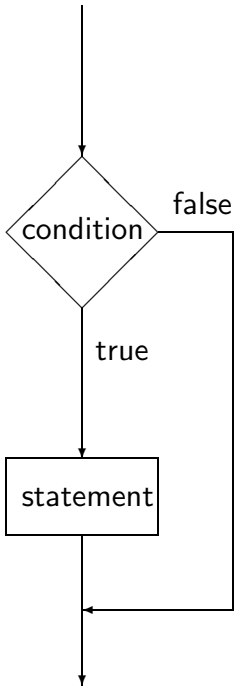
Wenn ... dann ...

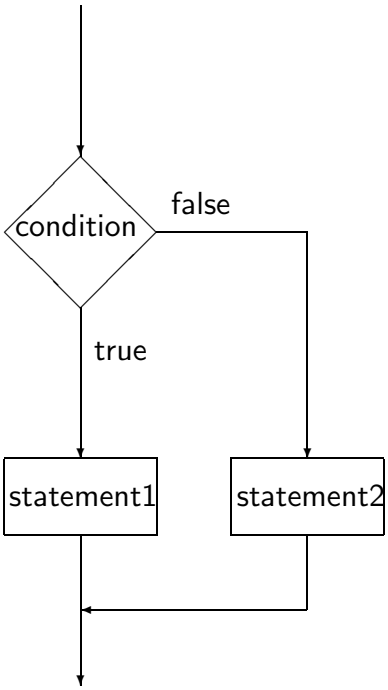
```
if (condition)
   statement
```




```
if (kontostand < 0)
  System.out.println("Oops ...");
```

... sonst ...

```
if (condition)
   statement1
else
   statement2
```




```
if (x < y)
  min = x;
else
  min = y;
```

ansonsten, falls ... I

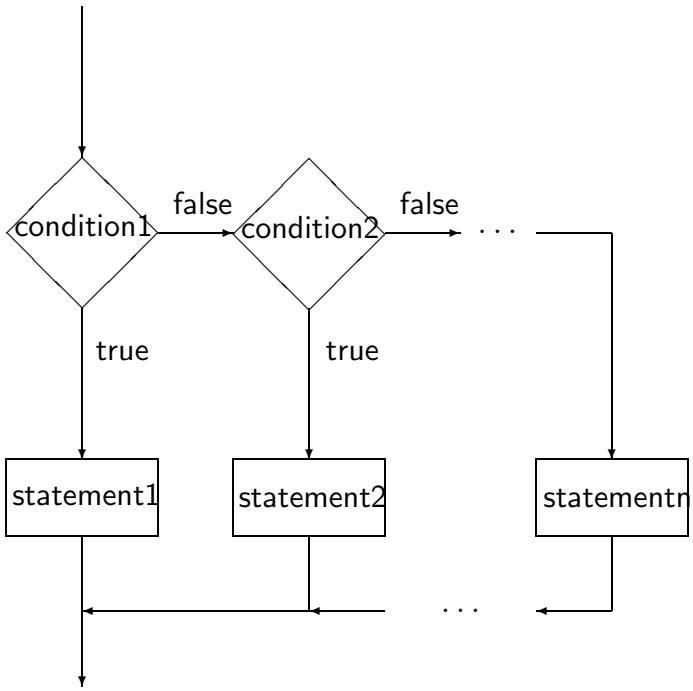
statt

```
if (condition1)
   statement1
else
   if (condition2)
      statement2
```

sieht man oft

```
if (condition1)
   statement1
else if (condition2)
   statement2
```

ansonsten, falls ... II




```
if (degreeCelsius < 8)
  System.out.println("zu kalt fuer Wein");
else if (degreeCelsius > 18)
  System.out.println("zu warm fuer Wein");
else if (degreeCelsius <= 12)
  System.out.println("OK fuer Weissen");
else if (degreeCelsius >= 15)
  System.out.println("OK fuer Roten");
else
  System.out.println("Weissherbst?");
```

Blöcke

Ein Block faßt eine Sequenz von Anweisungen zu einer zusammen:

{ *list of statements* }

Blöcke können geschachtelt werden.

```
if (increment) {
  ++x;
  ++y;
  System.out.println("new x: "+x);
  System.out.println("new y: "+y);
}
```

Einrückungsfalle

```
if (!handleY)
 if (incrementX)
   ++x;
else // Einrueckung irrefuehrend
  ++y;
```

Besser gleich klammern:

```
// jetzt richtig:
if (!handleY) {
 if (incrementX) {
   ++x;
 }
} else {
  ++y;
}
```

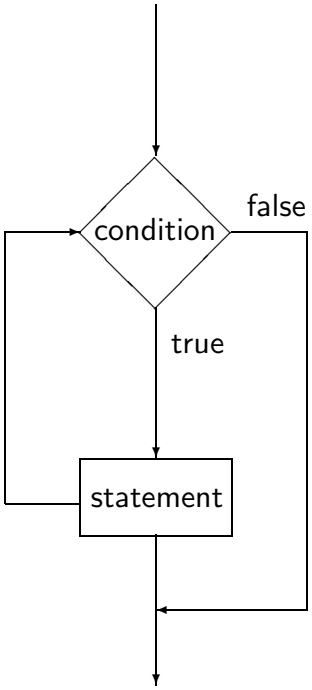
Sichtbarkeitsbereiche (scopes)

