Mehrstufige und mehrdimensionale Reihungen

In Java gibt es *einstufige Reihungen* (z.B. Reihungen der Typen `long[]` oder `String[]`) und *mehrstufige Reihungen* (z.B. Reihungen der Typen `long[][]` oder `String[][][]`). Mehrstufige Reihungen werden auch als *Reihungen von Reihungen* (und im Englischen als *nested arrays* oder als *arrays of arrays*) bezeichnet.

Außer *mehrstufigen* Reihungen gibt es auch noch *mehrdimensionale* Reihungen (engl. *multidimenional arrays*), z.B. in den Programmiersprachen Fortran und Pascal (aber nicht in Java). In der Sprache C++ gibt es sogar beide Arten: mehrstufige und mehrdimensionale Reihungen.

In der Einleitung der Sprachbeschreibung "The Java Language Specification" von James Gosling et. al. steht: "The language [Java] supports arrays of arrays, rather than multidimensional arrays" (siehe zum Beispiel https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf , page 4). Im Internet wird an vielen Stellen der Unterschied zwischen *mehrstufigen* und *mehrdimensionalen* Reihungen ignoriert und die Bezeichnung "mehrdimensional" auch auf mehrstufige Reihungen (z.B. in Java) angewendet. Das ist bedauerlich, da der Unterschied interessant und praxisrelevant ist.

Der Unterschied in Kürze: Eine mehrstufige Reihung enthält Referenzen (oder: Adressen, Pointer), die auf Komponenten-Reihungen zeigen. Diese Komponenten-Reihungen *können unterschiedlich lang* sein (aber natürlich kann man sie auch alle gleich lang machen). Man sagt auch (mit einem Fachbegriff aus der Druck- und Satz-Technik): Eine *mehrstufige* Reihung kann einen *Flatterrand* haben. Eine *mehrdimensionale* Reihung enthält keine Referenzen (die auf Komponenten-Reihungen zeigen) und ist garantiert "rechteckig" (d.h. sie hat garantiert *keinen* Flatterrand).

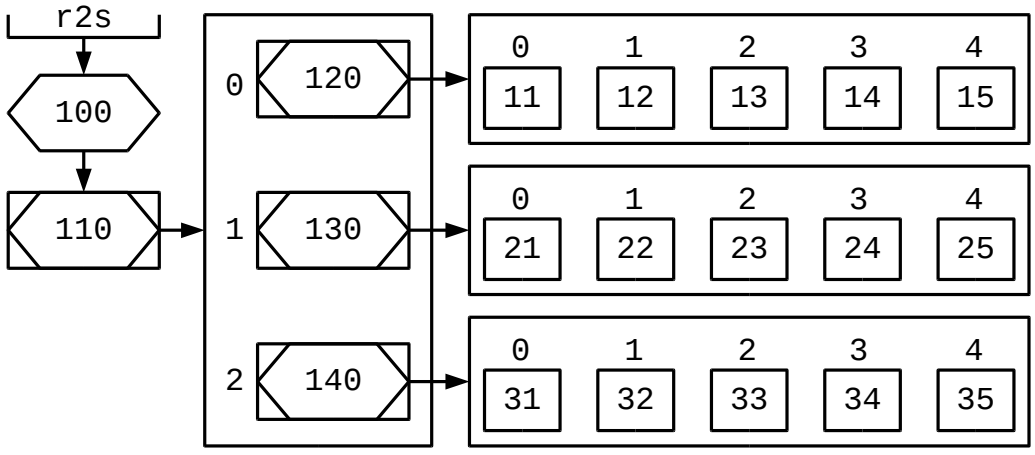
Die Begriffe *1-stufige Reihung* und *1-dimensionale Reihung* bedeuten dasselbe. Den Unterschied zwischen *mehrstufige* und *mehrdimensionale* Reihungen sollen die folgenden Beispiele deutlich machen.

Beispiel-01: Eine **2-stufige** Reihung von `long`-Variablen
(bei der alle Komponenten-Reihungen "freiwillig" gleich lang gemacht wurden, damit dieses Beispiel besser zum nachfolgenden **Beispiel-02** passt).

