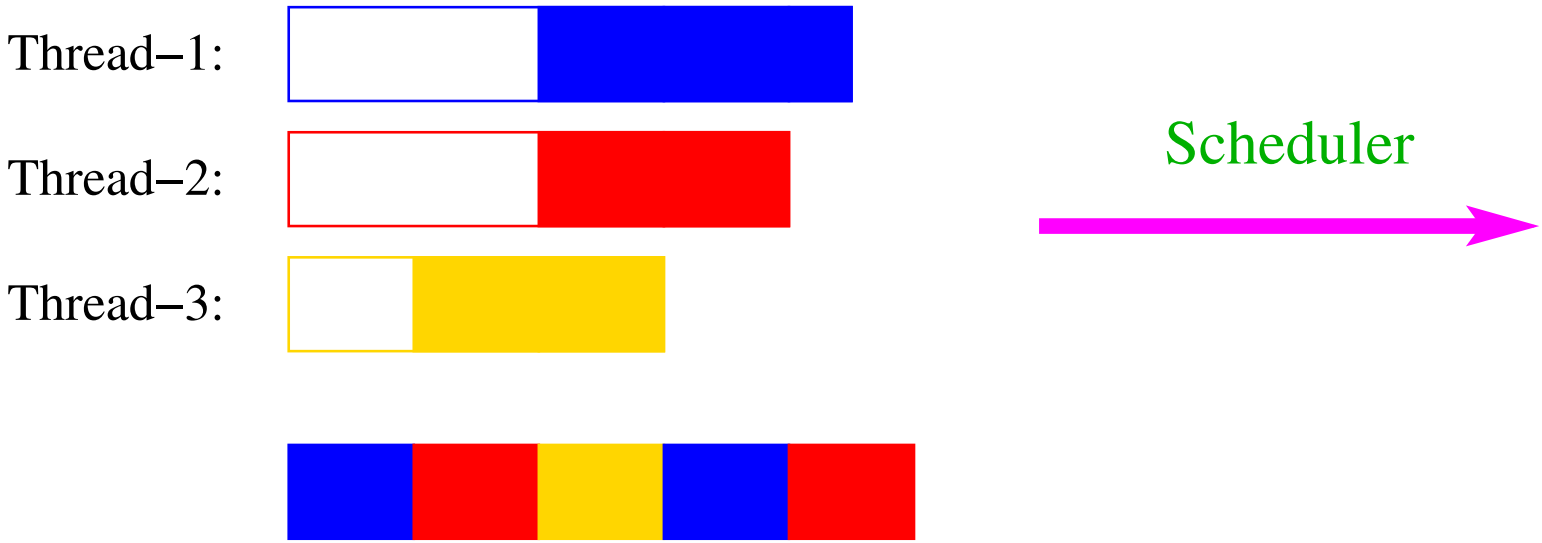
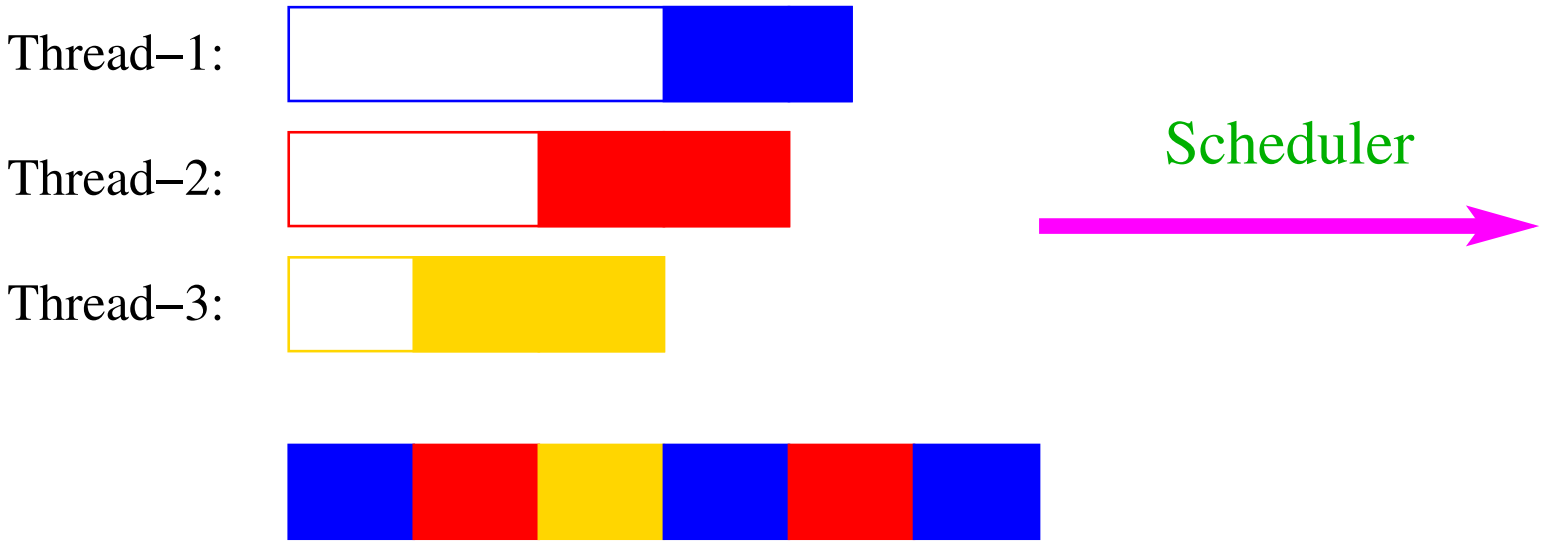
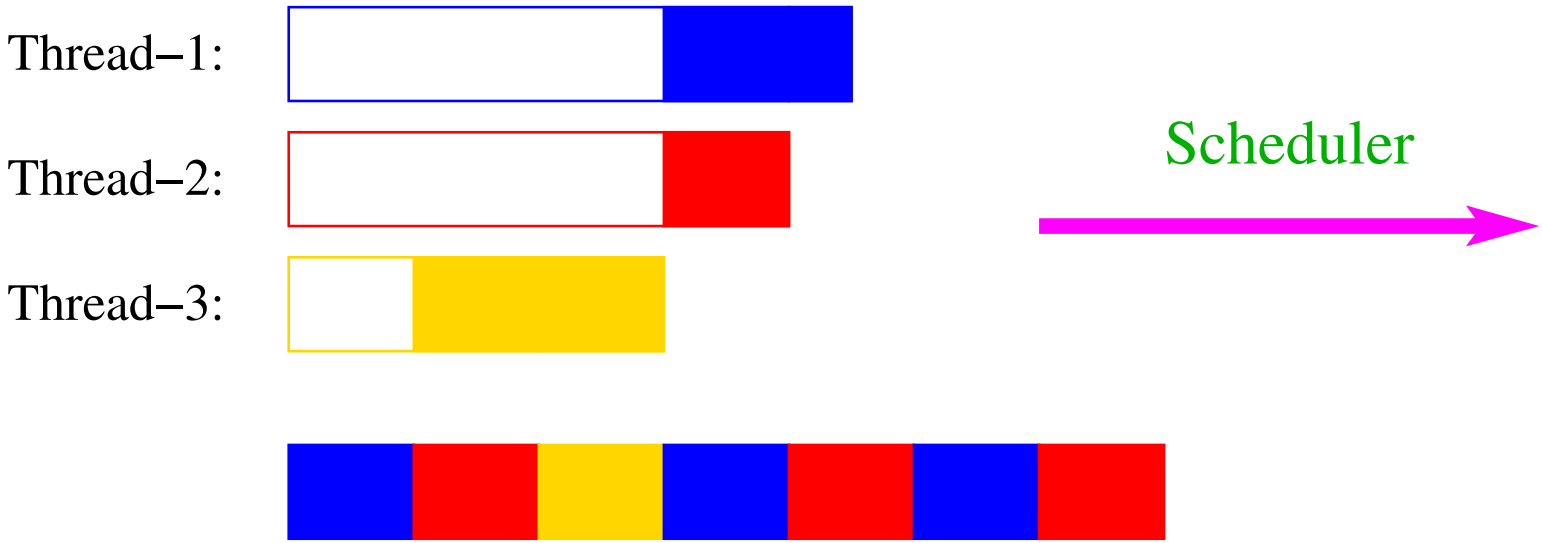
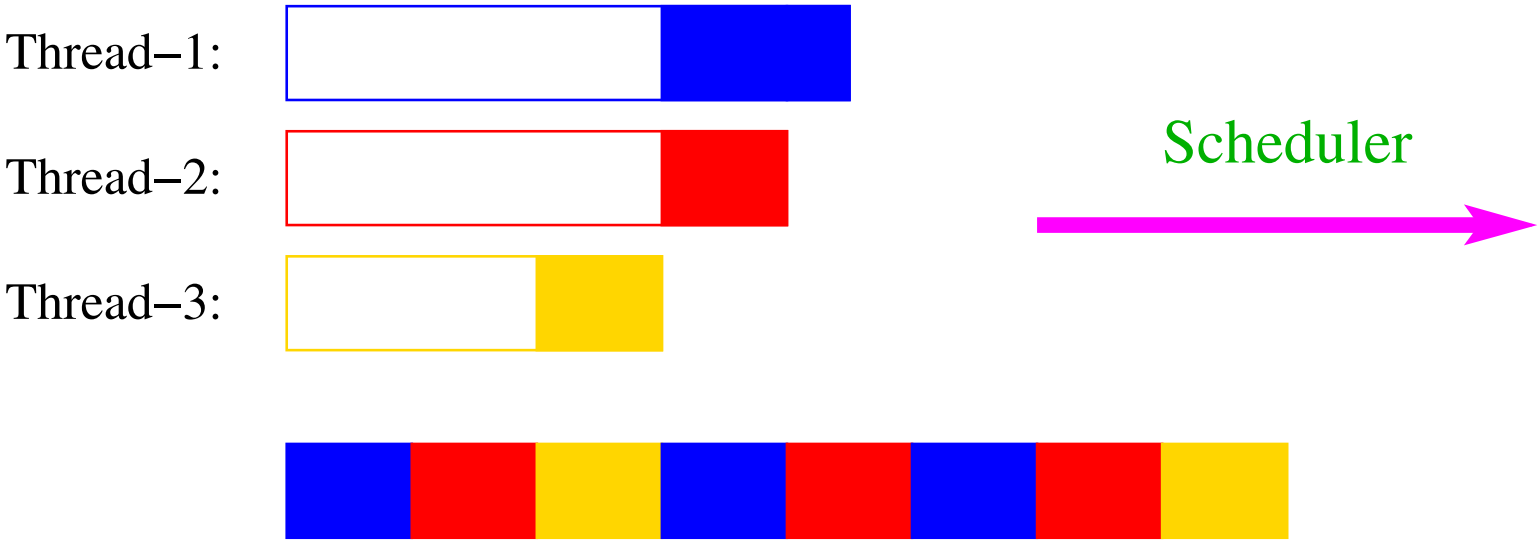
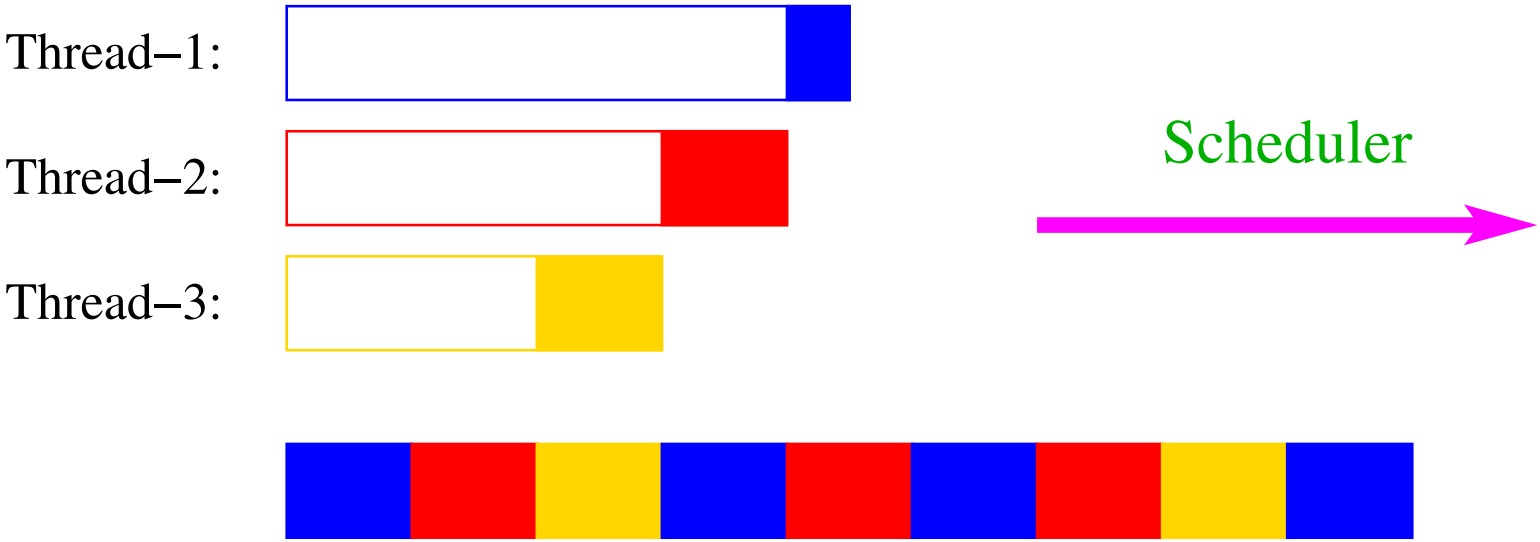
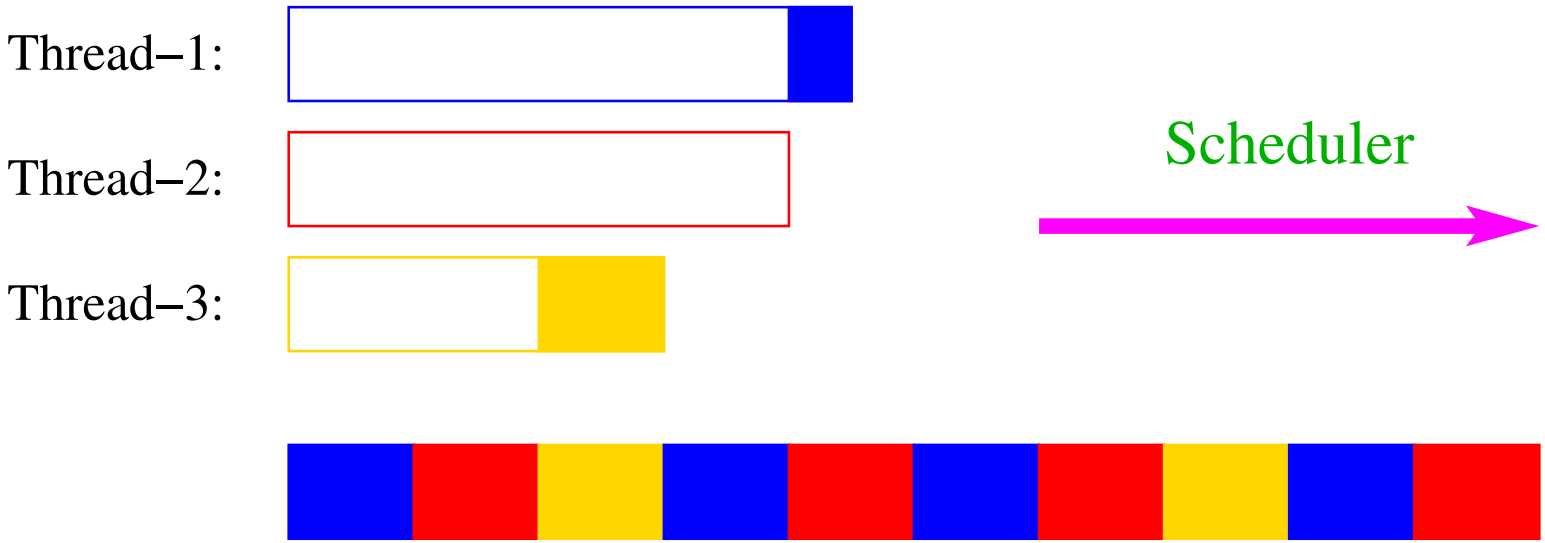
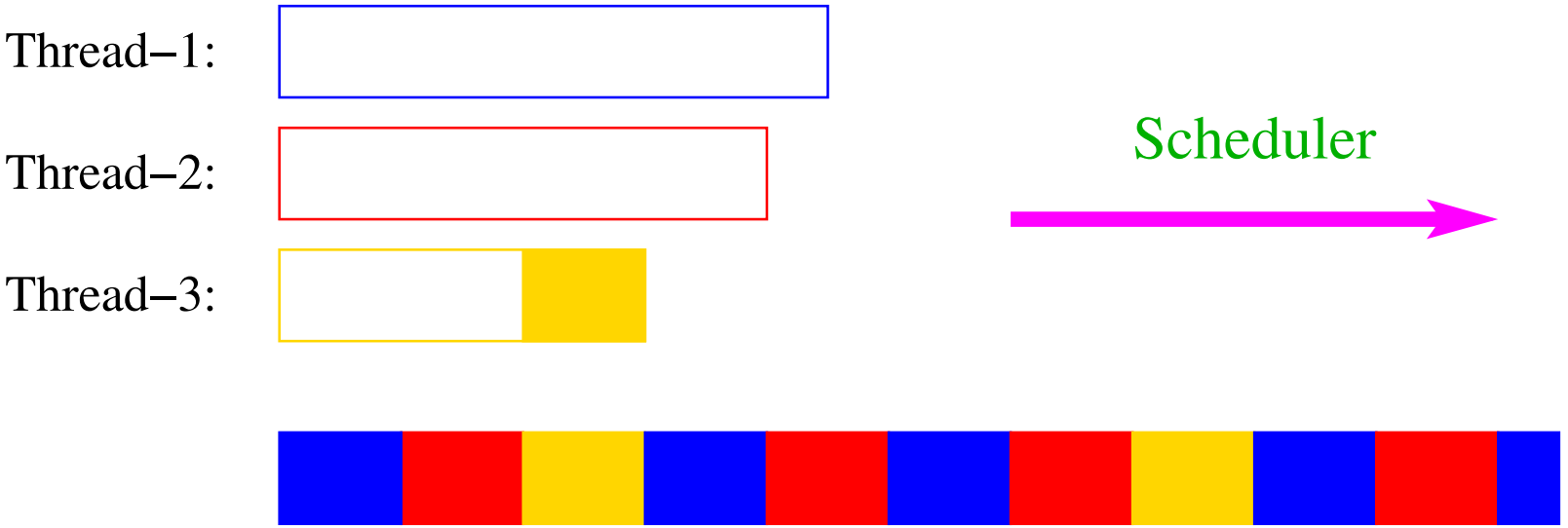
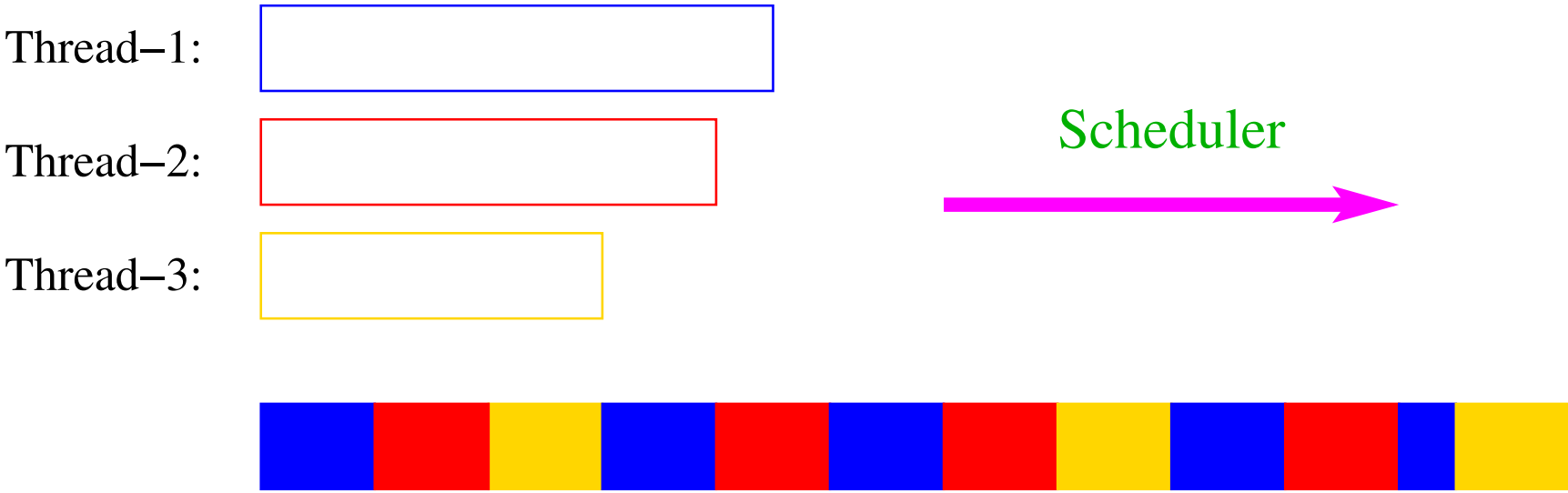
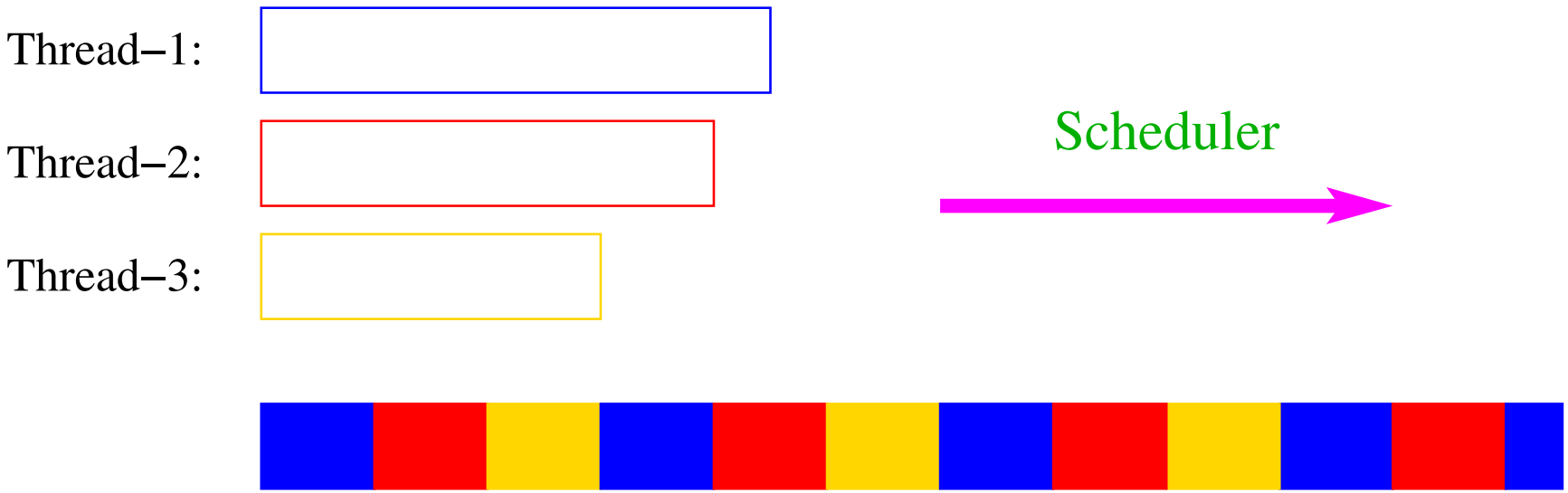
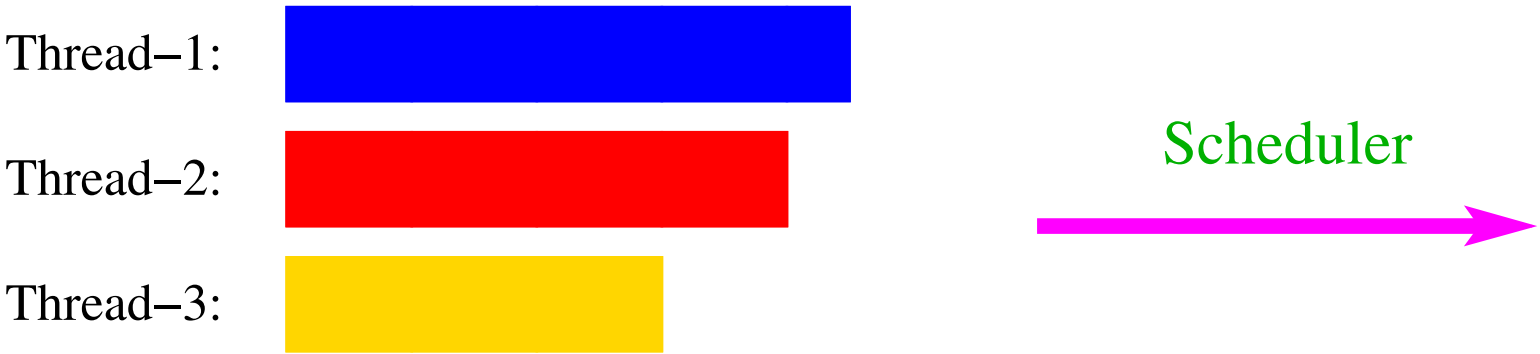
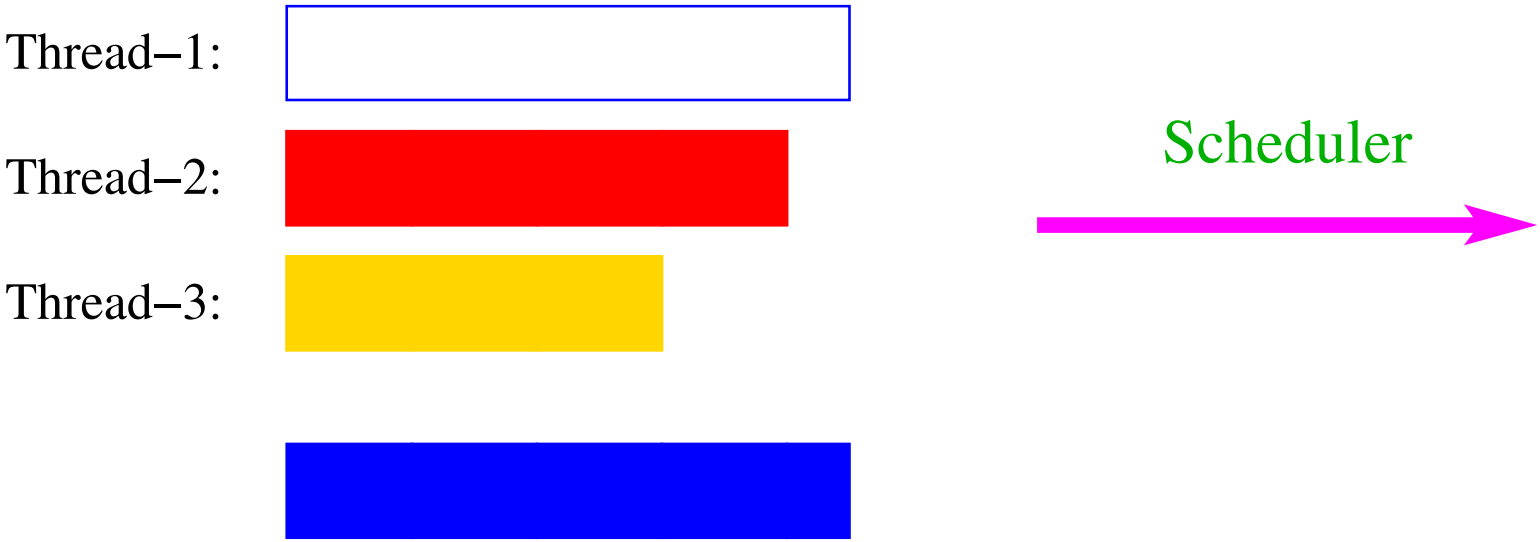
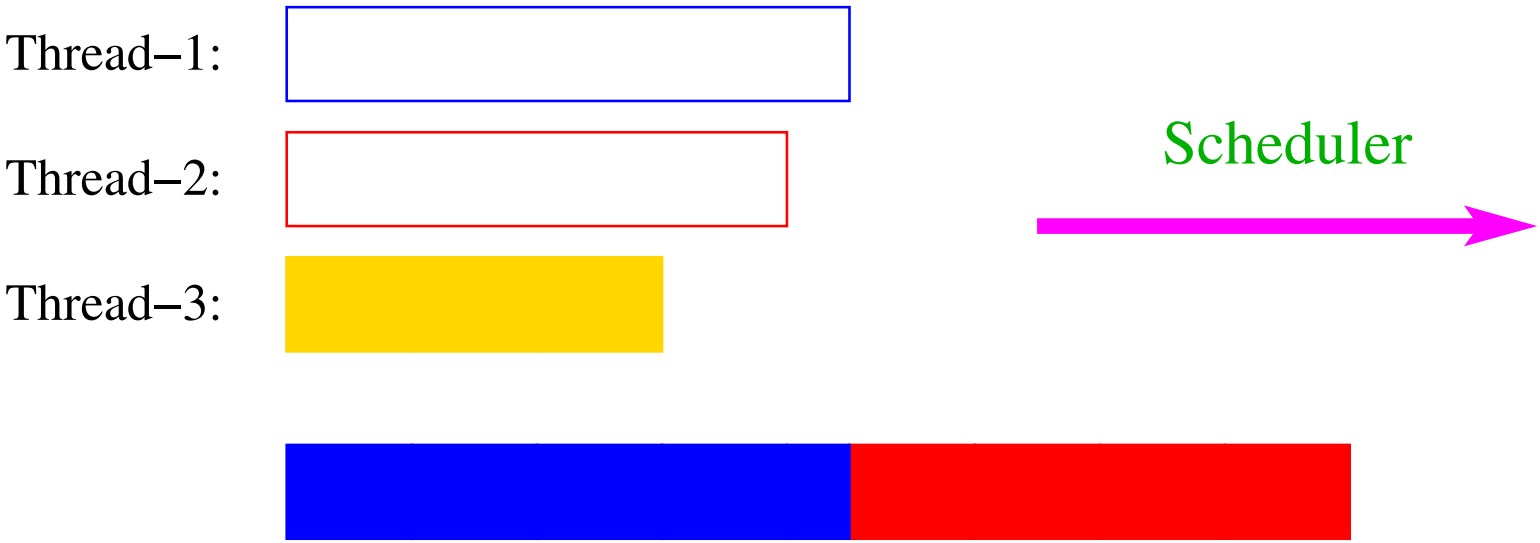
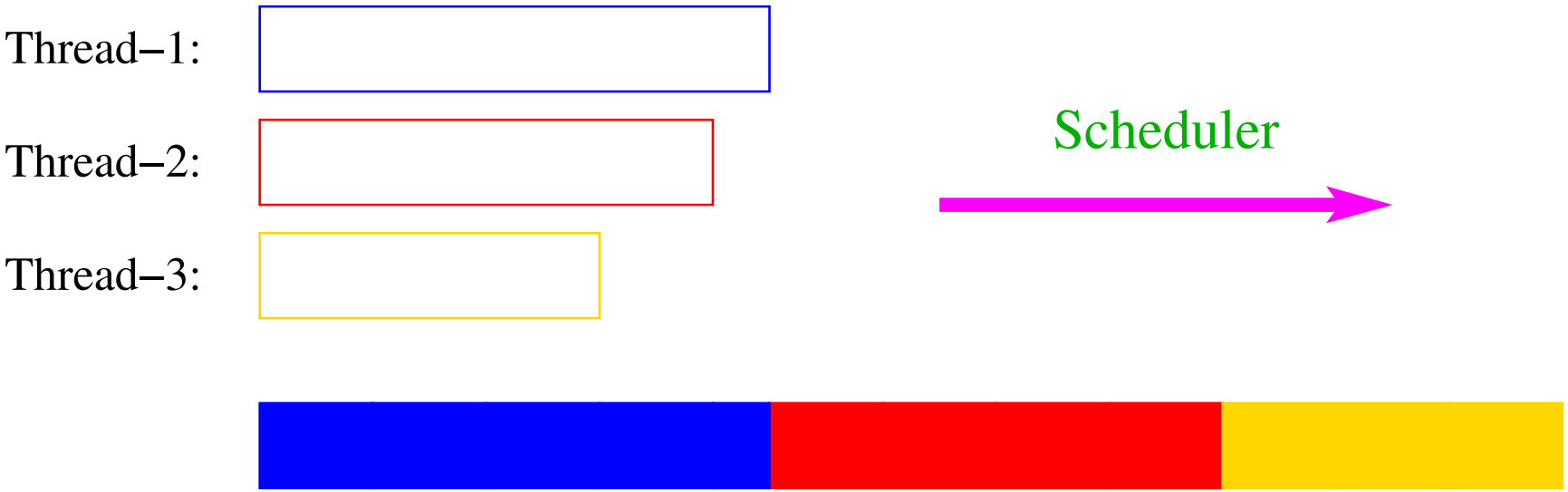
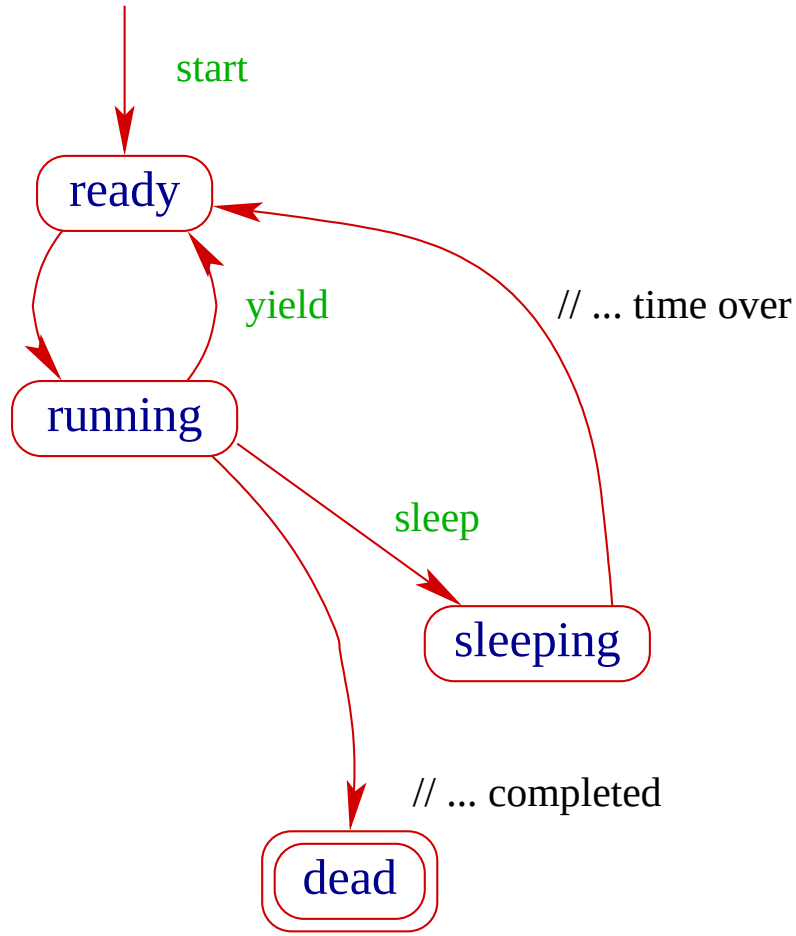
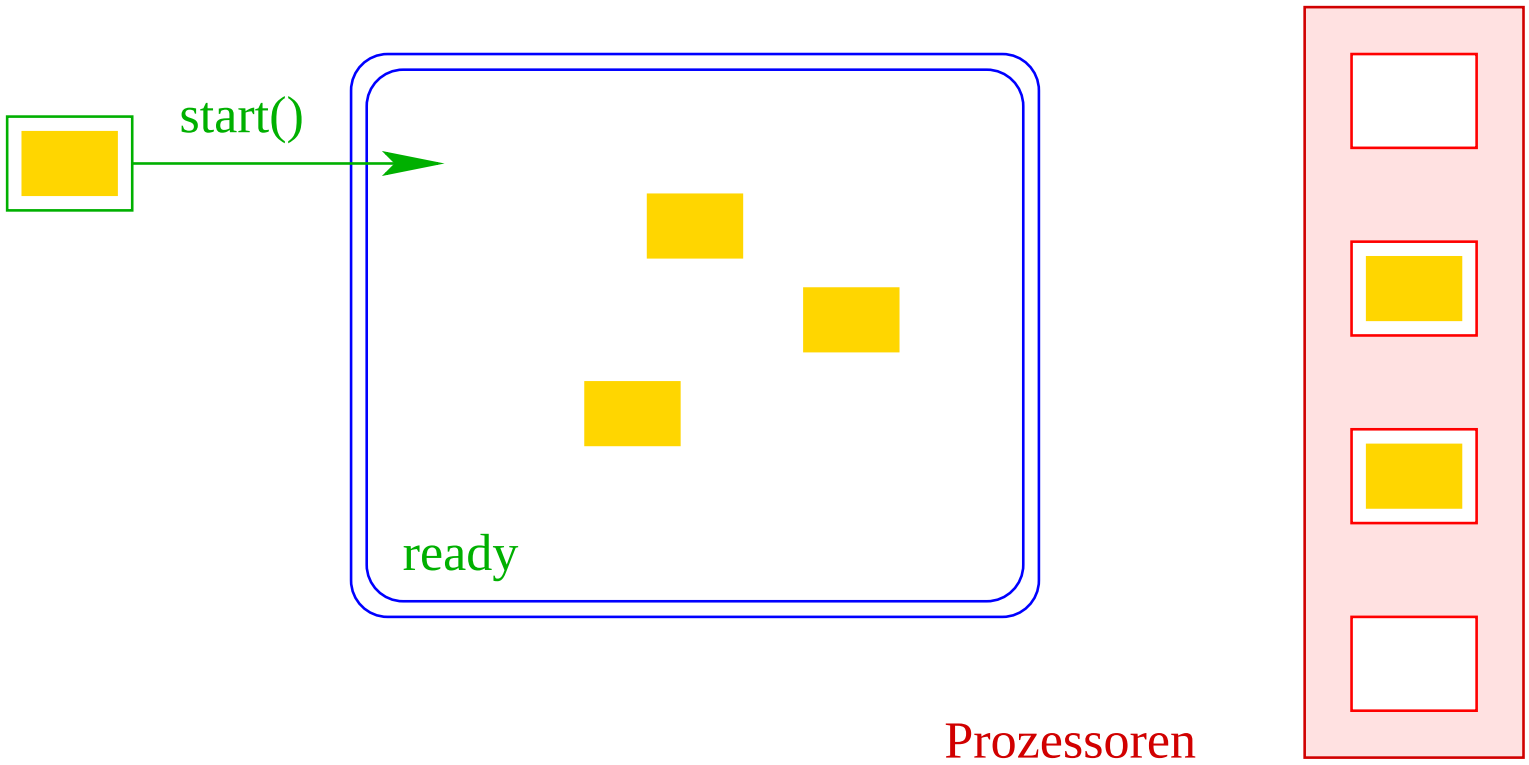
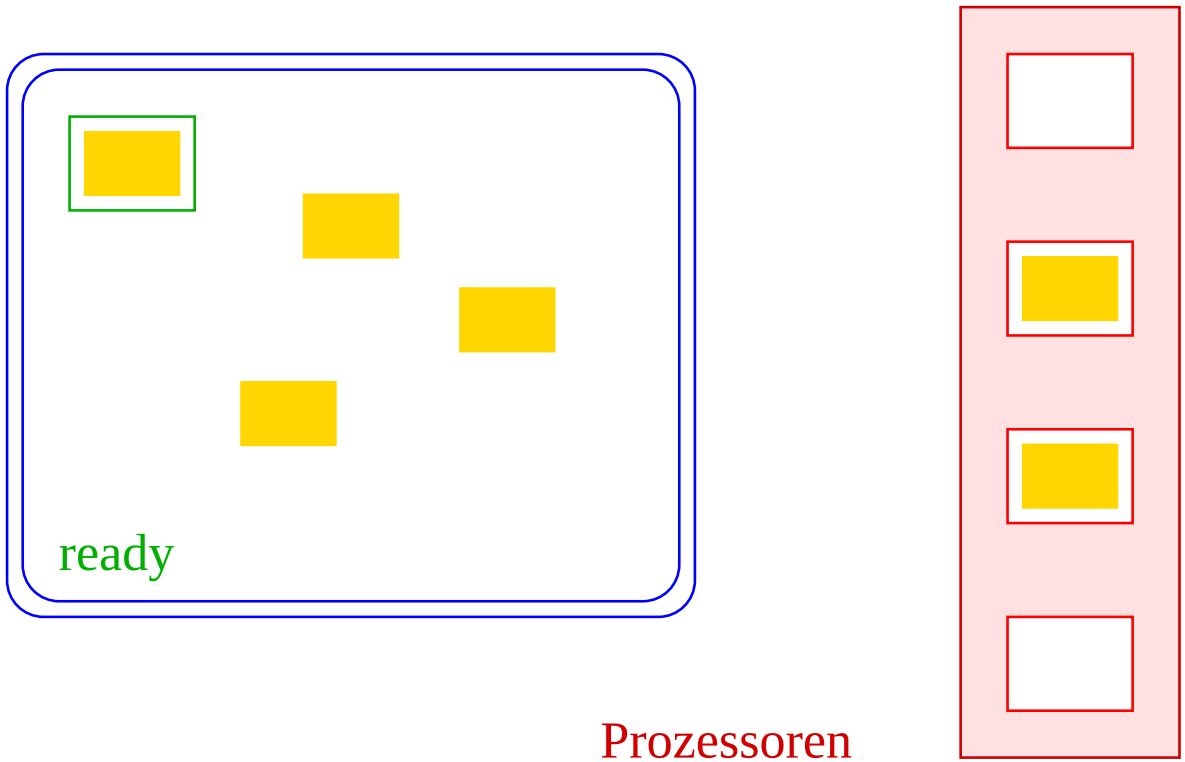
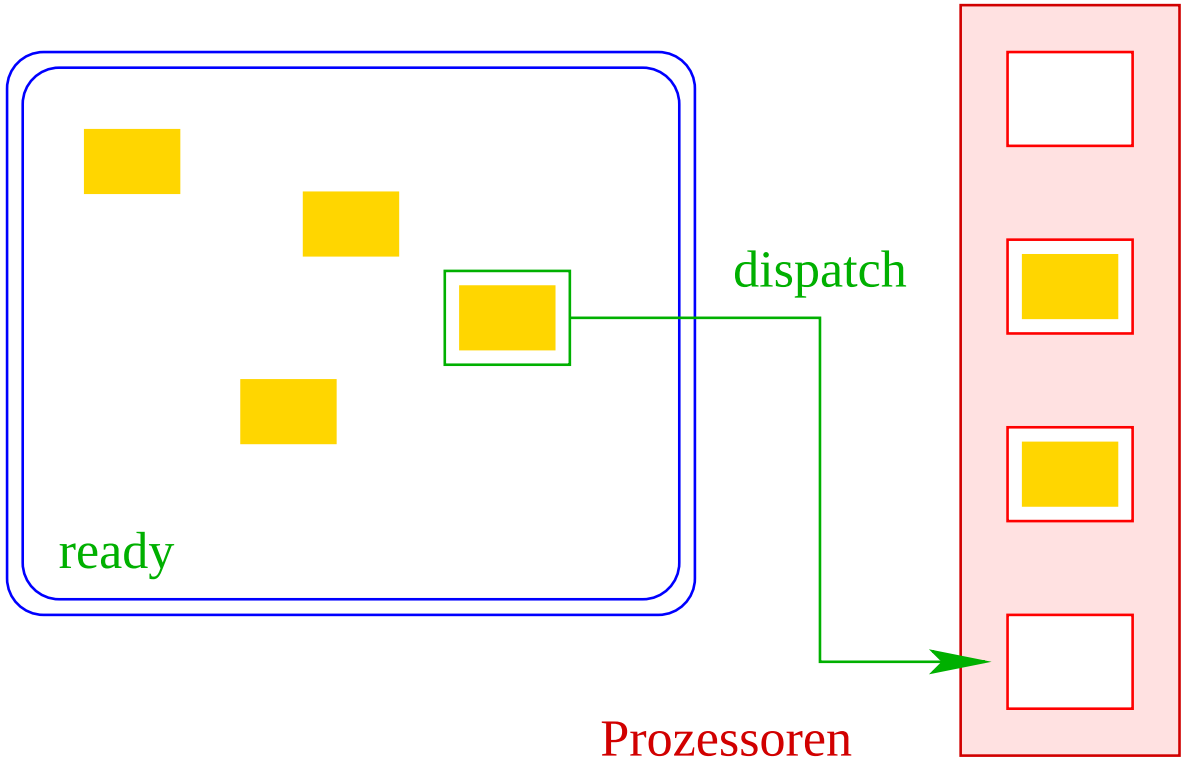
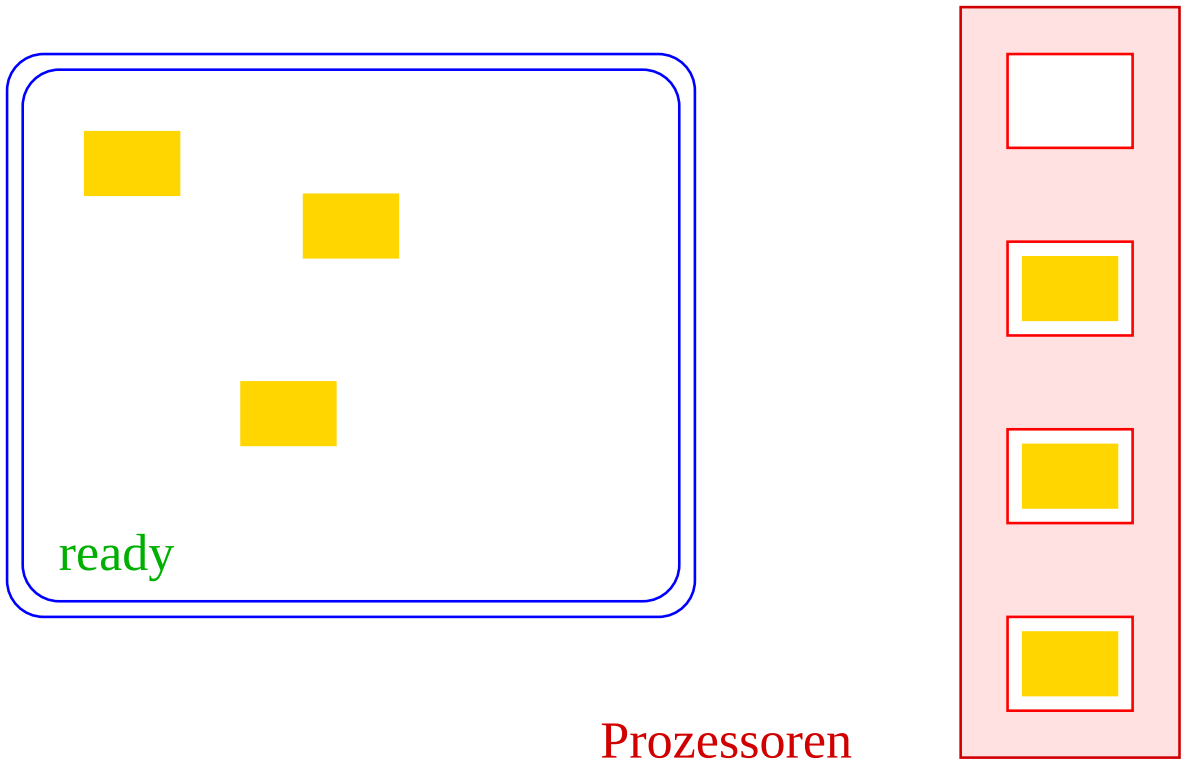
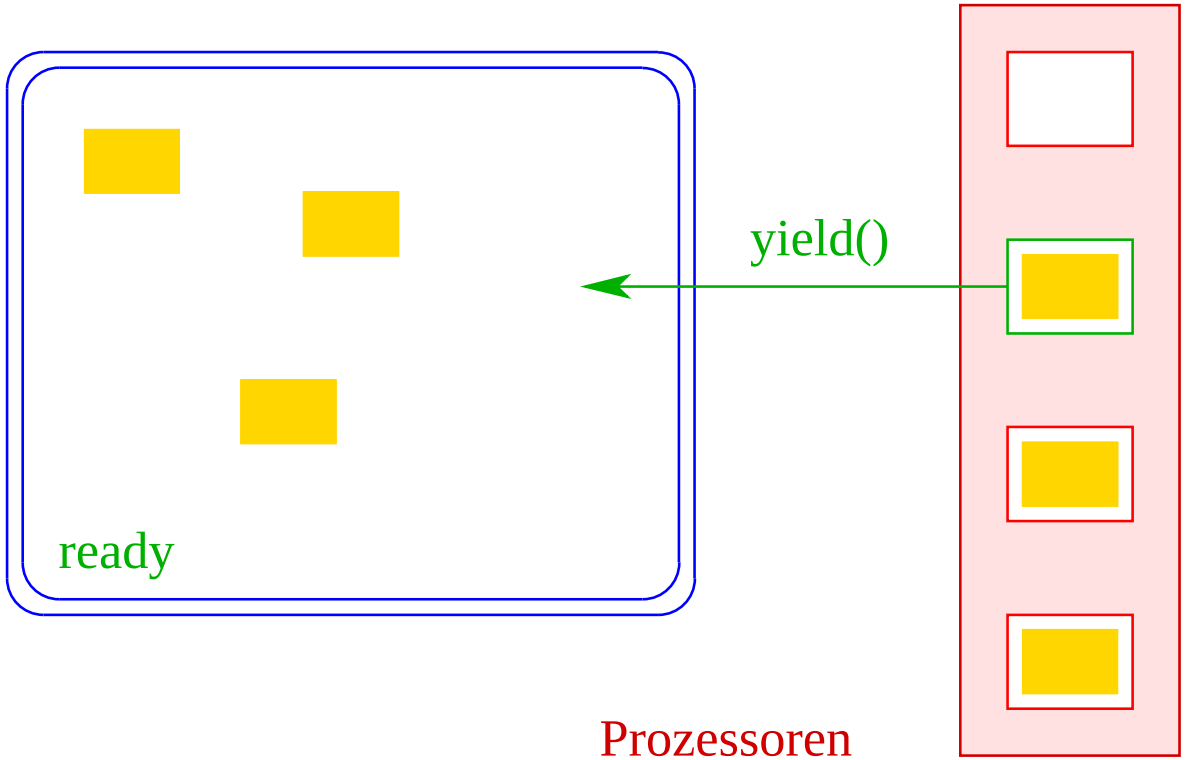
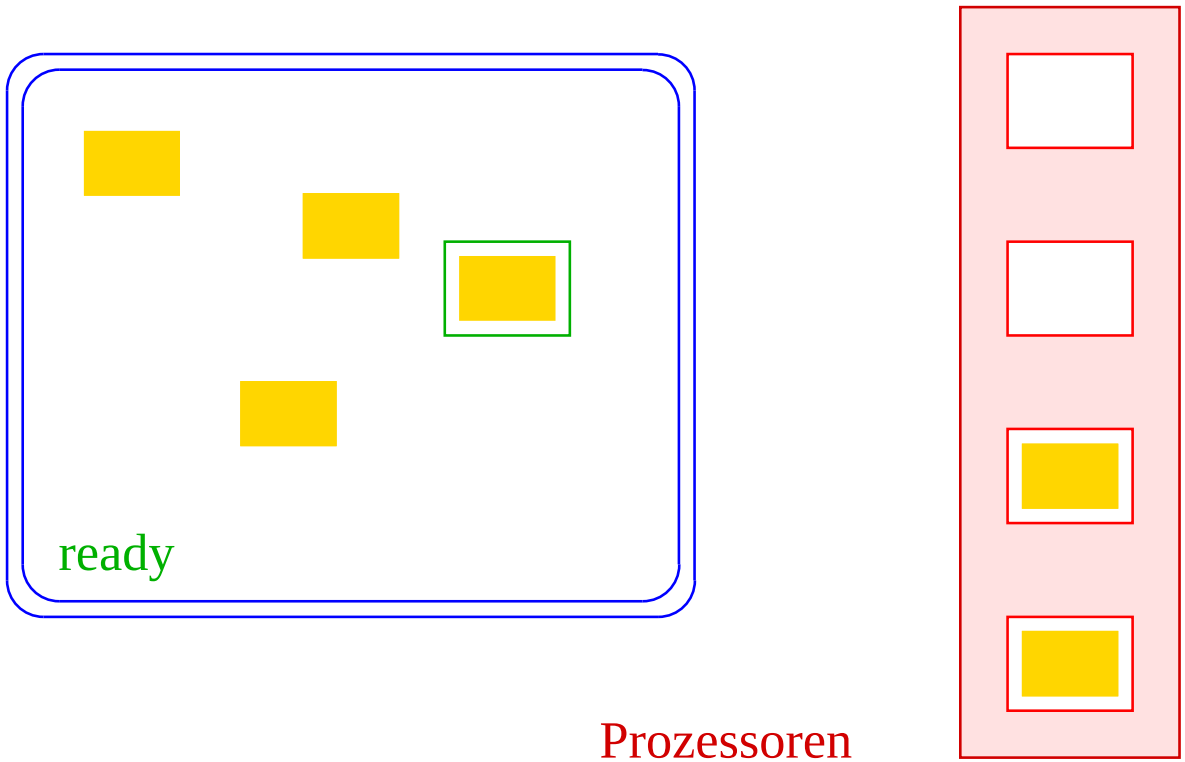
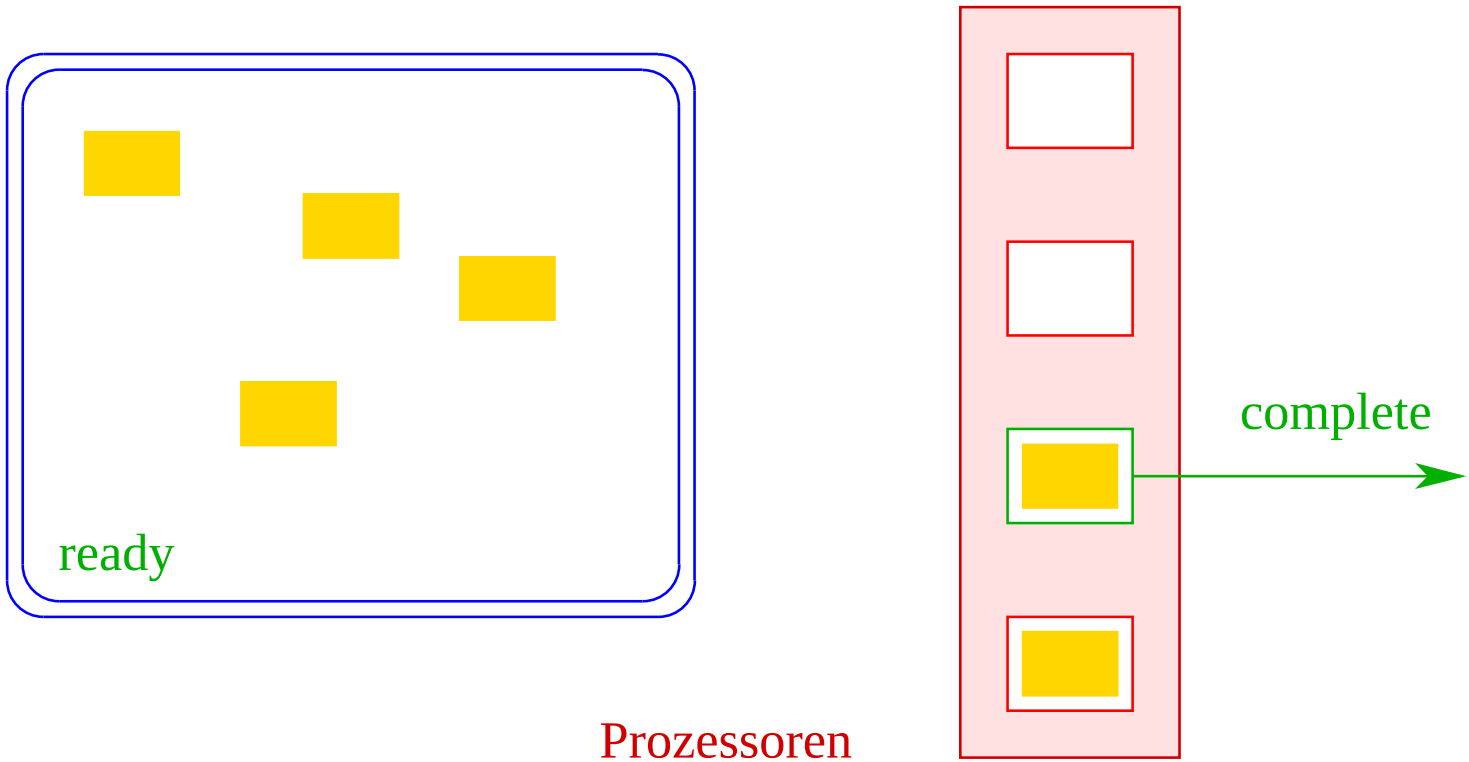
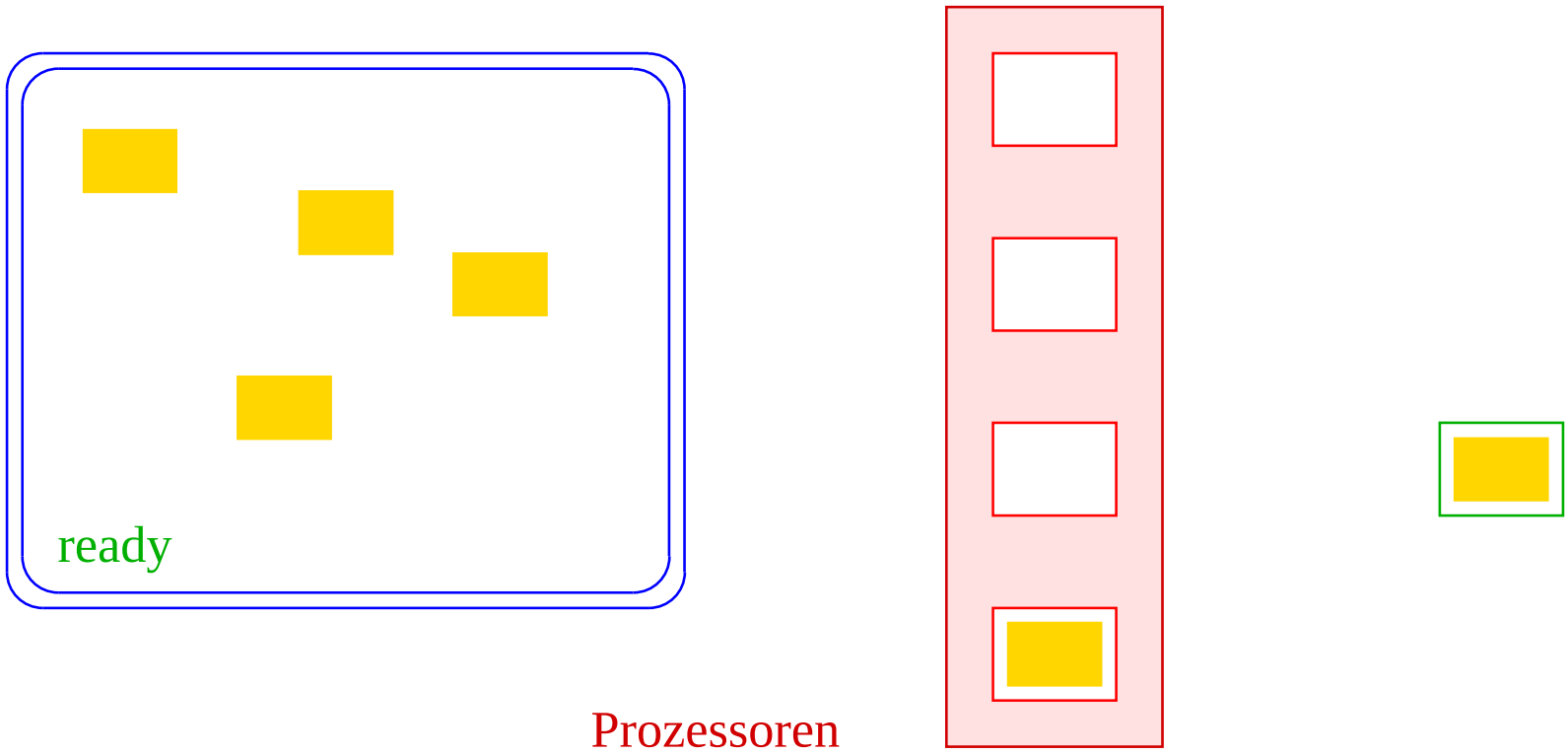
