

Inhalt

4	Anweisungen	4-2
4.1	<i>Strukturierte Programmierung</i>	4-2
4.1.1	Geschichte	4-2
4.1.2	Strukturierung im Kleinen	4-2
4.1.3	Der einarmige Bandit	4-4
4.2	<i>Ausdrucksanweisungen</i>	4-6
4.3	<i>Zusammengesetzte Anweisungen (Anweisungsblöcke)</i>	4-7
4.4	<i>Schleifenanweisungen</i>	4-8
4.4.1	while - Schleife	4-8
4.4.2	do - Schleife	4-8
4.4.3	for - Schleife	4-9
4.5	<i>Strukturbezogene Sprunganweisung</i>	4-11
4.6	<i>Auswahanweisungen</i>	4-13
4.6.1	if - Anweisung	4-13
4.6.2	switch-Anweisung	4-15

4 Anweisungen

4.1 Strukturierte Programmierung

4.1.1 Geschichte

In der Anfangszeit des Programmierens, bis in die sechziger Jahre des 20. Jh. hinein, gab es **keine Programmiermethodik**, keine Techniken und keine Regeln für das fachmännische Schreiben größerer Programme. Jeder Programmierer hatte vielmehr seine eigene Vorgehensweise.

Dabei wurden zu dieser Zeit bereits enorm große Programme geschrieben. Die **Komplexität** dieser großen Programme wuchs ihren Entwicklern über den Kopf. Ein durchaus üblicher Programm Quelltext von 100'000 Anweisungen ergibt einen Programtext von 2'000 Seiten, wenn man eine Anweisung pro Zeile und die Seite mit 50 Zeilen ansetzt. Dabei werden erläuternde Kommentare, die bei der Größe solch eines Programms unabdingbar sind, noch nicht einmal berücksichtigt. Die **Fehlerhäufigkeit** war groß. Bei der Beseitigung der gefundenen Fehler schlichen sich neue ein

⇒ **Chaos**.

Ende der sechziger Jahre fanden zwei von der **NATO** ausgerichtete Tagungen zu diesem Thema statt:

- 1968, „*Working Conference on Software Engineering*“ in Garmisch
- 1969, „*Working Conference on Software Engineering Techniques*“ in Rom

Auf ihnen wurden erstmalig die Probleme der Entwicklung großer Programme explizit benannt und diskutiert. Man stellte fest, dass Software das Ergebnis von **Ingenieurtätigkeit** ist und, wie jedes *industrielle Produkt*, der methodischen *Planung, Entwicklung, Herstellung* und *Wartung* bedarf. So entstand ein neues Wissenschaftsgebiet, welches heute unter dem Begriff **Software Engineering** bzw. **Softwaretechnik** läuft.

4.1.2 Strukturierung im Kleinen

Die Gestaltung der **Ablaufstruktur** der Algorithmen in der Programmierung lässt sich auf *zwei elementare Konstruktionen* zurückführen:

1. **Einfache Aktionen** (Wertzuweisungen, Prozeduraufrufe, ...)
2. **Binäre Verzweigungen zur Ablaufsteuerung** (bedingte Sprünge)

Viele Probleme bei der Programmierung waren auf die „Freiheit“ der Verwendung von Verzweigungen zurückzuführen. Deshalb war man bemüht, diese einzuschränken. **EDSGER W. DIJKSTRA** [1930-2002], ein niederländischer Informatiker, führte 1968 in einem Artikel „*Go To Statement Considered Harmful*“¹ aus: „*die Qualität eines Programms sei umgekehrt proportional zu der Anzahl der darin enthaltenen goto - Sprünge*“. Er schlug deshalb vor, dass man sich auf 5 Muster in den Sprachkonstrukten beschränken soll, die sogenannten **D-Diagramme**.

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

D-Diagramm

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

A B

if c then A end

if c then A else B end

while c do A end

Anweisungssequenz

Auswahanweisungen

Schleifenanweisung
3. Nichts sonst ist ein **D-Diagramm**.

