

Wo sind wir?

- Java-Umgebung
- Lexikale Konventionen
- Datentypen
- **Kontrollstrukturen**
- Ausdrücke
- Klassen, Pakete, Schnittstellen
- JVM
- Exceptions
- Java Klassenbibliotheken
- Ein-/Ausgabe
- Collections
- Threads
- Applets, Sicherheit
- Grafik
- Beans
- Integrierte Entwicklungsumgebungen

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

84

Kontrollstrukturen

Die Abfolge der Ausführung eines Java-Programms wird durch Anweisungen (engl. statements) gesteuert, die **wegen ihrer Wirkung** ausgeführt werden und **keine Werte haben** (im Gegensatz zu den Ausdrücken im nächsten Kapitel).

Kontrollstrukturen in Java:

if-Anweisung
switch-Anweisung
for-Anweisung
while-Anweisung
do-Anweisung
break-Anweisung
continue-Anweisung
return-Anweisung

Leere Anweisung
Ausdrucksanweisung
(inkl. Methodenaufruf)

Block
Anweisung mit Label
Deklarationsanweisung

synchronized-Anweisung (*Threads*)
throw-Anweisung (*Exceptions*)
try-Anweisung (*Exceptions*)

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

85

Anweisung mit Label

Anweisung mit Label



Allen Anweisungen darf man ein **Label** (ein Bezeichner) voranstellen. Ein Label kann im **Zusammenhang mit einer break- oder continue-Anweisung** genutzt werden, um den Kontrollfluss zu ändern.

Einfaches Beispiel:

```
meinLabel: k = 5;
```

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

86

Block

Block



Ein Block ist syntaktisch gesehen (genau **eine Anweisung**). Innerhalb des Blocks können gemischt Deklarationen lokaler Variablen und Anweisungen stehen. Die deklarierten Variablen sind nur innerhalb des Blocks bekannt (Gültigkeitsbereich), genauer gesagt vom Deklarationspunkt bis zum Ende des Blocks.

Ein Block wird z.B. überall dort benötigt, wo syntaktisch eine Anweisung erforderlich ist aber mehrere Anweisungen zur Bewältigung der Aufgabe benötigt werden.

Wie man im Diagramm sieht, kann man **Blöcke ineinander schachteln!**

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

87

Beispiel

```
// gibt die Summe aller benachbarter Elemente in einem Feld aus
void summeNachbarn(int[] a) {
    for(int i=1; i < a.length; i=i+1) {
        int j; // lokale Variable dieses Blocks
        j = a[i-1] + a[i]; // 2 Anweisungen sind nötig
        System.out.println(j);
    }
}
```

Die lokale Variable *j* wird **mit jedem Eintritt** in den Block **neu angelegt** und **wird gelöscht** (existiert nicht mehr), wenn der **Block verlassen** wird. Die Variable *j* ist außerhalb des Blocks nicht bekannt.

In geschachtelten Blöcken sind die lokalen Variablen **äußerer Blöcke in den inneren Blöcken bekannt**, aber nicht umgekehrt.

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

88

Deklarationsanweisung

Deklarationsanweisung



Die Deklaration lokaler Variablen (siehe Kapitel Variablen) ist syntaktisch auch eine Anweisung.

Beispiel:

```
// Lineares Suchen in einem Feld
// (liefert Index des gefundenen Elementes bzw. -1)
int suchen(int[] a, int suchwert) {
    int i; // Deklarationsanweisung
    for(i=0; i<a.length; i=i+1)
        if(a[i] == suchwert)
            return i;
    return -1;
}
```

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

89

Leere Anweisung

Leere Anweisung

Die leere Anweisung macht dort Sinn, wo **syntaktisch** eine Anweisung erforderlich ist.