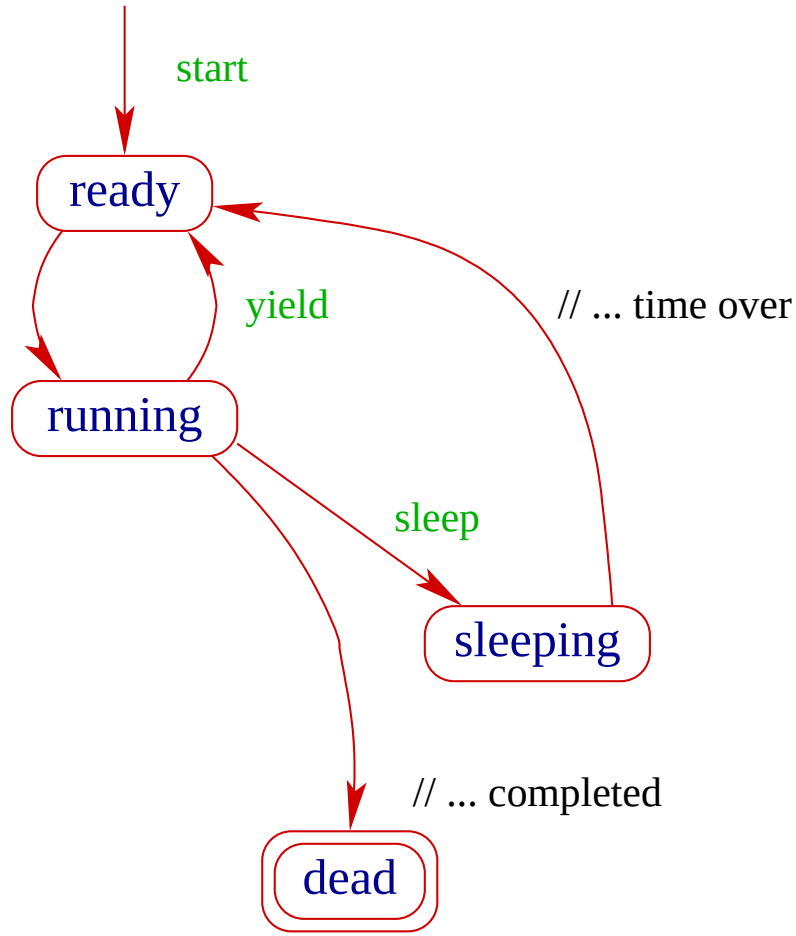
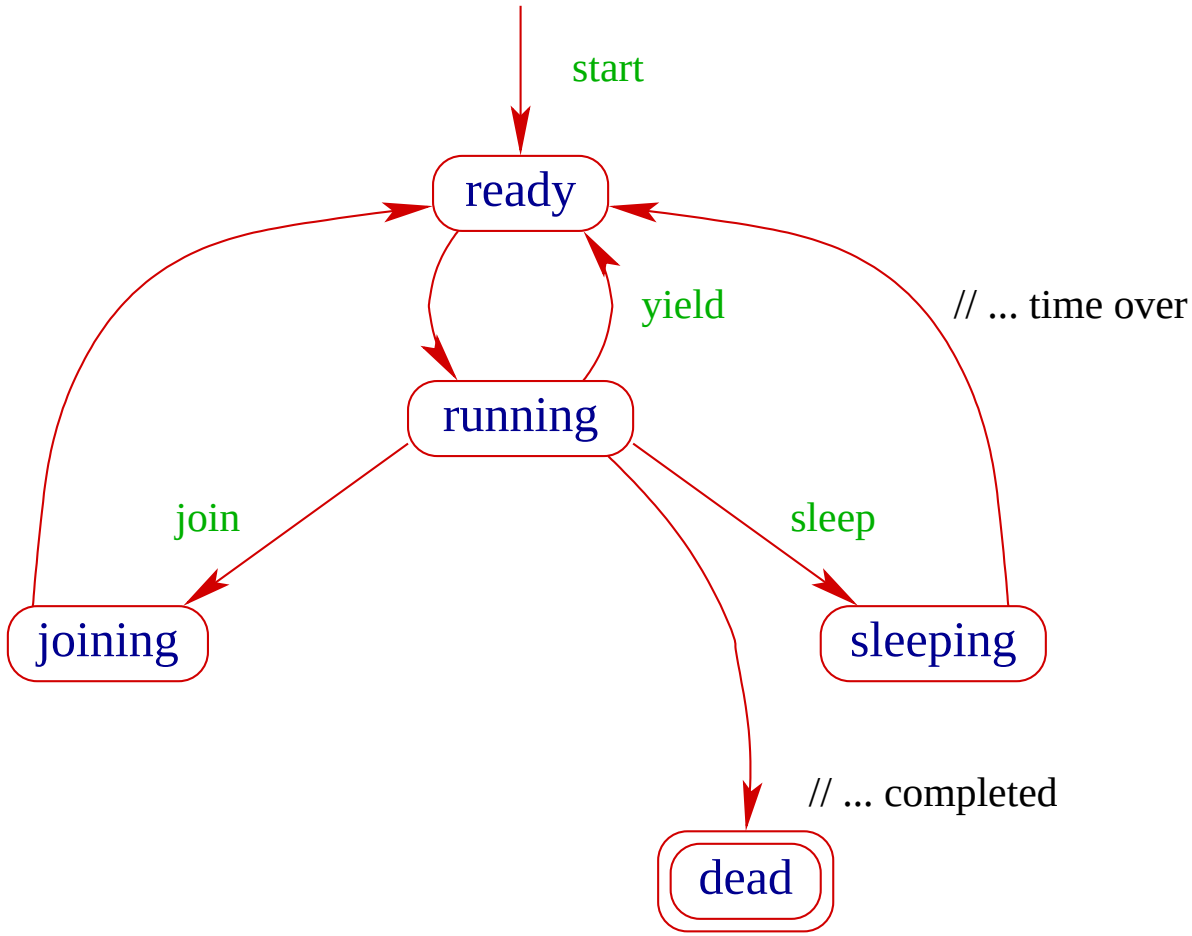
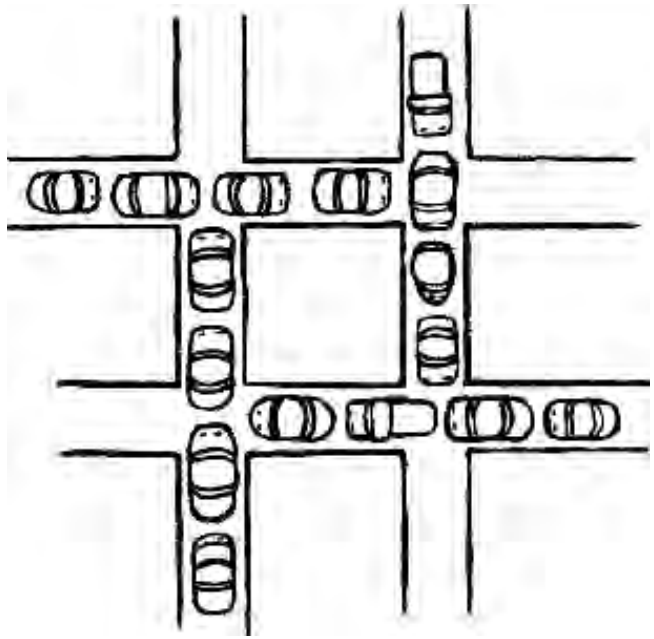
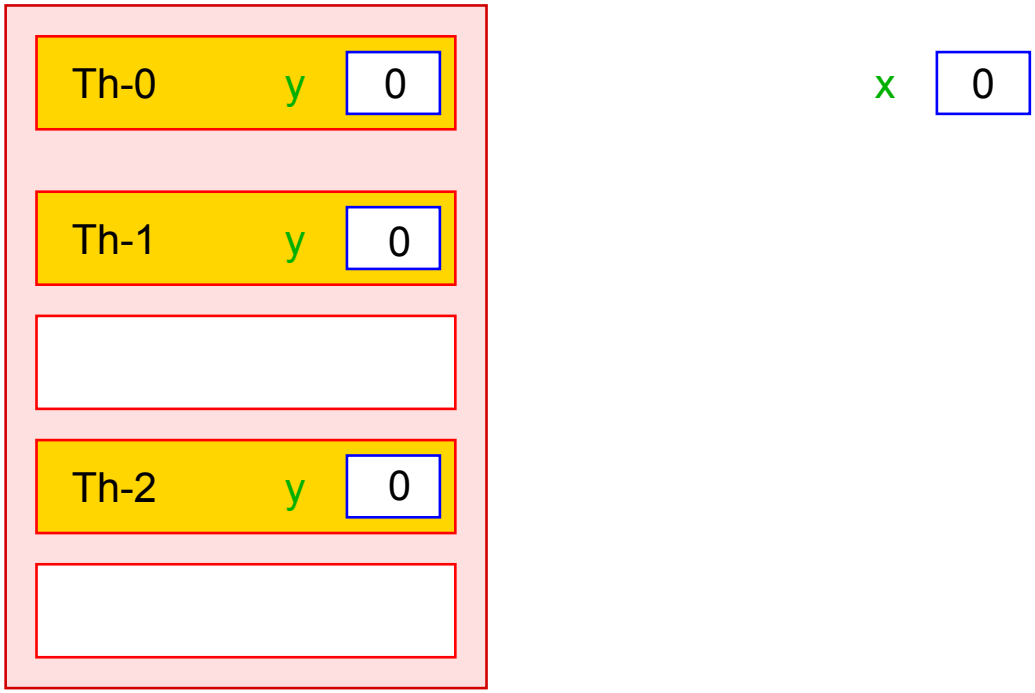
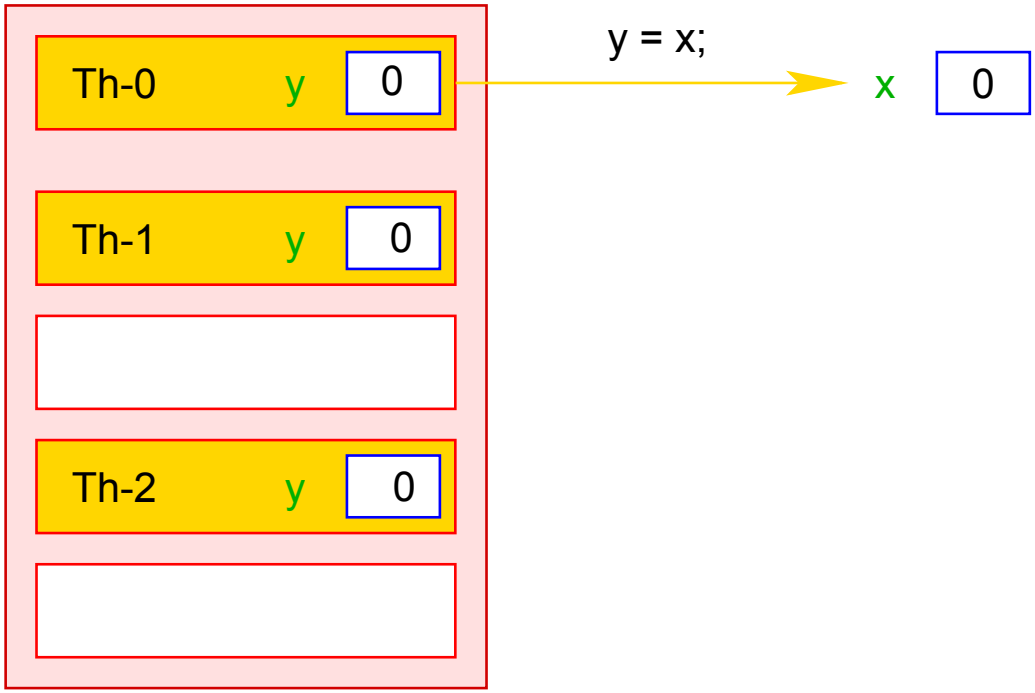
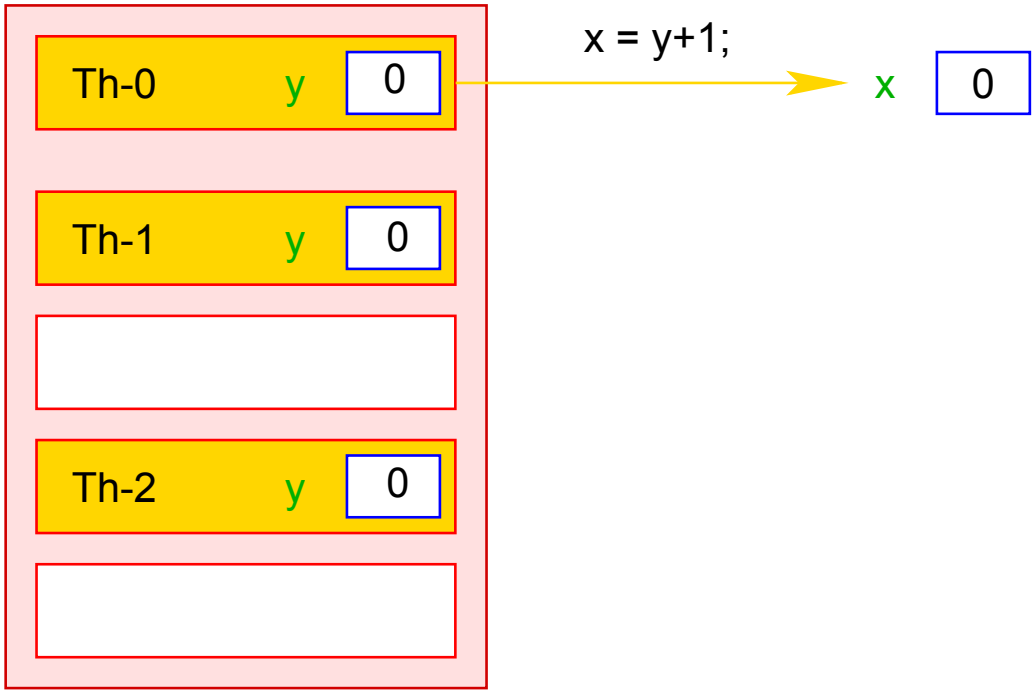
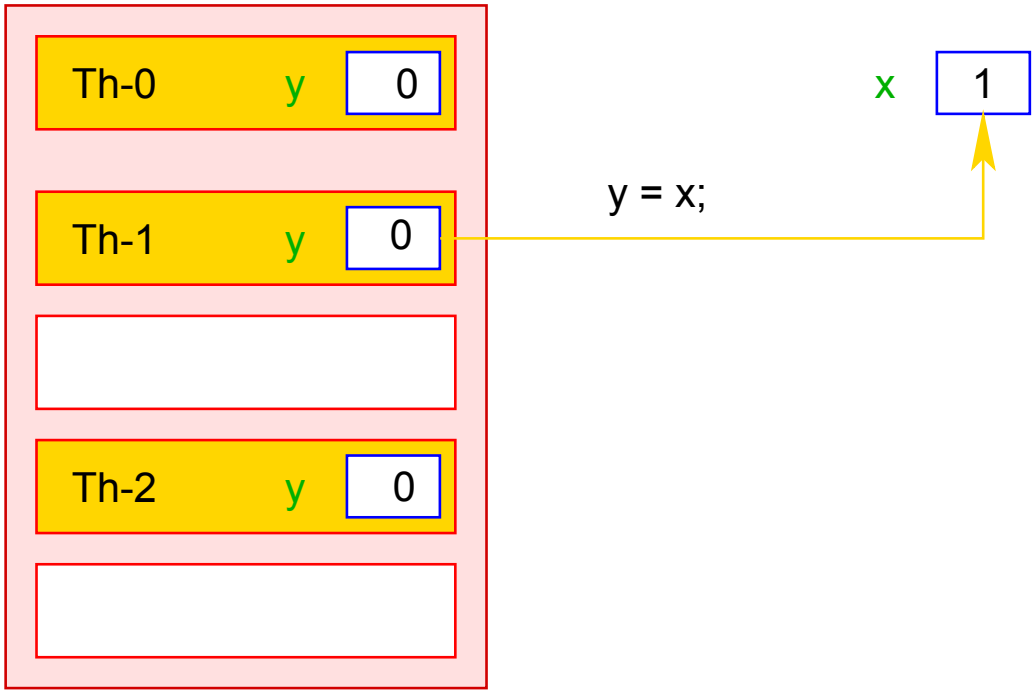
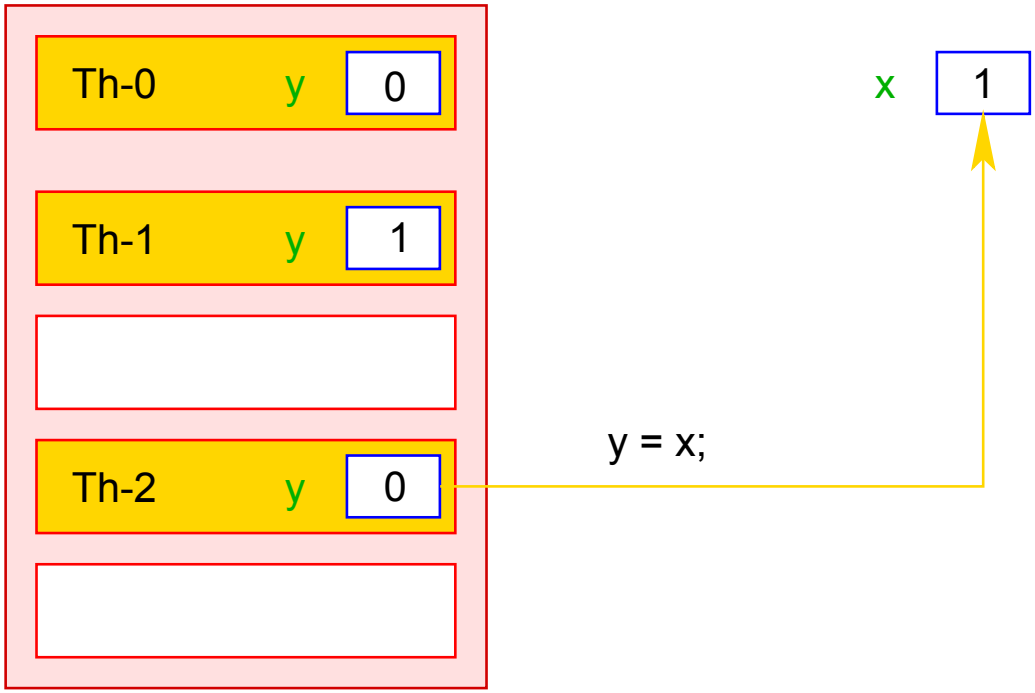
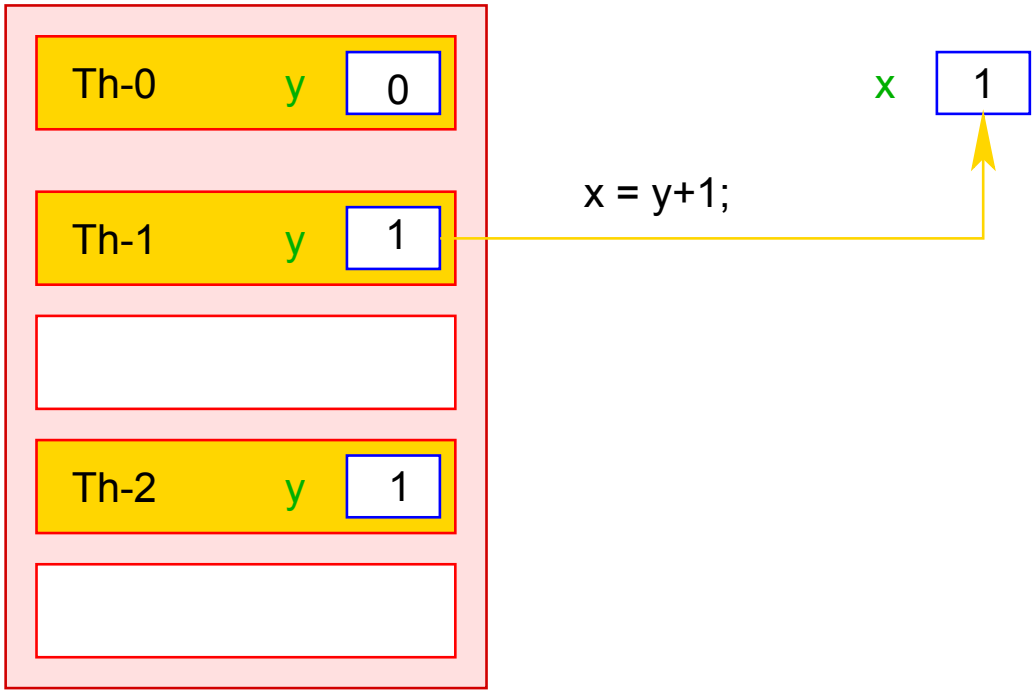
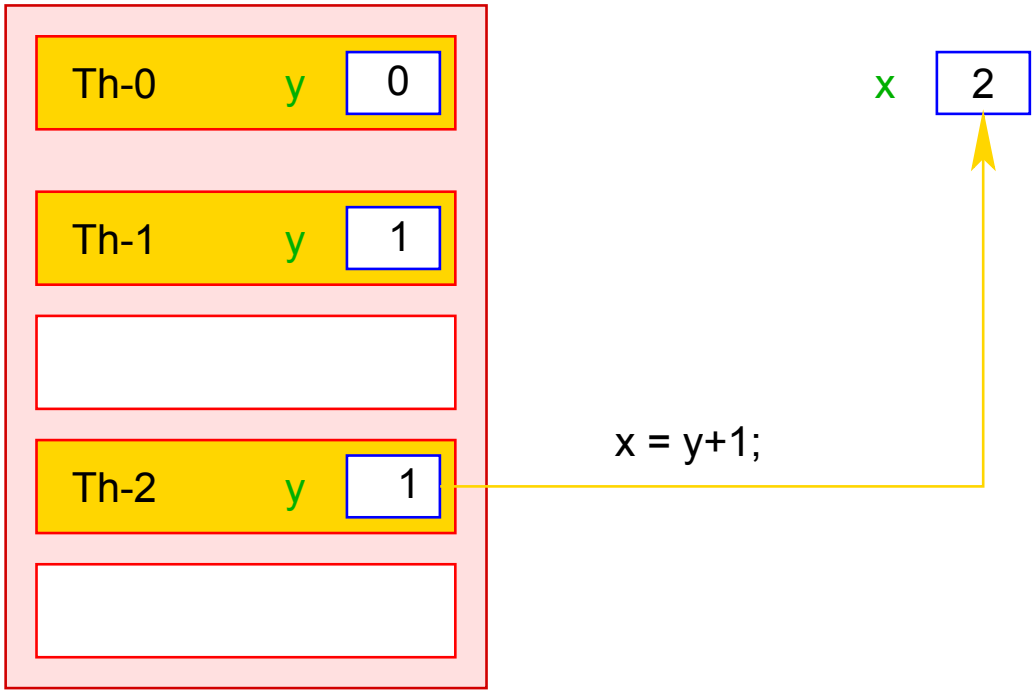
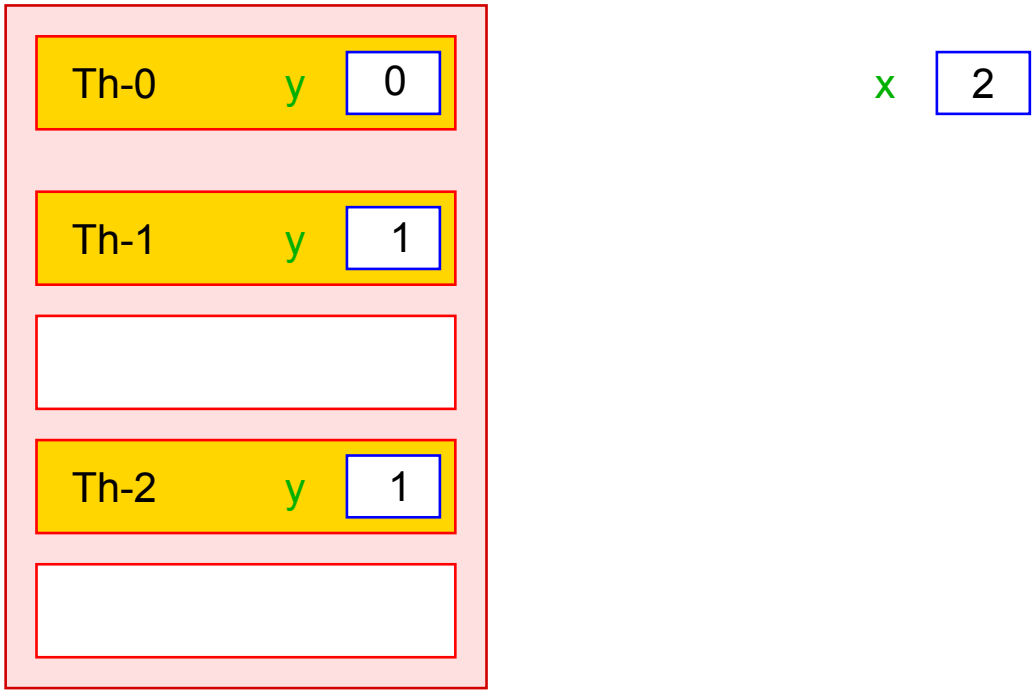
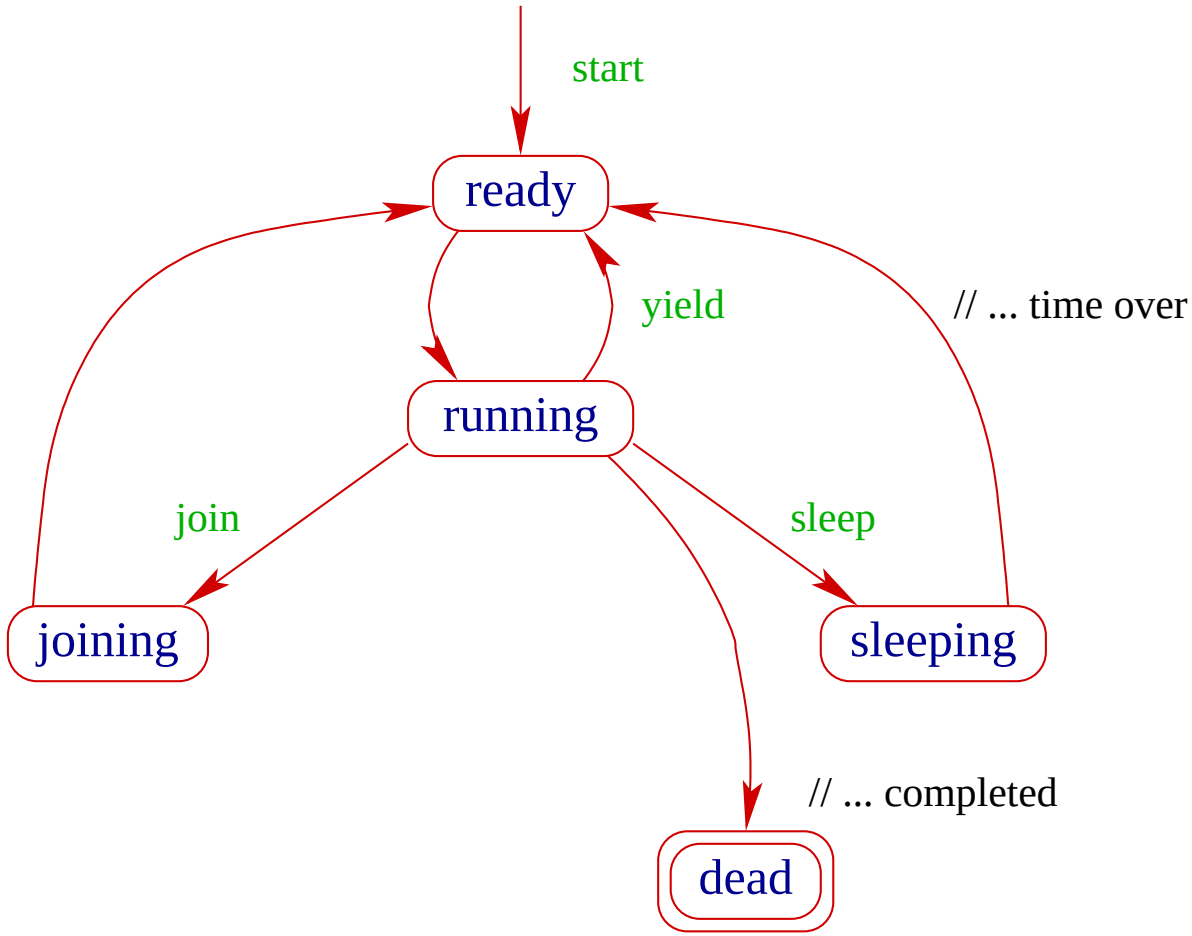
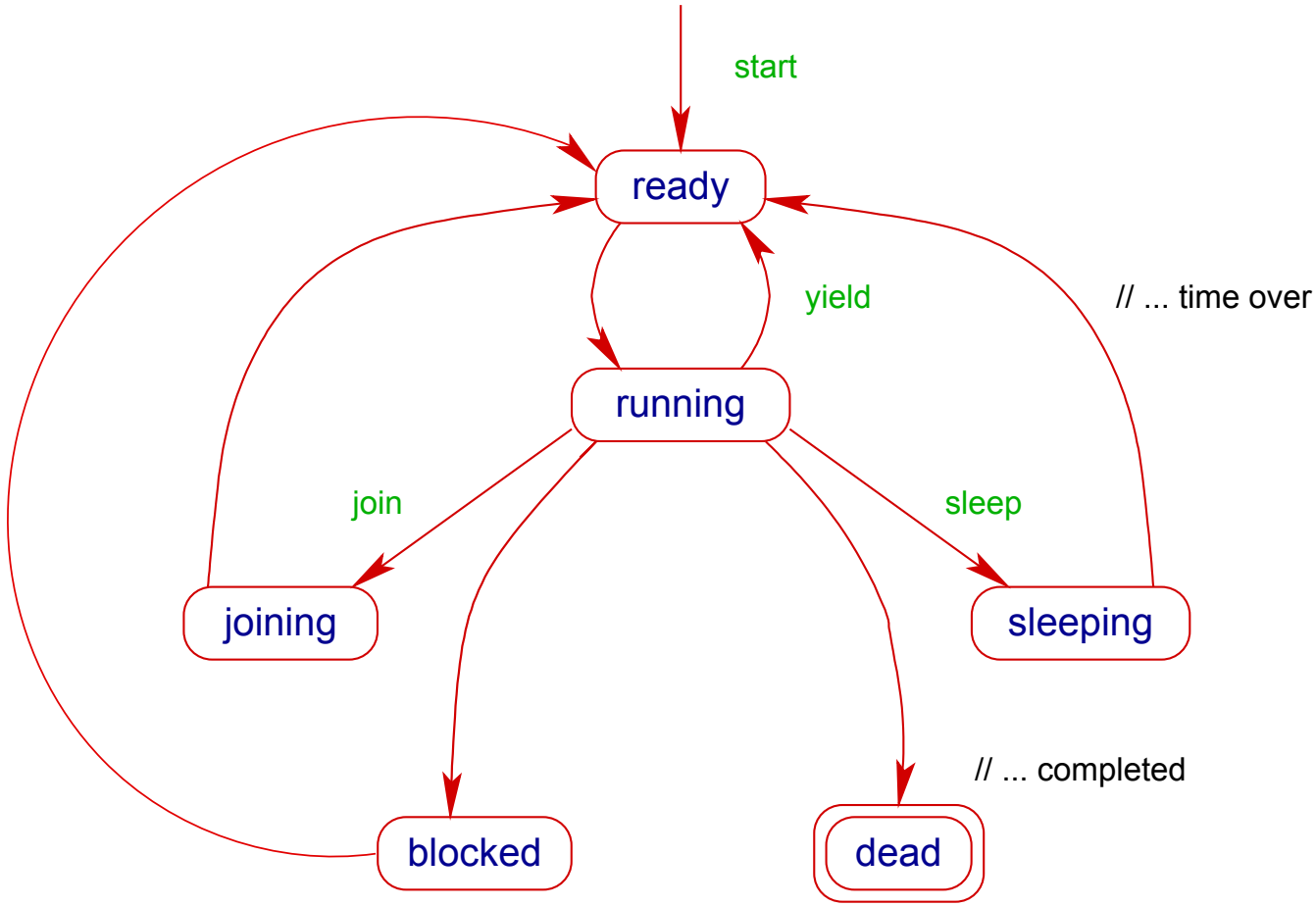
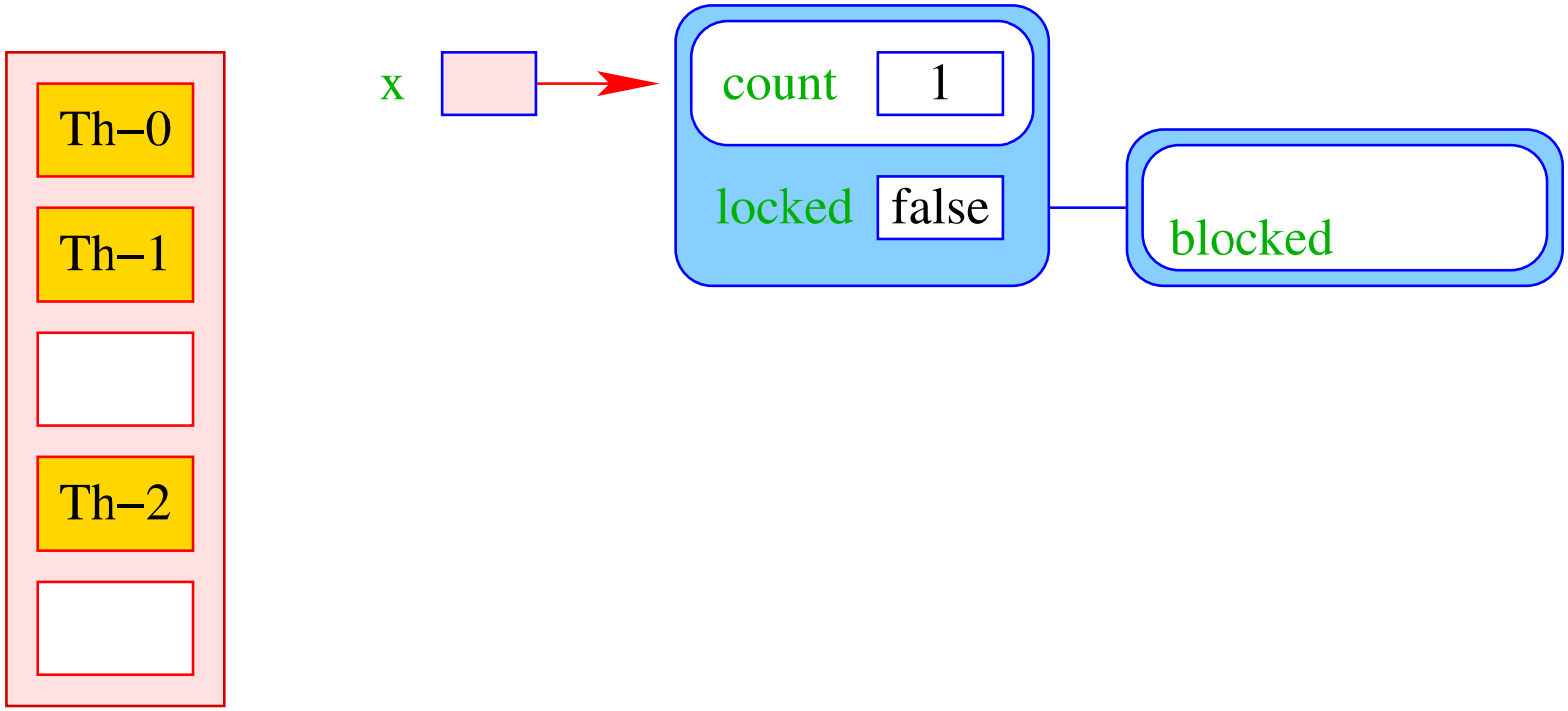
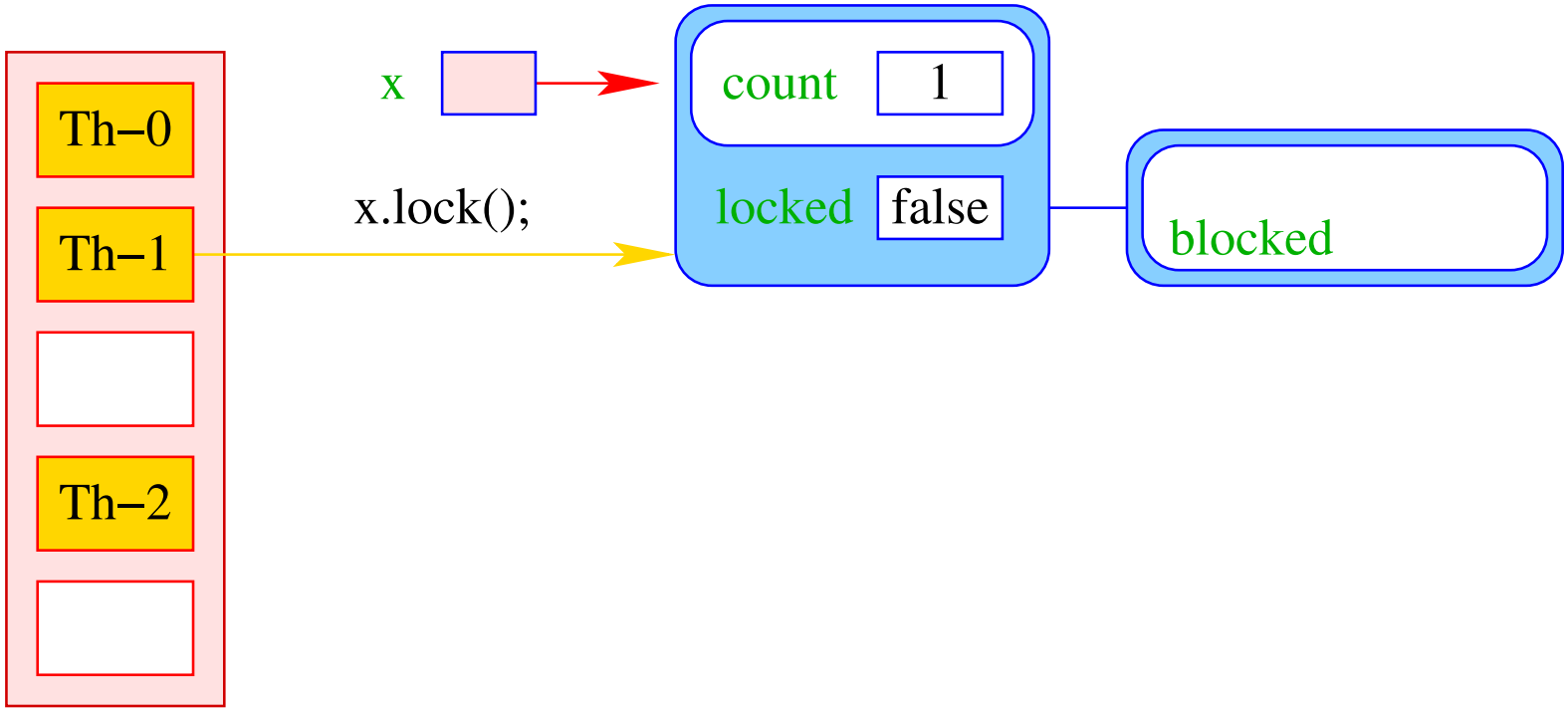
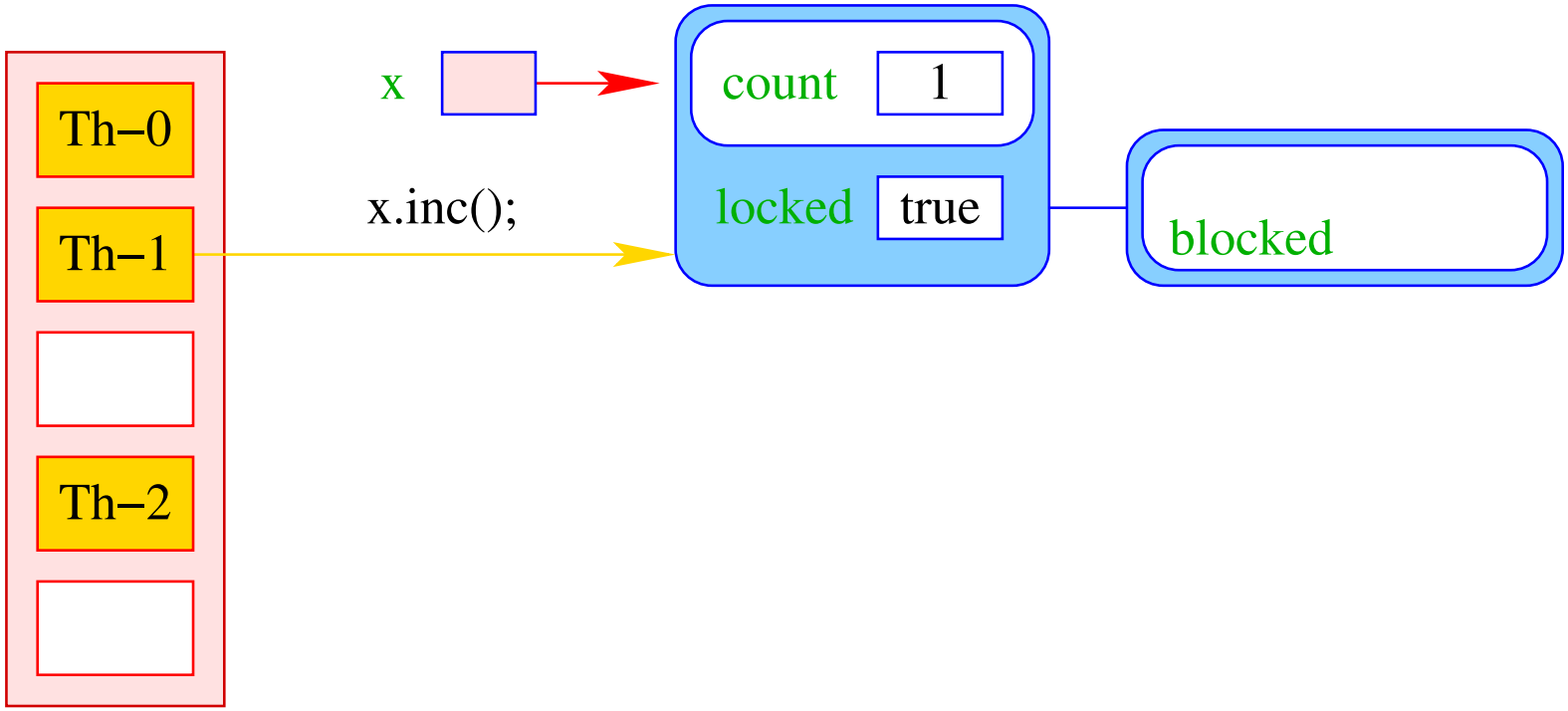
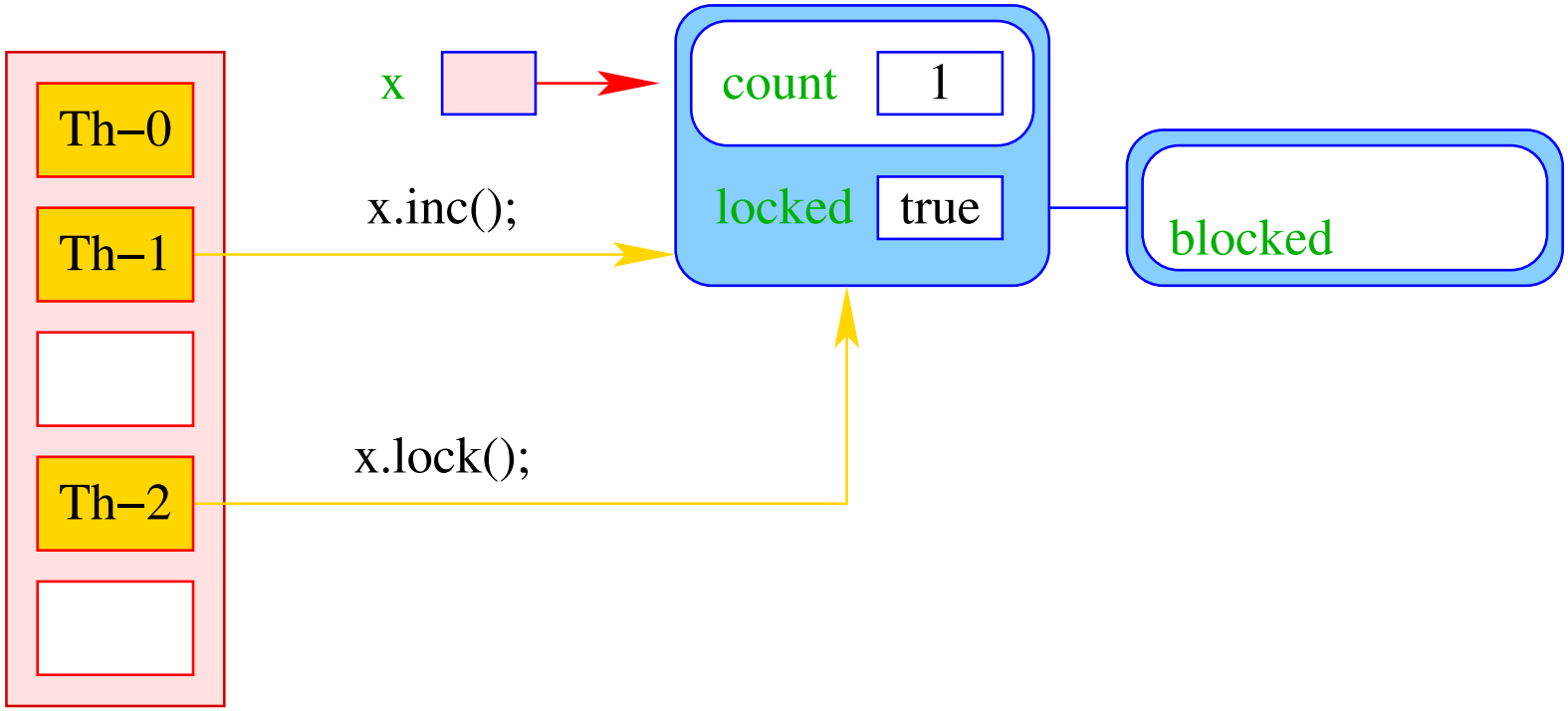
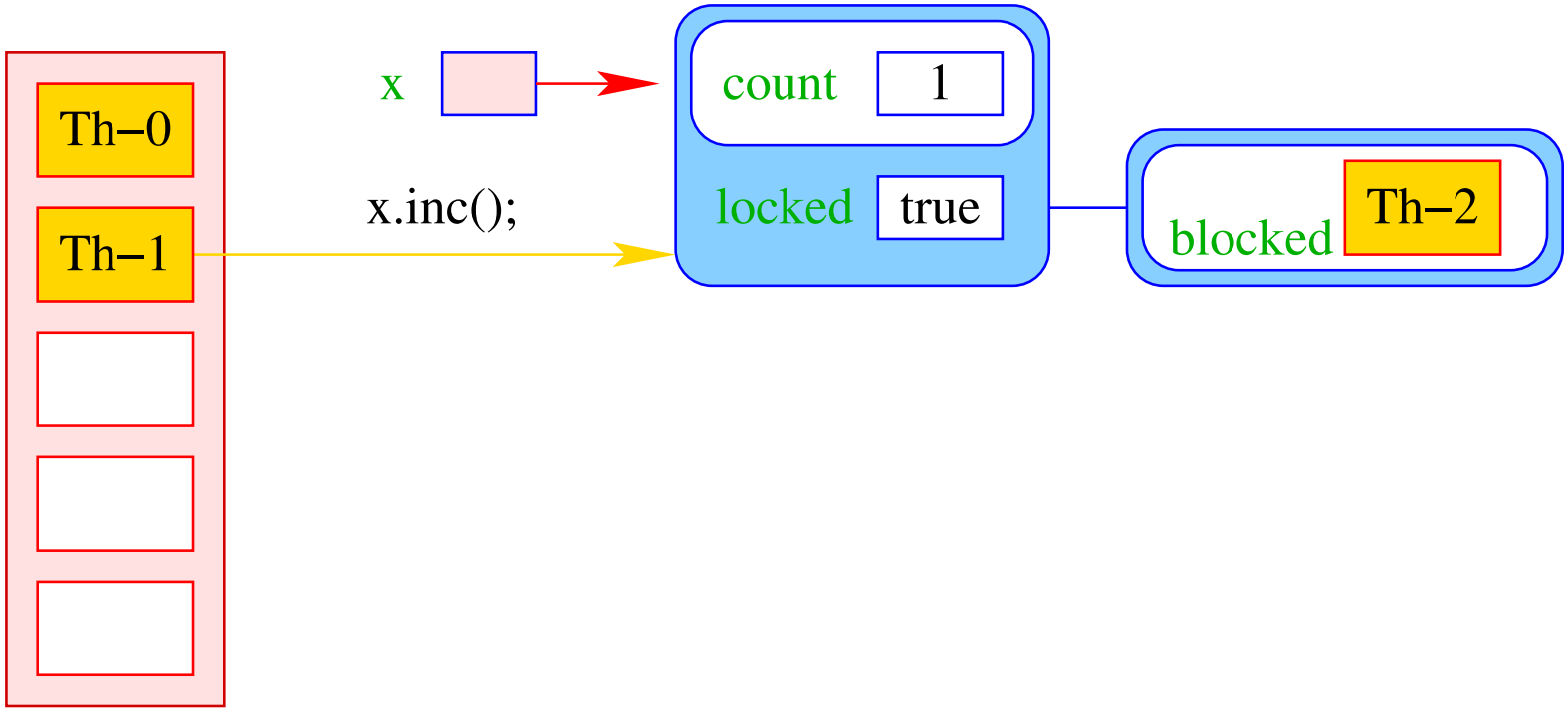
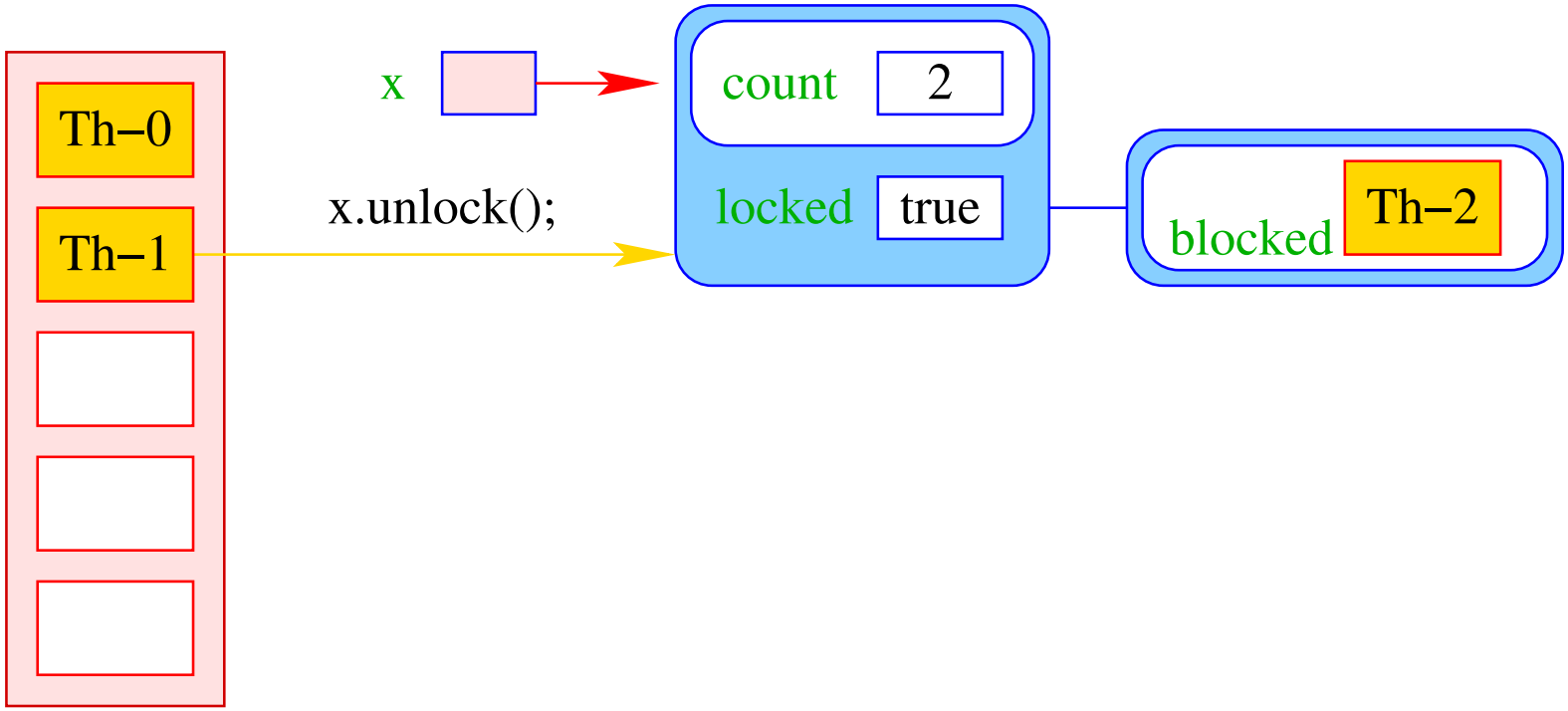
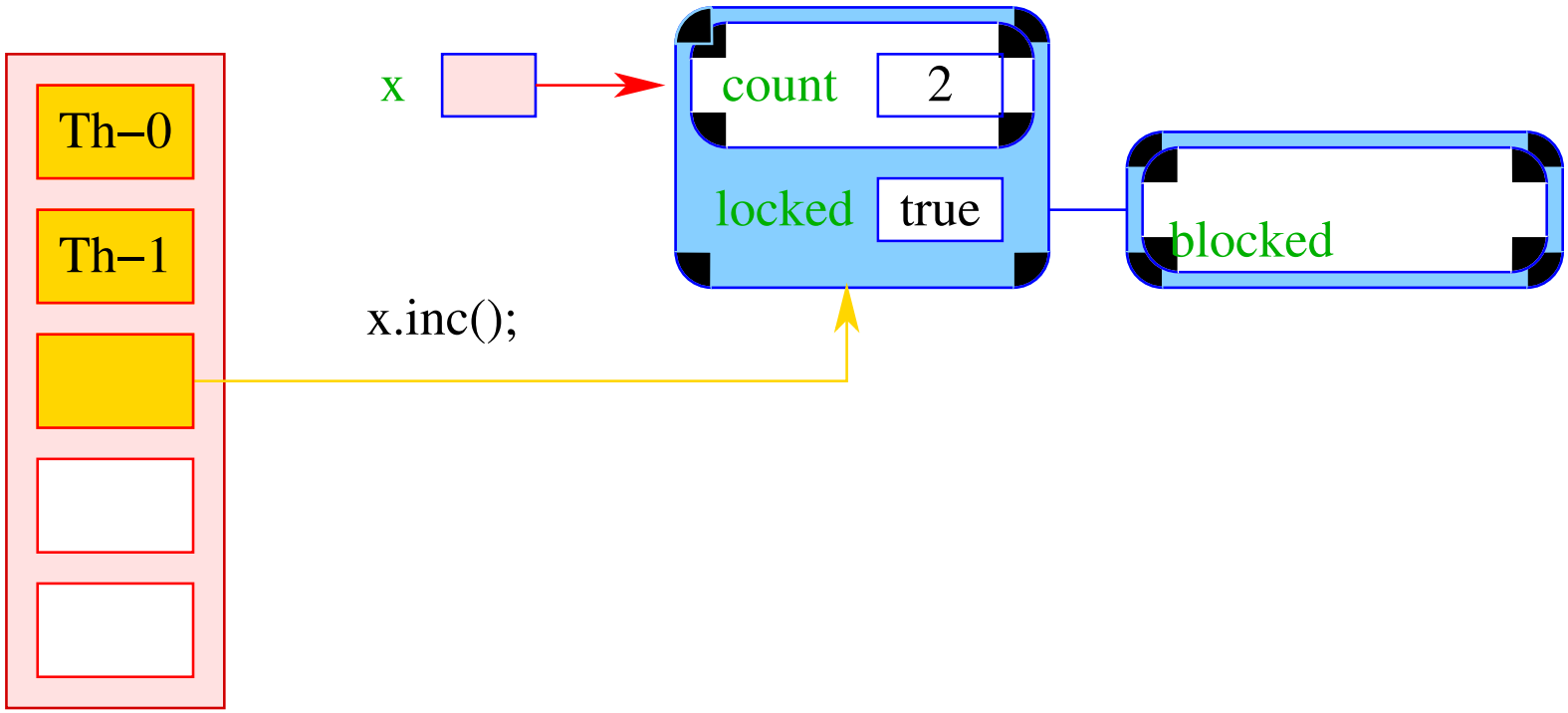
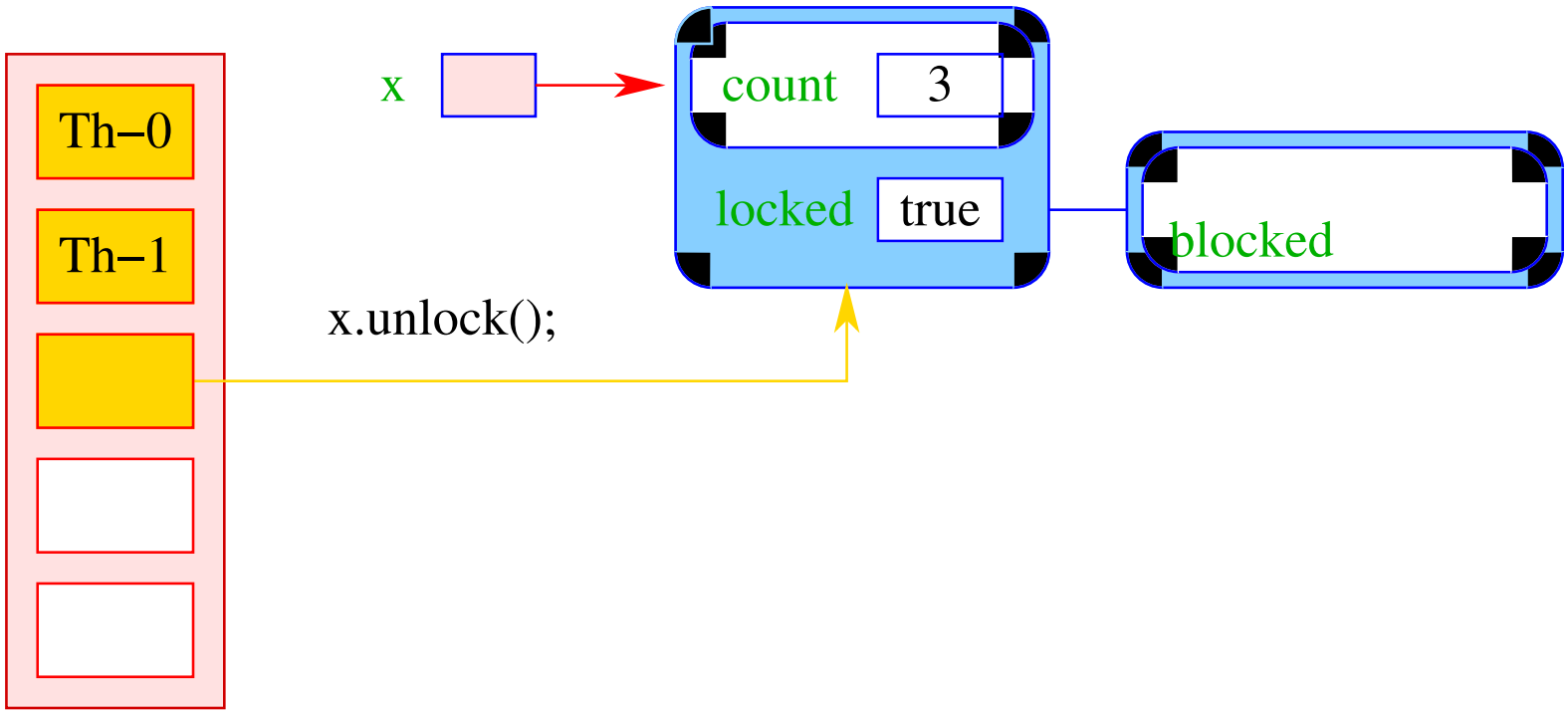
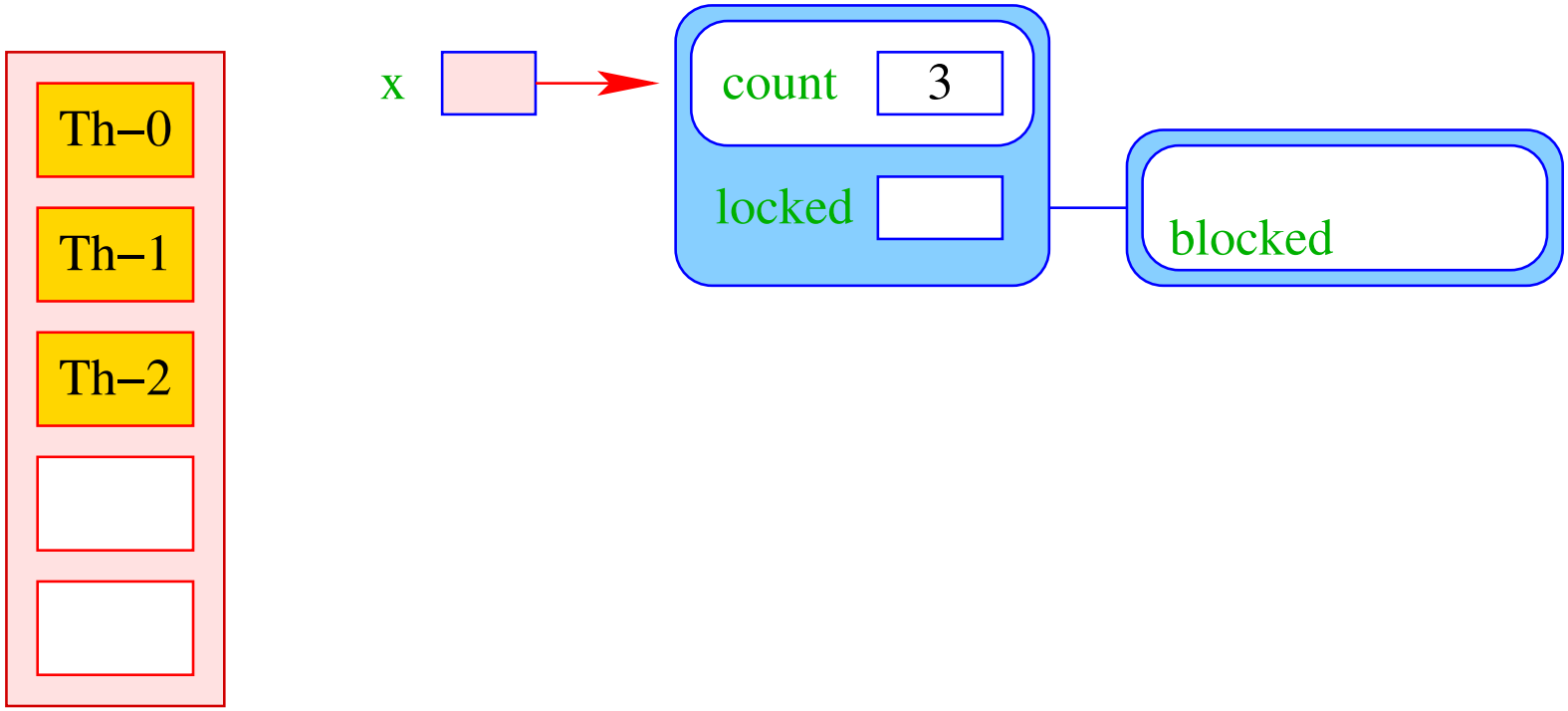
