Anweisungen

- Ausdrucksanweisung
- Deklarationsanweisung
- Block
- Bedingte Anweisung
- Fallunterscheidung
- Schleifen
- Strukturierte Sprunganweisungen

- `throw`-Anweisung
- `try`-Anweisung
- `synchronized`-Anweisung

- `assert`-Anweisung

Anweisungen

Die Ausdrucksanweisung

Ausdrücke werden durch ein angehängtes Semikolon zur Anweisung. Das Semikolon beendet die Anweisung.

Nicht alle Ausdrücke können Anweisungen werden, denn die meisten Ausdrücke wie z. B. `x < y` würden allein keinen Sinn ergeben.

Nur die folgenden Arten von Ausdrücken können so als Anweisungen verwendet werden:

- Zuweisungsausdrücke (= und *op*=)
- Inkrement- und Dekrementausdrücke (++ und --)
- Methodenaufrufe
- Objekterzeugungsausdrücke (new)

Das Semikolon allein bildet auch schon eine Anweisung, und zwar die **leere Anweisung**. Diese bewirkt *nichts*.

Sie wird aus syntaktischen Gründen eingeführt:
Wenn an einer Stelle des Programms eine Anweisung stehen *muss*, dort aber nichts zu tun ist, wird die leere Anweisung verwendet.

Anweisungen

Block und Deklarationsanweisung

Keine oder mehrere Anweisungen können durch geschweifte Klammern ({ und }) zu einem **Block** zusammengefasst werden.

Ein Block kann immer dort verwendet werden, wo eine einzelne Anweisung erlaubt ist, denn der Block *ist* eine Anweisung, wenn auch eine zusammengesetzte.

In einem Block können lokale Variablen deklariert werden. Die Variablendeklaration wird auch als **Deklarationsanweisung** bezeichnet.

Lokale Variablen müssen vor ihrer Verwendung entweder bei der Deklaration oder durch eine Wertzuweisung initialisiert werden. Wird eine lokale Variable vor ihrer Initialisierung verwendet, gibt der Compiler eine Fehlermeldung aus.

Lokale Variablen können nur in dem Block verwendet werden, der ihre Deklaration enthält. Die Variablen, die in einem Block deklariert werden, müssen unterschiedliche Bezeichner haben. Sie können aber denselben Bezeichner haben wie Variablen, die in einem anderen Block deklariert sind.

Anweisungen

Der Block

Beispiel:

```
public class Block {
  public static void main(String[] argv) {
    int zahl = 12;
    { // ein neuer Block
      double zahl1 = 12.234;
      short  zahl2 = 127;
      System.out.println("zahl = " + zahl
        + ", zahl1 = " + zahl1
        + ", zahl2 = " + zahl2);
      // also int zahl, double zahl1 und short zahl2
    }
    System.out.println("zahl = " + zahl); // int zahl
    { // noch ein neuer Block
      float zahl1 = 321.45f;
      int   zahl2 = 815;
      System.out.println("zahl = " + zahl
        + ", zahl1 = " + zahl1
        + ", zahl2 = " + zahl2);
      // also int zahl, float zahl1 und int zahl2
    }
  }
}
```

Ausgabe:

```
zahl = 12, zahl1 = 12.234, zahl2 = 127
zahl = 12
zahl = 12, zahl1 = 321.45, zahl2 = 815
```

Anweisungen

Der Block

Lokale Variablen dürfen keine anderen lokalen Variablen oder Parameter überdecken (d.h. sie dürfen nicht denselben Namen haben wie andere lokale Variablen in umfassenden Blöcken oder wie Parameter).

