Inhalt

5 Referenzdatentypen - Felder 5-2

5.1 Eindimensionale Felder - Vektoren *5-3*

5.1.1 Vereinbarung 5-3
5.1.2 Referenzen sind keine Felder 5-4
5.1.3 Kopieren eindimensionaler Felder 5-6
5.1.4 Beispiel „Sortieren eines Vektors“ 5-6

5.2 Zweidimensionale Felder - Matrizen *5-8*

5.2.1 Vereinbarung 5-8
5.2.2 Kopieren zweidimensionaler Felder 5-9
5.2.3 Beispiel „Tic Tac Toe“ 5-9

5 Referenzdatentypen - Felder

Bisher wurde nur der Umgang mit *Elementardatentypen* behandelt. Für komplexe Anwendungen reichen diese oft nicht aus. Man benötigt zusammengesetzte, sogenannte *strukturierte Daten*. **Referenzdatentypen** ermöglichen den Zugriff auf solche.
Im Unterschied zu Elementardatentypen werden auf Werte von Referenzdatentypen *nicht direkt*, sondern *indirekt* über **Referenzen**, zugegriffen. Eine Referenz ist eine Variable, die als Wert eine Adresse besitzt und somit auf einen Speicherbereich verweist (Zeiger).
Java unterscheidet zwei Arten von Referenzdatentypen, **Felder** und **Klassen**.

Definition Feld (Array)
Feld der Länge *n* $=_{df}$ ***n*-Tupel über einer Menge von Daten *gleichen* Typs**

***Felder* sind *strukturierte* Datentypen, die für ein Programm eine Einheit bilden und Daten *gleichen* Typs zusammenfassen.**
Die zusammengefassten Daten nennt man *Feldkomponenten*.

Arbeitsspeicher

Symbolische Adresse	*Adresse im Speicher*	*Inhalt der Speicherzelle*	*Interpretations-vorschrift*
	...	...	
b	*5e*	00	`short` (2 Byte)
	5f	6a	
	...	...	
r	*65*	a3	`char[]` (Referenz)
	...	...	
	a3	00	`char[0]` (2 Byte)
	a4	41	
	a5	00	`char[1]` (2 Byte)
	a6	75	
	a7	00	`char[2]` (2 Byte)
	a8	74	
	a9	00	`char[3]` (2 Byte)
	aa	6f	
	...	...	...

Elementardatentyp

```
short b = 106;
```

b ist eine Variable des *Elementardatentypen* `short`:
b hat die Speicheradresse $(5e)_{16} = 94$, 2 Byte Länge und den Wert $(00\ 6a)_{16} = 106$.

Referenzdatentyp

```
char[] r = { 'A', 'u', 't', 'o'};
```

r ist eine *Referenzvariable*:
r hat im Speicher die Adresse $(65)_{16}$. Dort steht als *Wert* wiederum eine Adresse $(a3)_{16}$. Diese zeigt auf den Anfang eines *Feldes*. Um das Feld auswerten zu können, muss die *Anzahl* und der *Typ* der *Feldkomponenten* bekannt sein.
Im Beispiel handelt es sich um ein Feld mit 4 Komponenten vom Typ `char`. Die einzelnen Feldkomponenten bestehen jeweils aus 2 Byte. Der Speicherbereich, auf den die Referenz verweist, umfasst damit insgesamt 8 Byte und repräsentiert 4 Zeichen im Unicode, hier „Auto“.

5.1 Eindimensionale Felder - Vektoren

5.1.1 Vereinbarung

Analog einfachen Variablen haben Feldvariablen einen **Namen** und einen **Typ**, aber **keinen Wert**. Daten, durch ein Feld zusammengefasst, werden in **Feldkomponenten** abgespeichert. Diese beinhalten die **Werte** eines Feldes.

Deklaration

Mit der Deklaration eines Feldnamens wird dem Compiler mitgeteilt, dass es sich um eine *Referenzvariable* handelt und *Speicherplatz für eine Adresse* zur Verfügung gestellt werden muss.