Ein Bezeichner ist ab seiner Deklaration sichtbar, und zwar bis zum Ende des Blocks in dem er vereinbart wurde.

```
int i = 0;
{
  int j = i;    // OK
  {
    int k = i; // OK
    int k = 0; // Fehler: k schon belegt
  }
  j = k;        // Fehler: k nicht bekannt
  int k = 0;    // OK: k neu belegen
}
++j;            // Fehler: j nicht bekannt
++i;            // OK
```

Solange I

```
while (condition)
    statement
```

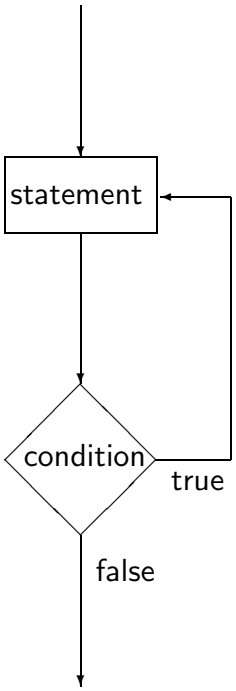


Solange II

```
// berechne ganzzahligen log von n
// zur Basis 2 fuer n > 0
int log = 0;
while (n > 1) {
  ++log;
  n /= 2;
}
System.out.println("log = "+log);
```

Mach ...Solange I

```
do
    statement
while (condition);
```

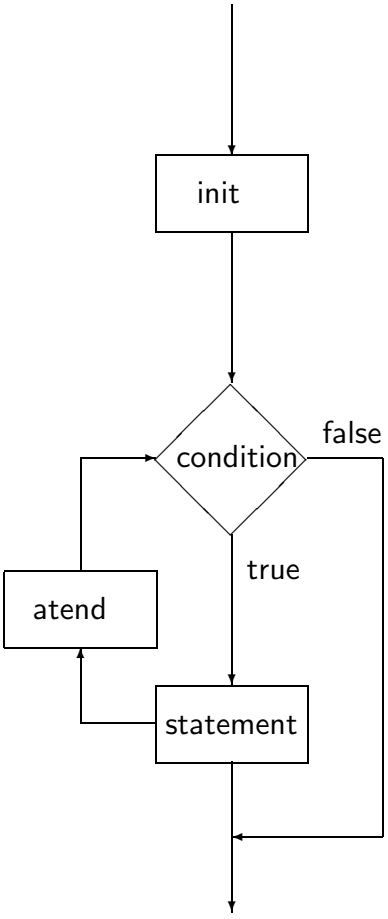


Mach ...Solange II

```
// berechne ganzzahligen log von n
// zur Basis 2 fuer n > 0
int log =-1;
do {
  ++log;
  n /= 2;
} while (n > 0);
System.out.println("log = "+log);
```

Laufanweisung (For-Schleife) I

`for` (*init*`;`*condition*`;`*atend*`)`
statement



init und *atend* sind Ausdrücke (für *init* auch Variablendeklaration für einen Typ erlaubt); falls mehrteilig durch `,`s getrennt

Laufanweisung (For-Schleife) II

```
// berechnet n! (n Fakultaet),
// d.h. 1*2*...*n fuer n>=0
int fakultaet = 1;
for (int i=0; i<n; ++i)
  fakultaet *= i;
System.out.println(n+"! = "+fakultaet);
```

Laufanweisung (For-Schleife) III

init, *condition* und *atend* können auch leer sein, die leere Bedingung (*condition*) gilt als konstant wahr

```
// unendliche Schleife:
for (;;) {
 // do something
}

// ein paar Zweierpotenzen
for (int i=0, j=1; i<=64; ++i, j*=2)
  System.out.println("2 hoch "+i+"="+j);
```

Auswahlanweisung I

```
switch (month) {
case 1: // ein sog. case-Label
  System.out.println("Januar");
  break;
case 2:
  System.out.println("Februar");
  break;
// usw. ...
case 12:
  System.out.println("Dezember");
  break;
default: // das default-Label
  System.out.println("Hey:");
  System.out.println("ungueltiger Monat!");
  break;
}
```

Auswahlanweisung II

- Auswahl über Ausdruck vom Typ `char`, `byte`, `short` oder `int`
- beliebige Reihenfolge der Label (auch `default`)
- kein Label darf doppelt vorkommen
- welche Label (z.B. ob `default`) vorkommen, ist beliebig
- ohne `break`: „*fallthrough* “
- letztes `break` optional