Beispiel: (o.k.)

```
public class Block1
{
  public static void main(String[] argv)
  {
    int zahl = 12;

    {
      double zahl1 = 12.234;
      short  zahl2 = 127;
    }

    {
      float zahl1 = 321.45f;
      int   zahl2 = 815;
    }
  }
}
```

Anweisungen

Der Block

Beispiel: (fehlerhaft)

```
public class Block2 {
  public static void main(String[] argv) {

    int zahl = 12;

    {
      double zahl1 = 12.234;
      short  zahl2 = 127;

      boolean zahl = true;
```

Variable 'zahl' is already defined in this method.

```
      {
        String[] argv;
```

Variable 'argv' is already defined in this method.

```
        char zahl1 = '!';
```

Variable 'zahl1' is already defined in this method.

```
        byte zahl2 = -128;
```

Variable 'zahl2' is already defined in this method.

```
      }
    }

    {
      float zahl1 = 321.45f;
      int   zahl2 = 815;
    }
  }
}
```

Anweisungen

Die bedingte Anweisung

`if (` ***boolescher-Ausdruck*** **`)`**
  Anweisung$_1$
`else`
  Anweisung$_2$

Der Ausdruck wird ausgewertet; er muss einen Wert vom Typ `boolean` liefern.

Ist der Wert des Ausdrucks `true`, wird Anweisung$_1$ ausgeführt und der `else`-Teil übersprungen.

Ist der Wert des Ausdrucks `false`, wird Anweisung$_2$ ausgeführt; Anweisung$_1$ wird übersprungen.

Soll mehr als eine Anweisung ausgeführt werden, muss ein Block verwendet werden.

Der `else`-Teil kann auch fehlen. Dann bewirkt die bedingte Anweisung nichts, falls der Ausdruck `false` liefert.

Anweisungen

Die bedingte Anweisung

Beispiel:

```
import utilities.TastaturEingabe;

public class BedingteAnweisung
{
  public static void main(String[] argv)
  {
    int i = TastaturEingabe.readInt("i: ");
    if (i != 5)
      System.out.println("leider verloren");
    else
      System.out.println("hurra, gewonnen");
   }
}
```

Das Programm liefert etwa folgenden Dialog:

```
i: 7
leider verloren
```

Anweisungen

Die bedingte Anweisung

Beispiele:

```
if(ch >= 'A' && ch <= 'Z')
  ch += 32;

if(x > y)
  max = x;
else
  max = y;

if(sum > 100)
{
  System.out.println(sum);
  sum = 0;
}
```

Anmerkung: Blöcke, die nur *eine* Anweisung enthalten, werden selten verwendet:

```
if(i <= 5)
{
  sum += i;
}

if(i <= 5)
  sum += i;
```

Anweisungen

Die bedingte Anweisung

Anmerkung:

Durch **Einrücken** der zu einem Block gehörenden Anweisungen wird die Lesbarkeit eines Programms erhöht.

Einrücken entweder so:

```
if(Bedingung)
{                    // "{" unter dem "i" des "if"
  Anweisung;
  ...
}                    // "}" unter dem "{"
```

oder so:

```
if(Bedingung) {      // "{" hinter der Bedingung
  Anweisung;
  ...
}                    // "}" unter dem "i" des "if"
```

Ein abschreckendes Beispiel (C-Code):

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Anweisungen

Die bedingte Anweisung

Oft muss unter mehr als zwei Alternativen ausgewählt werden.

Eine Folge von Tests kann dadurch aufgebaut werden, dass dem `else`-Teil einer vorhergehenden `if`-Anweisung ein weiteres `if` hinzugefügt wird.

Beispiele:

```
if(punkte >= 90)
  note = 1.0;
else if(punkte >= 80)
  note = 2.0;
else if(punkte >= 70)
  note = 3.0;
else if(punkte >= 60)
  note = 4.0;
else
  note = 5.0;
```

Anweisungen

Die bedingte Anweisung

geschachtelte bedingte Anweisungen

```
if(Bedingung)
{
  if(Bedingung)     // Zweifach-Auswahl
    ...
  else
    ...
}
else
{
  ...
  if(Bedingung)     // Einfach-Auswahl
    ...
}
```

Anweisungen

Die bedingte Anweisung

geschachtelte bedingte Anweisungen

Zuordnung von `else` zu `if`

Beispiel: Einreichen eines Schecks bei einer Bank

```
double scheckBetrag, scheckGebuehr = 0.00;
double guthaben;
boolean studentenKonto;
```