Komponententyp **[ ]** *Feldname* **;**

```
float[] vektor;
```

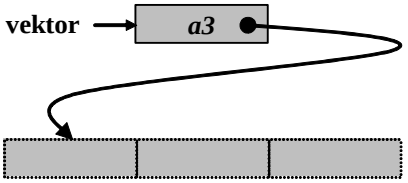
- Die Variable *Feldname* hat als Wert eine Adresse und ist ein **symbolischer Bezeichner** für den Feldanfang im Speicher. Anfangs zeigt diese „nirgendwohin", d.h. es handelt sich um eine sogenannte **null-Referenz**.
- Als *Komponententyp* kann ein *einfacher* aber auch ein *strukturierter Datentyp* eingesetzt werden.

Definition

Um ein Feld für den Gebrauch vorzubereiten, muss der notwendige Speicherplatz für die Komponenten reserviert werden. Dies geschieht mit Hilfe des ***new***-Operators.

Feldname **=** **new** *Komponententyp* **[** *Länge* **] ;**

```
vektor = new float[ 3];
```



- Unter der *Länge* eines Feldes versteht man die Anzahl seiner Feldkomponenten. Diese kann durch einen *ganzzahligen Ausdruck* festgelegt werden.
- Die *Länge* und der *Komponententyp* eines Feldes legen eindeutig den benötigten Speicherbedarf fest. Dieser wird *dynamisch zur Laufzeit* reserviert.

Deklaration und Definition

```
float[] vektor
  = new float[ IOTools.readInteger( "Anzahl der Werte: ")];
```

Zugriff auf die Feldkomponenten

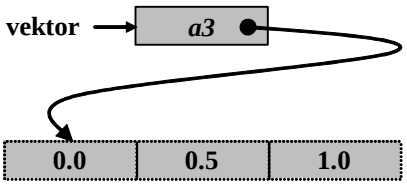
Auf einzelne Feldkomponenten wird über einen ganzzahligen **Index** durch den **[ ]-Operator** zugegriffen.

Feldname **[** *Index* **]**

```
vektor[ 0]      vektor[ 1]     vektor[ 2]
```

- Die einzelnen Feldkomponenten werden über ihren *Index* ($0 \leq Index < Länge$) identifiziert. Er kann durch einen *ganzzahligen Ausdruck* berechnet werden.
- Operationen mit Feldkomponenten sind analog den *einfachen Variablen* erlaubt.

Durchlaufen eines eindimensionalen Feldes




```
for( int i = 0; i < 3; i++)
  vektor[ i] = (float)( i / 2.);
```
[1]

Die **Laufvariable** `i` heißt auch **Indexvariable**, weil `i` alle möglichen Komponenten über den Index des Feldes `vektor` durchläuft.

Zu jedem Feld wird dessen *Länge* als ganzzahlige Variable `length` zusätzlich im Speicher abgelegt. Diese kann im Programm abgefragt werden und sollte grundsätzlich, anstatt einer expliziten Längenangabe, verwendet werden.

```
for( int i = 0; i < vektor.length; i++)
  vektor[ i] = (float)( i / 2.);
```

Explizite Anfangswertzuweisung (Initialisierung)
Felder können durch eine Wertemenge bei Ihrer Deklaration definiert und initialisiert werden. Die Länge des Feldes richtet sich nach der Anzahl der Werte in der Wertmenge. Für die Werte wird der notwendige Speicher bereitgestellt.

Komponententyp **[ ]** *Feldname* **=** *Wertmenge* **;**

- Die *Wertmenge* ist eine geordnete Menge von *Konstanten* entsprechend dem *Komponententyp*.
- Die *Länge* des Feldes wird *explizit* durch die Kardinalzahl der Menge festgelegt, der notwendige Speicher entsprechend bereitgestellt und mit den Mengenelementen als *Werte* belegt.