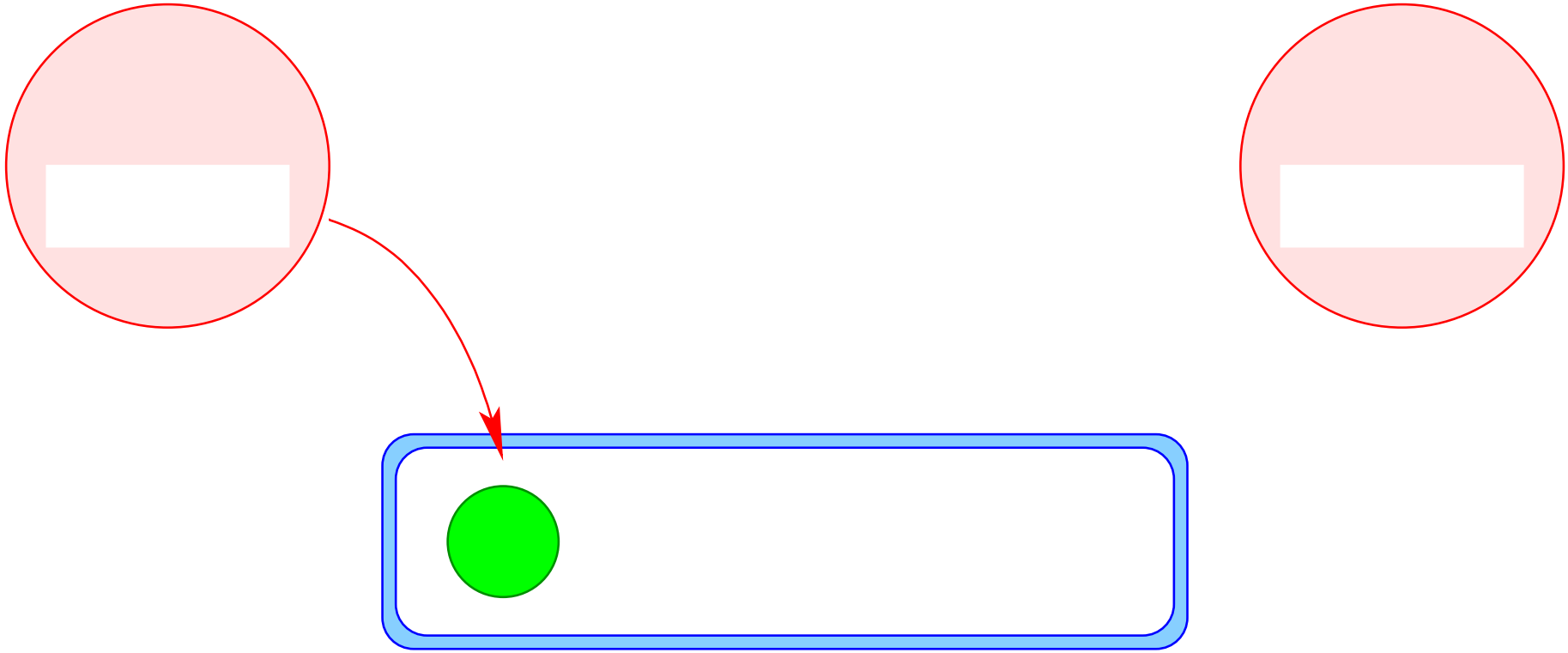
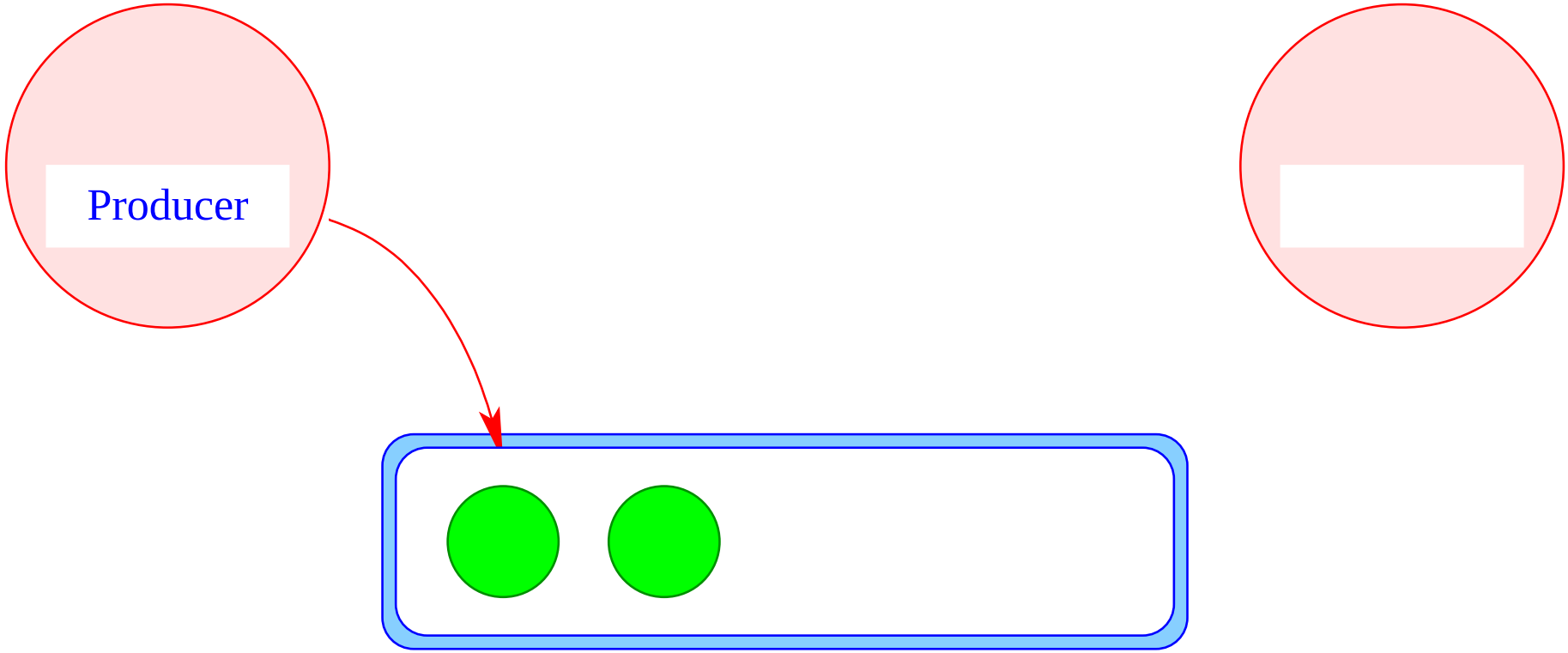
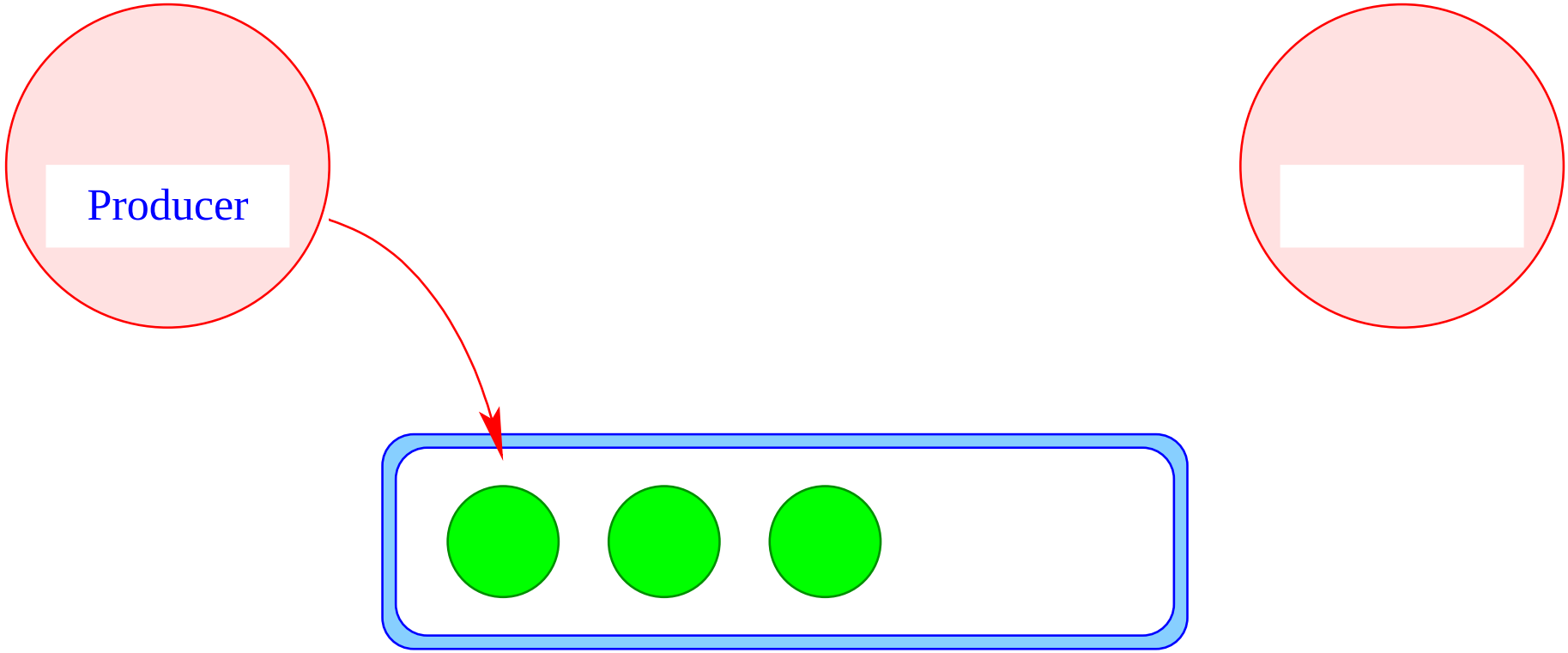
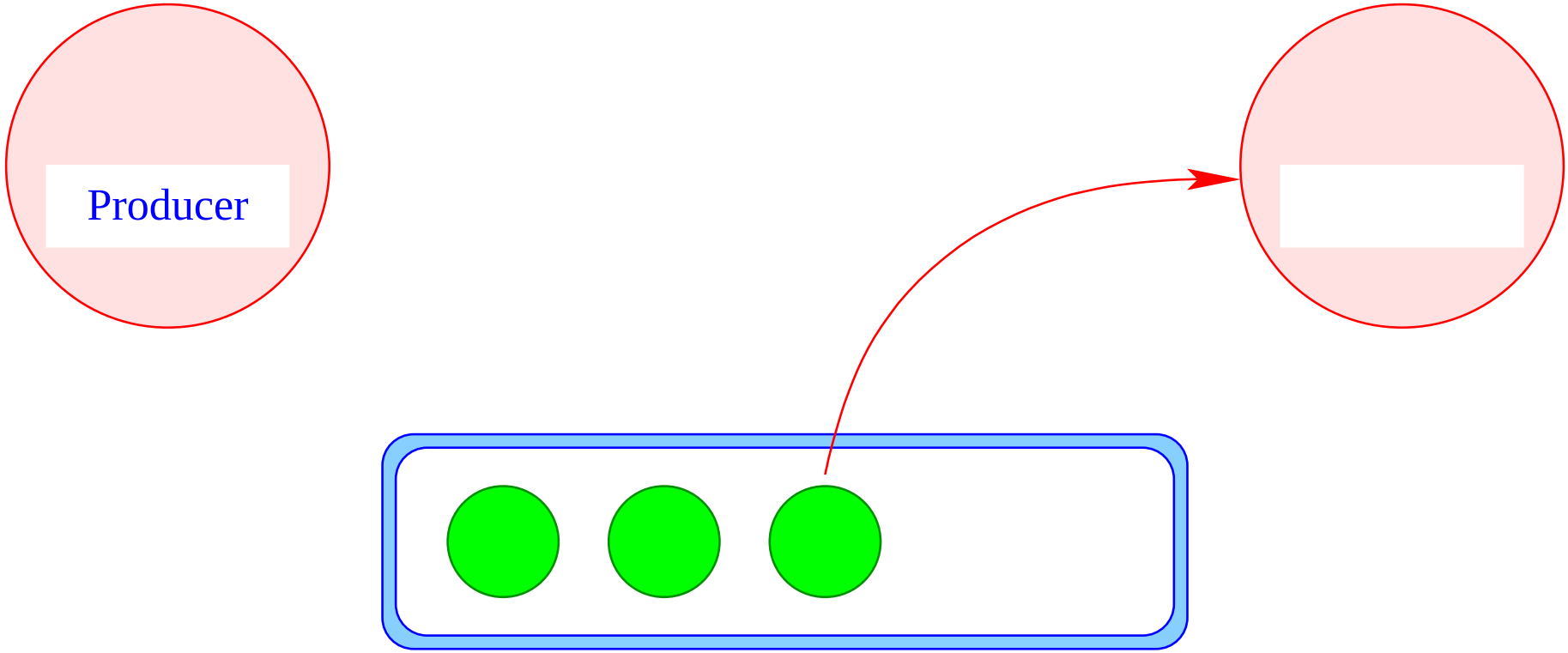
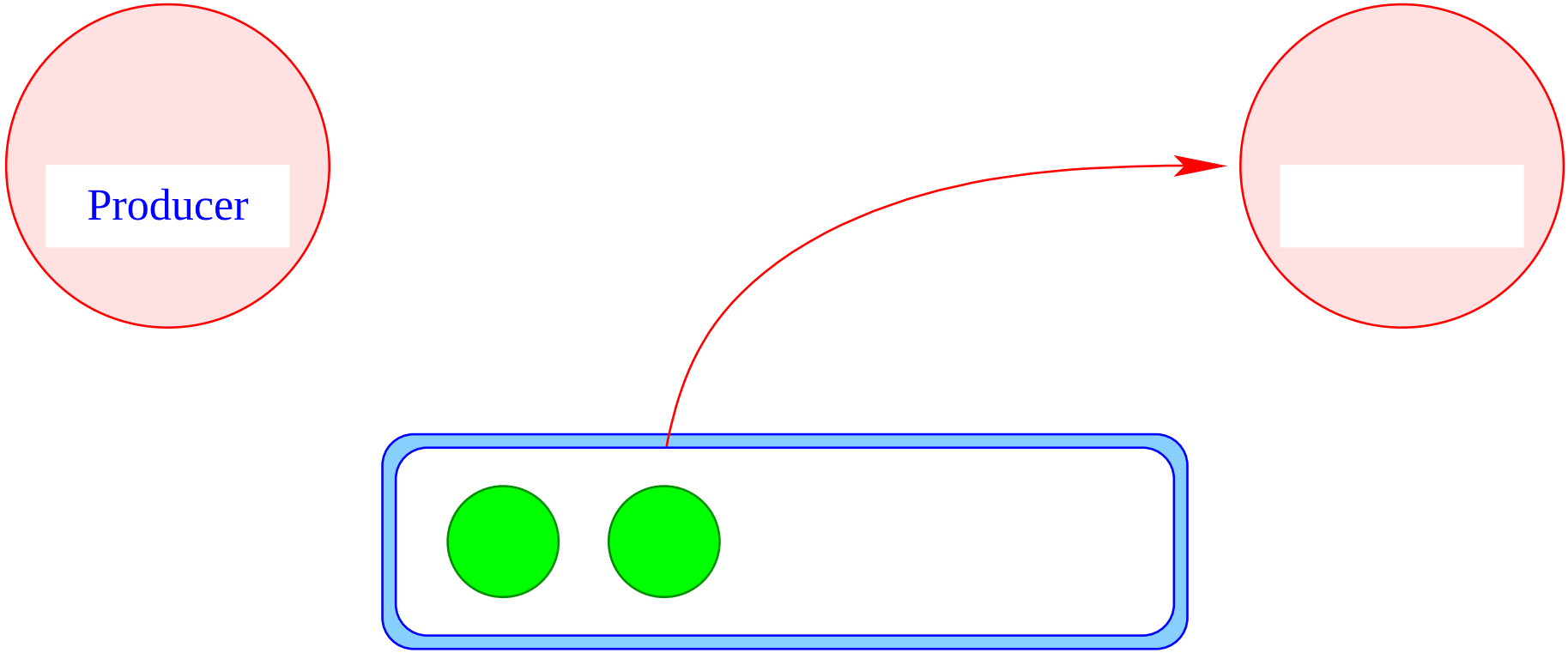
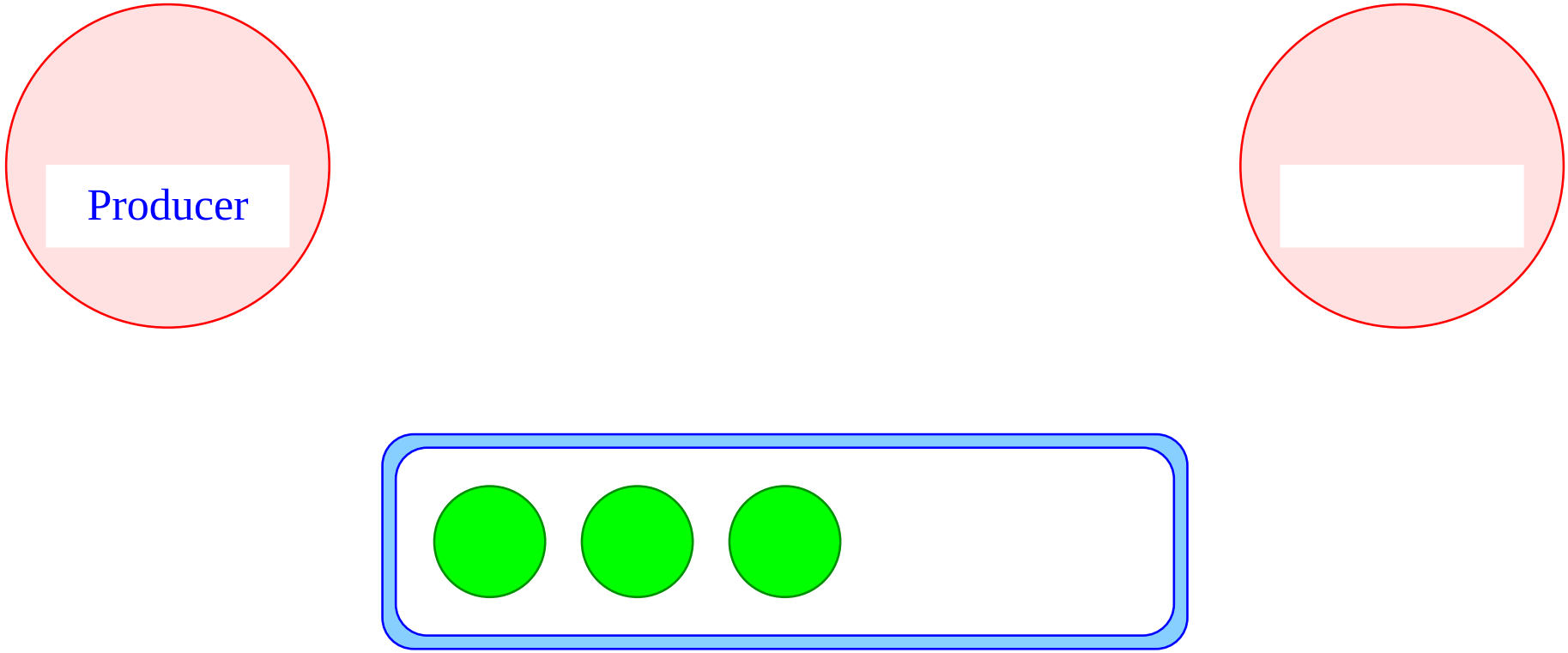
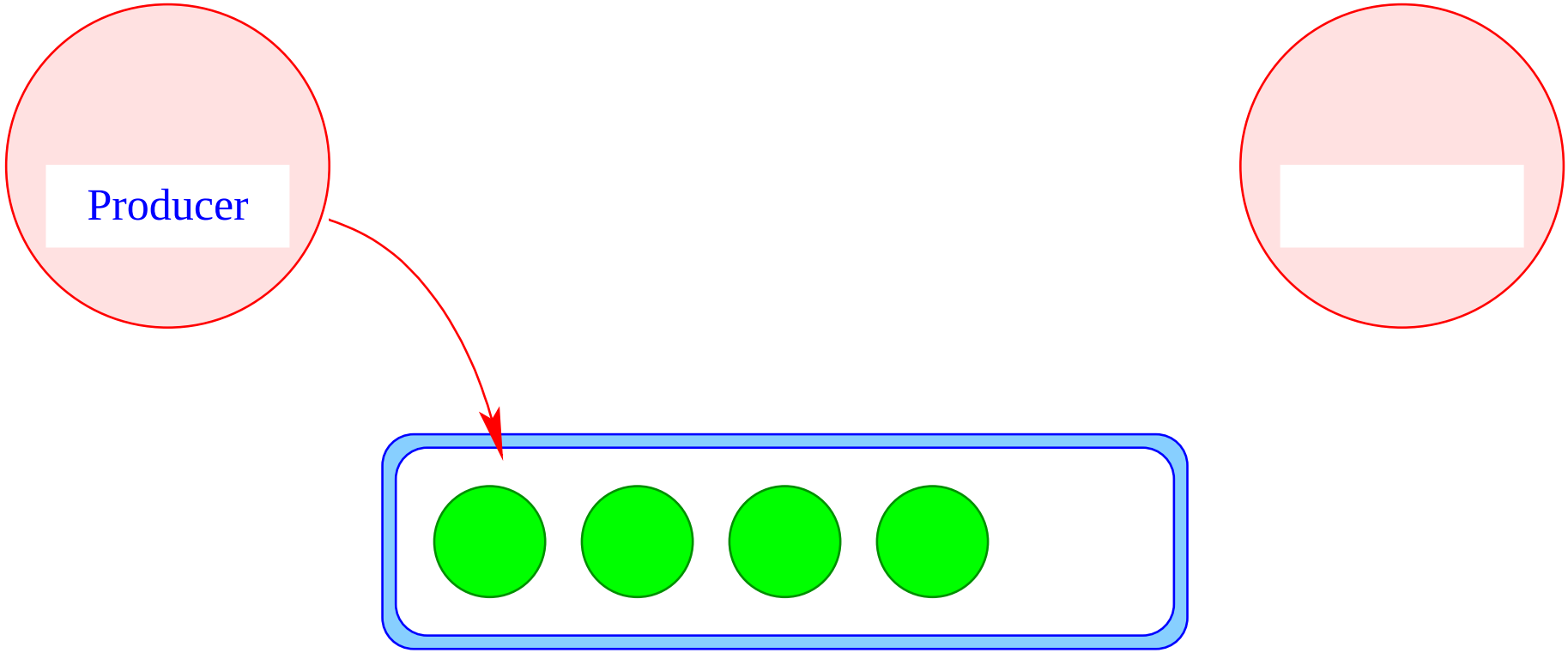
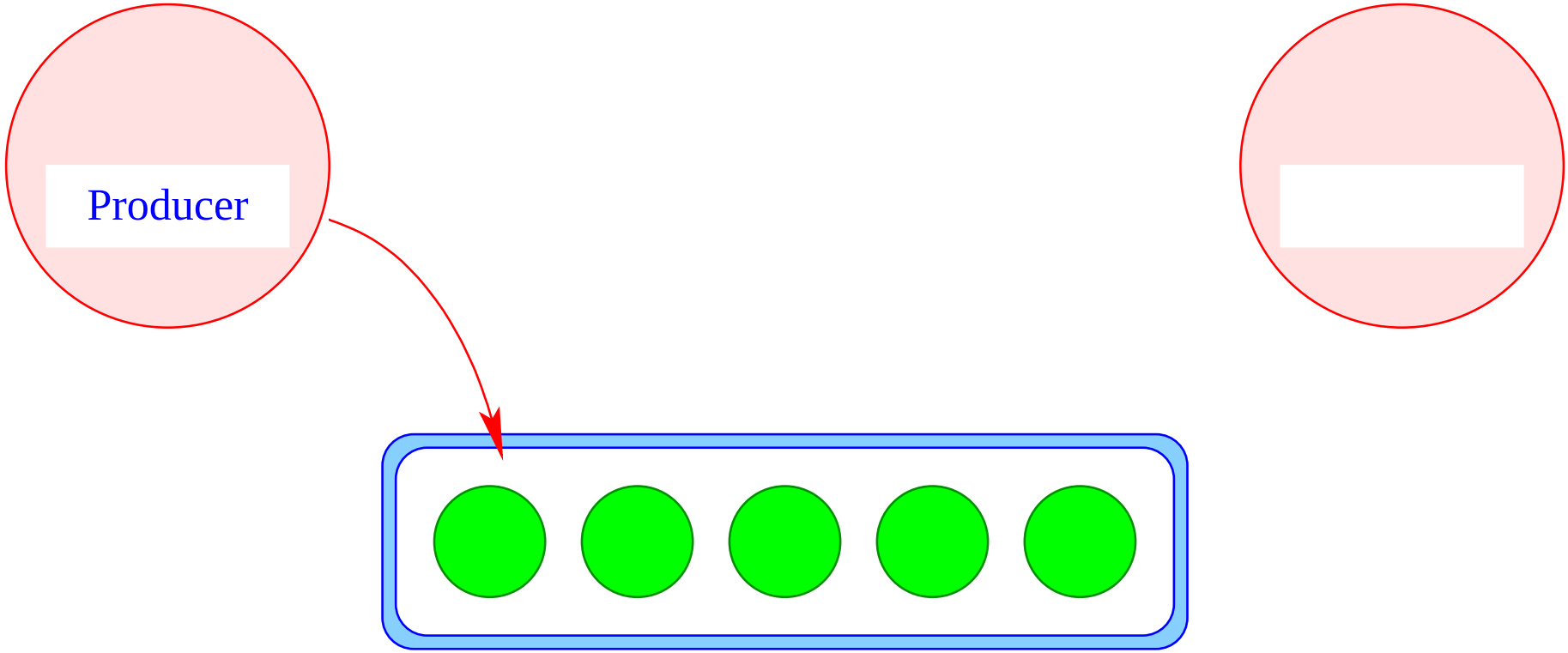
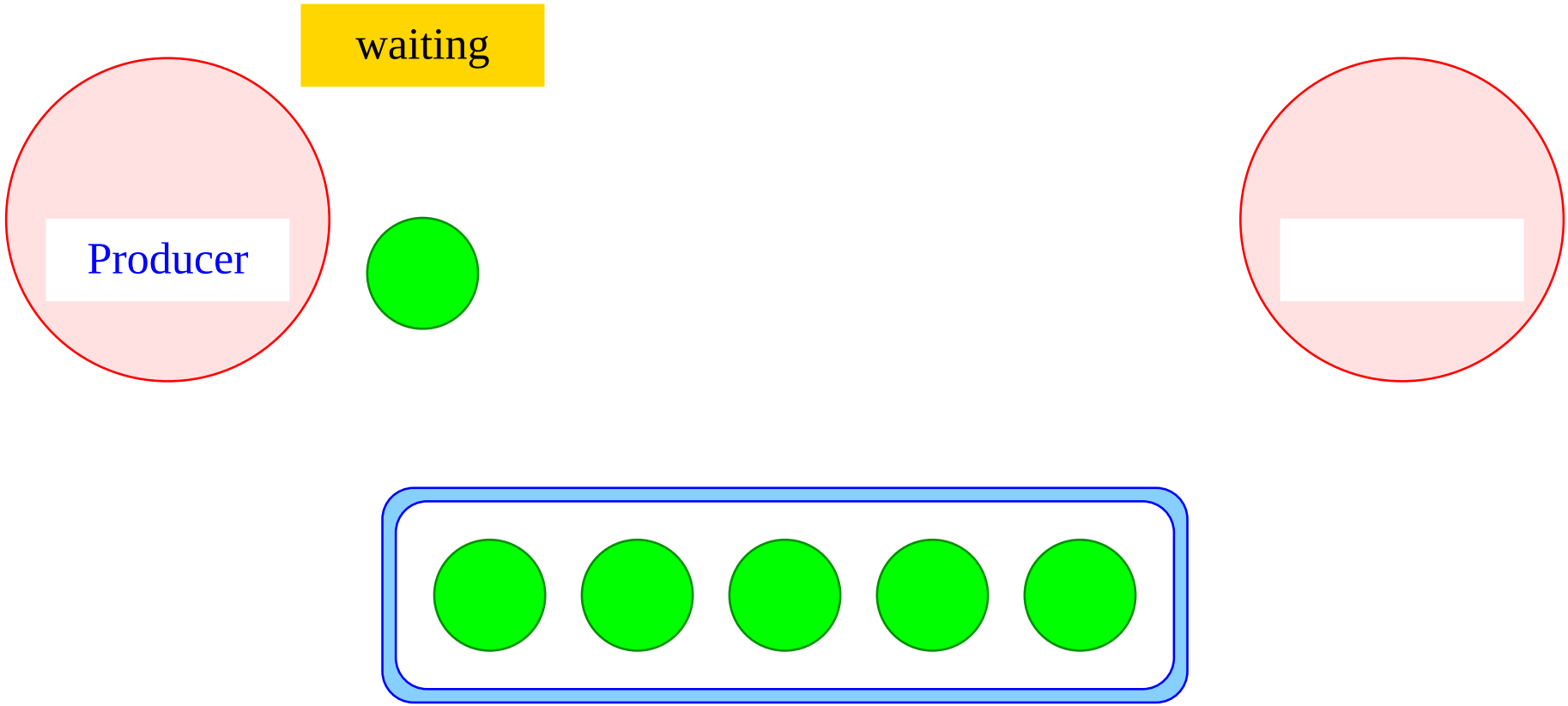
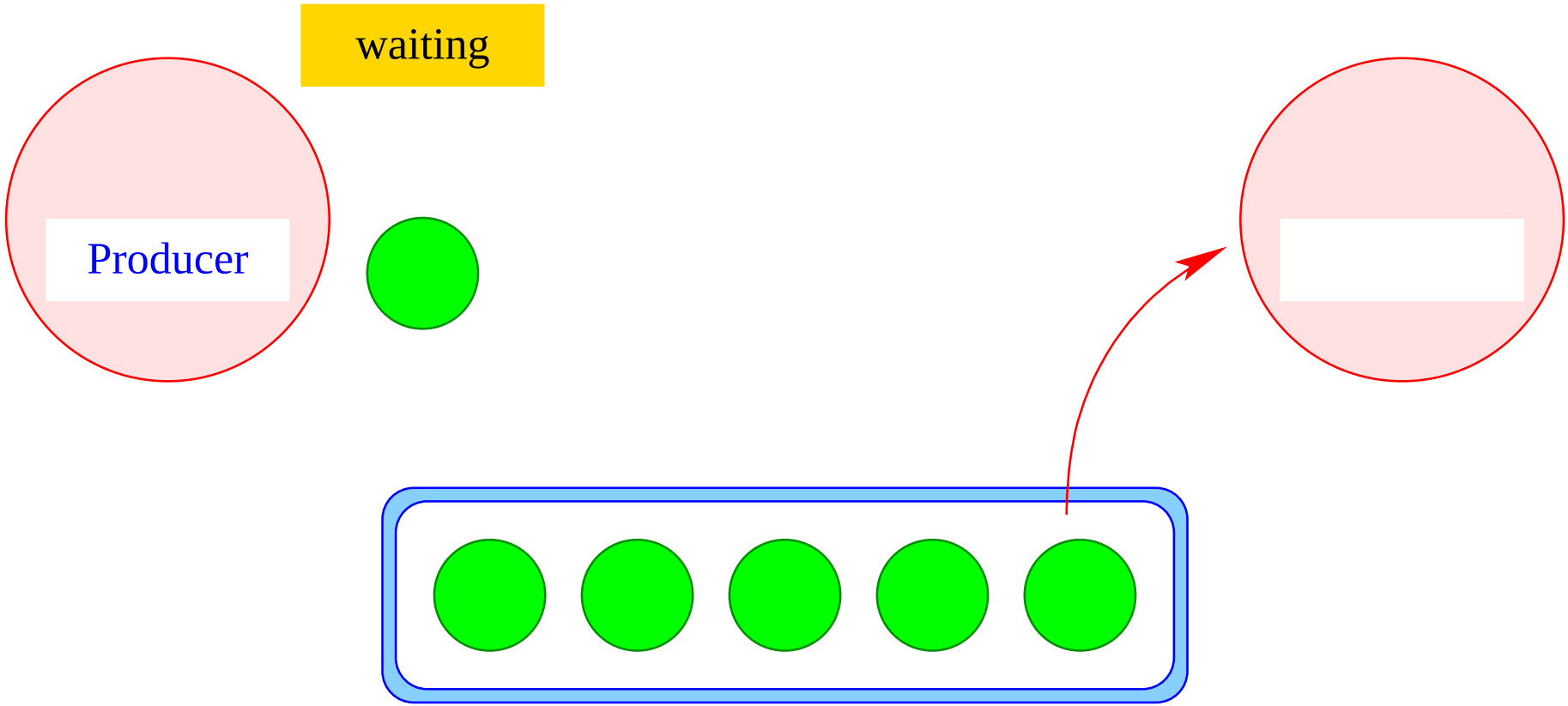
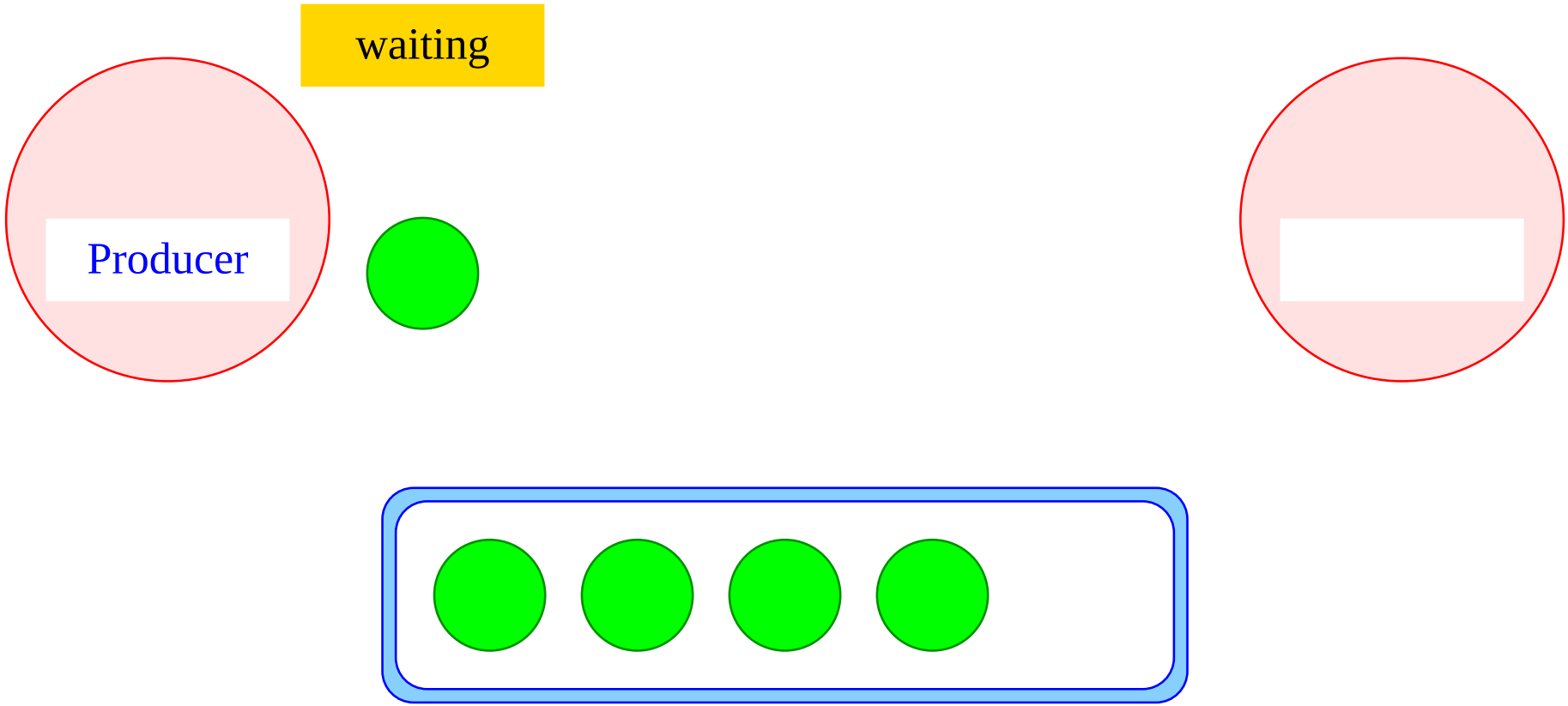
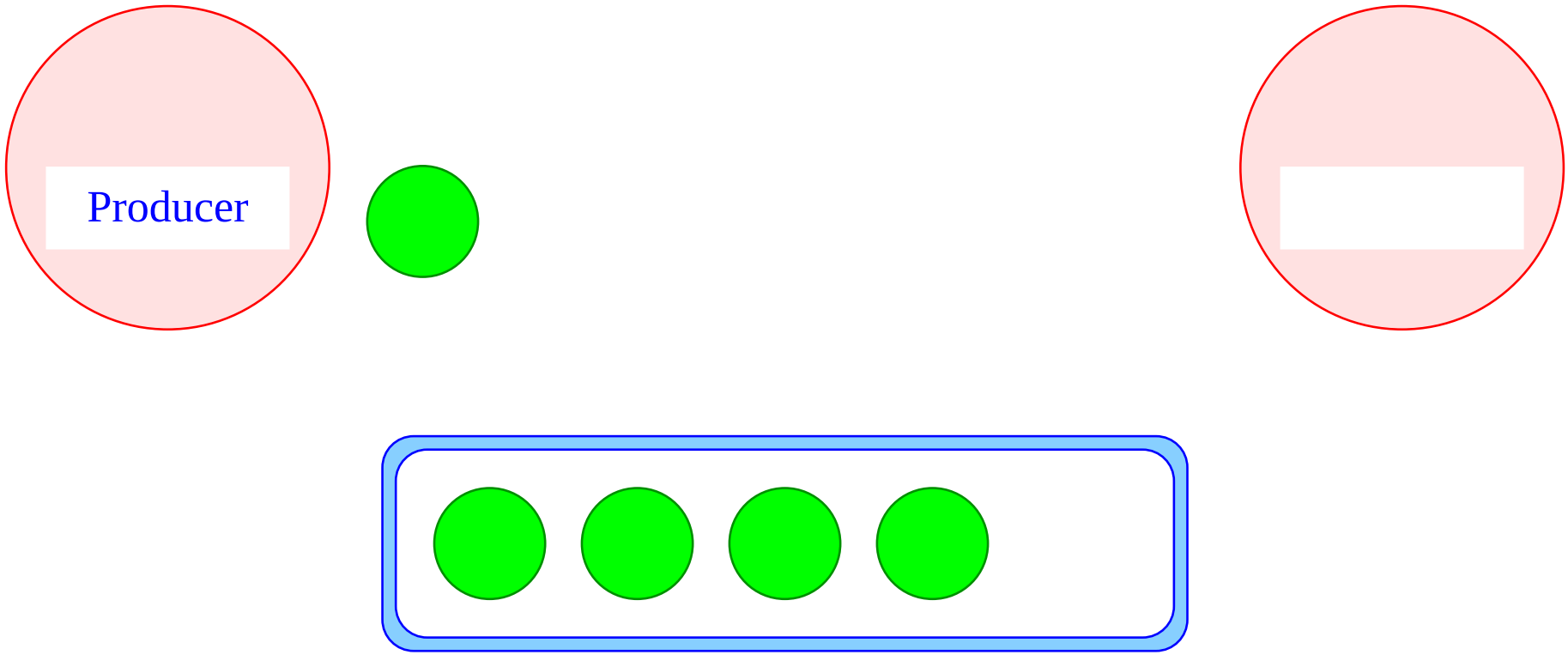
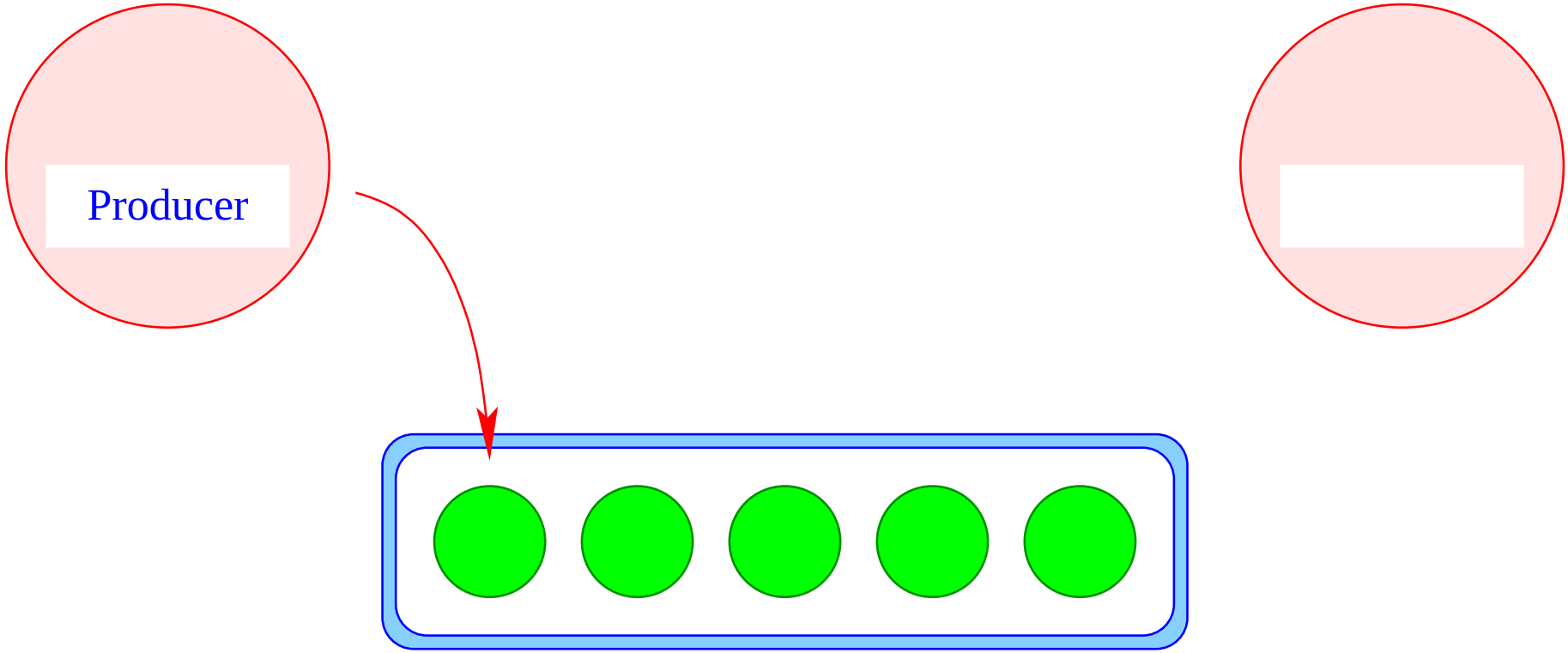
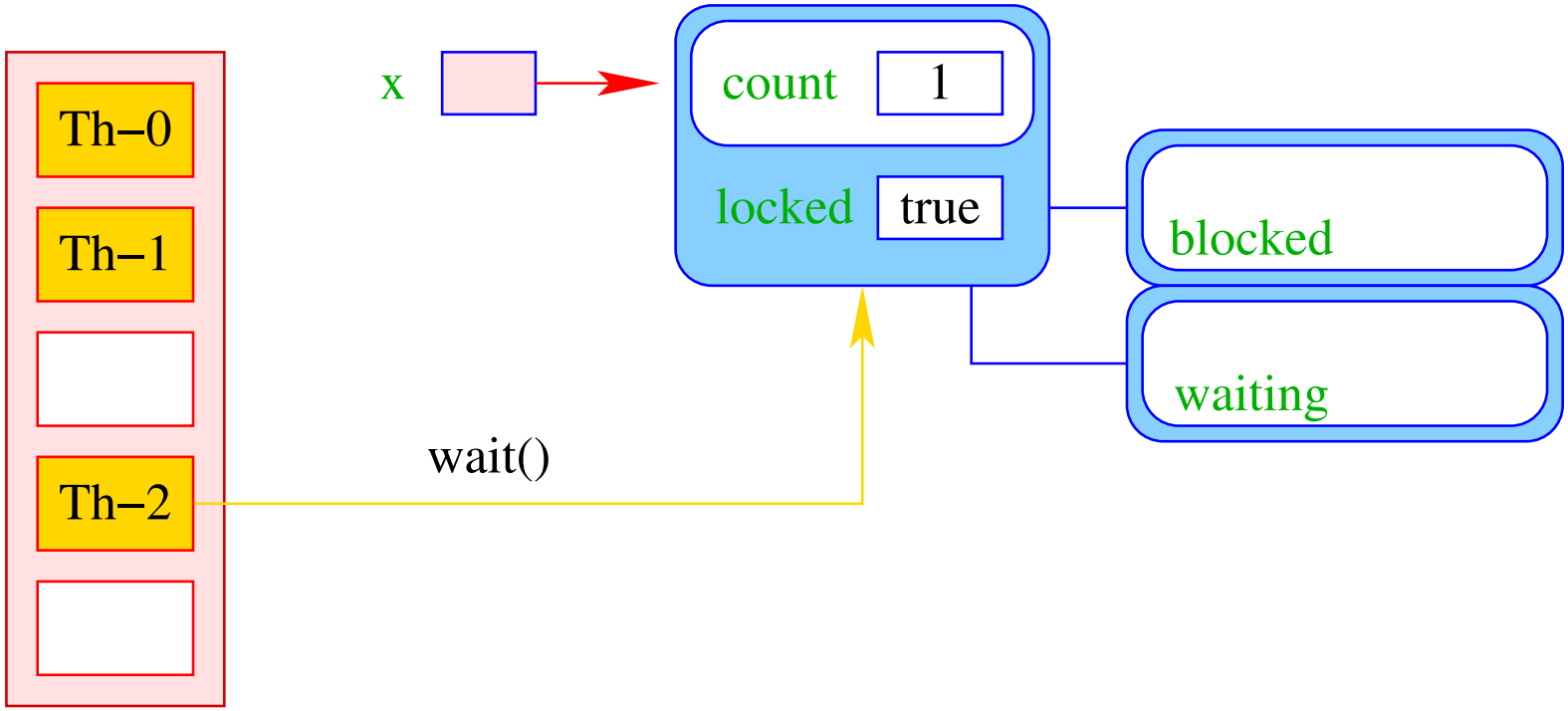
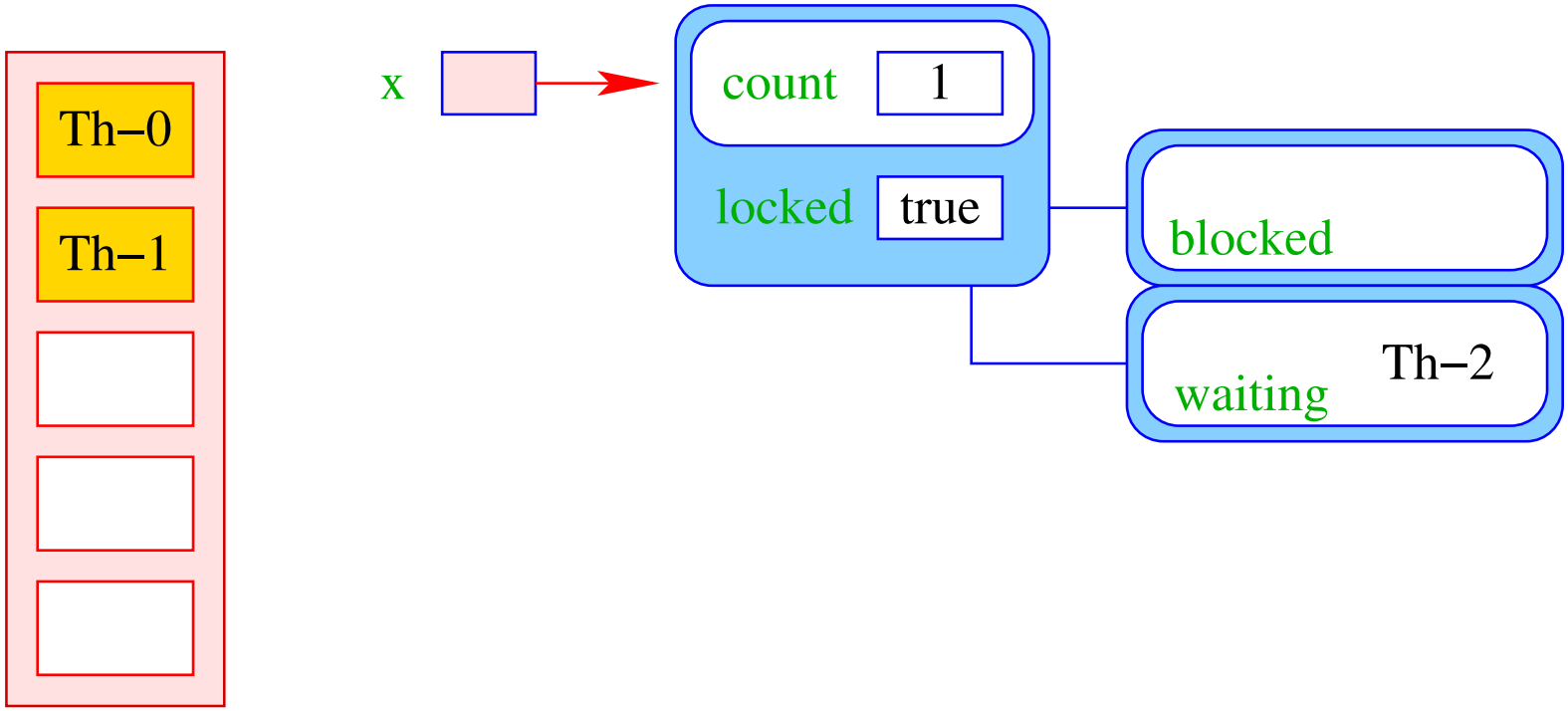
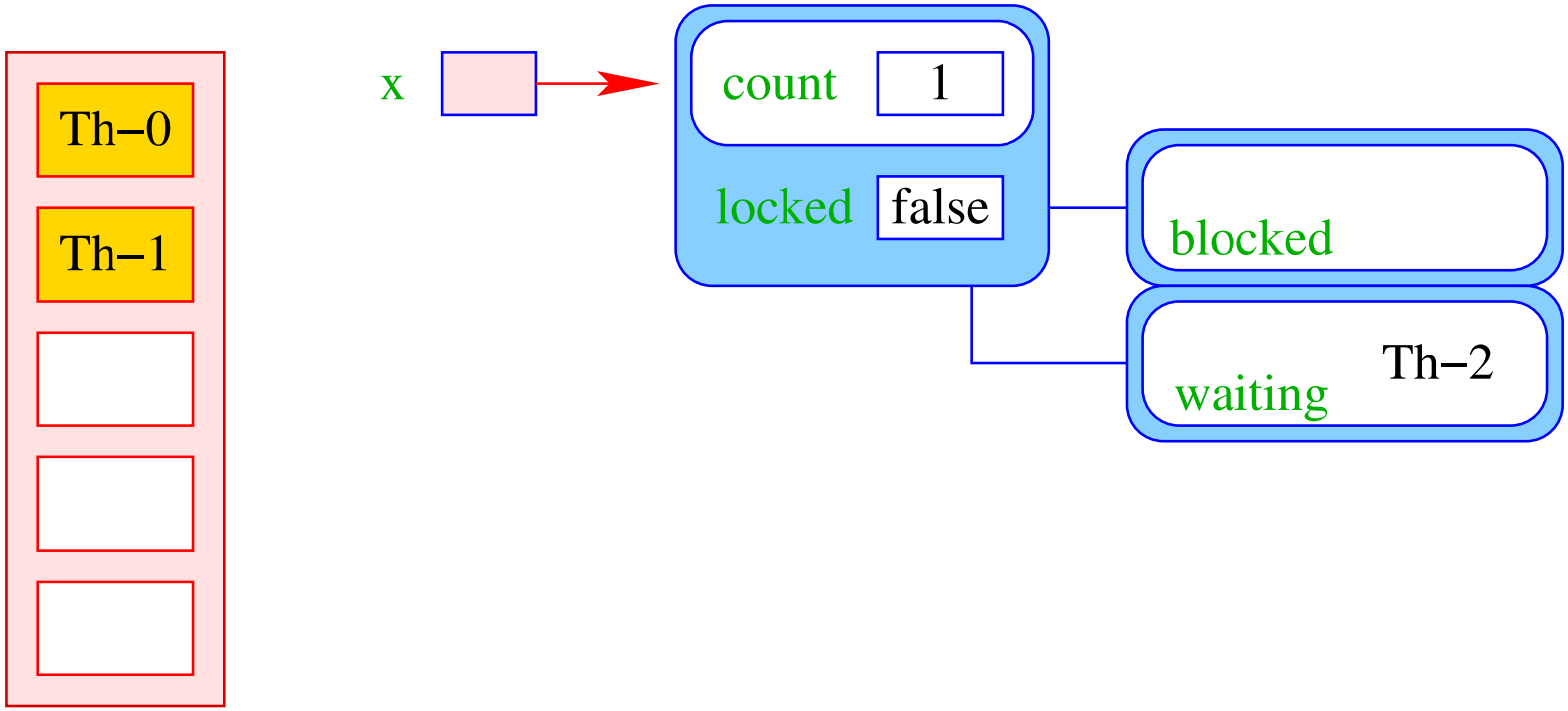
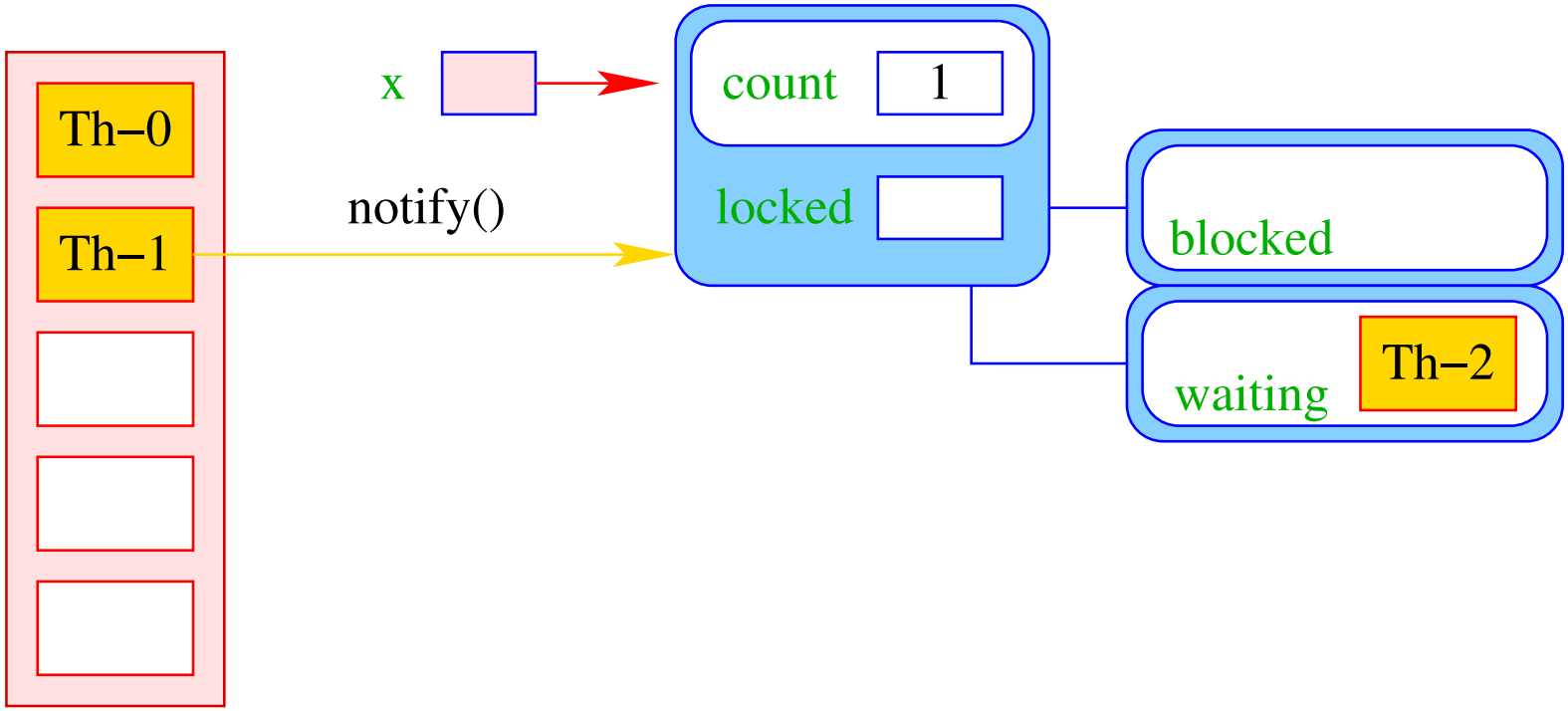
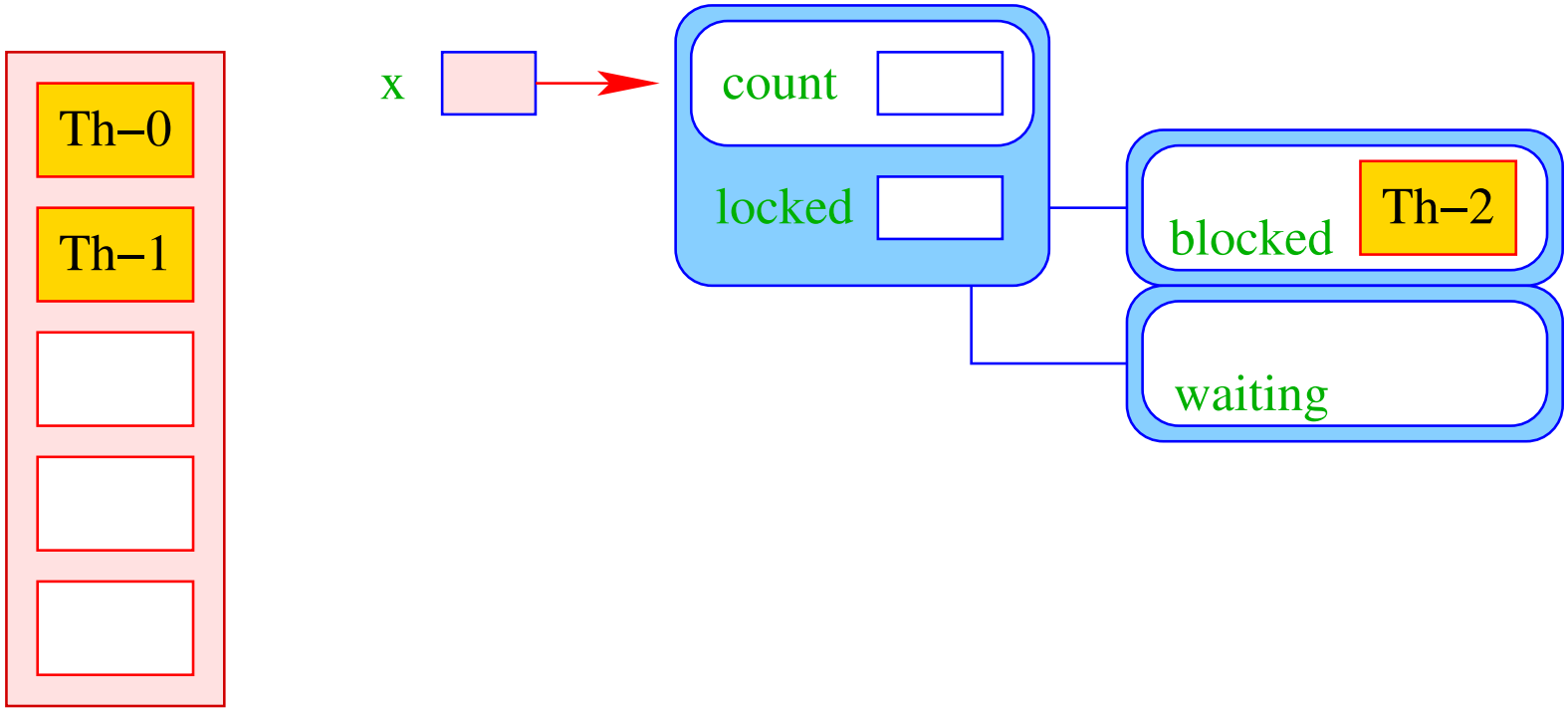
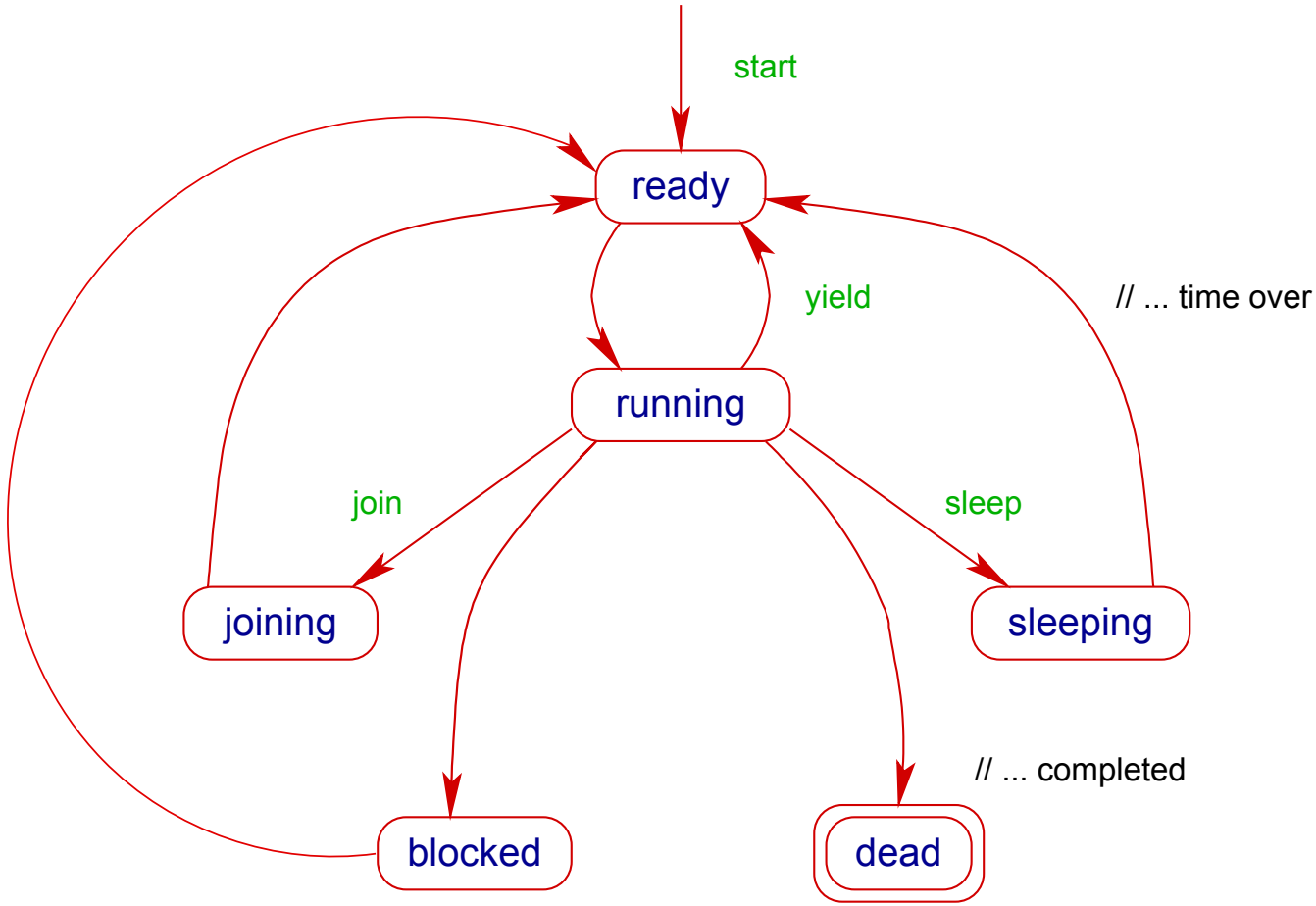
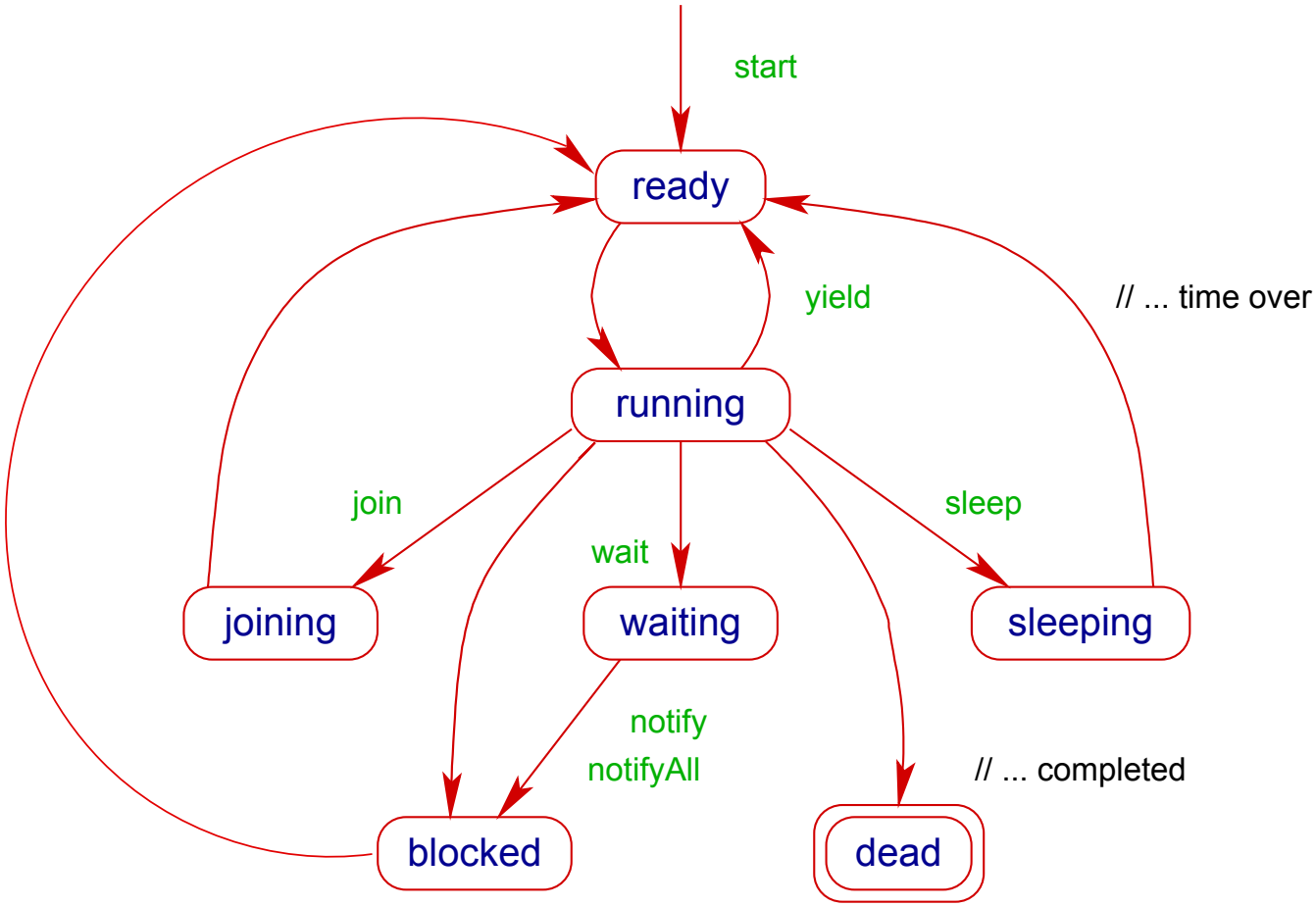
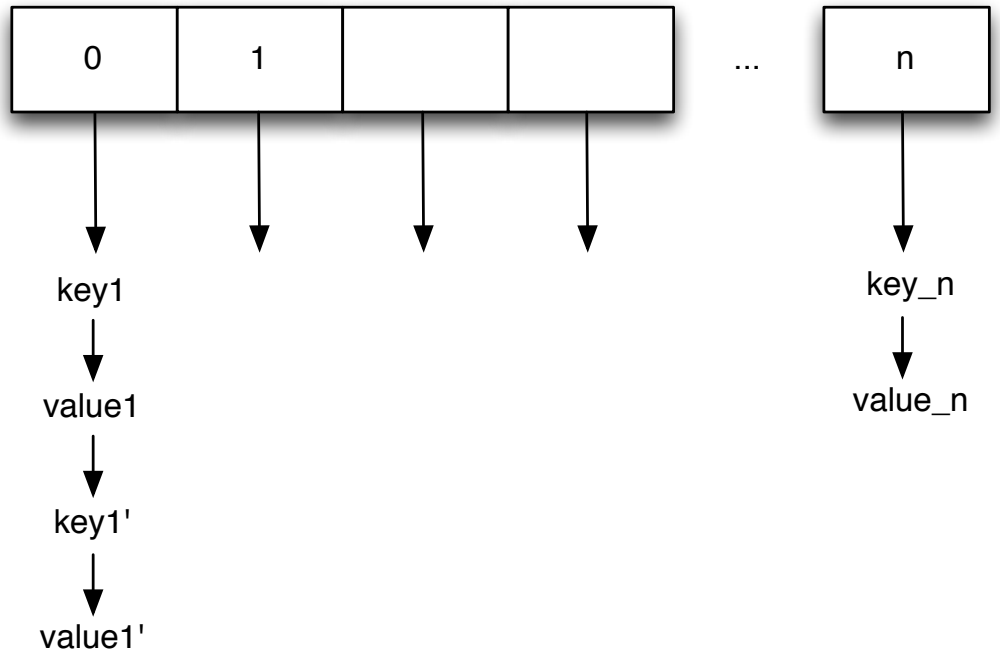
Position pos1 = Position.STURM;