Was wir ausdrücken wollen:

```
// Ist das Guthaben ausreichend?
if(scheckBetrag <= guthaben)
  // Nicht-Studenten zahlen 0,25 DM Gebuehr pro Scheck
  if(!studentenKonto)
    scheckGebuehr = 0.25;
// Konto ueberzogen: 20,- DM Ueberziehungsgebuehr
else
  scheckGebuehr = 20.00;
```

und wie der Compiler es interpretiert:

```
// Ist das Guthaben ausreichend?
if(scheckBetrag <= guthaben)
  // Nicht-Studenten zahlen 0,25 DM Gebuehr pro Scheck
  if(!studentenKonto)
    scheckGebuehr = 0.25;
  // Studenten zahlen 20,- DM Gebuehr pro Scheck
  else
    scheckGebuehr = 20.00;
```

Anweisungen

Die bedingte Anweisung

geschachtelte bedingte Anweisungen

Ein `else`-Teil ist immer an das letzte `if` gebunden, das keinen `else`-Teil hat.

Durch Verwendung eines Blocks wird die ursprüngliche Idee korrekt programmiert.

```
// Ist das Guthaben ausreichend?
if(scheckBetrag <= guthaben)
{
  // Nicht-Studenten zahlen 0,25 DM Gebuehr pro Scheck
  if(!studentenKonto)
    scheckGebuehr = 0.25;
}
// else wird ausgefuehrt bei Kontoueberziehung
else
  scheckGebuehr = 20.00;
```

Anweisungen

Die Fallunterscheidung

Soll unter mehreren Möglichkeiten ein Fall ausgewählt werden, kann man geschachtelte bedingte Anweisungen verwenden.

Beispiel:

Umrechnung der römischen Ziffern `I`, `V`, `X` und `L` in Dezimalwerte

```
char roemischeZiffer;
int dezimalWert;

if(roemischeZiffer == 'I')
  dezimalWert = 1;
else if(roemischeZiffer == 'V')
  dezimalWert = 5;
else if(roemischeZiffer == 'X')
  dezimalWert = 10;
else if(roemischeZiffer == 'L')
  dezimalWert = 50;
else
  System.out.println(
    "keine roemische Ziffer <= 50");
```

Die Fallunterscheidung

Bequemer und besser lesbar ist hier die **Fallunterscheidung**.

```
switch(roemischeZiffer)
{
  case 'I': dezimalWert = 1;
            break;
  case 'V': dezimalWert = 5;
            break;
  case 'X': dezimalWert = 10;
            break;
  case 'L': dezimalWert = 50;
            break;
  default:  System.out.println(
              "keine roemische Ziffer <= 50");
            break;
}
```

Anweisungen

Die Fallunterscheidung

```
switch ( Ausdruck )
{
  case KonstantenAusdruck1 : Anweisungen1
  case KonstantenAusdruck2 : Anweisungen2
    ...
  case KonstantenAusdruckn : Anweisungenn
  default : Anweisungenn+1
}
```

Der Ausdruck nach `switch` wird ausgewertet; er muss einen Wert vom Typ `char`, `byte`, `short` oder `int` liefern.

Ist das Ergebnis ein Wert, der hinter einem `case` steht, wird die Abarbeitung an dieser Stelle fortgesetzt.

Kommt der Wert nicht hinter einem `case` vor, so werden die Anweisungen nach `default` ausgeführt. Fehlt `default`, hat die Fallunterscheidung dann keine Wirkung.

Die Konstantenausdrücke in den `case`-Konstrukten müssen alle verschiedene Werte liefern, sonst meldet der Compiler einen Fehler.

Anweisungen

Die Fallunterscheidung

Beispiel: Olympische Spiele zwischen 1980 und 2002

```
switch(jahr)
{
  case 1980:
  case 1984:
  case 1988:
  case 1992: System.out.println(jahr +
               ": Sommer- und Winterspiele");
             break;
  case 1994:
  case 1998:
  case 2002: System.out.println(jahr +
               ": Winterspiele");
             break;
  case 1996:
  case 2000: System.out.println(jahr +
               ": Sommerspiele");
             break;
  default:   System.out.println(jahr +
               ": keine olympischen Spiele");
             break;
}
```

Anweisungen

Die Fallunterscheidung

Normalerweise wird die Anweisungsfolge hinter einem `case` mit dem Schlüsselwort **`break`** beendet.