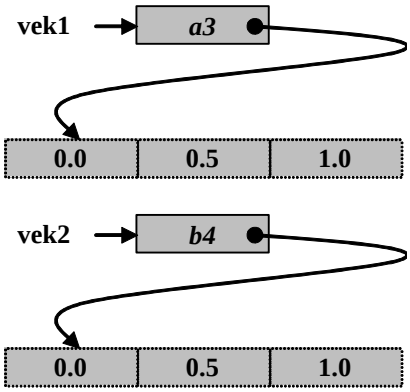
Mit der folgenden Initialisierung kann das obige Feld gleichermaßen vereinbart werden:

```
float[] vektor = { 0, .5f, 1};
```

5.1.2 Referenzen sind keine Felder

Beispiel 1

```
float[] vek1 = { 0, .5f, 1};
float[] vek2 = { 0, .5f, 1};

if( vek1 == vek2)   // false
```



Referenzen liefern bei einem *Vergleich* (==) i.R. den Wert `false`, da ihre *Adressen* verglichen werden!

[1] Da der Ausdruck `( i / 2.)` einen `double`-Wert liefert, muss dieser explizit in einen `float`-Wert umgewandelt werden.

Beispiel 2

```
float[] vek1 = { 0, .5f, 1};

vek1 *= 2;
// Fehler: operator * cannot be applied to int, float[]
```

***Komponentenweise* auszuführende Operationen sind *komponentenweise* aufzurufen.**

Beispiel 3

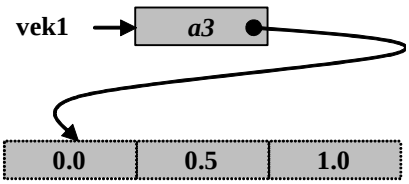
```
float[] vek1 = { 0, .5f, 1};
float[] vek2 = vek1;

for( int i = 0; i < vek1.length; i++) vek1[ i] *= 2;

for( int i = 0; i < vek1.length; i++)
  System.out.println( "1: " + vek1[ i] + ", 2: " + vek2[ i]);
```

Obwohl nur der Inhalt des Feldes `vek2` geändert wurde, erhält man als Ausgabe:

```
=>    1: 0, 2: 0
      1: 1, 2: 1
      1: 2, 2: 2
```

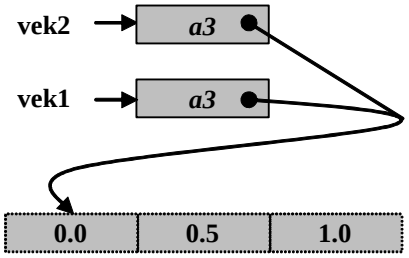
Verfolgt man die Verarbeitung der einzelnen Anweisungen, so kann man sich die Ursache der Ausgabe erklären:

1. `float[] vek1 = { 0, .5f, 1};`



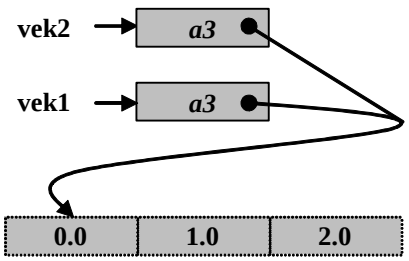
Dem Feld `vek1` wird ein Speicherbereich zugewiesen und mit den angegebenen Werten initialisiert.

2. `float[] vek2 = vek1;`



Da eine Referenz als Wert eine Adresse hat, wird dem Feld `vek2` die Adresse des Feldes `vek1` zugewiesen, d.h. beide Variablen verweisen auf *ein und dasselbe* Feld.

3.
```
for( int i = 0; i < vek1.length; i++)
    vek1[ i] *= 2;
```



Verändert man nun über eine der Feldreferenzen die Werte der Feldkomponenten, so wird diese Änderung auch von der anderen Feldreferenz aus sichtbar.

Der Zuweisungsoperator = zwischen Referenzdatentypen liefert zwar eine *Kopie einer Referenz*, aber *keine Kopie des Feldes*, auf welches die Referenz zeigt.

5.1.3 Kopieren eindimensionaler Felder

Wenn eine „*echte Kopie*“ erstellt werden soll, sind dafür zwei Schritte notwendig.

