7. Mehrprogramme, Funktionen, Methoden

Kapitel 7.1 aus Ratz, Scheffler, Seese von S. 163 – 173.

Das Programm Methtest.java zum Ausprobieren.

Wir haben unsere Programme bis jetzt die Struktur:

```
public class <Programmname> {
public static int <Methodenn><Param.> {
}
public static boolean[] <Methodenn><Param> {
}        // Hier evtl Methodendeklarationen
---------
public static class <Klassenname> {
      public int x
}
public static class <Klassenname> {
      public <Klassenname> ref
}        // Hier evtl Klassendeklarationen.
---------
public static void main(String[] args) {
}        // Hauptprogramm.
```

Zum Überladen von Methodennamen:

Es ist möglich mehrere Methoden gleichen Namens zu deklarieren:

```
public static float plus(int a, int b)
{
return (float)(a+b)
}

public static float plus(float a, float b)
{... //irgendetwas
}

float c;
int d;

plus(c,d) // Die zweite Definition
          // wird genommen.
```

Implizite Typanpassungen, sofern zulässig werden gemacht. Es wird die Definition genommen, für

die die wenigsten Typumwandlungen erforderlich sind. Im Zweifelsfall Compilerfehler:

```
public static void f(int a, long b)
{ ... }

public static int f(long c, long d)

f(1,2)  // Erste Definition, da
        // eine implizite Typumwandlung
        // 2. long. Die zweite Defi-
        // nition braucht 2 Umwandlungen.

f(1L,2L) // Die zweite Definition, klar.

// Weitere Deklaration
public static void f(long a, int b)
{ }

f(1,2)  // Fehlermeldung vom Compiler
        "ambiguous".
```

Zuerst die Methodendeklaration wird gewünscht, bei der die wenigsten Typangaben erforderlich sind.
Fehlermeldung vom Compiler bei Mehrdeutigkeit.

Das folgende ist von grundsätzlicher Wichtigkeit zum Verständnis von Methoden und deren Aufrufen. Was geschieht bei der Ausführung eines solchen Methodenaufrufs ?

Das Programm beginnt das Hauptprogramm mit

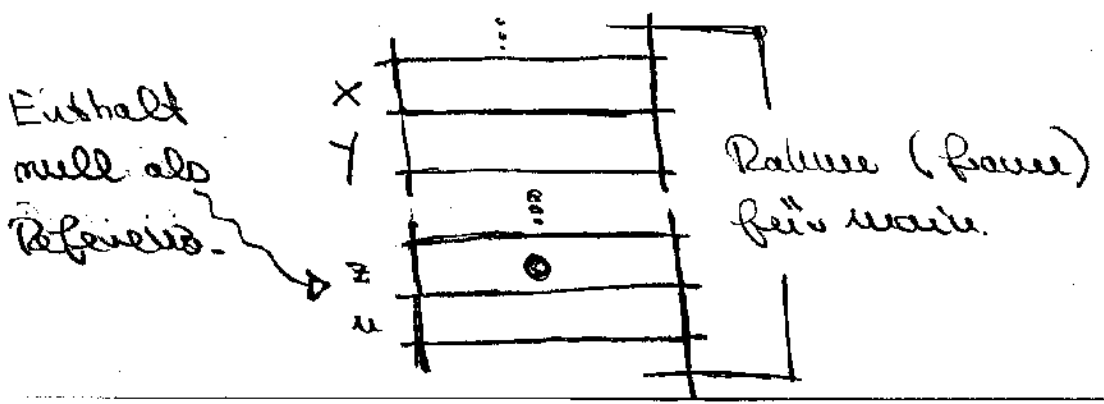
main(String[] args)

auszuführen. Die lokalen Variablen darf sind durch den Compiler bekannt.

Es wird im Hauptspeicher ein Bereich (Rahmen, Frame) angelegt, wo diese Variablen zu finden sind. Also sowas etwa

```
public class <Klassenname>
{

public static void main(String[] args){
int x;
float y;
int[] z;
double u;
}
}
```

Sei der Rahmen etwa



Kommt es jetzt zu einem Aufruf einer Methode, etwa

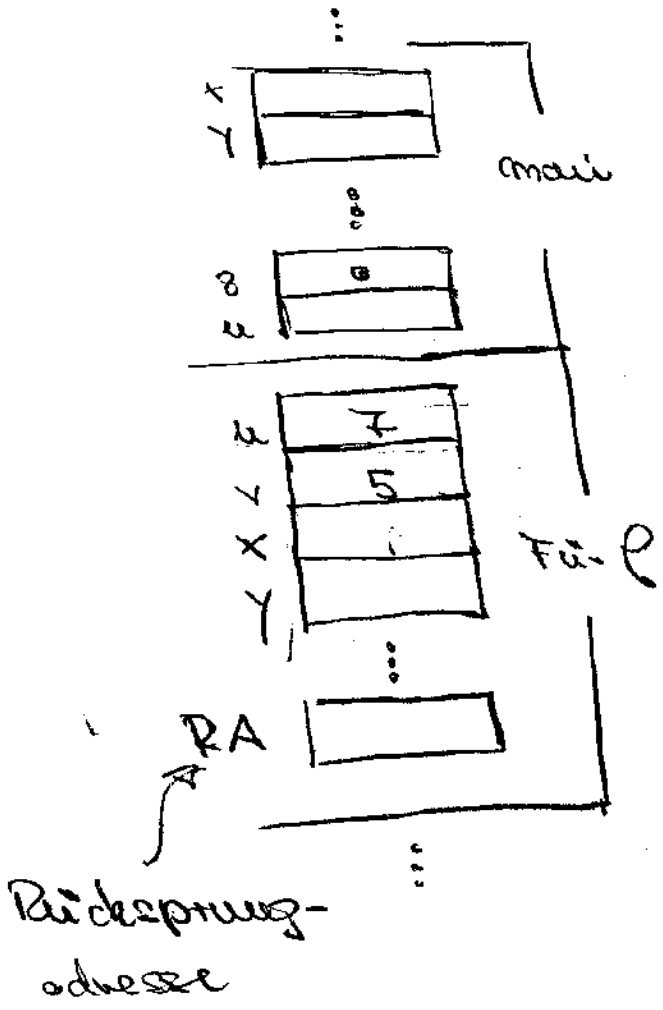
f(x, 5),

wobei x den Wert 7 hat und die Deklaration von f

```
public static int f(int u, int v)
{
  int x
  int y
}
```

den Aufruf in Frage liegt, so wird ein neuer Rahmen vor den von main gesetzt.

Unser Hauptspeicher sieht jetzt so aus:



Das Programm zu f wird in dem Rahmen von f ausgeführt.