Die Abarbeitung der Fallunterscheidung wird dann beendet und an die Anweisung hinter der schließenden geschweiften Klammer `}` der Fallunterscheidung verzweigt.

Fehlt `break`, werden die nächsten Anweisungen bis zu einem späteren `break` oder dem Ende der Fallunterscheidung der Reihe nach ausgeführt.

Beispiel: (fehlerhaft)

```
roemischeZiffer = 'I';

switch(roemischeZiffer)
{
  case 'I': dezimalWert = 1;
  case 'V': dezimalWert = 5;
  case 'X': dezimalWert = 10;
  case 'L': dezimalWert = 50;
  default:  System.out.println(
              "keine roemische Ziffer <= 50");
}
```

Ausgabe:

```
keine roemische Ziffer <= 50
```

Anweisungen

Schleifen

Schleifen sind Kontrollstrukturen, in denen Anweisungen wiederholt ausgeführt werden.

Java kennt die folgenden Schleifentypen:

- **`while`**-Schleife
- **`do-while`**-Schleife
- **`for`**-Schleife

Die `while`-Schleife

Bei einer `while`-Schleife wird jeweils *vor* der Schleife überprüft, ob sie zu wiederholen ist.

`while (` ***boolescher-Ausdruck*** **`)`**
Anweisung

Der Ausdruck wird ausgewertet; er muss einen Wert vom Typ `boolean` liefern.

Liefert er `true`, wird die Anweisung ausgeführt und der `while`-Ausdruck erneut ausgewertet.

Liefert er `false`, wird die Schleife abgebrochen und die Abarbeitung des Programms hinter der `while`-Schleife fortgesetzt.

Anweisungen

Die `while`-Schleife

Beispiel: Quadratwurzel einer Zahl berechnen

```
import utilities.TastaturEingabe;

public class Wurzel
{
  public static void main(String[] argv)
  {
    double zahl =
      TastaturEingabe.readDouble(
        "Gib Zahl ein: ");
    while (zahl < 0)
    {
      System.out.println(
        "Zahl darf nicht negativ sein!");
      zahl = TastaturEingabe.readDouble(
               "Neue Eingabe: ");
    }
    System.out.println("Wurzel aus " +
      zahl + " = " + Math.sqrt(zahl));
  }
}
```

Das Programm liefert etwa folgenden Dialog:

```
Gib Zahl ein: -43.567
Zahl darf nicht negativ sein!
Neue Eingabe: 43.567
Wurzel aus 43.567 = 6.60053028172737
```

Die `do-while`-Schleife

Bei einer `do-while`-Schleife wird jeweils *am Ende* der Schleife überprüft, ob sie zu wiederholen ist.

`do`
  Anweisung
`while ( `*`boolescher-Ausdruck`*` ) ;`

Nach der Abarbeitung der Anweisung wird der Ausdruck ausgewertet; er muss einen Wert vom Typ `boolean` liefern.

Liefert er `true`, wird die Anweisung wiederholt.

Liefert er `false`, wird die Schleife abgebrochen und die Abarbeitung des Programms hinter der `do-while`-Schleife fortgesetzt.

Anweisungen

Die `do-while`-Schleife

Beispiel: Menüsteuerung

```
import utilities.TastaturEingabe;

public class Menue
{
  public static void main(String[] argv)
  {
    char wahl;
    do
    {
      System.out.println("\n\nM E N U E \n\n");
      System.out.println("1: Hilfe");
      System.out.println("2: Datei drucken");
      System.out.println("3: Datei kopieren und drucken");
      System.out.println("E: Ende");
      wahl = TastaturEingabe.readChar("==>> ");
      switch(wahl) {
        case '1': System.out.println("HILFE");
                  break;
        case '3': System.out.print("KOPIEREN UND ");
                  // fall through
        case '2': System.out.println("DRUCKEN");
                  break;
        case 'e':
        case 'E': System.out.println("E N D E ! ! !");
                  break;
        default:  System.out.println("Falsche Eingabe!!");
                  break;
      }
      // warten auf <Return>:
      TastaturEingabe.warte();
    } while((wahl != 'E') && (wahl != 'e'));
  }
}
```

Die `for`-Schleife

Will man eine Schleife mit einem Zähler steuern, der von einem Anfangswert mit einer bestimmten Schrittweite bis zu einem Endwert läuft, wird die `for`-Schleife verwendet, die man oft auch *Zählschleife* nennt.