Beispiel:

```
int[] a = new int[10];
...
// Suche erstes Feldelement ungleich 0
for(int i=0; i < a.length; i++) {
    // Hier muss eine Anweisung kommen
    ;

    if (i < a.length)
        System.out.println("gefunden");
}
```

Rudolf Bernsdorf
FH Bonn-Rhein-Sieg Programmiersprache Java 90

Ausdrucksanweisung

Ausdrucksanweisung

Eine Ausdrucksanweisung macht Sinn, wenn der Ausdruck einen Seiteneffekt bewirkt, d.h. zum Beispiel eine Variable verändert (später mehr).

Beispiel:

```
// Die nächste Ausdrucksanweisung macht wenig Sinn, ist aber erlaubt
5 + 3;

/* Der nachfolgende Ausdruck hat als Seiteneffekt, dass der Wert der
 * Variablen a um eins erhöht wird.
 */
++a;

MeineKlasse o = new MeineKlasse();

// Aufruf der Methode kann z.B. eine Zustandsveränderung des Objekts bewirken
o.methode();
```

Rudolf Bernsdorf
FH Bonn-Rhein-Sieg Programmiersprache Java 91

if-Anweisung

if-Anweisung

Der Ausdruck wird ausgewertet. Ist der Wert `true`, wird die Anweisung nach) ausgeführt. Ist der Wert `false`, so wird die Anweisung nach dem `else` ausgeführt, sofern vorhanden.

Beispiel:

```
// liefert Index des ersten Feldelementes ungleich 0 bzw. -1, wenn alle 0
int ersteUngleichNull(int[] a) {
    int i;
    boolean gefunden = false;

    for(int i=0; !gefunden && (i < a.length); i=i+1)
        if( a[i] != 0)
            gefunden = true;

    if(gefunden)
        return i;
    else
        return -1;
}
```

Rudolf Bernsdorf
FH Bonn-Rhein-Sieg Programmiersprache Java 92

Bekanntes Problem I

Zu welchem if gehört das else?

Variante 1

```
if ( ausdruck_1 )
    if ( ausdruck_2 )
        anweisung_1;
else
    anweisung_2;
```

Variante 2

```
if ( ausdruck_1 )
    if ( ausdruck_2 )
        anweisung_1;
    else
        anweisung_2;
```

Beide Varianten sind nach dem Syntaxdiagramm möglich. In Programmiersprachen muss jedoch eindeutig geregelt werden, wie dies zu interpretieren ist. Java (wie alle anderen Programmiersprachen) definiert, dass **Variante 2** in diesem Fall genommen wird. Alternativ kann man dies durch Einführen eines Blocks auch im Programm eindeutig notieren.

Rudolf Bernsdorf
FH Bonn-Rhein-Sieg Programmiersprache Java 93

Bekanntes Problem II

Mehr als eine Anweisung im then-Fall oder else-Fall?

Fall 1 (so falsch)

```
if ( ausdruck )
    anweisung_1;
    anweisung_2;
else
    anweisung_3;
```

Fall 2 (so falsch)

```
if ( ausdruck )
    anweisung_1;
else
    anweisung_2;
    anweisung_3;
    anweisung_4;
```

Lösung: Verwendung eines Blocks.

```
if ( ausdruck ) {
    anweisung_1;
    anweisung_2;
}
else
    anweisung_3;
```

```
if ( ausdruck )
    anweisung_1;
else {
    anweisung_2;
    anweisung_3;
}
    anweisung_4;
```

Rudolf Bernsdorf
FH Bonn-Rhein-Sieg Programmiersprache Java 94

switch-Anweisung

switch-Anweisung

switch-Einträge

switch-Label

Rudolf Bernsdorf
FH Bonn-Rhein-Sieg Programmiersprache Java 95

Semantik

Zuerst wird der **switch-Ausdruck ausgewertet**. Danach wird dieser Wert der Reihe nach mit den **konstanten Ausdrücken verglichen**. Drei Fälle können eintreten:

- Der Wert **stimmt mit einem der konstanten Ausdrücke überein**. In diesem Fall werden die Anweisungen hinter dem entsprechenden Label bis zum Ende der switch-Anweisung (!) ausgeführt oder bis explizit der Kontrollfluss geändert wird (z.B. durch `break` oder `return`).
- Der Wert **stimmt mit keinem der Werte überein** und es existiert ein **default-Label**. Dann werden die Anweisungen hinter dem default-Label wie in Fall 1 beschrieben ausgeführt. Falls dort keine Anweisungen existieren, ist die switch-Anweisung beendet.
- Der Wert **stimmt mit keinem der Werte überein** und es existiert **kein default-Label**. Dann ist die gesamte switch-Anweisung beendet und die Ausführung fährt hinter der switch-Anweisung fort.

Anmerkung:
Üblicherweise beendet man die Behandlung eines Falls mit einer **break-Anweisung** (siehe Beispiel später).

Rudolf Bernhofer
FH Bonn-Rhein-Sieg

Programmiersprache Java

96

Einschränkungen

- Der switch-Ausdruck muss vom **Typ char, byte, short oder int** sein.
- Die konstanten Ausdrücke in den switch-Labels müssen **typ-kompatibel** zum switch-Ausdruck sein.
- Die konstanten Ausdrücke müssen **verschiedene Werte** haben.
- Höchstens **ein default-Label** ist zulässig.

Rudolf Bernhofer
FH Bonn-Rhein-Sieg

Programmiersprache Java

97

Beispiel 1

```

/* Bestimmt für kleine Zahlen, ob eine Zahl eine Primzahl ist.
   Rückgabe: 1=ist Primzahl, 0=ist keine Primzahl, -1: weiß nicht
*/
int primzahl(int x) {
    int resultat = -1;
    switch(x) {
        case 0:
        case 1:
        case 4:
        case 6:
        case 8:
        case 9: resultat = 0; // keine Primzahl
              break;

        case 2:
        case 3:
        case 5:
        case 7: resultat = 1; // Primzahl
              break;

        default: resultat = -1; // weiß nicht
                break; // nicht nötig, aber guter Stil
    }
    return resultat;
}

```

Rudolf Bernhofer
FH Bonn-Rhein-Sieg

Programmiersprache Java

98

Beispiel 2

```

void howMany1(int k) {
    switch(k) {
        case 1: System.out.println("one");
        case 2: System.out.println("two");
        case 3: System.out.println("many");
    }
}

void howMany2(int k) {
    switch(k) {
        case 1: System.out.println("one"); break;
        case 2: System.out.println("two"); break;
        case 3: System.out.println("many"); break;
    }
}

public static void main(String[] args) {
    howMany1(3); // Ausgabe: many
    howMany1(2); // Ausgabe: too many
    howMany1(1); // Ausgabe: one too many

    howMany2(1); // Ausgabe: one
    howMany2(2); // Ausgabe: two
    howMany2(3); // Ausgabe: many
}

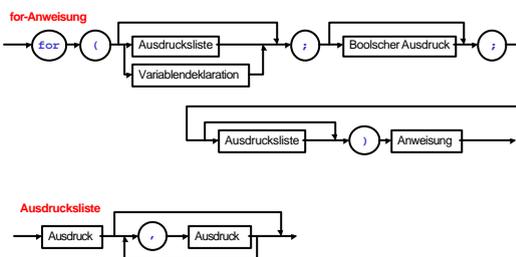
```

Rudolf Bernhofer
FH Bonn-Rhein-Sieg

Programmiersprache Java

99

for



Rudolf Bernhofer
FH Bonn-Rhein-Sieg

Programmiersprache Java

100

Semantik

- Zuerst wird der **Initialisierungscode einmalig ausgeführt** (hinter der öffnenden runden Klammer), in dem auch lokale Variable für die gesamte for-Anweisung (ab dem Deklarationspunkt) deklariert werden können.
- Zu **Beginn eines jeden Iterationsschritts** wird (falls vorhanden) der **boolesche Ausdruck ausgewertet**. Ist der **Ausdruck true oder nicht vorhanden**, so wird die Anweisung ausgeführt. Anschließend wird die **Ausdrucksliste** vor der schließenden runden Klammer (**Update-Teil**) von links nach rechts ausgewertet und mit dem nächsten Iterationsschritt (Schritt 2) begonnen. Ist der boolesche Ausdruck **false**, so wird die **Schleife verlassen**.