Beim Aufruf p(x, 5) muß also folgendes geschehen:

- Ein neuer Rahmen wird angelegt. D.h. es werden Register gesetzt, in denen sich der Prozessor merkt, wo der aktuelle Rahmen liegt, ...
- In den Rahmen wird Speicherplatz für die formalen Variablen der Methode und die lokalen Variablen der Methode plaziert.
- Es werden die Werte der aktuellen Parameter (hier x = 7 und 5) ausgerechnet und bei den formalen Parametern (hier u und v) gespeichert.

Das ist im Prinzip (!) alles - fast alles.

Irgendwann ist die Auffüllung von L am Ende. Wenn geschieht dann ? Man muß im mainz Inhalt der Stelle des Aufrufs von f(x,5) weitermachen. Jetzt kann es mehrere Aufrufe von f geben. Deshalb merkt man sich im Rahmen, wo am von f(x,a) Ende weitergemacht werden soll: die

Rücksprungadresse (RA).

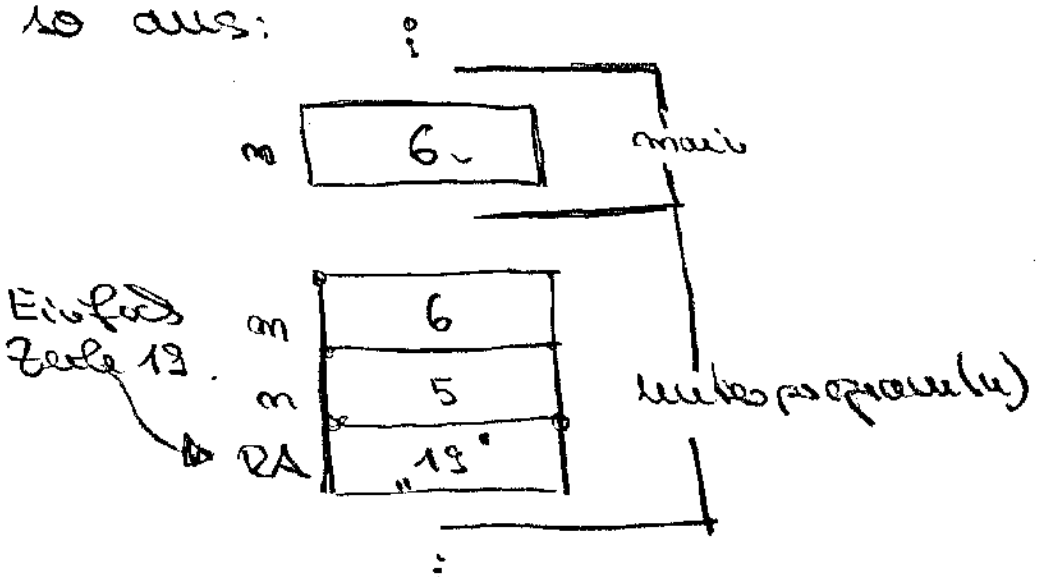
Wenn f(x,a) zu Ende geschieht folgendes:

- Eine eventuelle Ausgabe von P(x,5) wird übergeben (d.h. in den Rahmen von main eingetragen).
- Der Prozessor merkt sich RA in einem Register.
- Der Rahmen von P wird

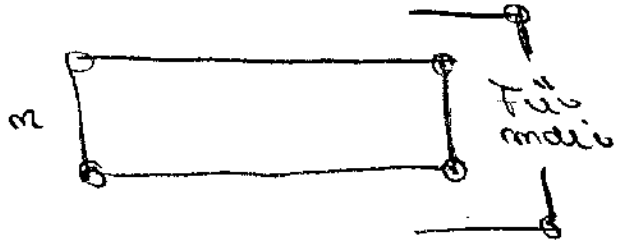
Bei dem Programm Aufruf Test.java auf S. 169 steht die Hauptspeicher nach Ausführung des Beginns des Aufrufs

unterprogramm(n)

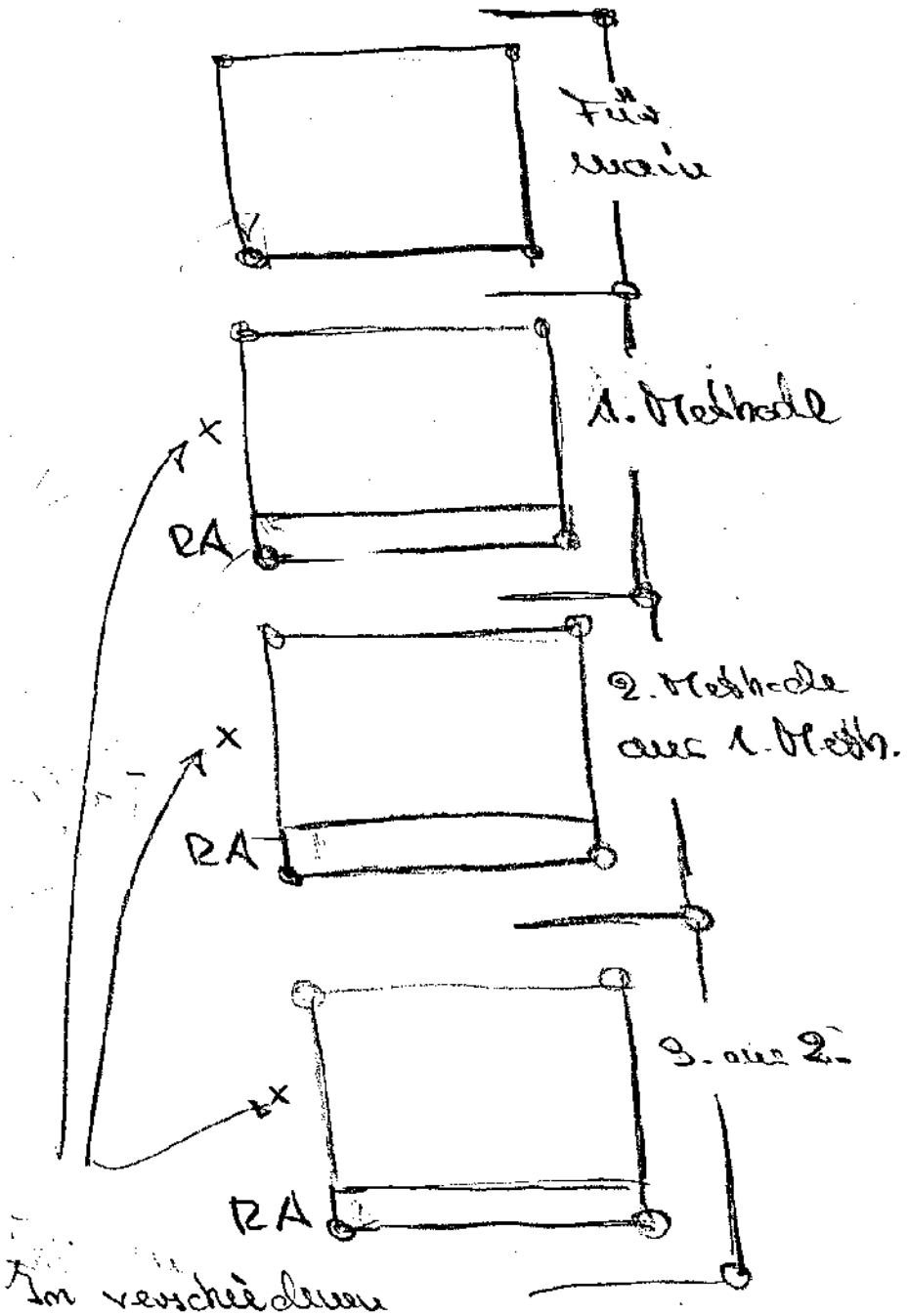
so aus:



Am Ende von mehrprogramm(4) bezieht sich wieder alles auf den aktuellen Rahmen.



Hätten wir innerhalb von mehrprogramm(n) weitere Methodenaufrufe; so würden weitere Rahmen generiert, etwa nach folgendem Prinzip:



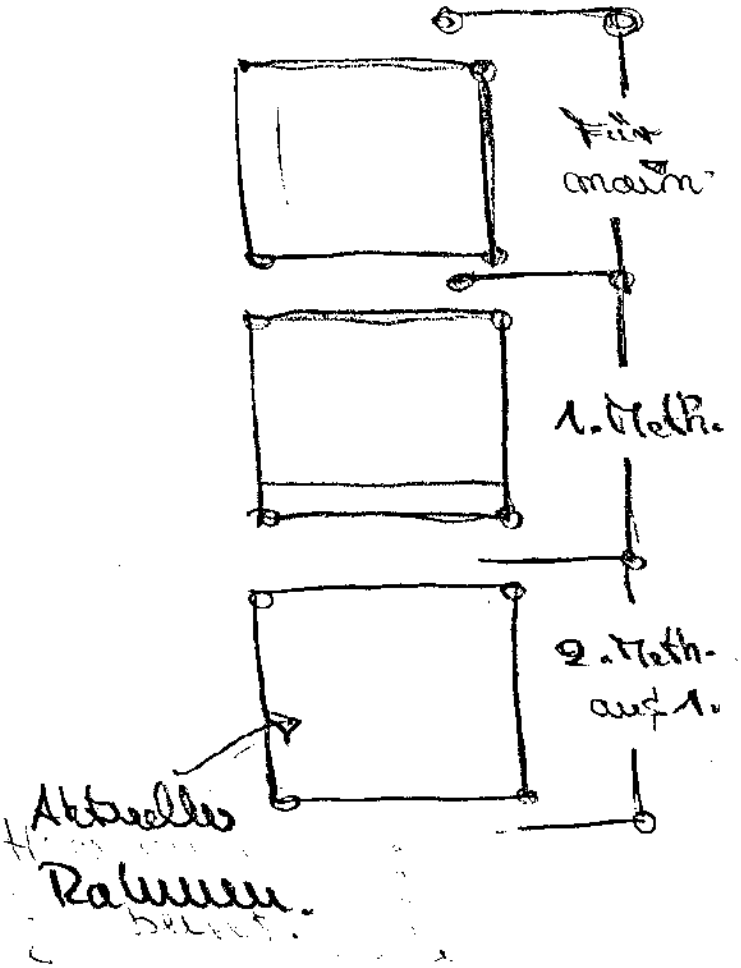
In verschiedenen

Räumen können

Variablen gleichen Namens

vorkommen!

Nach dem Ende von 3



Die Rahmen prüfen mit
den Aufrufen. Das Programm
arbeitet im aktuellen Rahmen,
dem vordersten (obersten, unterstem
– je nachdem) Rahmen.

Die Rahmen sind nach dem

- Kellerprinzip, Stapelprinzip, pushdown oder stack Prinzip

organisiert. Das heißt:

- Aktuelles Rahmen
  = Zuletzt erzeugter Rahmen
  ↑
  Neuster Rahmen.

Das ganze ist der

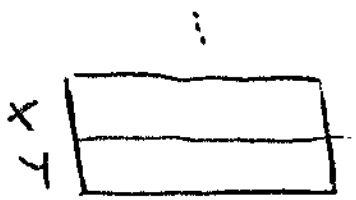
- Laufzeitstapel, run time stack, Prozedurkeller, ...

Da die Regeln zu gewöhnlichen
und in Methoden lokal deklarierten
Variablen behandelt, liegt sich
in dem vorliegenden Kontext auch
die Behandlung lokaler Variablen
in Blocks ein:

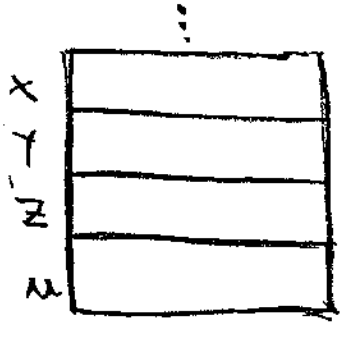
```
    ⋮

{ int x
  int y
    ⋮
  { int z      // Beachte: Hier noch
    int x      // einmal int x über-
    { ⋮        // setzungsfähig im
    { int z    // Unterschied zu Methoden!
      int u
      ⋮
    }
```

Man stellt sich am besten den aktuellen Rahmen vor, der dann in sich passiert:

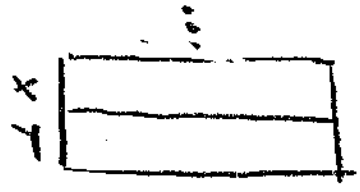
Zunächst



danach, x,y bleiben sichtbar,

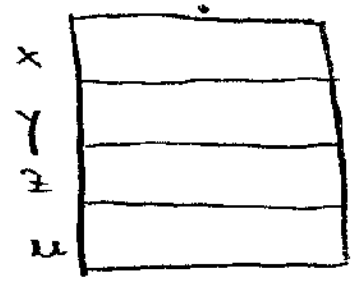


dann



z,u sind nach dem Block weg.

dann wieder:



Die Übergabe des Parameters erfolgt
in Java als

- Wertaufruf, call-by-value,
d.h. aktueller Parameter wird
ausgewertet und dem
aktuellen Frame übergeben.

Alternativen

- Referenzaufruf, call-by-reference.
Ist aktueller Parameter Variable
(Speicherplatz, Adresse). Der
formale Parameter wird ein
Alias des aktuellen.
Vermeidet Kopieren im
Laufzeit. Evtl. effizienter als

Verschiedene Fälle von Seiteneffekten.