1. *Kopie der Datenstruktur* - Als erstes muss man dafür sorgen, dass Original und Kopie auf *unterschiedliche Felder gleicher Datenstruktur* (*gleicher Feldttyp, gleiche Länge*) weisen.

```
float[] vektor = { 0, .5f, 1};
float[] kopie = new float[ vektor.length];
```

2. *Kopie der Daten* - Dann muss man einzeln die Werte der Komponenten des einen Feldes in die Komponenten des anderen Feldes übertragen.

```
for( int i = 0; i < vektor.length; i++)
  kopie[ i] = vektor[ i];
```

Anschließend kann man die Werte eines der Felder manipulieren, ohne die des anderen zu zerstören.

```
for( int i = 0; i < kopie.length; i++) kopie[ i] *= 2;
```

5.1.4 Beispiel „Sortieren eines Vektors“

Beispiel

In einem Feld werden Werte eingegeben, sortiert und anschließend wieder ausgegeben.

Vektor.java (Grobstruktur)

```
public class Vektor
{
  public static void main( String[] args)
  {
    // Deklaration und Definition
    // Vektoreingabe
    // Sortieren mit SelectSort
    // Vektorausgabe
  }
}
```

Vektor.java

```
// Vektor.java                                          MM 2014
import Tools.IO.*;                                   // Eingaben

/**
 * Sortieren eines Vektors mit Hilfe SelectSort.
 */
public class Vektor
{
/**
 * Eingabe, Sortieren und Ausgabe eines Vektors.
 */
  public static void main( String[] args)
  {
    // Deklaration und Definition
    float[] vektor
    = new float[ IOTools.readInteger( "Anzahl Werte: ")];

    // Vektoreingabe
    for( int i = 0; i < vektor.length; i++)
      vektor[ i] = IOTools.readFloat( "v[" + i + "] = ");

    // Sortieren mit SelectSort
    for( int i = 0; i < vektor.length - 1; i++)
    {
      int k = i;                                 // Suche Minimum
      for( int j = i + 1; j < vektor.length; j++)
        if( vektor[ k] > vektor[ j]) k = j;

      if( i != k)                                  // Vertausche
      {
        float temp = vektor[ i];
        vektor[ i] = vektor[ k];
        vektor[ k] = temp;
      }
    }

    // Vektorausgabe
    for( int i = 0; i < vektor.length; i++)
      System.out.print( "  " + vektor[ i]);

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

5.2 Zweidimensionale Felder - Matrizen

5.2.1 Vereinbarung

Eine Matrix ist ein eindimensionales Feld, deren Komponenten eindimensionale Felder sind. **Deklaration, Definition und Initialisierung** werden als Feld von Feldern ausgeführt. Für die **Deklaration** und **Definition** einer Matrix ergibt sich damit:

Deklaration und Definition

```
int[][] matrix;
matrix = new int[ 4][ 3];
```

bzw.

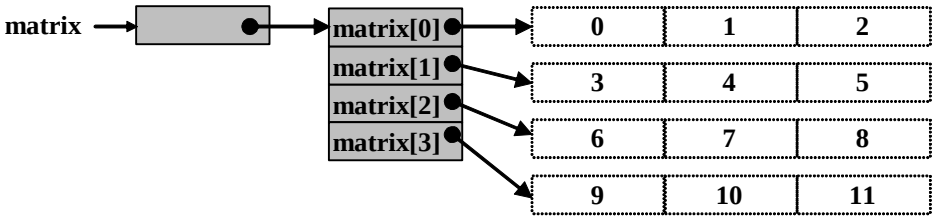
```
int[][] matrix = new int[ 4][ 3];
```

Zugriff auf die Feldkomponenten

```
matrix[ 0][ 0]      matrix[ 0][ 1]      matrix[ 0][ 2]
matrix[ 1][ 0]      matrix[ 1][ 1]      matrix[ 1][ 2]
matrix[ 2][ 0]      matrix[ 2][ 1]      matrix[ 2][ 2]
matrix[ 3][ 0]      matrix[ 3][ 1]      matrix[ 3][ 2]
```

Durchlaufen eines zweidimensionalen Feldes