Existiert der boolesche Ausdruck nicht, so kann man die Schleife nur mit einer **break-Anweisung** verlassen.

Rudolf Bernhofer
FH Bonn-Rhein-Sieg

Programmiersprache Java

101

Beispiel 1

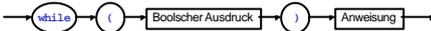
```
// berechnet die Summe aller Elemente eines Feldes
double feldSumme(double[] a) {
    double summe = 0.0;
    for(int i=0; i < a.length; i=i+1)
        summe = summe + a[i];
    return summe;
}
```

Beispiel

```
// berechnet die Summe aller Elemente eines zweidimensionalen Feldes
double feldSumme2(double[][] a) {
    double gesamtsumme = 0.0, vektorsumme;
    // Schleife über die erste Dimension
    for(int i=0; i < a.length; i=i+1) {
        // mehr als eine Anweisung ist im Schleifenrumpf, also ein Block
        double vektorsumme = 0.0;
        // und jetzt eine Schleife über die zweite Dimension
        for(int j=0; j < a[i].length; j=j+1)
            vektorsumme = vektorsumme + a[i][j];
        gesamtsumme = gesamtsumme + vektorsumme;
    }
}
```

while

while-Anweisung



Kopfgesteuerte Schleife:

Der boolesche Ausdruck wird ausgewertet. Ist der Wert `true`, so wird die Anweisung ausgeführt und die nächste Iteration mit der Auswertung des booleschen Ausdrucks begonnen. Ergibt der boolesche Ausdruck zu Beginn einer Iteration den Wert `false`, so bricht die Schleife ab.

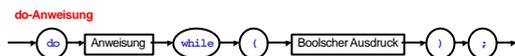
Beispiel 1

```
// berechnet die Wurzel einer Zahl nach dem Newton-Verfahren
double wurzel(double x, double epsilon) {
    double y, fehler;
    y = x; // Startwert
    if(x > y*y)
        fehler = x - y*y;
    else
        fehler = y*y - x;
    while(fehler > epsilon) {
        y = 0.5 * (y + x/y);
        if(x > y*y)
            fehler = x - y*y;
        else
            fehler = y*y - x;
    }
    return y;
}
```

Beispiel 2

```
// Endlosschleife
void herzschriftmacher() {
    while(true) {
        ermittle_werte();
        verarbeite_werte();
        korrigiere_steuerung();
    }
}
```

do



Fußgesteuerte Schleife:

Die Anweisung wird ausgeführt und anschließend der boolesche Ausdruck ausgewertet. Ist der Wert `true`, so wird die nächste Iteration begonnen, die mit der Ausführung der Anweisung wieder beginnt. Ergibt der boolesche Ausdruck am Ende einer Iteration den Wert `false`, so bricht die Schleife ab.

Soll mehr als eine "normale" Anweisung im Schleifenrumpf ausgeführt werden ausgeführt werden, so muss man diese in einen Block einschließen.

Beispiel

```
// berechnet die Wurzel einer Zahl nach dem Newton-Verfahren
double wurzel2(double x, double epsilon1) {
    double y, fehler;
    y = x; // Startwert
    do {
        y = 0.5 * (y + x/y);
        if(x > y*y)
            fehler = x - y*y;
        else
            fehler = y*y - x;
    } while (fehler > epsilon1);
    return y;
}
```

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

108

break

break-Anweisung



Das Ausführen einer break-Anweisung bewirkt ein **Verlassen des normalen Kontrollflusses**.