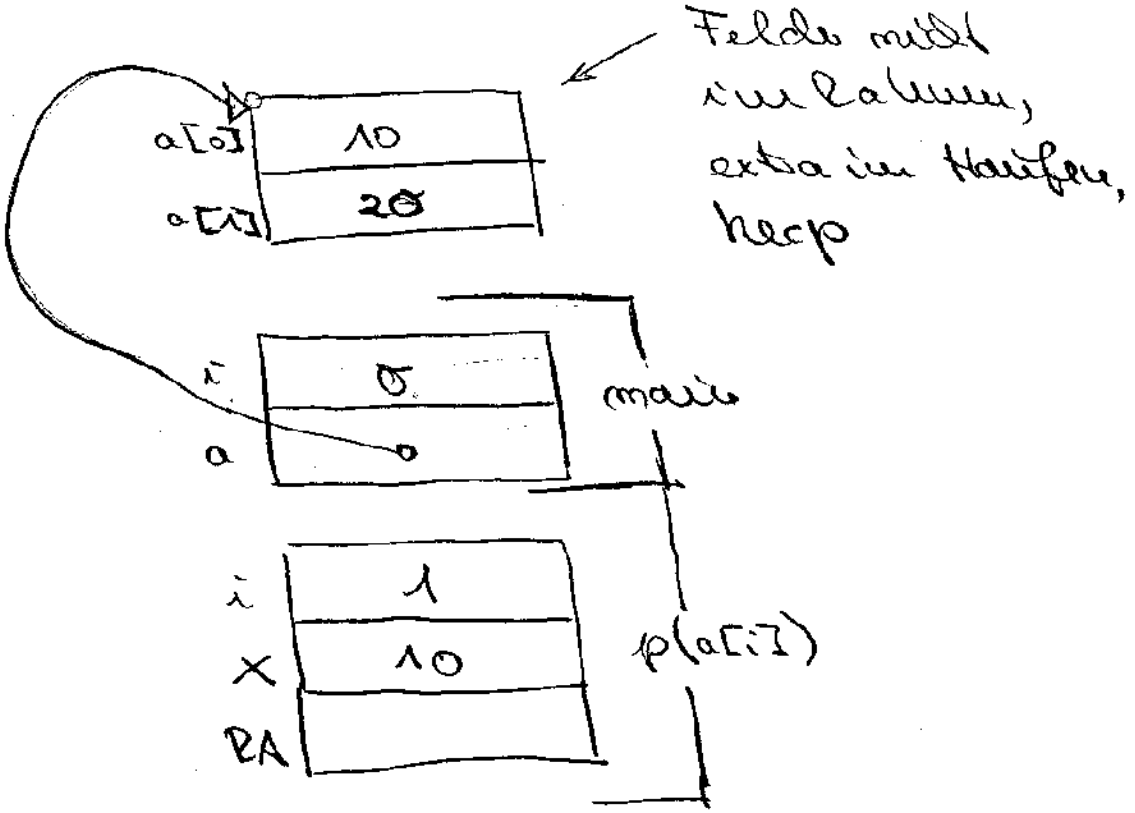
- Namensaufruf, call-by-name.
  Beim Aufruf einfaches schematisches Einkopieren des aktuellen Parameters so wie er ist.

Dazu ein Beispiel (Küchlin S. 154).

```
int [] a = new int [] {10, 20};

public static void p(int x) {
    int i = 0
    i = i + 1;
    x = x + 2
}
```

```
public static void main(String[] args)
{
    p(a[i])
    System.out.println(a[0]);
    System.out.println(a[1]);
}
```

Weiterlauf (Java):



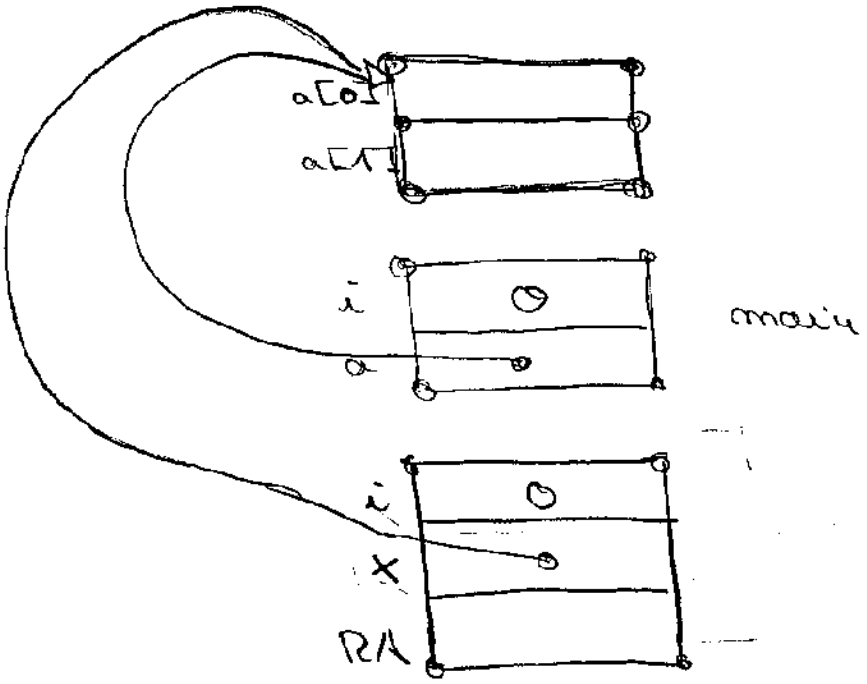
Dann x = 10. Ausgabe

10

20

(7.20)

Referenzaufruf (so nicht in Java):