Damit war der Anfang der modernen Softwareentwicklung markiert. Statt wilder Sprünge durch **goto**-Befehle sollten *if* und *while* den Programmfluss strukturieren. Die Beschränkung auf *D-Diagramme* wird als **goto-lose-Programmierung** bezeichnet. Fortan galten Programme mit **goto**, als sogenannte „Spaghetti-Programme“ *verwerflich*.

In den modernen Programmiersprachen findet man heute *mehrere Formen von Auswahl- und Schleifenanweisungen*. Hinzu sind auch noch einige *explizite strukturbezogene Sprunganweisungen* gekommen. Diese Erweiterung zur Ablaufsteuerung sieht man als **unbedenklich** an.

C- und 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;

4.1.3 Der einarmige Bandit

Spielregeln: Als Anfangskapital besitzt man 300 Euro. Der Einsatz wird festgelegt. Drei Zahlen werden gewürfelt. Sind alle drei Zahlen gleich, wird das Vierfache, sind zwei Zahlen gleich, das Doppelte des Einsatzes zurückgezahlt, sonst wird der Einsatz vom Kapital abgezogen.

Das Spiel endet, falls das Kapital aufgebraucht wurde bzw. falls es 500 Euro überschreitet.

Zunächst wird aus der Aufgabenstellung die Grobstrukturierung des Spiels entwickelt.

Bandit.java (Grobstruktur)

```
// Bandit.java MM 2005

import Tools.IO.*; // Ein- und Ausgaben

/**
 * // Spielerlaeuterung
 */
public class Bandit
{
    public static void main( String[] args)
    {
        int kapital = 300;

        do // Spielrunde
        {
            // Einsatzeingabe
            // Zug
            // Zugauswertung
        } while( kapital > 0 && kapital < 500);

        // Spielauswertung
    }
}
```

Das folgende Programm entstand durch schrittweise Verfeinerung.

Bandit.java

```
// Bandit.java MM 2005

import Tools.IO.*;

/**
 * Der Einarmige Bandit
 * Anfangskapital: 300 Euro
 * Sind alle drei Zahlen gleich, wird das Vierfache,
 * sind zwei Zahlen gleich, das Doppelte
 * des Einsatzes zurückgezahlt,
 * sonst wird der Einsatz vom Kapital abgezogen.
 * Das Spiel endet, falls das Kapital aufgebraucht
 * wurde bzw. falls es 500 Euro ueberschreitet.
 */
```

```
public class Bandit
{
    public static void main( String[] args)
    {
        int zahl1, zahl2, zahl3;
        int kapital = 300, einsatz, runden = 0, F;

        System.out.println( "Der Einarmige Bandit\n");

        do                                     // Spielrunde
        {
            runden++;

            do                                 // Einsatzeingabe
            {
                System.out.println
                ( "Dein Kapital: " + kapital + " Euro!");
                einsatz = IOTools.readInteger( "Dein Einsatz: ");
            } while( kapital < einsatz || einsatz <= 0);
            kapital -= einsatz;

            zahl1 = (int)( 9 * Math.random() + 1);    // Zug
            zahl2 = (int)( 9 * Math.random() + 1);
            zahl3 = (int)( 9 * Math.random() + 1);
            System.out.println
            ( "" + zahl1 + " " + zahl2 + " " + zahl3);

            F = 1;                                   // Zugauswertung
            if( zahl1 == zahl2) F++;
            if( zahl1 == zahl3) F++;
            if( zahl2 == zahl3) F++;
            if( F != 1) kapital += F * einsatz;
            /* System.out.println( "" + F); */

        } while( kapital > 0 && kapital < 500);

        if( kapital <= 0)                         // Spielauswertung
            System.out.println
            ( "" + runden + " Runden, Du bist Pleite!");

        if( kapital >= 500)
        {
            System.out.println
            ( "" + runden + " Runden, der Bandit ist Pleite!");
            System.out.println
            ( "Du hast ein Kapital von " + kapital + " Euro!");
        }
    }
}
```

4.2 Ausdrucksanweisungen

Syntax: `[Ausdruck] ;`

Das Semikolon `;` ist hier ein **Abschlusssymbol**, kein Trennzeichen.

