IV. Programmierung im Kleinen – Steuerung des Programmablaufs

1. Einführung – Wiederholungen, Verzweigungen und Blöcke
2. Wiederholungsanweisungen
3. Auswahlanweisungen

1. Einführung – Wiederholungen, Verzweigungen und Blöcke

- Schleifen – Wiederholte Ausführung von Anweisungen
- Programm-Verzweigungen
- Graphische Darstellungen von Programmstrukturen

Schleifen – Wiederholte Ausführung von Anweisungen

Motivation

- Bisher: Jede Anweisung wird genau einmal ausgeführt
- Weiterführung: Für komplexere Probleme sind sehr viele Anweisungen notwendig;
  - häufig müssen Anweisungsfolgen mehrfach durchlaufen werden,
  - häufig ist es nicht möglich, im Voraus zu bestimmen wie viele Anweisungen dies sind, da die Anzahl beispielsweise von der Eingabe des Benutzers abhängt
- Beispiele:
  - Ein Benutzer soll solange nach einer korrekten Eingabe gefragt werden, bis er die richtige eingibt
  - Ein Benutzer gibt eine Zahl `n` ein; berechne a_n für
    - $a_1 = 3$ (Initialisierung) und
    - mit der Vorschrift (iterativ) $a_{i+1} = (4 * a_i + 5*a_i^2) \% 13579$

Wiederholungsanweisungen (Schleifen)

- **Schleifen** führen Anweisungen wiederholt aus (Wiederholungsanweisung)
- Eine wiederholte Ausführung einer Schleife wird als Iteration bezeichnet
- In Java gibt es verschiedene Arten von Schleifen

  Die drei gebräuchlichsten Schleifenformate sind:

  - **`while`**-Schleife
  - **`do-while`**-Schleife
  - **`for`**-Schleife
- Es gibt noch ein weiteres Schleifenformat in Java (ab Java 5.0):
  - **`for`**-each-Schleife – erweiterte **`for`**-Schleife für Zählschleifen mit Aufzählungstypen (z.B. **`enum`**) und Arrays (Kapitel V)

Die grundlegende `while`-Schleife

- Eine **`while`**-Schleife führt eine Anweisung so lange aus, wie eine bestimmte Bedingung gültig ist
  - die Anweisung ist eine Einzelanweisung oder
  - ein Block bestehend aus mehreren Anweisungen

```
while (<boolean-expression>)
   <statement>
```

```
while (<boolean-expression>) {
   <statements>
}
```

- Semantik der Anweisung: Wenn das Programm zu einer **`while`**-Anweisung gelangt, wird die Bedingung (boolescher Ausdruck, `<boolean-expression>`) ausgewertet:
  - ist der Wert **`false`**, dann springt das Programm über den Rest der **`while`**-Schleife und fährt mit der nächsten Anweisung fort,
  - ist der Wert **`true`**, das wird die Anweisung (`<statement>`) bzw. der Block der Schleife (**`{`**`<statements>`**`}`**) ausgeführt; das Programm kehrt anschließend an den Anfang zurück und wiederholt den Prozess – bis die Bedingung **`false`** wird

Ein einfaches Beispiel

- Eine **`while`**-Schleife druckt die Ziffern 1 2 3 4 5

Beispiel – Zinsberechnung (Demo: Interest3.java)

```
1  public class Interest3 {
2      /**
3       * Klasse implementiert ein einfaches Programm zur Berechnung der Bankzinsen
4       * (interest), der sich für ein (interaktiv) gegebenes Grundkapital (principal)
5       * über 5 Jahre berechnet. Der jeweils am Ende eines Jahres verfügbare neue Betrag
6       * der Gesamtkapitals wird ausgegeben. Am Ende wird berechnet, wieviel Prozent
7       * Zuwachs es gegeben hat. Es werden vorgestellt:
8       * - Wiederholungsschleifen (while)
9       * - Bloecke (und Definition lokaler Variablen innerhalb von Bloecken)
10      * - Festlegung von Konstanten (final)
11      * Autor: David J. Eck (mit kleineren Ergänzungen hn, Aug.2010)
12      **/
13     public static void main(String[] args) {
14         double principal,  // Wert der Einlage (Grundkapital)
15                rate,       // jaehrliche Zinsrate
16                base,       // urspruenglicher Wert (nach Eingabe gespeichert)
17                increase;   // Zuwachs (in Prozent)
18         int years;         // zaehlt die Anzahl der Jahre
19         final int MAX_YEARS = 5;  // Berechnung fuer 5 Jahre (Konstante)
20
21         /* -- Eingabe ... */
22         TextIO.put("Eingabe des Investionsbetrags: ");
23         principal = TextIO.getlnDouble();
24         TextIO.put("Eingabe der jaehrlichen Zinsrate (Dezimal, nicht %): ");
25         rate      = TextIO.getlnDouble();
26
27         base = principal;  // Speichern des anfaenglichen Grundkapitals
28
29         /* -- berechne das Kapital am Ende eines Jahres (ueber mehrere Jahre) ... */
30         years = 0;
```

Fortsetzung …

Beispiel (Fortsetzung)

```
29          /* -- berechne das Kapital am Ende eines Jahres (ueber mehrere Jahre) ... */
30          years = 0;
31          while ( years < MAX_YEARS ) {
32              double interest;  // Zinsbetrag fuer das aktuelle Jahr
33
34              interest  = principal * rate;      // berechne den Zins
35              principal = principal + interest;  // berechne den neuen Wert des Kapitals
36
37              years = years + 1;  // zaehle das aktuelle Jahr ...
38
39              /* -- Ausgabe der Jahresabrechnung ... */
40              System.out.print("Der Wert des Kapitals (in EUR) nach " + years +
41                               " Jahren ist ");
42              System.out.printf("%1.2f", principal);
43              System.out.println();
44              }
45
46          /* -- um wieviel Prozent ist das Kapital gewachsen? */
47          if (base != 0.0) {
48              increase = 100.0 * (principal - base) / base;
49              System.out.println("Der Zuwachs des Kapitals betraegt " + increase + " %");
50              }
51          else
52              System.out.println("kein Grundkapital ...");
53      } // end main
54  } // end class Interest3
```

Hinweis: Es kommt bei Programmen mit Wiederholungsanweisungen immer wieder vor, dass aufgrund von Programmfehlern, Endlosschleifen auftreten! Der entsprechende Prozess eines endlos laufenden Programms kann (in Java) mittels **ctrl+C** (Strg+C) abgebrochen werden

Programm-Verzweigungen

Motivation

- Bisher: Folgen von Anweisungen werden sequenziell ausgeführt (linearer Programmablauf)
- Weiterführung: Für komplexere Probleme muss häufig entschieden werden, welcher Programmabschnitt ausgeführt werden soll –
  - die Verzweigung kann einfach sein, d.h. eine oder mehrere Anweisungen werden ausgeführt oder nicht,
  - die Entscheidung kann eine Alternative sein, oder
  - eine Mehrfachauswahl sein
- Beispiele:
  - Wenn der Wert einer Variablen größer ist als der einer zweiten, dann führe eine bestimmte Anweisung aus (die Anweisung kann auch mehrere Operationen umfassen)
  - Wenn es Montag oder Donnerstag ist, dann gehe zur PI-Vorlesung, sonst lies das Skript und bereite die Übungen vor

Die grundlegende `if`-Anweisung

- Auswahlanweisungen führen zur Trennung des Programmflusses in alternative Pfade unterschiedlicher Anweisungen
- Eine `if`-Anweisung (bedingte Anweisung) hat die folgende Form:
  - Entscheidung zwischen zwei Alternativen

```
if (<boolean-expression>)
   <statement>
else
   <statement>
```

  - Häufig wird entschieden, ob bei Erfüllung der Bedingung ein Kommando ausgeführt wird oder nichts gemacht wird (einfache Verzweigung)

```
if (<boolean-expression>)
   <statement>
```

- Die gleichen Formate für Anweisungen bestehend aus **Blöcken**:
  - Entscheidung zwischen zwei Alternativen

```
if (<boolean-expression>) {
   <statements>
}
else {
   <statements>
}
```

  - Entsprechend für die Entscheidung mit nur einem Pfad
    (der Pfad besteht aus einem Block mit mehreren Anweisungen)

```
if (<boolean-expression>) {
   <statements>
}
```

- Es gibt noch weitere Auswahlanweisungen in Java, bei denen zwischen mehreren Alternativen ausgewählt wird
  - **`switch`**-Anweisungen (so genannte bewachte Anweisungen)
  - **`? :`**-Operator zur kompakten (Operator-) Notation von **`if-else`** Anweisungen

Ein einfaches Beispiel

- Die Werte zweier Variablen, x und y, werden vertauscht, wenn der Wert von x größer ist als y

```
if ( x > y ) {
   int temp;      // temporaere Variable als Zwischenablage

   temp = x;      // speichere Kopie von x
   x    = y;      // kopiere y nach x
   y    = temp;   // kopiere temp nach y
}
```