System.out.println(pos1); // Ausgabe: STURM
```

Schema:

```
enum EnumName {
    Wert1, Wert2, Wert3, Wert4 , ...
}

EnumName variable = EnumName.EnumWert;
```

- enum-Werte sind Konstanten
- Einsatz von Enums, wenn eine feste Anzahl an Konstanten nötig ist (Wochentage, Himmelsrichtungen, Positionen, . . .)

Der Typ Enum (II)

Der Enum Typ Position wird vom Compiler in eine Klasse übersetzt, die in etwa so aussieht:

```
class Position extends Enum {
   // Enum-Werte als statische Variablen
   public static final Position TOR = new Position("TOR", 0);
   public static final Position ABWEHR = new Position("ABWEHR", 1);
   public static final Position MITTELFELD = new Position("MITTELFELD", 2);
   public static final Position STURM = new Position("STURM", 3);
   public static final Position RECHTSAUSSEN = new Position("RECHTSAUSSEN", 4);

   // Konstruktor
   private Position(String s, int i) {
      super(s, i);
   }

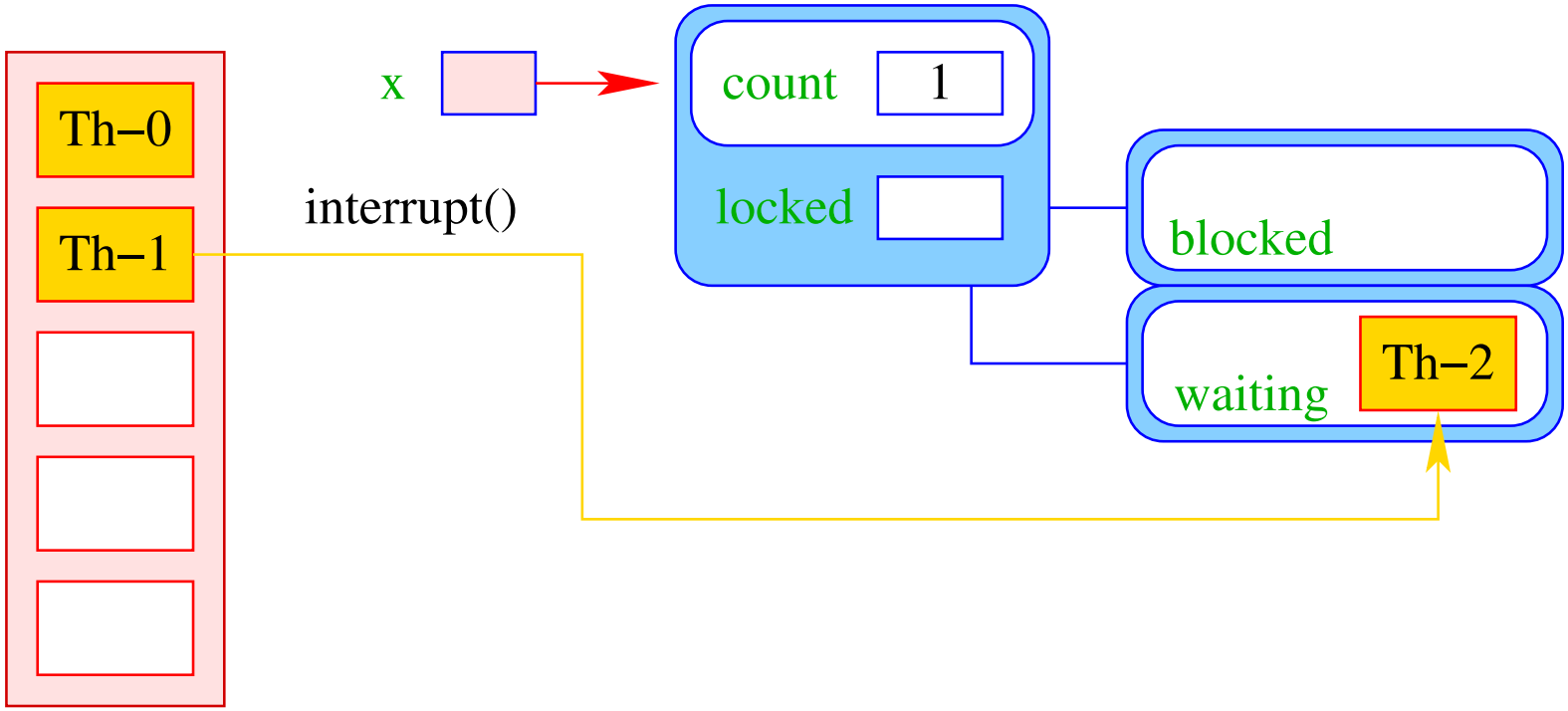
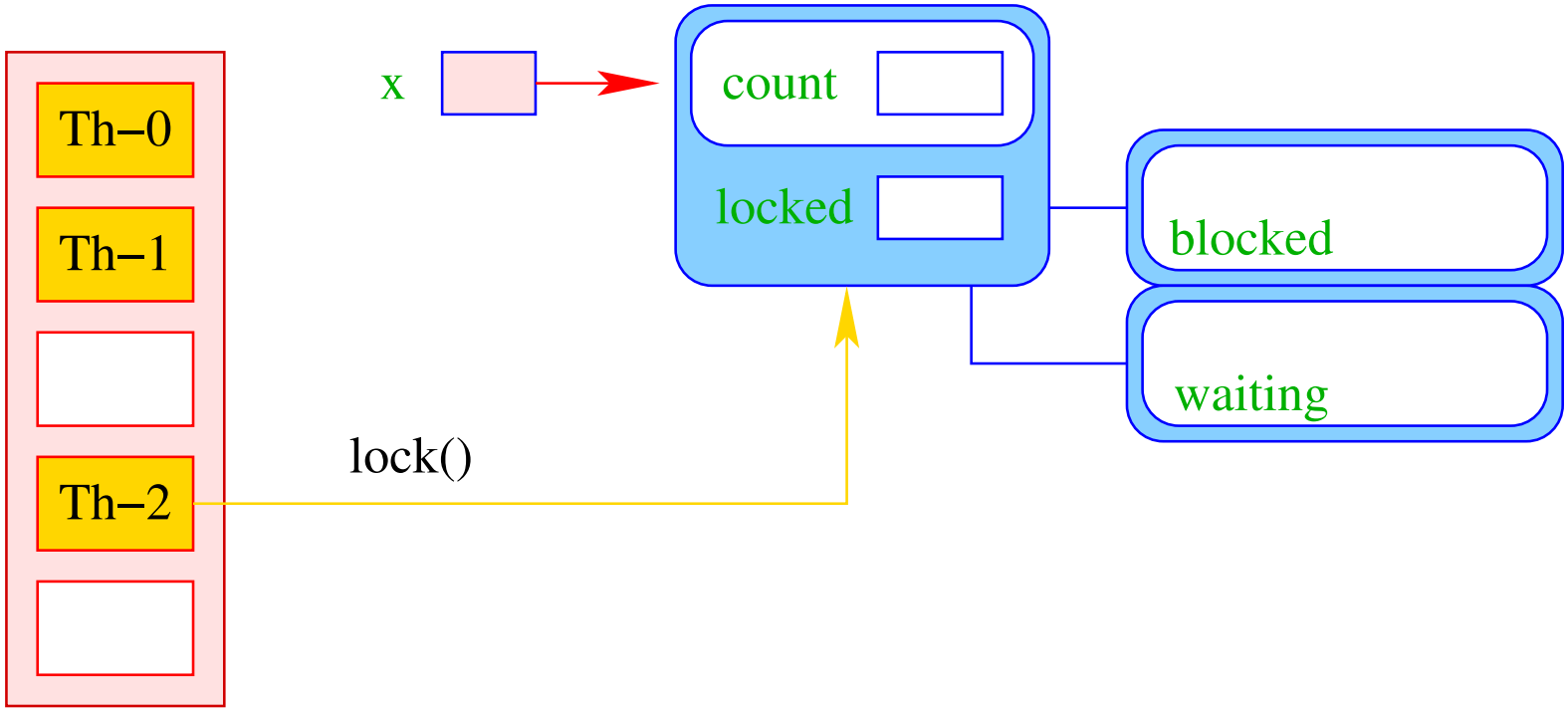
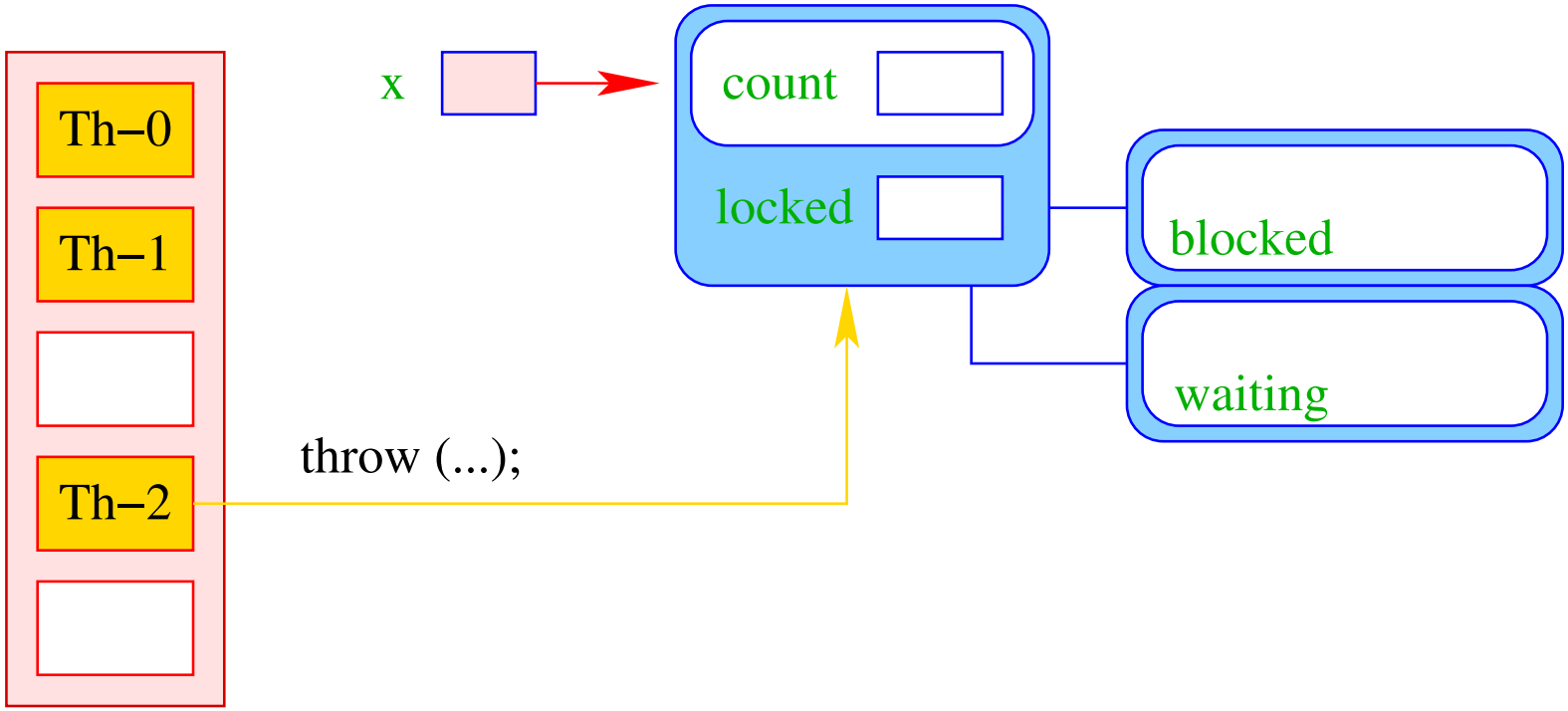
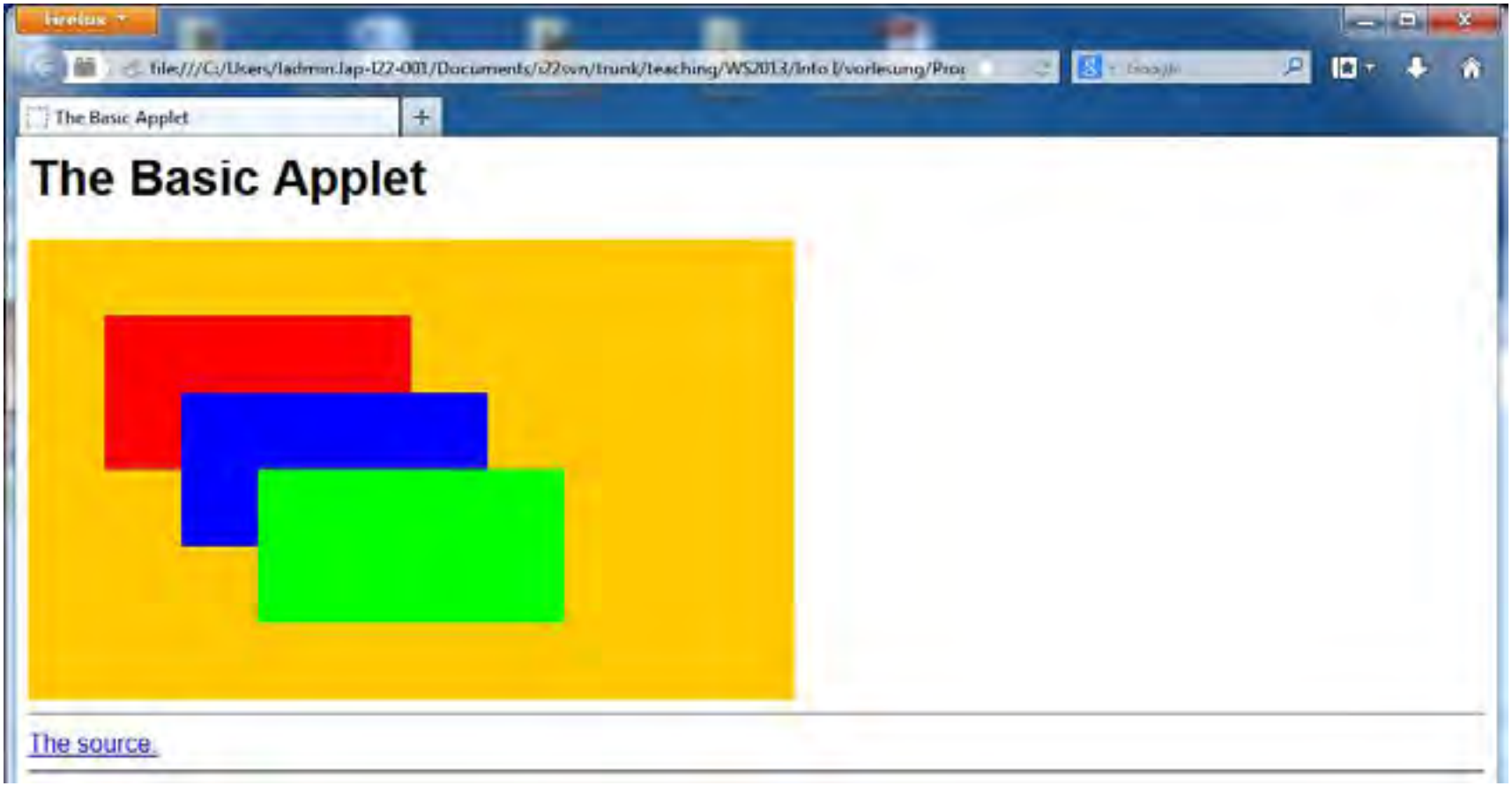
   // Methoden ...
}
```

Alles zusammen II: Beispiel Fußball





Alles zusammen II: Beispiel Fußball





Alles zusammen II: Beispiel Fußball







Maps: Wer spielt wo?



Alles zusammen III: Beispiel Fußball







Alles zusammen III: Beispiel Fußball



Alles zusammen III: Beispiel Fußball





Alles zusammen III: Beispiel Fußball



Übersicht

Ein- und Ausgabe

- Ein- und Ausgabe ist nicht Bestandteil von Java.
- Stattdessen werden (äußerst umfangreiche) Bibliotheken von nützlichen Funktionen zur Verfügung gestellt.
- Bisher haben wir die einfache `Terminal`-Klasse verwendet.

Ein- und Ausgabe

- Ein- und Ausgabe ist nicht Bestandteil von Java.
- Stattdessen werden (äußerst umfangreiche) Bibliotheken von nützlichen Funktionen zur Verfügung gestellt.
- Bisher haben wir die einfache `Terminal`-Klasse verwendet.

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



- Ein- und Ausgabe vom Terminal oder aus einer Datei werden als Ströme 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 bereit ...
- standardmäßig benutzt Java (zum Lesen und Schreiben) den Zeichensatz Latin-1 bzw. ISO8859_1.
- Diese externen Zeichensätze benötigen 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 ...
Zuerst eine (unvollständige) Übersicht ...

Übersicht


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

InputStream

- 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;      // int brauchen wir für -1
    while(-1 != (t = file.read()))
       System.out.print((char)t);
    System.out.print("\n");
  } // end of main
} // end of FileCopy
```

Erläuterungen

- 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
```

Beispiel

... 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 ...

Ausgabe

- 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.

Ausgabe

- 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
```

Erläuterungen

- 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
```

Erläuterungen I

- 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`.

Erläuterungen II

- 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

...

Nützliche Klassen



Reader

- 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.

Unterklassen

- 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.

Ausgabe

- 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.

Ausgabe

- 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 nun noch, 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.

Zusammenfassung

- Ein- und Ausgabe wird in Java durch diverse Varianten von Ein- und Ausgabeströmen realisiert
- Von diesen `InputStream`s und auf diese `OutputStream`s kann direkt gelesen/geschrieben werden. Das geschieht byteweise! (Und ja, wir haben gesehen, dass man byteweise bytes, chars, ints, etc. lesen/schreiben kann.)
- `Reader`s und `Writer`s arbeiten character-basiert. Der Unterschied liegt also in der unterstützten Codierung: Readers und Writers können direkt z.B. Unicode verstehen.

Übersicht

Testen

Jedes Programm muss getestet werden

Wichtig dabei:

- Auswahl der *Testfälle*
  - Klassifizierung der möglichen Eingabedaten
  - $\Longrightarrow$ Standardsituationen *und* Randfälle beachten
- Auswahl der *Testdaten*
  - Repräsentative Daten zu jedem Testfall

$\Longrightarrow$ Man muss versuchen, *alle* wichtigen Situationen abzudecken, in denen das Programm potenziell schiefgehen kann

Ein klassisches Quiz

- Der Benutzer gibt drei ganze Zahlen ein. Diese Zahlen repräsentieren die Länge der Seiten eines Dreiecks. Das Programm klassifiziert das Dreieck als gleichseitig, gleichschenklig, oder weder-noch.
- Was sind gute Testfälle?

Testfälle (Durchschnitt: 7.8/14)

- Gültiges weder-noch-Dreieck
- Gültiges gleichseitiges Dreieck
- Gültiges gleichschenkliges Dreieck (plus Permutationen, z.B. 4,3,3; 3,4,3; 3,3,4)
- Genau eine Seite Länge Null
- Eine Seite negative Länge
- Summe zweier nichtnegativer Seiten ergibt dritte Seite (plus Permutationen)
- Summe zweier nichtnegativer Seiten ist kleiner als dritte Seite (plus Permutationen)
- Alle Seiten Länge Null
- Nicht-ganzzahlige Werte
- $\neq$ 3 Parameter
- Erwartete Ausgabe?

16 Testfälle!

Was Testen ist ...

- Dynamischer Vergleich von tatsächlichem und erwünschtem (=spezifiziertem) Programmverhalten
- Ziele
  1. Nachweis, dass Funktionalität erfüllt ist
  2. Fehlerdetektion
- Unterschied psychologisch wesentlich
  - Sollten Programmierer ihre eigenen Programme testen?
  - Im Allgemeinen nein; aber das ergibt organisatorische und Kostenprobleme

Ein Testfall besteht immer aus Ein- und erwarteter Ausgabe (plus ggf. Ausführungsbedingungen). Für jeden Testfall sollte auch eine Begründung existieren, warum er ausgewählt wurde.

In XP sind Testfälle die (einzige) Spezifikation!

Drei Arten von Fehlern

1. Failure
   - Unterschied zwischen Spezifikation und beobachtbarem Verhalten
   - Manifestation zur Laufzeit
   - Wenn Testfälle als eigenständiges Programm implementiert werden, kann der Vergleich von erwarteter und tatsächlicher Ausgabe als Assertion formuliert werden. Gilt auch für einzelne Funktionen.
   - Gute Frameworks, etwa junit, verfügbar.
2. Error
   - Unterschied zwischen tatsächlichem und erwünschtem internen Programmzustand
   - Kann, muss aber nicht zu einer Failure führen
   - Kann zur Laufzeit „repariert“ werden (z.B. durch Redundanz)
   - Kann mit Assertions überprüft werden
3. Fault
   - Tatsächlicher oder vermuteter Grund für die Abweichung von tatsächlichem und erwünschtem beobachtbaren Verhalten oder dem Programmzustand

Was Testen nicht ist ...

- Qualitätsverbesserung
  - Das wird durch Debugging erreicht
- Fehlerlokalisierung (fault localization)
- Statische Codeanalyse, ob manuell oder automatisiert
  - Mathematische Beweise
  - Manche nennen verschiedene Lesetechniken (reviews, inspections, walkthroughs) „statisches Testen“. Wir nicht.

Warum Testen so schwierig ist

Was macht einen Testfall zu einem guten Testfall?

1. Fähigkeit, Fehler zu finden (Fehler nur mit Spezifikationen!)
   - Dann aber kein guter Testfall für ein fehlerfreies Programm
2. Fähigkeit, potentielle Fehler zu finden
   - Richtig. Aber was ist ein potentieller Fehler? Und was ist mit dem Aufwand?
3. Fähigkeit, wahrscheinliche Fehler mit angemessenem Aufwand zu finden
   - Kosten, Tests zu schreiben/auszuführen/auszuwerten
   - Kosten verbleibender Fehler, je nach Schwere
   - Testsuiten sollen aus Managementgründen so klein wie möglich (so gross wie nötig) sein und so kurze Testfälle wie möglich (so lang wie nötig) enthalten
   - Leichte Fault-Lokalisierung

Ausgezeichnet! Und viel zu abstrakt!

Testselektion

- Approximation „guter“ Testfälle
- Zentrales Problem
  - Mögliche Eingabedaten in Blöcke gruppieren, so dass (möglichst) jedes Element eines Blocks denselben Fehler provoziert
  - (Erfordert natürlich a priori–Wissen um Fehler)
- Lösung
  - Auf der Basis von Anforderungen
  - Auf der Basis von Wissen um typische Fehler
  - Zufällig (gleichverteilt oder gemäss Nutzerprofilen)
  - Strukturbasiert („jede Zeile einmal ausführen“, „jede Bedingung einmal zu wahr und einmal zu falsch auswerten“ usw.)

Wann sind wir fertig?

- Wenn die Zeit um ist
  - Nutzlos: Einfach gar nichts tun. In der Praxis Kriterium # 1.
- Wenn keine weiteren Fehler mit existierenden Tests gefunden werden
  - Nutzlos. Wähle irrelevante Tests
- Wenn ein strukturelles Kriterium erfüllt ist
  - Etwa: Jede Zeile einmal ausgeführt
  - Nützlich, wenn es einen Zusammenhang zwischen diesen strukturellen Kriterien und Fehlerdetektion gibt. (Dafür gibt es aber keine empirische Evidenz und gute Gründe, dass es dafür keine Evidenz gibt)
- Wenn eine vorher spezifizierte Anzahl Fehler gefunden wurde
  - Grossartig. Erfordert aber gute Abschätzungen

Warum Testen so schwierig ist II

- Erwartete Ausgabe ist essentiell. Also benötigen wir eine Spezifikation
  - Möglicherweise nur „keine Exception“: robustness testing. Dann aber nur ein sehr oberflächlicher Test der Funktionalität
- Wir wissen nach wie vor nicht, was gute Testselektionskriterien sind
  - Aus gutem Grund. U.a. das empirisch nachzuweisen, ist sehr schwierig

Anwendungsbasiertes Testen

- Datenbasiert, auf Basis der Eingabeparameter
- Kontrollflussbasiert, auf Basis angenommener „typischer“ oder „relevanter“ Interaktionen mit dem Programm (also Programmabläufe).
  - Betrachten wir nicht.

Datenbasiertes Testen: Beispiel



Datenbasiertes Testen: Beispiel



Datenbasiertes Testen: Beispiel



Category Partition Method

1. Identifiziere individuelle funktionale Einheiten, die separat getestet werden können. Bestimme für jede Einheit die das Verhalten der Einheit beeinflussenden Parameter und relevante Umgebungsvariablen.
   - Diese Parameter und Umgebungsvariablen definieren die „Kategorien“.
2. Bestimme die individuelen Wahlmöglichkeiten für jeden Parameter und jede Umweltvariable.
   - Das „partitioniert“ die „Kategorien“.
3. Bestimme Abhängigkeiten und Constraints zwischen den Wahlmöglichkeiten.
4. Erzeuge alle relevanten, die Constraints erfüllenden, Kombinationen von Werten für die Kategorien.
5. Transformiere diese Werte in ausführbare Testfälle incl. der erwarteten Ausgaben.

Grenzwerttesten

Wenn die Datentypen der Kategorien eine natürliche Ordnung mit Grenzen aufweisen, dann ergeben diese Grenzen (und Werte „leicht links“ und „rechts davon“) zusätzliche natürliche Kandidaten für Blöcke der partitionierten Kategorie.

- Beispiel natürliche Zahlen: -1, 0, 1 (und irgendwelche positiven Werte)
- Beispiel Liste: leere Liste (null), einelementige Liste (und diverse nicht-leere Listen)
- Beispiel Array: Länge Null, Länge 1 (und diverse größere Arrays)
- Beispiel String: null, Länge Null, Länge 1 (und größere Längen)
- Beispiel Dreiecke: s. frühere Folie
- ...

Diese Art des Testens beruht auf einem intuitiven Fehlermodell: An den Grenzfällen machen Programmierer Fehler.

Beispiel

Kategorien für das Testen eines Sortieralgorithmus

- Länge (null, 0, 1, > 1)
- Sortiertheit (aufsteigend, absteigend, identische Werte, nein)
- Einschluss negativer Elemente (einige negative, alle negativ, alle positiv)

Hausaufgabe: Was sind gute Testfälle für unsere Suchmaschine?

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

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



Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



Analyse

... 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

Warum ist Tic-Tac-Toe interessant?

- Analyse nichttrivial
- Eins der ersten Video-Spiele der Welt! (1952)



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.
- Nullsumme: Summe Gewinne und Verluste ist Null.

... im Beispiel

Konfiguration:



End–Konfigurationen:



Gewinn –1



Gewinn 0



Gewinn +2

Die Konfigurationsbewertungen 0, -1 und +2 sind rein zufällig gewählt (damit sind die Betrachtungen allgemeingültig für viele andere Spiele)!
Diese Zahlen sind nicht Gewinne oder Verluste!

Konfigurationswechsel

Spiel–Zug:



Allgemein: Spielverläufe als Baum



Wiederum sind die Zahlen in den Blättern abhängig vom konkreten Spiel! Bei TicTacToe würden -1, 0, +1 reichen —
und der Baum wäre viel größer!

Interpretation

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 und wollen einen möglichst kleinen (negativen) Wert erzielen. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

  Fall 2: Die Konfiguration ist rot: der Gegner ist am Zug und will einen möglichst großen (positiven) Wert erzielen. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Propagation der Gewinne



Propagation der Gewinne



Propagation der Gewinne



Propagation der Gewinne



Propagation der Gewinne



Strategien

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, wann immer das möglich ist ...

Spielverlauf



Strategien



Strategien



Strategien

Konkreter für Tic-Tac-Toe:

- +1, 0, -1 reichen!
- Das Maximum (bzw. Minimum) für beide Spieler ist 0
- D.h. es ist für keinen Spieler möglich, garantiert zu gewinnen
- Der Baum ist riesig!

Baum Tic-Tac-Toe



$$\text{Anzahl Knoten} = 9! + \frac{9!}{2!} + \frac{9!}{3!} + \cdots + 9 + 1 \sim 6,3 \cdot 10^5 \sim 2^{19}$$

Kodierung Tic-Tac-Toe

Koordinaten

0	1	2
3	4	5
6	7	8

Mögliche Spielkonfiguration

0	-1	1
0	1	0
0	-1	0

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`.

Idee der Implementierung

- Alternierend werden `MyChoice`- und `YourChoice`-Knoten erzeugt, d.h. die Spielerin beginnt. Das geschieht durch wechselseitige Aufrufe der Konstruktoren (bis maximal Tiefe 9)
- `new MyChoice(new PlayGround(), x)` liefert die Menge der gewinnenden oder unentschiedenen Teilbäume (Wurzel: Konfigurationen des Programms, das jetzt am Zug ist), falls die Spielerin mit Feld x begonnen hat
- `new YourChoice(new PlayGround(), y)` liefert die Teilbäume (Wurzel: Konfiguration der Spielerin, die jetzt am Zug ist), falls das Programm mit Feld y begonnen hat

Implementierung YourChoice

```
public class YourChoice implements PlayConstants {
   // implementiert alle möglichen Reaktionen der Spielerin (blau)
   // an den Konstruktor wird ein Brett übergeben sowie die Stelle, auf
   // die das Programm (ME) gesetzt hat
   // darauf aufbauend werden alle Kindkonfigurationen berechnet und bewertet
   // diese Kindkonfigurationen sind per answer(int _) abfragbar
   private PlayGround ground;  //aktuelle Konfiguration mit letztem Zug des Programms (ME)
   private int value;          //bestmögliches (negatives) Ergebnis
   private MyChoice[] answer;  //Verweise auf Folgekonfigurationen der Spielerin (YOU)
   public YourChoice(PlayGround g, int place) {
      ground = new PlayGround(g,place,ME);  //Programm (ME) besetzte zuletzt Feld place
      answer = new MyChoice[9];
      value = ME;               //schlimmstenfalls gewinnt das Programm!
      PossibleMoves moves = new PossibleMoves(ground);  // mögliche Züge der Spielerin (YOU)
      for(int choice = moves.next(); choice!=-1; choice = moves.next()) {
         if (ground.won(choice,YOU)) { //sofortiger Sieg!
           value = YOU;
           continue;           // verlasse aktuellen Schleifendurchlauf
         }
         answer[choice] = new MyChoice(ground,choice);  // indirekte Rekursion!
         int win = answer[choice].value();
         if (win < value) value = win;      // Wert dieses Knotens (this) ist das Minimum
                                            // der Werte der Kinder
      }
   }
   public int value() { return value;}
   public MyChoice answer(int place) {
      return answer[place];
   }
}  // end of class YourChoice
```

Erläuterungen I

- 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 1).
- Die Klasse `PlayGround` repräsentiert Spielbretter.
- Das aktuelle Spielbrett wird aus demjenigen des Vater-Knotens und dem von `ME` gewählten Zug gestimmt.

Erläuterungen II

- `MyChoice answers[choice]` enthält den Teilbaum für den möglichen 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();` liefert den Spielwert des aktuellen Spielbaums.
- `public MyChoice answer(int place);` liefert den Spiel-Teilbaum für den Zug `place`.

Implementierung MyChoice

```
public class MyChoice implements PlayConstants {
   // versucht, eine unentschiedene oder gewinnende vom Programm (ME) erreichbare
   // Folgekonfigurationen zu berechnen. Falls die existiert, wird sie in yours gespeichert
   // Übergabe eines Spielfelds und des letzten Zugs der Spielerin (YOU)
   private PlayGround ground;
   private YourChoice yours;              // nächste Konfiguration der Spielering (YOU)
   private int acceptableChoice, value;
   // letzter Zug der Spielerin (YOU) fand statt an Position place in Spielfeld g
   public MyChoice(PlayGround g, int place) {
      ground = new PlayGround(g,place,YOU);  // setze Stein der Spielerin
      if (ground.won(place,YOU)) {  // Spielerin hat gewonnen
       value = YOU;
       return;                      // keine indirekte Rekursion
      }
      acceptableChoice = ground.force(ME);
      if (acceptableChoice != -1) { // kann Programm in einem Zug gewinnen?
        value = ME;
        return;                      // keine indirekte Rekursion
      }
      acceptableChoice = ground.force(YOU);
      if (acceptableChoice != -1) { // könnte Spielerin in einem Zug gewinnen?
         // dann das entsprechende Feld besetzen
         yours = new YourChoice(ground,acceptableChoice);  // indirekte Rekursion
         value = yours.value();
         return;
      }
      ...
```

Implementierung MyChoice II

```
      // Spielerin hat nicht gewonnen; kann nicht direkt im nächsten Schritt gewinnen
      // und das Programm kann ebenfalls nicht in diesem Schritt gewinnen
      // Dann alle Möglichkeiten ausprobieren
      PossibleMoves moves = new PossibleMoves(ground);
      int tmp = moves.next();
      if (tmp == -1) {
         value = NONE;    // keine weiteren Züge: unentschieden
         return;          // keine indirekte Rekursion
      }
      value = YOU;        // schlimmster anzunehmender Fall: Spielerin gewinnt
      do {
         acceptableChoice = tmp;
         // indirekte Rekursion: Aufruf von YourChoice, das MyChoice aufrufen wird
         yours = new YourChoice(ground,acceptableChoice);
         if (yours.value() > YOU) { // falls unentschieden oder Programm gewinnt
            value = yours.value();
            return;                 // keine weitere indirekte Rekursion
         }
         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
```

Erläuterungen

- 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 ...

Akzeptable Strategien

- 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.

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();
   }
   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
```

Die Klasse Game I

- `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`.

Die Klasse Game II

- `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.

Und noch die Klasse PlayGround I

```
public class PlayGround implements PlayConstants {
   private int[] arena;
   public PlayGround() {
      arena = new int[9];
   }
   public PlayGround(PlayGround ground, int place, int who) {
      arena = (int[]) ground.arena.clone();
      arena[place] = who;
   }
   public boolean free(int place) {
      return (arena[place] == NONE);
   }
   public void clear() {
      for(int i=0; i<9; i++)
         arena[i] = NONE;
   }
   public void move(int place, int who) {
      arena[place] = who;
   }

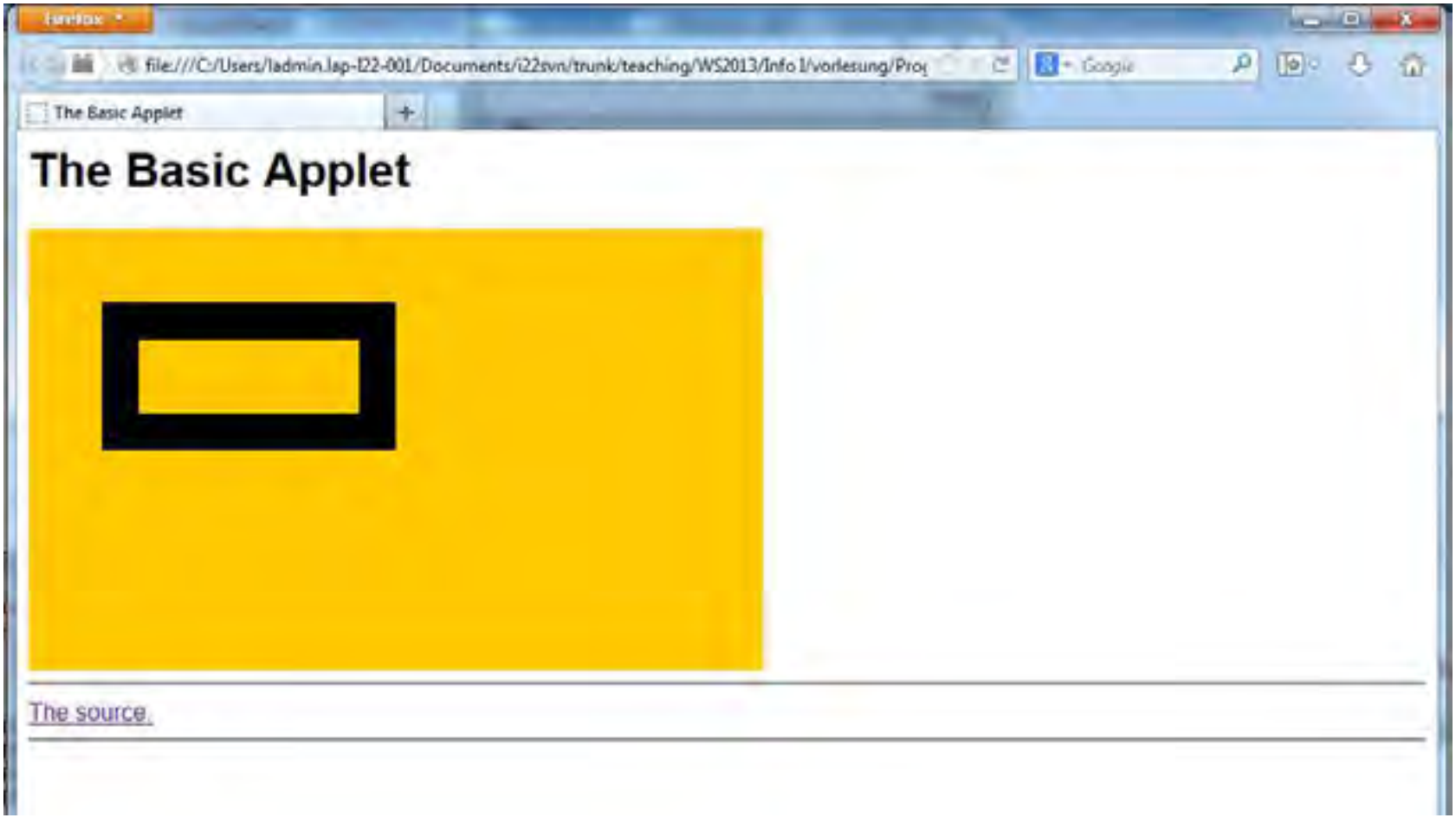
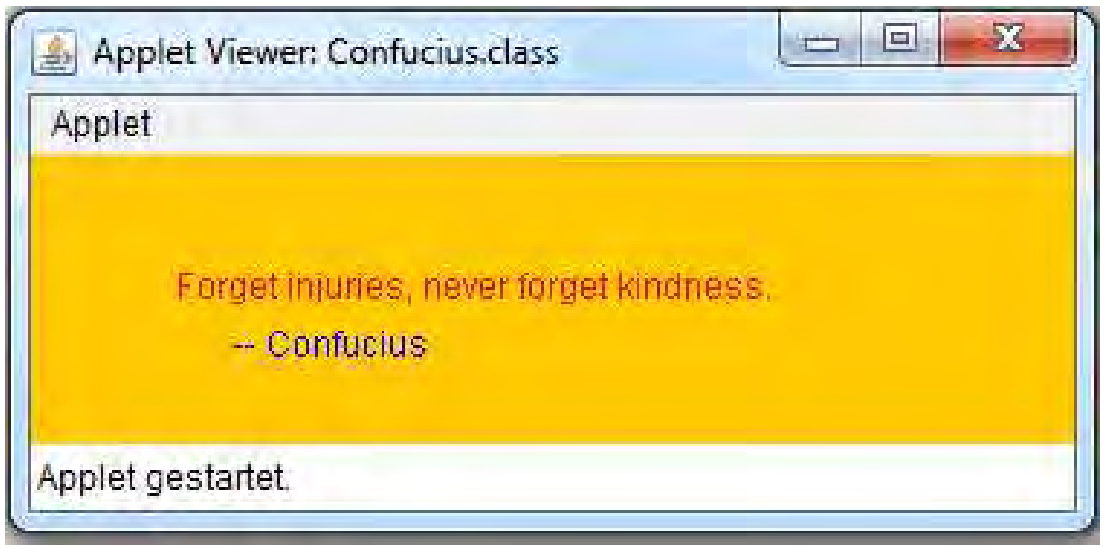
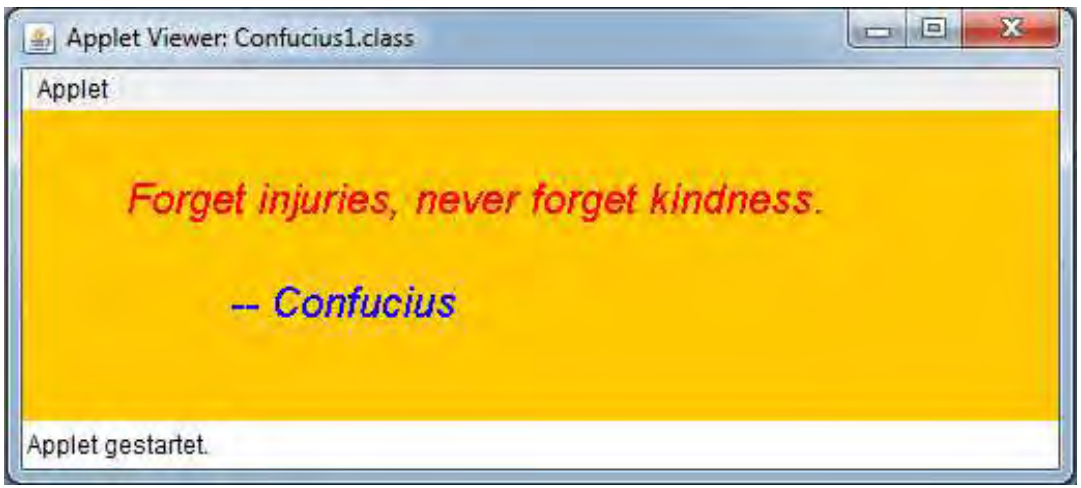
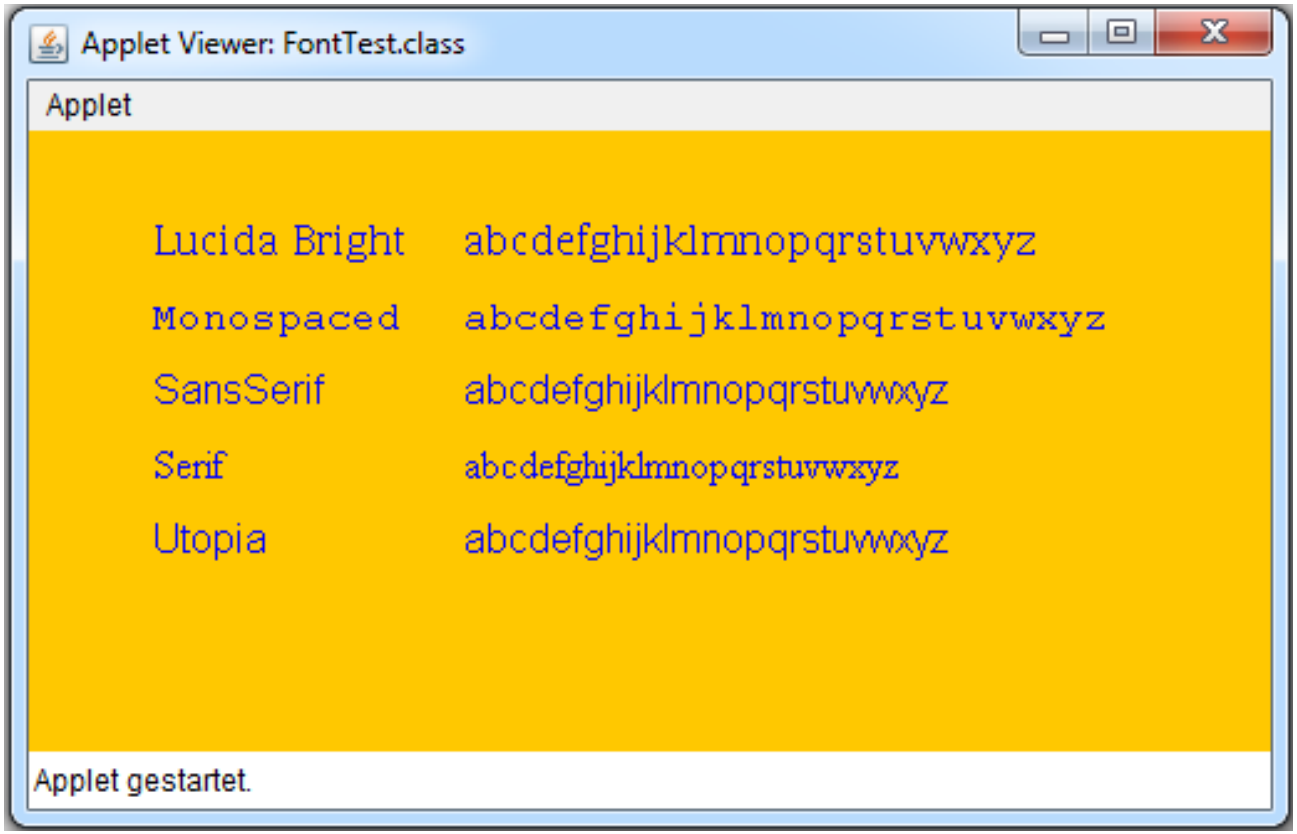
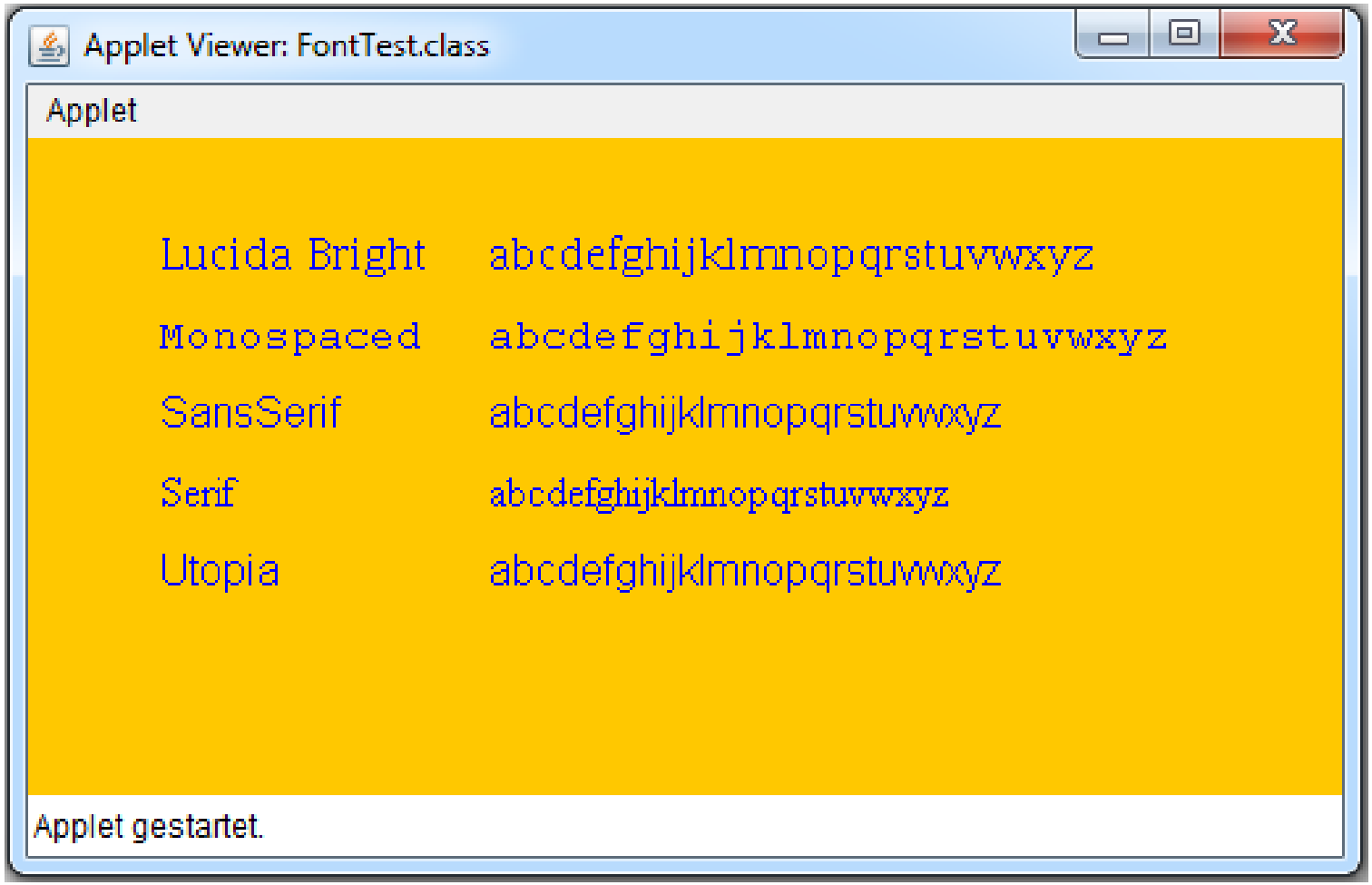
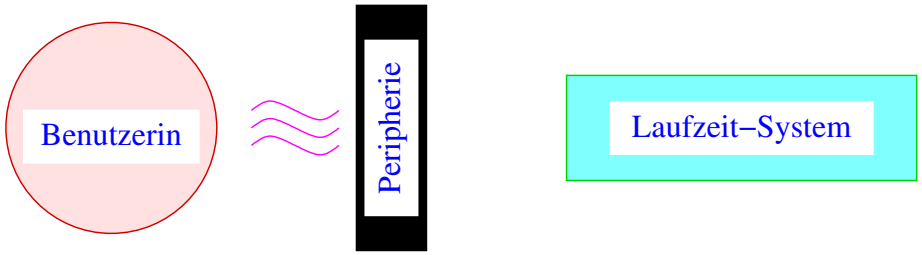
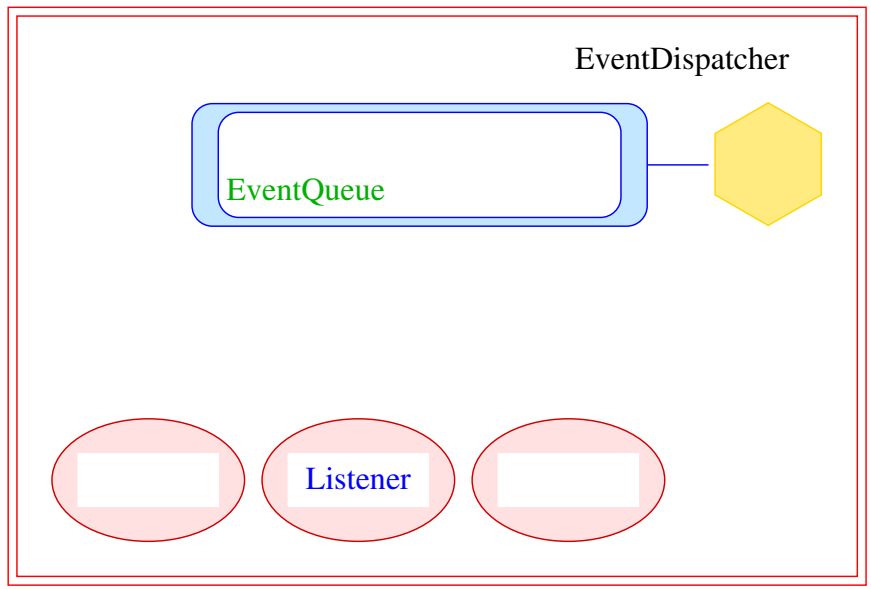
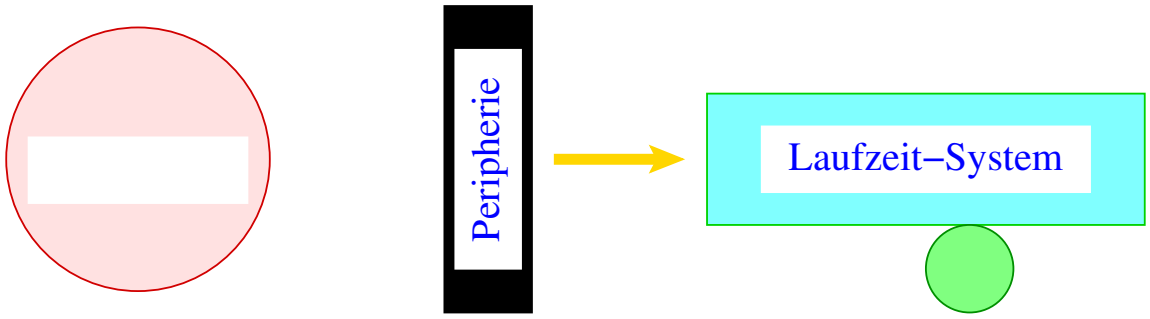
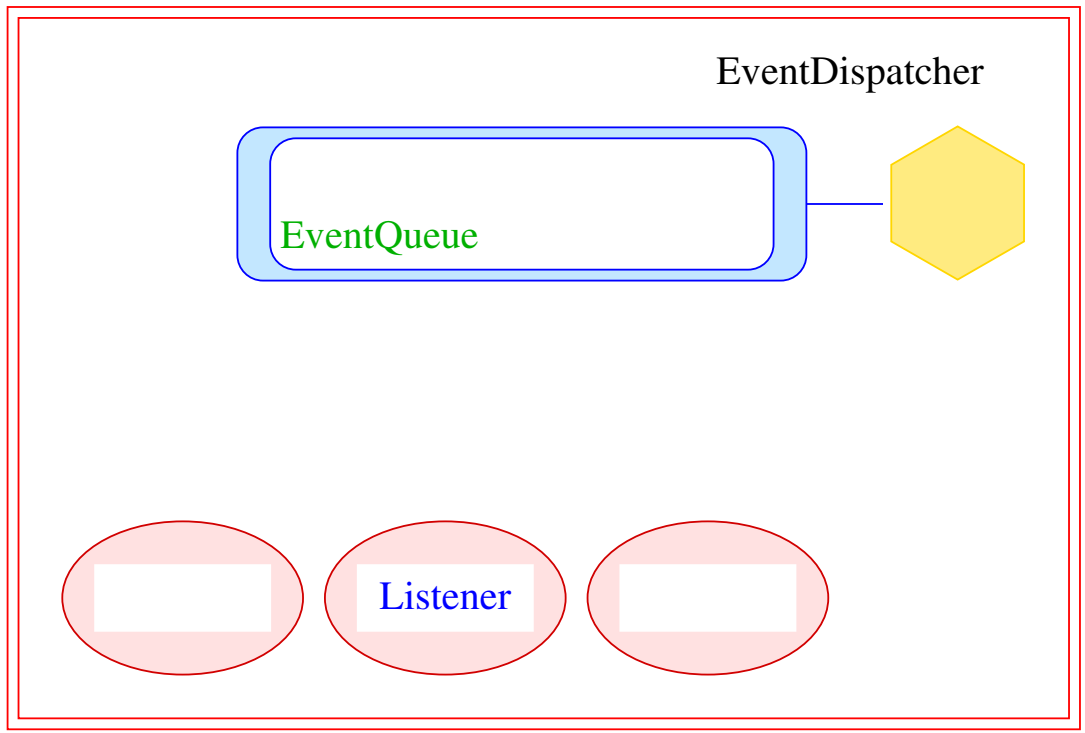
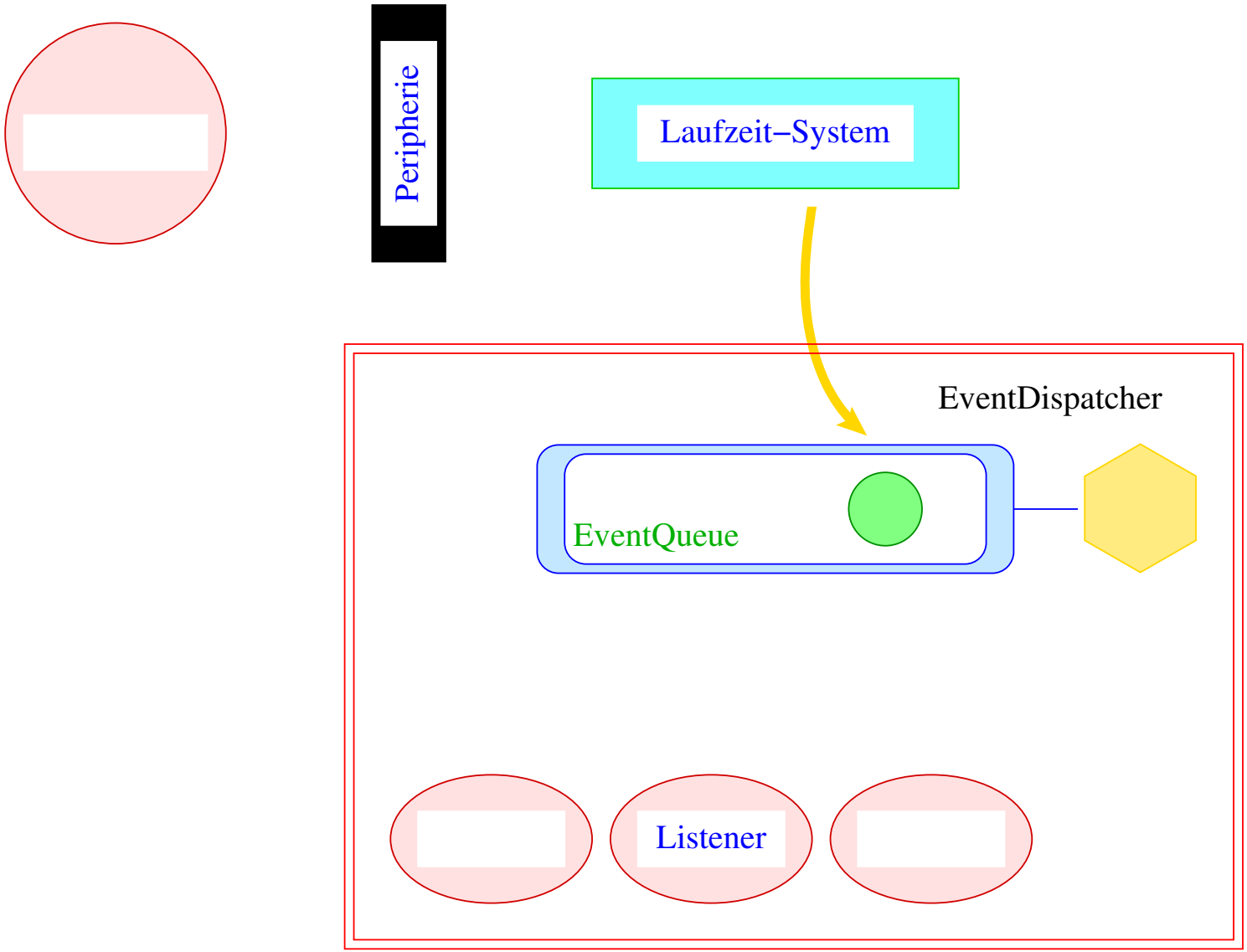
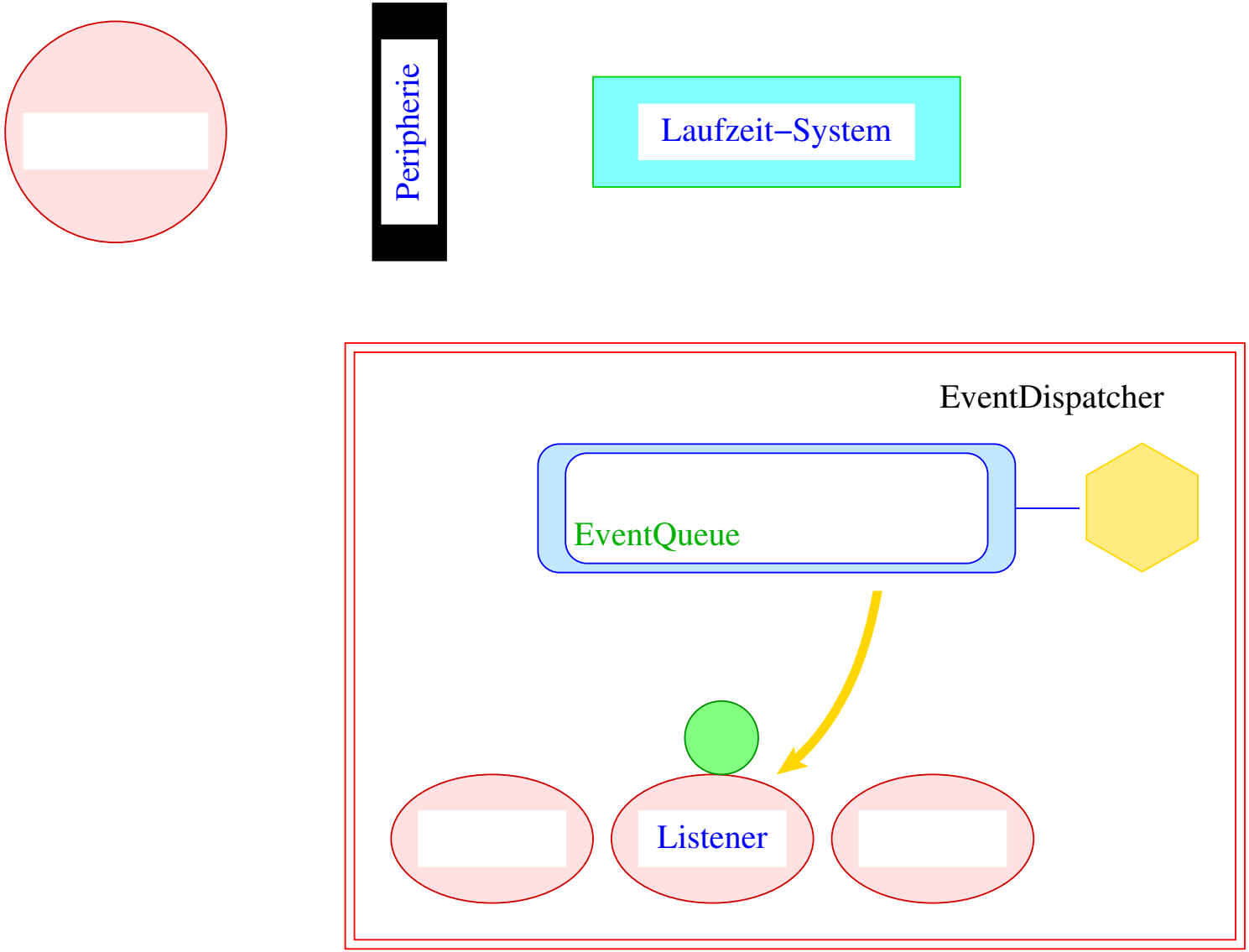
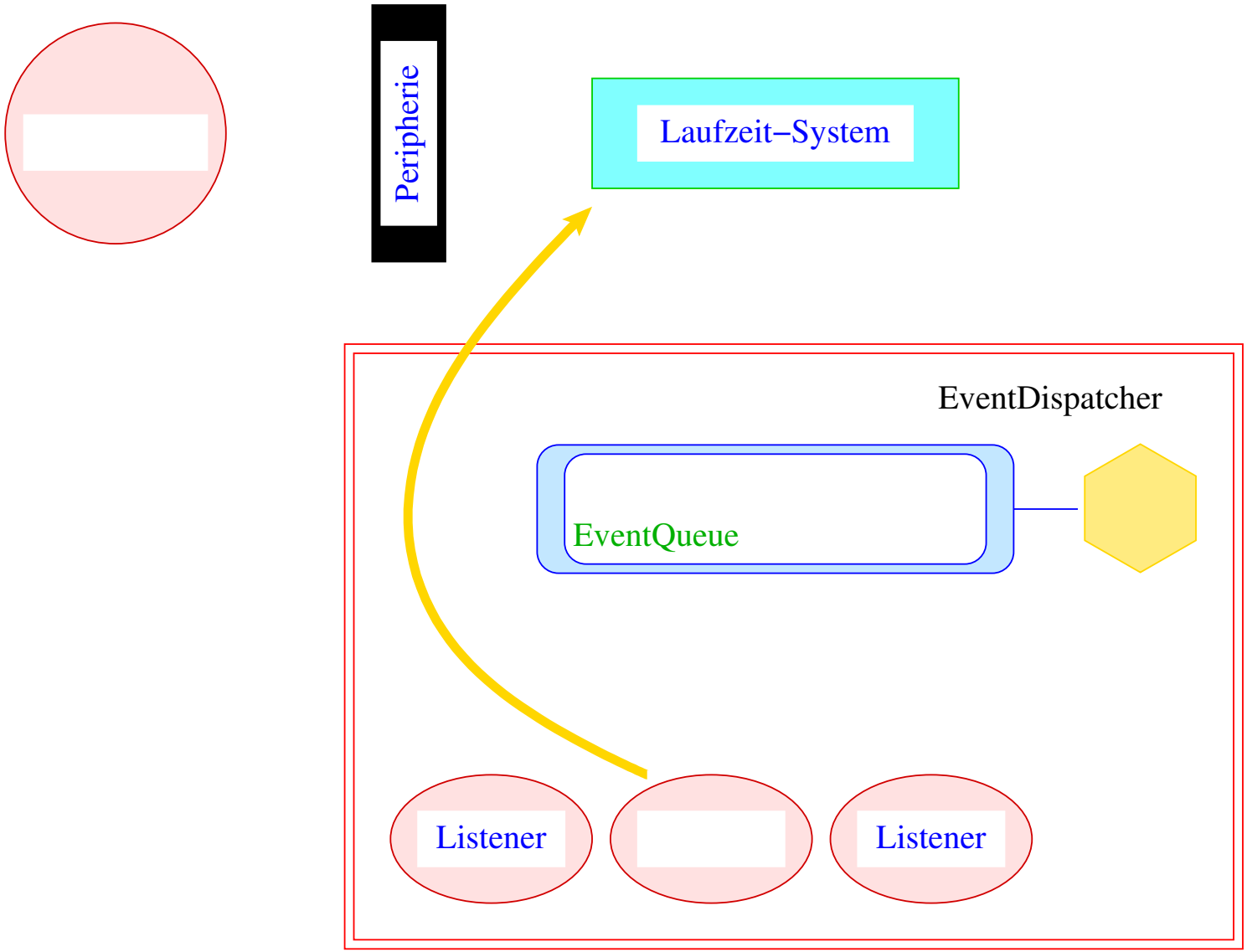
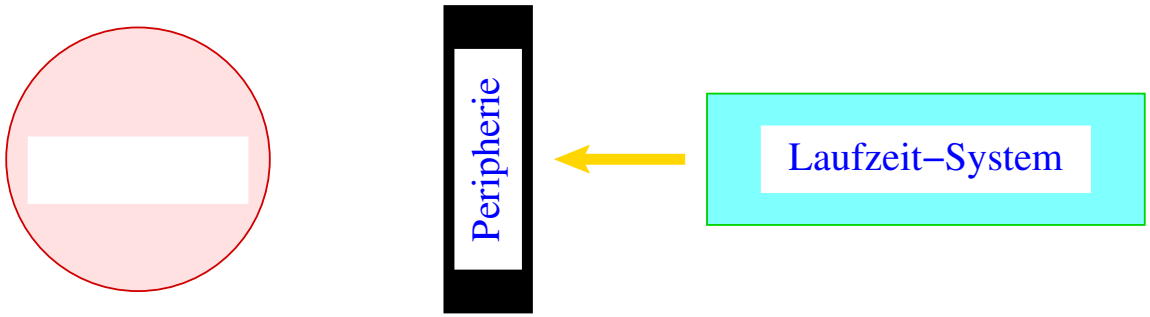
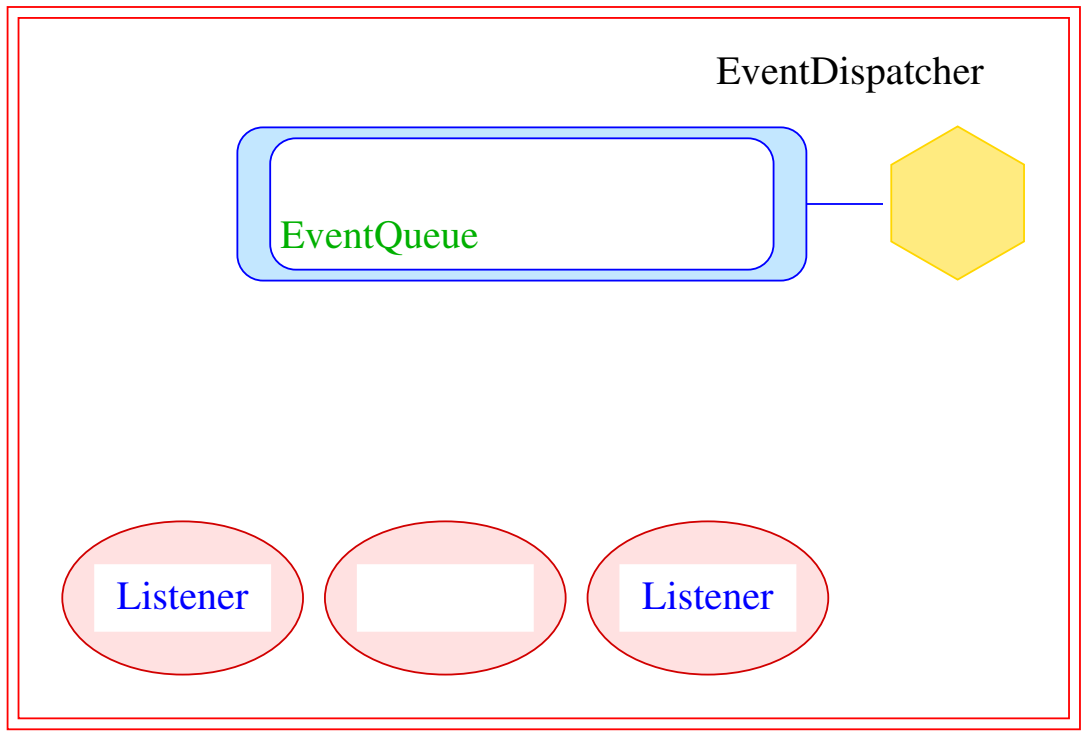
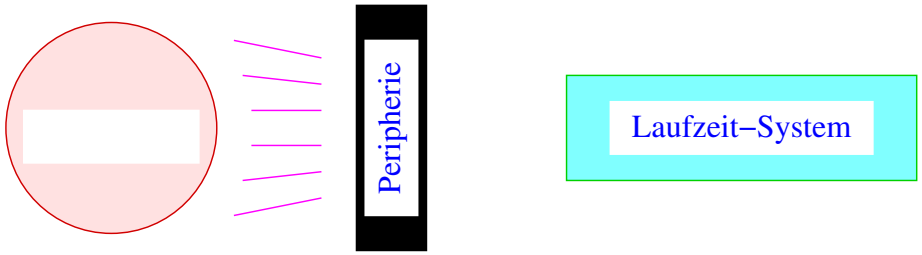
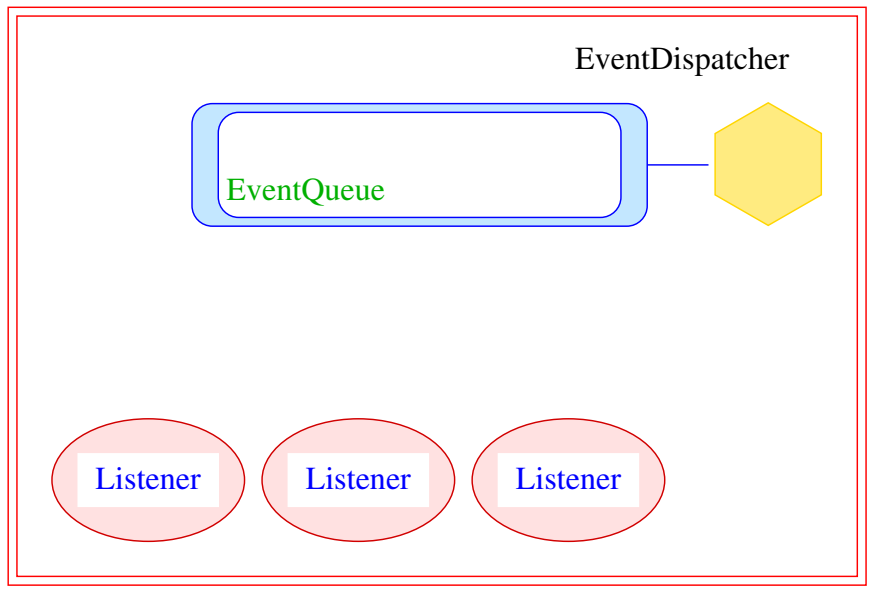
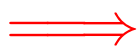
   // erfolgt durch clevere Berechnung im Feld der Länge 9
   private boolean winning(int l, int place, int who) {...}
   public boolean won(int place, int who) {...}
   public int force(int who) {...}
   ...
```

