

Inhalt

4	Einführung in die Programmiersprache Java (Teil II)	4-2
4.4	Strukturierte Programmierung.....	4-2
4.4.1	Strukturierung im Kleinen.....	4-2
4.4.2	Addierer (do-Schleife).....	4-3
4.4.3	Ein- Mal- Eins (for-Schleife, if-Anweisung)	4-4
4.4.4	Lineare Gleichung (if-else-Anweisung).....	4-5
4.4.5	Einfacher Rechner (switch-Anweisung)	4-7
4.5	Referenzdatentypen - Felder	4-10
4.5.1	Vektoren – eindimensionale Felder.....	4-11
4.5.2	Matrizen – zweidimensionale Felder	4-13
4.6	Referenzdatentypen - Klassen.....	4-18
4.6.1	Festlegen eines Strukturtyps – einer Klasse.....	4-18
4.6.2	Festlegen einer Struktur – eines Objekts einer Klasse.....	4-19

4 Einführung in die Programmiersprache Java (Teil II)

4.4 Strukturierte Programmierung

4.4.1 Strukturierung im Kleinen

EDSGER W. DIJKSTRA [1930-2002], niederländischer Informatiker, 1968:
„Go To Statement Considered Harmful“¹

D-Diagramm

1. Eine *einfache Aktion* ist ein **D-Diagramm**. **Anweisung**
2. Wenn *A* und *B* **D-Diagramme** sind und *c* eine *Bedingung* ist, so sind auch die folgenden Anweisungen **D-Diagramme**:

<i>A B</i>	Anweisungssequenz
<i>if c then A end</i>	Auswahanweisungen
<i>if c then A else B end</i>	
<i>while c do A end</i>	Schleifenanweisung
3. Nichts sonst ist ein **D-Diagramm**.

Java-Anweisungen:

Ausdrucksanweisungen
zusammengesetzte Anweisungen
Schleifenanweisungen
Auswahanweisungen
strukturbezogene Sprunganweisungen

Einfache Aktionen

Ausdrucksanweisungen *Ausdruck;*

Ablaufsteuerung

zusammengesetzte Anweisungen { *Anweisung Anweisung ... Anweisung* }

Auswahanweisungen

if(*Bedingung*) *Anweisung*
if(*Bedingung*) *Anweisung* else *Anweisung*

switch(*Ausdruck*)
{
 case *Konstante*: *Anweisung Anweisung ...*
 case *Konstante*: *Anweisung Anweisung ...*
 ...
 default: *Anweisung Anweisung ...*
}

Schleifenanweisungen

while(*Bedingung*) *Anweisung*
do *Anweisung* while(*Bedingung*);
for(*Ausdruck 1*; *Ausdruck 2*; *Ausdruck 3*) *Anweisung*

strukturbezogene Sprunganweisungen

continue;
break;
return;

¹ <http://www.acm.org/classics/oct95/>

4.4.2 Addierer (do-Schleife)

Wiederholtes Summieren von zwei Dezimalzahlen: Zunächst wird aus der Aufgabenstellung die *Grobstrukturierung* unter Verwendung des *EVA-Prinzips* entwickelt.

EinfacherAddierer.java (Grobstruktur)

```
public class EinfacherAddierer
{
    public static void main( String[] args)
    {
        char weiter;                // j oder n

        do
        {
            // Eingabe 1. Summand
            // Eingabe 2. Summand

            // Berechnung der Summe

            // Ausgabe Summe
            // Weiter
        } while( weiter == 'j');
    }
}
```

Ein Programm entsteht aus der Grobstruktur durch *schrittweise Verfeinerung*.