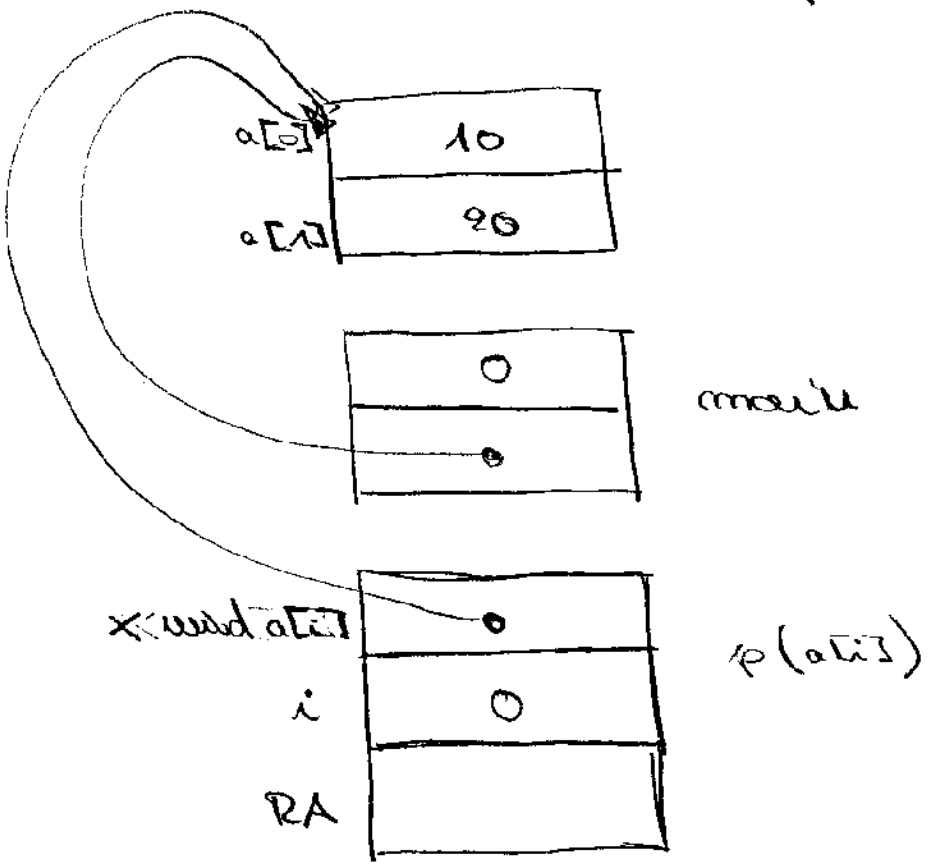


Da formale Parameter x wird zum Verweis auf den aktuellen a[0]. Ausgabe

12
20

(1.21)

Momentaufnahme (schon ganz nützlich im Labor):



Durch i = i+1 in p(a[i])

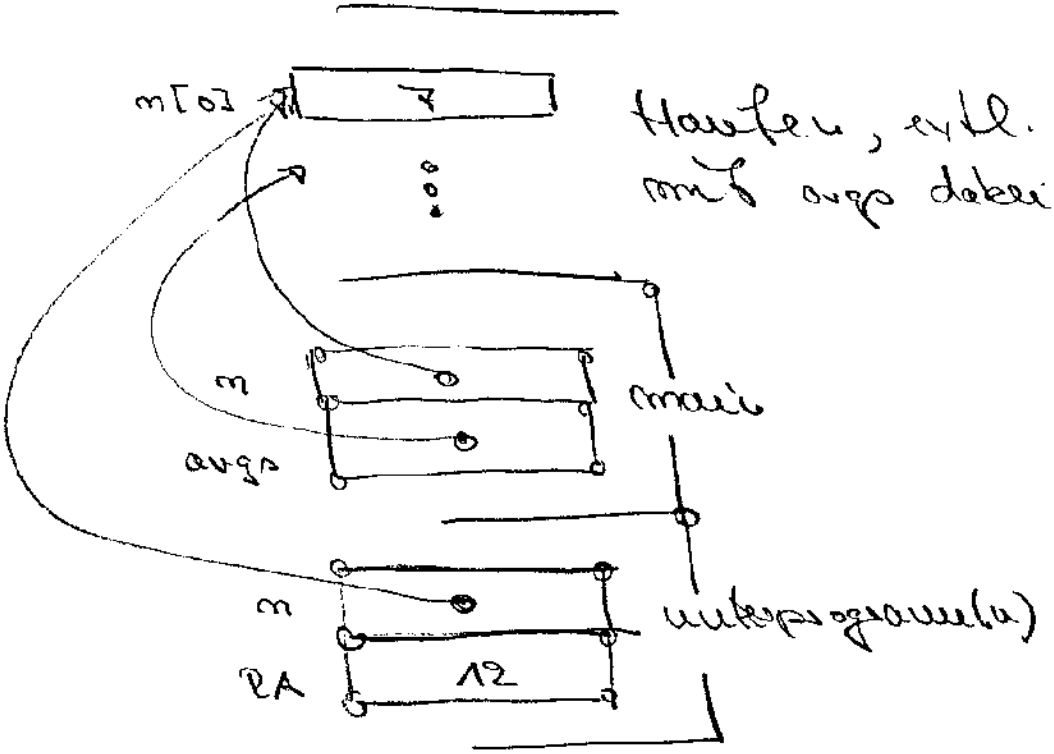
zeigt dann a[i] auf a[1].

Es wird a[1] geändert. Ausgabe

10

22 ,

Schließlich zu Feldern. Um das
unmöglich, langwierige Umkopieren
von Feldern zu vermeiden, werden
diese nicht in den Rahmen gespeichert,
sondern in den sogenannten Haufen
(hap). Keine Zugriffsbeschränkungen
wie im Lautzentkopf. Vgl. auch
Retz, Schiffler, Lesen S. 171 bis S. 175.
Lautzentheller und Haufen bei
Aufruf Test 2. java:



An den formalen Parameter n, eine Referenzvariable, wird der Wert von der Referenzvariable n übergeben. Im unterprogramm(n) wird das main im Hanoi geändert.

Abhilfe: Kopieren des Feldes. Kopie ans Hanoi. Problem: Nicht mehr

erweitbaus Speicherplatz (garbage collection im Laufzeitsystem integriert).

Am Buch bis S. 175. Dazu einige Erläuterungen.

Bisher hatten wir unsere Klassen mit Instanzvariablen ausgestattet:

```
public class <Klassenname> {
    public int x
    public float[] y
    :
}
```

Klassenvariablen werden mit static gekennzeichnet:

```
public class <Name> {
    public int x
    public static float a
    static long b
    :
}
```