Eine **break-Anweisung ohne Bezeichner** ist nur im Rumpf einer **switch**-, **while**-, **do**- oder **for**-Anweisung erlaubt und bewirkt ein **Verlassen dieser umschließenden Anweisung**. D.h. zum Beispiel, dass die umschließende **for**-Anweisung sofort beendet wird und mit der Anweisung nach der **for**-Anweisung fortgefahren wird.

Wird **zusätzlich ein Bezeichner** angegeben, so wird die **innerste Anweisung** gesucht, die diesen Bezeichner als Label hat, und dorthin verzweigt. Dies kann jede beliebige Anweisung sein.

Häufig werden **break**-Anweisungen **im Zusammenhang mit switch-Anweisungen** eingesetzt (siehe 2. Beispiel bei **switch**).

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

109

Beispiel

```
/* Ein Wert wird in einem Feld gesucht und der Index des Feldelementes
zurückgegeben. Existiert der Wert nicht im Feld, wird die Feldlänge
als Resultat zurückgegeben.
*/
```

```
int suchen(int a[], int suchwert) {
    int i;
    for (i=0; i < a.length; i=i+1) {
        if(a[i] == suchwert)
            break; // for-Schleife wird sofort verlassen
    }
    // Hier geht es anschließend weiter
    return i;
}
```

Frage: Wieso darf man nicht schreiben

```
for (int i=0; i < a.length; i=i+1) ...
```

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

110

continue

continue-Anweisung



Das Ausführen einer **continue**-Anweisung bewirkt ein **Verlassen des normalen Kontrollflusses**. Eine **continue**-Anweisung ist **nur im Rumpf einer while**-, **do**- oder **for**-Anweisung erlaubt.

Eine **continue**-Anweisung ohne Bezeichner bewirkt, dass die **Kontrolle an den innersten Schleifenkopf übergeben** wird. Bei einer **for**-Schleife wird dann der **Update-Teil** gefolgt von der Überprüfung der **Iterationsbedingung** ausgeführt, bei einer **while**- oder **do**-Schleife wird die **Iterationsbedingung** überprüft.

Ist ein **Bezeichner** angegeben, so wird die Kontrolle an die **innerste Schleife** übergeben die diesen Bezeichner als Label hat.

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

111

Beispiel

```
// Die Summe der Logarithmen aller Feldelemente größer als 0 wird ermittelt.
double logSumme(double a[]) {
    double summe = 0.0;
    for (int i=0; i < a.length; i=i+1) {
        if(a[i] <= 0.0)
            continue; // Iteration wird sofort beendet
        // Aufruf Logarithmusfunktion (natürlicher Logarithmus)
        summe = summe + StrictMath.log(a[i]);
    }
    return summe;
}
```

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

112

return

return-Anweisung



Die **return**-Anweisung **beendet die Methode** und übergibt die Kontrolle an die aufrufende Methode zurück.

Ist ein **Ausdruck** angegeben, wird dieser zuerst ausgerechnet und dieser Wert als **Ergebniswert** zurückgegeben. Der Ausdruck muss **typkompatibel** zum angegebenen Resultattyp der Methodendeklaration sein.

Wird **kein Ausdruck** angegeben, so muss die Methode als **void** deklariert sein oder es muss ein **Konstruktor** sein.

Rudolf Berrndorf
FH Bonn-Rhein-Sieg

Programmiersprache Java

113

Beispiel

```
// Liefert die Summe aller Feldelemente.
double feldSumme(float a[]) {
    float summe = 0.0f;
    for (int i=0; i < a.length; i=i+1)
        summe = summe + a[i];
    /* Hier findet eine automatische Anpassung des Typs
       durch den Compiler statt. Die Methode muss einen double liefern,
       summe ist aber vom Typ float, was typkompatibel zu double ist.
    */
    return summe;
}
```