Eine Schleife der Form:

```
for ( Initialisierungs-Ausdruck ;
      boolescher-Ausdruck ;
      Inkrement-Ausdruck )
   Anweisung
```

ist gleichwertig zu folgendem Programmstück (solange nicht `continue` vorkommt):

```
Initialisierungs-Ausdruck ;
while ( boolescher-Ausdruck )
{
   Anweisung
   Inkrement-Ausdruck ;
}
```

Anweisungen

Die `for`-Schleife

Beispiel: Fakultät berechnen

```
public class Fakultaet
{
  public static void main(String[] argv)
  {
    for (int n=1; n<=20; n++)
    {
      long produkt = 1;

      for (int i=1; i<=n; i++)
        produkt *= i;

      System.out.println(n + "! = " + produkt);
    }
  }
}
```

Im Initialisierungsteil der `for`-Schleife kann eine Schleifenvariable deklariert werden; sie ist nur in der Schleife gültig.

Anweisungen

Die `for`-Schleife

Der Initialisierungs- und der Inkrement-Ausdruck einer `for`-Schleife können eine durch Komma getrennte Liste von Ausdrücken sein.

Die durch Komma getrennten Ausdrücke werden von links nach rechts ausgewertet.

Beispiel:

Die Reihenfolge der Feldelemente wird umgekehrt. (Aus `{1, 2, 3, 4, 5}` wird `{5, 4, 3, 2, 1}`.)

```
int[] feld = {1, 2, 3, 4, 5};
int temp, i, j;

for(i=0, j=feld.length-1; j>i; i++, j--)
{
  temp = feld[i];
  feld[i] = feld[j];
  feld[j] = temp;
}
```

Anweisungen

Die `for`-Schleife

Jeder der drei-Ausdrücke in einer `for`-Schleife kann auch fehlen; die Semikolons müssen aber trotzdem gesetzt werden.

Werden der *Initialisierungs-Ausdruck* oder der *Inkrement-Ausdruck* ausgelassen, bleibt ihr Platz in der Schleife leer.

Beispiel: zwei gleichwertige `for`-Schleifen

```
for (int i = 1; i <= n; i++)
  produkt *= i;
```

```
int i = 1;
for ( ; i <= n ; )
{
  produkt *= i;
  i++;
}
```

Fehlt der *boolesche Ausdruck*, so wird statt seiner `true` angenommen.

Beispiel: eine Endlosschleife

```
for(;;) { ... }
```

Anweisungen

Schleifen

Beispiele: Berechnung der Summe 1 + 2 + 3 + ... + n

```
int n = ... ;

int i = 1, sum = 0;
while (i <= n)
{
  sum += i;
  i++;
}

int i = 1, sum = 0;
do
{
  sum += i;
  i++;
} while (i <= n);

int i, sum = 0;
for (i = 1; i <= n; i++)
  sum += i;

int sum = n * (n+1) / 2;
```

Anweisungen

Strukturierte Sprunganweisungen

Anweisungen können mit **Marken** (engl. *label*) versehen werden. Marken werden üblicherweise bei Blöcken oder Schleifen verwendet. Eine Marke ist ein Bezeichner und geht einer Anweisung voran:

Marke **:** ***Anweisung***

In Java gibt es keinen unbedingten Sprung (**`goto`**). Stattdessen gibt es die folgenden strukturierten Sprunganweisungen:

- **`break`**
- **`continue`**
- **`return`**

Strukturierte Sprunganweisungen

`break`

Die **`break`**-Anweisung kann nur innerhalb von Schleifen und Fallunterscheidungen (`switch`) stehen.