Ausdrücke sind *void-Methodenaufrufe*, *Wertzuweisungen*, *Inkrement-* und *Dekrement-Ausdrücke*.

```
i + 1;
++ i;
```

nicht erlaubt
i wird inkrementiert

Leeranweisung: `;`

Kreis.java

```
//Kreis.java
```

MM 2003

```
import Tools.IO.*;

/**
 * Dieses Programm berechnet
 * Umfang und Flaeche eines Kreises,
 * Beispiel fuer Ausdrucksanweisungen.
 */
public class Kreis
{
    public static void main( String[] args)
    {
        double r = IOTools.readDouble( "Radius r = ");
        double umfang = 2 * Math.PI * r;
        double flaeche = Math.PI * r * r;

        System.out.println( "Kreisberechnung r = " + r);
        System.out.println
        ( "Umfang   = " + umfang + "\nFlaeche = " + flaeche);
    }
}

/*
    Kreisberechnung r = 2.0
    Umfang   = 12.566370614359172
    Flaeche  = 12.566370614359172
 */
```

4.3 Zusammengesetzte Anweisungen (Anweisungsblöcke)

Syntax: { [lokale Deklaration ...] [Anweisung ...] }

Deklarationen und Anweisungen können in beliebiger Reihenfolge angeordnet werden und werden hintereinander entsprechend dieser abgearbeitet. Variablen sind vor ihrem ersten Gebrauch zu deklarieren. Blöcke können *geschachtelt* werden.

Leeranweisung: {}

Unterschied zu C: Bereits deklarierte Variablen können in einem inneren Block nicht noch einmal deklariert werden.

Bloecke.java

```
//Bloecke.java                                     MM 2003

/**
 * Dieses Programm untersucht die Gueltigkeit
 * und die Verfuegbarkeit von Variablen in
 * verschachtelten Anweisungsbloecken.
 */
public class Bloecke
{
    public static void main( String[] args)
    {
        int x = 0;
        System.out.println( "0: x=" + x);           /* 0 */
        {
            // int y = 1; int x = 1;
            // Fehler, x ist bereits definiert

            int y = 1; x = 1;
            System.out.println( "1: x=" + x);         /* 1 */
            System.out.println( "1: y=" + y);         /* 1 */
            {
                int z = 2; x = 2;
                System.out.println( "2: x=" + x);     /* 2 */
                System.out.println( "1: y=" + y);     /* 1 */
                System.out.println( "2: z=" + z);     /* 2 */
            }
            System.out.println( "1: x=" + x);         /* 2 */
        }
        // System.out.println( "1: y=" + y);
        // Fehler, die Variable y ist hier nicht gültig

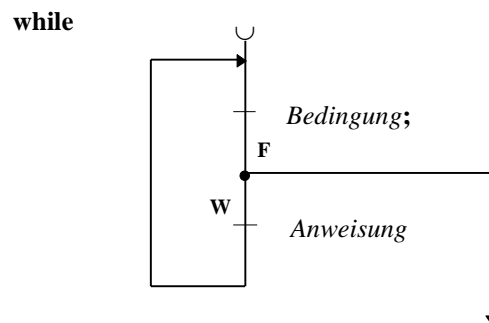
        System.out.println( "0: x=" + x);           /* 2 */
    }
}
```

4.4 Schleifenanweisungen

4.4.1 while - Schleife

Syntax: `while (Bedingung) Anweisung`

Die *while* Schleife ist eine **vorgeprüfte (pre-checked), abweisende Schleife**: Die *Bedingung* wird überprüft *bevor* die *Anweisung* ausgeführt wird. Solange die *Bedingung* wahr ist wird die *Anweisung* wiederholt.



```
final int MAX_VALUE = 10; int x = 0;
```

```
while( x < MAX_VALUE )
{
    System.out.println( "x = " + x + ", x * x = " + x * x );
    ++x;
}
```