EinfacherAddierer.java

```
// EinfacherAddierer.java                MM 2008
import Tools.IO.*;                        // Eingaben

/**
 * Einfacher Addierer,
 * addiert beliebig oft zwei Dezimalzahlen.
 */
public class EinfacherAddierer
{
    /**
     * Eingabe der Summanden,
     * Berechnen und Ausgabe der Summe;
     * Abbruch auf Wunsch des Nutzers.
     */
    public static void main( String[] args)
    {
        char weiter;                // j oder n

        do
        {
            // Eingabe 1. Summand
            double summand1
            = IOTools.readDouble( "Summand1 = ");
```

```

    // Eingabe 2. Summand
    double summand2
    = IOTools.readDouble( "Summand2 = ");

    // Berechnung der Summe
    double summe = summand1 + summand2;

    // Ausgabe Summe
    System.out.println
    ( summand1 + " + " + summand2 + " = " + summe);

    // Weiter
    weiter = IOTools.readChar( "Weiter(j/n)? ");
} while( weiter == 'j');

System.out.println( "Programm beendet");
}
}

```

4.4.3 Ein- Mal- Eins (for-Schleife, if-Anweisung)

Das kleine Einmaleins mit strukturierter Ausgabe (Es wird berücksichtigt, dass es ein-, zwei- und dreistellige Produkte gibt.):

EinMalEins.java (Grobstruktur)

```

public class EinMalEins
{
    public static void main( String[] args)
    {
        // Zeile
        for( int i = 1; i <= 10; i++)
        {
            // Spalte in der Zeile
            for( int j = 1; j <= 10; j++)
            {
                // Abstand
                // Produkt
            }
            // Zeilenvorschub
        }
    }
}

```

EinMalEins.java

```
//EinMalEins.java
```

MM 2003

```

/**
 * Das kleine 1x1.
 */
public class EinMalEins

```

```

{
/**
 * Zeilenweise Ausgabe des kleinen 1x1.
 */
public static void main( String[] args)
{
    // Zeile
    for( int i = 1; i <= 10; i++)
    {
        // Spalte in der Zeile
        for( int j = 1; j <= 10; j++)
        {
            // Abstand
            if( i * j < 10) System.out.print( " ");
            if( i * j < 100) System.out.print( " ");

            // Produkt
            System.out.print( "  " + i * j);
        }

        // Zeilenvorschub
        System.out.println();
    }
}
}

```

EinMalEins.out

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

4.4.4 Lineare Gleichung (if-else-Anweisung)

Lösen der Gleichung $ax+b=0$:

LinGleichung.java (Grobstruktur)

```

public class LinGleichung
{
    public static void main( String args[])
    {
        // Einlesen der Parameter a und b

        // Berechnung und Ausgabe, Fallunterscheidung
        if( a == 0)

```

```

        if( b == 0)           // unendlich viele Loesungen
        else                 // keine Loesung
    else                     // eine Loesung
    }
}

```

LinGleichung.java

```

//LinGleichung.java
import Tools.IO.*;
// Eingaben

/**
 * Berechnen einer linearen Gleichung.
 */
public class LinGleichung
{
/**
 * Eingabe der Parameter, Berechnung und Ausgabe.
 */
    public static void main( String[] args)
    {
        // Eingabe der Parameter a und b
        System.out.println( "Loesung ax + b = 0");

        double a = IOTools.readDouble( "a = ");
        double b = IOTools.readDouble( "b = ");

        // Berechnung und Ausgabe, Fallunterscheidung
        if( a == 0)
        {
            if( b == 0)           // unendlich viele Loesungen
                System.out.println("L = R");
            else                 // keine Loesung
                System.out.println("L = {}");
            else                 // eine Loesung
                System.out.println("L = {" + (-b/a) + "}");
        }
    }

    /** ----- */
    // Testbeispiel 3.0x + 4.0 = 0
    // L = {-1.3333333333333333}

    // Testbeispiel 0.0x + 2.0 = 0
    // L = {}

    // Testbeispiel 3.0x + 0.0 = 0
    // L = {-0.0} !!!

    // Testbeispiel 0.0x + 0.0 = 0
    // L = R

```

4.4.5 Einfacher Rechner (switch-Anweisung)

Rechner führt Grundrechenarten +, -, * und / aus: Operanden und Operator werden eingegeben, Ergebnis wird ausgegeben. Division durch Null wird abgefangen.