```
int k = 0;
for( int z = 0; z < matrix.length; z++)
  for( int s = 0; s < matrix[ 0].length; s++)
    matrix[ z][ s] = k++;
```



Das Durchlaufen einer Matrix erfolgt zeilenweise und erfordert eine verschachtelte `for`-Schleife mit *zwei* Indexvariablen (hier: `z` Zeilenindex, `s` Spaltenindex). Dabei gibt `matrix.length` die *Anzahl der Zeilen* von `matrix` an. Um die *Anzahl der Spalten* je Zeile zu ermitteln, wird mit `matrix[ 0].length` die Länge der 0. Zeile von `matrix` zu Hilfe genommen.

Explizite Anfangswertzuweisung (Initialisierung)

Matrizen können durch eine Menge von Mengen bei Ihrer Deklaration initialisiert werden. Die Zeilen- und Spaltenanzahl richtet sich nach der Anzahl der Wertmengen und der Anzahl der Werte in der Wertmengen. Für die Werte wird der notwendige Speicher bereitgestellt.

```
int[][] matrix
= { { 0, 1, 2}, { 3, 4, 5}, { 6, 7, 8}, { 9 , 10, 11}};
```

Im Beispiel wurde die obige 4 x 3 - Matrix `matrix` gleichermaßen deklariert, definiert und mit Werten initialisiert.

Eine Matrix ist eine *Referenz auf ein Feld von Referenzen*.

5.2.2 Kopieren zweidimensionaler Felder

Überträgt man die Erfahrungen für den *eindimensionalen* Fall auf den *zweidimensionalen*, so sind die folgenden Anweisungen aus den oben erläuterten Gründen notwendig:

1. *Kopie der Datenstruktur* - Datentyp, Zeilen- und Spaltenanzahl werden übernommen:

```
int[][] matrix
  = { { 0, 1, 2}, { 3, 4, 5}, { 6, 7, 8}, { 9 , 10, 11}};
int[][] kopie
  = new int[ matrix.length][ matrix[ 0].length];
```

2. *Kopie der Daten* - Die Daten sind mit Hilfe einer zweifach verschachtelten `for` - Schleife zu übertragen:

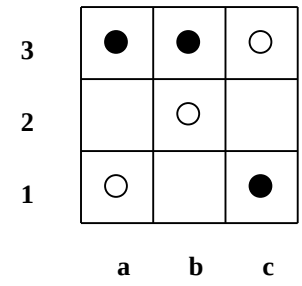
```
for( int z = 0; z < matrix.length; z++)
  for( int s = 0; s < matrix[ 0].length; s++)
    kopie[ z][ s] = matrix[ z][ s];
```

5.2.3 Beispiel „Tic Tac Toe“

Tic Tac Toe ist ein Spiel, welches auf einem Spielbrett mit *n* * *n* Karos und mit weißen und schwarzen Steinen gespielt wird. In der Ausgangsspielsituation sind alle Karos leer. Weiß und Schwarz besetzen abwechselnd ein leeres Karo, Weiß beginnt. Das Spiel wird in zwei Varianten gespielt:

1. Gewonnen hat der Spieler, dem es als erstem gelingt, *m* der eigenen Steine in eine waagerechte, senkrechte oder diagonale Reihe zu bringen.
2. Verloren hat der Spieler, der als erster *m* der eigenen Steine in eine waagerechte, senkrechte oder diagonale Reihe setzen muss.

Spielsituation auf einem 3*3 Brett, bei der 3 weiße Steine in die Diagonale gesetzt wurden:



Mit einem guten Partner wird das Spiel am 3*3 - Brett für beide Spielvarianten unentschieden enden.

Tic Tac Toe für ein 3*3 – Brett und der ersten Spielvariante:

TicTacToe.java (Grobstruktur)