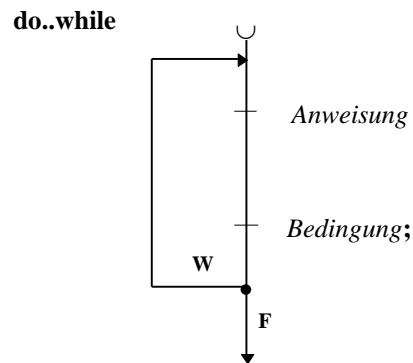
```
/*
    x = 0, x * x = 0
    x = 1, x * x = 1
    x = 2, x * x = 4
    x = 3, x * x = 9
    x = 4, x * x = 16
    x = 5, x * x = 25
    x = 6, x * x = 36
    x = 7, x * x = 49
    x = 8, x * x = 64
    x = 9, x * x = 81
*/
```

4.4.2 do - Schleife

Syntax: `do Anweisung while (Bedingung);`

Die *do* Schleife ist eine **nachgeprüfte (post-checked), nicht abweisende Schleife**: Die *Anweisung* wird ausgeführt *bevor* die *Bedingung* überprüft wird. Solange die *Bedingung* wahr ist wird die *Anweisung* wiederholt.

Das Semikolon am Ende der **do** - Schleife ist für die Syntaxanalyse notwendig!



```
final int MAX_VALUE = 10; int x = 0;

do
{
    System.out.println( "x = " + x + ", x * x = " + x * x );
    ++x;
} while( x < MAX_VALUE );
```

Zwischenergebnisse analog der **while**-Schleife.

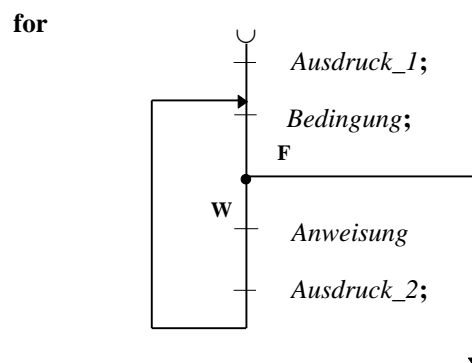
4.4.3 for - Schleife

Syntax: **for** ([*Ausdruck_1*] ; [*Bedingung*] ; [*Ausdruck_2*]) *Anweisung*

Das Semikolon fungiert hier als Trennzeichen!

Die **while** Schleife ist eine **vorgeprüfte (pre-checked), abweisende Schleife**.

1. Der *Ausdruck_1* wird ausgeführt.
2. Die *Bedingung* wird überprüft. Solange die *Bedingung* wahr ist, wird die *Anweisung* **und** anschließend der *Ausdruck_2* wiederholt.



Ausdruck_1, *Bedingung* und *Ausdruck_2* sind optional. Fehlt die *Bedingung*, so muss man explizit für den Abbruch sorgen!

```
final int MAX_VALUE = 10;

for( int x = 0; x < MAX_VALUE; x++)
    System.out.println( "x = " + x + ", x * x = " + x * x );
```

Zwischenergebnisse analog der *while*-Schleife.

Fakultätsberechnung s. Übungsaufgaben

Die Fakultät natürlicher Zahlen wird iterativ definiert:

$$0! = 1$$

$$n! = \prod_{k=1}^n k$$

private static long itFak(int n)

```
/**
 * Berechnung der Fakultät, 3 Varianten.
 */
private static long itFak( int n)
{
    long fak = 1;
    for( int i = 2; i <= n; i++) fak *= i;
    // oder auch: for( int i = 2; i <= n; fak *= i++);
    //           for( ; n > 1; fak *= n--);
    return fak;
}

/*
    20! = 2432902008176640000
    21! = -4249290049419214848
 */
```

Verschachtelte for-Schleifen

EinMalEins1.java

//EinMalEins1.java

MM 2003

```
/**
 * Dieses Programm erzeugt das kleine 1x1,
 * Beispiel fuer verschachtelte for-Schleifen.
 */
public class EinMalEins1
{
    public static void main( String[] args)
    {
        for( int i = 1; i <= 10; i++)
        {
            for( int j = 1; j <= 10; j++)
                System.out.print( " " + i * j);
            System.out.println( ""); // Zeilenvorschub
        }
    }
}
```