EinfacherRechner.java (Grobstruktur)

```
public class EinfacherRechner
{
    public static void main( String[] args)
    {
        char weiter;                // j oder n
        do
        {
            // Eingabe Operator
            // Eingabe Operanden
            // Berechnen und Ausgabe des Ergebnisses
            switch (op)
            {
                case '+':                // Addition
                    break;
                case '-':                // Subtraktion
                    break;
                case '*':                // Multiplikation
                    break;
                case '/':                // Division
                    break;
                default:
                    System.out.println( "Fehlerhafte Eingabe");
            }
            // Weiter
        } while( weiter == 'j');
    }
}
```

Einfacher Rechner.java

```
// EinfacherRechner.java                MM 2008
import Tools.IO.*;                       // Eingaben

/**
 * Einfacher Rechner,
 * fuehrt Grundrechenarten aus.
 */
public class EinfacherRechner
{
    /**
     * Eingabe Operator und Operanden,
     * Berechnen und Ausgabe des Ergebnisses;
     * Abbruch auf Wunsch des Nutzers.
     */
    public static void main( String[] args)
    {
        char weiter;                // j oder n
```

```
do
{
    // Eingabe Operator
    char op
    = IOTools.readChar( "Operation (+, -, *, /) ");
    // Eingabe Operanden
    double operand1
    = IOTools.readDouble( "Operand1 = ");
    double operand2
    = IOTools.readDouble( "Operand2 = ");

    // Berechnen und Ausgabe Ergebnis
    double ergebnis;
    switch (op)
    {
        case '+': // Addition
            ergebnis = operand1 + operand2;
            System.out.print
            ( operand1 + " + " + operand2 + " = ");
            System.out.println( ergebnis);
            break;

        case '-': // Subtraktion
            ergebnis = operand1 - operand2;
            System.out.print
            ( operand1 + " - " + operand2 + " = ");
            System.out.println( ergebnis);
            break;

        case '*': // Multiplikation
            ergebnis = operand1 * operand2;
            System.out.print
            ( operand1 + " * " + operand2 + " = ");
            System.out.println( ergebnis);
            break;

        case '/': // Division
            if( operand2 != 0)
            {
                ergebnis = operand1 / operand2;
                System.out.print
                ( operand1 + " / " + operand2 + " = ");
                System.out.println( ergebnis);
            }
            else System.out.println( "Division durch 0");
            break;

        default:
            System.out.println( "Fehlerhafte Eingabe");
    }
}
```

```
        // Weiter
        weiter = IOTools.readChar( "Weiter(j/n)? ");
    } while( weiter == 'j');

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

4.5 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.

Java unterscheidet zwei Arten von Referenzdatentypen, **Felder** und **Klassen**. 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 Speicherinhalt verweist (Zeiger).

Definition Feld

Feld der Länge n =_{df} n -Tupel über eine Menge von Daten *gleichen* Typs

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

Arbeitsspeicher

Symbolische Adresse	Adresse im Speicher	Inhalt der Speicherzelle	Interpretationsvorschrift
	
b	5e	00	short (2 Byte)
	5f	6a	
	
r	65	a3	Referenz
	
	a3	00	
	a4	41	
	

Elementardatentyp

```
short b = 106;
```

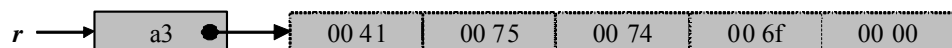


b ist eine Variable des *Elementardatentypen* short:

b hat im Speicher die Adresse (5e)₁₆, die Länge 2 Bytes und den Wert (00 6a)₁₆ = 106.

Referenzdatentyp