Ein `break` ohne Marke beendet die innerste Schleife bzw. Fallunterscheidung, in der das `break` enthalten ist; d. h., es wird an die schließende geschweifte Klammer `}` dieser Kontrollstruktur verzweigt.

`continue`

Die **`continue`**-Anweisung kann nur innerhalb von Schleifen auftreten.

Ein `continue` ohne Marke beendet den aktuellen Schleifendurchlauf der innersten Schleife, in der das `continue` enthalten ist, und springt an die *Wiederholungsbedingung* der Schleife.

In einer `for`-Schleife springt `continue` zum *Inkrementausdruck*.

Anweisungen

Strukturierte Sprunganweisungen

Beispiel: Produkt eingelesener Zahlen berechnen

```
import utilities.TastaturEingabe;

public class Break_Continue
{
  public static void main(String[] argv)
  {
     long produkt = 1;
     int zahl, n;
     n = TastaturEingabe.readInt(
         "Wieviele Zahlen maximal einlesen? ");
     for (int i = 1; i <= n; i++)
     {
        zahl = TastaturEingabe.readInt(
                  "naechste Zahl " +
                  "(beenden mit -1): ");
        if (zahl == 0)      // ueberspringen
           continue;
        if (zahl == -1)     // Abbruch
           break;
        produkt *= zahl;
    }
    System.out.println("Produkt = " + produkt);
  }
}
```

Anweisungen

Strukturierte Sprunganweisungen

Schleifen können geschachtelt werden. Manchmal möchte man nicht die *innerste* Schleife verlassen, sondern irgendeine umgebende Anweisung. Dazu benötigt man bei `break` und `continue` Marken.

Die Anweisung

```
break Marke;
```

verzweigt an das *Ende* der Anweisung bzw. des Blockes, an dessen *Anfang* diese Marke steht.

Die Anweisung

```
continue Marke;
```

verzweigt an das *Ende* der Anweisung, an dessen *Anfang* diese Marke steht und startet für die Anweisung einen weiteren Schleifendurchlauf, falls die Schleifenbedingung dies zulässt.

Anweisungen

Strukturierte Sprunganweisungen

Beispiel: `continue`- und `break`-Anweisungen mit Marke

```
months:
  for (int m=1; m<=12; m++) {
    // mache irgendetwas
    // geschachtelte Schleife
    for (int d=1; d<=31; d++) {
      // irgendetwas tägliches
      if (m==2 && d==28)
        continue months;
      // sonst etwas anderes
    }
    // weitere Anweisungen
}
```

```
months:
  for (int m=1; m<=12; m++) {
    // mache irgendetwas
    // geschachtelte Schleife
    for (int d=1; d<=31; d++) {
      // irgendetwas tägliches
      if (cost > budget)
        break months;
      // sonst etwas anderes
    }
    // weitere Anweisungen
}
cost = 0;
```

Anweisungen

Strukturierte Sprunganweisungen

`return`

Die **`return`**-Anweisung beendet die Ausführung einer Methode und kehrt zum Aufrufer zurück. Wenn die Methode keinen Wert zurückgibt, wird die einfache Rückgabeanweisung verwendet:

```
return;
```

Verfügt die Methode über einen Rückgabewert, muss `return` einen Ausdruck eines Typs enthalten, der dem Rückgabetyp zugewiesen werden könnte:

```
return Ausdruck;
```

Beispiel:

```
double nichtNegativ(double wert)
{
  if(wert < 0)
    return 0;      // eine 'int'- Konstante
  else
    return wert;   // ein 'double'
}
```

Strukturierte Sprunganweisungen

`return`

Beispiel: Hexadezimalziffer nach `int` konvertieren

```
public int hexValue(char ch)
{
  switch(ch)
  {
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
      return ch - '0';

    case 'a': case 'b': case 'c':
    case 'd': case 'e': case 'f':
      return ch - 'a' + 10;

    case 'A': case 'B': case 'C':
    case 'D': case 'E': case 'F':
      return ch - 'A' + 10;

    default:
      System.out.println(ch +
        " ist keine Hexadezimalziffer.");
      System.exit(-1);
      return -1;
  }
}
```