}

EinMalEins1.out

```

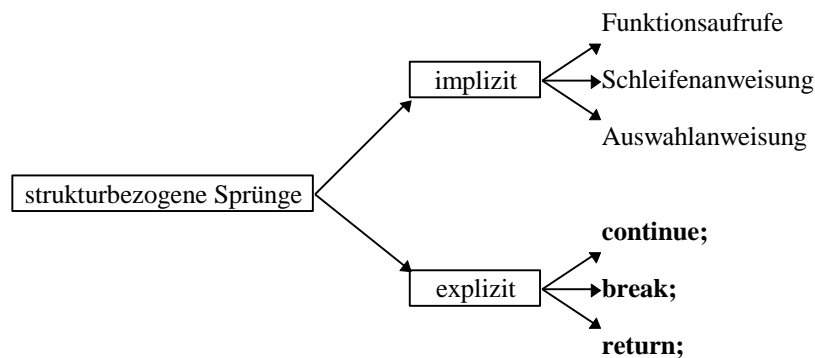
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100

```

4.5 Strukturbezogene Sprunganweisung

Durch die Strukturierungsmöglichkeiten einer Programmiersprache gibt es eine Vielzahl von Sprüngen, die beim Programmablauf ausgeführt werden. Das Verlassen einer Schleife erfolgt durch eine Bedingung. Eine Auswahlanweisung steuert das Durchlaufen eines Programmteils in Abhängigkeit einer Bedingung.

Es ist aber auch möglich, explizit ein Verlassen einer Anweisung zu erzwingen. Diese Vorgehensweise ist in den D-Diagrammen zwar nicht gestattet, wird aber in modernen Programmiersprachen trotzdem akzeptiert.

**Beenden einer Anweisung**

Syntax: `break;`

Die Abarbeitung der **Anweisung** wird abgebrochen.

Speziell für Schleifen kann ein Schleifendurchlauf abgebrochen werden.

Beenden eines Schleifendurchlaufs

Syntax: `continue;`

Die Abarbeitung des **Schleifendurchlaufs** wird abgebrochen, danach die Schleifenbedingung überprüft und entsprechend ihrem Wert die Schleife weiterbehandelt.

Beispiel *Formale Endlosschleifen**EndlosSchleifen.java*

//EndlosSchleifen.java

MM 2003

```
import Tools.IO.*;

/**
 * Dieses Programm summiert natuerliche Zahlen,
 * Beispiel fuer formale Endlosschleifen.
 */
public class EndlosSchleifen
{
    public static void main( String[] args)
    {
        int zahl = 0, summe = 0;
        do
        {
            zahl = IOTools.readInteger
                ( "Naechste Zahl (Abbruch mit -1): ");

            if( zahl < 0) break;           // Schleifenabbruch
                                   // Abbruch Schleifendurchlauf
            if( zahl == 0) continue;

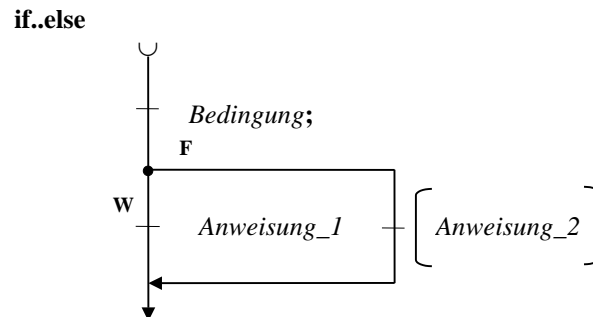
            summe += zahl;
            System.out.println( "Zahl = " + zahl);
            System.out.println( "Summe = " + summe);
        } while( true);           // Formale Endlosschleife
    }
}
```

4.6 Auswahlanweisungen

4.6.1 if - Anweisung

Syntax: **if** (*Bedingung*) *Anweisung_1* [**else** *Anweisung_2*]

1. Die *Bedingung* wird ausgewertet.
2. Ist ihr Wert wahr, so wird die *Anweisung_1* ausgeführt.
3. Ist ihr Wert falsch, so wird, falls vorhanden, die *Anweisung_2* ausgeführt.



