

Mehrstufige und mehrdimensionale Reihungen

In Java gibt es nur *mehrstufige Reihungen*, die auch als *Reihungen von Reihungen* bezeichnet werden (englisch: nested arrays, arrays of arrays). In Fortran und Pascal gibt es nur *mehrdimensionale Reihungen* (englisch: multi dimensional arrays). In ADA und in C/C++ gibt es *mehrstufige* und *mehrdimensionale* Reihungen.

Eine mehrstufige Reihung enthält Referenzen, die auf Komponenten-Reihungen zeigen. Diese Komponenten-Reihungen *können unterschiedlich lang* sein (aber man kann sie natürlich auch alle gleich lang machen). Man sagt auch (mit einem Fachbegriff aus der Druck- und Satz-Technik): Eine *mehrstufige* Reihungen kann einen *Flatterrand* haben, bei einer *mehrdimensionalen* Reihung ist das nicht möglich (d.h. alle Komponenten-Reihungen müssen gleich lang sein).

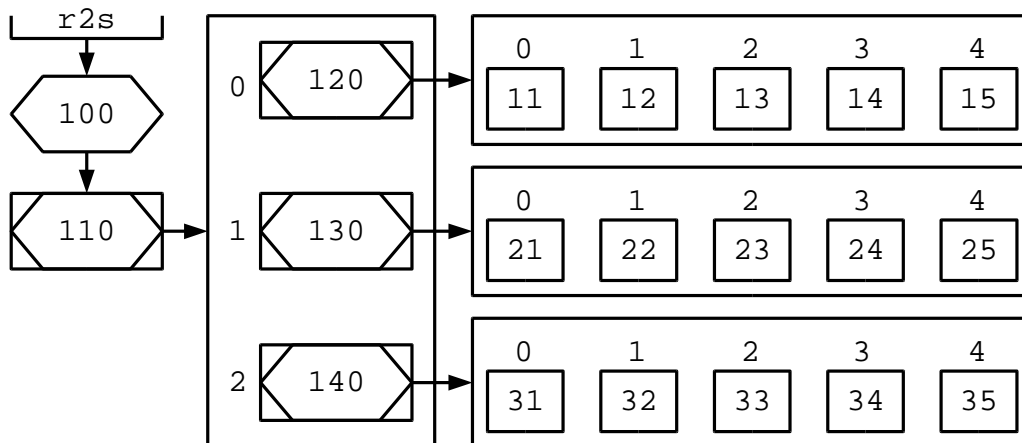
Die Begriffe *1-stufige Reihung* und *1-dimensionale Reihung* bedeuten dasselbe. Wodurch *mehrstufige* (z.B. *2-stufige* oder *3-stufige* oder ...) Reihungen und *mehrdimensionale* (z.B. *2-dimensionale* oder *3-dimensionale* oder ...) Reihungen sich unterscheiden, sollen die folgenden Beispiele veranschaulichen.

Beispiel-01: Eine 2-stufige Reihung von `long`-Variablen (bei der alle Komponenten-Reihungen "freiwillig" gleich lang gemacht wurden).

Die Vereinbarung der Reihung in Java:

```
long[][] r2s = {{11, 12, 13, 14, 15}, {21, 22, 23, 24, 25}, {31, 32, 33, 34, 35}};
```

Als Boje dargestellt sieht die Variable `r2s` etwa so aus:



Die Reihung `r2s` enthält 3 Komponenten vom Typ `long[]` (Reihung von `long`).

Jede Komponente `r2s[i]` ist eine Reihung, die 5 Komponenten vom Typ `long` enthält.

Von den 3 Komponenten der Reihung `r2s` (und ebenso von den 3 mal 5 gleich 15 `long`-Komponenten der drei Reihungen `r2s[0]`, `r2s[1]` und `r2s[2]`) wurden nur die *Werte* gezeichnet, ihre *Referenzen* und die unveränderliche Variable `length` wurden zur Vereinfachung weggelassen.

Die Bojen-Darstellung macht deutlich: In Wirklichkeit enthält die Reihung `r2s` gar keine Reihungen, sondern nur *Referenzen*, die auf Reihungen zeigen (im Beispiel sind das die Referenzen `<120>`, `<130>` und `<140>`). Die Reihungen mit den 5 `long`-Komponenten liegen *außerhalb* des Rechtecks, welches die Reihung `r2s` darstellt.