Auswahlanweisung III: Fallthrough

```
// fallthroughs:
for (int i=0; i<3; ++i) {
  System.out.println("i = "+i)
  switch (i) {
  case 0: System.out.println("case 0");
  case 1: System.out.println("case 1");
  case 2: System.out.println("case 2");
  }
}
```

ergibt:

```
i = 0
case 0
case 1
case 2
i = 1
case 1
case 2
i = 2
case 2
```

breaks in Schleifen

`break;`-Anweisung kann auch verwendet werden um Schleifen abzubrechen (`while`, `do` und `for`-Schleifen)

```
for (int i=0; i<n; ++i) {
  // irgendwas
  if (alreadyTooLate)
    break;
  // weitermachen
}
```

Bei geschachtelten Schleifen (oder einem `switch` innerhalb einer Schleife) wird immer das innerste Konstrukt beendet

Sprung aus der Tiefe

labeled breaks werden bei geschachtelten Schleifen (bzw.`switch`) verwendet, um das entsprechende äußere Konstrukt zu beenden

```
  for (int i=0; i<n; ++i) {
middlefor: // label fuer 2. for
    for (int j=0; j<n; ++j) {
      for (int k=0; k<n; ++k) {
        // ...
        if (jumpAway)
          break middlefor;
        // ...
      }
   }
  // hierhin 'breaken' wir
}
```

Weitermachen!

`continue;` wird verwendet, um wieder an den Anfang der Schleife zu springen

```
// Schleierfahndung ...
for (int id=0; id<n; ++id) {
  // berechne fuer id
  // boolean currIsSuspect
  if (!currIsSuspect)
    continue; // check next
  // mal id genauer ansehen
  // ...
}
```

`continue;` kann wie `break;` in der Variante mit *Label* verwendet werden

weitere Sprunganweisung (`return`): später

Felder (Arrays) deklarieren

*Array*s sind *n*-Tupel eines Datentyps, d.h. die Werte werden geordnet im Array abgelegt. Der Zugriff erfolgt über Indizes.

Deklaration eines (eindimensionalen) Array vom Typ *type* (Größe/Inhalt noch nicht festgelegt!):

type `[]` *identifier*;

alternativ auch die C-Schreibweise:

type identifier `[]`;

```
int[] a; // noch nicht initialisiert!
int b[]; // wie in C
```

Arrays anlegen

neues Array mit *n* Elementen wird angelegt durch `new` *identifier*`[]`, die Elemente werden dabei mit 0 (bzw. `false` oder `null`) initialisiert.

Bsp.:

```
int[] a1;
a1 = new int[5]; // int array mit 5 Elementen
// boolean array mit 8192 Elementen:
boolean[] a2 = new boolean[8*1024];
```

Arrays initialisieren

explizite Initialisierung durch { *valuelist* }

Bsp.:

```
String[] dayOfWeek =
  {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};

// nicht moeglich fuer Zuweisung
int[] daysPerMonth;
daysPerMonth = // Compilerfehler
  {31, 28, 31, 30, 31, 30,
   31, 31, 30, 31, 30, 31 };
// aber:
daysPerMonth = // OK
  new int[]{31, 28, 31, 30, 31, 30,
            31, 31, 30, 31, 30, 31 };
```

Arrays: Elementzugriff

Zugriff auf das (i+1)-te Element des Arrays a: `a[i]`

Elementanzahl von Array a: `a.length`, Datentyp der Länge ist `int`

Bsp.:

```
// alle Elemente von a ausgeben:
for (int i=0; i<a.length; ++i)
  System.out.println(a[i]);
// setzen der Elemente von a:
for (int i=0; i<a.length; ++i)
  a[i] = i+1;
```

Mehrdimensionale Arrays

```
double[][] matrix = new double[3][3];
int[][][] voxel = new int[100][100][100];
boolean[][] field =
 { {true, false}, {false, true}}
```

mehrdimensionale Arrays sind Arrays von Arrays; entsprechend brauchen sie nicht „rechteckig“ in ihrer Größe sein

```
int[][] triangle = new int[100][];
for (int i=0; i<triangle.length; ++i)
  triangle[i] = new int[i];

int[][] a =
{ {}, {1, 2, 3}, {9, 27} };

// Ausgabe:
for (int i=0; i<a.length; ++i)
  for (int j=0; j<a[i].length; ++j)
    System.out.println(a[i][j]);
```

Test For Echo

der Stringarray aus `main` enthaehlt die Kommandozeilenargumente

```
// (fast) wie das Echo-Programm von Unix
public class Echo {}
  public static void main(String[] argv) {
    for (int i=0; i<argv.length; ++i)
      System.out.print(argv[i]+" ");
    System.out.println();
  }
}
```

wird das Programm mit

```
java Echo test for echo
```

aufgerufen, so steht in `argv` der Wert

```
{"test", "for", "echo" }
```

Ausgabe des Programms:

```
test for echo
```

Mal was Einlesen ...

`Integer.parseInt(`*string*`)` wandelt den String *string* in einen `int` um,

`Double.parseDouble(`*string*`)` wandelt den String *string* in einen `double`-Wert um

```
public class Add {}
  public static void main(String[] argv) {
    int sum = 0;
    for (int i=0; i<argv.length; ++i)
      sum += Integer.parseInt(argv[i]);
    System.out.println("sum is "+sum);
  }
}
```