Das kleine Einmaleins mit strukturierter Ausgabe:

EinMalEins2.java

```
//EinMalEins2.java
```

MM 2003

```

/**
 * Dieses Programm erzeugt das kleine 1x1,
 * Beispiel fuer verschachtelte for-Schleifen.
 */
public class EinMalEins2
{
    public static void main( String[] args)
    {
        for( int i = 1; i <= 10; i++)
        {
            for( int j = 1; j <= 10; j++)
            {
                // Abstand
                if( i * j < 10) System.out.print( " ");
                if( i * j < 100) System.out.print( " ");
                System.out.print( " " + i * j);
            }
            System.out.println( ""); // Zeilenvorschub
        }
    }
}

```

EinMalEins2.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

Verschachtelte If-Anweisung: Lösung der linearen Gleichung $ax+b=0$:***LinGleichung.java***

//LinGleichung.java

MM 2003

```
import Tools.IO.*;

/**
 * Dieses Programm berechnet lineare Gleichungen,
 * Beispiel fuer verschachtelte if-Anweisungen.
 */
public class LinGleichung
{
    public static void main( String[] args)
    {
        // Einlesen der Parameter a und b
        System.out.println( "Loesung ax + b = 0");
        double a = IOTools.readDouble( "a = ");
        double b = IOTools.readDouble( "b = ");

        if( a == 0)
            if( b == 0) System.out.println("L = R");
            else System.out.println("L = {}");
        else System.out.println("L = {" + -b/a + "}");
    }
}
```

Bei geschachtelten *if* - Anweisungen wird einem *else* stets das letzte vorhergehende „freie“ *if* zugeordnet. Evtl. sind Leeranweisungen erforderlich.

Auswahanweisung oder bedingte Anweisung ?

Die bedingte Ausdrucksanweisung

$$x = \text{Bedingung} ? \text{Ausdruck}_1 : \text{Ausdruck}_2 ;$$

ist folglich äquivalent mit der Auswahanweisung

$$\text{if} (\text{Bedingung}) x = \text{Ausdruck}_1 ; \text{else } x = \text{Ausdruck}_2 ;$$

4.6.2 switch-Anweisung

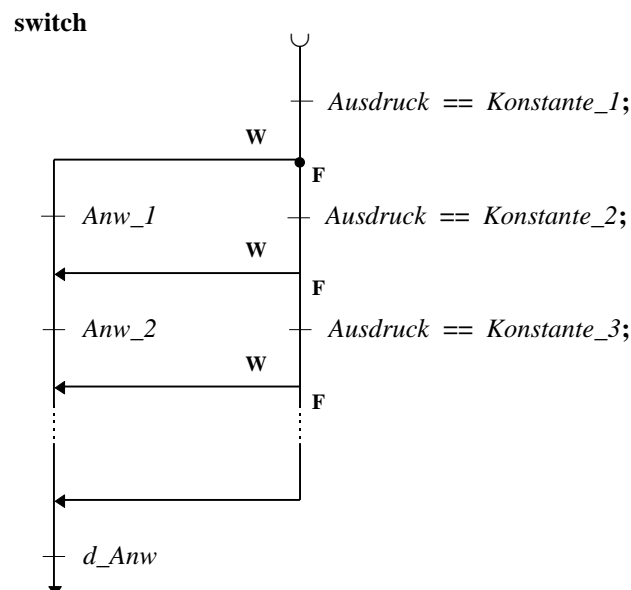
Syntax:

```
switch ( int_Ausdruck )
{
    case int_Konstante_1 : Anweisung_11 Anweisung_12 ...
    case int_Konstante_2 : Anweisung_21 Anweisung_22 ...
    ...
    [ default : d_Anweisung_1 d_Anweisung_2 ... ]
}
```

Der **default** - Fall kann an jeder Stelle in der Auswahlliste stehen.