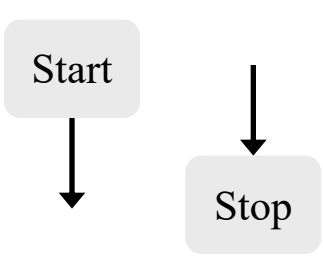
- Erläuterungen zum Ablauf und der Ausführung:
  - Die Bedingung (logischer Ausdruck) wird ausgewertet
  - Ergibt die Bedingung `x > y` den Wahrheitswert **`true`**, so wird der Block in **`{...}`** ausgeführt
  - Funktion des Blocks: Vertauschung der Inhalte von `x` und y
  - Dafür wird eine Zwischenvariable `temp` benötigt, die Variable wird lokal deklariert und hat nur eine Gültigkeit während der Ausführung des Blocks innerhalb der **`if`**-Anweisung (wenn die Bedingung **`true`** ergibt)
  - Ergibt die Bedingung `x > y` den Wahrheitswert **`false`**, so wird nichts ausgeführt

Graphische Darstellungen von Programmstrukturen

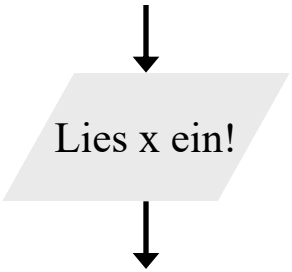
Flussdiagramme

Strukturelemente

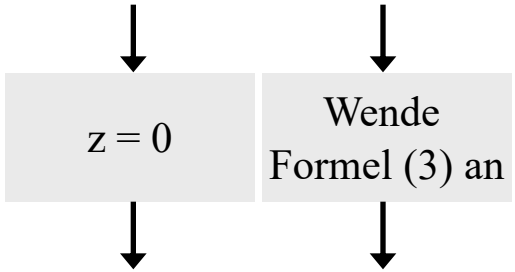
Flussdiagramme (auch: **Ablaufpläne**; nach DIN 66001) sind eine Methode zur informellen (d.h. Programmiersprachen-unabhängigen) Beschreibung von Algorithmen



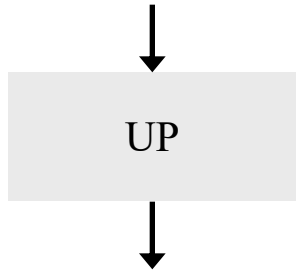
Anfang und Ende



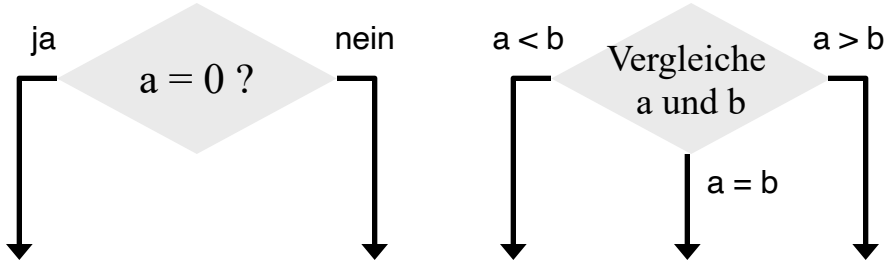
Ein-/Ausgabe



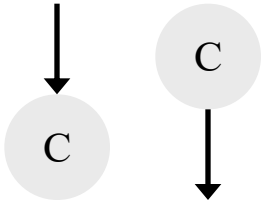
Elementare Anweisungen



Unterprogramm
Teil-/Subalgorithmus
(UP: Name)

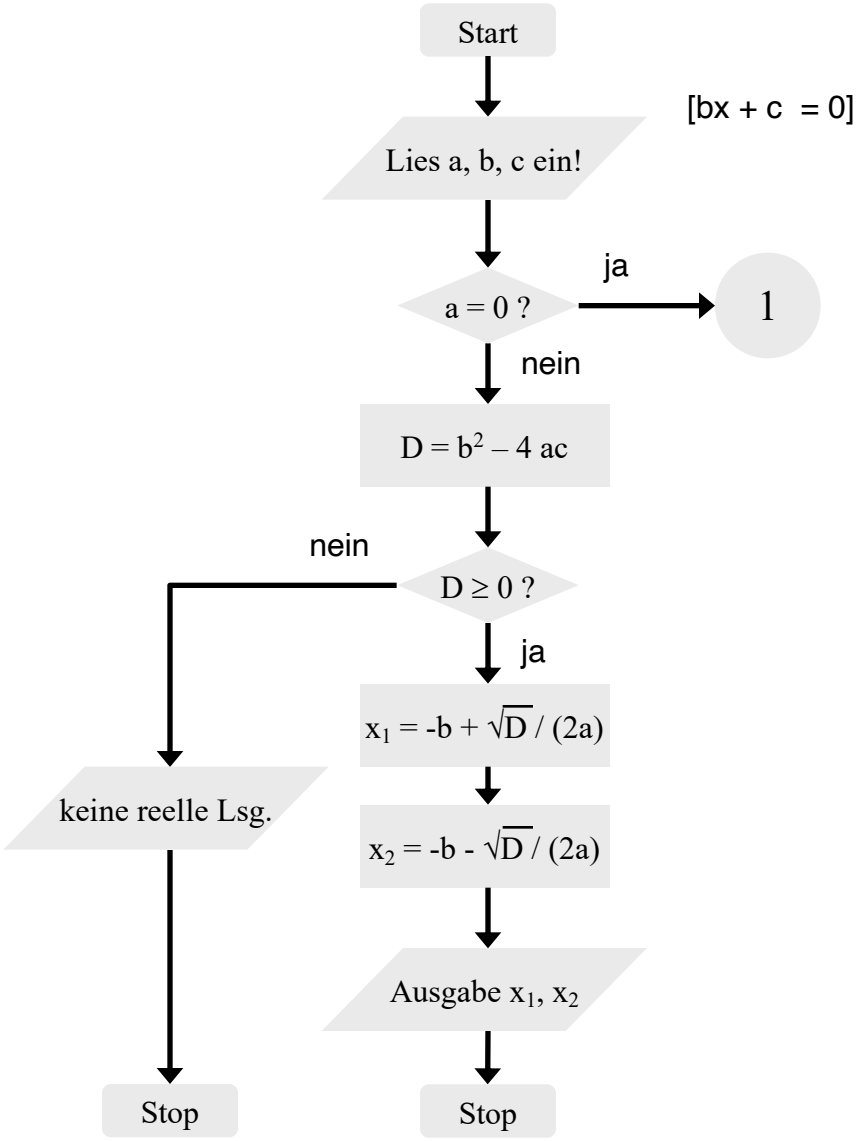


Verzweigung aufgrund einer Bedingung



Konnektor zur Verbindung von Diagrammen

Anwendungsbeispiel – Lösen einer quadratischen Gleichung

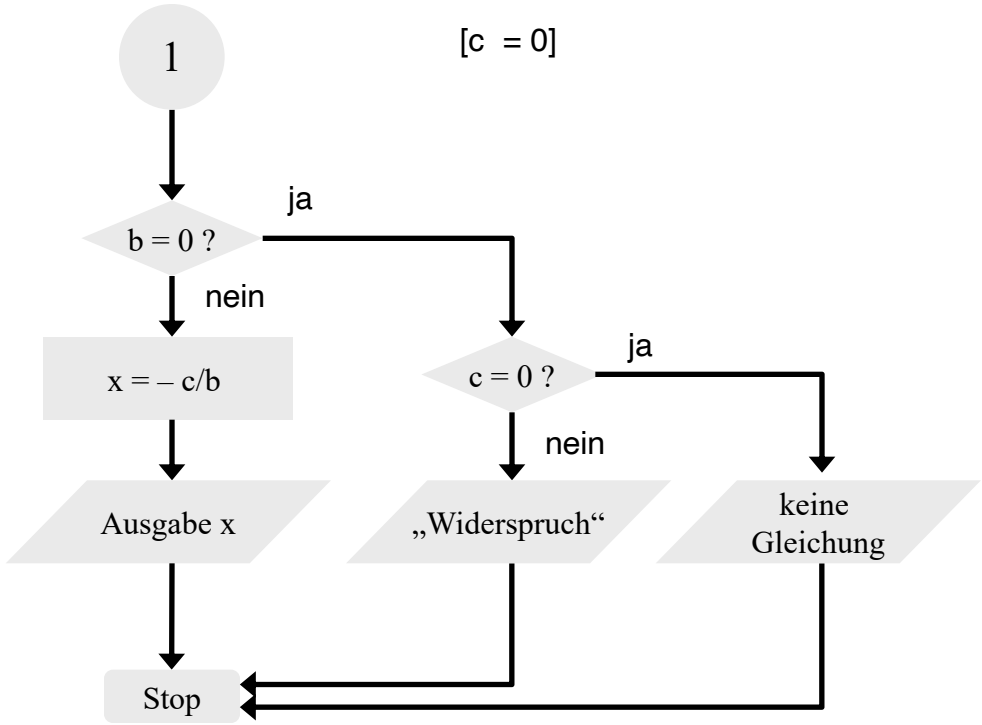


[bx + c = 0]