Und noch die Klasse PlayGround II

```
...
public static void main(String[] args) {
      // Spiel: wir verlieren aus Blödheit
    PlayGround ground = new PlayGround();
    ground.move(1,YOU);
    System.out.println((new MyChoice(ground,1)).acceptableChoice()); //0
    ground.move(0,ME);
    ground.move(4,YOU);
    System.out.println((new MyChoice(ground,4)).acceptableChoice()); //7
    ground.move(7,ME);
    ground.move(2,YOU);  //blöde blöde
    System.out.println((new MyChoice(ground,2)).acceptableChoice()); //6
    ground.move(6,ME);
    ground.move(3,YOU);
    System.out.println((new MyChoice(ground,3)).acceptableChoice()); //8
    ground.move(8,ME);
    System.out.println(ground.won(8,ME));  // gewonnen!
 }  // end of class PlayGround
```

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
- 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 ...

Komplexität

Zum Beispiel:

- Spielen wir Tic-Tac-Toe auf einem 4x4 Brett, gibt es ca. $16! \sim 10^{13}$ mögliche Spiele.
- Auf einem 5x5 Brett gibt es ca. $25! \sim 10^{25}$ mögliche Spiele.
- Es ist nicht so einfach, die Anzahl *interessanter* Spiele zu approximieren, sehe z.B. http://www.mathrec.org/old/2002jan/solutions.html
- Es gibt mindestens 10^{120} mögliche Schachspiele! (Shannon's number)
- Zum Vergleich: Es gibt ca. 10^{81} Atome im beobachtbaren Universum.

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.

Aber:

- I.a. ist eine solche Funktion nicht bekannt !
- Man muss mit unpräzisen bis fehlerhaften Bewertungs-Funktionen zurechtkommen ...
- ... aber die betrachten wir hier nicht!

3. Idee: xkcd



Quelle: http://xkcd.com/832/

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

Graphical User Interfaces

... und wenn am Ende des Semesters noch Zeit ist, kommen wir zum TicTacToe-Beispiel zurück und implementieren GUIs!

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

10. Nebenläufigkeit

Threads

- Monitore
- Semaphore und das Producer-Consumer-Problem
- Thread-sichere Datenstrukturen und RW-Locks
- Interrupts

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.

Beispiel: Garbage Collection



Beispiel: Garbage Collection



Threads

- 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.

Erzeugung



Variante 1

```
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
```

Erläuterungen

- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede Unterklasse von `Thread` sollte die 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.

Variante 2

```
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
```

Erläuterungen

- 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.
- Unterschied: Abgeleitete Klasse etwas einfacher zu verwenden; Verwendung der Interface-Variante erlaubt Erben von einer anderen Klasse

Mögliche Ausführungen

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

... oder:

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

Scheduling

- Ein Thread kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) oder Kern 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). U.a.:
  - Zeitscheiben-Verfahren
  - Naives Verfahren

Schedulingstrategie 1: Zeitscheiben-Verfahren

Strategie:

- 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.

Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Möglicher Schedule



Ein möglicher anderer Schedule

Eine andere Programm-Ausführung mag dagegen liefern:

Ein möglicher anderer Schedule

Eine andere Programm-Ausführung mag dagegen liefern:



Erläuterungen

- 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.

Schedulingstrategie 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 ...

Weitere mögliche Schedules



Weitere mögliche Schedules



Weitere mögliche Schedules



Weitere mögliche Schedules



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
```

Beispiel

... 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␣...
```

Implementierungen

- 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 berücksichtigt wird.

Einige Thread-Zustände



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.

Möglicher Ablauf



Möglicher Ablauf



Möglicher Ablauf



Möglicher Ablauf



Möglicher Ablauf



Möglicher Ablauf



Möglicher Ablauf



Möglicher Ablauf



Beachte

- Für jedes Threadobjekt `t` gibt es eine Schlange `ThreadQueue joiningThreads`.
- Threads, die auf Beendigung des Threads `t` warten, werden in diese Schlange eingefügt.
- Dabei gehen sie konzeptuell in einen Zustand joining über und werden aus der Menge der ausführbaren Threads entfernt.
- Beendet ein Thread seine Ausführung, werden alle Threads, die auf ihn warteten, wieder aktiviert.
- Wenn innerhalb von Thread `t1` die Methode `join()` eines Threads `t2` ausgeführt wird, also `t2.join()` in `t1`, dann unterbricht `t1` seine Ausführung und wartet auf die Beendigung von `t2`.

Erweitertes Zustandsdiagramm



Erweitertes Zustandsdiagramm



Beispiel (1)

```
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());
      }
   } ...
```

Beispiel (2)

```
    ...
    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
```

Variation

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
```

Ausführung

- Das Programm terminiert möglicherweise nicht!
- `task[0]` wartet auf `task[1]`, `task[1]` wartet auf `task[2]`, `task[2]` wartet ... auf `task[0]`
  ist ein möglicher Ablauf!
- Aber nicht gezwungenermaßen: Wenn der Schedule zufällig

```
[main]  t[0].start();
[t[0]]  t[1].join();
[main]  t[1].start();
...
```

  liefert, dann terminiert das Programm, weil `t[1].join()` für einen noch nicht gestarteten `t[1]` keinen Effekt auf `t[0]` hat!

Deadlocks



Quelle: http://www.cs.gmu.edu/cne/itcore/processes/Dead.html

- Jeder Thread geht in einen Wartezustand (hier: joining) über und wartet auf einen anderen Thread.
- Dieses Phänomen heißt auch Circular Wait oder Deadlock oder Verklemmung (↑Compilerbau) – eine unangenehme Situation, die man tunlichst vermeiden sollte.
  Aber das ist richtig schwierig!

Monitore: Beispiel 1

- 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.
- Beispiel Flugbuchung: Abfrage von Person P1 liefert „Flug verfügbar“. Wenn P1 diesen Flug buchen will, kann es aber sein, dass in der Zwischenzeit P2 bereits diesen selben (letzten) Flug gebucht hat. Wenn dann P1 den Flug erneut bucht, gibt es eine Inkonsistenz.
  Situationen dieser Art heissen Race Conditions.

Monitore: Beispiel 2

- Beispiel Dining Philosophers: Philosophen sitzen an einem runden Tisch. Jeder hat einen Teller vor sich; jeweils zwischen zwei Tellern liegt eine Gabel. Philosophen können denken oder essen. Wenn sie essen, müssen sie das mit zwei Gabeln tun. Dazu nehmen sie immer erst die linke, dann die rechte Gabel. Wenn alle Philosophen gleichzeitig ihre linke Gabel ergreifen und auf ihre rechte Gabel warten und ihre linke Gabel nicht abgeben, bevor sie gegessen haben, gibt es einen Deadlock.

Quelle: http://upload.wikimedia.org/wikipedia/commons/thumb/6/6a/Dining_philosophers.png/512px-Dining_philosophers.png

Monitore

- Ein Hilfsmittel, um geordnete Zugriffe zu garantieren, sind Monitore.

... ein Beispiel:

Beispiel

Drei Threads sollen eine gemeinsam genutzte Variable x individuell jeweils um 1 erhöhen:

```
public class Inc implements Runnable {
   private static int x = 0; // gemeinsame Ressource
   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);  // simuliert zufälliges Scheduler-Verhalten
      int y = x;
      System.out.println(s+ " read "+y);
      pause(4);  // simuliert zufälliges Scheduler-Verhalten
      x = y+1;
      System.out.println(s+ " wrote "+(y+1));
   }
```

Beispiel

```
    ...
    public static void main(String[] args) {
        (new Thread(new Inc())).start();
        pause(2);   // simuliert zufälligen Scheduler
        (new Thread(new Inc())).start();
        pause(2);   // simuliert zufälligen Scheduler
        (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`.

Mögliche Ausführung

```
> 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
```

Das heißt: Es wurden drei Inkrementoperationen ausgeführt, aber am Ende des Programms ist der Wert von `x` trotzdem 2, nicht 3!

Grund



Grund



Grund



Grund



Grund



Grund



Grund



Grund



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.

Ein weiterer Zustand



Ein weiterer Zustand



Locks: `lock()`

- Jedes Objekt obj mit `synchronized`-Methoden verfügt
  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; // betritt krit. Abschnitt
   else {                      // Lock bereits vergeben
      Thread t = Thread.currentThread();
      blockedThreads.enqueue(t);
      t.state = blocked;       // blockiere
   }
}  // end of lock()
```

... und `unlock()`

- 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.

Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



synchronized

```
public class Count {
   private int count = 0;
   public synchronized void inc() {
      String s = Thread.currentThread().getName();
      int y = count;     System.out.println(s+ "␣read␣"+y);
      count = 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.

Producer/Consumer

- 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.

Beispiel

Beispiel



Beispiel



Beispiel



Beispiel



Beispiel



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 ...

Warten



Warten



Warten



Warten



Warten



Warten



Warten



Warten



Realisierung

- 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();
}
```

Beispiel



Beispiel



Beispiel



notify()

- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand blocked und fügt ihn in `blockedThreads` ein:

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

- `notifyAll();` weckt alle wartenden Threads auf:

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

Beispiel



Beispiel



Zustandsdiagramm



Zustandsdiagramm



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
```

Erläuterung

- 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();
}
...
```

2. Idee (2)

```
   ...
   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.

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

up und down

- 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)

```
// fehlerhaft!
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);
   }
   ...
```

Anwendung (1. Versuch)

```
...
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
```

Funktioniert nicht!

- 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

```
    ...
    //non-synchronized methods!
    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 {
        Data result;
        occupied.down();
        synchronized (this) {
            result = a[first];
            first = (first+1)%cap;
        }
        free.up();
        return result;
    }
}   // end of corrected class Buffer
```

`synchronized`

- 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;
  Aber Achtung: Zuviel überflüssige Synchronisierung kann die positiven Resultate der Parallelisierung beeinträchtigen oder sogar eliminieren!
- ... 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.
- Heisenbugs!

Ein weiteres typisches Ziel

- Eine Datenstruktur soll gemeinsam von mehreren Threads benutzt werden.
- Jeder Thread soll (gefühlt) atomar auf die Datenstruktur zugreifen.
- Lesende Zugriffe sollen die Datenstruktur nicht verändern, schreibende Zugriffe dagegen können die Datenstruktur modifizieren.

Ein weiteres typisches Ziel

- Eine Datenstruktur soll gemeinsam von mehreren Threads benutzt werden.
- Jeder Thread soll (gefühlt) atomar auf die Datenstruktur zugreifen.
- Lesende Zugriffe sollen die Datenstruktur nicht verändern, schreibende Zugriffe dagegen können die Datenstruktur modifizieren.

Beispiel: Eine Hash-Tabelle mit Schlüsseln und Werten vom Typen `String`

HashTable



Erste Idee

Synchronisiere die Methodenaufrufe ...

```
// Hashtabelle. Schlüssel Strings, Werte Strings.
public class HashTable {
   private StringList[] a;
   private int n;
   public HashTable (int n) {
            a = new StringList[n];
            this.n = n;
   }
   public synchronized String lookup (String key) {...}
   public synchronized void update (String key, String value) {...}
}
```

Diskussion

- Zu jedem Zeitpunkt darf nur ein Thread auf die HashTable zugreifen.
- Für schreibende Threads ist das evt. sinnvoll.
- Threads, die nur lesen, stören sich gegenseitig aber überhaupt nicht !

Diskussion

- Zu jedem Zeitpunkt darf nur ein Thread auf die HashTable zugreifen.
- Für schreibende Threads ist das evt. sinnvoll.
- Threads, die nur lesen, stören sich gegenseitig aber überhaupt nicht !

$\Longrightarrow$ ReaderWriterLock

RW-Lock

- Ist entweder im Lese-Modus, im Schreibmodus oder frei.
- Im Lese-Modus dürfen beliebig viele Leser eintreten, während sämtliche Schreiber warten müssen.
- Haben keine Leser mehr Interesse, ist das Lock wieder frei.
- Ist das Lock frei, darf ein Schreiber eintreten. Das RW-Lock wechselt nun in den Schreib-Modus.
- Im Schreib-Modus müssen sowohl Leser als auch weitere Schreiber warten.
- Ist ein Schreiber fertig, wird das Lock wieder freigegeben ...

Realisierung

```
public class RW {
  // negativ: Schreibmodus (ein Schreiber)
  // positiv: Lesemodus (Anzahl der Leser)
  //       0: frei
  private int countReaders = 0;

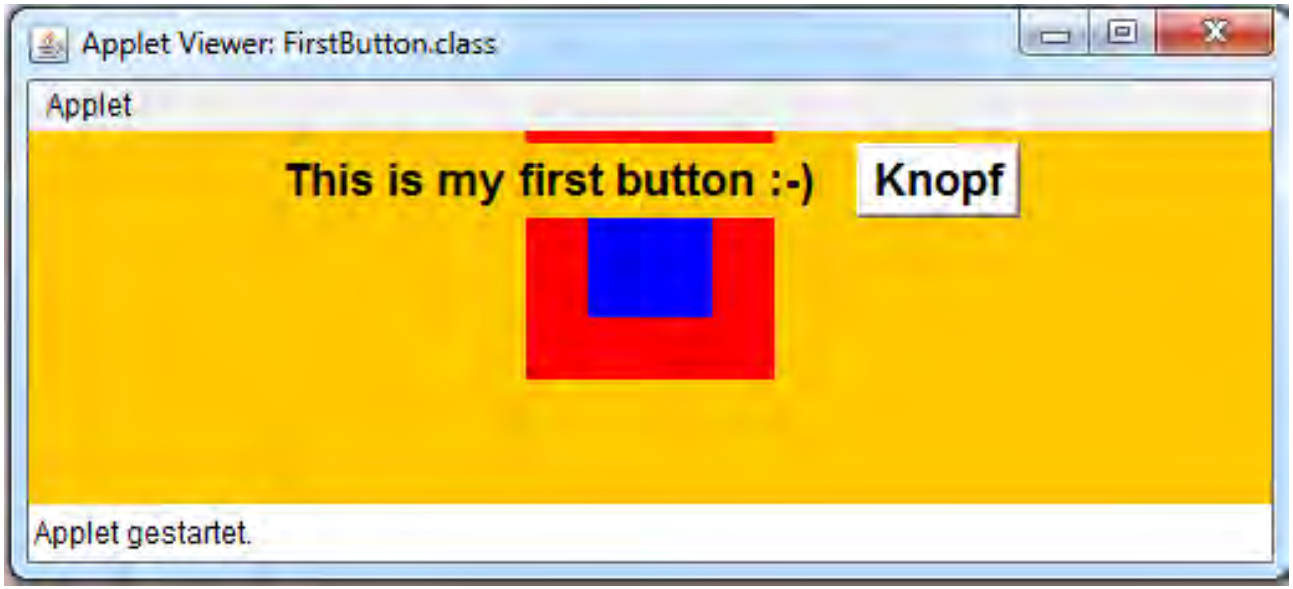
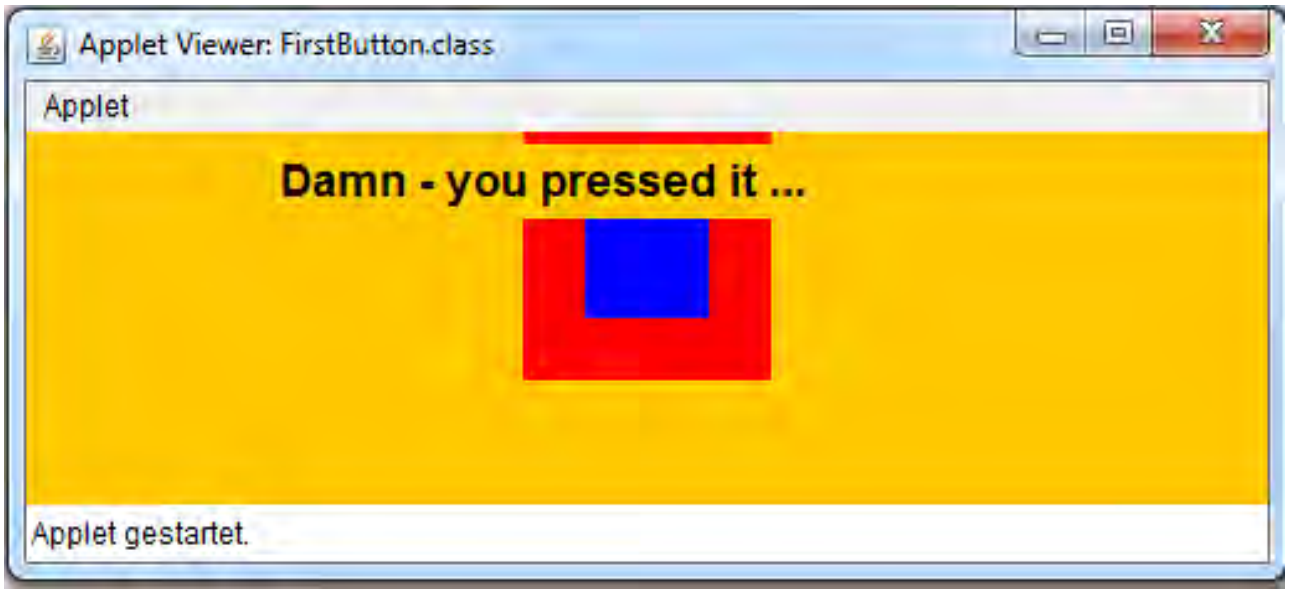
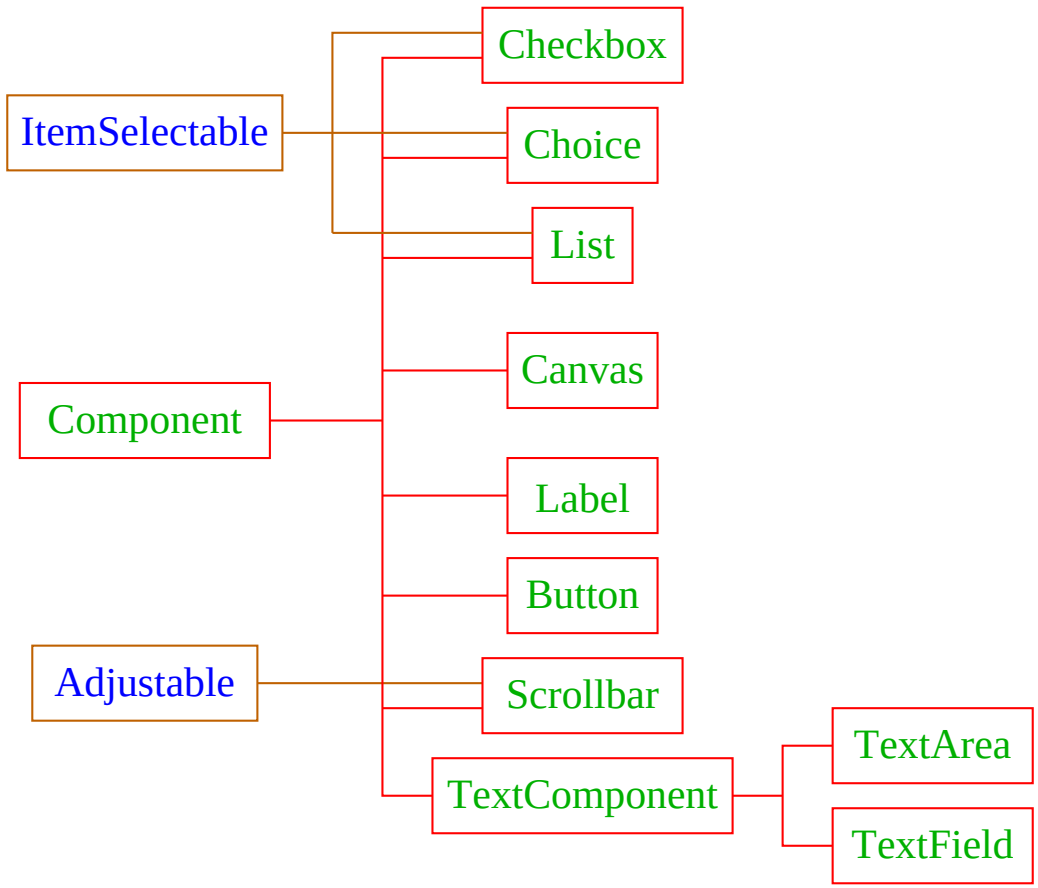
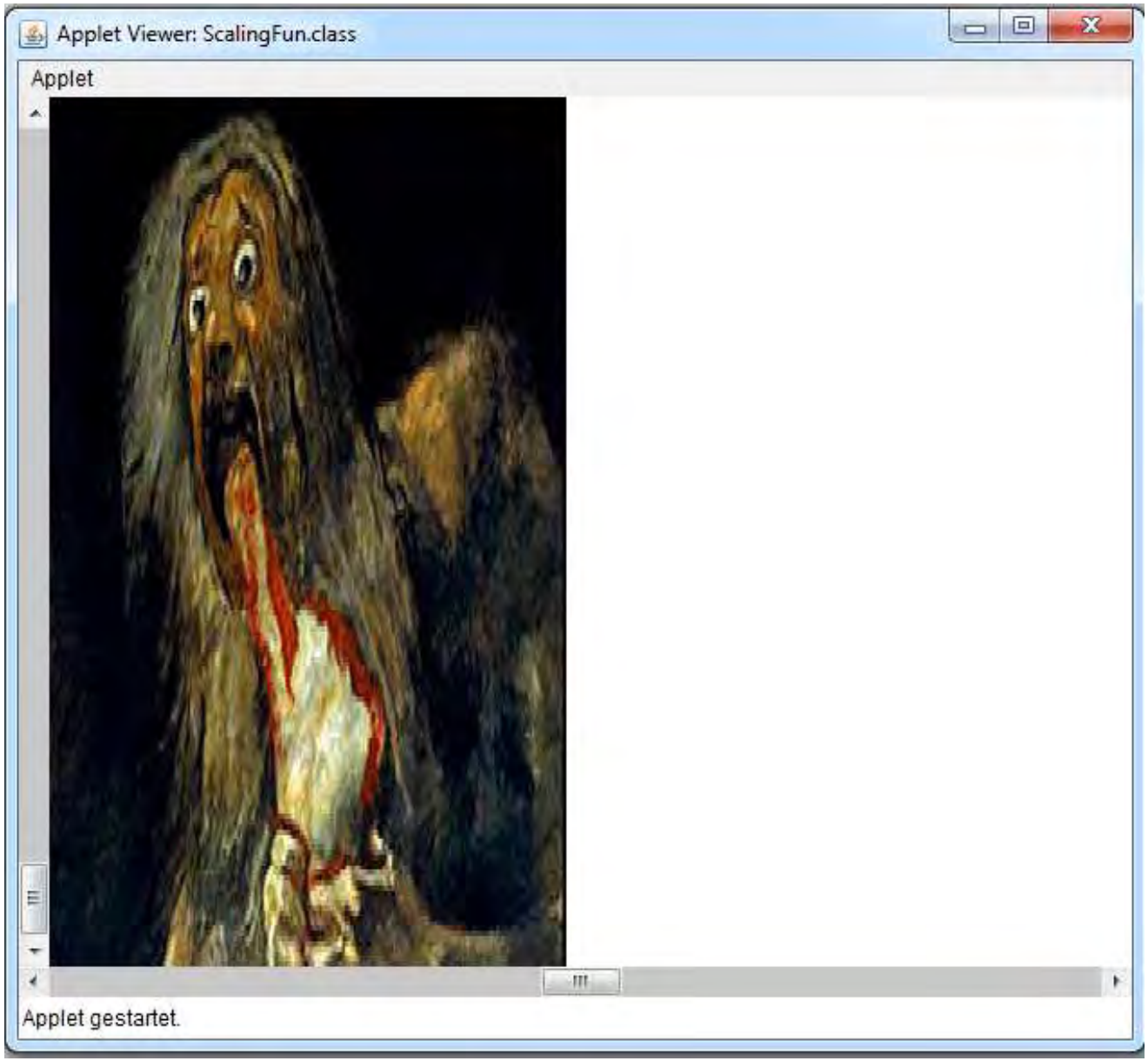
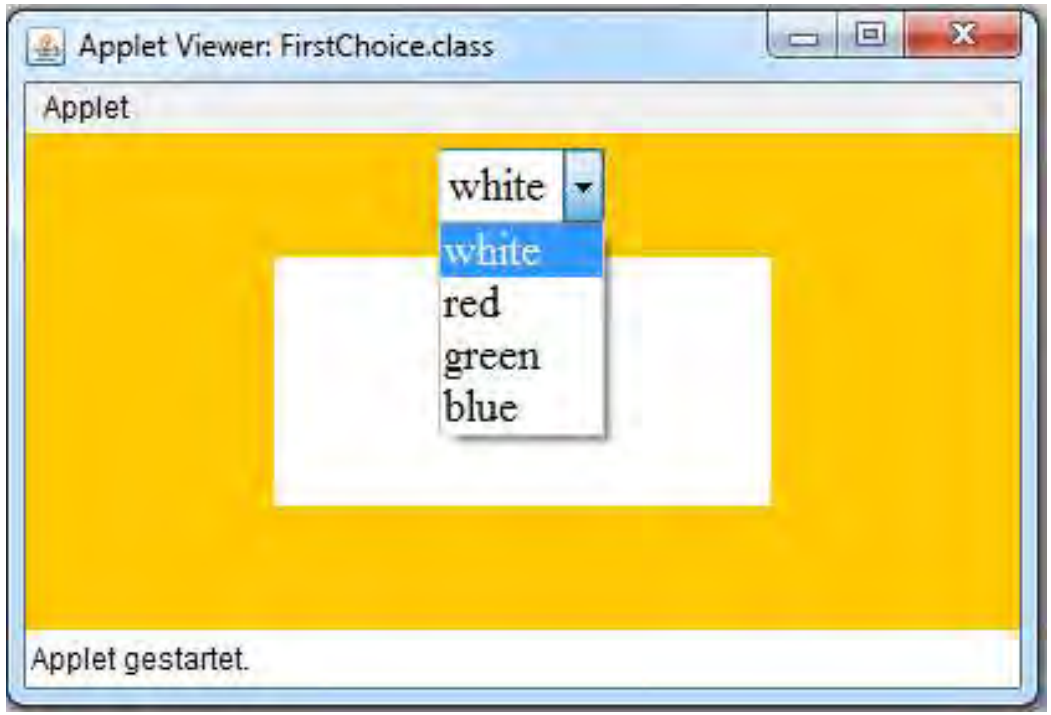
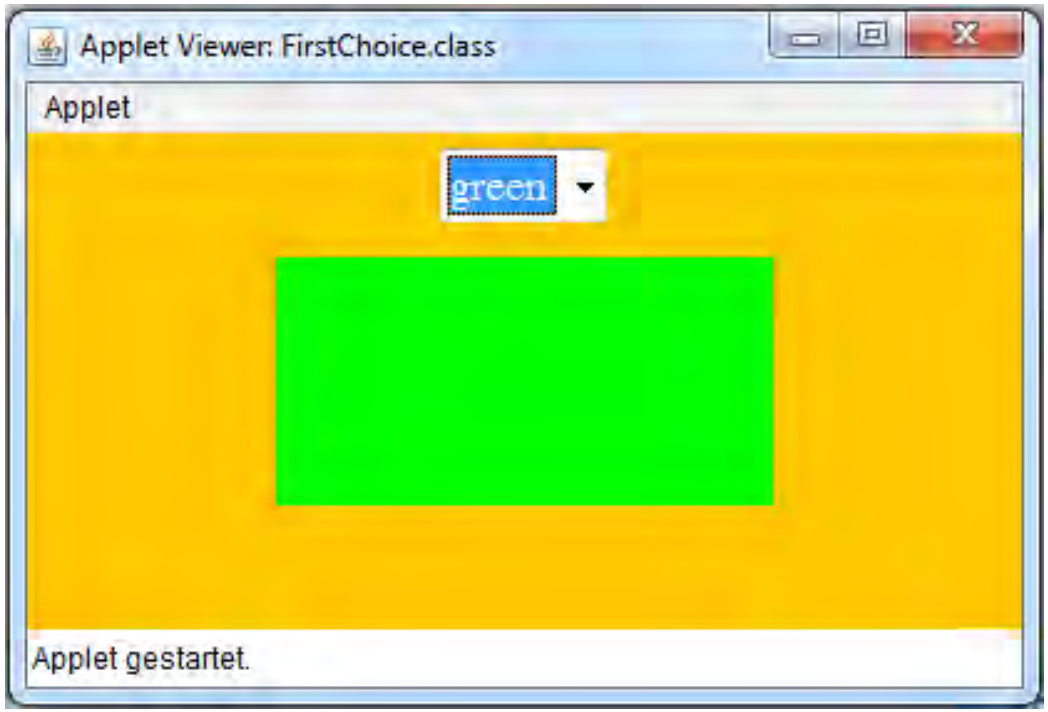
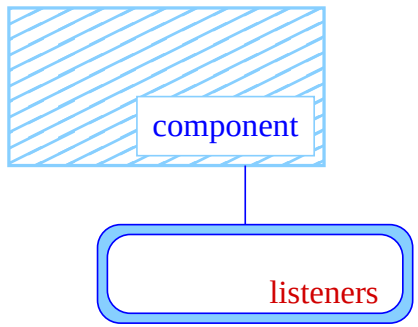
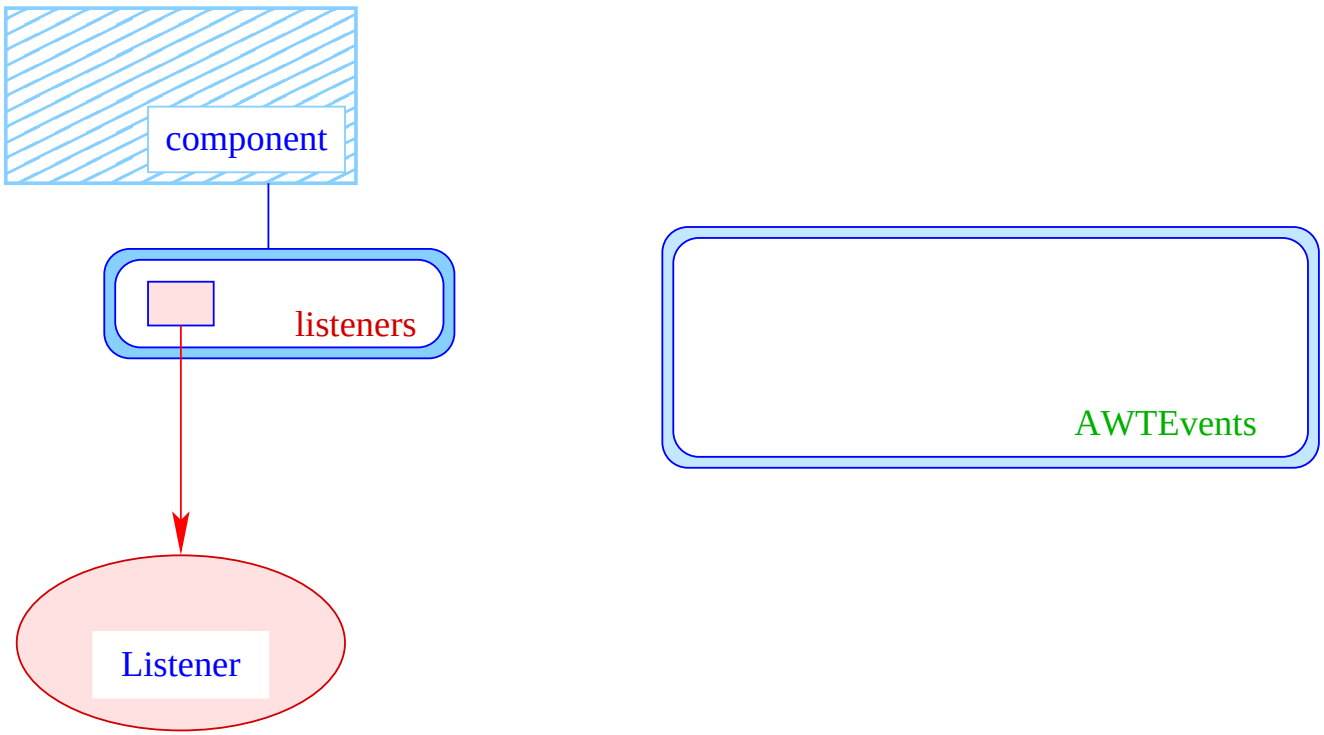
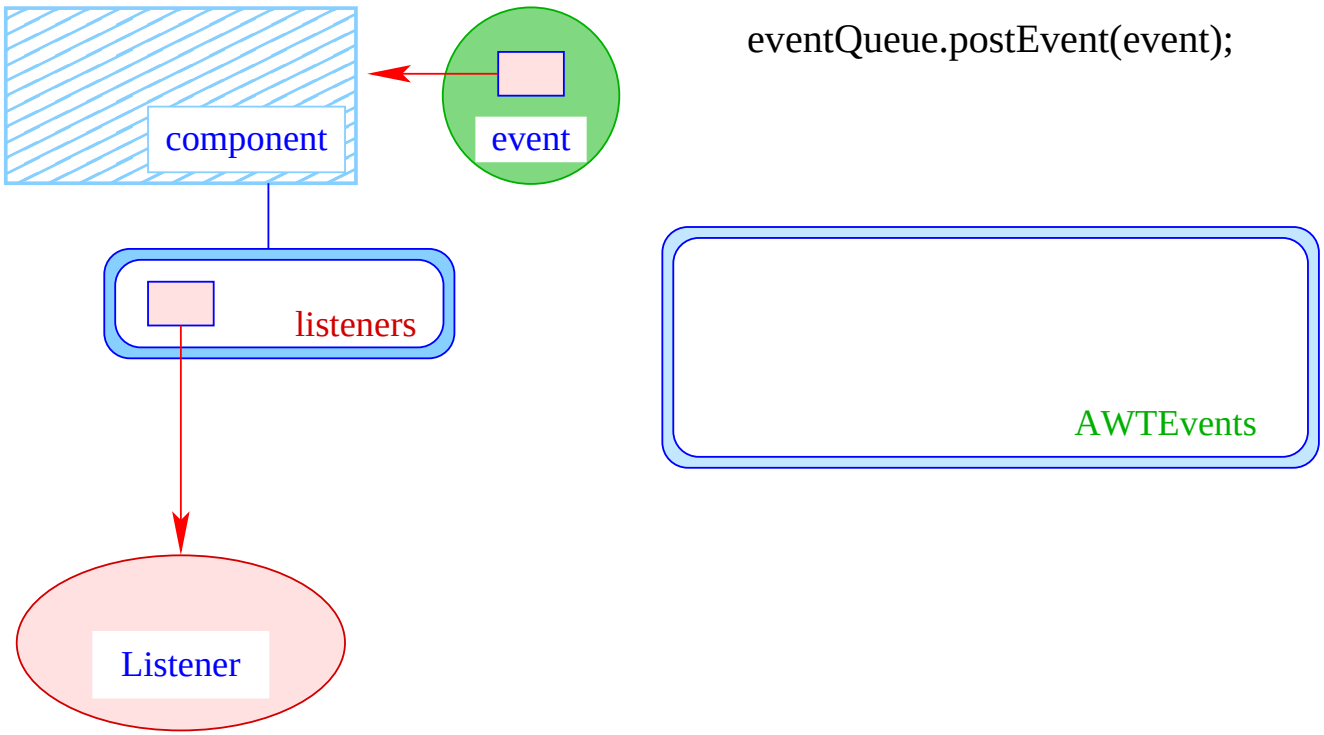
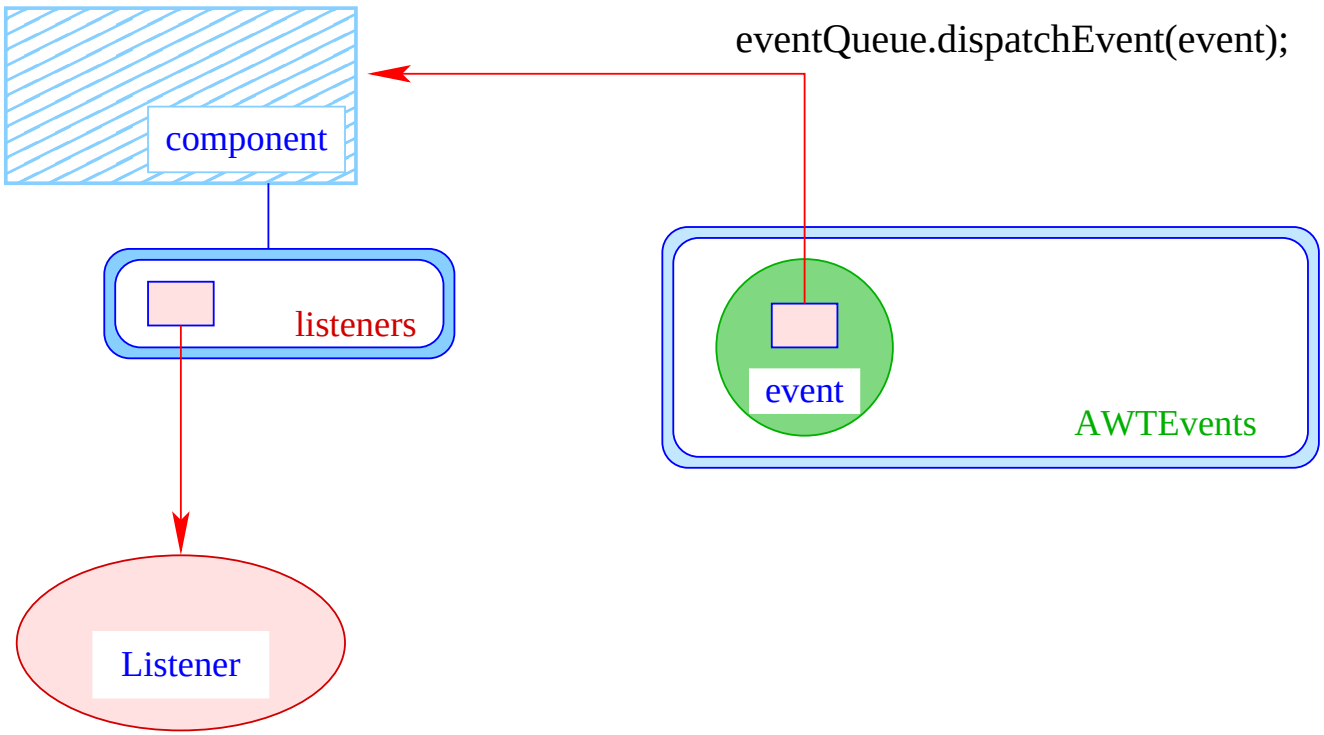
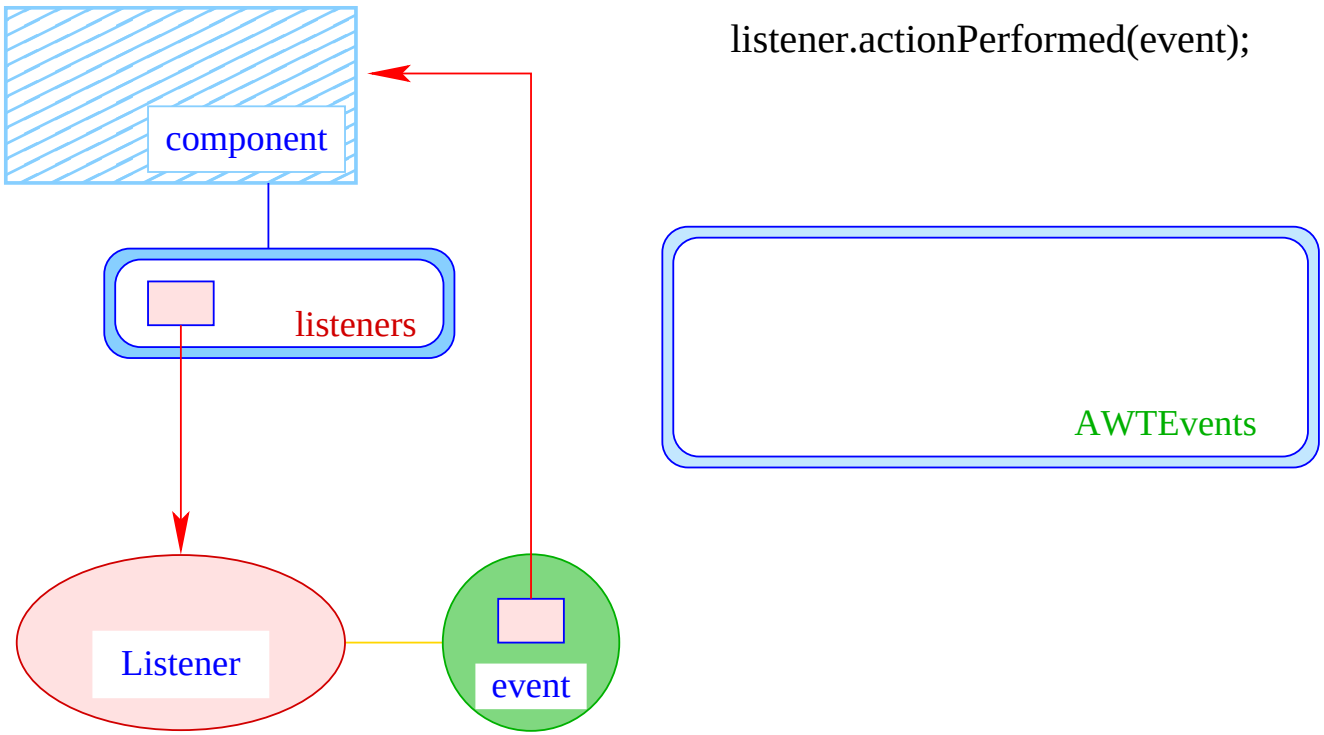
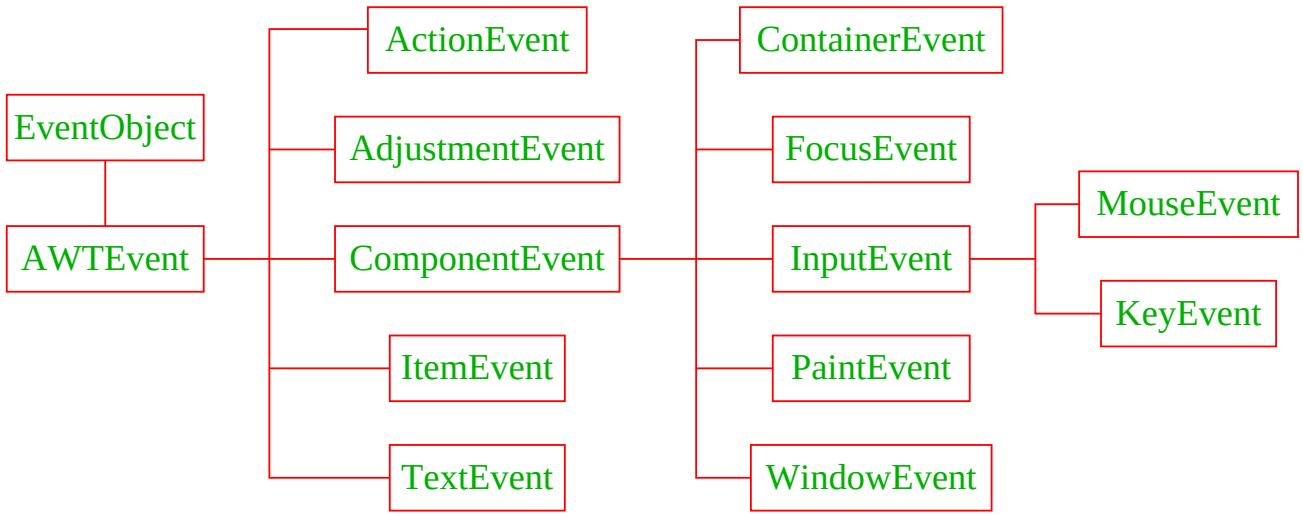
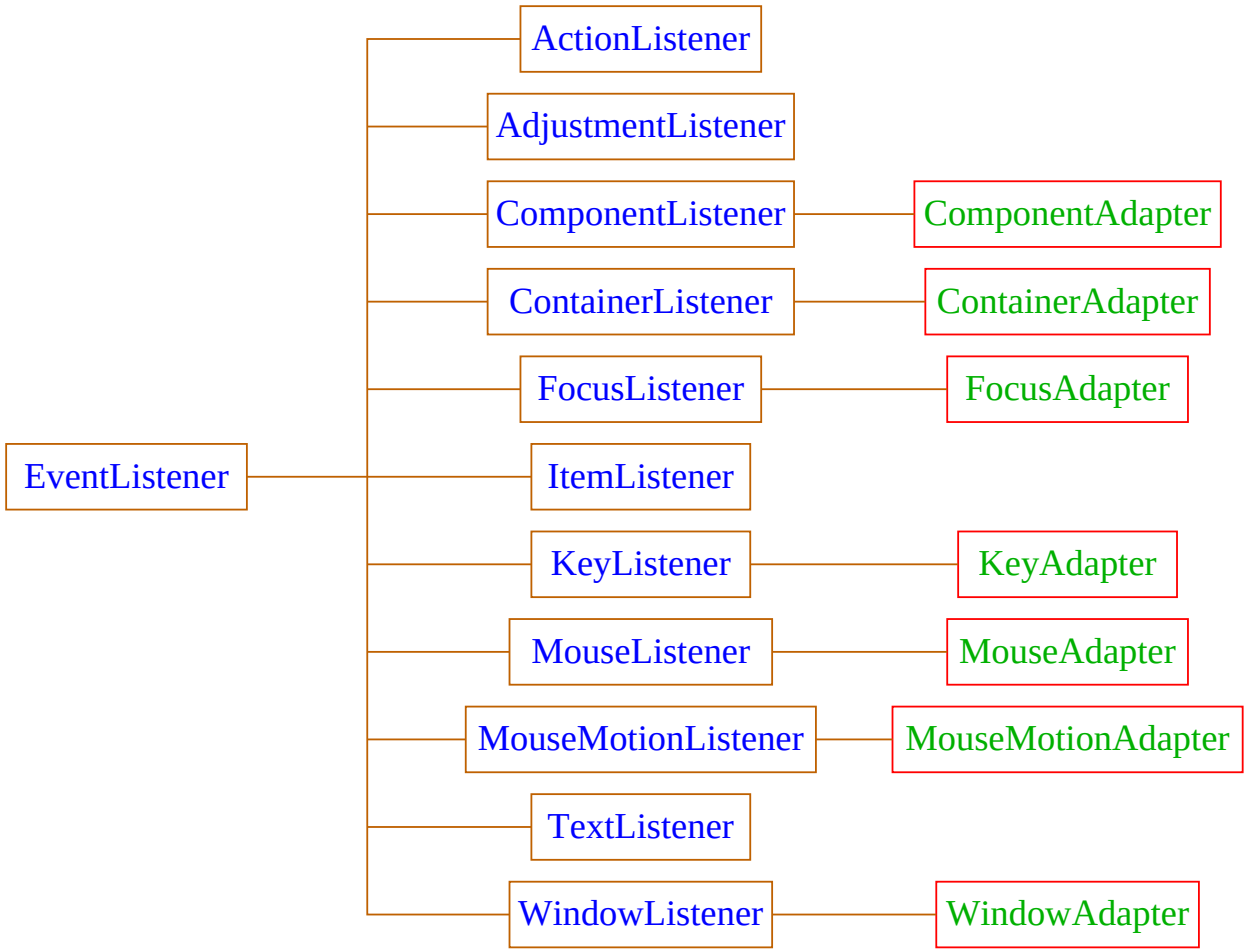
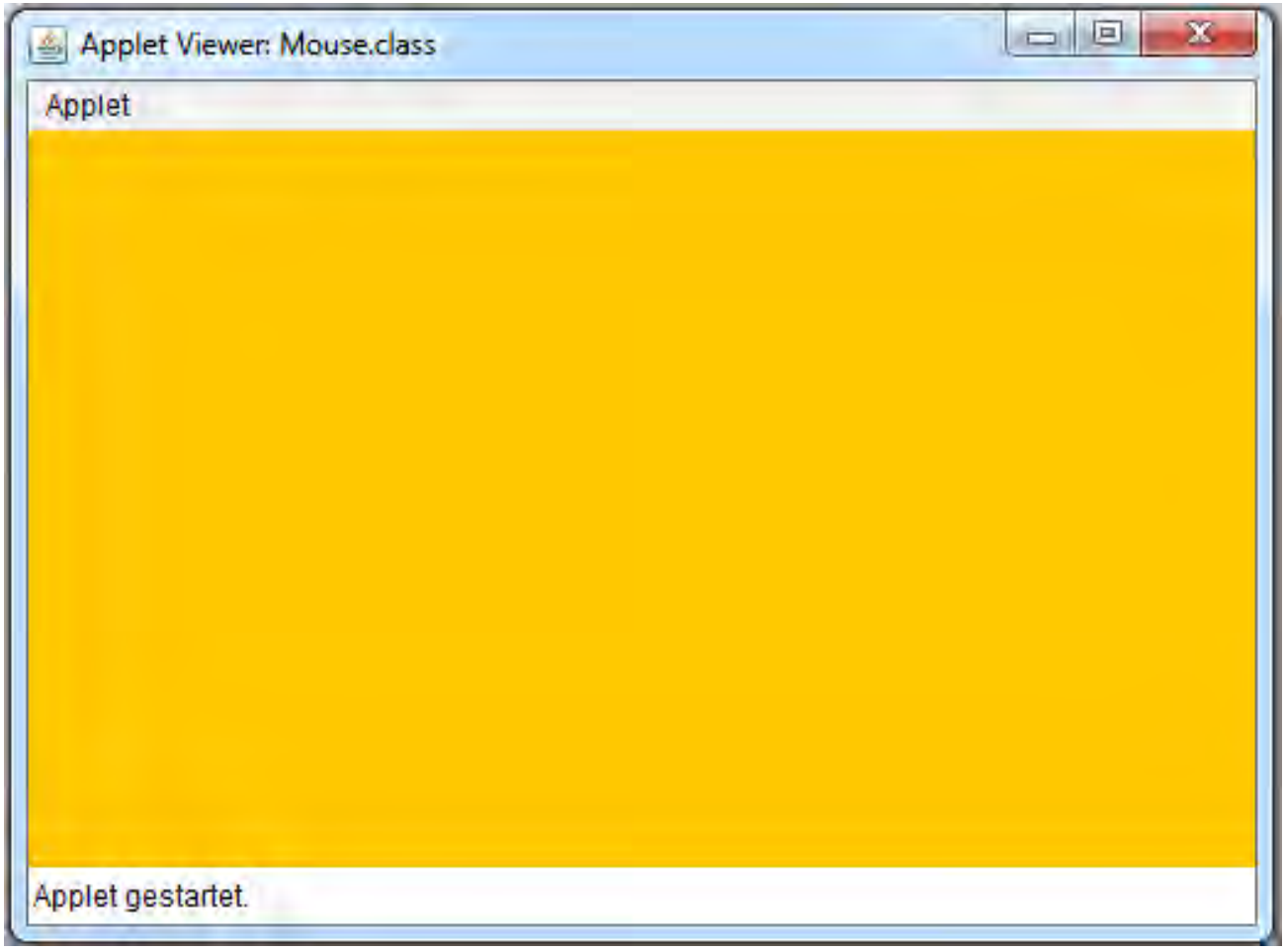
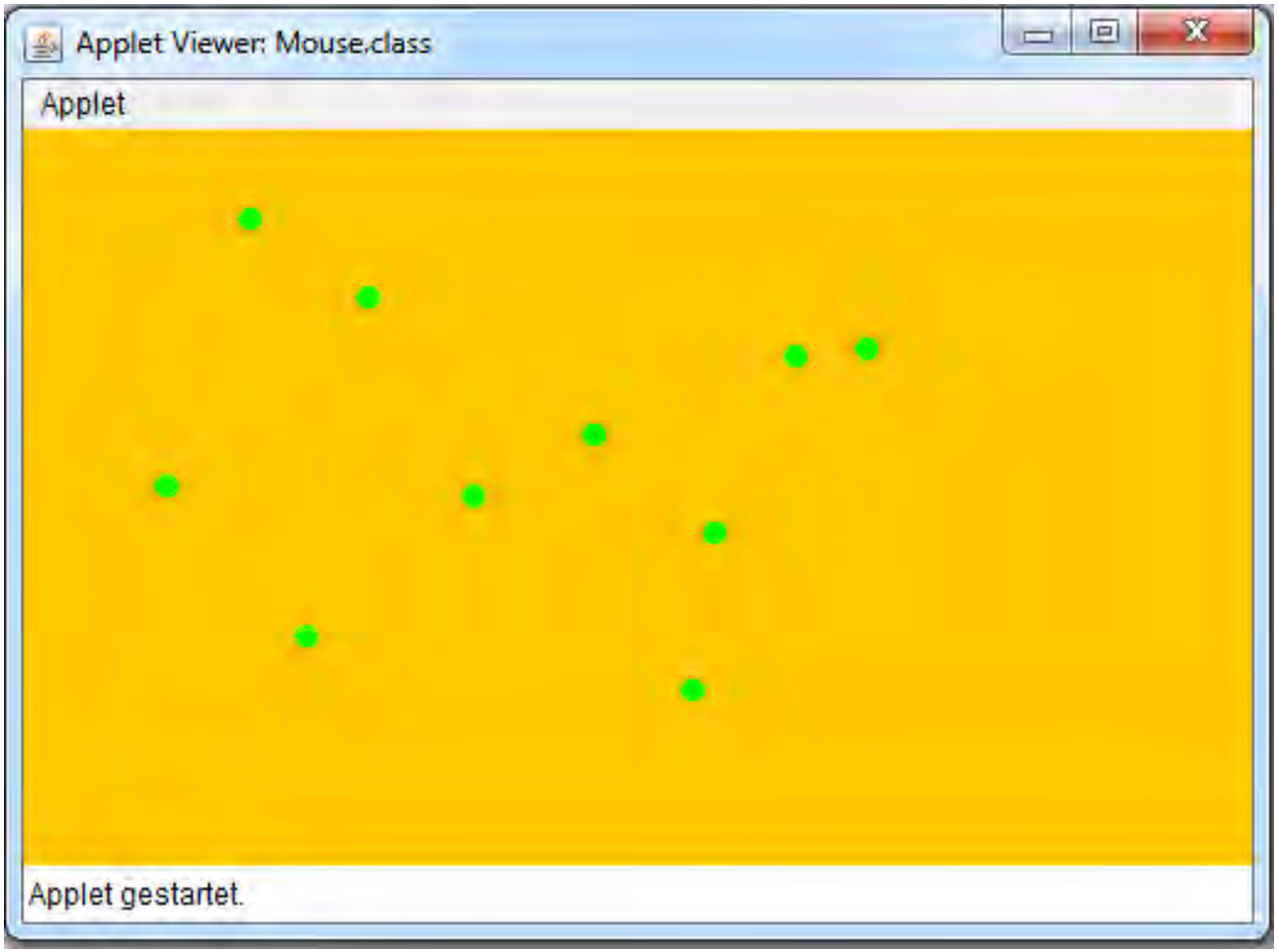
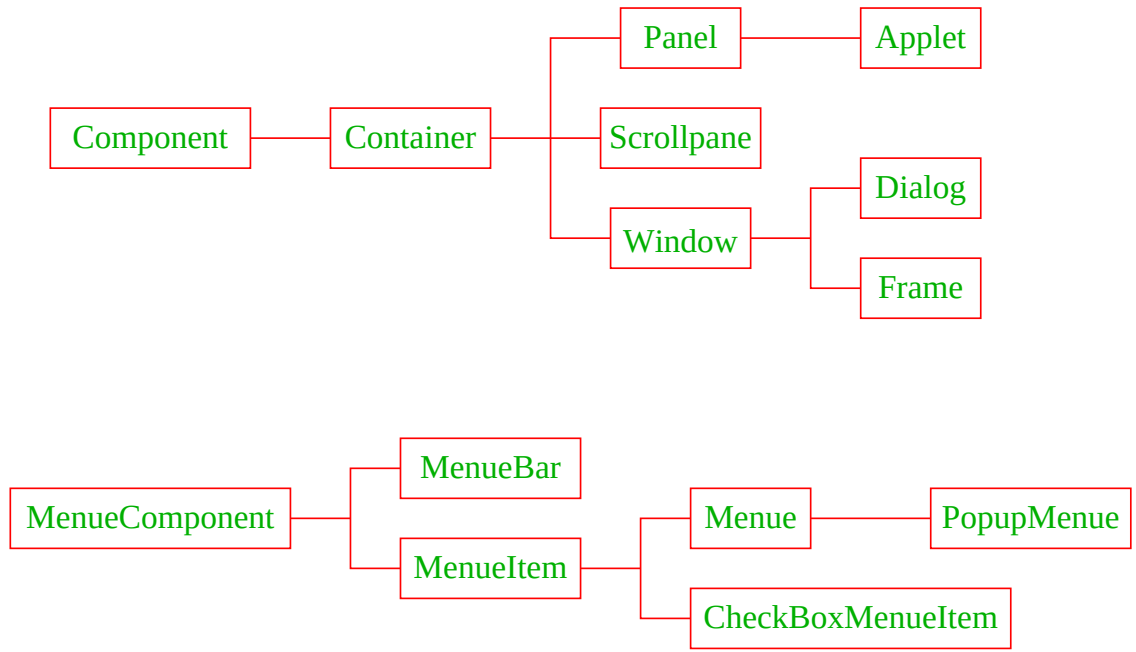
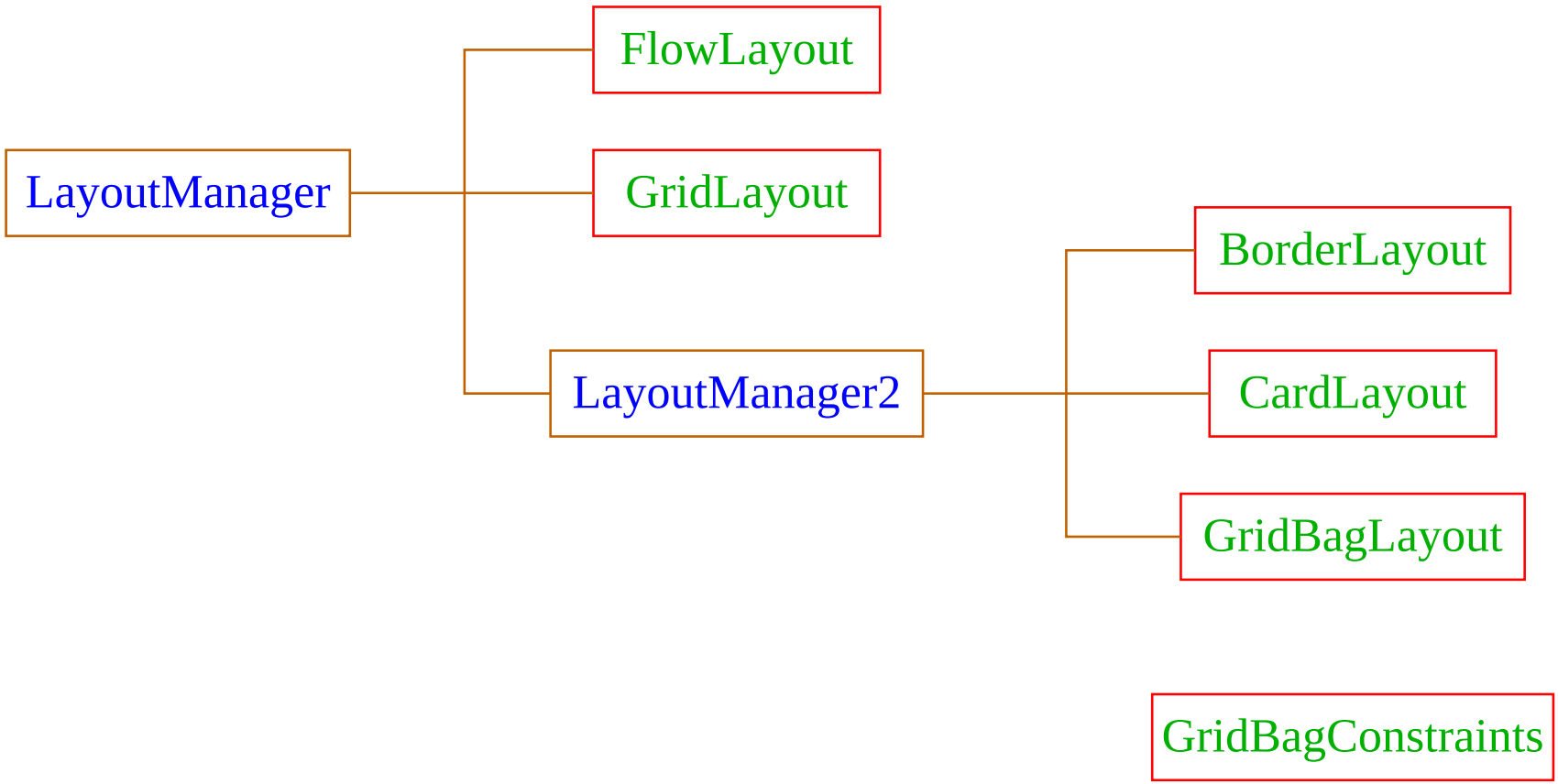
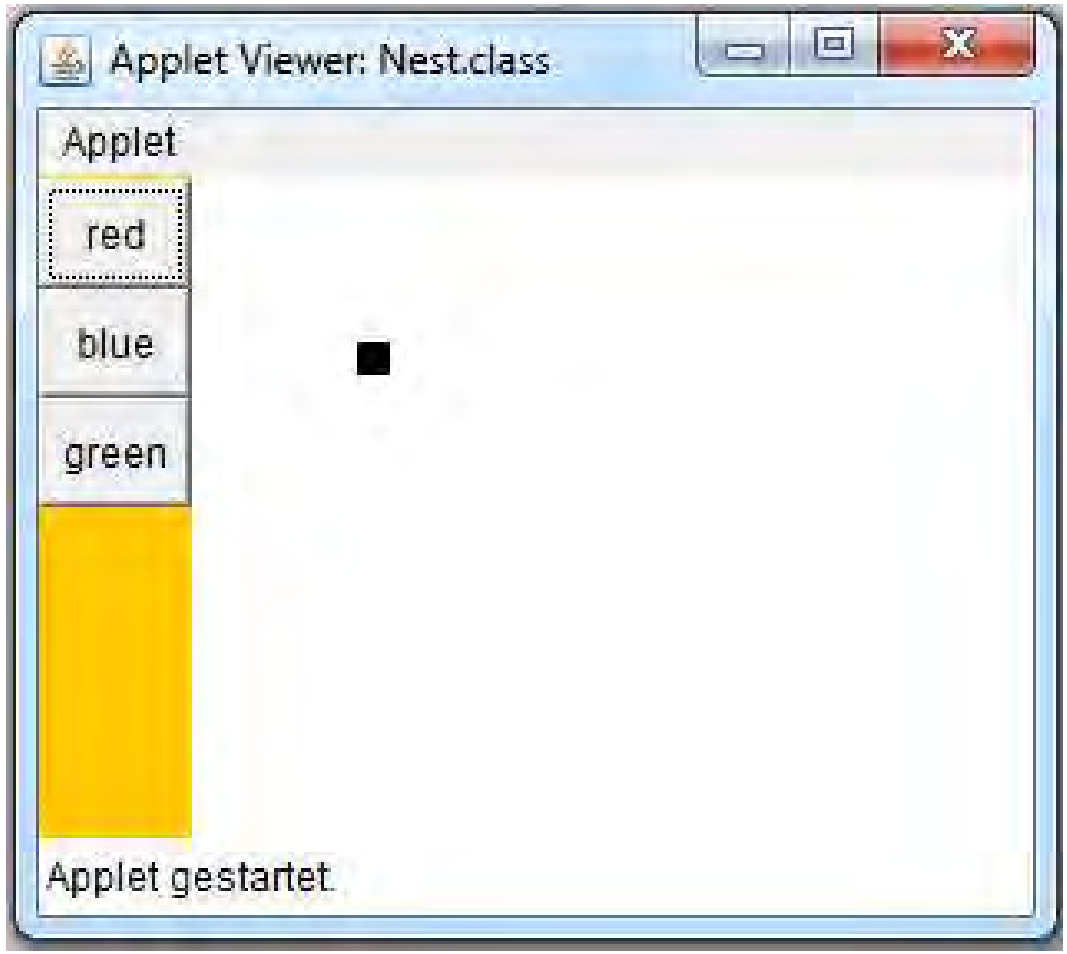
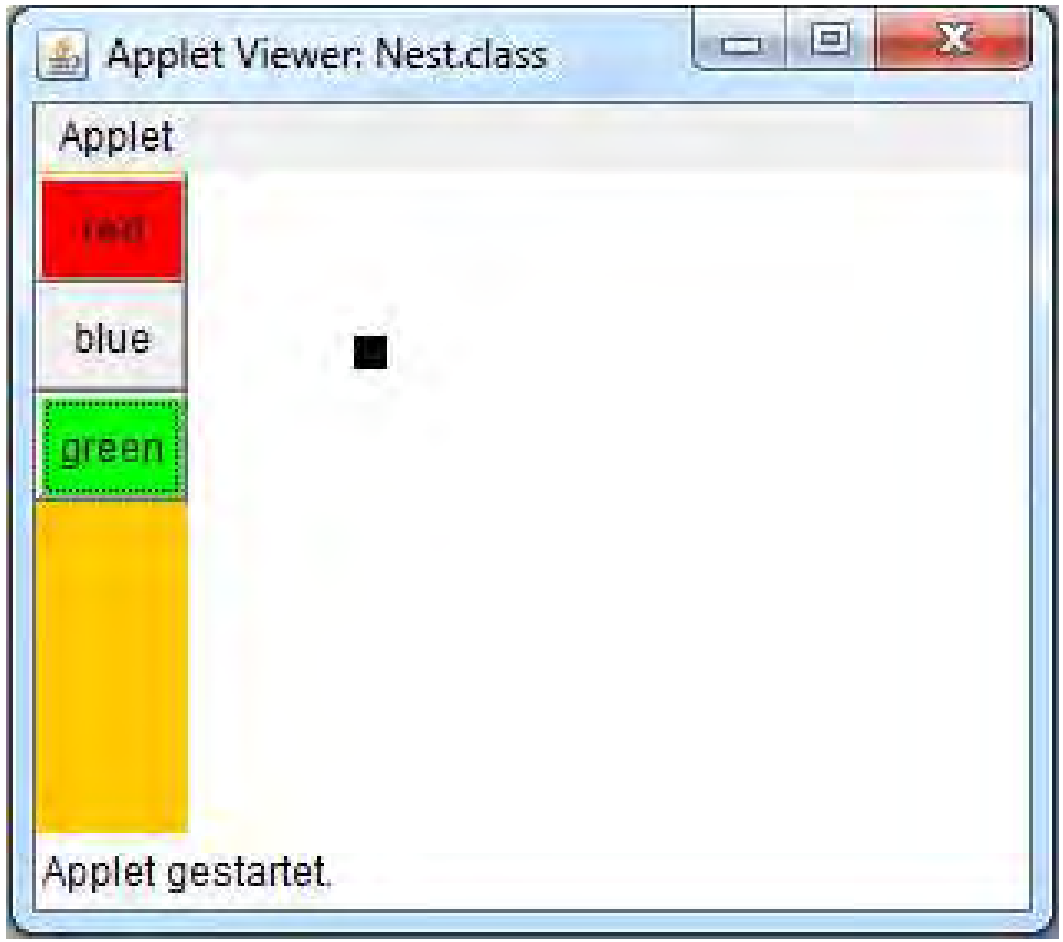
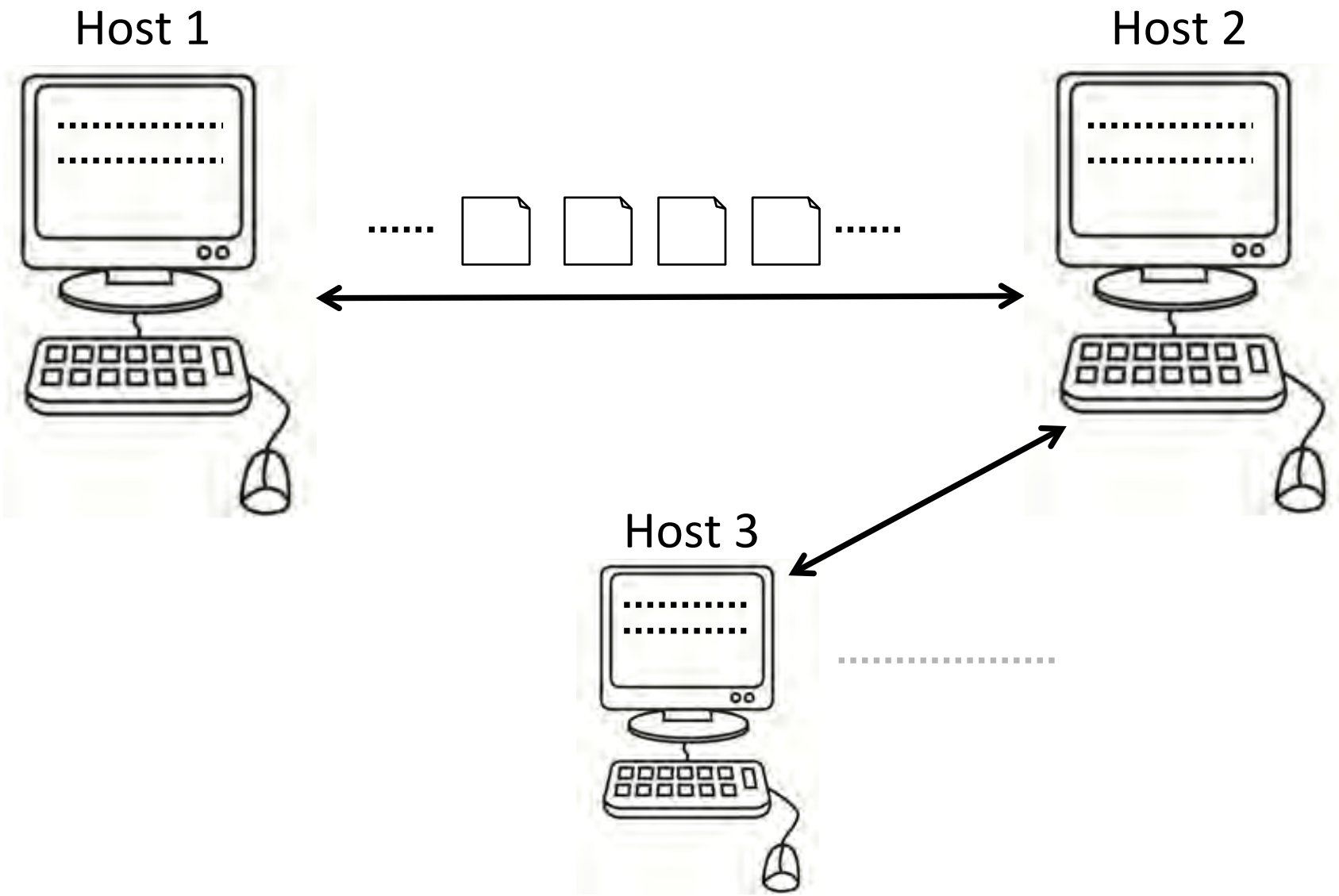
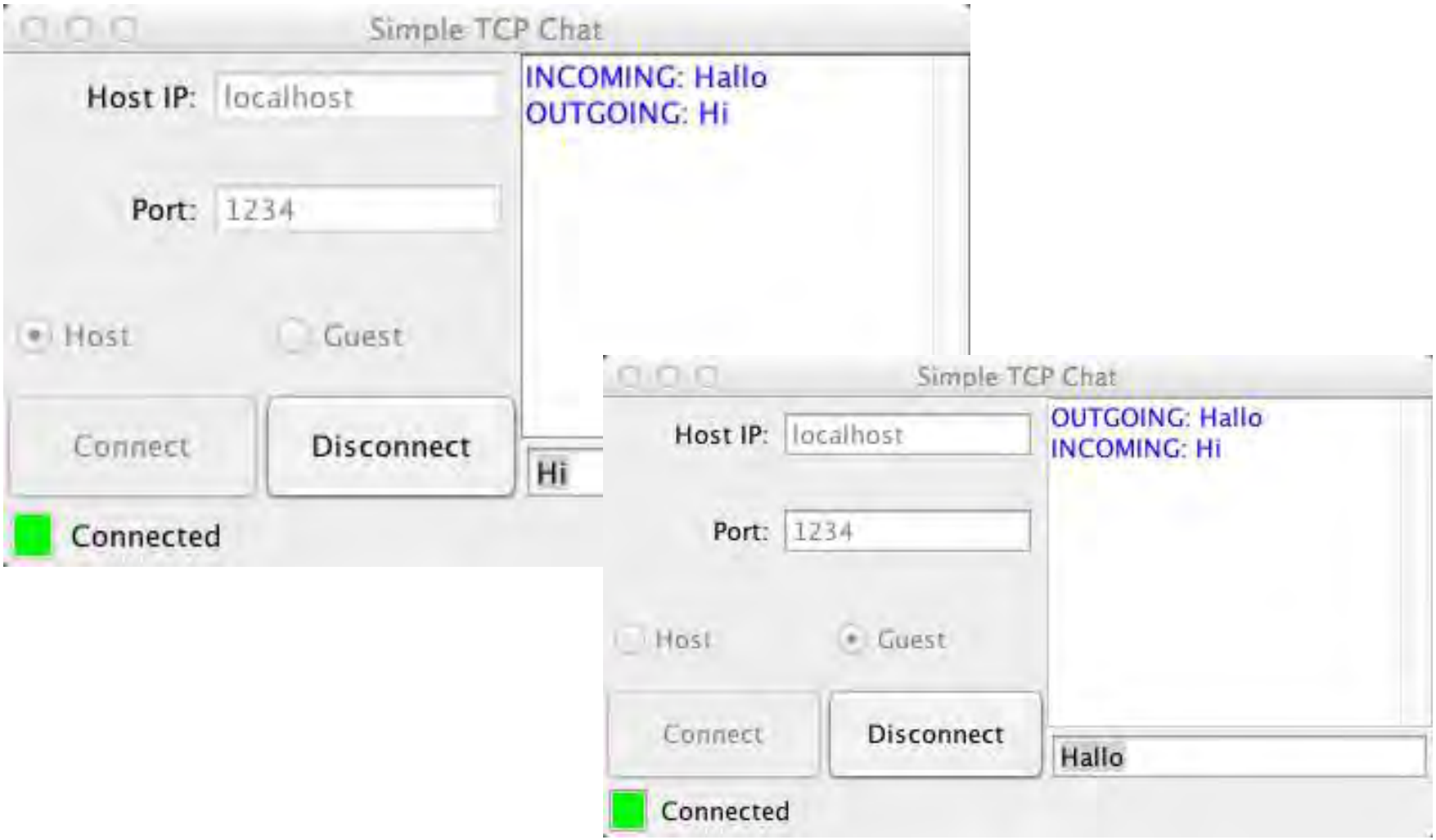
  public synchronized void startRead ()
                              throws InterruptedException {
        while (countReaders < 0) wait ();
        countReaders++;
 }
 public synchronized void endRead () {
        countReaders--;
        if (countReaders == 0)
              notifyAll ();
 }
 ...
```