Methoden

public static void <Name> <Para>

Klassenvariablen sind in allen

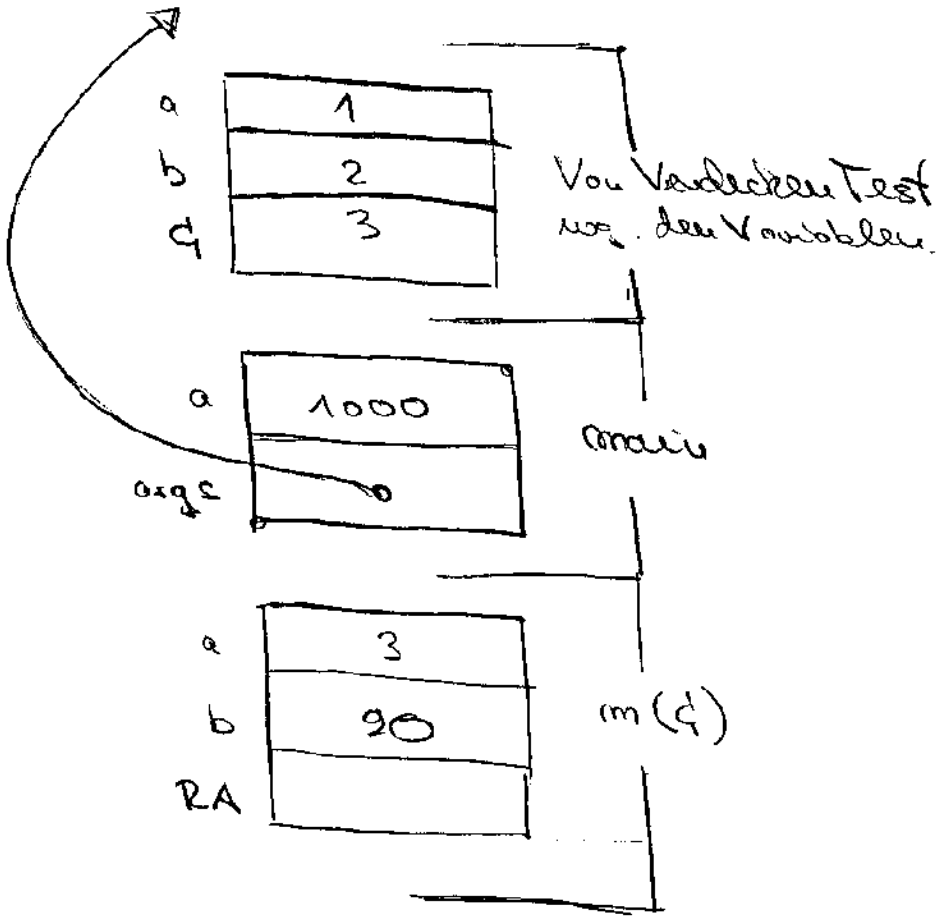
Instanzen einer Klasse gleich.

Für alle Instanzen existiert nur

eine Klassenvariable sozusagen.

Dazu eventuell später noch mehr.

Laufzeitbild von Verdeckertest:



Situation nach Beginn von m(q)

Ausgabe ausgesamt:

a= 1000 (Zeile 11

b= 2 12

m( )= 13

$a = 3$	Teile 5
$b = 28$	6
$q = 3$	7
$am(q) = 100$	13

↑

Erst wird das Ausgabeunde ausgerechnet als ganzes, dann wird ausgegeben.

7,22

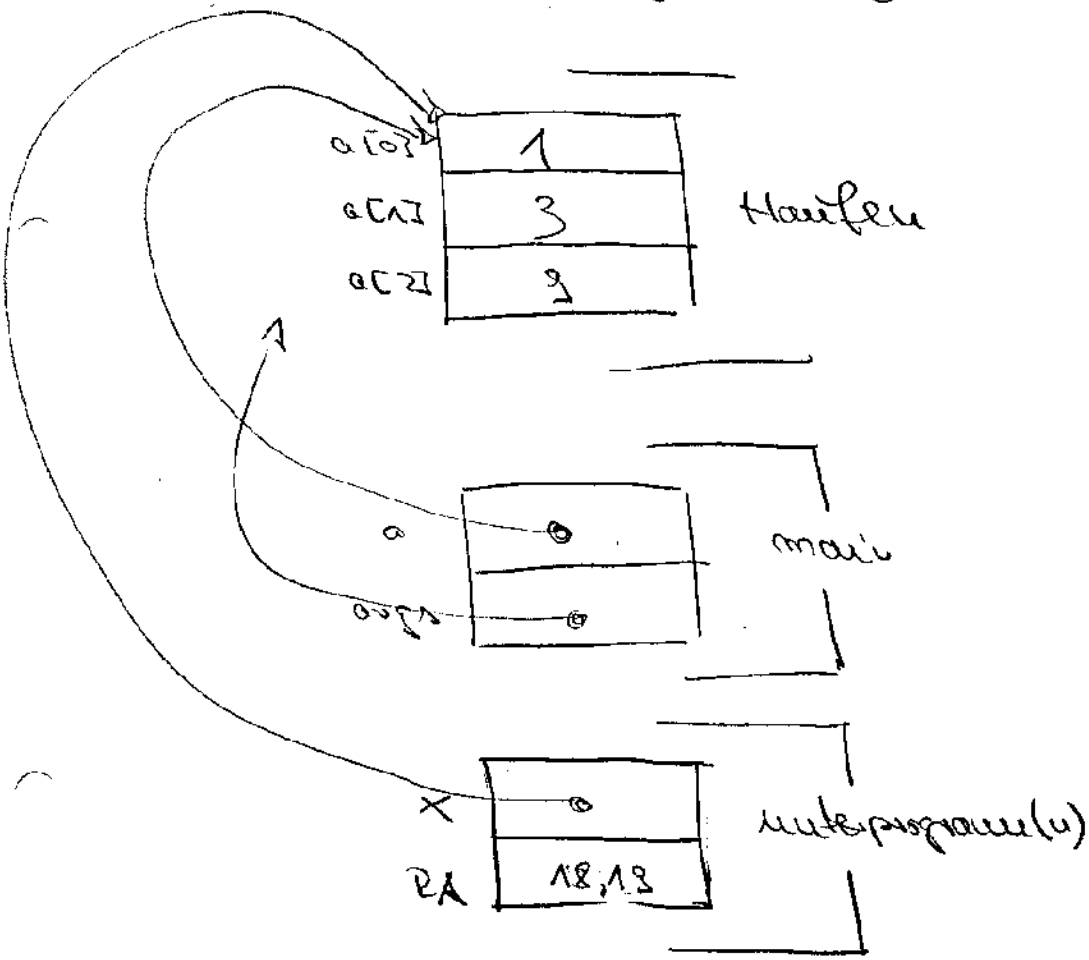
```
public class NocheinTest{
  //
  //
  //
  public static int[] unterprogramm(int[] x) {  // (1)
  x[0]= 7;  // (2)
  x = new int[2];  //(3)
  return x;
  }
  //
  // Hier ist das Unterprogramm zu Ende
  //
  public static void main(String[] args){
  int[] a = new int[3];
  a[0] = 1;
  a[1] = 3;
  a[2] = 9;
  a = unterprogramm(a); // (4)
  System.out.println(a[0] + "und " + a[1] );  //a[2] gibt Grenzueberschreitung,
  }
  }
```

// zum Laufzeit.

Schleifbild zu abschließendem Beispiel:

Nochmal Text. java von letzte Seite.