Quadratische Gleichung: $a x^2 + b x + c = 0$

Lösungsschema (für reelle Lösungen):

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \quad a \neq 0$$



[c = 0]

Bewertung zu Flussdiagrammen

Vorteilhaft

- Einfach verständlich
- Ermöglicht Darstellung auf verschiedenen Abstraktionsebenen
- Viele der für imperative Algorithmen benötigten Konstrukte darstellbar
- Ermöglicht genaue Beschreibung der Ablauflogik
- Zur Darstellung einfacher algorithmischer Abläufe gut geeignet
- Diagramme direkt codierbar in ausführbare Programmstücke

Nachteilig

- Bei größeren Algorithmen rasch unübersichtlich
- Keine Darstellbarkeit von Datenstrukturen und Objekten
- Keine Unterstützung bei der systematischen Konstruktion von Algorithmen
- Keine Unterscheidbarkeit von normalen Verzweigungsanweisungen und Schleifenrücksprüngen (Wiederholungen)
- Flusspfeil kann mit jedem anderen Flusspfeil zusammengeführt werden (**goto**)
- Goto-Anweisungen entsprechen nicht der modernen Auffassung bzgl. Strukturiertheit von Programmen

2. Wiederholungsanweisungen

- Abweisende Schleifen – **`while`**-Anweisung
- Annehmende Schleifen – **`do..while`**-Anweisung
- Zählschleifen – **`for`**-Anweisung
- Geschachtelte Schleifen
- Unterbrechung – **`break`**- und **`continue`**-Anweisung
- Aufzählungen und **`for`**-each-Schleifen

Abweisende Schleifen – `while`-Anweisung

Zur Erinnerung – Kurzwiederholung

- **Schleifen** führen Einzel- oder mehrere Anweisungen (in einem Block) wiederholt aus (Wiederholungsanweisung); die wiederholte Ausführung wird als Iteration bezeichnet
- Die drei gebräuchlichsten Schleifenformate in Java sind:
  - **`while`**
  - **`do-while`**
  - **`for`**
- Für neu definierte Aufzählungstypen (in Java 5.0) kann auf die Elemente entsprechend ihrer Ordnung in Schleifen zugegriffen werden:
  - **`for`**-each-Schleifen (erweiterte Zählschleifen)

Aufbau einer `while`-Schleife

Die **`while`**-Schleife wiederholt eine Anweisung, solange eine Bedingung erfüllt ist (vgl. Kapiteleinführung)

```
while (<Bedingung>)
   <Anweisung>
```

Erläuterungen:

- **<Bedingung>** (Schleifenkopf) ist ein boolescher Ausdruck
- **<Anweisung>** (Schleifenrumpf) ist
  - eine beliebige (Einzel-) Anweisung oder
  - eine Folge von Anweisungen, eingeschlossen in **`{ ... }`** (ein Block)

```
while (<Bedingung>) {
   <Anweisungen>
}
```

Hinweis: Die Anweisung oder der Block kann wiederum aus beliebigen Anweisungen bestehen, also selbst Wiederholungen oder Auswahl-Anweisungen enthalten

Ausführung und Abarbeitung

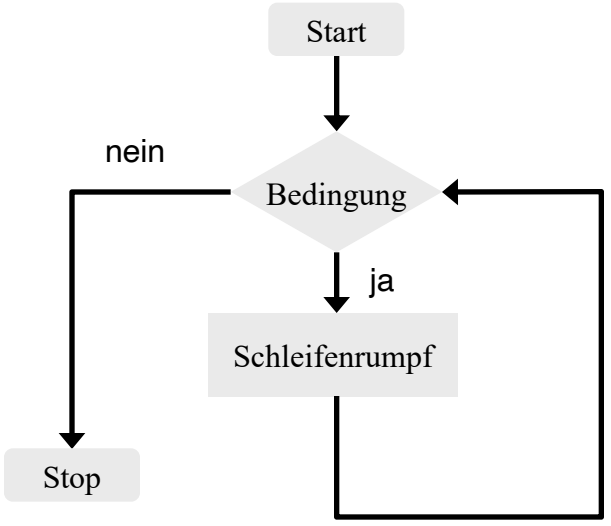
Abarbeitung einer `while`-Schleife

1. Die Bedingung (Schleifenkopf) wird ausgewertet
2. Die Anweisung (Schleifenrumpf) wird wiederholt ausgeführt, solange die Bedingung erfüllt (**`true`**) ist; ebenso, wenn die Bedingung fehlt
3. Ist die Bedingung nicht oder nicht mehr erfüllt (**`false`**), so wird die Ausführung der **`while`**-Schleife beendet

Bsp.:

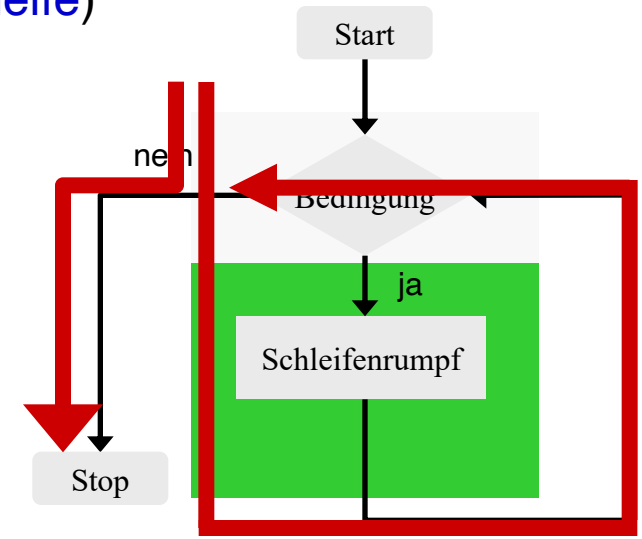
```
int k   = 1,
    fak = 1;

while (k <= 7) {
   fak = fak * k;
   k++;
}
```

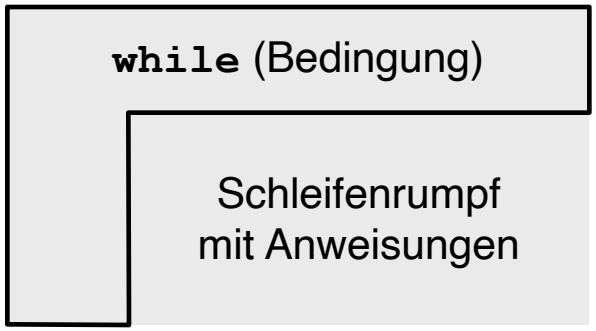
Liefert Ergebnis: `7! = 5040`



Die `while`-Schleife hat einen abweisenden Charakter

- Eine **`while`**-Schleife verwendet eine Vorabprüfung, in der die Bedingung vor Eintritt und Ausführung des Schleifenrumpfes ausgewertet wird (abweisende Schleife)





- Der Schleifenrumpf wird 0-, 1- oder mehrmals ausgeführt

```
k = 1;
while (k <= 7) {
   fak = fak * k;
   k++;
}
```

Annehmende Schleifen – `do..while`-Anweisung

Aufbau einer `do..while`-Schleife

Die **`do..while`**-Schleife wiederholt eine Anweisung, bis eine Bedingung nicht mehr erfüllt ist

```
do <Anweisung>
while (<Bedingung>);
```

Erläuterungen:

- **<Anweisung>** (Schleifenrumpf) ist
  - eine beliebige (Einzel-) Anweisung oder
  - eine Folge von Anweisungen, eingeschlossen in **`{ ... }`** (ein Block)

  (die Anweisung oder der Block kann wiederum aus beliebigen Anweisungen bestehen, also selbst Wiederholungen oder Auswahl-Anweisungen enthalten)

```
do {
   <Anweisungen>
} while (<Bedingung>);
```