Realisierung

```
...

public synchronized void startWrite ()
                          throws InterruptedException {
        while (countReaders != 0) wait ();
        countReaders = -1;
}

public synchronized void endWrite () {
        countReaders = 0;
        notifyAll ();
}
}
```

Diskussion

- Die Methoden `startRead()`, und `endRead()` sollen eine Leseoperation eröffnen bzw. beenden.
- Die Methoden `startWrite()`, und `endWrite()` sollen eine Schreiboperation eröffnen bzw. beenden.
- Die Methoden sind `synchronized`, damit sie selbst atomar ausgeführt werden.
- Die unterschiedlichen Modi eines RW-Locks sind mit Hilfe des Zählers `countReaders` implementiert.
- Ein negativer Zählerstand entspricht dem Schreib-Modus, während ein positiver Zählerstand die Anzahl der aktiven Leser bezeichnet ...

Diskussion II

- `startRead()` führt erst dann kein `wait()` aus, wenn das RW-Lock entweder frei oder im Lese-Modus ist.

  Dann wird der Zähler inkrementiert.

- `endRead()` dekrementiert den Zähler wieder.

  Ist danach das RW-Lock frei, werden alle wartenden Threads benachrichtigt.

Diskussion II

- `startRead()` führt erst dann kein `wait()` aus, wenn das RW-Lock entweder frei oder im Lese-Modus ist.

  Dann wird der Zähler inkrementiert.
- `endRead()` dekrementiert den Zähler wieder.

  Ist danach das RW-Lock frei, werden alle wartenden Threads benachrichtigt.
- `startWrite()` führt erst dann kein `wait()` aus, wenn das RW-Lock definitiv frei ist.

  Dann wird der Zähler auf -1 gesetzt.
- `endWrite()` setzt den Zähler wieder auf 0 zurück und benachrichtigt dann alle wartenden Threads.

Die HashTable mit RW-Lock

```
public class HashTable {
        private RW rw;
        private StringList[] a;
        private int n;
        public HashTable (int n) {
                rw = new RW();
                a = new StringList[n];
                this.n = n;
        }
        ...
```

Realisierung

```
public String lookup (String key) throws
                              InterruptedException {
      String result;
      rw.startRead();
      int i = Math.abs(key.hashCode())%n;
      if(!a[i])  //key existiert nicht
        result = null;
      else
        result = a[i].lookup(key);
      synchronized (rw) {
           rw.endRead();
           return result;
      }
}
...
```

Realisierung

```
public String lookup (String key) throws
                              InterruptedException {
      String result;
      rw.startRead();
      int i = Math.abs(key.hashCode())%n;
      if(!a[i])  //key existiert nicht
        result = null;
      else
        result = a[i].lookup(key);
      synchronized (rw) {
           rw.endRead();
           return result;
      }
}
...
```

- Da `lookup()` auch nicht weiß, wie mit einem interrupt umgegangen werden soll, wird diese Exception weitergeworfen.
- Damit zwischen Freigabe des RW-Locks und der Rückgabe des Ergebnisses kein Schreibvorgang möglich ist, wird die Rückgabe des RW-Locks mit dem `return` gekoppelt.

Realisierung

```
        public void update(String key, String value) throws
                                                InterruptedException {
            rw.startWrite();
            int i = Math.abs(key.hashCode())%n;
            if (!a[i]) a[i] = new StringList(key,value);
            else a[i].update(key,value);
            rw.endWrite();
        }
}
```

Realisierung

```
        public void update(String key, String value) throws
                                                InterruptedException {
            rw.startWrite();
            int i = Math.abs(key.hashCode())%n;
            if (!a[i]) a[i] = new StringList(key,value);
            else a[i].update(key,value);
            rw.endWrite();
        }
}
```

- Die Methode `update(String key, String value)` der Klasse `StringList` sucht nach einem Eintrag für `key`. Wird ein solcher gefunden, wird dort das Wert-Attribut auf `value` gesetzt.

  Andernfalls wird ein neues Listenobjekt für das Paar `(key,value)` angefügt.

Diskussion

- Die neue Implementierung unterstützt nebenläufige Lese-Zugriffe auf die HashTable.
- Ein einziger Lesezugriff blockiert aber Schreibzugriffe — selbst, wenn sie sich letztendlich auf andere Teillisten beziehen und damit unabhängig sind ??
- Genauso blockiert ein einzelner Schreibzugriff sämtliche Lesezugriffe, selbst wenn sie sich auf andere Teillisten beziehen ??

Diskussion

- Die neue Implementierung unterstützt nebenläufige Lese-Zugriffe auf die HashTable.
- Ein einziger Lesezugriff blockiert aber Schreibzugriffe — selbst, wenn sie sich letztendlich auf andere Teillisten beziehen und damit unabhängig sind ??
- Genauso blockiert ein einzelner Schreibzugriff sämtliche Lesezugriffe, selbst wenn sie sich auf andere Teillisten beziehen ??

$\Longrightarrow$ Eingrenzung der kritischen Abschnitte:
Unsynchronisiertes Finden der richtigen Liste;
Dann synchronisiertes Lesen/Schreiben in dieser Liste

Realisierung: `StringList[] a` wird `ListHead[] a`

```
class ListHead {
        private StringList list = null;
        private RW rw = new RW();
        public String lookup(String key) throws
                                    InterruptedException {
                rw.startRead();
                if(!list) result = null;
                else      result = list.lookup(key);
                synchronized (rw) {
                   rw.endRead();
                   return result;
                }
        }
        public void update(String key, String value) throws
                                    InterruptedException {
                rw.startWrite();
                if (!list) list = new StringList(key,value);
                else       list.update(key,value);
                rw.endWrite();
        }
}
```

Diskussion

- Jedes Objekt der Klasse `ListHead` enthält ein eigenes RW-Lock zusammen mit einer Liste (die `null` sein kann).
- Die Methoden `lookup()` und `update` wählen erst (unsynchronisiert) die richtige Liste aus, um dann geordnet auf die ausgewählte Liste zuzugreifen ...

```
// in der Klasse HashTable UNSYNCHRONISIERTER Zugriff:

public String lookup (String key) throws InterruptedException {
        int i = Math.abs(key.hashCode())%n;
        return a[i].lookup(key);  // darin synchronisierter Zugriff
}
public void update (String key, String value)
                        throws InterruptedException {
        int i = Math.abs(key.hashCode())%n;
        a[i].update(key, value); // darin synchronisierter Zugriff
}
```

Interrupts

Problem:

- Zeit ist kostbar.
- Gerne möchte man nach einiger Zeit das Warten abbrechen ...
- Oder einem Thread mitteilen, dass er nicht länger warten soll.

Interrupts I

- Analog zur Klassen-Methode `public void sleep(int msec)` der Klasse `Thread` gibt es die Objekt-Methoden

```
public void wait(int msec);
public void wait(int msec, int nsec);
```

  der Klasse `Object`, die das Warten auf `msec` Millisekunden bzw. `msec` Millisekunden und `nsec` Nanosekunden begrenzen.
- Jedem Thread kann ein Interrupt-Signal gesendet werden.
- Jeder Thread verfügt über ein Flag `boolean interrupted`, das anzeigt, ob er ein Interrupt-Signal erhielt.

Interrupts II

Die Objekt-Methode `public void interrupt();` der Klasse `Thread` sendet einem `Thread`-Objekt ein Interrupt-Signal mit den folgenden Effekt:

1. Das Flag `interrupted` wird gesetzt;
2. Der Zustand des Threads wird auf ready gesetzt – sofern er nicht running oder blocked ist;
3. Wartete der Thread vorher (z.B. auf die Beendigung eines anderen Thread oder ein `notify()`), wird er aus der Warteschlange entfernt.

Interrupts III

In Abhängigkeit vom vorherigen Zustand `oldState` führt der reaktivierte Thread die folgenden Aktionen aus:

```
switch(oldState) {
   case waiting:   lock();
        break;
   case sleeping:
   case joining:   throw (new
        InterruptedException ("operation␣interrupted"));
        break;
   case joiningIO: throw (new InterruptedIOException ());
        break;
}
```

Interrupts



Interrupts



Interrupts



Beispiel

```
public class Simple implements Runnable {
   public void run() {
      synchronized(this) {
         try { wait();}
         catch (InterruptedException e) { System.out.println(e);}
      }
   }
   public static void main(String[] args) throws InterruptedException {
      Thread t = new Thread(new Simple());
      t.start();        System.out.println("Simple thread started.");
      t.interrupt();    System.out.println("Interrupt sent ...");
      t.join();         System.out.println("Joining simple thread.");
   }
}  // end class Simple
```

Ausgabe

```
Simple thread started.
Interrupt sent ...
java.lang.InterruptedException: operation interrupted
Joining simple thread.
```

- Für ein `Runnable` der Klasse `Simple` wird ein Thread gestartet.
- Der neue Thread reiht sich in die Warteschlange für das `Simple`-Objekt ein und geht in den Zustand waiting.
- Bei Ankunft des Interrupt wird der Thread aufgeweckt.
- Mit einem `lock()` betritt er seinen kritischen Abschnitt, um dort eine `InterruptedException` zu werfen.
- Der Thread beendet sich, und weckt sämtliche Threads seiner Schlange `joiningThreads`, d.h. den Thread `main`, auf.

Bemerkungen

- Ein laufender Thread kann Interrupts ignorieren
- Ein wartender Thread kann die `InterruptedException` erst werfen, wenn er wieder das Lock erhielt
- Wartet ein Thread (im Zustand joiningIO) auf Beendigung einer IO-Operation (z.B. auf Eingabe vom Terminal) und erhält ein Interrupt, dann wirft er eine `InterruptedIOException`.
- Keine Exception entkommt aus einem Thread.
- Die Objekt-Methoden `run()` der Klasse `Thread` wie des Interface `Runnable` deklarieren keinerlei Exceptions.
- `InterruptedException`s und `InterruptedIOException`s müssen darum wie alle anderen Exceptions innerhalb des Thread abgefangen werden.

Anwendung: Timeout für Terminal-Input

- Eingabe auf `System.in` blockiert die Programmausführung.
- Um Warten auf Eingabe zeitlich zu begrenzen, starten wir einen Thread der Klasse `TimeOut`.
- Dieser soll nach einer gewissen Zeit das Warten unterbrechen.
- Wurde die Eingabe-Operation allerdings vorher erfolgreich beendet, soll der `TimeOut`-Thread selbst beendet werden.
- Nützliche Methoden der Klasse `Thread`:
  - `public static boolean interrupted()` — testet, ob das `interrupted`-Flag gesetzt ist, und setzt es auf false.
  - `public boolean isInterrupted()` — analog, nur wird das Flag nicht geändert

Code

```
import java.io.*;
public class TimeOut extends Thread {
   private Thread m;
   private int msec;
   public TimeOut(int n) {
      m = Thread.currentThread();
      msec = n;
   }
   public synchronized void run() {
      try {
         wait(msec);
         //If this has not been interrupted, interrupt m
         if(!Thread.interrupted()) m.interrupt();
      } catch (InterruptedException e) { }
   }
   public synchronized void kill() {
   //If the calling thread has not been interrupted, interrupt this
      if (!Thread.interrupted()) interrupt();
   }
   ...
```

Was passiert?

1. Fall:

Die Input-Operation wurde nicht unterbrochen. Dann wird die `synchronized`-Methode `kill()` aufgerufen.
Stellt diese doch noch einen Interrupt fest, ist offenbar `TimeOut` beendet. Nichts muss passieren.
Stellt diese keinen Interrupt fest, wartet der `TimeOut`-Thread (entweder in der Schlange `waitingThreads` oder hinter dem `wait()` auf die Fortsetzung des kritschen Abschnitts). Dann erhält er einen Interrupt. Kommt daraufhin der `TimeOut`-Thread in seinen kritischen Abschnitt, wird er sich einfach beenden.

Was passiert?

2. Fall:

Der `timeOut`-Thread beendet ungestört sein Warten und erwirbt erneut das Lock.
Dann sendet er einen Interrupt und beendet sich. Die Operation `kill()` testet das Interrupt-Flag mithilfe der Klassen-Methode boolean Thread.interrupted() – und setzt es dadurch zurück.

Code

```
    public static void echo(BufferedReader bu) {
       try { System.out.println(bu.readLine()); }
       catch (InterruptedIOException e) {
          System.err.println("Sorry, timeout!"); }
       catch (IOException e) {
          System.err.println("Sorry, general IO exception!"); }
    }
    public static void main(String[] args) {
       BufferedReader stdin = new BufferedReader
          (new InputStreamReader(System.in));
       TimeOut t = new TimeOut(5000);
       t.start();
       System.out.println("TimeOut thread started.");
       echo(stdin);
       t.kill();
       System.out.println("Timed input completed.");
    }
}   // end class TimeOut
```

... liefert z.B.

```
> java TimeOut
TimeOut thread started.
abc
abc
Timed input completed.
```

... oder:

```
> java TimeOut
TimeOut thread started.
Sorry, timeout!
Timed input completed.
```

Nochmals 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 (aber nicht übertreiben!).
- ... 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.

Und wie testet man so etwas?

... kommen Sie in meine Vorlesung im Hauptstudium!

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

11. Applets und GUIs

- Applets
  - Malen mit der Klasse Graphics
  - Schreiben mit Graphics
  - Animation
- Graphische Benutzer-Oberflächen
  - Einfache AWT-Komponenten
  - Ereignisse
  - Schachtelung von Komponenten
- Anwendung GUIs mit Netzwerkprogrammierung
- Netzwerk
  - Terminologie
  - Java-Klassen für Netzwerkprogrammierung
- Anwendung: Chat
  - GUI mit Java-Swing

Applets

Applets sind Java-Programme in einer HTML-Seite.

Ziele

- Audio-visuelle Gestaltung einer Seite
- Animation, d.h. Darstellen von Abfolgen von Bildern (evt. parallel zur Wiedergabe von Ton)
- Interaktion

Technik, Ästhetik, Sicherheit

- 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
  - Ggf. müssen im Java Control Panel (javacpl) Sicherheitseinstellungen geändert werden.
- Statt im Browser können Applets auch mithilfe des Programms `appletviewer` betrachtet werden.

Beispiel



HTML-Code

```
<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>
```

Hypertext Markup Language (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>`.

Tags – 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.

HTML: Abkürzungen und Sonderzeichen

- Bei leeren Elementen kann das End-Tag verkürzt werden:

  `<br />` — Zeilen-Umbruch;
  `<hr />` — horizontale Linie.

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

  `&lt;` — "<"
  `&gt;` — ">"
  `&auml;` — "ä"

Attribute

- 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 ausführbarer 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-Code des Applets

```
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!");
   }
   ...
```

Erläuterungen

- Ein neues Applet erweitert die Klasse `java.applet.Applet`.
- Ein Applet benötigt keine Klassen-Methode `main()`.
- 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 bei Start des Applets die Objekt-Methode `init()` aufgerufen.
  4. Dann wird die Objekt-Methode `start()` aufgerufen.
  5. ...

Applet-Code: paint

```
    ...
    public void paint(Graphics g) {
// g muss man nicht selbst erzeugen!
        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
```

Erläuterungen II

- 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`. Das muss man nicht selbst erzeugen!
- Auf dem sichtbaren (Unterklasse von) `Graphics`-Objekt kann nun gemalt werden ...

Events

- 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 die Beendigung der `start()`-Methode.
- Weitere Ereignisse beziehen sich auf die Maus oder die Tastatur.
- Das Applet fungiert als Consumer der Ereignisse.
- Es konsumiert ein Ereignis "Fenster wurde sichtbar(er)", indem es den Aufruf `paint(page)` (für das aktuelle Applet) ausführt.

  Dadurch wird das Bild wiederhergestellt.

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.

Malen mit `Graphics`

- Die Klasse `Graphics` stellt Methoden zur Verfügung, um einfache graphische Elemente zu zeichnen, z.B.:
  - `public void drawLine(int xsrc, int ysrc, int xdst, int ydst)` — 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])`.

Malen II

- 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

```
...
public void paint(Graphics g) {
  // ersetzt frühere Methode
  g.setPaintMode();
  AuxGraphics g1=new AuxGraphics(g);
  g1.drawRect(50,50,200,100,25);
  showStatus("... painting the Basic Applet!");
}

// inner class für breite Ränder
  private 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);
    }
  }
} // end of Applet Basic
```

Der Rand wird aus vier kleineren Rechtecken zusammengesetzt.

Im Browser



Java2D

- Seit der Java-Version 1.2 ist das `Graphics`-Objekt, auf dem das Applet malt, aus der Unterklasse `Graphics2D`.
- Diese Klasse bietet 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

Text

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 zunächst ein Applet, das Zeichen mit dem voreingestellten Font darstellt.

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
```

Erläuterungen

- `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.
- Effekt:



Schriftqualität

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



Schriftart 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
```

Font

- Eine Java-Schrift wird repräsentiert durch ein Objekt der Klasse `Font`.
- 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.

Schriften im Applet



Varianten

- 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)` und `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)

Dimensionierung

- 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.

Beispiel I

```
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);
      }
   ...
```

Beispiel II

```
    ...
    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
```

Erläuterungen

- Die Objekt-Methoden `public FontMetrics getFontMetrics()` und `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.

Im Appletviewer



Animationen

- 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.

Code

```
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!");
         }
      }
   }
   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
```

Erläuterungen

- 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.
- Hier nicht betrachtet: Das Ergebnis kann miserabel aussehen, wenn der Bildaufbau der Animation länger dauert. Das behebt man durch Buffering: Das Bild wird erst in einem (unsichtbaren) Puffer fertiggestellt und dann auf einen Schlag angezeigt.

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 ...

Übersicht

GUIs

Eine graphische Benutzer-Oberfläche (GUI) ist i.a. aus mehreren Komponenten zusammen gesetzt, die einen 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.

Übersicht


Benutzerin
Peripherie
Laufzeit–System


EventDispatcher
EventQueue
Listener

Übersicht


Peripherie
Laufzeit–System


EventDispatcher
EventQueue
Listener

Übersicht


Peripherie
Laufzeit–System
EventDispatcher
EventQueue
Listener

Übersicht


Peripherie
Laufzeit–System
EventDispatcher
EventQueue
Listener

Übersicht


Peripherie
Laufzeit–System
EventDispatcher
EventQueue
Listener
Listener

Übersicht


Peripherie
Laufzeit–System


EventDispatcher
EventQueue
Listener
Listener

Übersicht


Peripherie
Laufzeit–System


EventDispatcher
EventQueue
Listener
Listener
Listener

Ereignisse

- 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 (oder Objekte eines anderen GUI-Frameworks).
- 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 von Ereignissen

- 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.



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

Ein Knopf (Button)

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

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

Knopf II

```
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

Erläuterungen

- 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)` und `void add(Component c, int i)` fügen die Komponente `c` zum Applet hinten bzw. an der Stelle `i`) hinzu.

Knopf II

```
    ...
    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) {
        // es gibt nur einen Knopf, also nur ein ActionEvent!
        label.setText("Damn - you pressed it ...");
        System.out.println(e);
        remove(button);
    }
} // end of Applet FirstButton
```

Erläuterungen

- 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 Folgendes:
  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.

FirstButton



FirstButton



Mehrere Knöpfe und Events

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class FirstButton extends Applet  implements ActionListener {
   Label label;
   Button button, button2;
   public void init() {
      setBackground(Color.orange);
      setFont(new Font("SansSerif", Font.BOLD, 18));
      setLayout(new GridLayout(3,1));  // kommt später!
      label = new Label();
      label.setText("Mein␣erster␣Knopf!");
      add(label);
      button = new Button("Hier");
      button.setBackground(Color.white);
      button.addActionListener(this);
      add(button);
      button2 = new Button("Hier␣auch");
      button2.setBackground(Color.white);
      button2.addActionListener(this);
      add(button2);
   }
   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);
   }
```

Mehrere Knöpfe und Events

```
  ...
  public void actionPerformed(ActionEvent e) {
    if(e.getActionCommand().equals("Hier")) {
      label.setText("Toll! Knopf 1 mit Label "+e.getActionCommand());
      try {Thread.sleep(1000);} catch (Exception e1) {}
      label.setText("-------------------------------------");
      try {Thread.sleep(1000);} catch (Exception e1) {}
      label.setText("Mein erster Knopf!");
    }
    else {
      label.setText("Toll! Knopf2 mit Label "+e.getActionCommand());
      try {Thread.sleep(1000);} catch (Exception e1) {}
      label.setText("-------------------------------------");
      try {Thread.sleep(1000);} catch (Exception e1) {}
      label.setText("Mein erster Knopf!");
    }
    System.out.println(e);
  }
} // end of Applet FirstButton
```

Diverse LayoutManagers kommen später!

Graphische Darstellung

- 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 ...

Einfache AWT-Komponenten: Übersicht



Erläuterung

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 Scrollbar I

```
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");
   }
   ...
```

Erläuterung I

- `Scrollbar`-Objekte erzeugen `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`.

Erläuterung II

- Der Konstruktor `public Scrollbar(int dir,int init, int slide, int min, int max);` erzeugt einen `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.
- Dasselbe Objekt kann mehrmals als Listener auftreten

Erläuterung III

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 (kommt später).

BorderLayout

- 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.

Beispiel Scrollbar II

```
    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
```

Erläuterungen

- 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 ...

Scrollbar-Applet

Scrollbar-Applet



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);
   }
   ...
```

Erläuterungen

- `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.

Fortsetzung Choice-Liste

```
    ...
    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
```

Erläuterungen

- 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:

Zur Laufzeit



Zur Laufzeit



Ereignisse I

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

Ereignisse II



component.addActionlistener(listener);

Ereignisse II



Ereignisse II



Ereignisse II



Ereignisse II



Ereignisse III

- 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()`

Ereignisse IV

- 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



Listeners I

- 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.

Listeners II



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.

Beispiel MouseListener

```
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
```

Erläuterungen

- 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()` und `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.

MouseListener im Applet

```
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
```

Erläuterungen

- 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.

Zur Laufzeit



Zur Laufzeit



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



Erläuterungen

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.

Code I

```
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
...
```

Erläuterungen

- 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.

Code II

```
...
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
```

Erläuterungen

- Mithilfe der Objekt-Methoden `void setLocation(int x, int y)` und `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 `LayoutManager`s:

- 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
- Diese LayoutManagers finden Sie hier:
  http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
  Spielen Sie damit!

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



Erläuterungen

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 Knopfdruck soll die Farbe des Knopfs wechseln;
- Links daneben eine weiße Fläche mit einem schwarzen Quadrat.

Ausführung


Applet Viewer: Nest.class
Applet
red
blue
green
Applet gestartet

Ausführung


Applet Viewer: Nest.class
Applet
red
blue
green
Applet gestartet.

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.

Code I

```
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);
   }
   ...
```

Erläuterungen

- 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 ...

Code II

```
    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 Hintergrundfarbe orange ist, modifizieren wir die Farbe. Ansonsten setzen wir sie auf orange zurück.

Code III

```
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
```

Erläuterungen

- 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.

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

11. Applets und GUIs

- Applets
  - Malen mit der Klasse Graphics
  - Schreiben mit Graphics
  - Animation
- Graphische Benutzer-Oberflächen
  - Einfache AWT-Komponenten
  - Ereignisse
  - Schachtelung von Komponenten
- Anwendung GUIs mit Netzwerkprogrammierung
- Netzwerk
  - Terminologie
  - Java-Klassen für Netzwerkprogrammierung
- Anwendung: Chat
  - GUI mit Java-Swing

Anwendung GUIs: Netzwerkprogrammierung

Als Anwendung der GUI-Konzepte schreiben wir ein einfaches Chat-Programm.

Dazu müssen wir zunächst verstehen, wie zwei Nutzer (oder Computer) über ein *Netzwerk* miteinander kommunizieren können.

Netzwerk I

A (Computer)-Network is a

> *..collection of autonomous computers interconnected by a single technology...*
>
> *A. Tanenbaum*

- Komposition von (phyiskalisch) verteilt liegenden elektronischen Systemen, die durch eine Kommunikationsmechanismus verbunden sind.
- Zum Beispiel das *Internet*

Netzwerk II



Terminologie I

- Host: Rechner, der an ein Netzwerk angeschlossen ist.
- Netzwerkprotokoll: Regeln, wie Daten zwischen Hosts ausgetauscht werden dürfen
  - Internetprotokoll (IP) zur Vermittlung von Datenpaketen
  - Transmission Control Protocol (TCP): Verbindungsorientierte und zuverlässige Datenübertragung (Sicherstellung der Reihenfolge und Zustellung einzelner Datenpakete)
  - User Datagram Protocol (UDP): Verbindungslose Übertragung von Daten, keine Garantie der Zustellung

Terminologie II

- IP-Adresse: Nummer die einen Host weltweit eindeutig in einem Netzwerk identifiziert
  - IPv4: Zusammensetzung aus vier Dezimalzahlen (à 8bit) getrennt durch einen Punkt, z.B. 192.0.0.1
  - IPv6: Zusammensetzung acht hexadezimalen Blöcken (à 16bit) getrennt durch einen Doppelpunkt, z.B. 2001:db8:0:8d3:0:8a2e:70:7344
- Host-/Domainname: Symbolischer Name für eine IP-Adresse
  - Leichter zu merken
  - Lose Kopplung zwischen Host und symbolischem Namen
  - z.B. *localhost* $\leftrightarrow$ *127.0.0.1* oder *www.google.com* $\leftrightarrow$ *173.194.70.99*

Terminologie III

- Dienst: Eine Funktion, die von einem Server einem Client zur Verfügung gestellt wird
  - Server: Programm, das einen Dienst für einen Client bereitstellt
  - Client: Fordert vom Server einen Dienst an
  - *Client-Server-Modell*
- Port: Ist eine Nummer (16 bit, 0-65535), die einen Ein- oder Ausgabekanal für einen Prozess/Dienst auf einem Host spezifiziert, z.B. Port 80 für HTTP-Dienst.
- Socket: Fungiert als Schnittstelle zwischen einem Prozess/Dienst und einem Netzwerk(-protokoll).

Java-Klassen für Netzwerkprogrammierung I

- *java.net.ServerSocket*
  - Lauscht serverseitig auf einem Port
  - Verbindungsaufbau zur Übertragung von TCP/IP Paketen.
- *java.net.Socket*
  - Repräsentiert einen Endpunkt für die Kommunikation zwischen zwei Hosts
  - Zum Empfangen und Versenden von Datenpaketen (Server + Client)
- *java.net.DatagramSocket*: Zum Versand und Empfang von UDP-Paketen

Java-Klassen für Netzwerkprogrammierung II

Daten versenden: in den *java.io.OutputStream* des Sockets schreiben →*java.io.PrintWriter*

```
1  Socket skt = new Socket(hostIP, port);
2  //Printwriter mit Socket-Outputstream erzeugen
3  PrintWriter out = new PrintWriter(
4          skt.getOutputStream(), true);
5  //In Stream schreiben
6  out.print("test");
7
8  //Schließe offene Streams und Sockets
9  out.close();
10 skt.close();
```

Java-Klassen für Netzwerkprogrammierung III

Daten empfangen: aus dem *java.io.InputStream* des Sockets lesen
→ *java.io.BufferedReader*

```
1  ServerSocket srvr = new ServerSocket(1234);
2  //Wartet auf eine eingehende Verbindung
3  Socket skt = srvr.accept();
4  //BufferedReader mit Socket-Inputstream erzeugen
5  BufferedReader in = new BufferedReader(
6          new InputStreamReader(
7                  skt.getInputStream()) );
8  //Können wir lesen
9  while (!in.ready()) {}
10
11 //Lese eine Zeile
12 System.out.println(in.readLine());
13
14 //Offene Streams und Sockets schließen
15 in.close(); skt.close(); srvr.close();
```

Anwendung: Chat I[1]

Chat zwischen Host und Guest



[1] http://www.cise.ufl.edu/~amyles/tutorials/tcpchat/

Anwendung: Chat II

Die Implementierung besteht aus folgenden Klassen

- *Gui*.java
  - Repräsentiert das User Interface
  - Erzeugt und setzt zusammen grafische Elemente
  - Behandelt Benutzerinteraktionen, z.B. Mausklicks
- *Chat*.java
  - Steuert den Programablauf
  - Erzeugt und beendet die Verbindung zwischen den Teilnehmern
  - Steuert den Informationsfluss zwischen GUI und Sockets zum Senden und Empfangen von Nachrichten
- *ConnectionStatus*.java, *RunChat*.java, *ActionAdapter*.java: Hilfsklassen

GUI mit Java-Swing I

- *java.swing*-Paket: Bibliothek zur Programmierung grafischer Benutzeroberflächen
  - Erweiterung von *java.awt*
  - Modularer Aufbau
  - *Pluggable Look-and-Feel*, Erscheinungsbild und Verhalten von grafischen Elementen ist austauschbar, auch zur Laufzeit ohne Neustart der Applikation
  - *Leichtgewichtige Komponenten*, Rendering grafischer Elemente unabhängig von Betriebssystemkomponenten, Elemente sehen auf allen Plattformen gleich aus, keine direkte Abb. von GUI-Komponenten auf Platform (*Peer*-Klassen)
  - *Accessibility*: Interaktionstechniken für Menschen mit Behinderung, z.B. Lesegeräte für Blinde

GUI mit Java-Swing II

- Analoge grafische Komponenten wie im *java.awt*-Paket
- *java.swing.JFrame*: Repräsentiert ein Fenster
  - *setDefaultCloseOperation*: Legt fest was beim Schießen des Fenster passieren soll.
  - *setContentPane*: Set den Inhalt des Fenster
  - *setSize*: Größe des Fensters festlegen
  - *getPreferredSize*: Liefert die Optimale Größe zu diesem Fenster, initiale Berechnung durch LayoutManager.
  - *setLocation*: Bewegt das Fenster an eine gewünschte Position
  - *setVisible*: Macht das Fenster sichtbar auf dem Bildschirm

GUI mit Java-Swing III

- *java.swing.JPanel*: Container zur Gruppierung grafischer Elemente
  - *add*: Fügt grafische Elemente zum Panel hinzu
- *java.swing.JScrollPane*: Container für zur Gruppierung grafischer Elemente, mit Scrollbalken
- *java.swing.JTextArea*: Zur Darstellung von mehrzeiligen Text.
  - *setLineWrap*: Automatischer Zeilenumbruch bei zu langen Zeilen.
  - *setEditable*: Legt fest ob Element editierbar ist.
  - *setForeground*: Schriftfarbe ändern.

GUI mit Java-Swing IV

- *java.swing.JTextField*: Zur Darstellung von einzeiligem Text.
  - *setBackground*: Legt die Hintergrundfarbe fest.
- *java.swing.JLabel*: Darstellung von Text oder Bildern, nicht editierbar, reagiert nicht auf Events
- *java.swing.JButton*: Schaltfläche um eine Aktion/Event auszulösen
  - *setActionCommand*: Legt den Action-Code fest, welcher in einem Event-Handler genutzt werden kann.
  - *addActionListener*: Fügt einen Actionlistener zu der Schaltfläche hinzu um auf Events zu reagieren.
  - *setEnabled*: Aktiviert die Schaltfläche.
- *java.swing.JRadioButton*: Sich gegenseitig ausschließende Kontrollelemente (Kästchen)

GUI mit Java-Swing V

- *java.swing.ButtonGroup*: Erzeugt einen multiple-exclusion Scope, nur ein Button darf aktiviert werden.
- *java.awt.event.FocusAdapter*: Implementiert *java.awt.event.FocusListener* um auf Focus-Events zu reagieren, z.B. *focusLost*.
- *java.awt.event.ActionListener*: Listener Interface um auf Events zu reagieren.
  - *actionPerformed*: Wird bei Events aufgerufen.
- *SwingUtilities.invokeLater*: Reiht übergebenen Thread in EventDispatcher-Thread ein. Wir nutzen es zum synchronen Update von Swing-Elementen.

Fazit

- Es existieren viele weitere Frameworks in Java zur Implementierung verteilter Anwendungen: RMI, CORBA, Apache MINA,...
  - Unterstützen unterschiedliche Technologien und Protokolle
  - Kapseln deren Komplexität
- Probieren Sie's aus!

Überblick

Teil 1:	Einführung
Teil 2:	Objektbasierte Programmierung
Teil 3:	Kontrollstrukturen
Teil 4:	Felder
Teil 5:	Einige Abstrakte Datentypen
Teil 6:	Objektorientierung
Teil 7:	Rekursion
Teil 8:	Fortgeschrittene Programmierkonstrukte
Teil 9:	Ein weiteres größeres Beispiel
Teil 10:	Nebenläufigkeit
Teil 11:	Applets und GUIs
Teil 12:	Beyond Java

Übersicht

12. Beyond Java

Motivation

Wie verarbeitet die Java-VM eigentlich in Bytecode compilierten Java-Code? Und wie wird der Java-Code eigentlich in Java-Bytecode übersetzt?
Dazu müssen wir verstehen,

- Wie legale (Java-)Programme aussehen.
  Wir zeigen deshalb, wie man die Syntax einer Programmiersprache beschreibt.
- Wie die virtuelle Maschine aufgebaut ist und wie sie Bytecode-Instruktionen ausführt.
  Wir zeigen deshalb, wie eine VM funktioniert.
- Wie man Java-Programme in Instruktionen übersetzt, die die VM verstehen kann.
  Wir zeigen deshalb für eine einfache Variante von Java (MiniJava), wie Java-Sourcecode in Bytecode übersetzt wird.

Syntax: "Lehre vom Satzbau"

- Formale Beschreibung des Aufbaus der "Wörter" und "Sätze", die zu einer Sprache gehören;
- Im Fall einer Programmier-Sprache Festlegung, wie Programme aussehen müssen.

Hilfsmittel bei natürlicher Sprache

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

Hilfsmittel bei Programmiersprachen 1

- 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?

Hilfsmittel bei Programmiersprachen 2

- 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 ...

Semantik

Die Bedeutung eines Programms ist

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

Semantik

Die Bedeutung eines Programms ist

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

Achtung!

Ist ein Programm syntaktisch korrekt, heißt das noch lange nicht, dass es auch das "Richtige" tut, d.h. semantisch korrekt ist !!!

Reservierte Wörter

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

Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten Zeichen:

letter ::= $ | _ | a | ... | z | A | ... | Z

digit ::= 0 | ... | 9

Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten Zeichen:

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

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

Was ist ein erlaubter Name?

Schritt 2: Angabe der Anordnung der Zeichen:

name ::= letter (letter | digit)*

- Erst kommt ein Zeichen der Klasse letter, dann eine (eventuell auch leere) Folge von Zeichen entweder aus letter oder aus digit.
- Der Operator "*" bedeutet "beliebig häufige 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.
  ```

Beispiele

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

... sind keine legalen Namen.
  ```

Beispiele

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

... sind keine legalen Namen.

Achtung:

Reservierte Wörter sind als Namen verboten !!!
  ```

Ganze Zahlen

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

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

Ganze Zahlen

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

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

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

Beispiele

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

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen (-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

und Klammern aufgebaut sind, heißen reguläre Ausdrücke[2] (↑Automatentheorie).

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

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

Reguläre Ausdrücke

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Wörtern 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;
- $$\begin{array}{lll} \text{exp} & ::= & (\texttt{e}|\texttt{E})(+|-)?\ \text{digit}\ \text{digit}^* \\ \text{float} & ::= & \text{digit}\ \text{digit}^*\ \text{exp} \quad | \\ & & \text{digit}^*\ (\text{digit}\ .\ |\ .\ \text{digit})\ \text{digit}^*\ \text{exp}? \end{array}$$
  – alle Gleitkomma-Zahlen ...

Compilierung Phase 1: Scanner

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).

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`

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.

MiniJava (1)

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

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

MiniJava (2)

expr	::=	number | name | (expr) |
		unop expr | expr binop expr
unop	::=	−
binop	::=	− | + | ∗ | / | %

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

MiniJava (3)

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

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

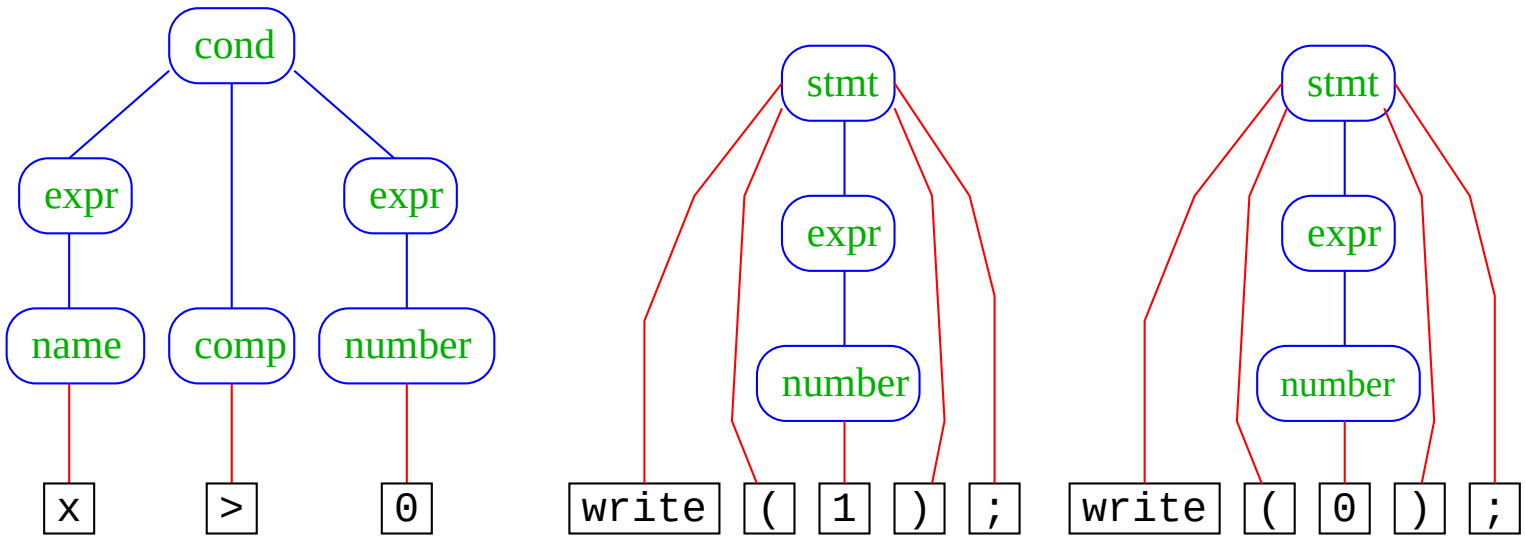
Beispiel

```
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

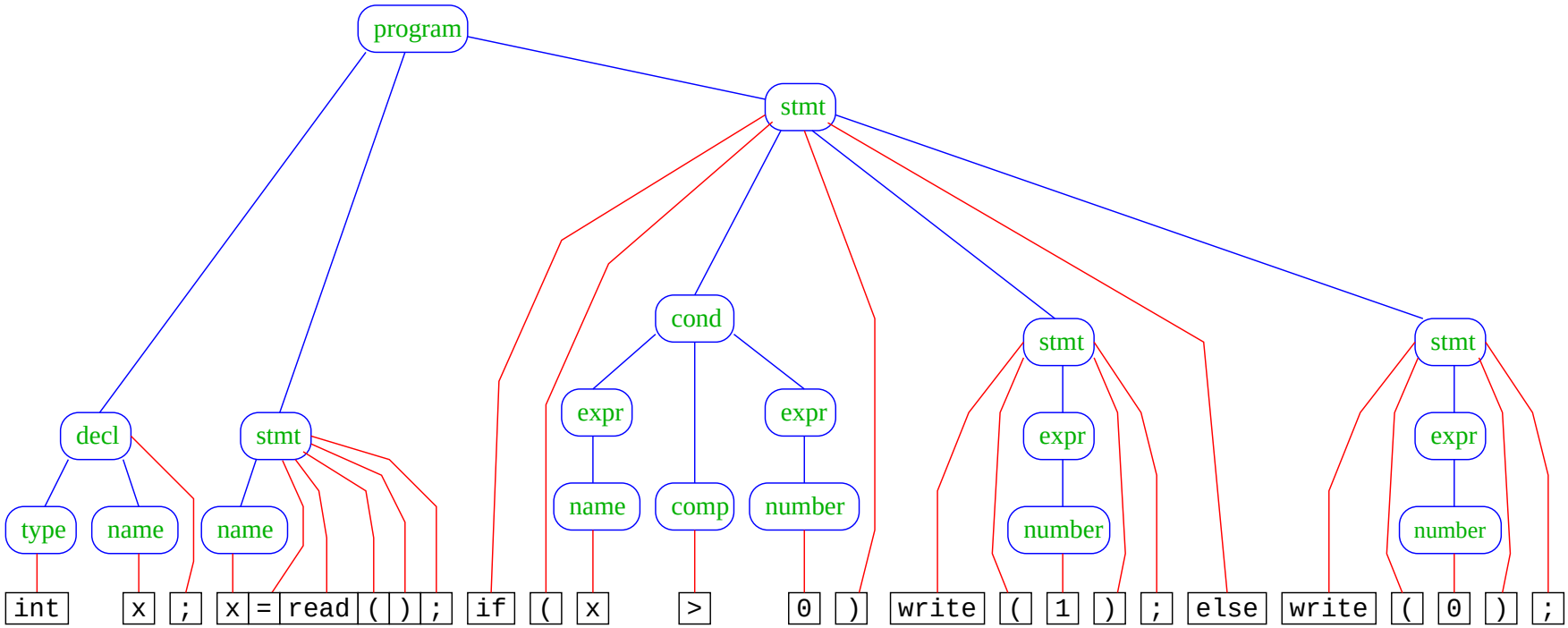
Syntax-Bäume für `x > 0` sowie `write(0);` und `write(1);`



Blätter: Wörter/Tokens

Innere Knoten: Namen von Programm-Bestandteilen

Syntax-Bäume



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.
- In Phase 2 des Compilierens baut der Parser den Syntaxbaum.