Beispiel-02: Eine 2-dimensionale Reihung von long-Variablen

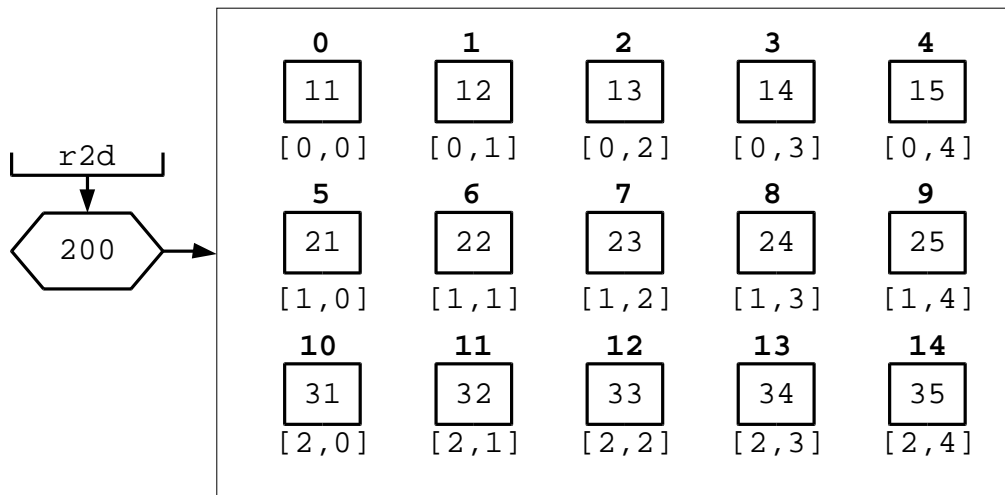
(vereinbart in einer Java-ähnlichen Phantasie-Sprache):

```
long[3, 5] r2d = {{11, 12, 13, 14, 15}, {21, 22, 23, 24, 25}, {31, 32, 33, 34, 35}};
```

Zum Typ `long[3, 5]` (den es in Java *nicht* gibt!) sollen 2-dimensionale Reihungen gehören, die aus 3x5 long-Variablen bestehen ("3 Zeilen mit je 5 Spalten").

Nebenbei: Hier wird angenommen, dass dieser Typ `long[3, 5]` einem primitiven Typ in Java (oder einem value type in C#) entspricht, und nicht einem Referenz-Typ oder Pointer-Typ.

Als Boje dargestellt sieht die Variable `r2d` etwa so aus:



Diese Darstellung soll deutlich machen, dass die Reihung `r2d` nur (3 mal 5 gleich 15) long-Komponenten enthält, und keine Reihungen oder Referenzen-die-auf-Reihungen-zeigen.

Jede dieser 15 Komponenten hat einen Index zwischen 0 und 14, wie bei einer 1-stufigen Reihung. Mit diesen Indizes greift der *Ausführer* auf die Komponenten zu.

Für den *Programmierer* hat jede der 15 Komponenten zusätzlich auch ein *Index-Paar*, z.B. [0, 0] oder [1, 3] oder [2, 2] etc. Der erste Index eines solchen Paares bezeichnet eine Zeile und der zweite eine Spalte.

Zwischen dem einen Index i einer Komponenten (z.B. 8) und dem Index-Paar $[i_1, i_2]$ derselben Komponenten (im Beispiel: [1, 3]) kann man ziemlich leicht wie folgt hin und her rechnen.

Für 2-dimensionale Reihungen der Größe 3 x 5 gilt:

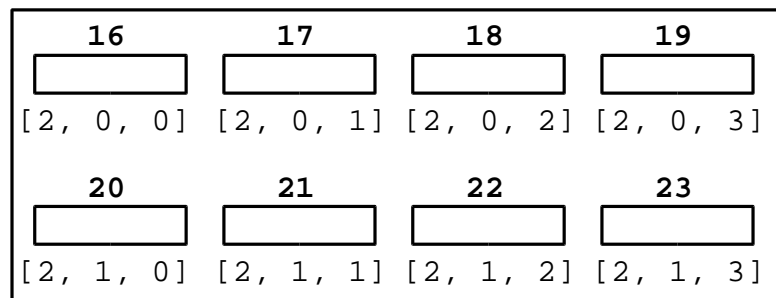
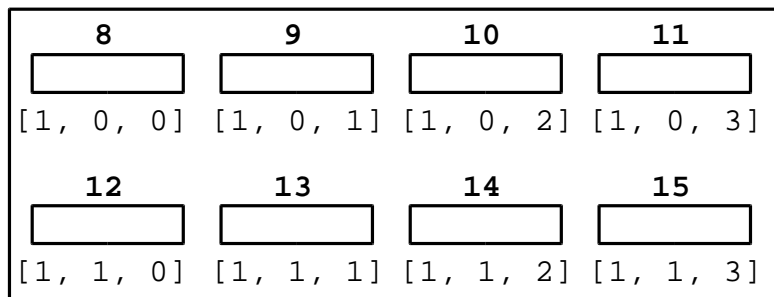
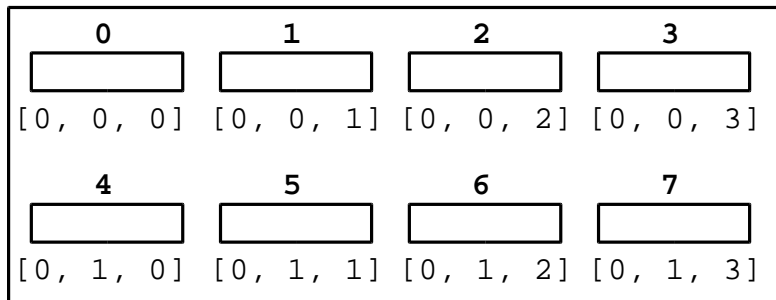
i	->	$[i/5, i\%5]$
$i_1*5 + i_2$	<-	$[i_1, i_2]$

Allgemein gilt für eine 2-dimensionale Reihungen der Größe $d_0 \times d_1$:

i	->	$[i/d_1, i\%d_1]$
$i_1*d_1 + i_2$	<-	$[i_1, i_2]$

Der Ausführer muss nicht bei jedem Zugriff den 2-dimensionalen Index $[i_1, i_2]$ des Programmierers in seinen eigenen 1-dimensionalen Index i umrechnen: Wenn alle Komponenten der Reihung (oder mehrere hintereinanderliegende Komponenten) bearbeitet werden sollen, braucht der Ausführer seinen Index i nur um 1 zu erhöhen, um von einer Komponenten zur nächsten zu kommen. Deshalb können *mehrdimensionale* Reihungen in wichtigen Fällen etwas schneller sein als *mehrstufige* Reihungen (wenn man einen guten Compiler hat). Außerdem enthalten mehrdimensionale Reihungen keine Referenzen, die auf Komponenten-Reihungen zeigen und benötigen deshalb weniger Speicherplatz.

Beispiel-03: Eine 3-dimensionale Reihung r_{3d} der Größe $3 \times 2 \times 4$ (3 Ebenen mit je 2 Zeilen mit je 4 Spalten) sieht etwa so aus:



Diese Reihung hat $3 \times 2 \times 4$ gleich 24 Komponenten. Jede Komponente hat einen Index i zwischen 0 und 23, wie bei einer 1-stufigen Reihung. Mit diesen Indizes greift der *Ausführer* auf die Komponenten zu.

Für den *Programmierer* hat jede der 24 Komponenten zusätzlich auch ein *Index-Tripel*, z.B. $[0, 0, 0]$ oder $[1, 1, 3]$ oder $[2, 0, 2]$ etc. Der erste Index eines solchen Paares bezeichnet eine Ebene, der zweite eine Zeile und der dritte eine Spalte.

Zwischen dem *einen* Index i einer Komponenten (z.B. 10) und dem *Index-Tripel* $[i_1, i_2, i_3]$ derselben Komponenten (im Beispiel: $[1, 0, 2]$) kann man ziemlich leicht wie folgt hin und her rechnen. Für 3-dimensionale Reihungen der Größe $3 \times 2 \times 4$ gilt:

i	->	$[i/(2 \cdot 4), (i \bmod (2 \cdot 4))/4, i \bmod 4]$
$i_1 \cdot (2 \cdot 4) + i_2 \cdot 4 + i_3$	<-	$[i_1, i_2, i_3]$

Allgemein gilt für eine 3-dimensionale Reihungen der Größe $d_0 \times d_1 \times d_2$:

i	->	$[i/(d_1 \cdot d_2), (i \bmod (d_1 \cdot d_2))/d_2, i \bmod d_2]$
$i_1 \cdot (d_0 \cdot d_1) + i_2 \cdot d_2 + i_3$	<-	$[i_1, i_2, i_3]$