- **<Bedingung>** (Schleifenkopf) ist ein boolescher Ausdruck

Ausführung und Abarbeitung

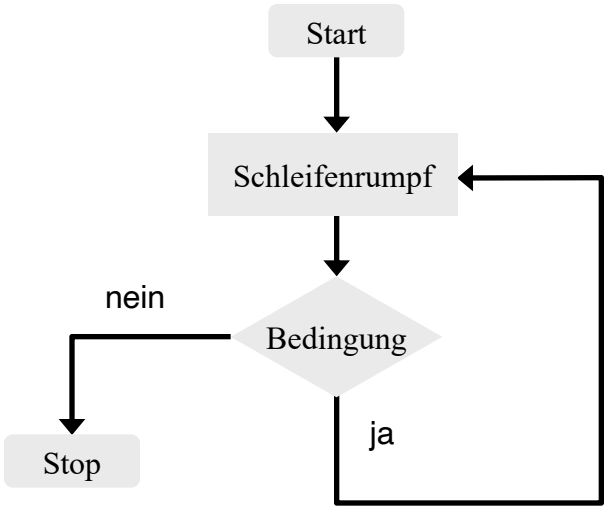
Abarbeitung einer `do..while`-Schleife

1. Die Anweisung (Schleifenrumpf) wird ausgeführt
2. Die Bedingung wird ausgewertet
3. Die Anweisung (Schleifenrumpf) wird wiederholt ausgeführt, solange bis die Bedingung nicht mehr erfüllt (**`false`**) ist

Bsp.: Die annehmende Schleife implementiert die Iteration in Form einer inkrementellen Schleife

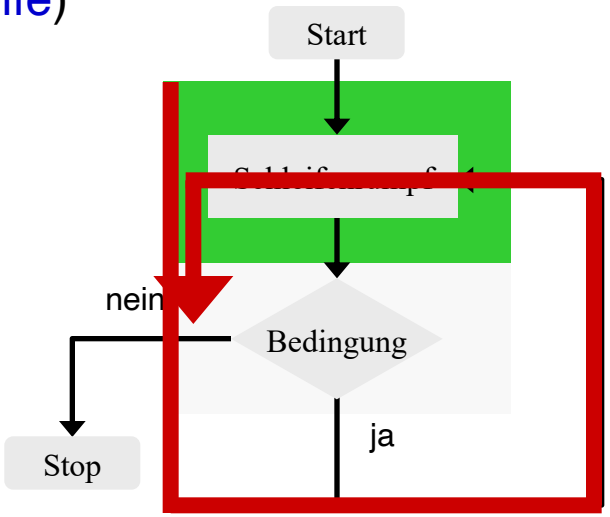
```
int k   = 1,
    fak = 1;

do {
   fak = fak * k;
   k++;
} while (k <= 7);
```

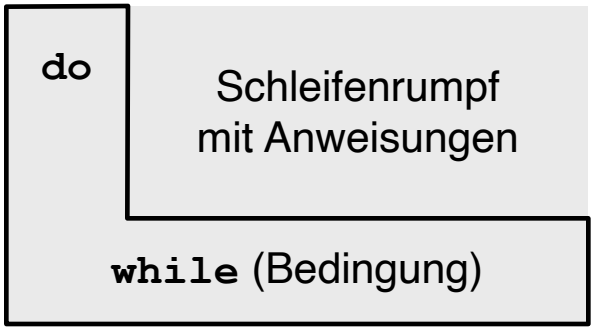
Liefert Ergebnis: `7! = 5040`



Die `do..while`-Schleife hat einen annehmenden Charakter

- Eine **`do..while`**-Schleife verwendet eine Nachprüfung – der Schleifenrumpf wird bereits einmal ausgeführt, bevor die Bedingung getestet wird (annehmende Schleife)





- Der Schleifenrumpf wird also 1- oder mehrmals ausgeführt (d.h. der Schleifenrumpf wird 1-mal mindestens ausgeführt)

Bsp.:

```
k = 7;  // als negative Zaehlschleife ...
do {
   fak = fak * k;
   k--;
} while (k > 0);
```

Wie transformiert man eine `do..while`- in eine `while`-Schleife?

Problem: Annehmende Schleifen sind häufige Fehlerquellen … vermeiden!

Lösung: Problemformulierung basierend auf abweisenden Schleifen

Gegeben sei eine **`do..while`** Schleife:

```
do {
   <Anweisungen>
} while (<Bedingung>);
```

- Transformation: Lösung mit Code-Verdoppelung

```
<Anweisungen>;
while (<Bedingung>) {
   <Anweisungen>;
}
```

- Transformation: Lösung mit Schalter-Variablen

```
boolean schalter = true;

while (schalter) {
   <Anweisungen>;
   schalter = Bedingung;
}
```

Zählschleifen – `for`-Anweisung

Aufbau einer `for`-Schleife

Besonderheiten

- Eine (Zähl-) Schleife besteht meist aus drei Teilen:
  - Initialisierung, z.B. `int i = 1;`
  - Abfrage der Schleifenbedingung, z.B. `i <= 7`
  - Fortschaltung, z.B. `i++`
- Beispiel (genaue Deklaration, nächste Seite)

```
int k   = 1,
    fak = 1;

while (k <= 7) {
   fak = fak * k;
   k++;
}
```

```
int k,
    fak = 1;

for (k = 1; k <= 7; k++)
   fak = fak * k;
```

Struktur einer `for`-Schleife

```
for (<Vorbereitung>; <(Start-)Bedingung>; <Fortschaltung>)
   <Anweisung>;
```

Erläuterungen:

- **<Vorbereitung>** ist …
  - eine oder
  - mehrere durch Komma getrennte Anweisungen (Initialisierung: Zählvariable) oder
  - eine Variablen-Deklaration
- **<(Start-)Bedingung>** ist ein boolescher Ausdruck
- **<Fortschaltung>** ist …
  - eine Anweisung oder
  - mehrere durch Komma getrennte Anweisungen
- **<Anweisung>** ist …
  - eine beliebige Anweisung oder
  - eine Folge von Anweisungen, eingeschlossen in `{ . . . }` (ein Block)

```
for (<Vorbereitung>; <(Start-)Bedingung>; <Fortschaltung>) {
   <Anweisungen>;
}
```

Ausführung und Abarbeitung

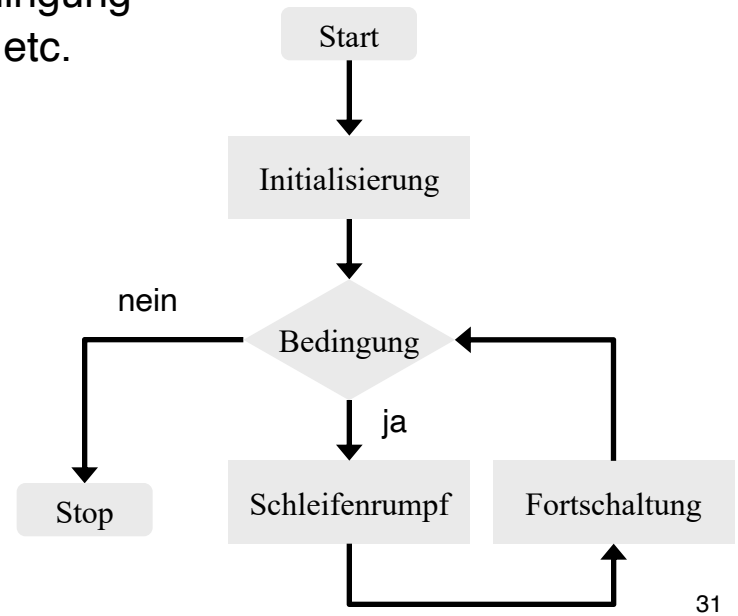
Abarbeitung einer `for`-Schleife

1. Die Schleife wird initialisiert (Zählvariable)
2. Die Bedingung wird ausgewertet
3. Falls die Bedingung erfüllt ist, wird der Schleifenrumpf (Anweisung, Anweisungsfolge) ausgeführt
4. Anschließend wird die Fortschaltung ausgeführt (Änderung der Zählvariable(n))
5. Die Bedingung wird erneut ausgewertet; falls die Bedingung erfüllt ist, wird erneut der Schleifenrumpf ausgeführt, etc.