```
char[] r = "Auto";
```



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. Handelt es sich zum Beispiel um ein Feld mit 4 Komponenten vom Typ char, so bestehen die einzelnen *Feldkomponenten* jeweils aus 2 Byte und repräsentieren 4 Zeichen im Unicode, hier „Auto“. Der Speicherbereich, auf den die Referenz verweist, umfasst damit insgesamt 8 Byte.

4.5.1 Vektoren – eindimensionale Felder

Deklaration eines Vektors

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

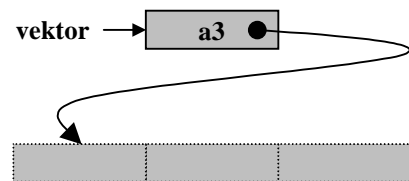
```
Komponententyp [ ] Feldname ;
float[] vektor;
```



Definition eines Vektors (eines neuen Vektors)

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

```
Feldname = new Komponententyp [ Länge ] ;
vektor = new float[ 3];
```



oder beides zusammen:

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

Zugriff auf Vektorkomponenten

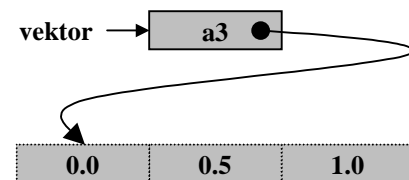
Auf einzelne Feldkomponenten wird über ein **Index** zugegriffen. Dieser ist ganzzahlig und muss in den Feldgrenzen liegen, d.h. $0 \leq \text{Index} < \text{Länge}$.

```
Feldname [ Index ]
vektor[ 0]      vektor[ 1]      vektor[ 2]
```

Durchlaufen eines Vektors

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

Die **Laufvariable** *i* heißt auch **Indexvariable**, weil *i* alle möglichen Indizes des Feldes *vektor* durchläuft.



Zu jedem Feld wird dessen *Länge* zusätzlichen im Speicher als ganzzahlige Variable mit dem Namen *length* abgelegt. Diese kann im Programm abgefragt 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 initialisiert werden. Die Länge des Feldes richtet sich nach der Anzahl der Werte in der Wertemenge. Für die Werte wird der notwendige Speicher bereit gestellt. Mit der folgenden Initialisierung kann der obige Vektor ebenfalls vereinbart werden.

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

Beispiel

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

Vektor.java (Grobstruktur)

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

Vektor.java

```
// Vektor.java                                MM 2009
import Tools.IO.*;                             // Eingaben

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

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

        // 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();
}
}

```

4.5.2 Matrizen – zweidimensionale Felder

Matrizen sind eindimensionale Felder, deren Komponenten eindimensionale Felder sind. **Deklaration, Definition und Initialisierung** werden analog den eindimensionalen Feldern ausgeführt. Für die **Deklaration** und **Definition** einer Matrix ergibt sich damit:

Deklaration einer Matrix

```
int[][] matrix;
```

Definition einer Matrix(einer neuen Matrix)

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

Zugriff auf Matrixkomponenten

```
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]
```

oder beides zusammen:

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

Durchlaufen einer Matrix

```
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

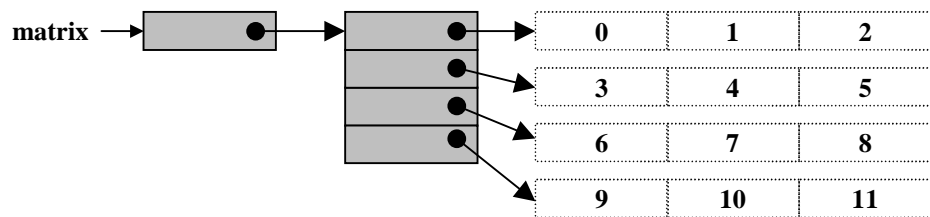
Zeile zu ermitteln, wird mit `matrix[0].length` die Länge der 0. Zeile von `matrix` zu Hilfe genommen.

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 bereit gestellt.

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

Im Beispiel wurde analog eine 4 x 3 - Matrix `matrix` deklariert, definiert und mit Werten initialisiert.