Um "die Systematik" hinter den Formeln für die Umrechnung eines 1-dimensionalen Index i in einen mehrdimensionalen Index $[i_0, i_1, \dots]$ leichter erkennbar zu machen, ergänzen wir die Formeln um ein paar unnötige, aber harmlose Rechenoperationen wie z.B. $* 1$ oder $/ 1$ etc. wie folgt:

2-dimensional d_0*d_1 : $i \rightarrow [i_0, i_1]$

$$i_0 = (i \% (d_0*d_1)) / (d_0*1)$$

$$i_1 = (i \% (d_1)) / (1)$$

3-dimensional $d_0*d_1*d_2$: $i \rightarrow [i_0, i_1, i_2]$

$$i_0 = (i \% (d_0*d_1*d_2)) / (d_1*d_2*1)$$

$$i_1 = (i \% (d_1*d_2)) / (d_2*1)$$

$$i_2 = (i \% (d_2)) / (1)$$

4-dimensional $d_0*d_1*d_2*d_3$: $i \rightarrow [i_0, i_1, i_2, i_3]$

$$i_0 = (i \% (d_0*d_1*d_2*d_3)) / (d_1*d_2*d_3*1)$$

$$i_1 = (i \% (d_1*d_2*d_3)) / (d_2*d_3*1)$$

$$i_2 = (i \% (d_2*d_3)) / (d_3*1)$$

$$i_3 = (i \% (d_3)) / (1)$$

Aufgabe-01: Die hier angegebenen Formeln enthalten einige eigentlich überflüssige Rechenoperationen. In welchen Zeilen ist eine $\%$ -Operation überflüssig?
In welchen Zeilen ist eine $/$ -Operation überflüssig?
Welche $*$ -Operationen sind überflüssig?

Aufgabe-02: Wie sehen die entsprechenden Formeln für den 5-dimensionalen Fall aus?

Anmerkung: In der offiziellen Spezifikation der Programmiersprache Java, siehe z.B.

<https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>

steht auf S. 4 folgender Abschnitt:

"Chapter 10 describes arrays. Array accesses include bounds checking. Arrays are dynamically created objects and may be assigned to variables of type Object. The language supports arrays of arrays, rather than multidimensional arrays."

Das bedeutet, dass es in Java nur *mehrstufige* Reihungen (engl. nested arrays) gibt und keine *multidimensionalen* Reihungen (engl. multidimensional arrays). Im Internet werden in vielen Artikeln über Java trotzdem *mehrdimensionale Reihungen* (oder: *mehrdimensionale Arrays*) erwähnt, wo eigentlich *mehrstufige Reihungen* gemeint sind.

Lösung-01: Die hier angegebenen Formeln enthalten einige Rechenoperationen, die dabei helfen, "die Systematik" der Formeln deutlich zu machen, die aber eigentlich überflüssig sind.

In welchen Zeilen ist eine %-Operation überflüssig?

In welchen Zeilen ist eine /-Operation überflüssig?

Welche *-Operationen sind überflüssig?

In den Formeln für i_0 ist jeweils die Operation $i \ \% \ \dots$ überflüssig, weil der zweite Operand dieser Operation immer größer als i ist und deshalb das Ergebnis der Operation immer gleich i ist.

Im 5-dimensionalen Fall enthält die Formel für i_4 eine überflüssig Operation $\dots / 1$,

im 4-dimensionalen Fall enthält die Formel für i_3 eine überflüssig Operation $\dots / 1$,

...

Alle Operationen $\dots * 1$ sind natürlich auch überflüssig (unterstreichen aber die Systematik).

Lösung-02: Wie sehen die entsprechenden Formeln für den 5-dimensionalen Fall aus?

5-dimensional $d_0 * d_1 * d_2 * d_3 * d_4$: $i \rightarrow [i_0, i_1, i_2, i_3, i_4]$

$$i_0 = (i \% (d_0 * d_1 * d_2 * d_3 * d_4)) / (d_1 * d_2 * d_3 * d_4 * 1)$$

$$i_1 = (i \% (d_1 * d_2 * d_3 * d_4)) / (d_2 * d_3 * d_4 * 1)$$

$$i_2 = (i \% (d_2 * d_3 * d_4)) / (d_3 * d_4 * 1)$$

$$i_3 = (i \% (d_3 * d_4)) / (d_4 * 1)$$

$$i_4 = (i \% (d_4)) / (1)$$