Bsp.:

```
int k,
    fak = 1;

for (k = 1; k <= 7; k++)
   fak = fak * k;
```

Liefert Ergebnis: `7! = 5040`



Weitere Details zu `for`-Schleifen

- Eine innerhalb der Vorbereitung deklarierte Variable ist nur innerhalb der Schleife gültig

Bsp.:

```
int fak = 1;

for (int k = 1; k <= 7; k++)
   fak = fak * k;
```

- In Zählschleifen wird die **`for`**-Anweisung verwendet, um den Schleifenrumpf für jeden Wert einer Laufvariable zwischen Unter- und Obergrenze einmal auszuführen
  - **Inkrementelle** Zählschleife

```
for (<variable> = <min>; <variable> <= <max>; <variable>++) {
   <Anweisungen>;
}
```

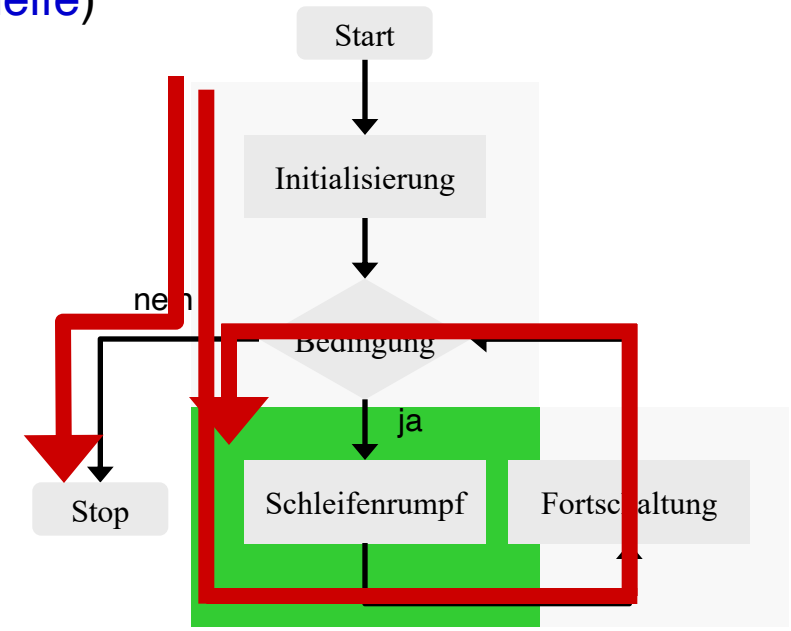
  - **Dekrementelle** Zählschleife

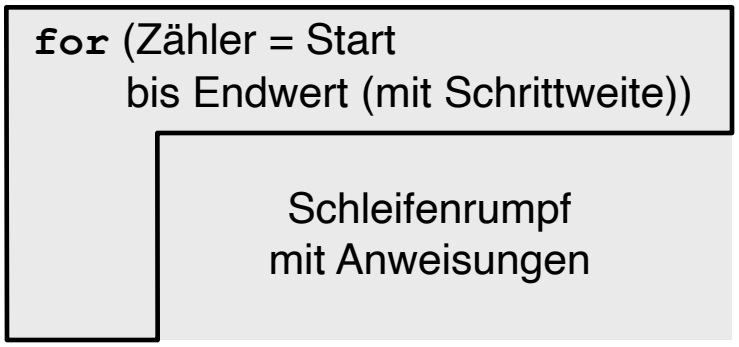
```
for (<variable> = <max>; <variable> >= <min>; <variable>--) {
   <Anweisungen>;
}
```

Der Test in der Abbruchbedingung ist stets eine Fehlerquelle: *Soll der letzte Wert mit zum Ergebnis beitragen oder nicht?*

Die `for`-Schleife hat einen abweisenden Charakter

- Eine **`for`**-Schleife verwendet eine Vorabprüfung, in der die Bedingung vor Eintritt und Ausführung des Schleifenrumpfes ausgewertet wird (abweisende Schleife)





- Der Schleifenrumpf wird 0-, 1- oder mehrmals ausgeführt
- Die Fortschaltung (Änderung der Zählvariablen) kann auch eine feste Schrittweite betragen, die von 1 verschieden ist

```
int acc = 0;

for (k = 1; k <= 8; k = k + 2)
    acc = acc + k;
```

Unterschiede und Gemeinsamkeiten zwischen den Schleifen

Verwendung und Eigenschaften

- Jede der 3 Arten von Schleifen ist auch durch eine beliebige der anderen beiden ersetzbar

  Jede der Schleifen hat jedoch einen Einsatzzweck, für den sie besonders gut geeignet ist

- **`while`** vs. **`do..while`** Schleife

  Bei der **`do..while`** Schleife wird der Schleifenkörper auf alle Fälle einmal ausgeführt, auch wenn die Bedingung bereits zu Beginn nicht erfüllt ist

  Hinweis: Aus diesem Grund ist die **`do..while`** Schleife nur mit großer Vorsicht anzuwenden, da sie eine häufige Fehlerquelle darstellt

- **`for`** vs. **`while`** Schleife

  Die **`for`** Schleife ist besonders gut geeignet, wenn es darum geht den Schleifenkörper ein bestimmte Anzahl mal zu wiederholen

Beispiel – Ausgabe von Zahlenfolgen

Aufgabe: Es sollen die geraden Zahlen zwischen 2 und 20 ausgegeben werden

Gesuchtes Ergebnis: 2 4 6 8 10 12 14 16 18 20

Lösungen:
(die Variable **`int`** n sei deklariert)

- **`for`**-Schleife mit 2er-Inkrement

```
for (n = 2; n <= 20; n = n + 2)
   System.out.print(n + " ");
```

- **`while`**-Schleife mit 2er-Inkrement

```
n = 2;
while (n <= 20) {
   System.out.print(n + " ");
   n = n + 2;
}
```

- **`do..while`**-Schleife mit 2er-Inkrement

```
n = 0;
do {
   n = n + 2;  // n wird VOR Ausgabe inkrementiert
   System.out.print(n + " ");
} while (n < 20);
```

- **`for`**-Schleife mit 1er-Inkrement, aber nur gerade Zahlen werden gedruckt

```
for (n = 2; n <= 20; n++)
   if ((n % 2) == 0)  // ist n eine gerade Zahl?
      System.out.print(n + " ");
```

- Eine besondere Lösung, die den Dozenten stutzig macht … ;-)

```
for (n = 1; n <= 1; n++)
   System.out.print("2 4 6 8 10 12 14 16 18 20");
```

Geschachtelte Schleifen

Geschachtelte Zählschleifen

- Zur Erinnerung: Schleifenkörper können aus beliebigen Anweisungen zusammen gesetzt sein

  Daraus folgt: Schleifenkörper können auch selbst wieder Schleifen enthalten

  Bsp.: Drucke eine Tabelle der Zeilen- und Spaltenprodukte

```
for (rowNo = 1; rowNo <= 12; rowNo++) {
   for (int n = 1; n <= 12; n++) {
      // formatiertes Zahlen-Drucken mit vier Zeichen
      System.out.printf("%4d", n * rowNo);
   }
   // Zeilenvorschub am Zeilenende
   System.out.println();
}
```

Erläuterungen und Fragen:

- Für jeden Zeilenindex werden die Produkte der Zeile mit n = 1 .. 12 gedruckt
- *Welche Klammern { . . . } sind in dem gezeigten Beispiel nicht notwendig?*

- Oft hängt die Anzahl der Iterationen der inneren Schleife von der Laufvariable der äußeren Schleife ab

Bsp.: Drucken eines Dreiecks aus '*'-Elementen

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Hinweise:
- die Höhe des Dreiecks wird durch eine Konstante (**`final`**) angegeben (hier: 10)
- die Anzahl der Elemente innerhalb einer Schicht wird durch deren äußeren Index gesteuert; die Zählschleife hierfür ist inkrementell

```
final int LEVEL_MAX = 10;

for (int level = 1; level <= LEVEL_MAX; level++) {
   for (int k = 1; k <= level; k++)
      System.out.print("*");  // Ausgabe einer Schicht
                              // des Dreiecks

   System.out.println();      // neue Zeile ...
}
```

Beispiel – Kalenderblatt

Aufgabe

- Es soll die Ausgabe eines Kalenderblattes in Java programmiert werden
- Das Programm soll mit beliebigen Monatslängen (28 – 31 Tage) sowie unterschiedlichen Start-Tagen des Monats zurechtkommen
- Die **Ausgabe des Programms** könnte in etwa so aussehen:

```
Mo  Di  Mi  Do  Fr  Sa  So
                1   2   3   4
5   6   7   8   9   10  11
12  13  14  15  16  17  18
19  20  21  22  23  24  25
26  27  28  29  30  31
```

Lokal definierte Variablen
(Gültigkeit nur innerhalb des jeweiligen Blocks)

Eine mögliche Lösung …

```
int firstDay =  3,
    noOfDays = 31,
    noOfRows = (noOfDays + firstDay - 1) / 7 + 1;

System.out.println("Mo Di Mi Do Fr Sa So");

for (int row = 0; row < noOfRows; row++) {
    for (int col = 0; col < 7; col++) {
        int day = row * 7 + col + 1 - firstDay;

        if (day > noOfDays || day <= 0)
            System.out.print("   ");
        else if (day < 10)
            System.out.print(day + "  ");
        else
            System.out.print(day + " ");
    }
    System.out.println();
}
```

Was noch fehlt …

- Eingabe des Jahres, Monats und Starttages durch den Benutzer
- automatische Ermittlung der Anzahl der Tage des Monats
- …

Unterbrechungen – `break`- und `continue`-Anweisung

Die `break`-Anweisung

Die Ausführung einer inneren umschließenden **`while`**-, **`do..while`**-, **`for`**- oder **`switch`**-Anweisung soll in einigen Fällen sofort beendet werden

Bsp.: Primzahlen berechnen und ausgeben

```
for (int k = 1; k < 50; k++) {
   int j;
   for (j = 2; j < k; j++) {
      if ((k % j) == 0)
         break;
   }
   if (k == j)
      System.out.println("  " + k);
}
```

liefert Ergebnis: Primzahlen von 1 bis 50

umschließende Schleife (relativ zur **`break`**-Anweisung)