Haufen und Laufzeitkelle nach (1):

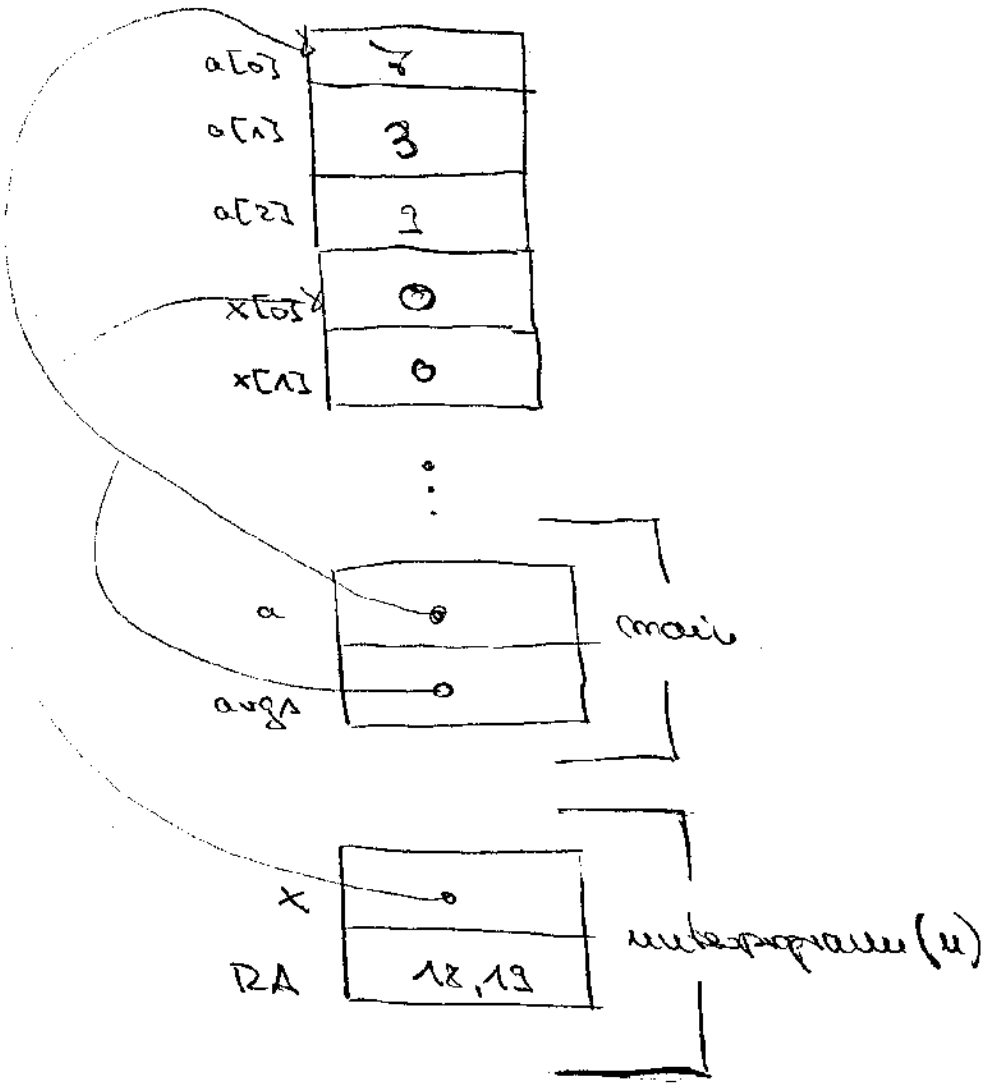


Nach (2) alles gleich außer

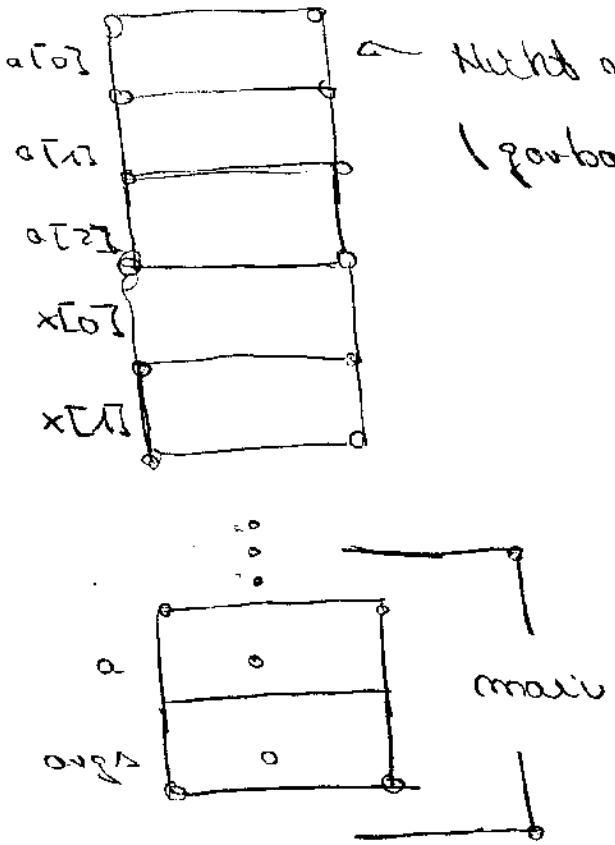
x[0] = a[0] = 7.

x wird ja wie die Referenzvariable a übergeben.