1. Der *int_Ausdruck* wird berechnet und sein Wert wird mit den *int_Konstanten* verglichen.
2. Bei Gleichheit wird die Arbeit dort **fortgesetzt**. Bei Ungleichheit mit **allen** *int_Konstanten* wird, falls vorhanden, bei **default** fortgesetzt.

Ablaufdiagramm einer **switch** - Anweisung für den Sonderfall, das **default** am Ende der Auswahlliste steht:



Semantische Falle:

Entgegengesetzt zu anderen Programmiersprachen haben wir es hier nicht mit einer üblichen Fallunterscheidung zu tun, denn nach dem Einsetzen der Abarbeitung von Anweisungen werden alle darauffolgenden Anweisungen auch abgearbeitet. Um das zu verhindern, muss man mit **break**; die **switch** - Anweisung explizit abbrechen.

Natürlich kann man sich die hier verwendete Implementierung auch zunutze machen. op2 sei der Subtrahend einer Subtraktion, welche hier auf eine Addition zurückgeführt wird:

```
...    case Sub : op2 = - op2;
      case Add : ...
```

ZahlWort.java

//ZahlWort.java

MM 2003

```
import Tools.IO.*;

/**
 * Dieses Programm bildet Zahlwoerter,
 * Beispiel fuer Switch-Anweisungen.
 */
public class ZahlWort
{
    public static void main( String[] args)
    {
        int n = IOTools.readInteger
            ( "Eingabe Zahl zwischen 100 und 999: ");

        switch( n / 100)                                // Hunderter
        {
            case 1: System.out.print( "Ein"); break;
            case 2: System.out.print( "Zwei"); break;
            case 3: System.out.print( "Drei"); break;
            case 4: System.out.print( "Vier"); break;
            case 5: System.out.print( "Fuenf"); break;
            case 6: System.out.print( "Sechs"); break;
            case 7: System.out.print( "Sieben"); break;
            case 8: System.out.print( "Acht"); break;
            case 9: System.out.print( "Neun"); break;
            default:
                System.out.print( "Falsche Zahl\n");
                return;
        }
        System.out.print( "hundert");
        switch( n % 100)
        {
                                                    // Sonderfälle
            case 1: System.out.print( "eins"); break;
            case 11: System.out.print( "elf"); break;
            case 12: System.out.print( "zwoelf"); break;
            case 16: System.out.print( "sechzehn"); break;
            case 17: System.out.print( "siebzehn"); break;
            default:
                switch( n % 10)                    // Einer
                {
                    case 0: /* nichts */ break;
                    case 1: System.out.print( "ein"); break;
                    case 2: System.out.print( "zwei"); break;
                    case 3: System.out.print( "drei"); break;
                    case 4: System.out.print( "vier"); break;
                    case 5: System.out.print( "fuenf"); break;
                    case 6: System.out.print( "sechs"); break;
                    case 7: System.out.print( "sieben"); break;
                    case 8: System.out.print( "acht"); break;
                    case 9: System.out.print( "neun");
                }
            }
        }
    }
}
```



```
    }  
    // 13, 14, 15, 18, 19 ohne und  
    if( n % 100 / 10 > 1) System.out.print( "und");  
  
    switch( n % 100 / 10) // Zehner  
    {  
        case 0: /* nichts */ break;  
        case 1: System.out.print( "zehn"); break;  
        case 2: System.out.print( "zwanzig"); break;  
        case 3: System.out.print( "dreissig"); break;  
        case 4: System.out.print( "vierzig"); break;  
        case 5: System.out.print( "fuenfzig"); break;  
        case 6: System.out.print( "sechzig"); break;  
        case 7: System.out.print( "siebzig"); break;  
        case 8: System.out.print( "achtzig"); break;  
        case 9: System.out.print( "neunzig");  
    }  
    }  
    System.out.println( "");  
}  
}
```