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.

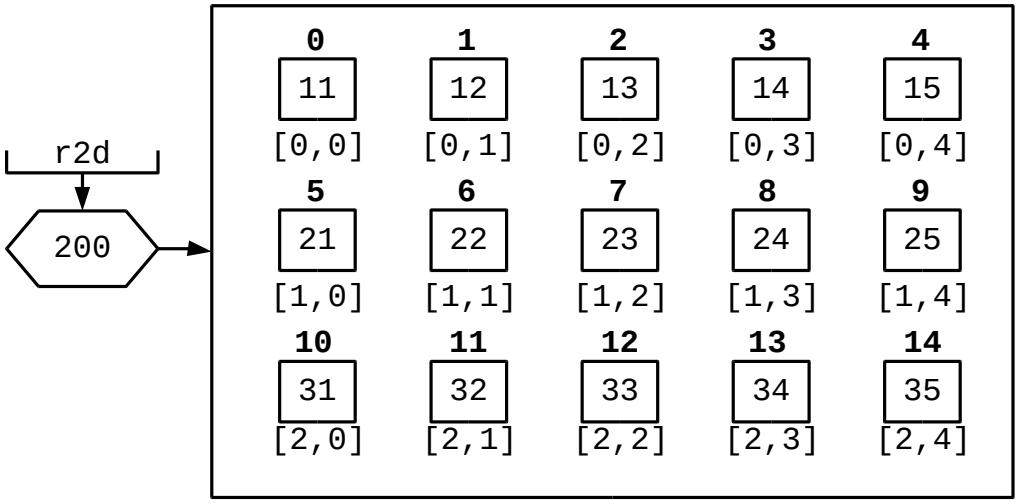
Beispiel-02: Eine **2-dimensionale** Reihung von `long`-Variablen
(vereinbart in einer Java-ähnlichen Phantasie-Sprache):

```
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 Phantasie-Typ `long[3, 5]` einem primitiven Typ in Java (oder einem value type in C#) entspricht, und nicht einem Referenztyp 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 `[i1, i2]` 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]
i1*5 + i2	<-	[i1, i2]

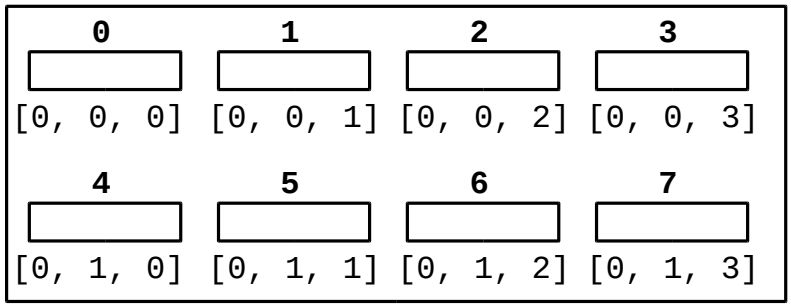
Allgemein gilt für 2-dimensionale Reihungen der Größe `d0 x d1`:

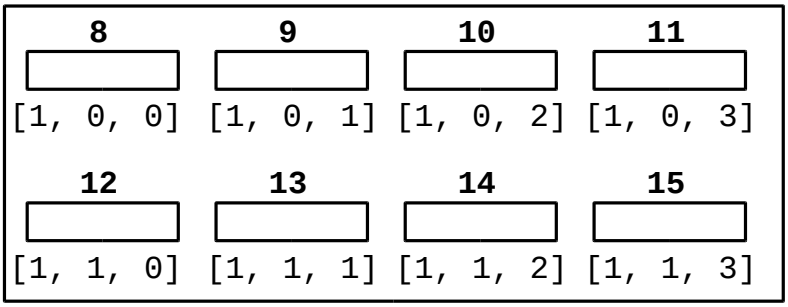
i	->	[i/d1, i%d1]
i1*d1 + i2	<-	[i1, i2]

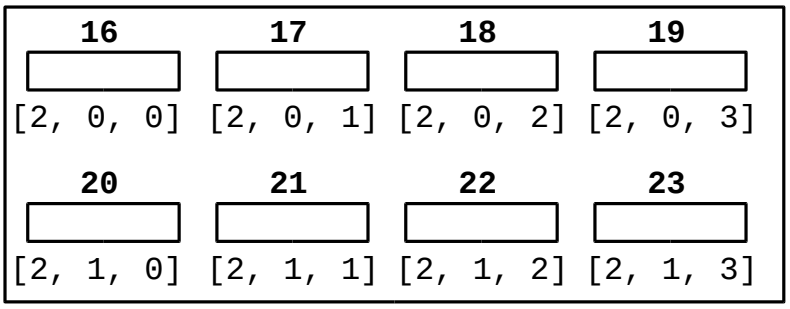
Der Ausführer muss nicht bei jedem Zugriff den 2-dimensionalen Index `[i1, i2]` des Programmierers in seinen eigenen 1-dimensionalen Index `i` umrechnen: Wenn alle Komponenten der Reihung (oder mehrere hintereinanderliegende Komponenten) bearbeitet werden sollen, braucht der Ausführer seinen Index `i` 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 `r3d` der Größe 3 x 2 x 4
(3 Ebenen mit je 2 Zeilen mit je 4 Spalten) sieht etwa so aus:







Diese Reihung hat 3*2*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 `[i1, i2, i3]` derselben Komponenten (im Beispiel: `[1, 0, 2]`) kann man ziemlich leicht wie folgt hin und her rechnen. Für 3-dimensionale Reihungen der Größe 3 x 2 x 4 gilt:

`i`	`->`	`[i/(2*4), (i%(2*4))/4, i%4]`
`i1*(2*4) + i2*4 + i3`	`<-`	`[i1, i2, i3]`

Allgemein gilt für 3-dimensionale Reihungen der Größe `d0 x d1 x d2`:

`i`	`->`	`[i/(d1*d2), (i%(d1*d2))/d2, i%d2]`
`i1*(d0*d1) + i2*d2 + i3`	`<-`	`[i1, i2, i3]`

Um "das System" oder "die Systematik" hinter den Formeln für die Umrechnung eines 1-dimensionalen Index `i` in einen mehrdimensionalen Index `[i0, i1, ...]` 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 `d0*d1: i -> [i0, i1]`