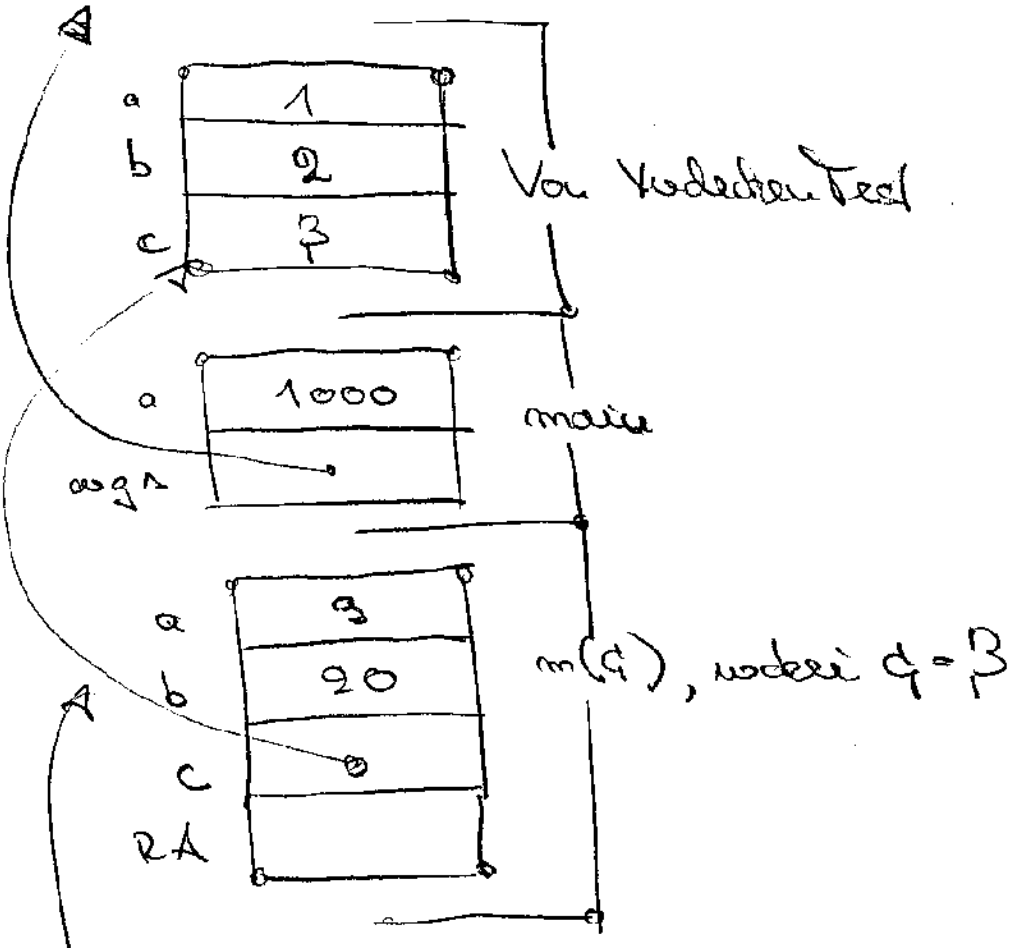
Nodo (3)



Schleifblock mode(4), s. auch a = x



Schließlich noch eine Korrektur zu S. 7.26.



Die Klassenvariable 4 bleibt sichtbar. Es ist auch möglich, 4 in m(a int) zu ändern.

1.33

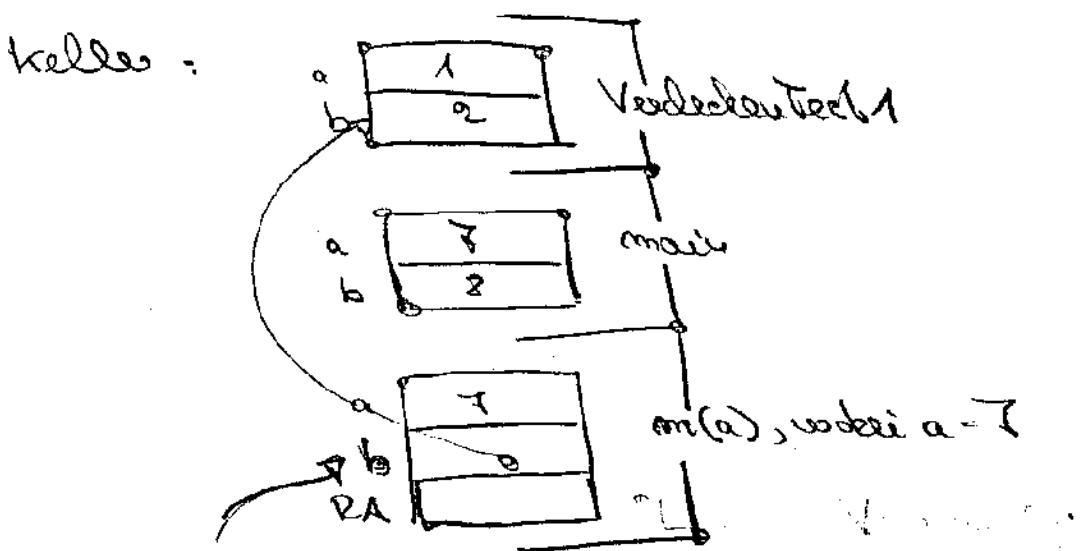
Noch ein Nachtrag zum Verdeckungsproblem, die Programme auf den folgenden Seiten

Verdecken Test 1. java

Verdecken Test 2. java.

```
public class VerdeckenTest1{  // Hier ein Test von Methoden.
  //
  // Die Deklaration beginnt mit
  // 2 Klassenvariablen: "static"
  //
  static int a = 1, b = 2;
  //
  // Jetze eine Methode
  //
  public static void m(int a){
     System.out.println("Der aktuelle " +
                        " Parameter fuer a  ist " + a);
     System.out.println("Der aktuelle " +
                        " Wert  fuer b  ist " + b);
     }
  //
  // Die Deklarationen enden.
  //
  public static void main(String[] args) {
  int a = 7, b= 8;
  //
  //
  // Deklaration von lokalen Variablen .
  //
  //
  //
  m(a) ;// Die Frage ist, welches b wird in diesem
        // Aufruf genommen.  Das von der Klasse, b= 2.
   }
  }
```

Evgibl Ausgabe a = 7, b = 2.



Wieso das lokale b aus main?

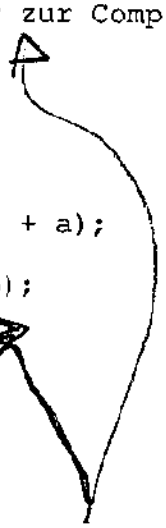
Das b kann in m(int a) auch geändert werden.

Also Regel: Lokale Variablen einer aufrufenden Methode sind in einer aufgerufenen Methode prinzipiell nicht sichtbar.

Das ist auch vollkommen sinnvoll. Sonst würde sich ein Programm nach Änderung von Namen von lokalen Variablen anders verhalten.

Klassenvariablen sind sichtbar, sofern nicht überschrieben.

```
public class VerdeckenTest2{  // Hier ein Test von Methoden.
  //
  // Die Deklaration beginnt mit
  // 2 Klassenvariablen: "static"
  //
  static int a = 1 ;// Im Vergleich zu VerdeckenTest1 , b = 2;
  //                  Bekommen Compilerfehler, da b in Zeile 15
  //                  nicht bekannt ist. Offensichtlich werden alle
  //                  Variablen ausser Parameter zur Compile-Zeit
  //                  sozusagen identifiziert.
  //
  // Jetze eine Methode
  //
  public static void m(int a){
     System.out.println("Der aktuelle " +
                        " Parameter fuer a  ist " + a);
     System.out.println("Der aktuelle " +
                        " Wert  fuer b  ist " + b);
     }
  //
  // Die Deklarationen enden.
  //
  public static void main(String[] args) {
  int a = 7, b= 8;
  //
  //
  // Deklaration von lokalen Variablen .
  //
  //
  //
  m(a) ;// Die Frage ist, welches b wird in diesem
        // Aufruf genommen.
   }
  }
```



Dieses Programm übersetzt
gar nicht erst. Aber der
b ist vom Compiler nicht
identifiziert.