Quizfrage:
Könnten wir hier **j** auch innerhalb der for-Schleife deklarieren?
→ **for** (**int j** =2; …)

Hinweise: Die **`break`**-Anweisung veranlasst das Programm, die umschließende Schleife (oder das umschließende **`switch`**) zu verlassen und mit der nächsten Anweisung nach der schließenden Schleife fortzufahren; die Angabe von **Sprungzielen** ermöglicht die Fortsetzung an einer beliebigen Stelle (z.B. außerhalb verschachtelter Schleifen)

Die `continue`-Anweisung

- Die Ausführung springt sofort zum Ende der aktuellen Iteration

Bsp.: Zeichenhäufigkeit in Texten bestimmen

```
String s = "Alle anfallenden Arbeiten auf Andere abschieben, " +
           "anschließend anscheißen, aber anständig!";
int numAs = 0;

for (int i = 0; i < s.length(); i++) {
   if ((s.charAt(i) != 'a') && (s.charAt(i) != 'A'))
      continue;

   numAs++;
}
System.out.println(numAs + " a/A s");
```

liefert Ergebnis: 11 a/A‘s

Hinweise: Die **`continue`**-Anweisung veranlasst das Programm, die umschließende Schleife mit der nächsten Iteration zu starten – ohne die restlichen Anweisungen des Blocks in der aktuellen Iteration auszuführen; die Angabe von **Sprungzielen** ermöglicht die Fortsetzung an einer beliebigen Stelle

Aufzählungen und `for`-each-Schleifen

♣ ist nicht auf enum-Typen beschränkt!

Zählschleifen auf Aufzählungsmengen

- Ab Java 5.0 ist ein **`for`**-Schleifen-Typ definiert, der auf Datenstrukturen (als Kollektion von Datenelementen) operiert ♣
- **Aufbau** der Zählstreifen für Aufzählungstypen

```
for (<enum-Typ-Name> <Variable> :
     <enum-Typ-Name>.values())
   <Anweisung>;
```

oder für einen Block mit mehreren Anweisungen

```
for (<enum-Typ-Name> <Variable> : <enum-Typ-Name>.values()) {
   <Anweisungen>;
}
```

Erläuterungen:

- **<Variable>** ist lokal in der **`for`**-Schleife definiert (vom Typ <enum-Typ-Name>)
- **<enum-Typ-Name>.values()** ist eine Funktion, die eine Liste der Werte der Aufzählung liefert (`values()` ist eine Klassenmethode für **`enum`**)

Bsp.: Drucken einer Liste von Wochentagen

Die Enumeration wurde bereits definiert:

```
enum Day {MON, TUE, WED, THU, FRI, SAT, SUN};
```

Folge von Ausgaben in einer Zählschleife:

```
for (Day d : Day.values())
   System.out.println(d + " ist der " +
                      (d.ordinal()+1) + ". Tag der Woche");
```

liefert das Ergebnis:

```
MON ist der 1. Tag der Woche
TUE ist der 2. Tag der Woche
WED ist der 3. Tag der Woche
THU ist der 4. Tag der Woche
FRI ist der 5. Tag der Woche
SAT ist der 6. Tag der Woche
SUN ist der 7. Tag der Woche
```

Anmerkungen:

- Die Indizes der Liste der Werte beginnen bei 0
- Lesart für diese **`for`**-Schleife: `for` *each* `Day d` *in* `Day.values()`, d.h. dass `:` die Bedeutung "*in*" hat

3. Auswahlanweisungen

- Bedingte Programmverzweigung – **`if`**-Anweisung
- Bewachte Anweisungen – **`switch`**-Anweisung
- Die leere Anweisung
- Vergleichsoperatoren – **`?  :`**-Operator

Bedingte Programmverzweigung – `if`-Anweisung

Zur Erinnerung – Kurzwiederholung

- **Auswahlanweisungen** führen zur Verzweigung des Programmflusses in alternative Pfade mit unterschiedlichen Anweisungen
- Eine Verzweigung (bedingte Anweisung) kann
  - einfach sein, d.h. eine oder mehrere Anweisungen werden ausgeführt oder nicht
  - die Entscheidung kann eine Alternative sein
- Die gebräuchlichsten Auswahlformate für diese Verzweigungen in Java sind:
  - **`if`**
  - **`if-else`**
- Weitere Auswahlanweisungen in Java
  - **`switch`**-Anweisungen (bewachte Anweisungen)
  - **`? :`**-Operator zur kompakten (Operator-) Notation von **`if-else`** Anweisungen

Aufbau einer `if`-Anweisung

Bedingte Anweisungen (**`if`**-Anweisungen) ermöglichen **Fallunterscheidungen** und **Verzweigungen** in Programmen ([…]: optionaler Teil)

```
if (<Bedingung>)
   <Anweisung-1>
[else
   <Anweisung-2>]
```

Erläuterungen:

- **<Bedingung>** ist ein boolescher Ausdruck
- **<Anweisung>** (Anweisung-1 bzw. Anweisung-2) ist
  - eine beliebige (Einzel-) Anweisung oder
  - eine Folge von Anweisungen, eingeschlossen in **`{ ... }`** (ein Block)

Hinweis: In einigen **Programmiersprachen** wird die Struktur der Verzweigung noch stärker verdeutlicht, indem die Verzweigungen jeweils durch reservierte Worte eingeleitet werden:

```
if (<Bedingung>) then <Anweisung-1> [else <Anweisung-2>]
```

Für Anweisungen, die durch einen Block definiert sind, ergibt sich das allgemeine Format von bedingten Anweisungen ([…]: optionaler Teil):

```
if (<Bedingung>) {
   <Anweisungen-1>
}
[else {
   <Anweisungen-2>
}]
```

Hinweis: Die Anweisung oder der Block kann wiederum aus beliebigen Anweisungen bestehen, also selbst Wiederholungen oder Auswahl-Anweisungen enthalten; zur **Vermeidung von Fehlern** kann man **immer Blöcke** mit `{...}` verwenden

Bsp.:

```
String s = "Alle anfallenden Arbeiten auf Andere abschieben";
int numAs = 0;

for (int i = 0; i < s.length(); i++){
   if ((s.charAt(i) == 'a') || (s.charAt(i) == 'A'))
      numAs++;
}
System.out.println(numAs + " a/A s");
```

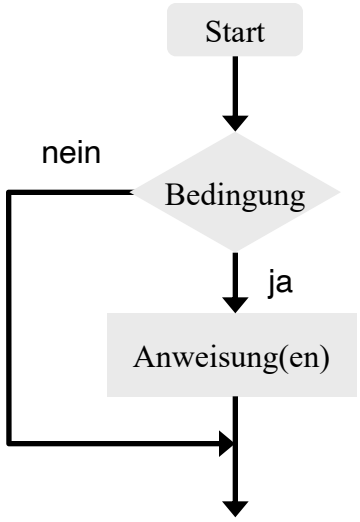
Ausführung und Abarbeitung

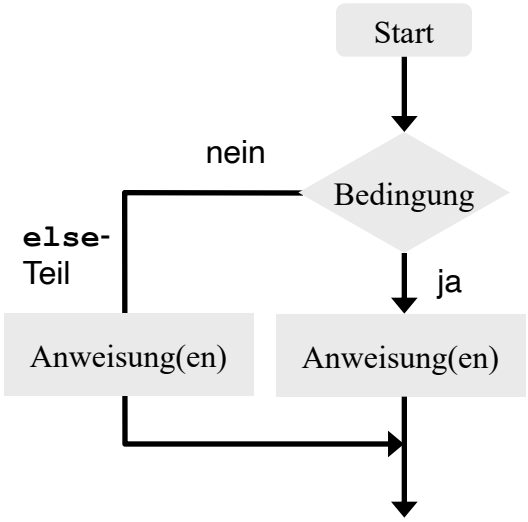
Abarbeitung einer `if`-Anweisung

1. Die Bedingung wird ausgewertet
2. Falls die Bedingung erfüllt ist, d.h. die Auswertung den Wert **`true`** ergibt, wird die darauf folgende Anweisung (oder Anweisungen) ausgeführt (**then**-Zweig)
3. Falls die Bedingung nicht erfüllt ist, wird die Anweisung nicht ausgeführt; falls vorhanden, wird in diesem Fall die dem reservierten Wort **`else`** folgende Anweisung (oder Anweisungen) ausgeführt (**else**-Zweig)

Bsp.:

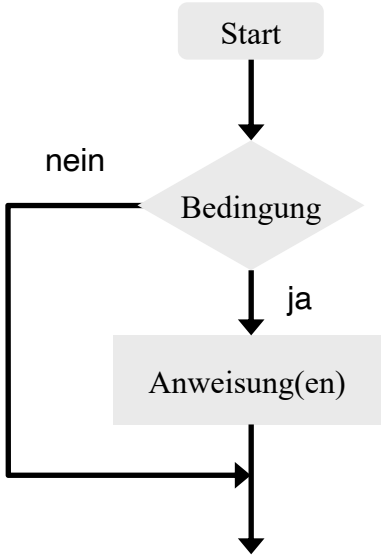
```
int x;

if (x < 0)
   x = 0;  // max{x, 0}
...;  // hier weiter
```