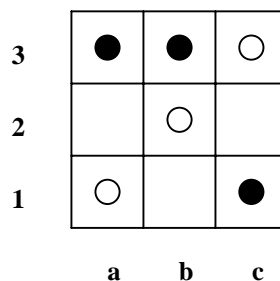
⇒ Eine Matrix ist eine Referenz auf ein Feld von Referenzen.

Beispiel

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
import Tools.IO.*;
// Eingaben

MM 2009

/**
 * 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] != ' ');

    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");
}
}
```

4.6 Referenzdatentypen - Klassen

Eigenschaften von Feldern

- Ein Feld fasst Daten zusammen, die für ein Programm eine Einheit bilden.
- **Alle Komponenten eines Feldes besitzen den gleichen Typ.**

Der letzte Punkt ist eine Einschränkung, unterschiedliche Typen sind oft wünschenswert. Im Gegensatz zu Feldern sind **Strukturen** Sammlungen von Daten *verschiedenen* Typs.

Definition Struktur

Struktur =_{df} **Tupel über eine Menge von Daten *verschiedenen* Typs**

**Strukturen sind strukturierte Datentypen, die für ein Programm eine Einheit bilden und Daten *verschiedenen* Typs zusammenfassen.
Die zusammengefassten Daten sind die *Strukturelemente*.**

Strukturen werden in *zwei* Schritten festgelegt: Zunächst wird ein **Strukturtyp** und anschließend eine **Struktur von diesem Typ** deklariert.

4.6.1 Festlegen eines Strukturtyps – einer Klasse

In Java verwendet man eine **Klasse** zur Modellierung eines neuen *Strukturtyps*. Die Strukturelemente werden als **Attribute** bezeichnet, weil in ihnen die Eigenschaften einer Struktur gespeichert werden.

Deklaration einer Klasse

```
public class Klassenname { Deklaration Deklaration... }
```

public legt fest, dass die Klasse eine öffentliche, eine für jeden zugreifbare Klasse ist. Die Deklarationen legen die in der Klasse zusammengefassten Datentypen fest.

Beispiel

Ein Datum besteht aus drei Werten, dem Tag, dem Monat und das Jahr. Man könnte es als `int` - Feld der Länge 3 auffassen. Dann muss man sich aber merken, dass die 0. Komponente der Tag, die erste Komponente der Monat und die 2. Komponente das Jahr darstellen. Als Klasse bezeichnet man die Attribute mit ihrem Namen, so dass die Daten leichter lesbar werden.

Datum.java

```
public class Datum
{
    int tag;
    int monat;
    int jahr;
}
```

Klasse Datum

Datum	
tag:	int
monat:	int
jahr:	int

Eine Klasse wird in einer *eigenen Datei* mit dem Namen der Klasse abgespeichert.

4.6.2 Festlegen einer Struktur – eines Objekts einer Klasse

Betrachten wir jetzt ein konkretes Datum, z. B. den Geburtstag von **Gottfried Wilhelm Leibniz** am 1. 7. 1646. Man benötigt eine Variable, mit der man das Datum verarbeiten kann. Solche Variablen zu einer Klasse nennt man auch **Objekte** dieser Klasse.

Analog den Feldern werden Objekte in zwei Schritten erzeugt:

Deklaration eines Objektes einer Klasse

Dem Compiler wird mitgeteilt, dass es sich um eine Referenzvariable handelt und Speicher für eine Adresse benötigt wird.

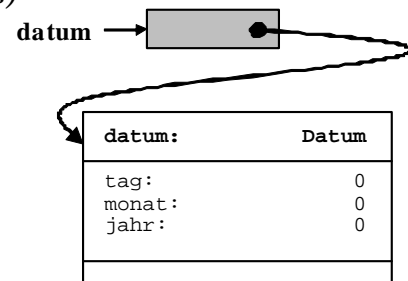
```
Klassenname Objektname ;  
Datum datum;
```



Definition eines Objektes einer Klasse (eines neuen Objektes)

Der notwendige Speicherplatz wird wiederum mittels dem **new**-Operator bereitgestellt.

