

Folge 7 - Arrays

7.1 Die Klasse Liste

Kennen Sie den Spruch vom “Sprung ins kalte Wasser?”. Genau das machen wir nämlich jetzt. Mir ist keine vernünftige und originelle Einleitung in das Thema “Arrays” eingefallen, also kommt jetzt einfach der Sprung ins kalte Wasser. Betrachten Sie folgenden Quelltext der Klasse **Liste**:

```
public class Liste
{
    int[] zehnZahlen;

    public Liste()
    {
        zehnZahlen = new int[10];
    }

    public void erzeugen()
    {
        for (int i=0; i<10; i++)
            zehnZahlen[i] = i*i;
    }

    public void ausgeben()
    {
        for (int k=0; k<10; k++)
            System.out.println("Zahl "+k+" = "+zehnZahlen[k]);
    }
}
```

Wir wollen nun diesen ersten Quelltext dieses Kapitels ausführlich besprechen.

Die Klasse **Liste** soll es ermöglichen, zehn int-Zahlen zu verwalten. Um diese zehn Zahlen zu speichern, wurden nicht etwa zehn einzelne Variablen deklariert:

```
public class Liste
{
    int zahl1, zahl2, zahl3, zahl4, zahl5
        zahl6, zahl7, zahl8, zahl9, zahl10;
```

sondern eine einzige, nämlich die Variable **zehnZahlen**.

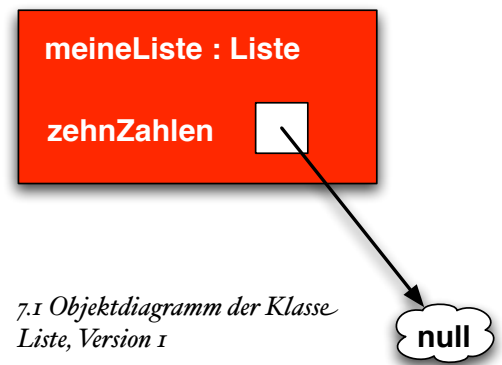
```
public class Liste
{
    int[] zehnZahlen;
```

Hinter dem Schlüsselwort `int` steht eine geöffnete eckige Klammer und dann sofort eine geschlossene eckige Klammer. Diese eckigen Klammern sind ganz typisch für **Arrays**. Die eckigen Klammern hinter `int` besagen soviel wie “jetzt wird nicht nur eine `int`-Variable deklariert, sondern eine ganze Reihe von `int`-Variablen”. Eine solche Reihe von Variablen gleichen Typs bezeichnet man als **Array** oder im Deutschen auch als **Feld**. Barnes und Kölling sprechen in ihrem Buch auch gern von “**fixed size collections**”.

```
public class Liste
{
    int[] zehnZahlen;
    int zehnZahlen;
```

Wenn Sie die eckigen Klammern hinter dem Datentyp vergessen, so wird kein Array deklariert, sondern eine einzige normale Variable.

Die Abbildung 7.1 zeigt mit Hilfe eines Objektdiagramms die Situation nach dem Deklarieren des Attributs **zehnZahlen**. Bei diesem Attribut handelt es sich - technisch gesehen - um einen **Zeiger** (Pointer) auf den noch zu initialisierenden Array. Da der Array noch nicht wirklich vorhanden ist, hat der Zeiger **zehnZahlen** den Wert null. Er zeigt sozusagen nirgendwo hin.

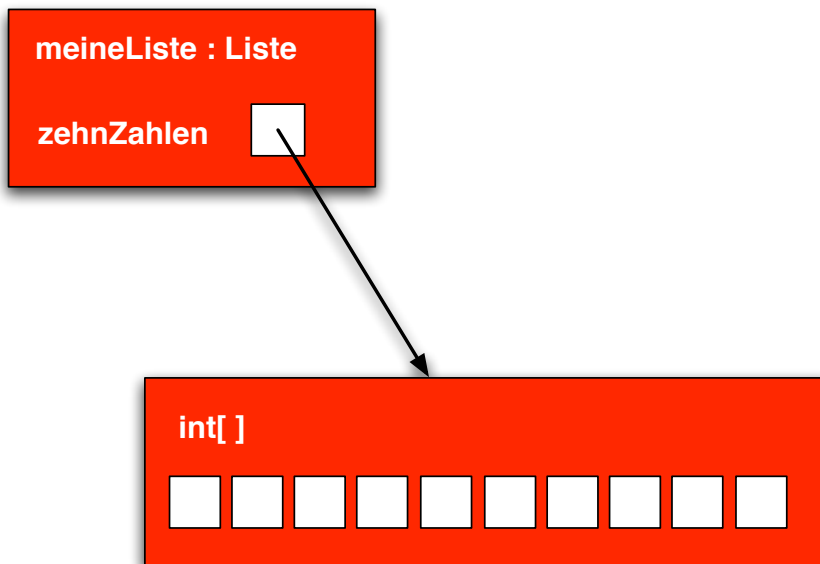


Im nächsten Schritt müssen wir den Array initialisieren. Das geschieht normalerweise im Konstruktor der Klasse:

```
public Liste()
{
    zehnZahlen = new int[10];
}
```

Ähnlich wie bei der Initialisierung eines Objektes wird hier der Operator **new** eingesetzt. Außerdem geben wir bei der Initialisierung auch die Anzahl der Array-Elemente an, die wir erzeugen wollen.