```
i0 = (i % (d0*d1)) / (d0*1)
i1 = (i % (   d1)) / (   1)
```

3-dimensional `d0*d1*d2: i -> [i0, i1, i2]`

```
i0 = (i % (d0*d1*d2)) / (d1*d2*1)
i1 = (i % (   d1*d2)) / (   d2*1)
i2 = (i % (      d2)) / (      1)
```

4-dimensional `d0*d1*d2*d3: i -> [i0, i1, i2, i3]`

```
i0 = (i % (d0*d1*d2*d3)) / (d1*d2*d3*1)
i1 = (i % (   d1*d2*d3)) / (   d2*d3*1)
i2 = (i % (      d2*d3)) / (      d3*1)
i3 = (i % (         d3)) / (         1)
```

Aufgabe-01: Wie sehen die entsprechenden Formeln für den 5-dimensionalen Fall aus?

Aufgabe-02: 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?

Lösung-01: Wie sehen die entsprechenden Formeln für den 5-dimensionalen Fall aus?

5-dimensional `d0*d1*d2*d3*d4: i -> [i0, i1, i2, i3, i4]`

```
i0 = (i % (d0*d1*d2*d3*d4)) / (d1*d2*d3*d4*1)
i1 = (i % (   d1*d2*d3*d4)) / (   d2*d3*d4*1)
i2 = (i % (      d2*d3*d4)) / (      d3*d4*1)
i3 = (i % (         d3*d4)) / (         d4*1)
i4 = (i % (            d4)) / (            1)
```

Lösung-02: 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 `i0` ist jeweils die Operation `i % ...` 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 `i4` eine überflüssig Operation `... / 1`,
im 4-dimensionalen Fall enthält die Formel für `i3` eine überflüssig Operation `... / 1`,
...

Alle Operationen `... * 1` sind natürlich auch überflüssig.