```
Objektname = new Klassenname ();  
datum = new Datum();
```



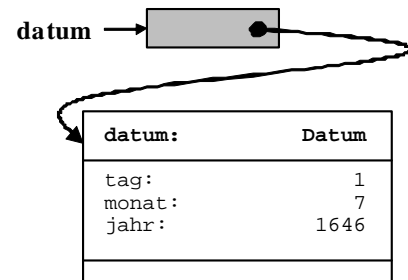
oder beides zusammen:

```
Datum datum; = new Datum();
```

Zugriff auf die Attribute

Der **Zugriff** auf die einzelnen Instanzvariablen erfolgt durch den **Punkt-Operator**.

```
Objektname . Attribut  
datum.tag = 1;  
datum.monat = 7;  
datum.jahr = 1646;
```



Objekt der Klasse Datum

Explizite Anfangswertzuweisung (Initialisierung)

```
Datum datum = { 1, 7, 1646 }
```

Beispiel

Ein Datum soll eingegeben werden, auf Korrektheit überprüft und wieder in einer üblichen Form ausgegeben werden. Dazu ist ein Objekt Datum zu erzeugen und anschließend dessen Attributen kontrolliert einzugeben und auszulesen.

DatumsEingabe.java (Grobstruktur)

```
public class DatumsEingabe  
{  
    public static void main( String[] args)  
    {  
        // Datum  
        // Datumseingabe
```

```

        // Jahr (1600 .. 3000)
        // Monat (1 .. 12)
        // Tag (1 .. Anzahl Tage im Monat)
        // Datumsausgabe
    }
}

```

DatumsEingabe.java

```

// DatumsEingabe.java
import Tools.IO.*;
import Tools.*;

/**
 * Eingabe eines Datums,
 * Gregorianischer Kalender gilt seit 1582.
 */
public class DatumsEingabe
{
    /**
     * Eingabe eines Datums,
     * Ueberpruefen der Korrektheit,
     * Ausgabe des Datum.
     */
    public static void main( String[] args)
    {
        System.out.print( "Datumseingabe");

        // Datum
        Datum datum = new Datum();

        // Datumseingabe
        do
        {
            datum.jahr =
                IOTools.readInteger( "Jahr (1600 .. 3000): ");
        } while( datum.jahr < 1600 || datum.jahr > 3000);

        do
        {
            datum.monat =
                IOTools.readInteger( "Monat (1 .. 12): ");
        } while( datum.monat < 1 || datum.monat > 12);

        boolean schaltJahr = false; // Schaltjahr
        int cc = datum.jahr / 100;
        int jj = datum.jahr % 100;
        if( jj == 0) schaltJahr = cc % 4 == 0;
        else schaltJahr = jj % 4 == 0;

        int anzahl = 0; // Anzahl der Tage im Monat
        switch( datum.monat)
        {

```

MM 2009
// Eingaben

```
case 1: case 3: case 5: case 7: case 8: case 10:
case 12: anzahl = 31; break;

case 4: case 6: case 9:
case 11: anzahl = 30; break;

case 2: if( schaltJahr) anzahl = 29;
        else anzahl = 28;
}

do                                     // Tag
{
    datum.tag =
    IOTools.readInteger( "Tag (1 .. " + anzahl + "): ");
} while( datum.tag < 1 || datum.tag > anzahl);

// Datumsausgabe
System.out.println
( datum.tag + "." + datum.monat + "." + datum.jahr);
}
}
```

Da man Datumseingaben immer überprüfen muss, wäre es günstig, die Datumsüberprüfung mit der Klasse `Datum` fest zu verbinden. Diese Möglichkeit ergibt sich mit einem neuen *Programmierparadigma*, der **objektorientierten Programmierung**.