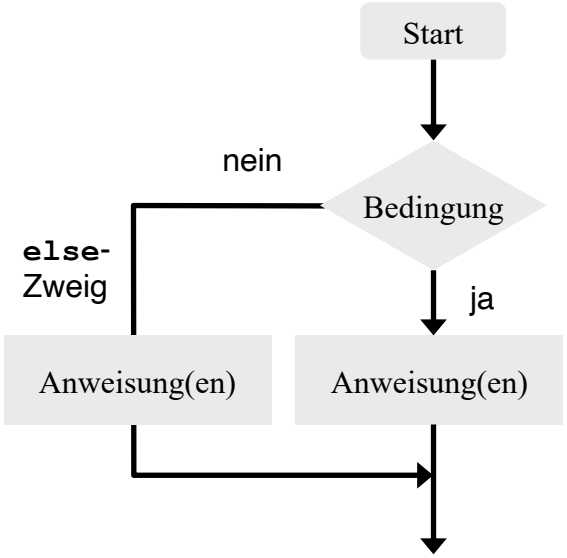




Darstellungen der Struktur von `if`-Anweisungen

Darstellung von Auswahlanweisungen mit Flussdiagrammen





Das *dangling* `else`-Problem

- Die Anweisung-1 (then-Teil) kann selbst aus beliebigen Anweisungen, z.B. aus weiteren **`if`**-Anweisungen, bestehen

```
if (<Bedingung>)
   <Anweisung-1>
[else
   <Anweisung-2>]
```

- Ein **Problem** in der **Interpretation** der Anweisung entsteht, wenn `<Anweisung-1>` eine **`if`**-Anweisung ist, die keinen **`else`**-Teil besitzt – *gehört der else-Teil zur äußeren oder inneren Verzweigung (als Alternative zu Bedingung-1 oder zu Bedingung-2)?*

```
if (<Bedingung-1>)
   if (<Bedingung-2>)
      <Anweisung-1>
else
   <Anweisung-2>
```

⟺

```
if (<Bedingung-1>)
   if (<Bedingung-2>)
      <Anweisung-1>
   else
      <Anweisung-2>
```

Wichtig: Die Einrückung ist für den Computer bedeutungslos!

Lösung (Interpretation):

- Bei geschachtelten Anweisungen ohne Gruppierung wird der else-Zweig immer dem innersten **`if`** zugeordnet, zu dem es gehören kann
- Durch die Gruppierung von Anweisungen mittels **`{ . . . }`** kann eine andere Zuordnung erzielt werden

```
if (<Bedingung-1>) {
   if (<Bedingung-2>)
      <Anweisung-1>
}
else
   <Anweisung-2>
```

Mehrfachauswahl mit `if`-Anweisungen

Die `if...else if`-Konstruktion

```
if (<Bedingung>)
   <Anweisung-1>
else
   <Anweisung-2>
```

- Wenn <Anweisung-2> (der else-Teil der `if`-Anweisung) selbst eine `if`-Anweisung ist, können mehrere Unterscheidungen getroffen werden
- **Format** der Anweisung (für 3-fach Auswahl):

```
if (<Bedingung-1>)
   <Anweisung-1>
else
   if (<Bedingung-2>)
      <Anweisung-2>
[  else
      <Anweisung-3>]
```

```
if (<Bedingung-1>)
   <Anweisung-1>
else if (<Bedingung-2>)
   <Anweisung-2>
[else
   <Anweisung-3>]
```

Übliche Formatierung entsprechend der Java-*Styleguides*

Bsp.: Ausgabe unterschiedlicher Text-Nachrichten

```
if (temperature <= 15)
   System.out.println("bis 15°: Es ist kalt ... ");
else if (temperature < 32)
   System.out.println("Unter 32°: Es ist angenehm ... ");
else
   System.out.println("Ab 32° und darüber: Es ist heiß ... ");
```

Zusammensetzung mehrer `else-if`‘s

- Durch Zusammensetzung mehrerer **`else-if`**-Anweisungen können beliebig viele Unterscheidungen getroffen werden
- **Format** der Anweisung:

```
if (<Bedingung-1>)
    <Anweisung-1>
else if (<Bedingung-2>)
    <Anweisung-2>
else if (<Bedingung-3>)
    <Anweisung-3>
    :                          // (weitere Faelle)
else if (<Bedingung-N>)
    <Anweisung-N>
[else
    <Anweisung-(N+1)>]
```

Wichtig:

- Es wird eine Bedingung nach der anderen ausgewertet, bis ein boolescher Ausdruck **`true`** liefert; die zugehörige Anweisung wird danach ausgeführt – die restlichen Bedingungen werden übersprungen
- Jede der Anweisungen kann auch ein Block (Gruppierung durch `{...}`) sein: Deren (Einzel-) Anweisungen werden ausgewertet, sofern die zugehörige Bedingung erfüllt ist

Bewachte Anweisung – `switch`-Anweisung

Aufbau einer `switch`-Anweisung

- Bewachte Anweisungen ermöglichen die Auswahl von Anweisungsfolgen aus einer Menge von Alternativen
- In Java wird eine bewachte Anweisung mit Hilfe einer **`switch`**-Anweisung realisiert:

```
switch (<Ausdruck>) {
   case <Konstante-1>:
      <Anweisungen-1>
      break;
   case <Konstante-2>:
      <Anweisungen-2>
      break;
      :  // weitere case-Marken
   case <Konstante-N>:
      <Anweisungen-N>
      break;
[  default:  // optionale Auswahl
      <Anweisungen-(N+1)>]
}
```

Erläuterungen:

- Der Wert von **<Ausdruck>** kann
  - ein ganzzahliger Typ (**`int`**, **`short`**, **`byte`**),
  - **`char`** oder
  - ein Aufzählungstyp
  - ein String ♣

  sein; aber keine reelle Zahl (**`real`**, **`double`**)
- Eine Marke **`case`** **<Konstante>`:`** bezeichnet die Sprungposition, an der die Ausführung der Auswahlalternative beginnt, wenn das Ergebnis der Auswertung des Ausdrucks den Wert der Konstante liefert

```
switch (<Ausdruck>) {          case labels
    case <Konstante-1>:
        <Anweisungen-1>
        break;
    case <Konstante-2>:
        <Anweisungen-2>
        break;
    :       // weitere case-Marken
    case <Konstante-N>:
        <Anweisungen-N>
        break;
[   default:  // optionale Auswahl
        <Anweisungen-(N+1)>]
}
```

- Die letzte Alternative verwendet das Label **`default:`** und bezeichnet die Sprungposition für Anweisungen, wenn das Ergebnis der Auswertung des Ausdrucks nicht in der Liste der *case labels* geführt wird; die Verwendung des *default-labels* ist optional
- **<Anweisung>** ist eine beliebige Folge von Programm-Anweisungen
- **`break`** beendet die Ausführung der **`switch`**-Anweisung und setzt das Programm nach der **`}`**-Klammer fort; die Angabe von **`break`** ist optional

♣ **ab Java 7**

Ausführung und Abarbeitung

Abarbeitung einer `switch`-Anweisung

1. Auswertung des (ganzzahligen) Ausdrucks
2. Auswahl des dem Wert des Ausdrucks (*case label*) entsprechenden Zweiges
3. Bearbeitung der nachfolgenden Anweisungen bis zum nächsten **`break`** (oder bis zum Ende der **`switch`**-Anweisung)
4. Passt keiner der Zweige zum Wert des Ausdrucks, so wird – falls vorhanden – der **`default`**-Zweig ausgeführt; existiert kein **`default`**-Zweig wird die Bearbeitung der **`switch`**-Anweisung beendet

Bsp.: Ausgabe von Text-Nachrichten (vgl. S.54)


```
switch (temperatureCategory) {
    case 1:
        System.out.println("Ab 32° und darüber: Es ist heiß ... ");
        break;
    case 2:
        System.out.println("Unter 32°: Es ist angenehm ... ");
        break;
    default:
        System.out.println("bis 15°: Es ist kalt ... ");
}
```

Die Rolle von `break` in `switch`-Anweisungen

- Die Verwendung von **`break`** in **`switch`**-Anweisungen ist optional
- Die Bearbeitung eines Zweiges (beginnend mit einem *case label*) endet erst beim nächsten **`break`** oder am Ende der switch-Anweisung (`}`-Klammer)

  Konsequenz: Hat man am Ende eines Zweiges das **`break`** vergessen, werden auch die Anweisungen des nächsten Zweigs bearbeitet, etc. (Achtung: häufige Fehlerquelle!)