Historisches

Noam Chomsky,
MIT

John Backus, IBM
Turing Award
(Erfinder von Fortran)

Peter Naur,
Turing Award
(Erfinder von Algol60)

Achtung

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

Beispiel:

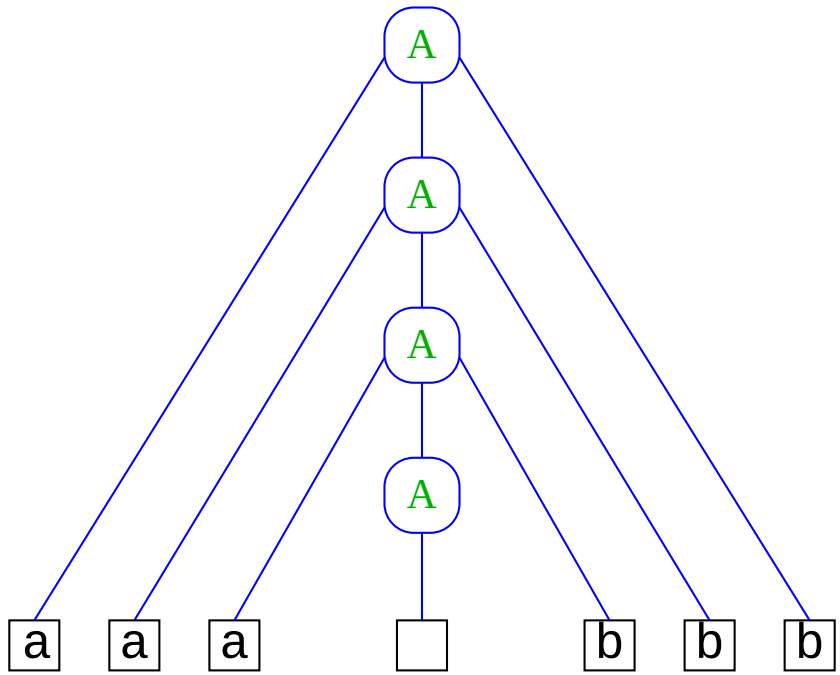
$$\mathcal{L} = \{\epsilon, \mathtt{ab}, \mathtt{aabb}, \mathtt{aaabbb}, \ldots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

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

Beispiel

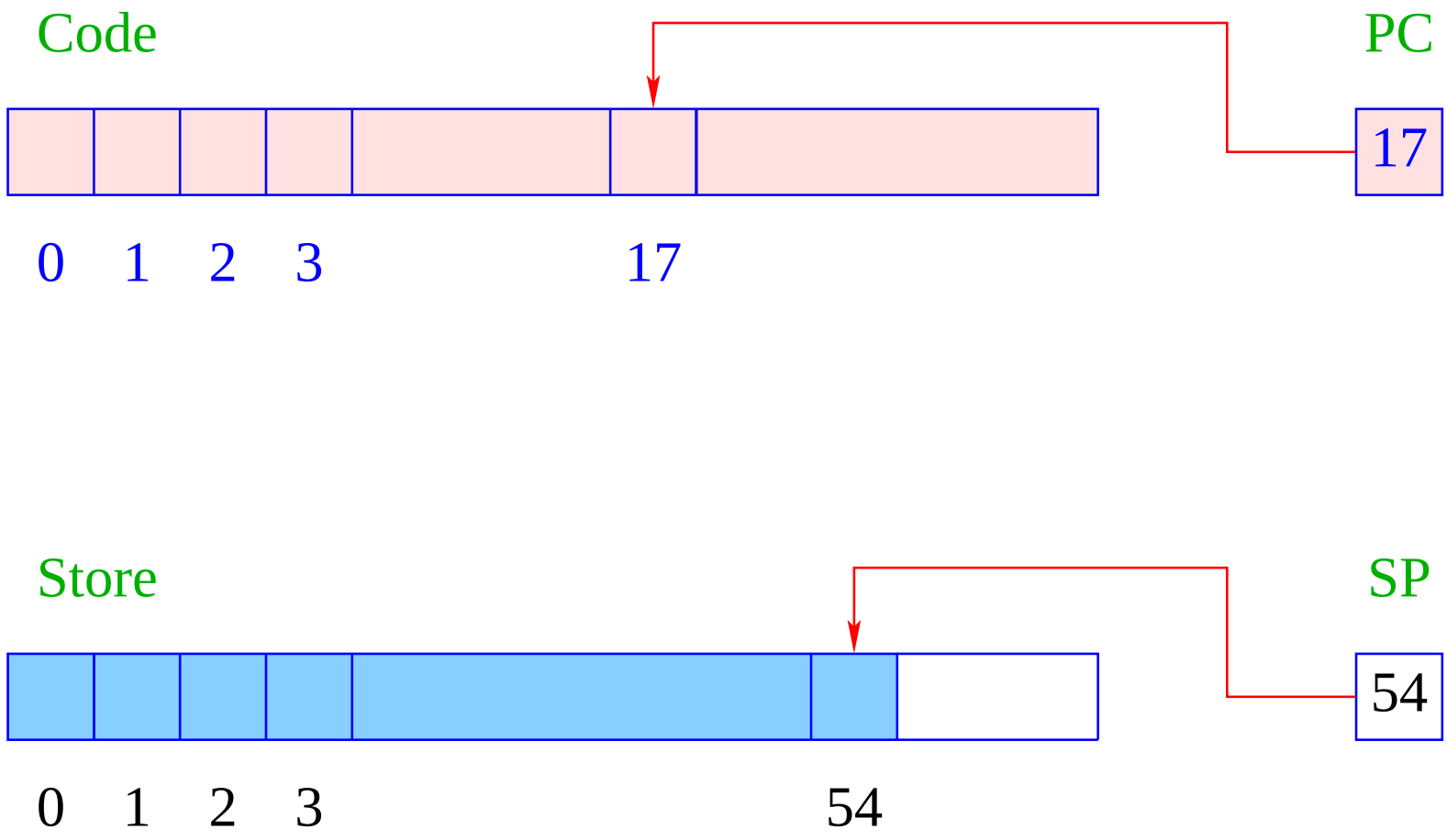
Syntax-Baum für das Wort `aaabbb` :



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

Übersicht

Architektur der JVM



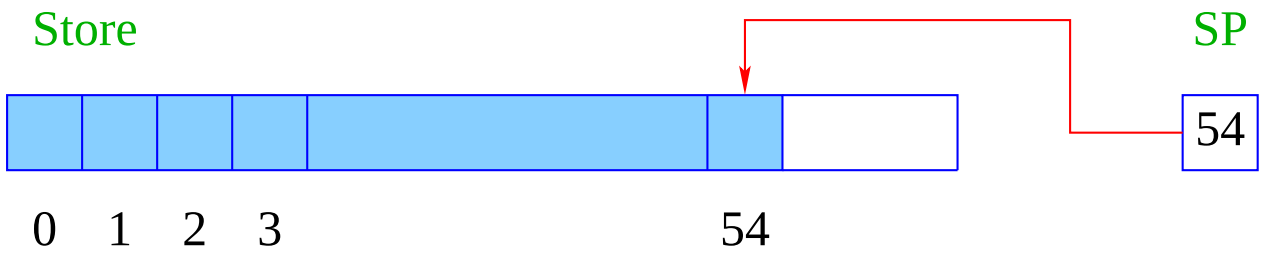
Komponenten

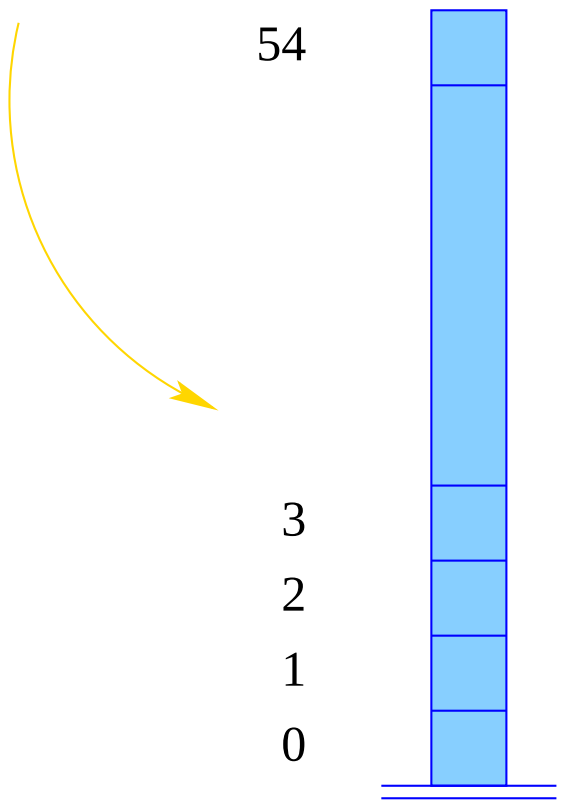
Code	=	enthält JVM-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter –
		zeigt auf nächsten auszuführenden Befehl;
Store	=	Speicher für Daten;
		jede Zelle kann einen Wert aufnehmen;
SP	=	Stack-Pointer –
		zeigt auf oberste belegte Zelle.

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





Store = Stack

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
```

Ein Beispiel-Programm

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

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

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

Semantik der Instruktionen

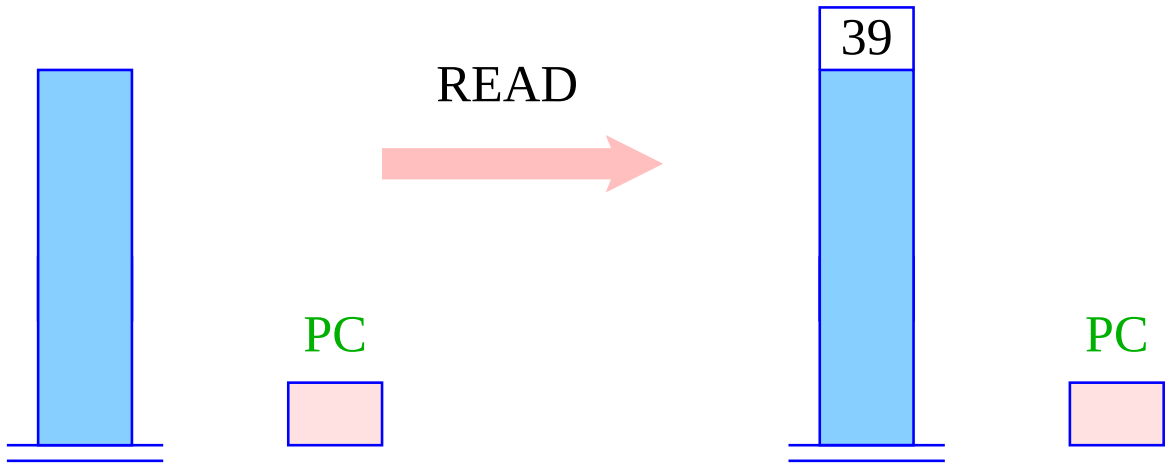
Bevor wir erklären, wie man MiniJava in JVM-Code übersetzt, erklären wir, was die einzelnen Befehle bewirken.

Idee:

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

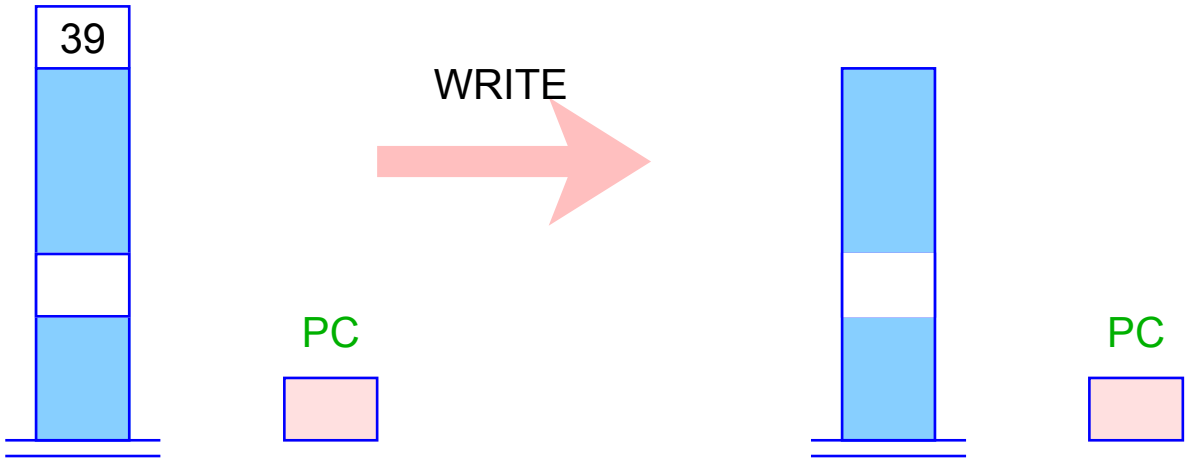
Betrachten wir als Beispiele die IO-Befehle READ und WRITE.

read



... falls 39 eingegeben wurde

write

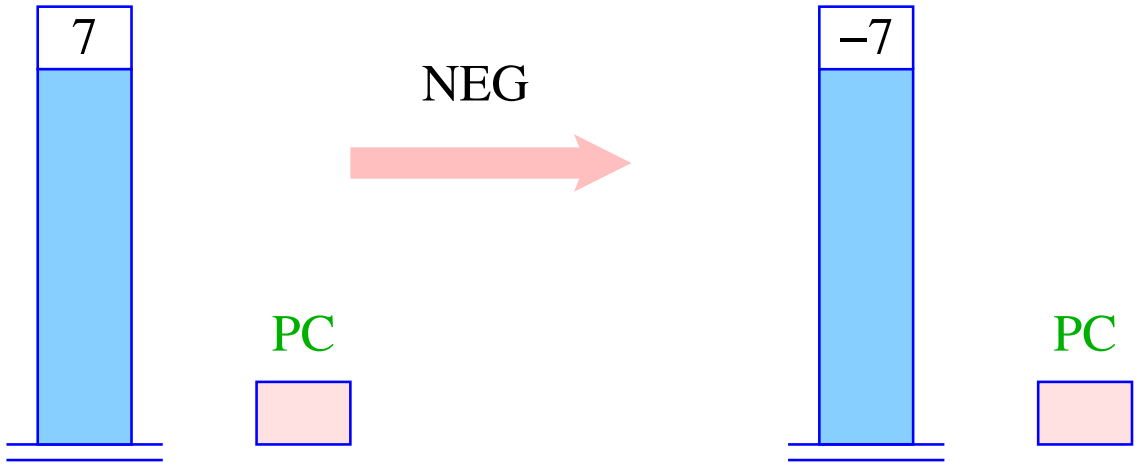


... wobei 39 ausgegeben wird

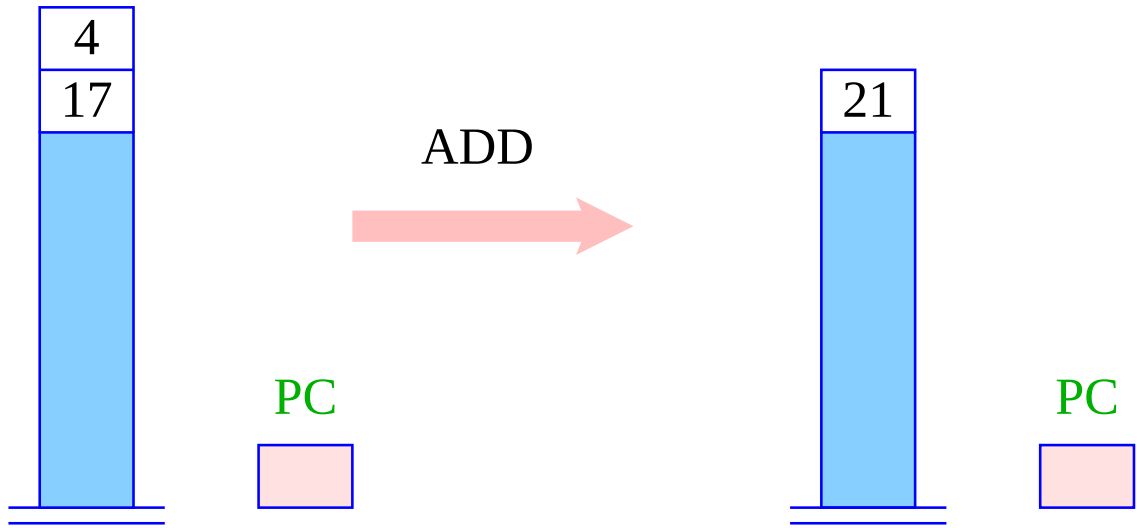
Arithmetik

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

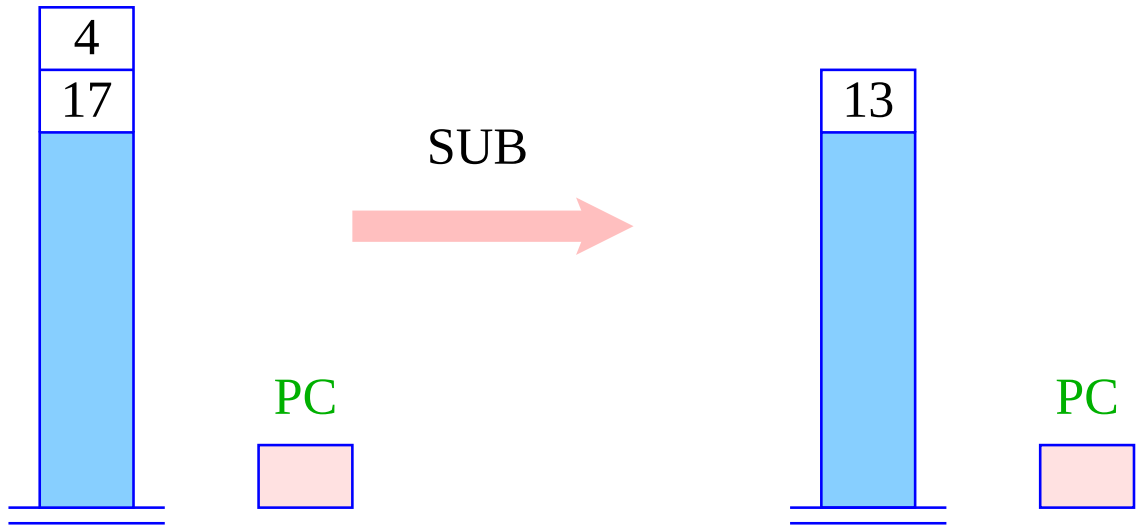
Negation



Addition



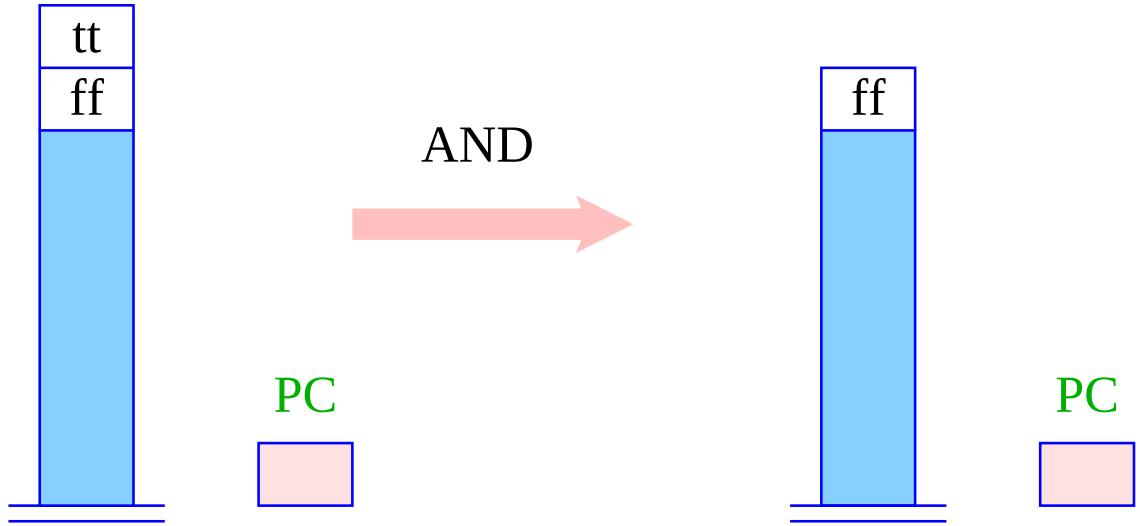
Subtraktion



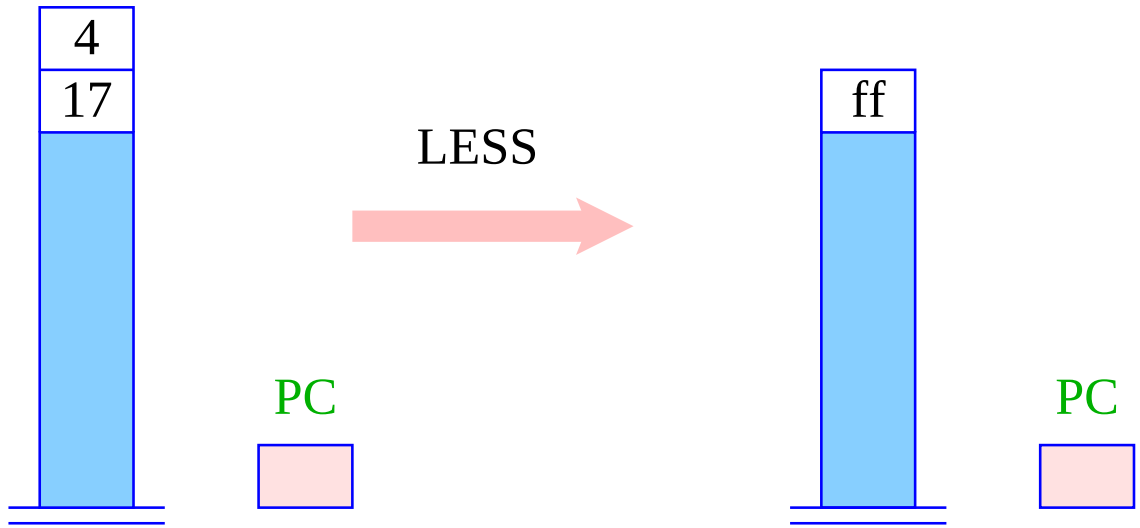
Et cetera

- 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.

Logisches Und



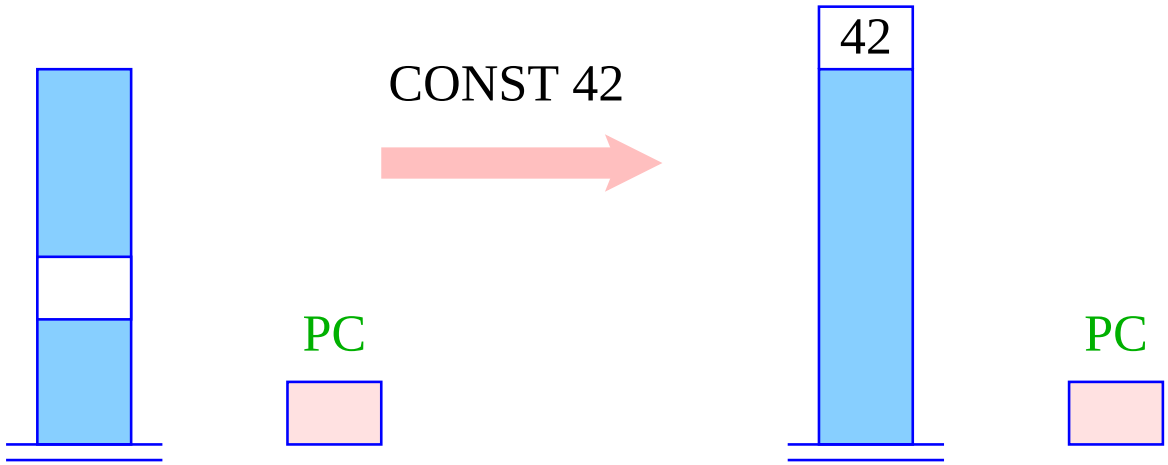
Kleiner oder gleich



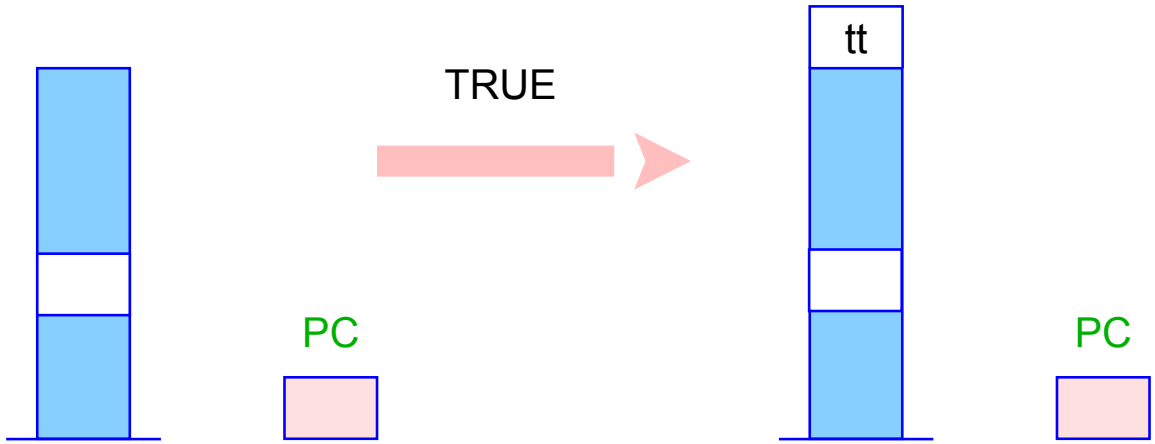
Laden und Speichern

- 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.
- Die i-te Zelle kann außerhalb des Stacks liegen (und tut das normalerweise).

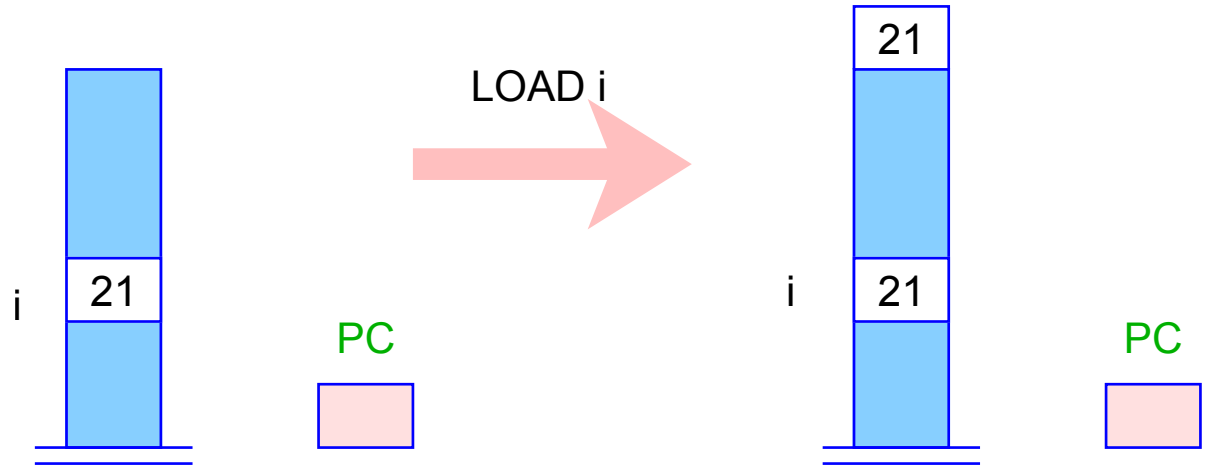
Numerische Konstanten



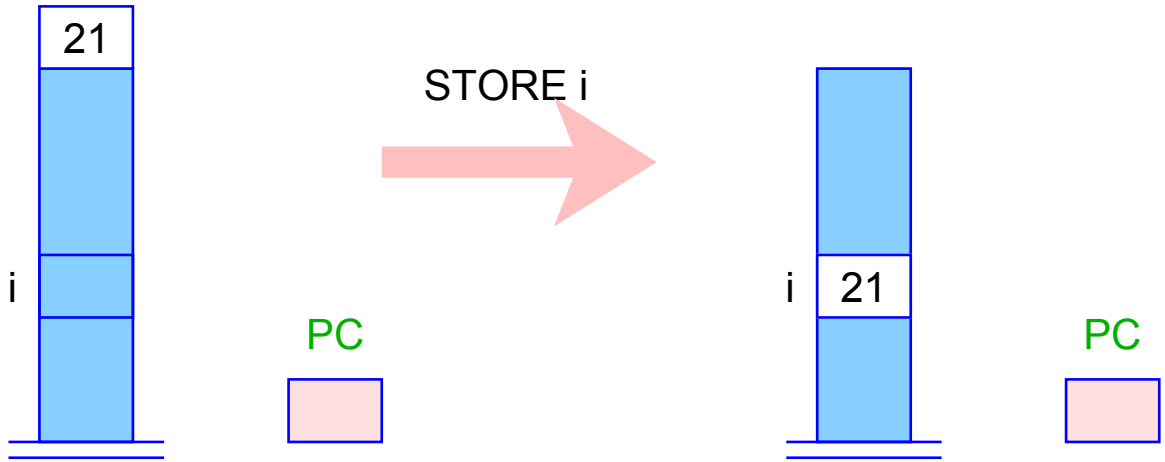
True



Wert lesen



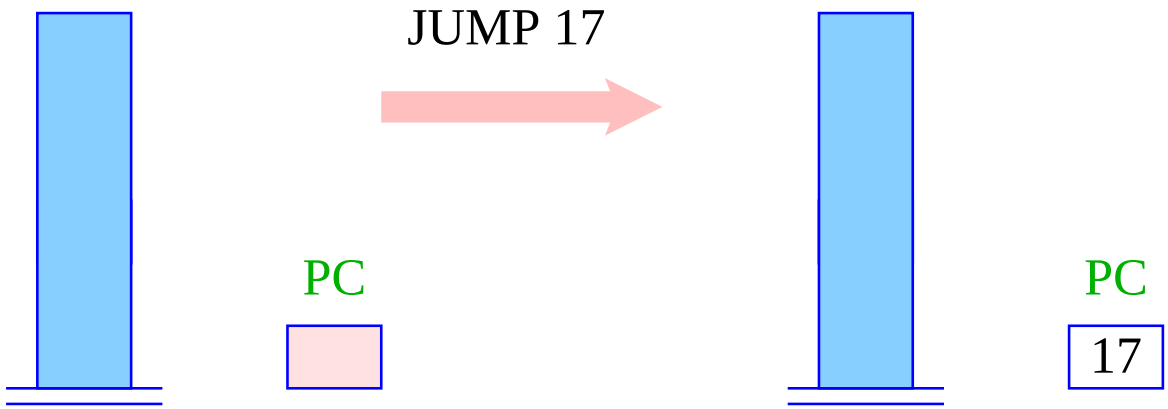
Wert speichern



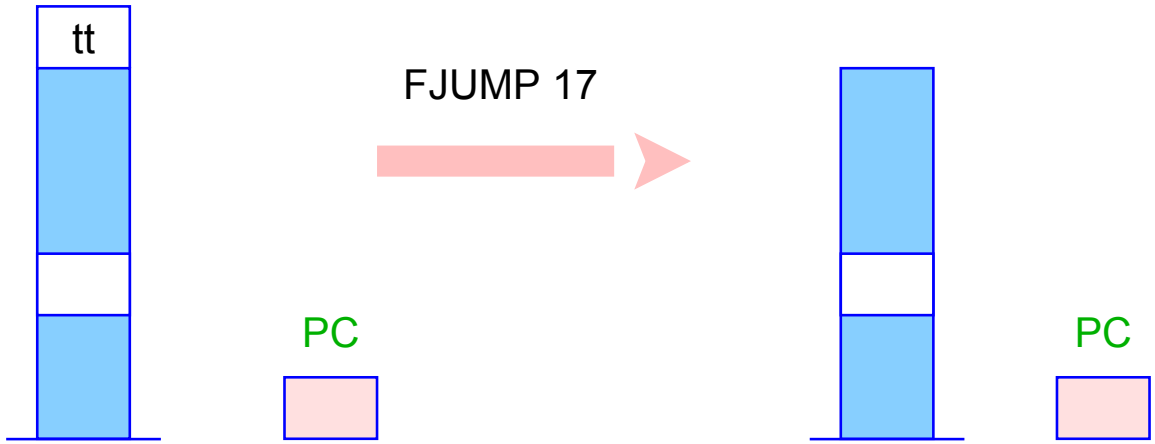
Sprünge

- 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.

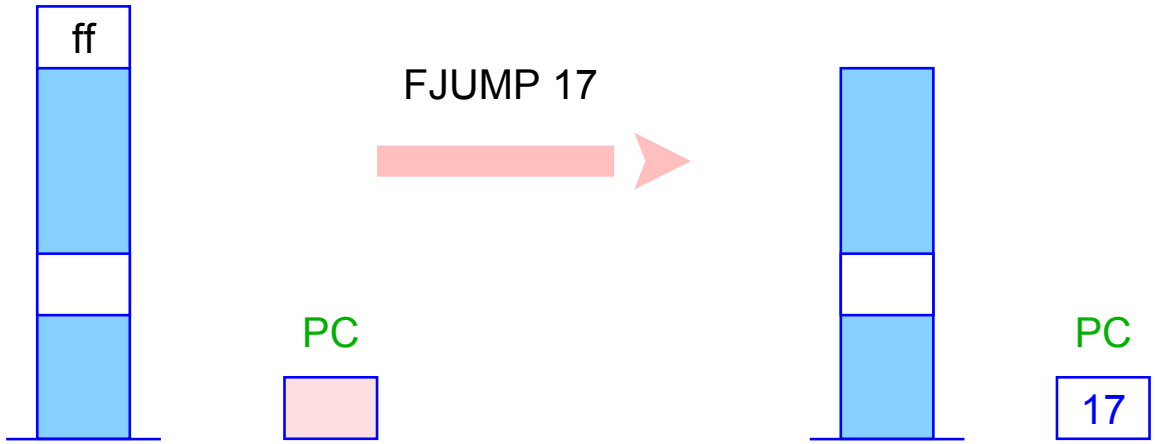
Unbedingter Sprung



Bedingter Sprung



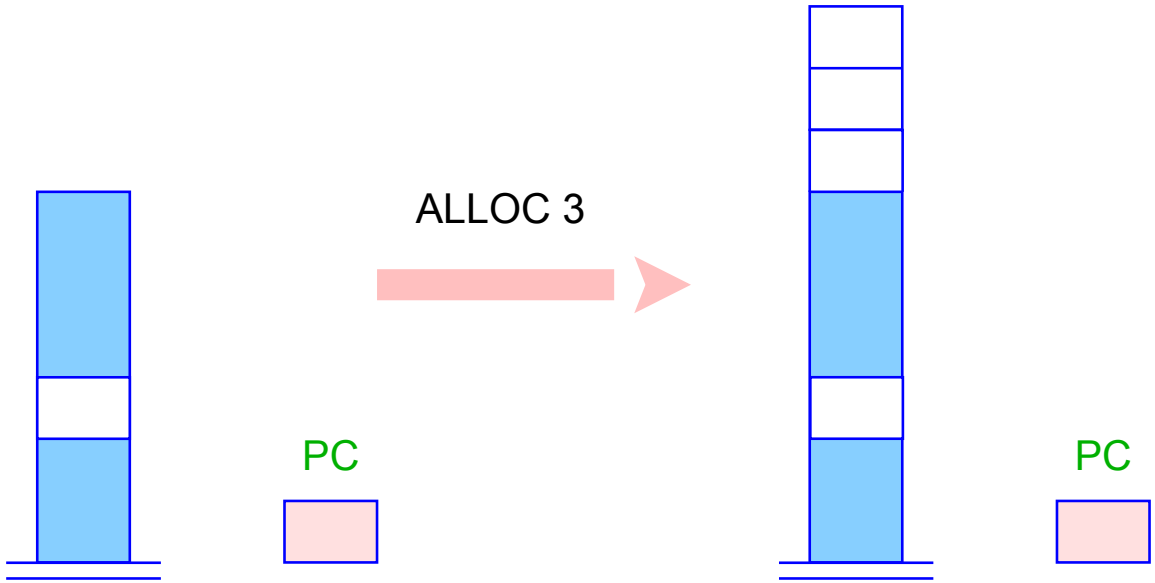
Bedingter Sprung



Allokation 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.
- Funktioniert so einfach nur, wenn alle Deklarationen am Anfang des Programms stehen!
- In der Definition von MiniJava haben wir auf Deklarationen verzichtet ... jetzt nehmen wir an, dass sie am Anfang des Programms stehen.

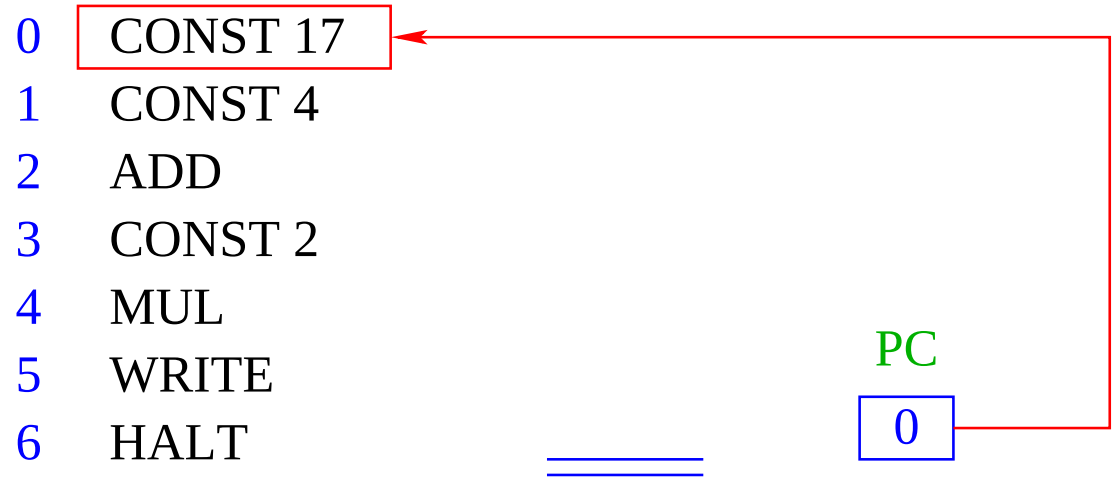
Speicherallokation



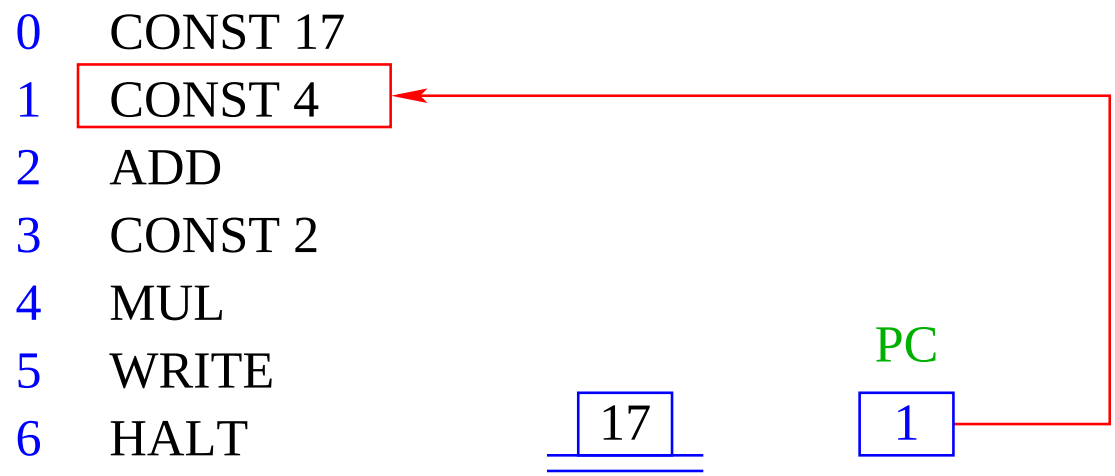
Ein Beispiel-Programm

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

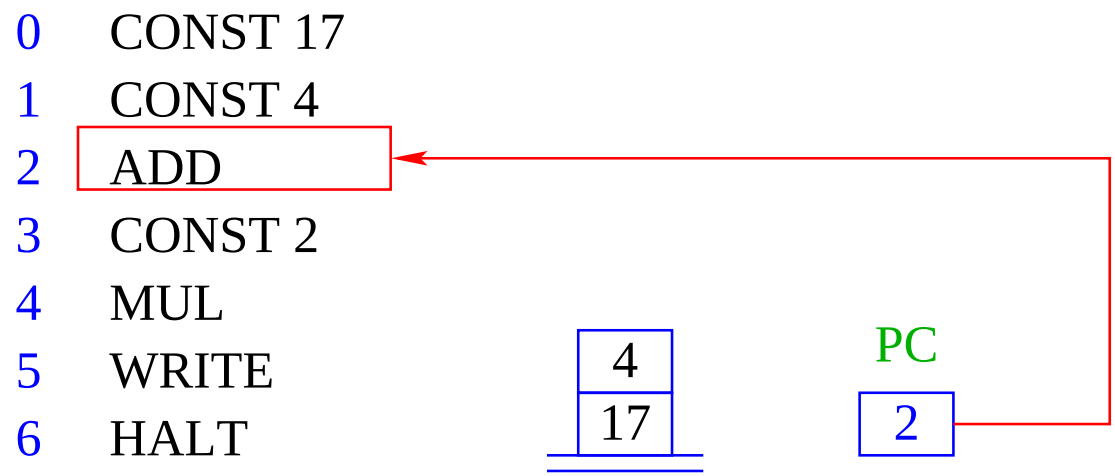
Ablauf



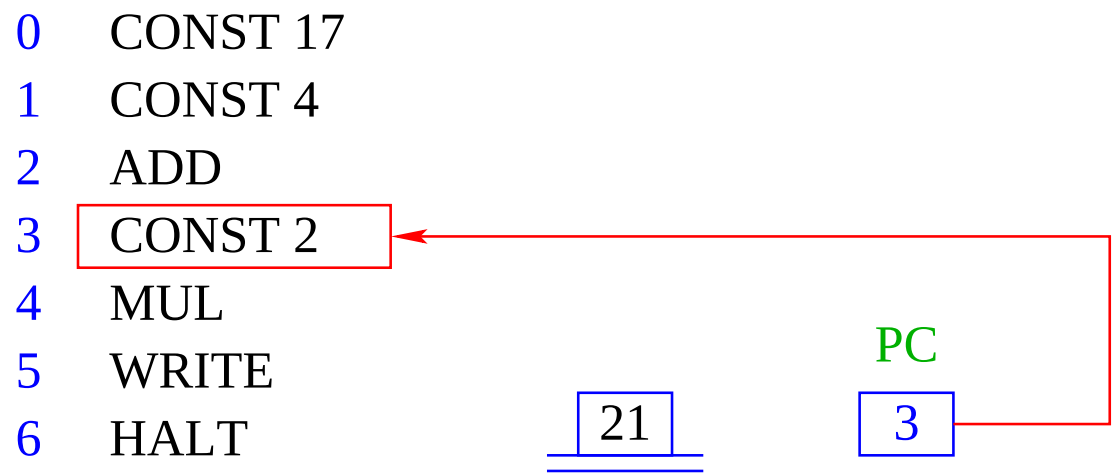
Ablauf



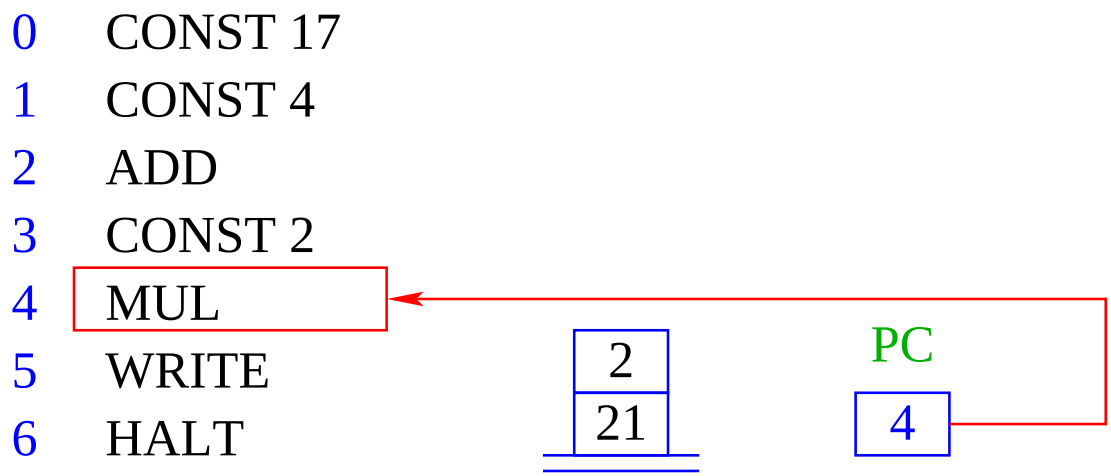
Ablauf



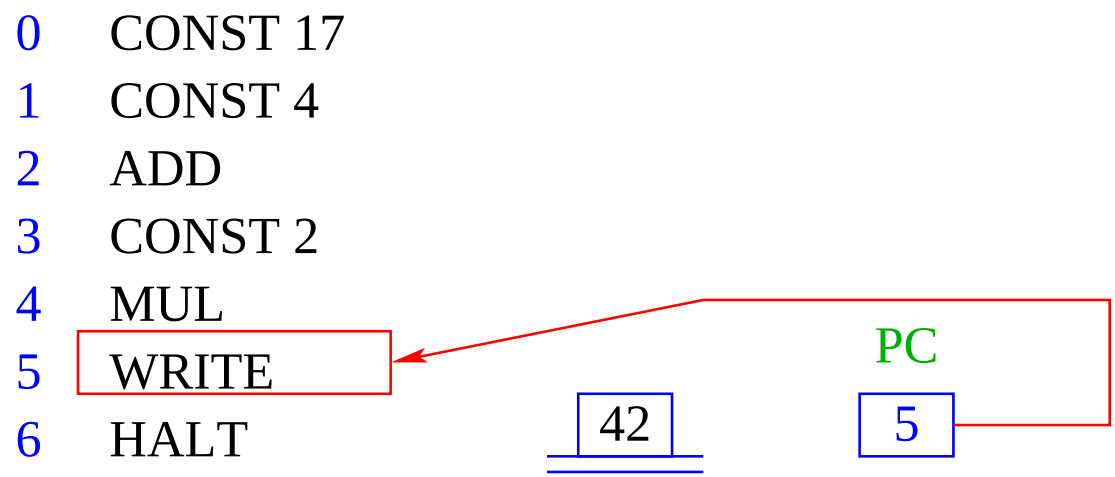
Ablauf



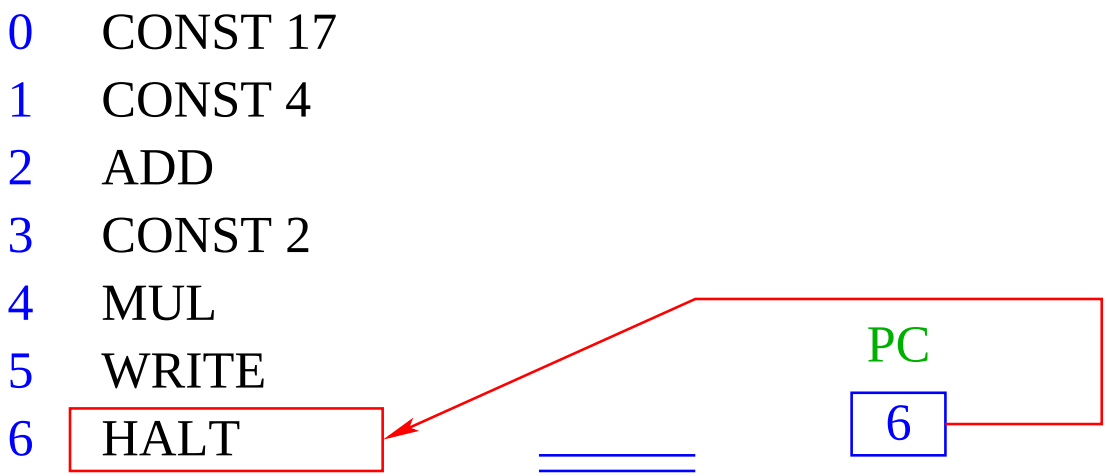
Ablauf



Ablauf



Ablauf



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.

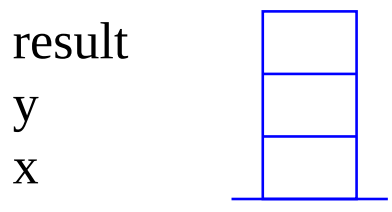
Deklarationen

Betrachte Deklaration

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

Idee:

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



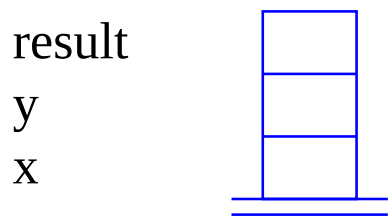
Deklarationen

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

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.

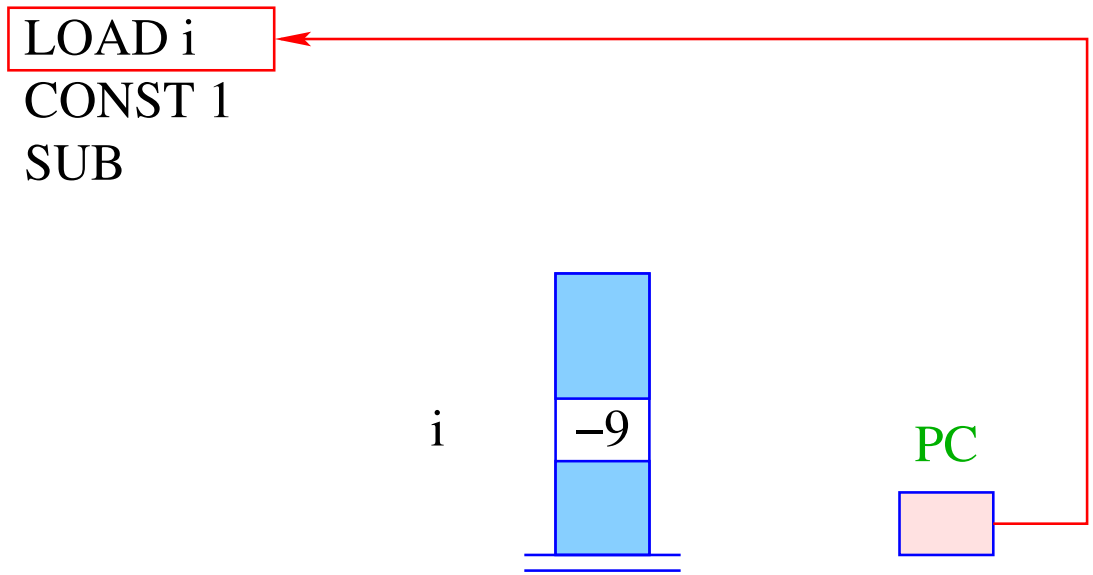
Übersetzung von Ausdrücken

Idee:

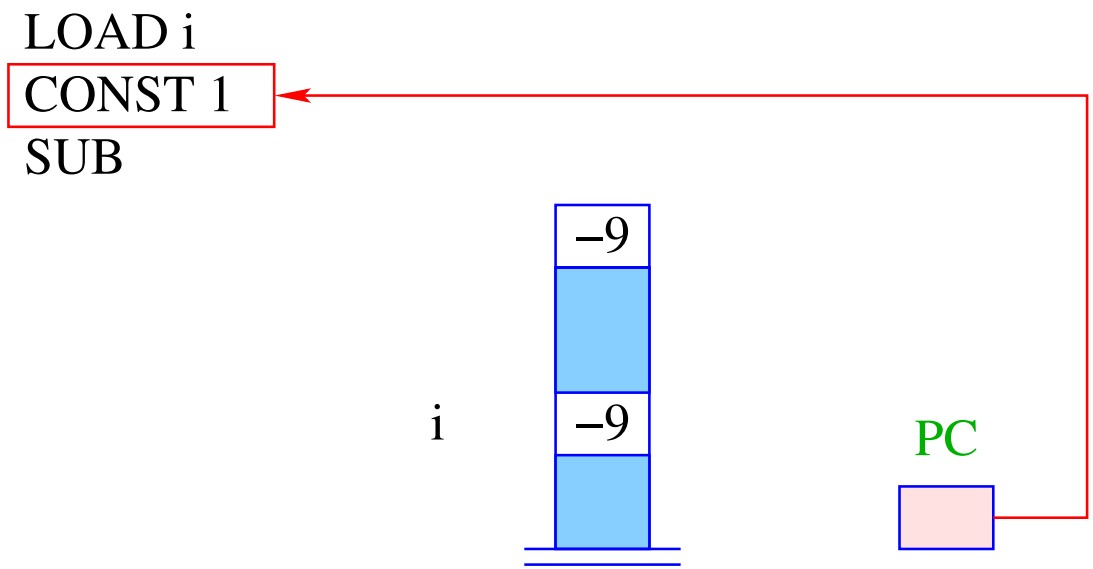
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

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

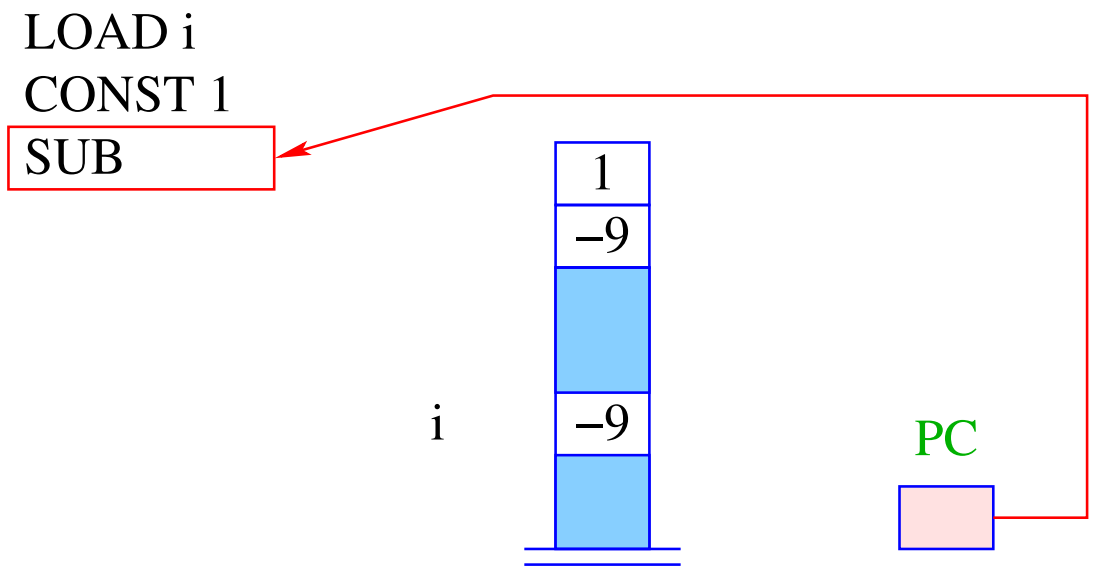
Ablauf



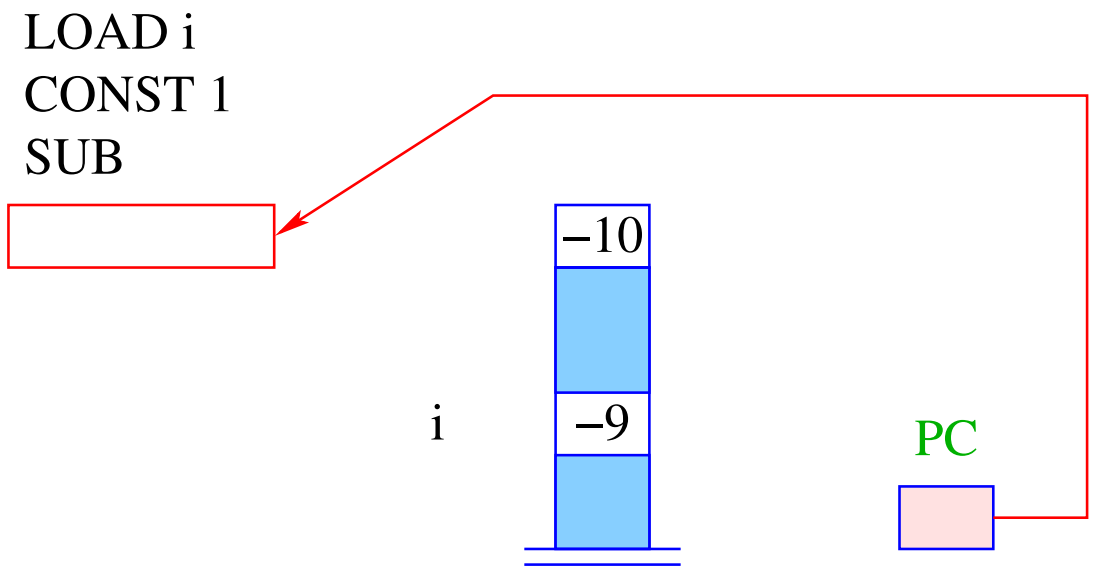
Ablauf



Ablauf



Ablauf



Allgemein

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

... analog für die anderen Operatoren ...

Beispiel

Sei expr der Ausdruck: `(x + 7) * (y - 14)`
wobei `x` und `y` die 0. bzw. 1. Variable sind.
Dann liefert die Übersetzung:

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

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!

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

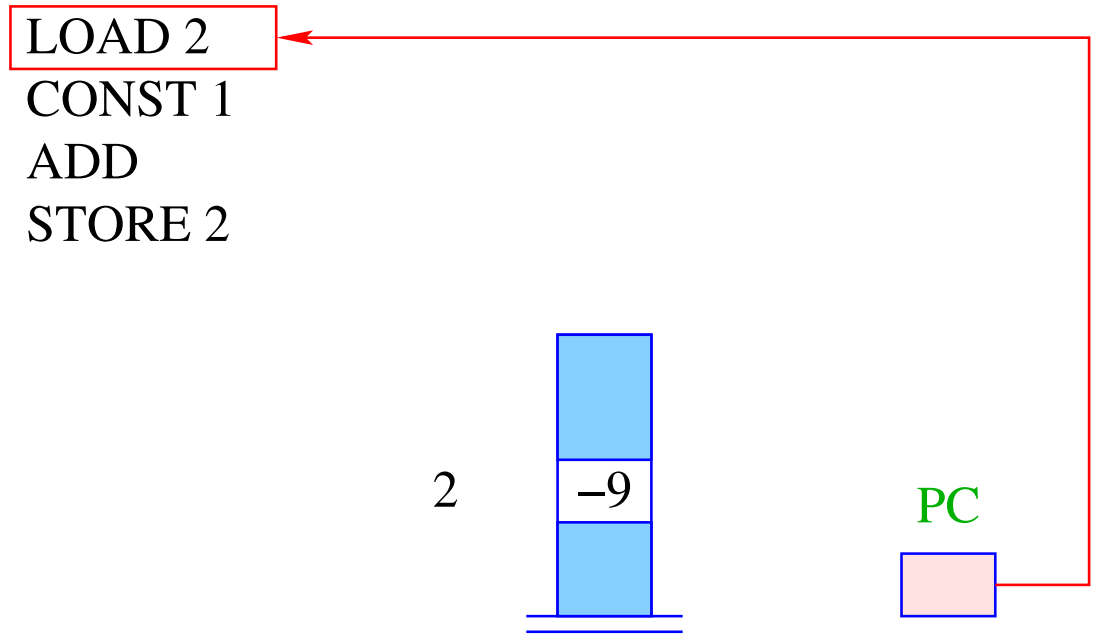
Übersetzung von	`x = expr;`	=	Übersetzung von `expr` STORE i

Beispiel

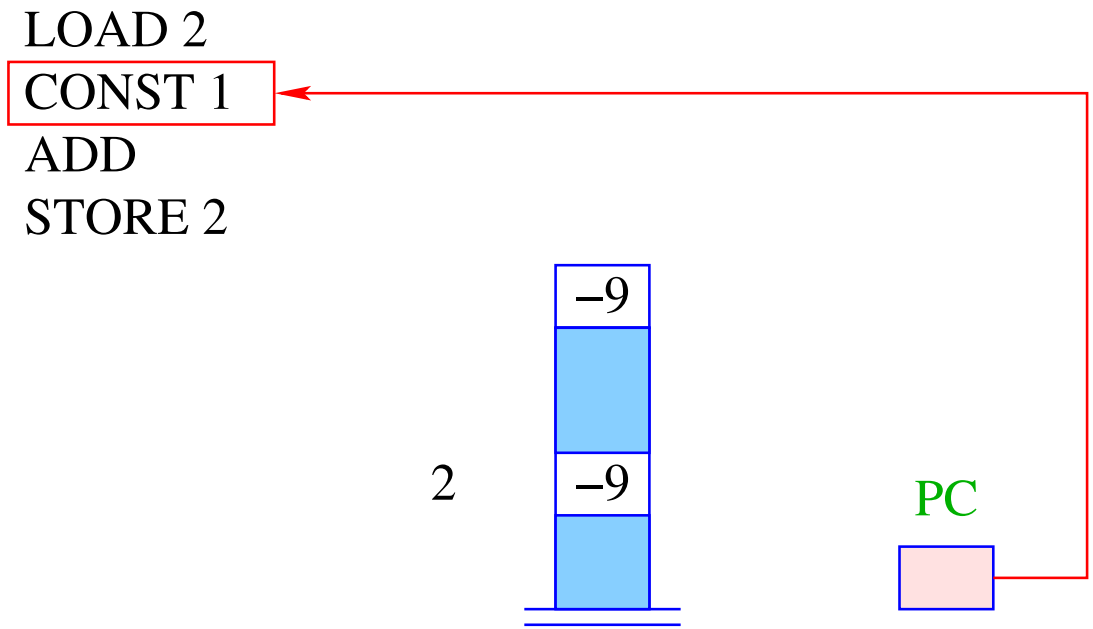
Für `x = x + 1;` (`x` die 2. Variable) liefert das:

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

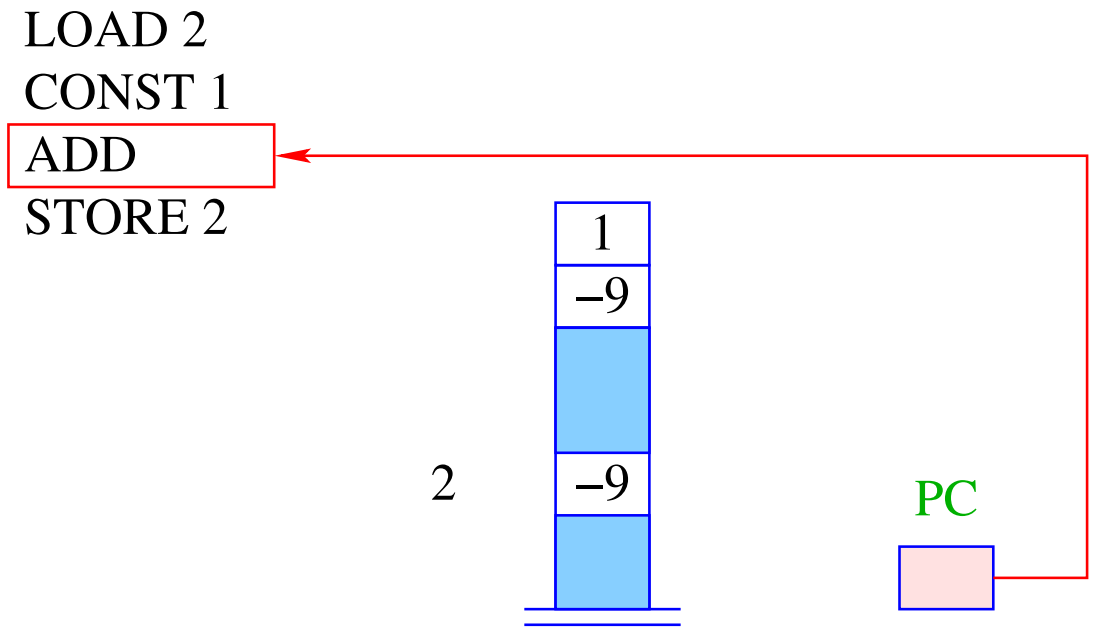
Ablauf



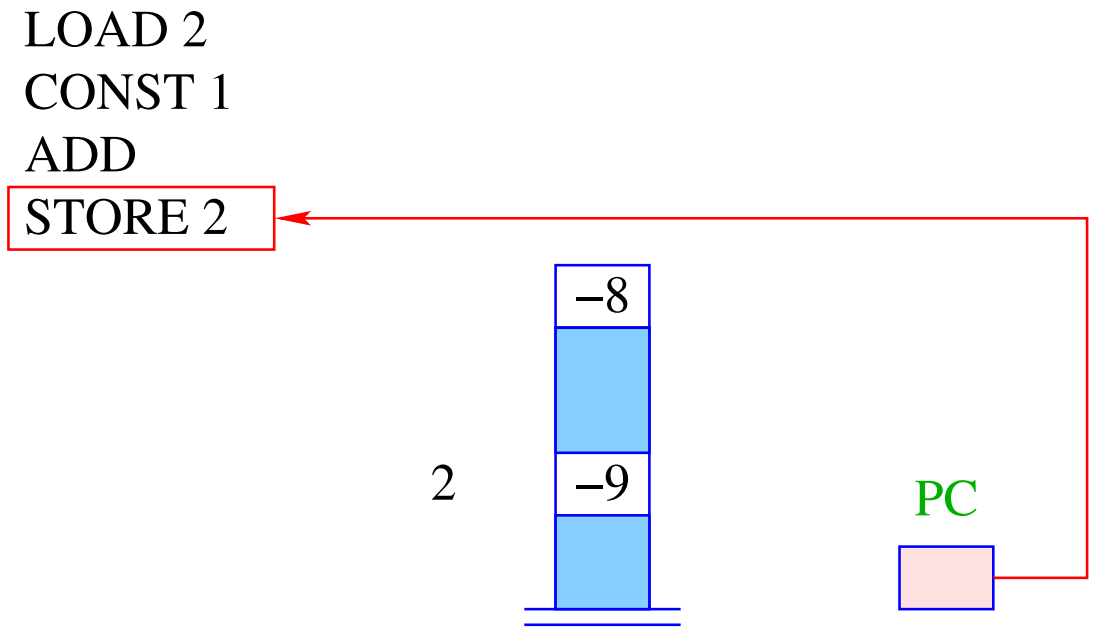
Ablauf



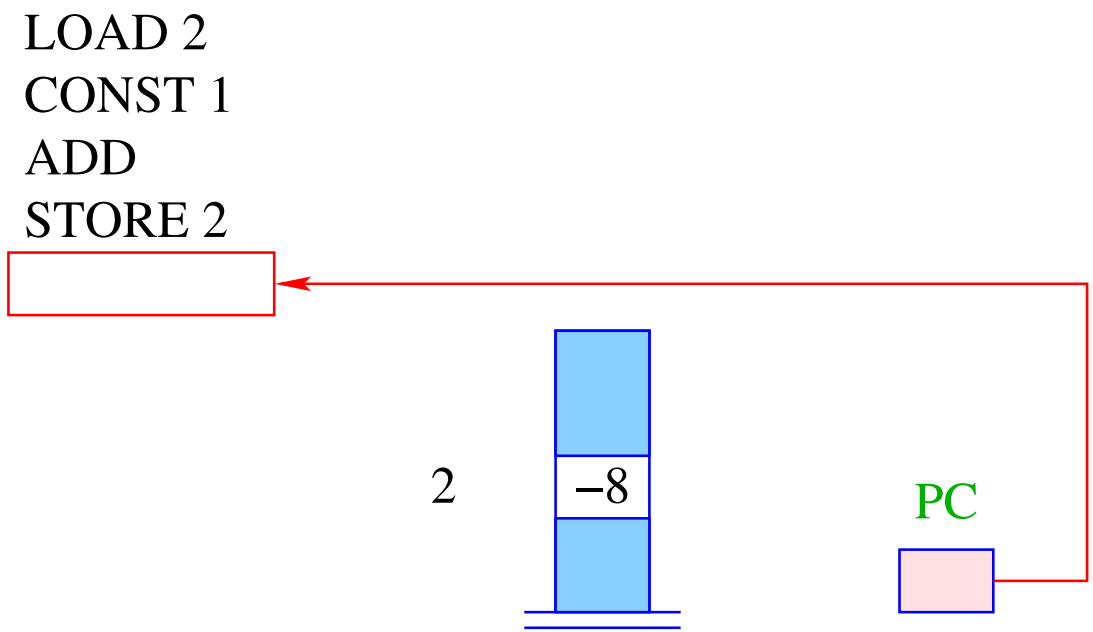
Ablauf



Ablauf



Ablauf



read und write

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

Übersetzung von if-Statements

Bezeichne stmt das `if`-Statement

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

Idee:

- Wir erzeugen erst einmal Befehlsfolgen für cond, stmt_1 und stmt_2.
- Diese ordnen wir hintereinander 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.

Übersetzung

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

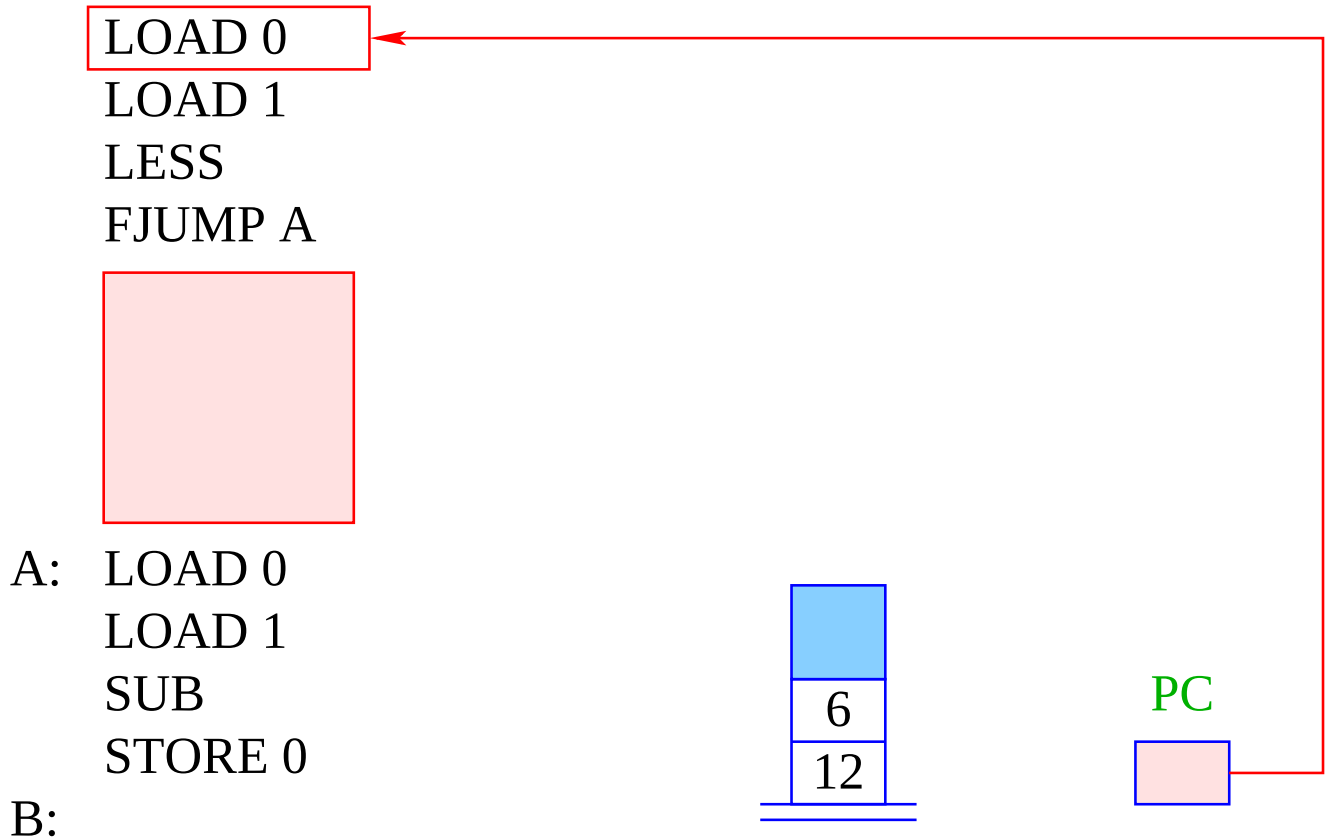
Für das Statement:

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

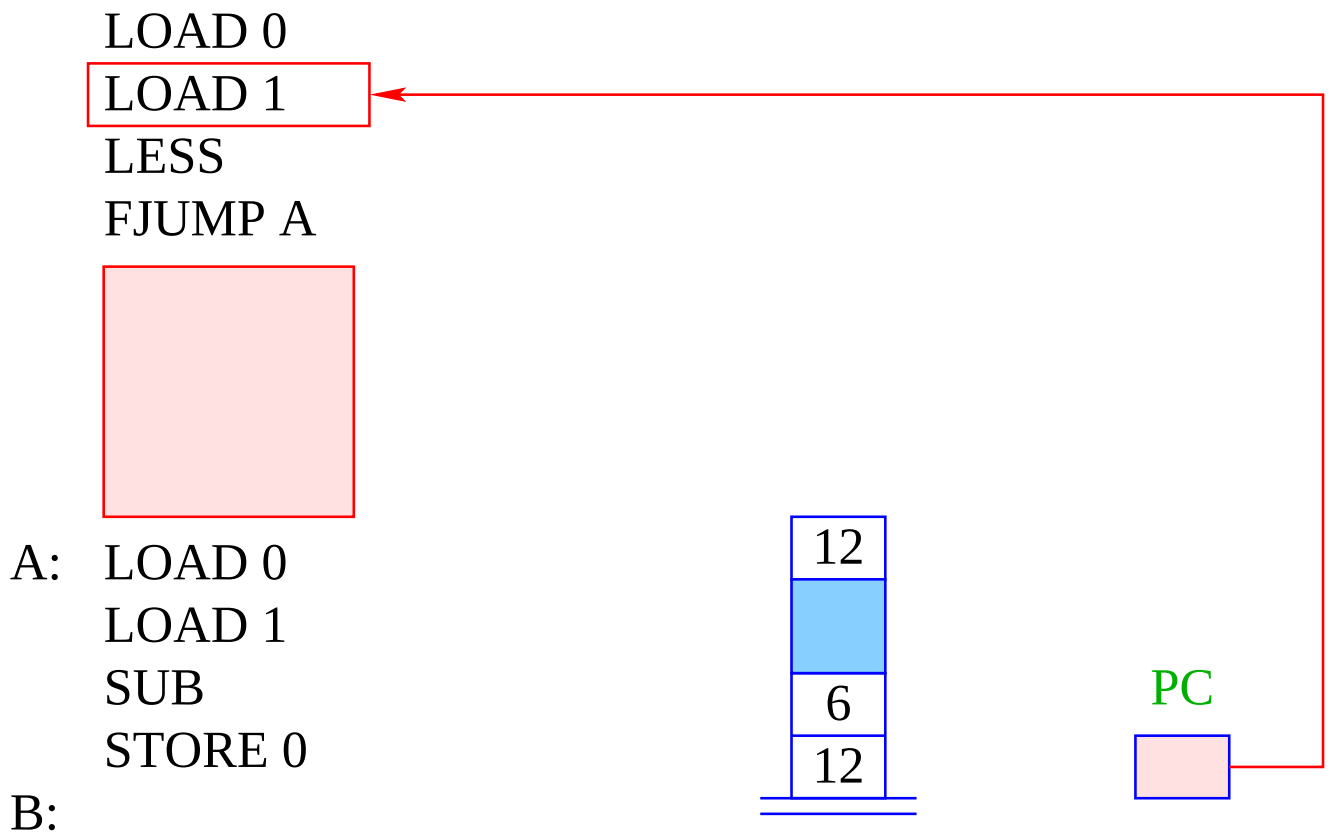
(`x` und `y` die 0. bzw. 1. Variable) ergibt das:

```
    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:  ...
```

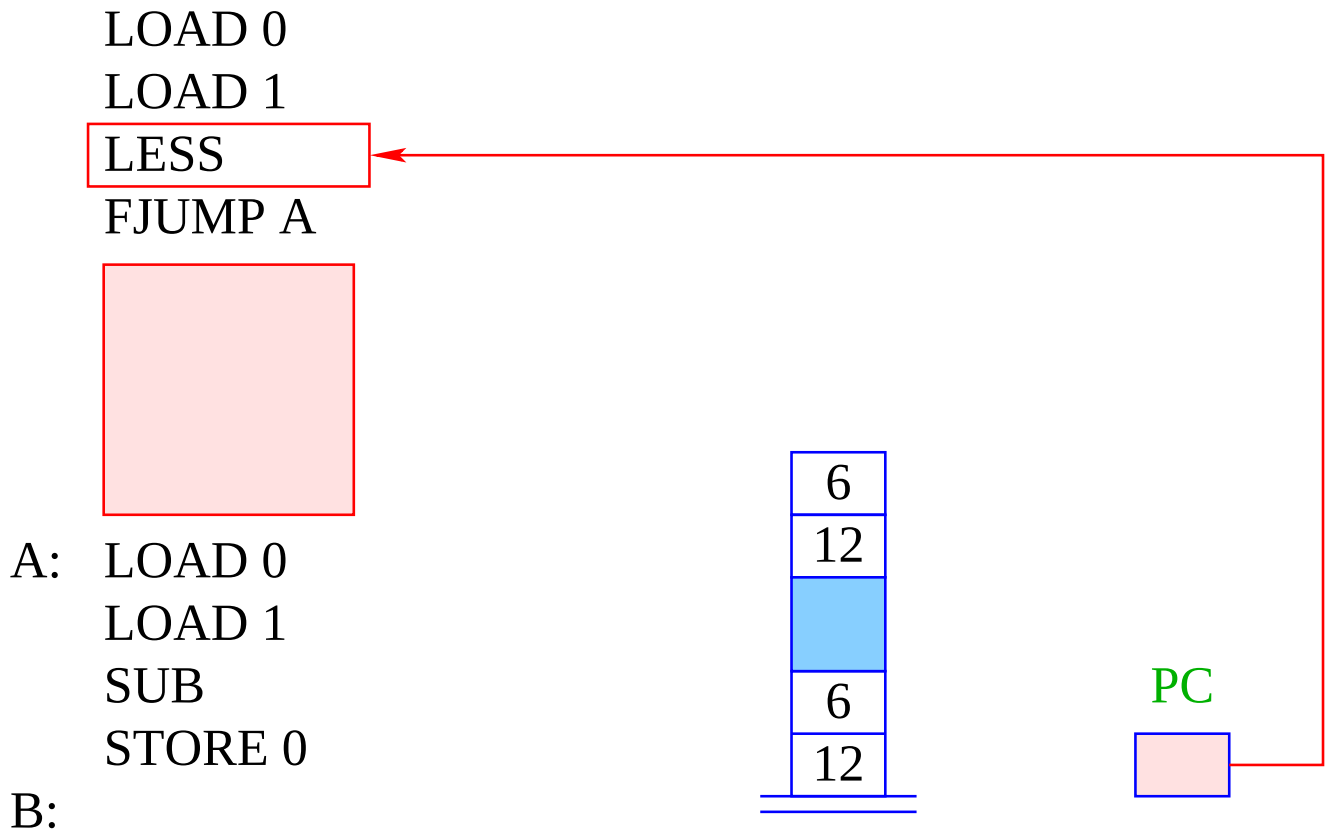
Ablauf



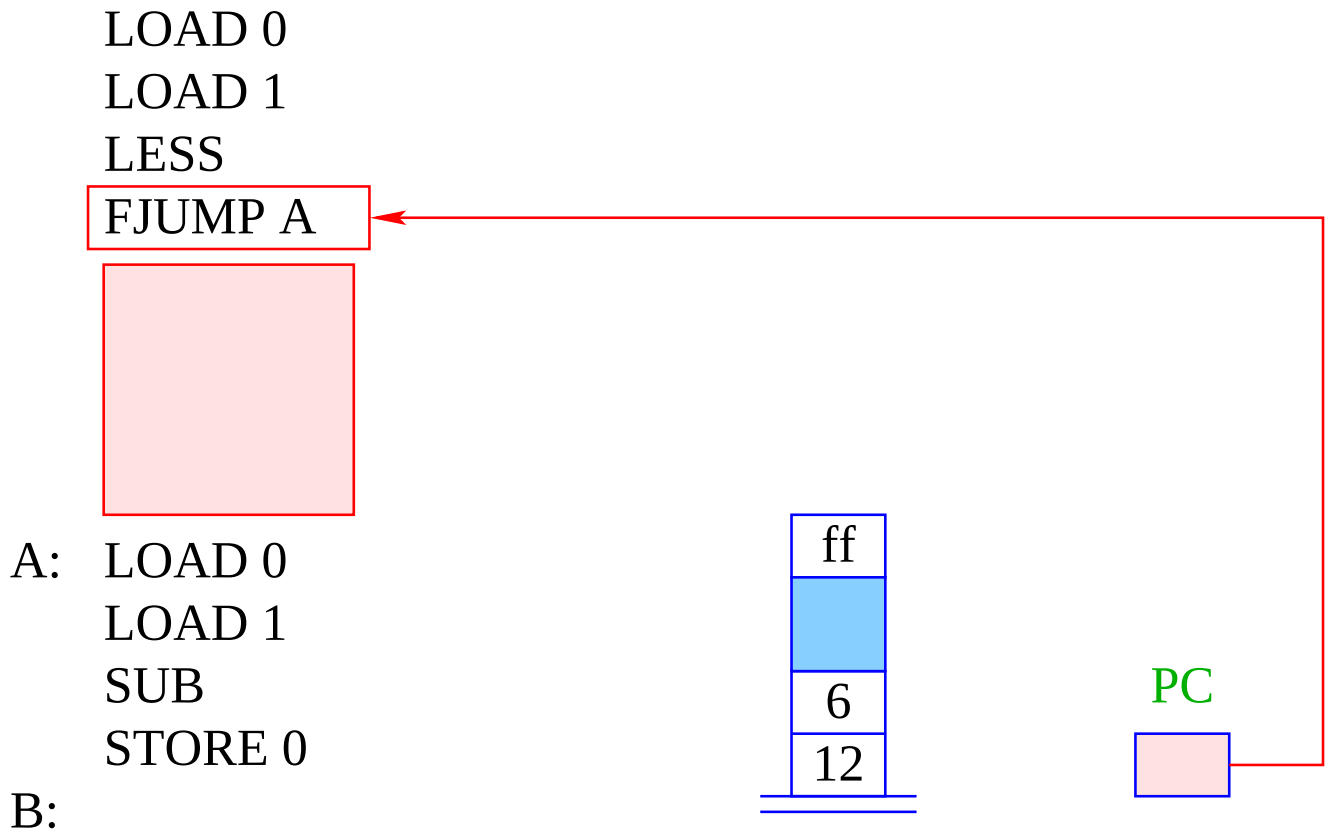
Ablauf



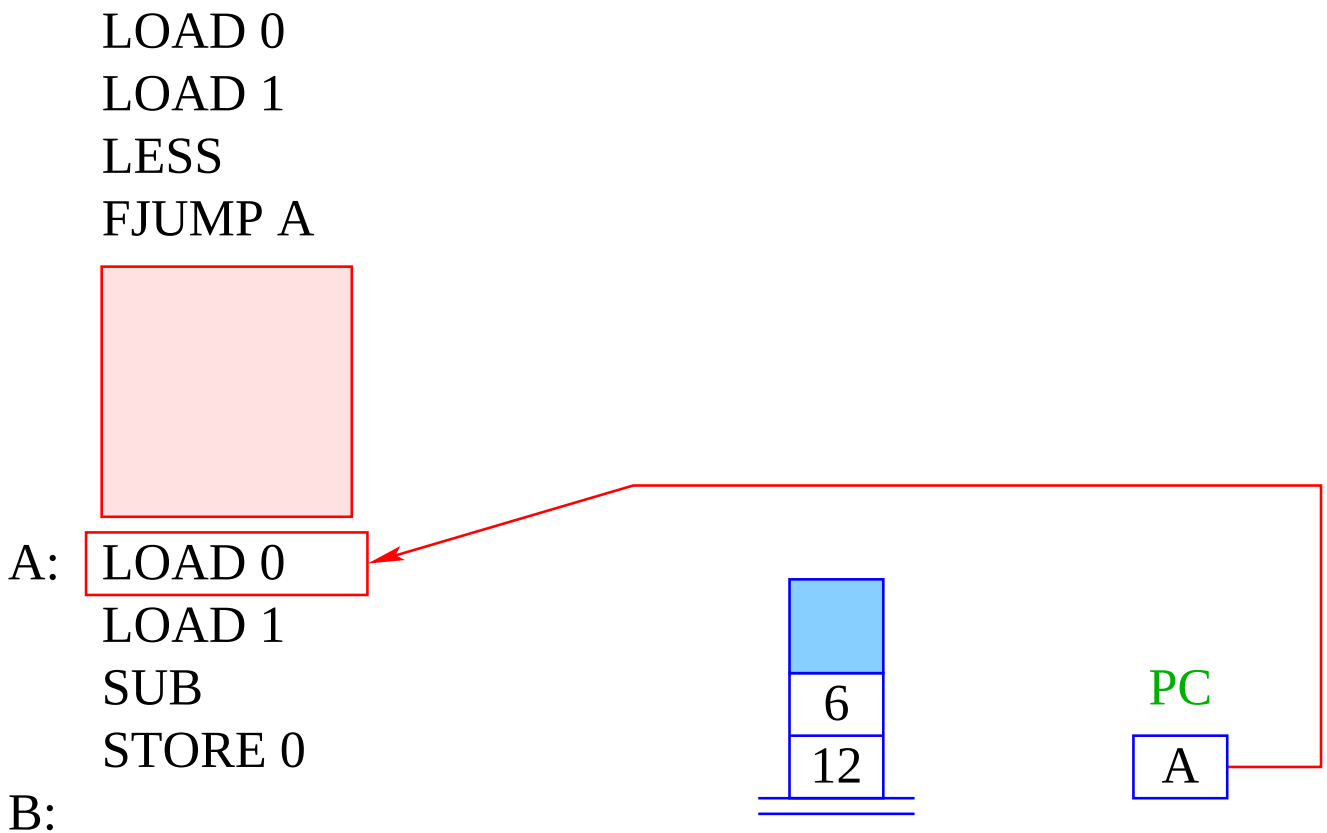
Ablauf



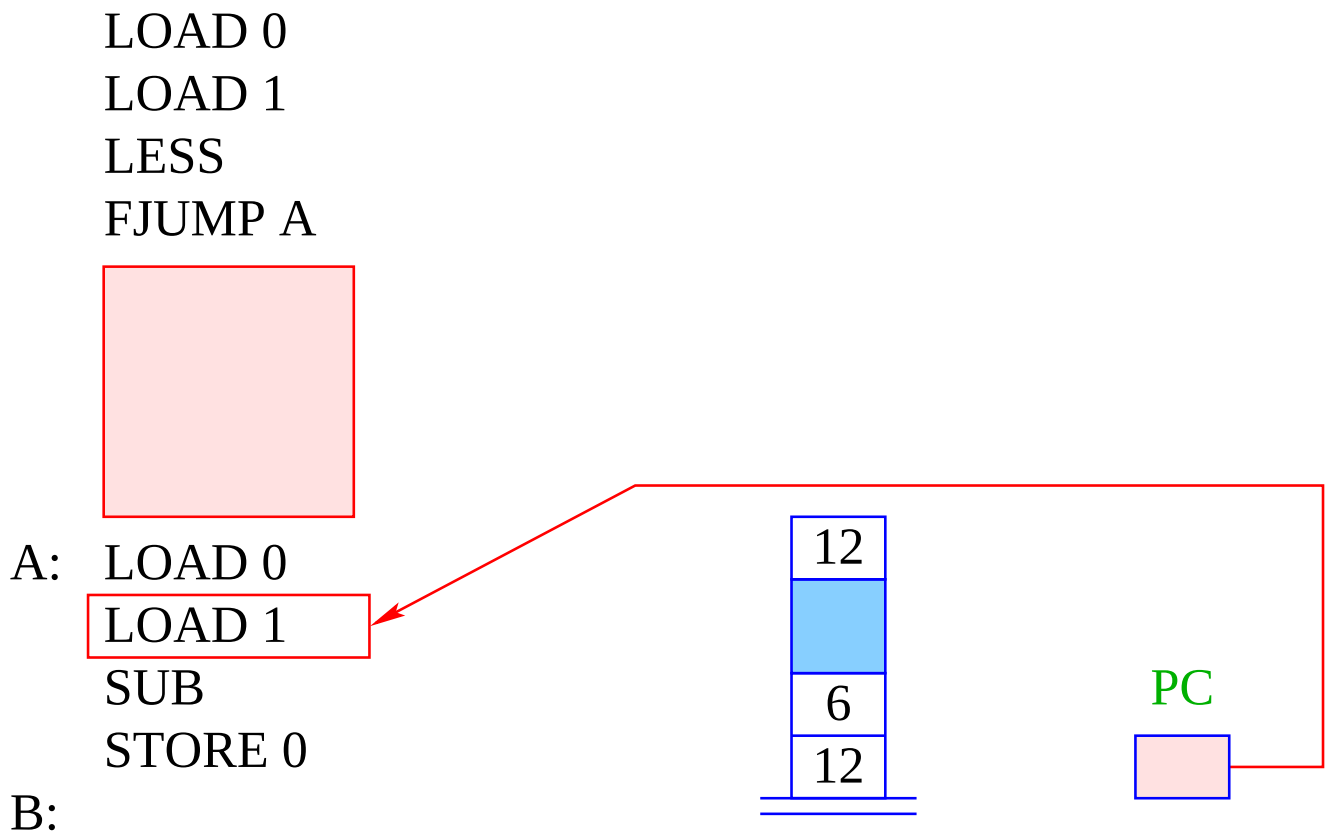
Ablauf



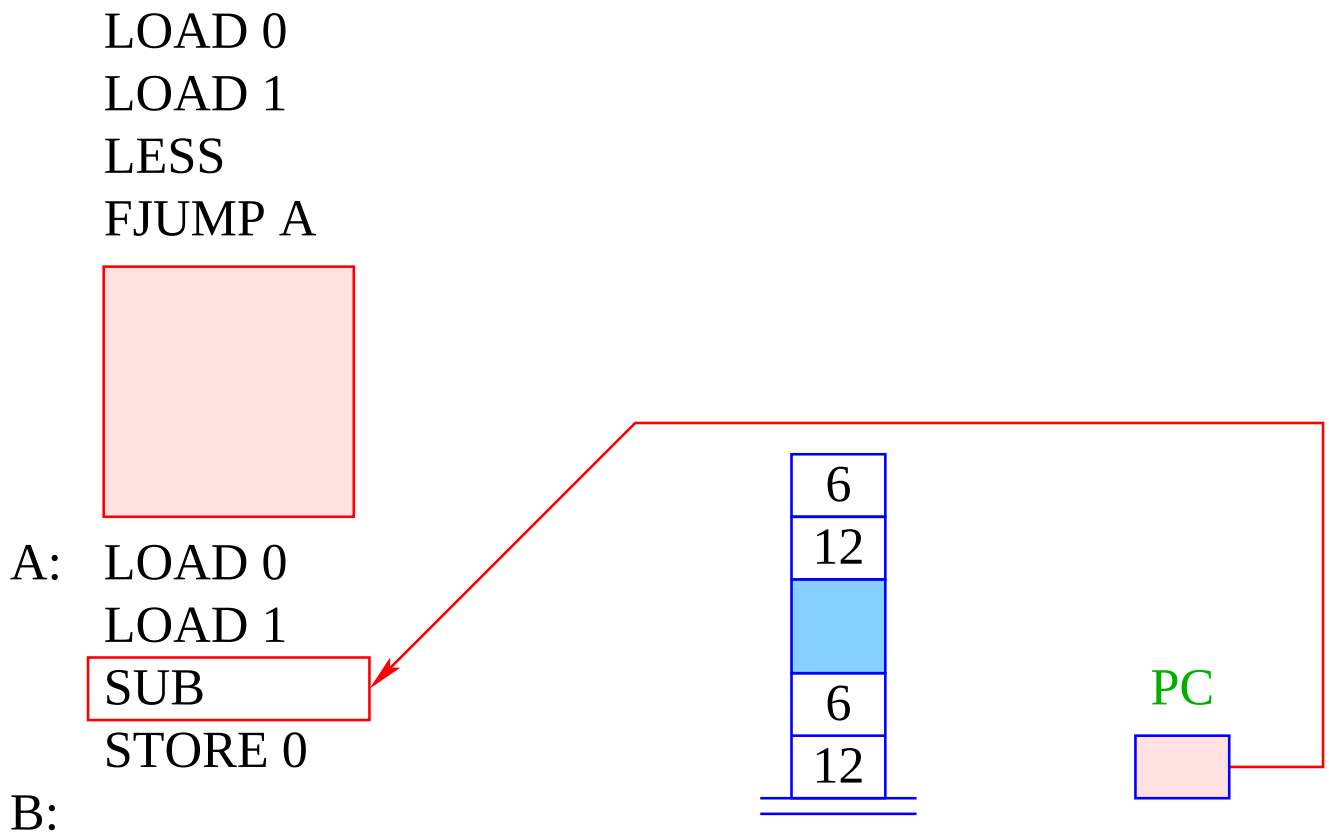
Ablauf



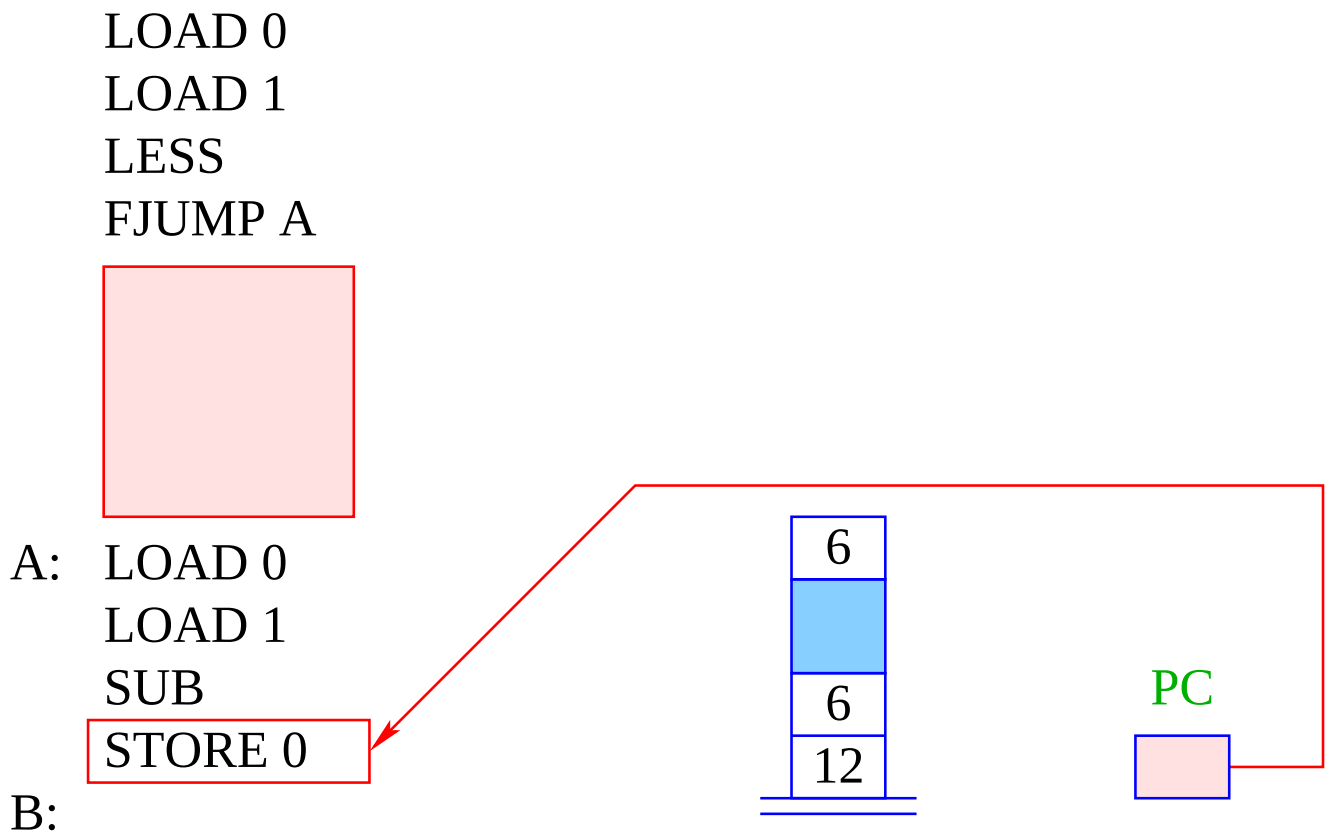
Ablauf



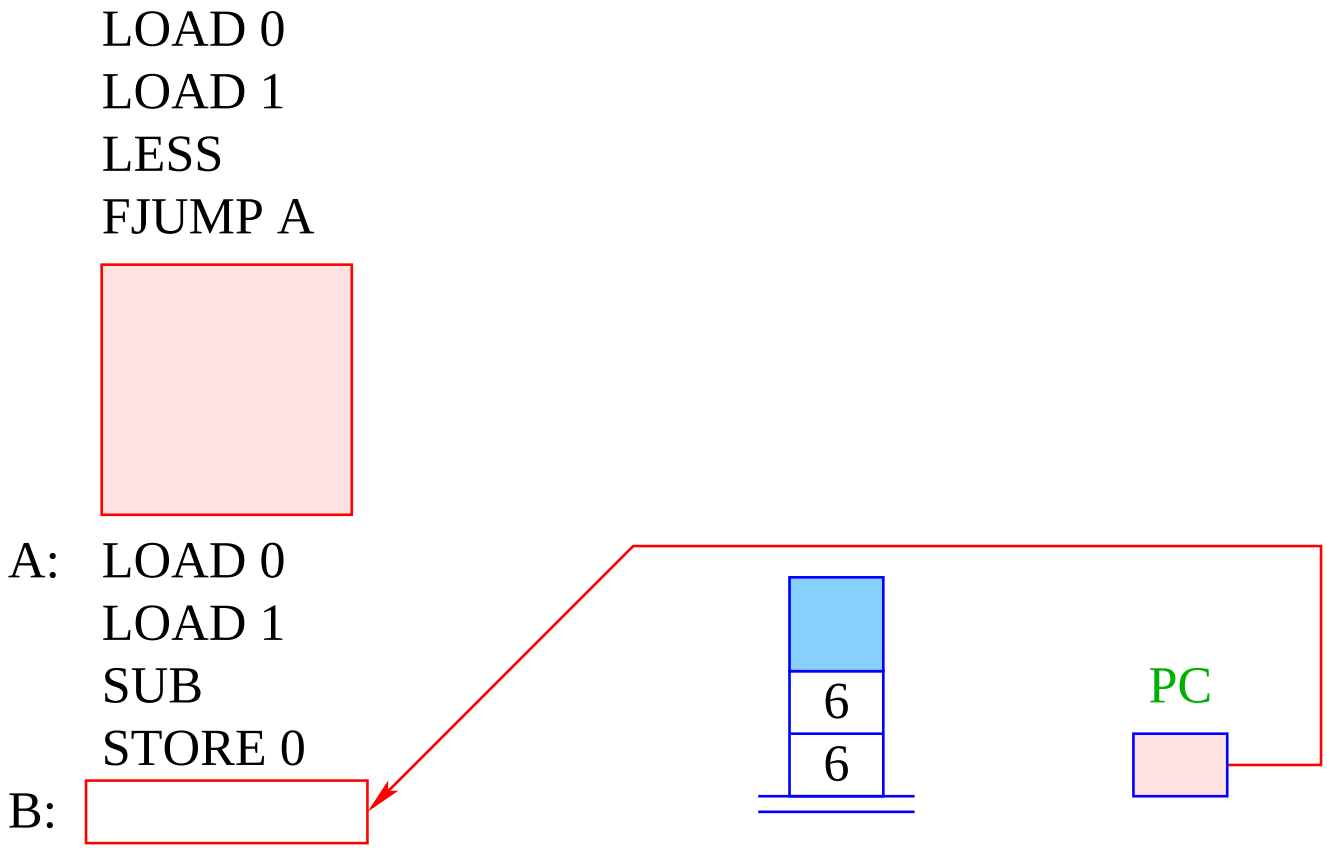
Ablauf



Ablauf



Ablauf



Übersetzung von while-Statements

Bezeichne stmt das `while`-Statement

$$\texttt{while (cond)}\ \mathrm{stmt}_1$$

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.

Übersetzung

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:   ...
```

Übersetzung von Statement-Folgen

Idee:

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

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

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.

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

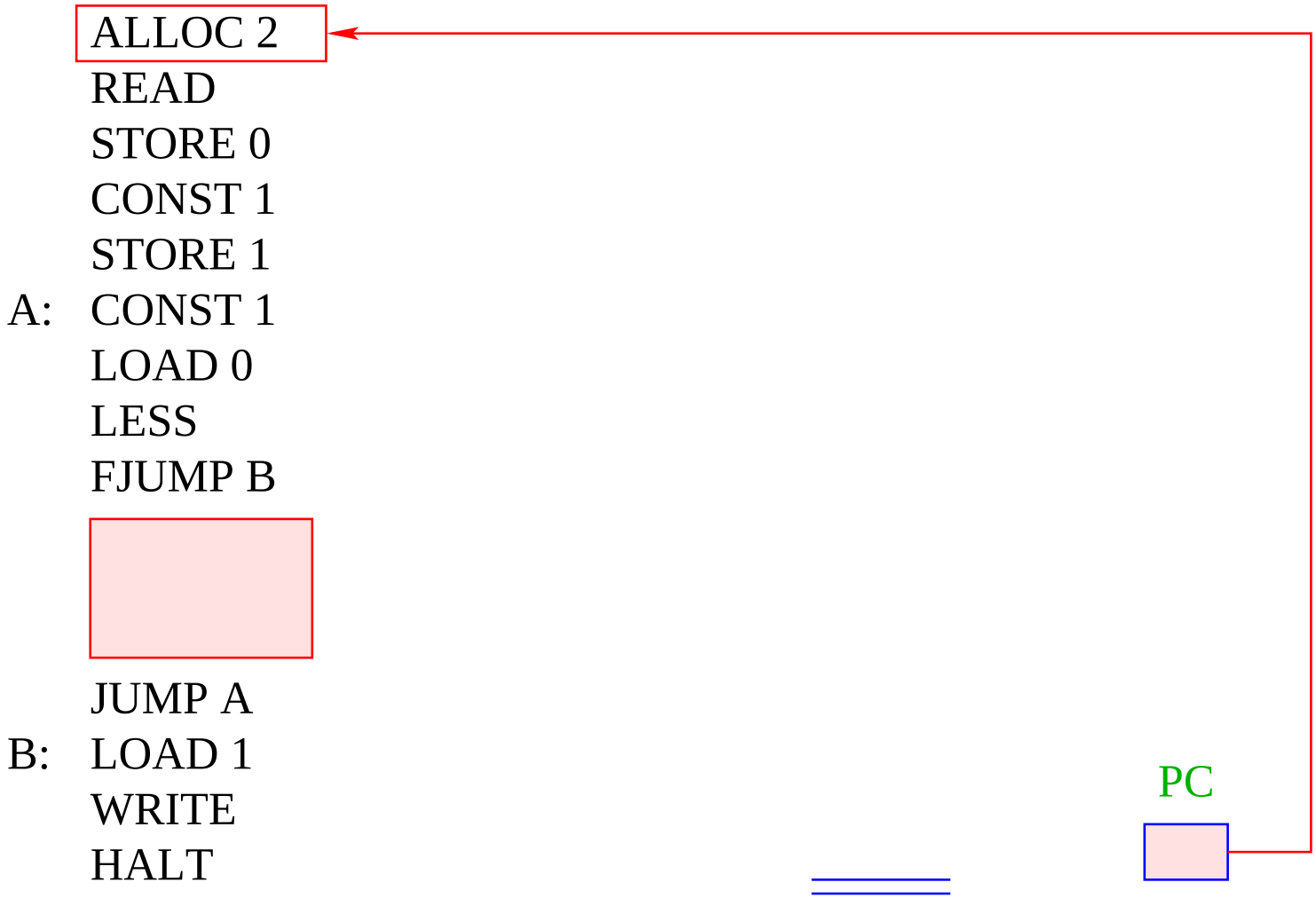
```
int x, y;
x = read();
y = 1;
while (1 < x) {
    y = y * x;
    x = x - 1;
}
write(y);
```

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

Übersetzung

```
    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
```

Ablauf


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

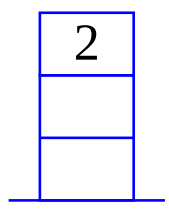
Ablauf

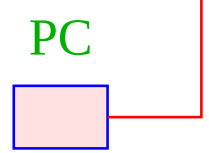
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

Ablauf

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

```
      JUMP A
B:    LOAD 1
      WRITE
      HALT
```





Ablauf

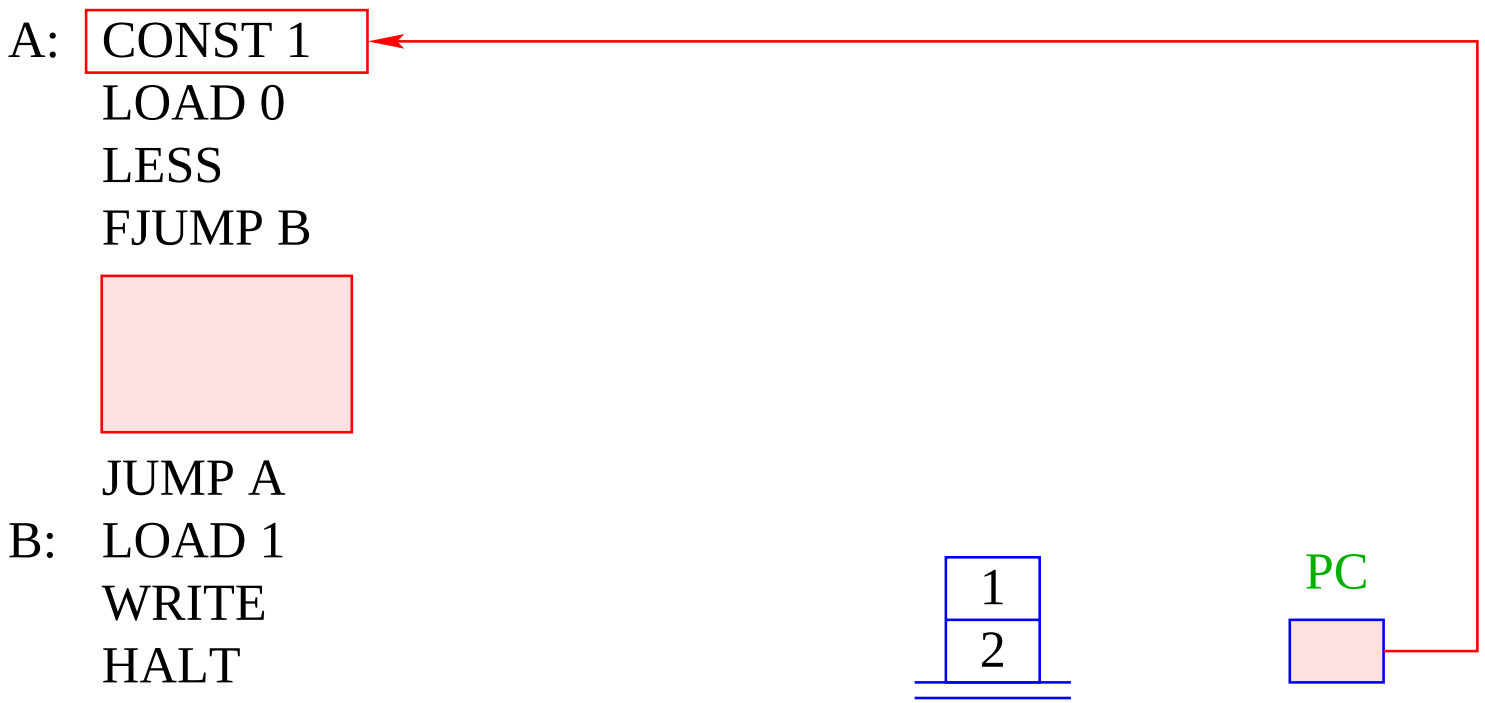
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

Ablauf

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

        JUMP A
B:      LOAD 1
        WRITE
        HALT
```

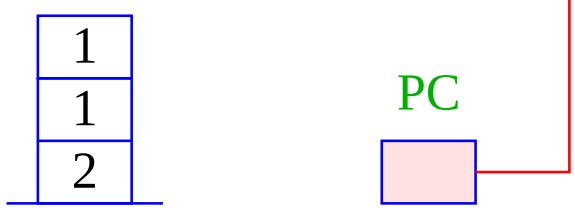
Ablauf

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

        JUMP A
B:      LOAD 1
        WRITE
        HALT
```



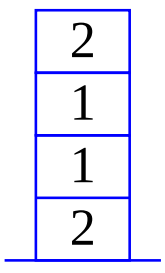
Ablauf

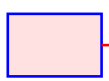
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```



Ablauf

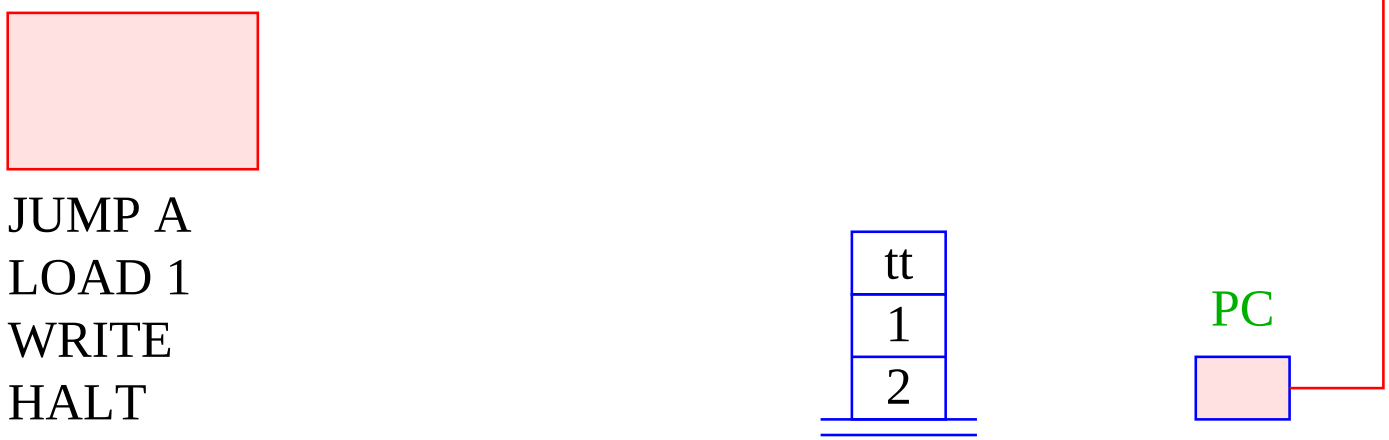
```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B

     JUMP A
B:   LOAD 1
     WRITE
     HALT
```





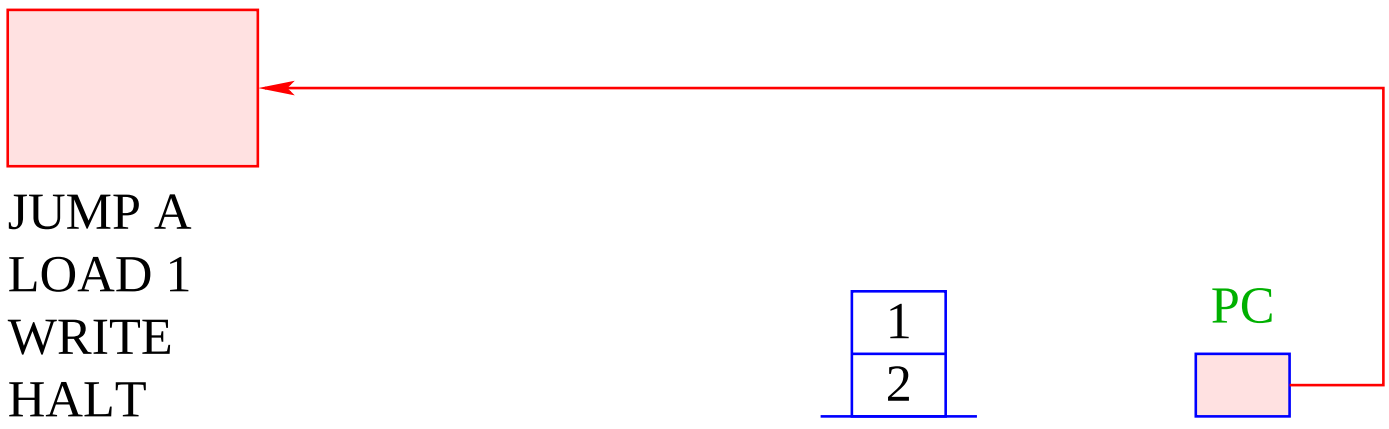
Ablauf

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



Ablauf

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

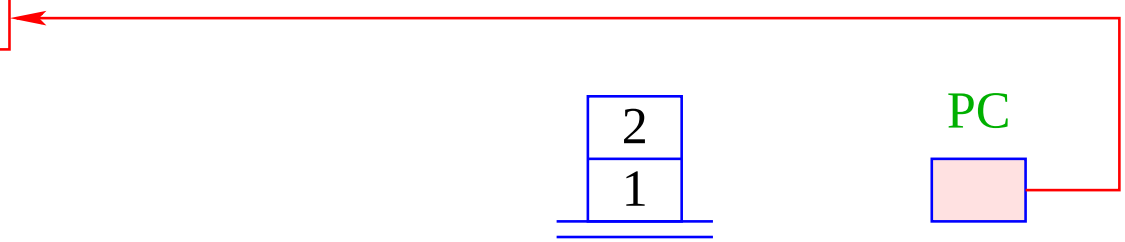



```
    JUMP A
B:  LOAD 1
    WRITE
    HALT
```

Ablauf

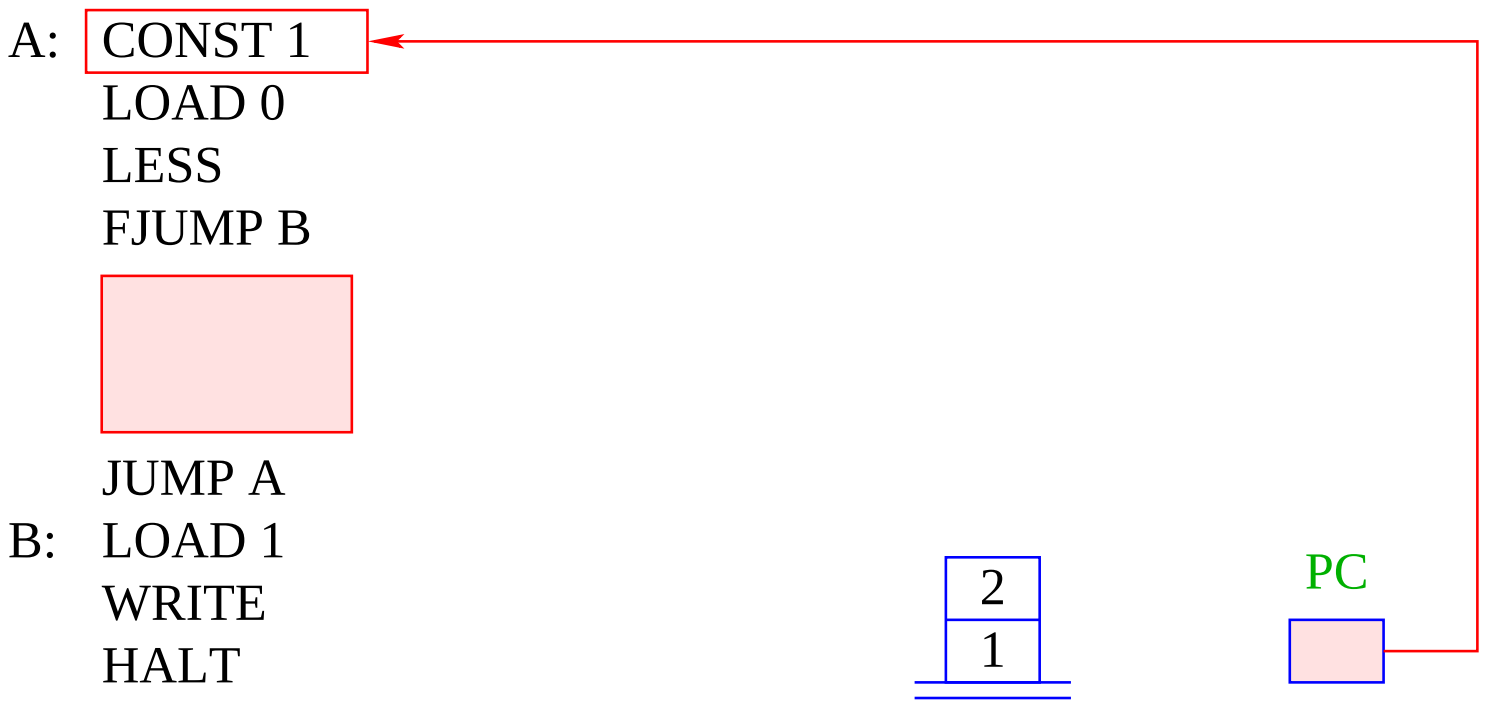
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B
```

```
    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



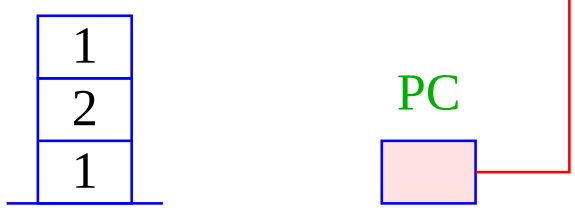
Ablauf

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

        JUMP A
B:      LOAD 1
        WRITE
        HALT
```



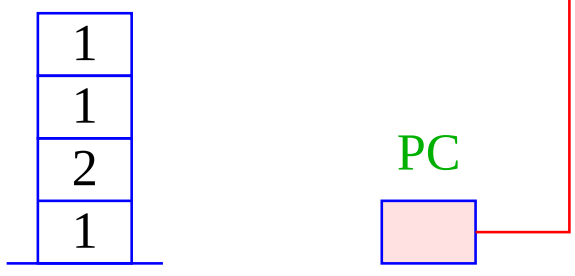
Ablauf

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

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



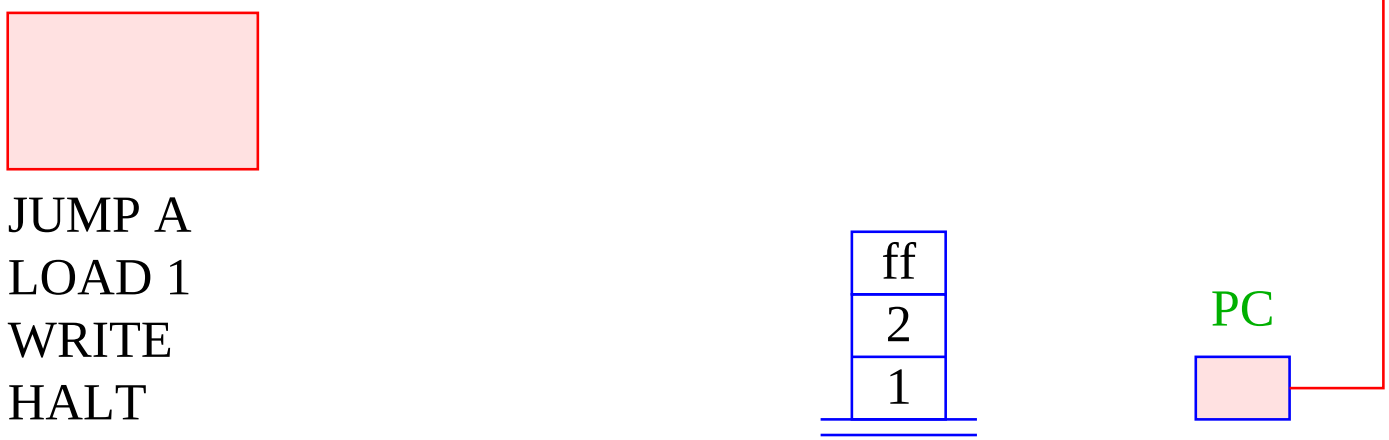
Ablauf

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

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



Ablauf

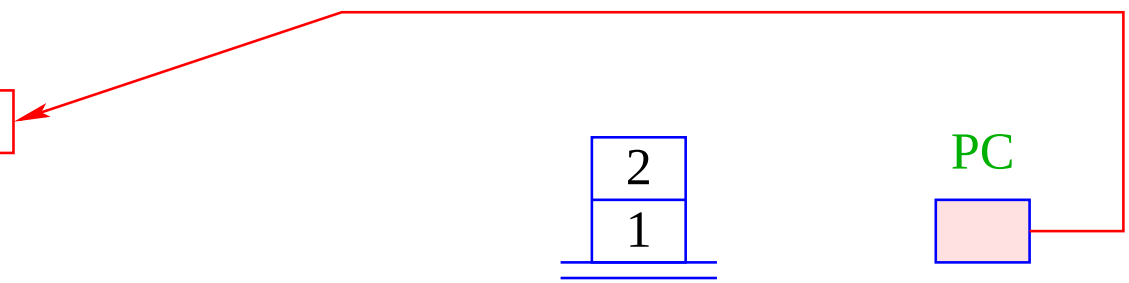
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B
      JUMP A
B:    LOAD 1
      WRITE
      HALT
```



Ablauf

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

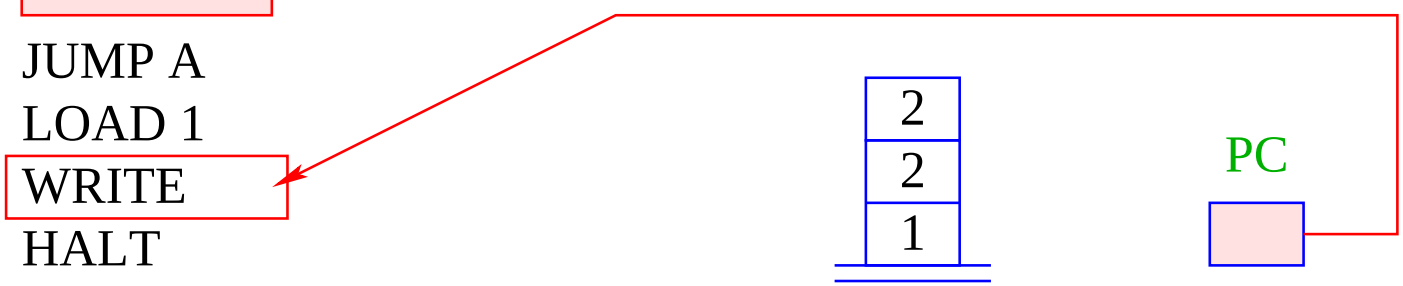
```
    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



Ablauf

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

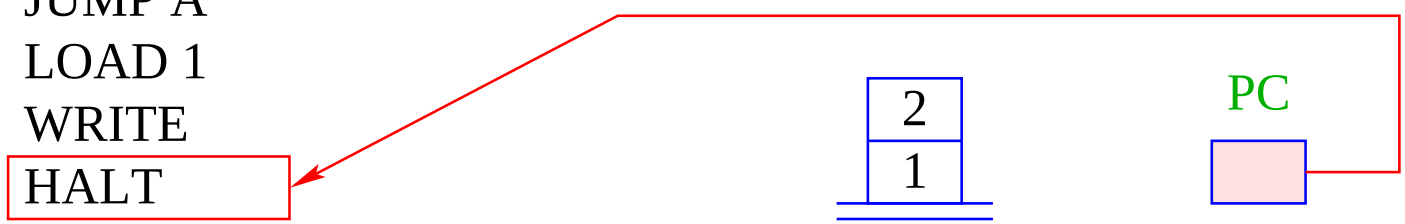
```
    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



Ablauf

```
    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