Nach der Initialisierung ist der Array als solcher vorhanden, es existieren jetzt also zehn Array-Elemente, die allerdings noch keinen definierten Wert haben. Betrachten wir wieder ein Objektdiagramm, das die entsprechende Situation graphisch darstellt:

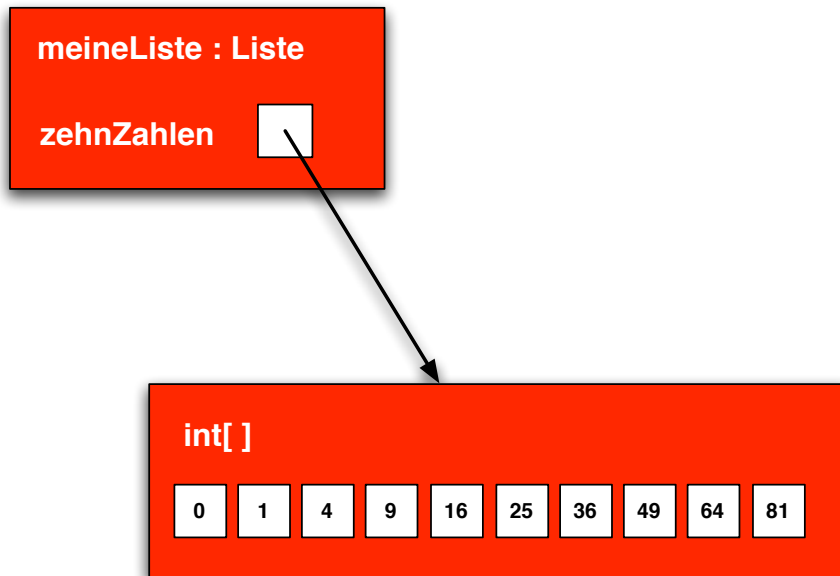


7.2 Objektdiagramm der Klasse *Liste*, Version 2

Der Zeiger **zehnZahlen** zeigt nicht mehr auf null, sondern auf einen konkreten Bereich im Arbeitsspeicher des Rechners. Dieser Bereich wird durch den unteren roten Kasten symbolisiert. In diesem Bereich sind jetzt auch zehn Speicherplätze für **int**-Zahlen reserviert, allerdings haben diese Speicherplätze noch keine Werte. Für die Wertbelegung ist die manipulierende Methode **erzeugen()** zuständig.

```
public void erzeugen()
{
    for (int i=0; i<10; i++)
        zehnZahlen[i] = i*i;
}
```

Die zehn Array-Elemente werden hier mit Werten belegt. Das erste Array-Element erhält den Wert 0, das zweite Array-Element den Wert 1, das dritte den Wert 4 und so weiter bis 81 für das letzte Element. Stellen wir die Situation nach dem Aufruf von **erzeugen()** graphisch dar, so erhalten wir wieder ein Objekt-Diagramm:



7.3 Objektdiagramm der Klasse Liste, Version 3

Ist Ihnen schon aufgefallen, dass jedes einzelne Array-Element über seinen **Index** angesprochen wird. Wenn wir das erste Array-Element mit dem Wert 1 belegen wollen, so schreiben wir

```
zehnZahlen[0] = 1;
```

Richtig gesehen: **Der Index des ersten Array-Elements ist nicht etwa 1, sondern 0.** Bei einem Array aus zehn Elementen ist der Index des letzten Elementes also nicht 10, sondern 9.

```
public void ausgeben()
{
    for (int k=0; k<10; k++)
        System.out.println("Zahl "+k+" = "+zehnZahlen[k]);
}
```

In der Methode **ausgeben()** werden die zehn Array-Elemente in der Konsole ausgegeben. Auch hier wird wieder eine `for`-Schleife eingesetzt, bei der die Laufvariable `k` den Index des auszugebenden Elementes bestimmt.

Wenn Sie nicht wollen, dass alle 100 Zahlen untereinander ausgegeben werden, so lassen Sie doch einfach immer zehn Zahlen nebeneinander ausgeben. Wie das geht? Ganz einfach - mit einer doppelten `for`-Schleife wie z.B.:

```
for (int zeile=0; zeile<10; zeile++)
{
    for (int spalte=0; spalte<10; spalte++)
        System.out.print("    "+zahl[zeile*10+spalte]);
    System.out.println();
}
```

Der Befehl **System.out.print()** schreibt die Zahlen direkt nebeneinander - ohne Zeilenvorschub. Wenn die zehn Zahlen einer Zeile ausgegeben wurden, muss aber ein Zeilenvorschub erzeugt werden. Daher wird der Befehl **System.out.println()** aufgerufen.