```
public class TicTacToe
{
  public static void main( String[] args)
  {
```

```
      // Spielerlaeuterung

      // leeres Spielbrett
      // 2 Spieler

      // Spielstart
      boolean fertig;
      // Spielrunde
      do
      {
        // aktueller Spieler
        // Stein setzen
        // Spielbrett zeigen
        // Auswerten
          // Zeilentest
          // Spaltentest
          // positiver Diagonalentest
          // negativer Diagonalentest
        // Fertig
      } while( fertig);
  // Spielauswertung
    }
  }
```

TicTacToe.java

```
  // TicTacToe.java                                        MM 2014
  import Tools.IO.*;                                  // Eingaben

  /**
   * TicTacToe - Spiel fuer zwei Personen.
   */
  public class TicTacToe
  {
  /**
   * Spieler setzen abwechseln drei Steine,
   * gewonnen hat der Spieler, der seine Steine
   * in einer Zeile, Spalte oder Diagonalen hat.
   */
    public static void main( String[] args)
    {
      // Spielerlaeuterung
      System.out.println
      ( "Spieler setzen abwechseln drei Steine, ");
      System.out.println
      ( "gewonnen hat der Spieler, der seine Steine ");
      System.out.println
      ( "in einer Zeile, Spalte oder Diagonalen hat.");

      // leeres Spielbrett
      final int laenge = 3;
      char[][] brett = new char[ laenge][ laenge];
```

```
                                        // Spielbrett leeren
for( int z = 0; z < brett.length; z++)
  for( int s = 0; s < brett[ 0].length; s++)
    brett[ z][ s] = ' ';

// 2 Spieler
char[] spieler = { 'x', 'o'};

// Spielstart
int dran = 0;                          // aktueller Spieler
int runde = 0;                             // Rundenzaehler
boolean fertig = false;
boolean gewonnen = false;

// Spielrunde
do
{
  // aktueller Spieler
  dran = 1 - dran;
  runde++;

  System.out.println
  ( "Spieler " + spieler[ dran] + " ist dran!");
  System.out.println();

  // Stein setzen
  int z, s;
  do
  {
    do
    {
      z = IOTools.readInteger( "Zeile (0<=z<=2), z = ");
    } while( z < 0 || z > 2);
    do
    {
      s = IOTools.readInteger( "Spalte(0<=s<=2), s = ");
    } while( s < 0 || s > 2);
  } while( brett[ z][ s] != ' ');            // besetzt

  brett[ z][ s] = spieler[ dran];

  // Spielbrett zeigen
  System.out.println();
  for( z = 0; z < brett.length; z++)
  {
    System.out.print( " | ");
    for( s = 0; s < brett[ 0].length; s++)
      System.out.print( brett[ z][ s] + " | ");
    System.out.println();
  }
  System.out.println();
```

```
      // Auswerten
      for( z = 0; z < brett.length; z++)      // Zeilentest
        if( brett[ z][ 0] != ' ' &&
            brett[ z][ 0] == brett[ z][ 1] &&
            brett[ z][ 0] == brett[ z][ 2])
        {
           gewonnen = true;
           break;
        }

      if( !gewonnen)                          // Spaltentest
        for( s = 0; s < brett[ 0].length; s++)
          if( brett[ 0][ s] != ' ' &&
              brett[ 0][ s] == brett[ 1][ s] &&
              brett[ 0][ s] == brett[ 2][ s])
          {
             gewonnen = true;
             break;
          }

                                  // positiver Diagonalentest
      if( !gewonnen && brett[ 0][ 0] != ' ' &&
          brett[ 0][ 0] == brett[ 1][ 1] &&
          brett[ 0][ 0] == brett[ 2][ 2])
          gewonnen = true;

                                  // negativer Diagonalentest
      if( !gewonnen && brett[ 0][ 2] != ' ' &&
          brett[ 0][ 2] == brett[ 1][ 1] &&
          brett[ 0][ 2] == brett[ 2][ 0])
          gewonnen = true;

      // Fertig
      fertig = runde == laenge * laenge || gewonnen;
    } while( !fertig);

    // Spielauswertung
    if( gewonnen)
      System.out.println
      ( "Sieger: Spieler " + spieler[ dran]);
    else
      System.out.println( "Patt!");

    System.out.println( "Spiel beendet");
  }
}
```