Bsp.:

```
int index = 1;

switch ( index ) {
    case 0:
    case 1:
        System.out.print("0 oder 1");
    case 2:
        System.out.print("2");
    case 3:
        System.out.print("3");
        break;
    case 4:
        System.out.print("4");
        break;
    default:
        System.out.println("?");
}
```

Achten Sie stets darauf, dass kein **`break`** vergessen wurde!

ergibt für `index=1` die Ausgabe: `0 oder 123`

Ein konstruktives Beispiel

```
int n;

switch ( n ) {
    case 1:
        System.out.println("Die Zahl ist 1.");
        break;
    case 2:
    case 4:
    case 8:
        System.out.print("Die Zahl ist 2, 4 oder 8.");
        System.out.println("(Vielfache von 2)");
        break;
    case 3:
    case 6:
    case 9:
        System.out.print("Die Zahl ist 3, 6, oder 9");
        System.out.println("(Vielfache von 3)");
        break;
    case 5:
        System.out.println("Die Zahl ist 5.");
        break;
    default:
        System.out.println("Die Zahl ist 7 oder außerhalb des " +
                           "Bereiches 1 bis 9.");
}
```

Menüs und `switch`-Anweisungen

Eine Anwendung von **`switch`**-Anweisungen ist die Steuerung von Menü-Aus/-Eingaben

(Demo: MenueSelections.java)

```
1  public class MenueSelections {
2      /**
3       * Das Programm stellt ein Auswahl-Menue dar und liest eine Eingabe. Es werden
4       * vorgestellt:
5       * - Auswahlanweisung (switch) mit break-Anweisung
6       * - leere Anweisungen
7       * Autor: hn, Aug.2010)
8       **/
9      public static void main(String[] args) {
10         int optionNumber;    // Option Mass-Einheit (Zahl), ausgewaehlt durch Benutzer
11         double measurement,  // numerischer Eingabewert (Benutzereingabe einer Messung)
12                              // Mass-Einheit haengt von der gewaehlten Option ab
13                inches;       // Wert (Messung) - in inch umgerechnet
14
15         /* -- Anzeige-Menue und Benutzer-Eingabe der Eingabe-Option */
16         TextIO.putln("Welche Einheit fuer Ihr Eingabe? " +
17                      "(1-inches / 2-feet / 3-yard / 4-miles)");
18         TextIO.put("Geben Sie die gewuenschte Einheit ein (1, 2, 3, 4): ");
19         optionNumber = TextIO.getlnInt();
20
21         /* -- Einlesen des Messwerts und Umrechnung in inch */
22         switch (optionNumber) {
23             case 1:
24                 TextIO.put("Eingabe (Wert) [inch]: ");
25                 measurement = TextIO.getDouble();
26                 inches      = measurement;
27
28                 TextIO.putln("Wert = " + inches + " [inch]");
29                 break;
30             case 2:
31                 TextIO.put("Eingabe (Wert) [feet]: ");
32                 measurement = TextIO.getDouble();
33                 inches      = measurement * 12;
34
35                 TextIO.putln("Wert = " + inches + " [inch]");
36                 break;
37             case 3:
38                 TextIO.put("Eingabe (Wert) [yards]: ");
39                 measurement = TextIO.getDouble();
40                 inches      = measurement * 36;
41
42                 TextIO.putln("Wert = " + inches + " [inch]");
43                 break;
44             case 4:
45                 TextIO.put("Eingabe (Wert) [miles]: ");
46                 measurement = TextIO.getDouble();
47                 inches      = measurement * 12 * 5280;
48
49                 TextIO.putln("Wert = " + inches + " [inch]");
50                 break;
51             default:
52                 TextIO.putln("***Fehler!*** Ungueltige Option");
53                 System.exit(1);
54         }
55
56         /* -- weitere Möglichkeiten der Umrechnung - feet, yard, miles ... */
57         ;;;  // leere Anweisungen
58
59     } // end main
60 } // end class MenueSelections
```

Aufzählungen in `switch`-Anweisungen

Der Typ im Ausdruck der **`switch`**-Anweisung kann auch ein Aufzählungstyp (**`enum`**) sein

```
switch (<Ausdruck>) {
    case <Konstante-1>:
        <Anweisungen-1>
        break;
    :
}
```

Bsp.: Monate und Jahreszeiten

```
enum Season {SPRING, SUMMER, FALL, WINTER};

Season currentSeason;

switch (currentSeason) {
    case WINTER:  // (nicht Season.WINTER!)
        System.out.println("Dezember, Januar, Februar");
        break;
    case SPRING:
        System.out.println("Maerz, April, Mai");
        break;
    case SUMMER:
        System.out.println("Juni, Juli, August");
        break;
    case FALL:
        System.out.println("September, Oktober, November");
        break;
}
```

Hinweis: Entgegen der sonstigen Struktur werden die Werte eines Aufzählungstyps in den *case labels* nicht mittels <Variablen-Name> und Selektor angesprochen

Die leere Anweisung

Anweisungen

- In einer Anweisung, die nur aus '`;`' (Semikolon) besteht, wird keine Aktion ausgeführt

Bsp.:
```
if (done)
   ;  // (leere Anweisung)
else
   System.out.println("noch nicht fertig ...");
```

Diskussion: Die leere Anweisung in der **`if`**-Anweisung kann äquivalent durch einen Block (**`{...}`**) repräsentiert werden:

```
if (done) {
}
else
   System.out.println("noch nicht fertig ...");
```

- Überflüssige Zeilenabschlüsse nach einem Block … (keine Konsequenzen)

Bsp.:
```
if (x < 0) {
   x = -x;
};    // kein ; nach dem Ende des Blocks }
```

Leere Anweisungen können Fehlerquellen sein

- Im Zusammenspiel mit Wiederholungsanweisungen können falsch platzierte Semikolons zu **Programmfehlern** führen (semantische Fehler!)

Derartige Fehler sind häufig sehr schwer zu finden!

Bsp.:
```
for (int k = 0; k < 10; k++);
   System.out.println("Hallo");
```

Diskussion:
- Das `;` am Ende der **`for`**-Anweisung ist eine leere Anweisung, die hier 10× ausgeführt wird
- Die Ausgabe (`System.out.println(...)`) wird nach Beendigung der **`for`**-Schleife 1× ausgeführt (die Einrückung zur Formatierung ist irrelevant; Syntax)

Entspricht:
```
for (int k = 0; k < 10; k++) {
}
System.out.println("Hallo");
```

- Manche Fehler werden bei der Übersetzung (*compilation*) entdeckt

Bsp.:
```
for (int k = 0; k < 10; k++);
   System.out.println("Hallo zum " + k + ". mal");
```

Diskussion: Die Ausgabeanweisung `System.out.println(...)` ist hier nicht Bestandteil der **`for`**-Schleife, somit ist die Variable `k` in der Ausgabe undefiniert

Vergleichsoperatoren – ? :-Operator

Aufbau einer Anweisung mit ? :-Operator

- Der ? :-Operator ist eine verkürzte Schreibweise der **if**-Anweisung – Operator-Notation von **if-else** Klauseln
- **Format** der ? :-Anweisung (3-stelliger Operator):

```
<Bedingung> ? <Ausdruck-1> : <Ausdruck-2>
```

Erläuterungen:

- **<Bedingung>** ist ein boolescher Ausdruck, Resultat **true** oder **false**
- **<Ausdruck-1>** wird ausgewertet, wenn die Bedingung **true** ergibt, ergibt die Bedingung **false**, dann wird stattdessen **<Ausdruck-2>** ausgewertet; dies sind beliebige typkompatible Ausdrücke
- Der ? :-Operator ist selbst ein Ausdruck, liefert also einen Ergebniswert

Bsp.:

```
int next = ((n % 2) == 0) ? (n/2) : (3*n+1);
```

Erläuterung: Die Klammern (..) sind nicht notwendig, dienen aber der besseren Lesbarkeit des Ausdrucks!

Abarbeitung

```
<Bedingung> ? <Ausdruck-1> : <Ausdruck-2>
```

1. Die Bedingung wird ausgewertet
2. Falls die Bedingung erfüllt ist, d.h. die Auswertung den Wert **`true`** ergibt, wird der Ausdruck-1 ausgewertet (**then**-Zweig) und das Ergebnis als Wert des Ausdrucks zurück gegeben
3. Falls die Bedingung nicht erfüllt ist, d.h. die Auswertung den Wert **`false`** ergibt, wird der Ausdruck-2 ausgewertet (**else**-Zweig) und das Ergebnis als Wert des Ausdrucks zurück gegeben

Bsp.:

```
int a = 8,
    b = 3,
    c = 5;
int minAB,
    maxABC;

minAB  = (a < b) ? (a) : (b);
maxABC = (a > b) ? ((c > a) ? (c) : (a)) :
                   ((c > b) ? (c) : (b));
```